Jürgen Dobaj, BSc

Design and Evaluation of a Lock-Free Concurrent Software-System

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing.,Dr.techn. Christian Josef Kreiner

Institute for Technical Informatics

Graz, January 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

The purpose of this Master's thesis is the development and evaluation of a highly concurrent lock-free task-parallel and data flow oriented measurement and control software for the automation of automotive test benches. To efficiently transfer the massive amount of data that is generated during vehicle-testing an intrusive bounded lock-free causal FIFO queue algorithm was developed. The algorithm is especially designed to reduce the heap, the cache and the main memory contention. Thus the queue is particularly suitable for the deployment in data flow oriented software applications. Additionally the queue facilitates intra-process as well as inter-process communication, even between 32-bit and 64-bit processes, by allowing data throughput ratios close to the machine's maximum main memory bandwidth.

The developed software design scales well with the progressive increase of the core count in future CPUs and the applied implementation techniques best exploit the features of modern processor architectures. Indeed, the design is applicable for a wide range of measurement and automation tasks and it is not limited to automotive testing purposes only.

Kurzfassung

Das Ziel dieser Masterarbeit ist die Entwicklung und Evaluierung einer hochgradig nebenläufigen, blockierungsfreien, aufgabenparallelen und datenflussorientierten Mess- und Regelungs-Software für die Automation von Automotive-Prüfständen. Um die enorme Menge an Daten, welche während eines Testlaufs generiert werden, effizient zu verarbeiten wurde ein "intrusive bounded lock-free causal FIFO" Queue-Algorithmus implementiert. Der Algorithmus wurde entwickelt um speziell die Beanspruchung von Heap, Cache und Hauptspeicher zu reduzieren. Aus diesem Grund ist die Queue besonders für den Einsatz in datenflussorientierten Anwendungen geeignet. Die entwickelte Queue kann nicht nur zur prozessinternen Kommunikation verwendet werden, sondern auch für die Kommunikation zwischen mehreren Prozessen. Die Queue erlaubt außerdem die Kommunikation zwischen 32-Bit und 64-Bit Prozessen und der erreichte Datendurchsatz ist nahe an der Grenze der auf dem Rechner maximal verfügbaren Hauptspeicherbandbreite.

Das entwickelte Softwaredesign skaliert hervorragend mit dem fortschreitenden Anstieg der Prozessorkernanzahl in zukünftigen Prozessoren. Weiter ermöglichen die eingesetzten Implementierungstechniken die effiziente Nutzung der Funktionen und Ressourcen moderner Prozessorarchitekturen. Das Softwaredesign kann für verschiedenste Mess- und Automatisierungsaufgaben verwendet werden und ist nicht auf den Automotive-Testing Bereich beschränkt.

Acknowledgments

Firstly, I would like to express my thanks to Christian Kreiner for his supervision and guidance throughout the writing of this thesis.

I am grateful to Kristl, Seibt & Co Gesellschaft m.b.H for the sponsorship of this project. In particular I would like to thank Günter Schröfl and Norbert Buch for always having useful feedback on hand and for their expertise.

Finally, I would like to express my gratitude to my parents and to my girlfriend. This thesis would not have been possible without their support and encouragement throughout my years of study and during the process of writing this thesis.

Jürgen Dobaj

Graz, January 22, 2017.

Contents

List of abbreviations

ALU arithmetic logic unit

BCE base core equivalent

CAS compare-and-swap

CPU central processing unit

DB database

DUT device under test

FIFO first in - first out

GPU graphics processing unit

HDD hard disk drive

IO input/output

ISA instruction set architecture

MMU memory management unit

MPSC multiple-producer single-consumer

MPMC multiple-producer multiple-consumer

NUMA non-uniform memory access

RAM random access memory

RMW read-modifiy-write

RTE runtime environment

RTS real time system

SIMD single instruction, multiple data

SMT simultaneous multi-threading

SPSC single-producer single-consumer

SPMC single-producer multiple-consumer

SPU signal processing unit

SoC system on chip

STM software transactional memory

List of Figures

List of listings

1 Introduction

The rapid moving components and the massive data traffic in modern vehicles provide plenty of interesting and challenging problems to the field of automotive testing. One of these challenges is the accurate measurement of vehicle states during a test run and the test automation itself. In order to be well-equipped for testing forthcoming vehicle generations with even faster moving mechanical parts and increasingly high data traffic, the purpose of this Master's Thesis is the design and the evaluation of a highly concurrent software system for automotive testing and test automation. The software system should best exploit the hardware parallelism of modern CPUs, it should be flexible in test automation and easy to maintain. Figure 1.1 schematically illustrates a typical automotive test bench environment and outlines the scope of this work.

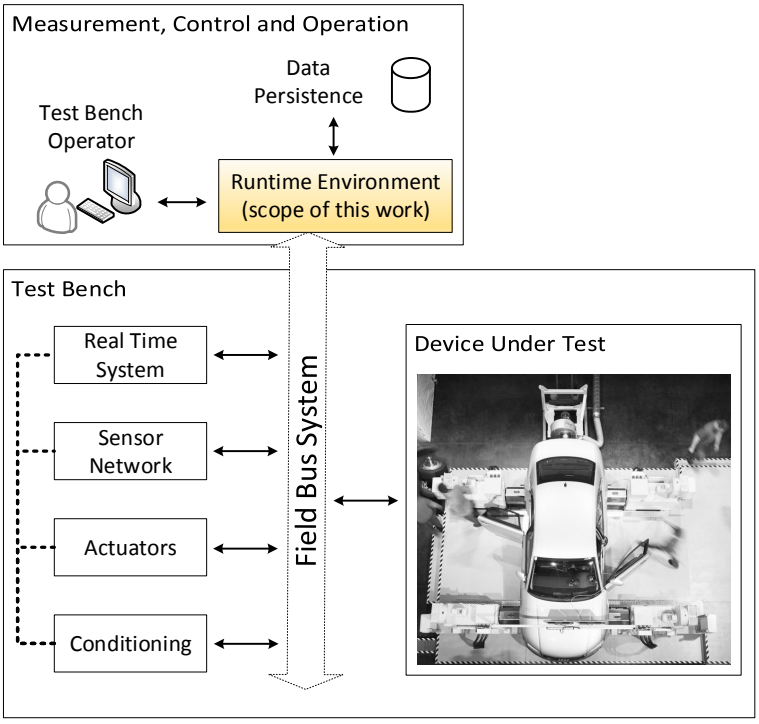

Figure 1.1: **Automotive test bench setup.** A typical automotive test bench setup consists of a sensor network, actuators, the device under test and a supply infrastructure, which provides conditioned oil, gas, water, air and electricity for all other test bench components. The components are interconnected via a field bus system an the *Runtime Environment* is responsible for monitoring, controlling, measuring and visualizing the system states during the test execution.

This work is conducted in cooperation with the company *KS Engineers* [1], which delivers solutions in the fields of automotive engineering, industrial automation and building facilities. For automotive industry *KS Engineers* designs and builds test benches for research and development purposes as well as for end of line testing. These test benches are automated with the company-developed automation software, Tornado (*Tornado System* 2016), which provides measurement, control and report functions. The development of the Tornado System started in the early *90*'s and over time it evolved to the high performance measurement and automation system, which today is used by leading manufacturers and suppliers for testing their products (*References* 2016). The Tornado System can be configured for various testing purposes including the testing of transmissions, gearboxes, power trains, combustion and electric engines (e.g. car, truck, ship, racing), battery testing/simulation, testing of whole vehicles and more. The data accumulated during the test execution is managed by the automation software. It is also possible to integrate third-party tools for this purpose. In order to be geared up for testing the forthcoming vehicle technologies with its increasing data traffic, the next generation measurement and automation software system should be designed and evaluated.

Prospect

The summary of the related work in chapter 2 starts with an overview of the hardware and software aspects in the parallel programming domain, followed by the explanation of the fundamental laws of parallelism. Chapter 2 closes with the description of the well established parallel programming concepts and the fundamentals of lock-free programming.

Chapter 3 explains the automotive testing problem domain and the developed software design. In order to develop a suitable software design we decided to use the systems engineering approach (Haberfellner, 2015). We started to precisely analyzed the problem domain and its environment. For this purpose we elaborated several different aspects of the domain by creating

a) a data oriented view (see data flow diagram in figure 3.3),
b) a component oriented view (see domain model in figure 3.1) and
c) a software developer's view (see architectural stack in figure 3.2)

. Each view revealed new constraints and/or supplemented the other views. This made the identification of the main issues easier and allowed us to address them in the subsequent design and development phases. Creating the views, the software design and the implementation was an iterative process. Whenever something was unclear or underspecified the corresponding components were refined.

The problem origin is in the field of automotive testing and automotive test automation. However, to finally obtain a software design that is applicable for a wide range of measurement and automation systems, we generalized the functionality and data flow.

[1] www.ksengineers.at

During the development we focused on the common problems: The increase of the application's parallel portion and the efficient inter-thread communication and data buffering.

It turned out that the developed software design is best implemented using first in - first out (FIFO) queues for the data and task synchronization. Thus chapter 4 introduces a universal fine-grained queue interface and an intrusive bounded lock-free causal FIFO queue algorithm. Both are especially designed to best support the data flow oriented software design described in chapter 3. The developed queue is an universally applicable synchronization mechanism that is suitable for intra-process and inter-process communication, even between 32-bit and 64-bit processes.

The queue was especially designed to reduce the cache and main memory contention, which is confirmed by the measurement results in chapter 6. The results show that the queue algorithm allows data throughput ratios close to the test platform's maximum main memory bandwidth. Finally, chapter 7 discusses the software design decisions and the results of the performance tests.

2 Related work

2.1 The free lunch is over. Yet, in the jungle, we are fighting for the next free lunch!

In 2005 Herb Sutter described, in "The Free Lunch Is Over" (Sutter, 2005), that the times for software developers were going to become harder. Since the launch of microprocessors in the early 1970s the number of transistors doubled roughly every two years following Moore's law (Moore, 1965). With the transistor's doubling also the available processing power doubled. Every new hardware release directly boosted the performance of single-threaded applications. Encouraged by Amdahl's law (Amdahl, 1967), which stated that the maximum theoretical speedup through parallelization is limited by the sequential portion of an application, the hardware vendors focused on making single-threaded applications faster by increasing the clock speed, the cycle performance and the cache sizes. Around 2004 the processors achieved a single core performance of up to $\sim 4\ GHz$. From a technical and economic point of view it was no longer affordable to further increase the single core speed, because of power consumption and power dissipation.

Still, the exponential increase of transistors continued and the hardware industry started to build multi core processor. It turned out that multiple cores running at a lower frequency were able to execute more instructions per chip per cycle at the same power budget than a single core processor at higher frequency. However, the hardware evolutions no longer brought significant performance gain to single-threaded applications and the software industry saw their products performance to stagnate or to become even slower. The free lunch was over (Sutter, 2005) and the software developers were faced with the non trivial problem to transform single-threaded programs into parallel executable multi-threaded applications.

The introduction of multicore processors was just the starting point for building massively parallel mainstream computer systems. With the integration of central processing units (CPUs) and compute-capable graphics processing units (GPUs) onto the same die the time of heterogeneous multicore processors began. The lowest level of heterogeneity is achieved by **a)** the integration of multiple cores with the same instruction set architecture (ISA). In this case some cores are more complex (more internal concurrency) to boost the performance of sequential program fractions, while the other smaller cores increase the throughput of the scalable parallel program fractions. The smaller cores are also less power hungry, which improves the performance per watt. The integration of cores with **b)** different ISAs increases the level of heterogeneity. In this case one or more general

purpose cores share a die with special purpose processing units, such as signal processing units (SPUs) and GPUs, which can run certain kinds of code faster and more power efficient. Since 2009, GPUs are commonly used to execute mainstream code, instead of doing graphics calculations only.

With the integration of hundreds and thousands of (heterogeneous) cores onto one die we started to reach physical limits. Due to the power consumption and the thermal heat dissipation it is no longer possible to power all transistors at the same time, which is denoted as the dark silicon problem. Of course, we could (indeed we do) continue to integrate more transistors onto one die, but if we are not able to use the additional processing capabilities we still have reached the end of multicore scaling (Esmaeilzadeh et al., 2011).

The field of multicore processors is a hot research area where scientists and the industry are trying to push the limits. New memory technologies such as 3D stacking, optical interconnects and memristors are explored to make the devices smaller, more energy efficient and to optimize the memory latency and memory throughput. Novel scheduling algorithms try to address the memory bottlenecks and power consumption by optimizing the task assignment to cores. Power optimization techniques such as near threshold operation, dynamic voltage/frequency scaling and clock/power gating make their way into mainstream processors. (Henkel et al., 2015) (Wang, Kenli Li, and Keqin Li, 2016) (Vajda, 2011)

To overcome the end of multicore scaling we must invent groundbreaking technologies or other possibilities to boost the performance of our processing intensive and (hopefully) highly parallel applications. This is, were cloud computing comes in. At roughly the same time, when the industry started to build heterogeneous processors, cloud computing made its way to the mainstream via Amazon Web Services, Microsoft Azure, Google App Engine, Verizon and many others. From a developers point of view, the cloud offers hardware (or infrastructure) as a service with apparently endless scalability, while the requirements on an application running in the cloud are even tighter compared to those applications running on multicore processors only. Next to exploiting the features and resources of local heterogeneous multicore processors, the cloud application must also be able to efficiently utilize distributed cores and distributed memory systems. The configuration details of the cloud should be on the one hand transparent to the application and the code, but on the other hand the advanced user may want to govern the data traffic and/or may want to assign the cores manually for performance optimizations.

The transparent utilization of local and distributed heterogeneous hardware as well as the programming language support and the code portability depict a fascinating research area for the operating system vendors, the compiler community and the software developers. In order to successfully shift towards highly parallel applications that run on heterogeneous and/or distributed system, we have to efficiently exploit the ”hardware jungle” (Sutter, 2012b). Yet, in the jungle, we are fighting for the next free lunch.

2.2 Laws of parallelism

Amdahl's law

Amdahl's law (Amdahl, 1967) states that the maximum theoretically achievable speedup through parallelization is limited by the sequential portion of an application, no matter how much cores are available for the computation. Amdahl published his findings in 1967 to encourage the improvement of single-threaded hardware performance for large scale computing. Mathematically the law is expressed by

$$Speedup = \frac{1}{1 - P + \frac{P}{N}} \tag{2.1}$$

where P denotes the parallel portion of the application and N is the number of available hardware cores. For an infinite number of cores the equation simplifies to $Speedup = 1/(1-P)$. Hence, every application with a sequential portion has an upper speedup limit independent of the core count.

Gustafson's Law

Gustafson's law (Gustafson, 1988) was formulated in 1988 and was contradictory to Amdahl's law. Gustafson experimentally showed that practical applications with a sequential portion between 40% *to* 80% scale well with the increasing number of cores. He obtained speedup factors larger than 1000 by using 1024 processing cores, while according to Amdahl's law a program with a sequential fraction of 80% could run roughly five times faster at maximum. Gustafson concluded that the parallel portion of the evaluated programs scales with the increase of the problem size and that the sequential parts (input/output (IO), initialization and startup, sequential bottlenecks,...) remain constant in the problem size. Mathematically this observation is expressed as

$$Speedup = N - (N - 1) * np \tag{2.2}$$

where np is the sequential portion of the program and N is the number of available cores. The law suggests that independent of the sequential program fraction an infinite speedup can be obtained. This is, of course, contradictory to Amdahl's law.

The equivalence of Amdahl's law and Gustafson's law

Amdahl's law and Gustafson's law seemed to be adversary until, in the year 1996, Shi proved the equivalence of the two laws and showed that they are both expressing the same inherent law of parallelism (Shi, 1996). Shi mathematically showed that the parallel program fraction in Amdahl's law is not equivalent to the parallel fraction in Gustafson's law, which was continuously misinterpreted. He showed that there is a mathematical relation which transforms the sequential portion from Gustafson's law to Amdahl's law.

This proved that the two equations are equivalent and that both are expressing the same law, but in different formulations. Amdahl predicts that the maximum speedup is limited by the sequential program fraction, while Gustafson pretends endless scalability. Shi concluded that the speedup factor depends on the application's properties and that it is not possible to reliably determine the sequential fraction of a particular program by measuring. The basic fact of both laws remains: we have to tackle the sequential program portions to achieve a close to linear speedup with the increasing number of cores. (Shi, 1996) (Vajda, 2011)

Amdahl's law for multicore processors

In 2008 Hill and Marty published a theoretical survey about the applicability of Amdahl's law to multicore processor architectures (Hill and Marty, 2008). To investigate the impact of the core count and the architectural layout they defined an abstract mathematical model with the base core equivalent (BCE) as the smallest available processing unit. A multicore processor model contained a total number of n BCEs and multiple BCEs could be clustered to a more powerful core with the approximated processing performance $perf(r) \approx \sqrt{r}$, where r depicted the number of BCEs in the cluster. The model did not include the interconnection hardware, the memory system and dynamic power effects. Hill and Marty analyzed three processor architectures, the results are illustrated in figure 2.1:

- A **symmetric multicore chip** consisting of homogeneous cores
- An **asymmetric multicore chip** built of heterogeneous cores with the same ISA
- An hypothetical **dynamic multicore chip** that is able to dynamically harness all cores for either sequential or parallel program execution

The mathematical model of the **symmetric multicore chip** was defined as

$$Speedup_{symmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)*\frac{n}{r}}} \tag{2.3}$$

where f was the parallel program fraction. In this model r clustered BCEs with the performance $perf(r)$ executed the sequential program portion $1-f$, while $\frac{n}{r}$ BCEs ran the scalable parallel fraction f with the performance $perf(r)$ (all cores run with $perf(r)$ in the symmetric case). The speedup of the symmetric model is limited by the sequential program fraction $1-f$, as predicted by Amdahl.

In the **asymmetric multicore** scenario $1+n-r$ BCEs with performance 1 were available for the execution of the parallel fraction. The single more powerful core with performance $perf(r)$ could execute both, the sequential and the parallel program parts. For this case the speedup can be expressed as

$$Speedup_{asymmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}} \tag{2.4}$$

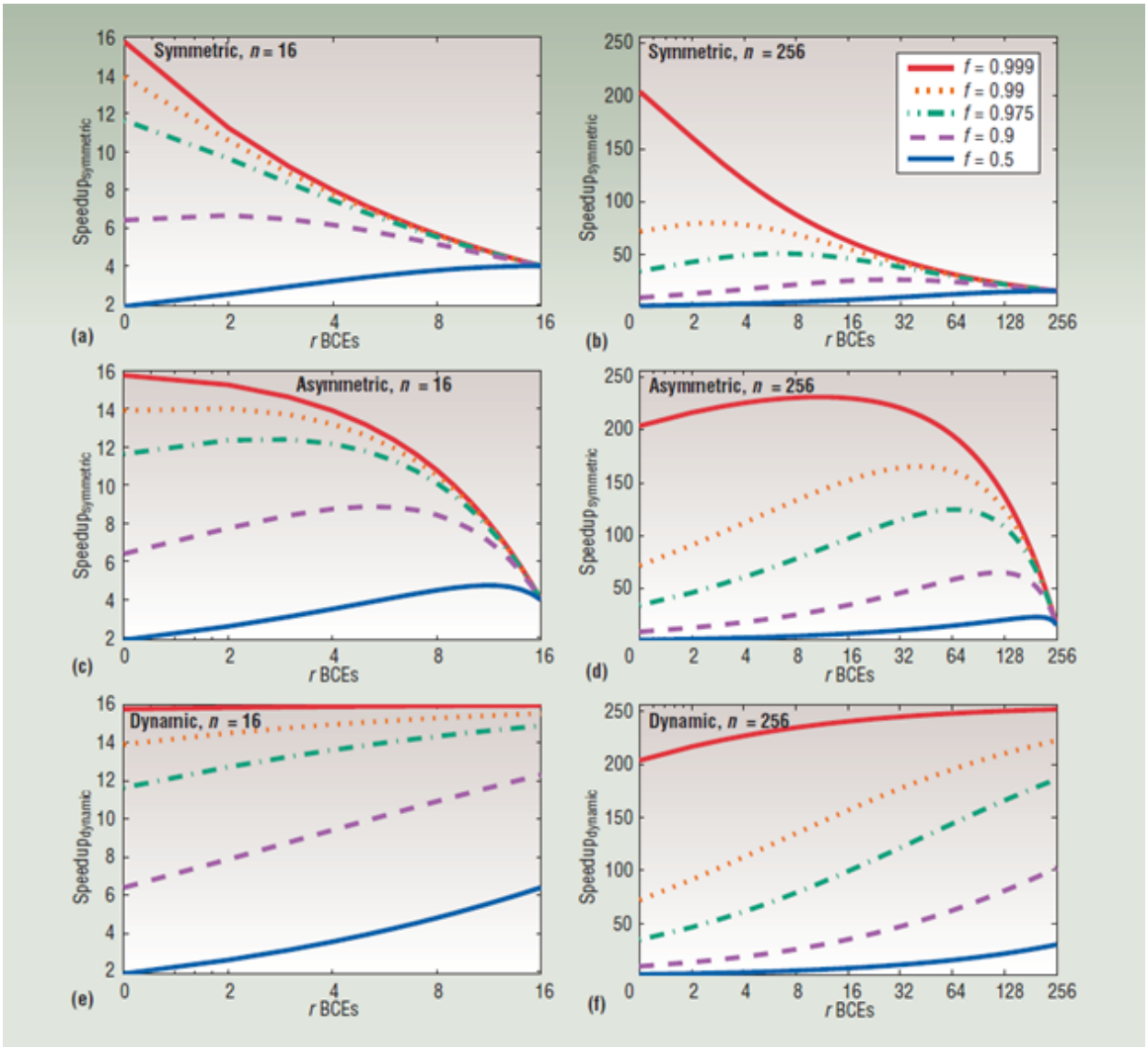

Figure 2.1: **Speedup according to Amdahl's law on multicore processor**. Speedup of (a, b) symmetric, (c, d) asymmetric, and (e, f) dynamic multicore chips with n = 16 BCEs (a, c, and e) or n = 256 BCEs (b, d, and f). In the equations and graphs continuous approximation was used instead of rounding down to an integer number of cores. Image source: "Amdahl's Law in the Multicore Era" (Hill and Marty, 2008).

The graphs (**c**)(**d**) in figure 2.1 reveal that in the asymmetric multicore scenario there exists a sweet spot for each application type, where the speedup reaches a maximum. Additionally the speedup obtained by an asymmetric chip is always larger or equal to the maximum speedup in the symmetric scenario. The model shows that asymmetric architectures can relax the impact of Amdahl's law, but different applications may have different sweet spots. Thus a static asymmetric multicore design is not sufficient to satisfy all kinds of applications.

Hence, Hill and Marty analyzed the speedup of a hypothetical **dynamic multicore chip**. This chip could use the performance $perf(r)$ of r BCEs in its sequential operating mode and the performance of all n BCEs in its parallel mode for the code execution. The mathematical model of such a chip is defined as

$$Speedup_{dynamic}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}} \tag{2.5}$$

The graphs (**e**)(**f**) in figure 2.1 show that in the dynamic scenario all application types would scale well with the increasing number of cores and that the obtained speedup results are always greater or equal to the asymmetric multicore speedups. Such a dynamic architecture would not be limited by Amdahl's law. However, (today) we do not know how to build such dynamic multiprocessor architectures, but techniques such as frequency/voltage scaling, thread-level speculation or helper threads could imitate such a dynamic system behavior (Vajda, 2011).

Hill and Marty wanted to encourage the community to continue with the research in boosting both the parallel as well as the sequential core performance. Especially with the idea of a dynamic multicore chip, they inspired to think outside the box.

2.3 Parallel programming

Due to the stagnating technology scaling in 2004 (see section 2.1) it became necessary to build multi-threaded applications to profit from the improvements in multicore architectures. In this section we give an overview of the fundamental parallel programming concepts, the available synchronization primitives and the hardware components and features of common multicore architectures that are crucial for the implementation of parallel applications.

2.3.1 Problem decomposition

When building parallel applications we want to execute as many instructions in parallel as possible and try to minimize the sequential fraction in our applications. Thus, when (re)designing an application we start with the identification of the exploitable concurrency in our problem domain (top-down approach). Once we have identified the parallel fractions we map them into our program domain by creating for example architecture diagrams, data flow diagrams and class diagrams, which can be used for the subsequent system implementation (bottom-up approach). The diagram in figure 2.2 illustrates the *finding concurrency* design space and highlights its place in the pattern language presented in (Mattson, Sanders, and Massingill, 2004).

To solve a problem in parallel, we first have to decompose it. The two fundamental decomposition techniques are *task (or functional) decomposition* and *data decomposition*. The

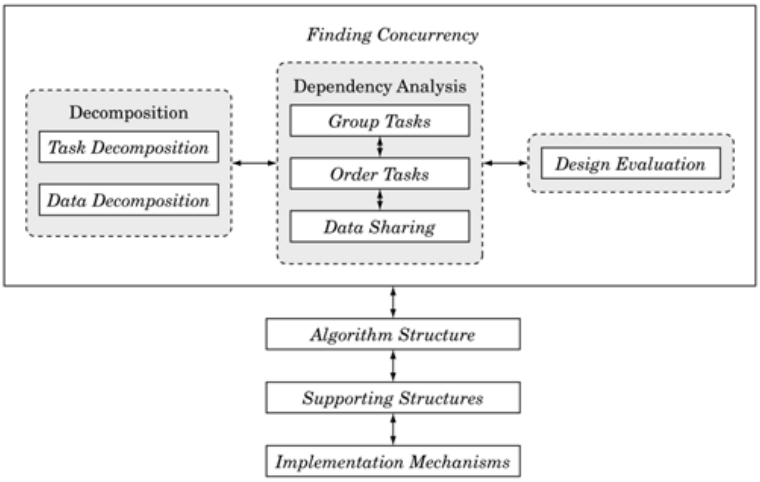

Figure 2.2: **Overview of the finding concurrency design space in the pattern language for parallel programming (Mattson, Sanders, and Massingill, 2004)**. Images source: "Patterns for Parallel Programming" (Mattson, Sanders, and Massingill, 2004).

target of *task decomposition* is the division of tasks or functions into smaller partitions that run in parallel (with more or less dependencies). Functional decomposition can be separated into static and dynamic decomposition techniques and is directly supported by the operating system and the hardware via thread-level parallelism.

If the partitioning of tasks or functions is known at the design time, then the functions can be statically decomposed and the system design targets to support the parallelization of that functions. The problem with static decomposition is that it does not scale, if the core count increases. Dynamic task decomposition, of course, scales well if new resources are added. A simple dynamic decomposition technique is to launch a new thread for each new task/function that should be executed. However, on processors with only a view cores or massive task parallelism this might lead to performance issues for the whole system because of oversubscription (Iancu et al., 2010). In such cases it would be better to combine static and dynamic decomposition techniques by using a thread pool that implements a work stealing algorithm (or a similar userspace technique). The thread pool would only launch a maximum number of tasks, which prevents oversubscription and additionally eliminates the thread creation and destruction overhead.

Data decomposition aims to execute the same function on multiple data sets in parallel. This is generally required for graphics calculations, vector/matrix operations and applications were a function should be applied on all elements of an array. If the elements can be processed independently, then all functions can be calculated simultaneously on different cores. Data parallelism with smaller data sets is natively supported by SIMD instructions on mainstream CPUs, while larger data sets can be more efficiently processed on GPUs.

The *pipeline pattern* is an example where data decomposition and static function decomposition are combined. A pipeline is structured into stages and in general a stream of data elements is shifted through the pipeline stages. At each stage a function is applied to the data element at that stage. The functions on each stage may be different and can be

processed in parallel on multiple cores. In software a stage is usually implemented as a thread and buffers are used to synchronize the stages. Thus the *pipeline pattern* (Mattson, Sanders, and Massingill, 2004) is closely related to the *pipes and filters pattern* (Buschmann, Henney, and D. C. Schmidt, 2007).

This section provided an overview of the basic decomposition techniques and also described some patterns and use cases. A comprehensive pattern language for parallel programming can be found in the "Patterns for Parallel Programming" (Mattson, Sanders, and Massingill, 2004), the POSA series (especially (Buschmann, Henney, and D. C. Schmidt, 2007) and (D. Schmidt et al., 2000)) and, of course, the web.

2.3.2 Synchronization primitives

So far we have discussed how problems are decomposed into simpler/smaller chunks that can be executed in parallel on the available resources. However, to generate the final result we have to synchronize the execution of these chunks in some way and the tasks must also be assigned to the processor resources. This is what modern operating systems were made for. A modern multicore operating system enables software applications to efficiently utilize hardware resources and additionally provides several layers of abstraction, which easies the software development and data synchronization. From a software developers perspective the most important operating system features are the thread scheduling, the balanced thread assignment to the available hardware cores as well as the physical and virtual memory management. Covering all those topics would go far beyond this work. An extensive survey about modern operating systems is provided by the equally named book from Andrew S. Tanenbaum (Tanenbaum, 2001). Here, we only want to discuss the inter-thread synchronization mechanisms, which are vital for programming cache-coherent shared-memory multicore processors.

Blocking synchronization

The traditional method to synchronize the access on shared resources in multi-threaded applications is the utilization of *blocking primitives*, which include locks, critical sections, semaphores, condition variables or more advanced mechanisms such as monitors, futures and promises. However, these primitives have several drawbacks. Whenever a lock is exclusively owned by one thread, then all other threads that attempt to acquire the lock are blocked and have to wait until the owner releases the lock. This property might lead to subtle error conditions like deadlocks, livelocks or priority inversion. If the system is properly designed then these failures can be avoided. Nevertheless, even in correctly implemented and well designed systems blocking synchronization primitives limit the system performance on multicore architectures by increasing the sequential portion of the application due to the following reason:

[a] Lock-overhead: Acquiring and releasing locks are generally expensive operations that have to be frequently executed. Additionally, locks require memory resources that have to be allocated, initialized and destructed.

[b] Lock-contention: If the contention on a specific lock is high, then the task switching overhead due to thread suspension influences the overall system performance. Moreover, suspended threads can not do any work at all and the highly contented lock may serialize the execution of several threads, which might drastically increase the sequential program portion.

[c] Locks are typically used to build critical code regions that linearize the execution of code lines or whole methods, instead of just synchronizing the access to memory regions.

Let us consider an unbounded FIFO queue to map the above listed items onto a common problem. The queue q should support the following two methods:

- *q.enq(x)* enqueues the value x. The method will always succeed, because the queue is unbounded and we assume that the system never runs out of memory.
- *q.deq(y)* dequeues the value y. The method only succeeds if the queue is non-empty.

To synchronize the access to the queue we can either use a *coarse grained* or a *fine grained* locking approach. If we decide to use the *coarse grained* approach a single lock is responsible for the entire synchronization. This drastically increases the sequential portion of the application by serializing all method calls (item [b]). On the other hand there is only one lock, which reduces the number of acquire/release operations (item [a]).

In the *fine grained* locking scenario we use multiple locks to lower the contention between enqueue and dequeue operations. Compared to the *coarse grained* case, this scenario decreases the sequential program fraction (item [b]), but increases the number of acquire/release calls (item [a]). Implementing fine grained locking algorithms is generally more complex. These two examples show that a trade-off between lock-overhead and lock-contention is required.

However, if locks are used then item [c] can only be addressed by optimizing the method implementations to require as little cycles and memory operations as possible, which reduces the runtime of a single method call. Thus a thread can execute the method faster, which lowers the probability that a thread owning the lock is preempted by the scheduler, which consequently reduces the probability that another thread is blocked due to a pending method call.

Non-blocking synchronization

Non-blocking synchronization primitives do not suffer from the problems explained in the previous section. As the name already implies, a non-blocking synchronization mechanism does not block the execution of other threads that compete for the same resource. Thus, even if a thread is preempted, it is guarantied that the rest of the system can make progress. This is established by using low level primitives, so called atomic

operations, which serialize the access to a memory region, instead of serializing the execution of multiple code lines. Thus non-blocking algorithms can significantly increase the parallel fraction of applications, which facilitates the efficient resource utilization on multicore chips and hence yields performance improvements and enhanced scalability.

However, the implementation of correct non-blocking data structures and algorithms is by far more complex and error-prone, which justifies the existence of blocking primitives. All blocking primitives rely on non-blocking primitives as there basic building blocks. Atomic instructions do not only synchronize the access to a single memory location, but they can also be used to enforce the synchronization of large data sets based on the memory ordering policies provided by the processor architecture. Atomic instructions can further be used to enforce a weaker memory ordering, which may reduce the contention on the caches and the interconnection infrastructure.

2.4 Lock-free programming

2.4.1 Hardware basics

The block-diagram in figure 2.3 schematically illustrates the components of a modern multicore processor. The processor has four cores with two hardware threads per core, an out of order execution engine and several load and store buffers to hide the latency of memory operations. The cores communicate over the shared L3 cache and the system bus connects the cores and other SoCs integrated on the die.

The four cores support the same ISA and hence it does not matter on which core a program or the operating system is executed. However, if some programs are able to exploit data parallelism, then these applications may use program libraries or compilers, which automatically harness the GPU to execute certain operations and code sequences massively parallel.

Today's general purpose cores use inherent hardware parallelism to speedup single threaded programs by implementing super-scalar pipelining, instruction reordering, out-of-order execution, branch prediction and simultaneous multi-threading (SMT). SMT addresses the problem of limited instruction level parallelism in legacy applications by enabling the simultaneous execution of two or more instructions streams per core. This is achieved by the integration of one or more hardware threads, which share the ALUs, the out-of-order execution unit, the L2 cache and several other resources on the same core.

To outline the expended effort of the hardware vendors to provide massive hardware parallelism, we take a look at the Intel Itanium II processor from the year 2002, which is illustrated in figure 2.4. The Itanium II processor consisted of 211 million transistors, but only one percent of all transistors on the die were used to manipulate the data. The remaining 99% were responsible for latency hiding by efficiently moving the data through the processor.

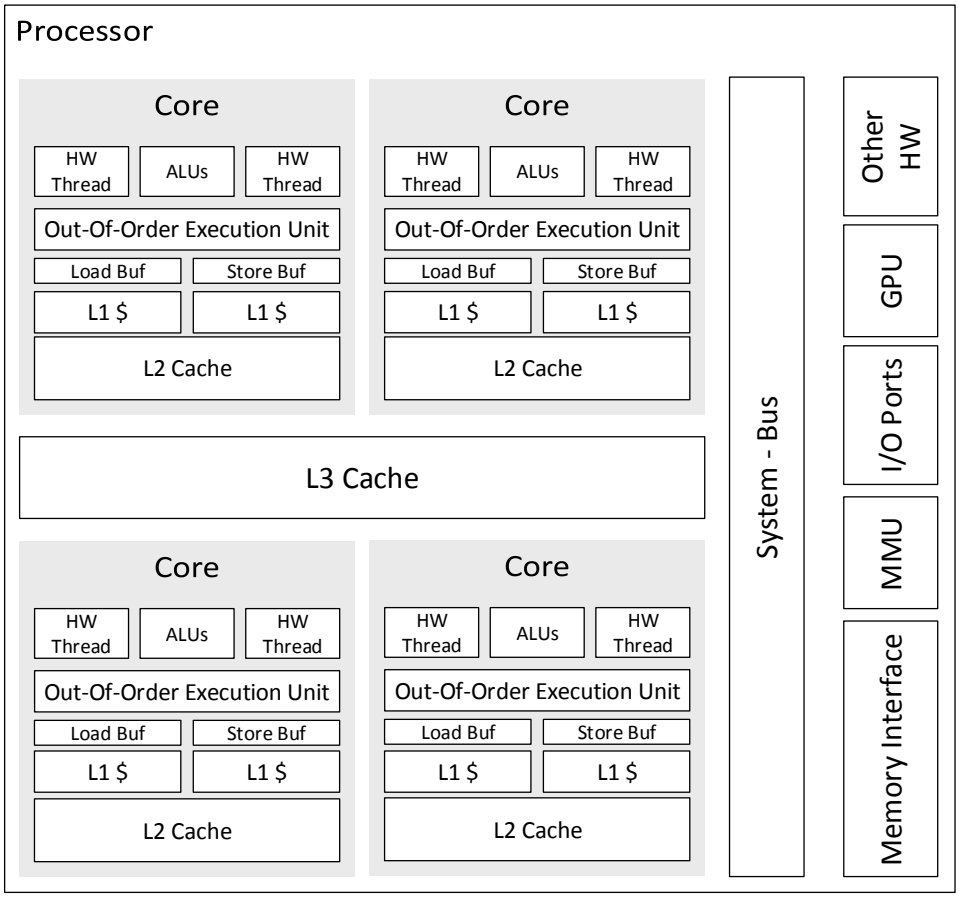

Figure 2.3: **Block-diagram of a modern multicore processor**. The block-diagram illustrates an example of a modern multicore processor. The processor has four cores with two hardware threads per core, an out of order execution engine and several load and store buffers to hide the latency of memory operations. Each hardware thread has a private L1 cache. The ALUs, the L2 cache and several other resources are shared between the two hardware threads on a single core. The last level cache is shared between all cores and the processor maintains full cache coherency.
The system bus connects the cores to the peripherals and to the other SoCs on the die. As the diagram depicts the last level cache and the system bus are highly contended resources and hence they are common bottlenecks in todays data hungry applications. In most modern processors the MMU is integrated to allow the parallel access to the main memory and I/O devices. The integrated GPUs are not as powerful as their parents located on the graphic cards, but cheaper and ready to be harnessed for data parallel computations.

The processor in figure 2.3 implements a cache-coherent non-uniform memory access (NUMA) memory architecture. Non-uniform memory access means that the access time to data depends on the relative location of that data to the core. Data that is located in the L1 cache can be for example faster accessed than data located in the L2 cache or the main-memory. Cache-coherent means that the data coherency is maintained via a cache coherence protocol and that all cores access the same main memory. Examples of other memory architectures are **a)** unified memory access architectures, **b)** NUMA RAM architectures and **c)** incoherent disjoint memory architectures (Sutter, 2012b).

A common cache coherency protocol is the MESI protocol: MESI is the acronym for the

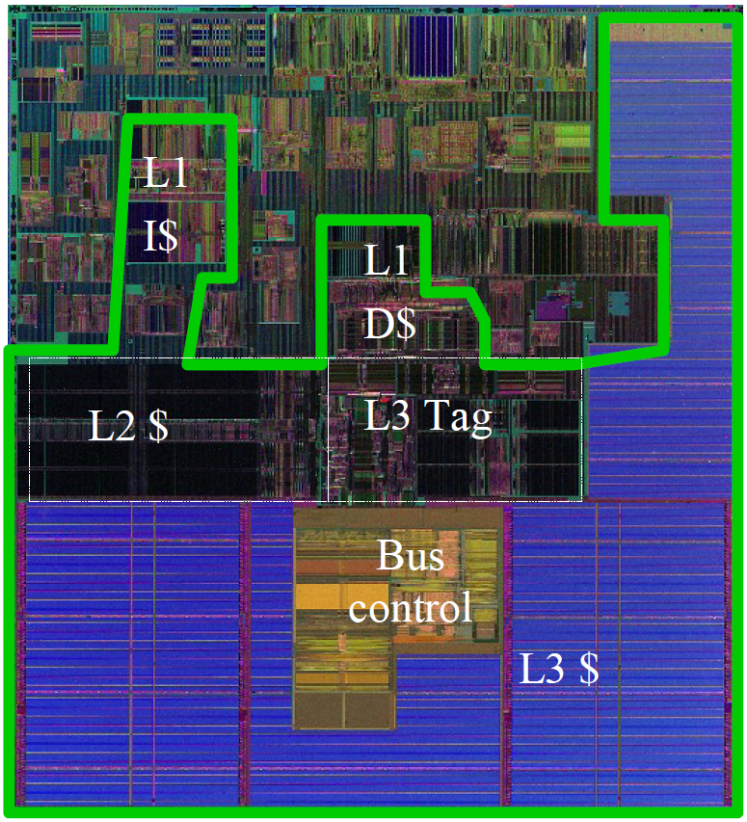

Figure 2.4: **Intel Itanium II processor**: The figure illustrates an Intel Itanium II processor that consists of 211 million transistor. ~ 85% of the transistors are used for implementing the caches. 99% of all transistors on the chips are responsible for data moving and latency hiding, while only 1% of the implemented transistors on the die do data transformations. Images source: (Patterson, 2004)

cacheline state names in this protocol: Modified-Exclusive-Shared-Invalid. The states and the corresponding processor interactions of the MESI protocol on and Intel 64 and IA-32 architecture are illustrated in figure 2.5. A cache is structured into chucks of a certain size, so called cachelines. A typical cachline size is 64 byte, which depicts the smallest entity that can be written or loaded from the main memory. The cache coherency mechanism is transparent to the software.

The cache coherency protocol alone is not sufficient to maintain a consistent view of the main memory content. An additional policy is required, which defines the ordering of how cores and processors are allowed to issue store and load operations to a specific memory location. This policy is referred to as the *memory model*, the *memory ordering* or the *memory-ordering model*. However, operations can only be ordered if they are indivisible. Thus the hardware supports *atomic operations* that establish the basis of the memory model. The atomicity of operations is achieved by the following three interdependent mechanisms (*Intel 64 and IA-32 Architectures Software Developer's Manual* 2016):

- **Guarantied atomic operations**: Atomicity of load and store operations can be guarantied if the affected data is correctly aligned and does not exceed a certain size.

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	Out of date	Valid	Valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	Does not go to the system bus.	Does not go to the system bus.	Causes the processor to gain exclusive ownership of the line.	Goes directly to the system bus.

Figure 2.5: **MESI cache line states**: The table illustrated the cache coherency protocl that is defined for the current Intel 64 and IA-32 architecture. The cache is structured into equally sized chunks, which are called cachelines. A cachline is the smallest entity that can be written or loaded from the main memory. The cores, however, are able to address single bytes and bits in one cacheline. Images source: (*Intel 64 and IA-32 Architectures Software Developer's Manual* 2016)

On Intel architectures the following operations can be always executed atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

While on newer Intel architectures the memory model was extended to also support guarantied atomicity for:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus
- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

- **Bus locking**: If the LOCK signal is issued then all requests to the system bus (or an equivalent link) of other bus participants are blocked. The processor automatically asserts the LOCK signal during memory critical operations. The LOCK signal can also be manually asserted for special instructions (usually read-modify-write) to synchronize the access to shared memory regions.
- **Cache locking**: The cache coherency protocol ensures that atomic operations can be carried out on cached data structures.

The strictest memory consistency model that is available in todays multiprocessors is called *sequentially consistent*. A memory model or a system is *sequentially consistent* if

> *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* (Lamport, 1979)

In other words, this memory model provides a sequential view of a multi-threaded program as if all issued instructions where executed in some particular sequence by a single thread. A sequential view makes it easy for software developers to reason about a particular program outcome, which makes the sequentially consistent model popular. However, a lot of synchronization between the cores and processors is required to maintain a *sequentially consistent* memory model. Thus modern CPUs implement

weaker models per default, which allows performance enhancing operations such as instruction reordering or out-of-order execution. Figure 2.6 illustrates a possible outcome of a write ordering on a multicore system that implements a weak memory ordering model.

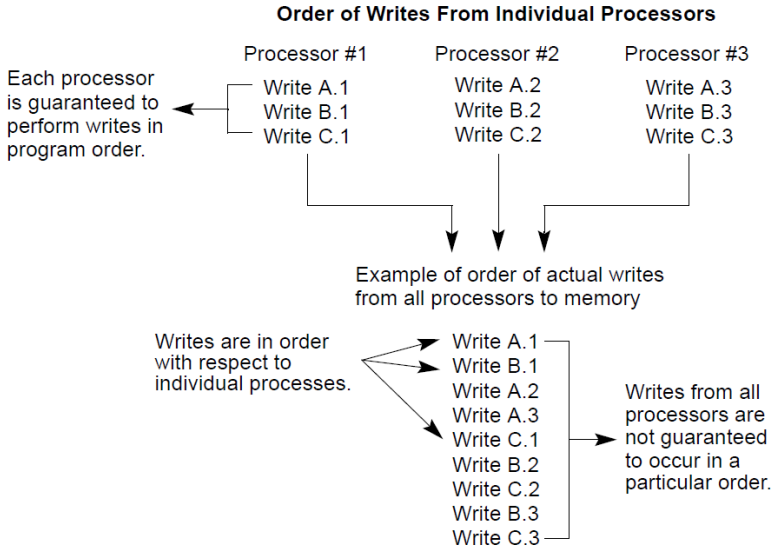

Figure 2.6: **Example of write ordering in multiple-processor systems**. Images source: (*Intel 64 and IA-32 Architectures Software Developer's Manual* 2016)

The following scenario describes a possible reordering of write and read operations to different memory locations within the same program: A write operation is cached in the core's store buffer. Before this write operation is issued, the core could already have finished a read operation that occurs later in the program order, which effectively reorders a read operation with an older write operation to a different memory location. A sequential consistent memory ordering policy would not allow such a reordering.

2.4.2 Software basics

The default memory ordering of the processors is weaker than a sequentially consistent ordering, which is usually not sufficient to safely synchronize the access to shared resources in multi-threaded applications on a multicore processor. Thus the hardware provides mechanisms to influence (weaken or strengthen) the memory ordering. This mechanisms include atomic operations and special instructions, so called memory fences (e.g. *SFENCE*, *LFENCE*, *MFENCE*), that enforce a specific memory ordering.

The commonly supported atomic operations are atomic loads, atomic stores and read-modifiy-write (RMW) operations. Atomic loads and stores where already discussed in section 2.4.1. From a software developers point of view atomic loads/stores can be used for two purposes. On the one hand they allow atomic reads/writes to a specific memory location and on the other hand they can be used like memory fence instructions to enforce

a memory ordering onto other memory operations. (For details regarding memory fences and instruction ordering with atomics please consider the software developer's manual of your hardware vendor and/or your programming language specification.)

Atomic loads and stores, however, are not sufficient to synchronize the access to resources that are concurrently modified by multiple threads. In such situations the RMW instructions must be used. The most commonly supported RMW operations are:

- **Test and set (T&S)**: The T&S instruction atomically reads a memory location, sets the memory location to one and returns the previously read value of the memory location. The T&S instruction is generally used to implement locks.
- **Compare and swap (CAS)**: The compare-and-swap (CAS) instruction atomically compares the value of a memory location with a supplied value and, if the values are equal, the content of that memory location is swapped with a second supplied value. CAS operations are used to implement locks as well as lock-free algorithms. The CAS instruction is the only operation in this list that is sufficient to implement lock-free algorithms.
- **Load-linked and store-conditional**: This instructions implement a more pipeline friendly version of the T&S instruction. Hence, these two operations are also used to implement locks.

At least one RMW operation must be supported by the processors ISA to enable the synchronization of multi-threaded applications. The blocking primitives (locks, critical sections, semaphores, ...) rely on the above listed low level atomic operations or memory fence instructions.

Linearizability

The term *linearizability* was defined by Herlihy and Wing in 1990 for concurrent objects to formally and informally argue about the correctness of concurrent object oriented programs (Herlihy and Wing, 1990).

- The principle concept of *linearizability* is that each method call should appear to take effect instantaneously at some point between its invocation and response.
- *Linearizability* has also the *non-blocking* property, which states that a pending invocation of a method is never required to wait for another bending invocation to complete (Herlihy and Shavit, 2008).

Atomic operations are linearizable by its nature. They are indivisible and have a success-or-fail definition. Thus atomic operations take effect instantaneously and always return without blocking the execution of other invocations. Lock-based methods instead are not linearizable: A lock-based method **appears** to take effect instantaneously (the critical code region serves as a *linearization point*), but a pending method invocation might block another method invocation, which does not fulfill the *non-blocking* property.

Let us consider the FIFO queue again. If we use a blocking primitive to synchronize the queue access, it is relatively easy to argue about the correctness of the algorithm. As long

as the lock that protects the shared resources is owned by a thread, no other thread is able to access that resource. The critical section of each method can also serve as the *linearization point*, which makes the method to apparently take effect instantaneously, but effectively blocks other threads to not violate the invariants. A non-blocking algorithm instead must be designed to make all changes visible by executing a single atomic instruction. If this atomic instruction succeeds, then the protected resources are correctly updated, otherwise the changes have to be discarded. The atomic instruction does not block other threads of execution. This atomic instruction serves as the *linearization point* in the algorithm.

Some of the modern processors implement also a transactional memory system, which allows the concurrent execution of several memory operations without blocking other threads. The hardware makes the memory operations to appear atomically by automatically verifying, if the thread that started the transaction, was the one and only thread that modified the corresponding memory location at the end of the transaction. If this is true, then the transaction is valid and all memory locations will be updated. Otherwise the transaction fails. Algorithms that use hardware transactional memory systems for data synchronization are also *linearizable*. It is possible to imitate transactional memory in software, which is called software transactional memory (STM). If the STM system is implemented based on non-blocking algorithms, then also the STM system fulfills the *linearizability* requirements.

Progress conditions

We can define the following blocking and non-blocking progress conditions for algorithms, data structures and objects (Herlihy and Shavit, 2008):

- **Blocking**: An object has the blocking progress condition, if a pending method invocation can block because of another pending invocation. The blocking progress condition generally results from the use of blocking synchronization primitives such as locks, critical sections, semaphores, condition variables, monitors, futures and promises. (A spinlock is also a blocking primitive.)
- **Wait-free**: A method is wait-free if it guarantees that every call finishes within a finite number of steps. Wait-freedom is a non-blocking progress condition. A bounded wait-free method sets a finite upper bound for the number of required steps. A wait-free method is always lock-free, but not vice versa. Wait-free algorithms are especially interesting in systems were real-time behavior is required, because a wait-free algorithm guaranties that all threads make progress if they take some steps. However, wait-free algorithms might lack performance and thus sometimes weaker progress guaranties are acceptable.
- **Lock-free**: A method is lock-free if it guaranties that infinitely often *some* method call finishes in an finite number of steps. Lock-freedom is a non-blocking progress condition. Fast lock-free algorithms might be preferable to slower wait-free algorithms, although some threads could starve, which is, however, in most situations very unlikely.

In this section we did not claim for completeness, instead we wanted to define the terminology and the hardware features that are subsequently used in this work to implement a lock-free highly concurrent software system. The following references served as the basic literature for writing this section and provide a comprehensive work of references: (Vajda, 2011), (Sutter, 2012a), (Tanenbaum, 2001) (Herlihy and Shavit, 2008), (Williams, 2012), (Sutter, 2012b), (Wrinn, 2007a), (Wrinn, 2007b) and (*Intel 64 and IA-32 Architectures Software Developer's Manual* 2016).

3 System architecture and design

The complexity of an automotive test setup can range from a simple component test to a road simulation test of a four wheel drive vehicle, or the automation of a complete test field consisting of several different test benches. Considering the overall field of automotive testing would go far beyond the scope of this work. Thus this work focuses on the design and the evaluation of a software system for the automation of a single automotive test bench, as illustrated in figure 1.1. In order to evolve a general software design that is applicable for a wide range of measurement and automation systems and not only for automotive testing, the considered functionality and data flows are generalized. Of course, the examples and considerations always relate to automotive testing and the experiences gained during the last two decades of development in this field, are also taken into account. If you are interested in already implemented systems and the whole range of software functions, please visit http://www.ksengineers.at/en/Automotive-Testing.

3.1 Domain specification

The core components of the test bench, depicted in figure 1.1, are the *real time system* and the *runtime environment*. Both units are responsible for managing and controlling the main data traffic on their specific layer. The data flow diagram in figure 3.3 illustrates the generalized data flow of the runtime environment (RTE) during the test execution. Of course, before the *Test Bench* can be started the start-up engineer (and/or the test bench operator) have to set up the test and measurement specification as well as the test bench automation. In order to provide a flexible and easy-to-use system configuration all input/output-signals and states are mapped to user defined variables. These variables can be used in scripts, equations, calculations and charts like variables in a programming language. In general each variable represents a physical value, such as time, velocity or temperature. Thus the user can assign the variable's unit, sampling rate, aliases and more. The measurement specification defines the storage format and whether a variable should be recorded or not. The display specification determines the variables that should be visualized in diagrams, controls and views during the test execution. Complex test protocols and the test automation are specified via scripting languages. The software drivers are responsible for the interaction with the measurement hardware located at the test bench. During the test execution the test bench operator is able to change the display and measurement specification. The measurement data and the

system states are continuously logged either to the local hard disk drive (HDD) or to a remote database (DB).

3.1.1 Component oriented view

Before the data flow of the runtime environment (RTE) is further examined, the domain model in figure 3.1 provides a component oriented view of the domain. The separation of an automotive test bench setup into two layers, as shown in figure 1.1, is also reflected by the two sub-domains, the *test bench environment* and the *industrial PC* with its operating system.

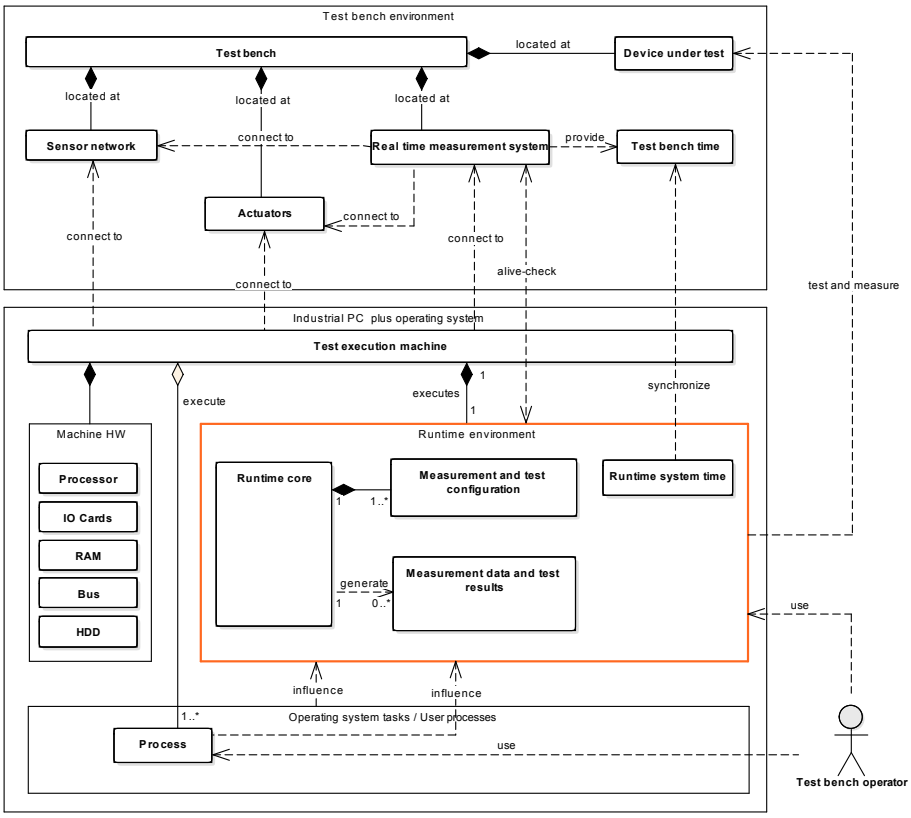

Figure 3.1: **Domain model of the test bench setup.** The testing domain can be separated into two sub-domains, the *test bench environment* and the *industrial PC*. The *industrial PC* executes a general purpose operating system and the resources are shared among the *runtime environment* and all other running processes. The *runtime environment* synchronizes its time source with the time base provided by the *real time system*, which is located at the *test bench environment*. In the case of a hardware or software failure several components, including the *real time system* and the *runtime environment*, can trigger an emergency stop. Hence, the data processing of the *runtime environment* must be designed to allow the reaction on important events within an accurate time slot.

The *test bench environment* comprises all components to measure and influence the

physical states of the device under test (DUT), while the *industrial PC* is responsible for monitoring, controlling and recording these physical states. The real time system (RTS) accurately times the control signals and further collects the measurement data, appends time stamps and sends the packaged data to the runtime environment (RTE). Of course, the RTS is not the only data source of RTE, because the RTE is also able to directly fetch data from the sensor network. If the data is retrieved from the sensor network, the RTE must generate the time stamps by itself. Hence, the *runtime system time* must be synchronized with the *test bench time*.

In order to provide accurate measurement results and correct control signals the data processing of the RTE must be designed to react on events within an accurate time slot. The size of such a time slot depends on the defined sampling rate of the variables and the priority level of the events. In the case of a hardware or software failure the reaction time must be as short as possible. Next to the RTE and the basic operating system tasks, the test bench operator can execute several other applications (eg. test data post-processing), which also occupy the hardware resources of the *industrial PC*. Nevertheless, the RTE must remain responsive even in the case of a resource shortage, because if the *alive-check* signal between RTE and RTS exceeds a hard time limit, the RTS triggers an emergency stop (and vice versa). To guaranty a safe operation of the test bench also several other components can trigger an emergency stop.

3.1.2 Software developer's view

The architectural stack in figure 3.2 further refines the domain model and provides the view on the involved components and tools from the perspective of a software developer. The amount of data traffic that must be processed by the overall system depends on the testing requirements and is determined by the complexity of the test bench automation and the required accuracy of the measurement data (sample rate and sample size). In order to design a highly responsive and flexible software system that scales with the technological progress in information technology (see section 2.1), it is important to take the hardware features of modern processors and its interconnection technologies into account. The hardware layer in figure 3.2 shows these key components. The green shaded boxes depict the components of a modern CPU that manage the data traffic inside the processor. The efficient use of these components is crucial for the performance of a concurrent application and even more important for a software system with high data traffic, like a measurement and automation system.

The DUT is inter-connected with the control and measurement devices, which are linked to the IO-cards. The IO-cards use standard bus protocols and/or memory-mapped IO to make the data available for the software applications. The application layer, especially the device drivers, utilize functions of the middleware layer to access this data. The middleware layer forwards the requests to the processor. The processor (green shaded boxes) manages all data accesses of the application and the data transfer between the application threads. The application's data flow must be designed to efficiently use the processor hardware and should scale well with the future technological evolutions.

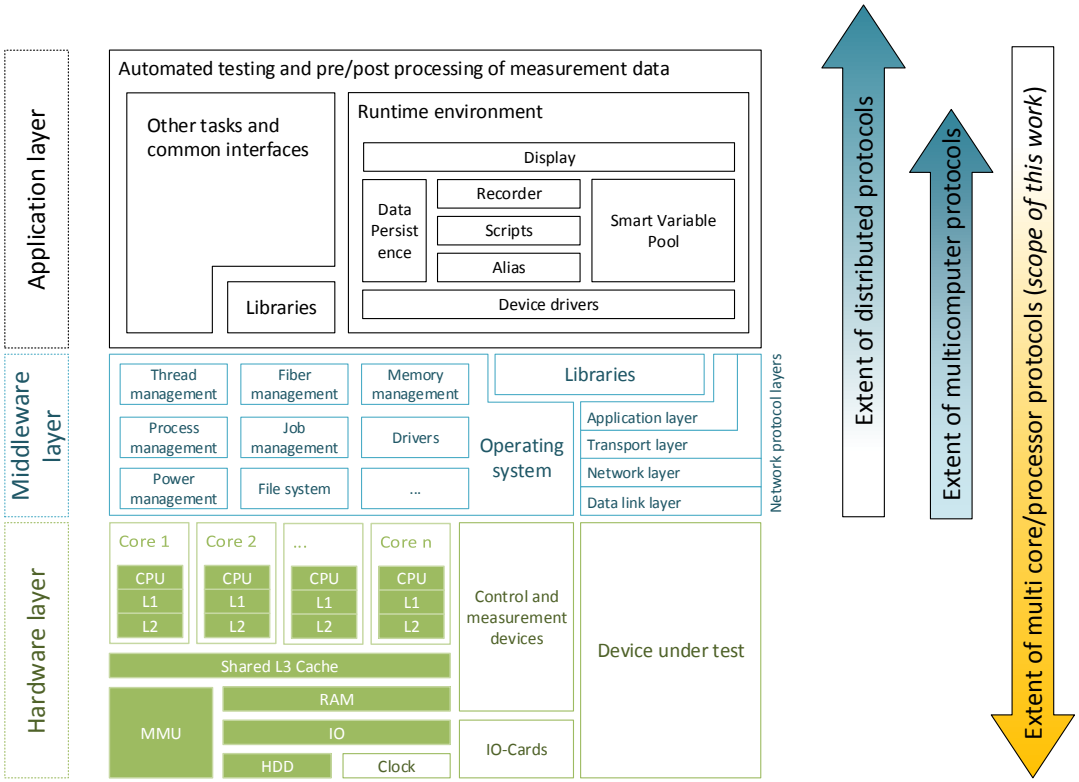

Figure 3.2: **Architectural stack of the test bench setup.** The illustrated stack depicts the domain components from a software developers point of view and is structured into three layers. The arrows on the right hand side indicate the applicability of different concurrency protocols per layer. The RTE is located at the application layer and utilizes the operating system components and the network protocol layer for the interaction with the hardware devices. The IO-cards provide the interface to the test bench hardware and allow an easy integration of various hardware devices with different bus protocols.

3.1.3 Data flow oriented view

From the component and developer views we return to the data flow diagram shown in figure 3.3, where the diagram illustrates the data flow during the execution of a test run. Here the system is structured into three layers. The *test bench layer* comprises the device under test (DUT), the actuators and the measurement hardware. The *runtime layer* depicts the core software functions that are required for the measurement and automation tasks. The *persistence layer* shows the different data storage opportunities.

The physical states of the DUT and other hardware components are mapped to the variables of the *runtime layer*. A *runtime variable* can represent a measurement value **DF1**, a control signal **DF6** or a variable, which is required for the data exchange between the software components or in calculations. The software drivers establish the connection to the hardware devices and encapsulate the protocols from the business logic of the

RTE. The input data from the test bench **DF1** is buffered by the software drivers and forwarded **DF2** to the *smart variable pool*, which maintains the *runtime variables*. The set of all *runtime variables* represents the physical states of the test bench hardware and the virtual states of the software itself.

For the automation of a test bench, with a considerable complexity, about 3000 to 6000 variables are required. The sample rate of a single *runtime variable* can be individually configured and may range from $1Hz$ up to $10kHz$. The *runtime layer* receives the data samples with approximately 800 $\frac{kSamples}{s}$ from the hardware devices **DF1**. In general, a sample consists of a single floating point value with a time stamp and an identification number, but a sample may also be a vector of several hundred floating point values or a text of an arbitrary size. The software drivers buffer the samples, which are then forwarded to the variable pool **DF2**. Each sample must be processed by the *runtime layer* within an accurate time slot. The processing of a sample includes the updating of the associated *runtime variable* together with its alias resolution, filtering and the updating of all calculations and components, which are referenced by the specific *runtime variable*. In order to reduce the computational expense down-sampling is applied. The processing steps are condensed in the data flow diagram by the *Execute Scripts*, the *Resolve Aliases* and the *Smart Variable Pool* components. Section 3.2 reveals more details about the data processing steps.

The *measurement specification* determines the variables, which should be recorded and their corresponding storage format and destination. The *display specification* defines the views, diagrams, graphs and controls that are displayed during the test execution. Both specifications may be updated while a test run is active.

In order to reduce the amount of slow write operations the measurement data is only periodically stored to disk. The slow storage operations should also be decoupled from the fast data processing steps of the *runtime layer*. This would yield a more responsive and predictable system, because the *runtime layer* data flow is no longer affected by the non deterministic waiting for completed IO operations.

The set of all *runtime variables* reflects the hardware and software states of the system and hence, the variable pool is used by all software components for managing the test automation and the measurement tasks. The efficiency of the variable pool access will significantly influence the overall performance of the software system.

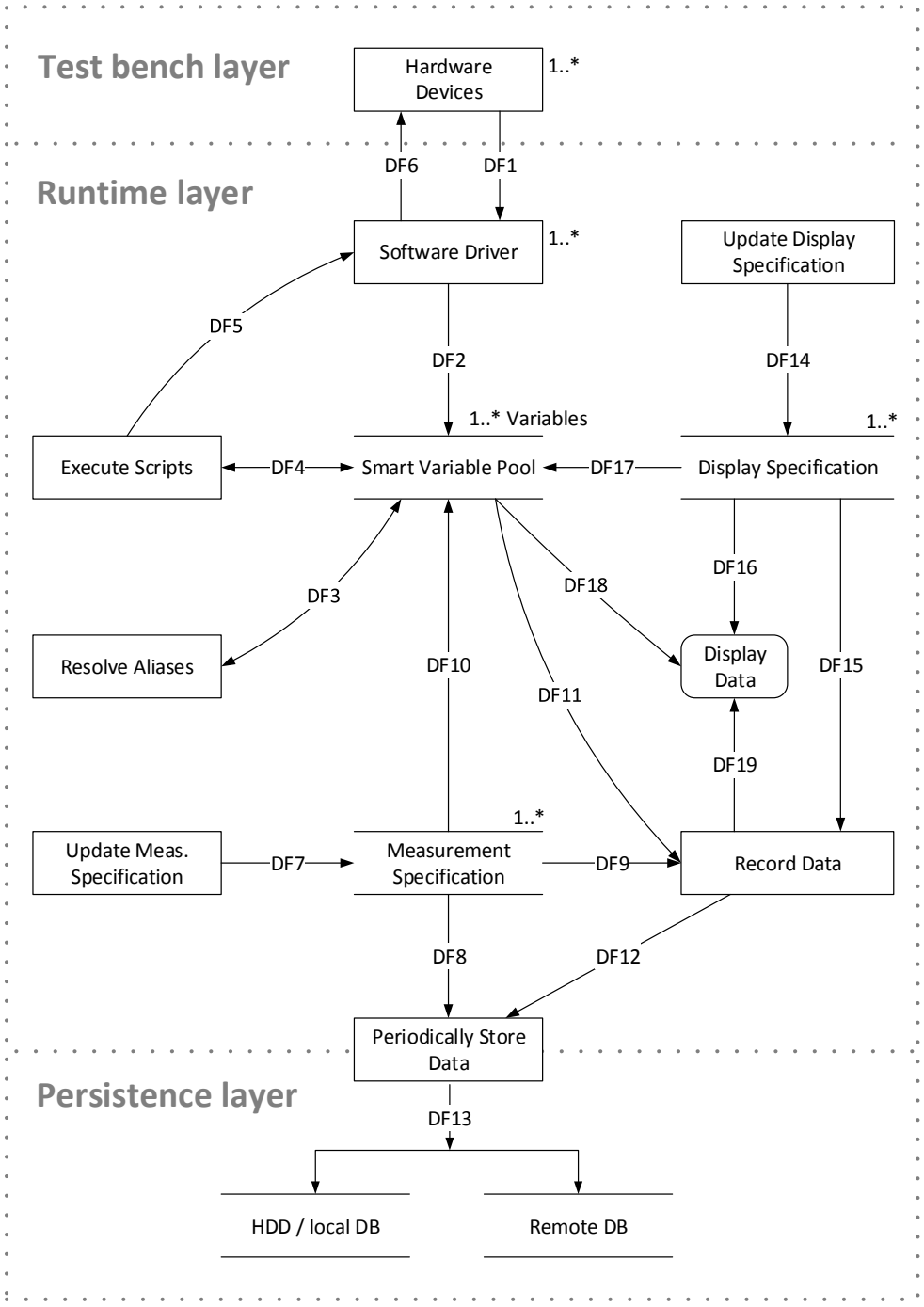

Figure 3.3: **Data flow diagram of the *runtime environment*.** The diagram shows the data flow of the *runtime environment*. The *test bench layer* comprises the device under test (DUT), the actuators and the measurement hardware. The *runtime layer* depicts the core software functions that are required for the measurement and automation tasks. The *persistence layer* shows the different data storage opportunities.
The *runtime environment* as a whole initializes the measurement hardware, controls and monitors the test bench states and is responsible for collecting and storing the measurement data during the test execution. The test bench operator configures the mapping of input/output-signals and system states to *variables*. The *variables* are maintained by the *smart variable pool*. Several *software drivers* establish the connection to the measurement hardware and the user is able to visualize and control the interaction of all components by using scripting languages, various build-in controls like views, charts, controllers, Matlab models and other tools.

3.2 Software design of the runtime environment

Taking into account the required functionality of the runtime environment (RTE) and the domain constraints presented in this the previous section, the software design depicted in figure 3.4 evolved. The class diagram reveals the implementation of the core functions, the system setup, the system configuration and the data flow dependencies during the test execution.

At the beginning of a test the *System Builder* loads the system configuration and performs the construction and initialization of all components. The initialization phase of an automotive test bench is rather complex, thus the *System Builder* is implemented using the builder pattern (Gamma et al., 1994) in combination with a state machine (state pattern (Gamma et al., 1994)). The software design is targeted towards a flexible automation system, which can be used for various scenarios. Hence, nearly all data types inherit from the *Configuration Object* type. To persistently store the configuration, a configuration object must be uniquely identifiable. The unique identifiers are also used during the test execution for the inter-component dependency resolution.

The tasks illustrated at the right bottom corner depict the active system components. For each connected hardware device located at the test bench, there exists one *Driver Task* object that is responsible for the communication. The *Driver Task* fetches and buffers the sensor data, which is then forwarded to the *Data Processing Task*. The *Data Processing Task* **a)** maintains the *Smart Variable Pool* and manages the triple buffering, **b)** is the only task with write-access to the *Smart Variable Pool* and **c)** executes all data processors that are listed in the yellow shaded *processor* group box. All other tasks are only allowed to read from the pool. If another task wants to set a variable value, the task must push an update-request to the *Data Processing Task*. If a variable update-request, eg. from the *Driver Task*, is received, the *Data Processing Task* starts with checking the type and the time stamp of the received data. Based on the type, time stamp and sample rate only some or the complete data processing steps have to be executed. The most important processing steps are listed in the yellow shaded *processor* group. The dependences and the execution order of the processing steps are rather complex and may also dynamically change based on the system configuration. Thus the pipes and filters pattern (Buschmann, Henney, and D. C. Schmidt, 2007) represents an accurate method to express these dependencies in a clear and comprehensible way.

The complexity of a variable update is the main reason, why the *Data Processing Task* is the only component with write access to the *Smart Variable Pool*. All variables that are affected by a variable update are not allowed to be accessed by any other component during the update process, because this might lead to broken invariants, which would further result in undefined behavior. Acquiring exclusive access to all affected variables would be too expensive. Hence, the triple buffering is implemented. The triple buffering allows the *Data Processing Task* to process a variable update, while all other tasks are still able to read (transactional) from the *Smart Variable Pool*.

All tasks can further register for the observation of *runtime variables*. If a task observes a variable, the *Data Processing Task* pushes all state changes of that variable directly to the observer, which enables the observer to receive all variable updates without performing a single transactional read. This design reduces the contention and synchronization overhead on each variable access and further lowers the system load of the *Measurement Task*. If the observation of variables would not be possible, the *Measurement Task* would have to actively poll the variable states.

The class diagram also depicts that the settings, which can be changed during the test execution, are linked to the *runtime variables* and are encapsulated by the *Smart Variable Pool*. Thus all configuration updates must also be pushed to the *Data Processing Task*. So the *Data Processing Task* manages all system state changes, which results in a simplification of the code that is required for handling the order- and timing-dependencies of the state transitions. Furthermore the debugging of the system is facilitated, because all state transitions can be easily observed by the developer, displayed by the *UITask* or logged to disk.

3.2.1 Data synchronization and task synchronization

In order to choose a suitable synchronization mechanism let us first recall the properties of the runtime environment (RTE). The RTE is data flow oriented. The physical states are measured by the sensors and are then transfered to the RTE. Based on the received data the RTE performs user defined data transformations. The transformation results are transmitted back to the hardware devices and/or are persistently stored. The amount of received data ($\sim$ 800 $\frac{kSamples}{s}$) is much higher than the *Data Processing Task* is able to completely process. Hence, down-sampling is applied and only the transformations, which are affected by a specific data sample, are recalculated. The storage operations and the data transmission require significantly more time than the data transformations, because the data transformations can be completely executed in-memory and do not comprise IO-operations.

In short, there are two discrepancies. **a)** The RTE receives more data samples per time interval, than it is able to completely process and **b)** the RTE generates more data signals and measurement data per time interval, than the execution machine is able to store or transmit back. Both phenomenas can be reduced to the producer/consumer problem. In scenario a) the hardware devices represent the producers and the RTE is the consumer. In scenario b) the situation is vice versa. From the perspective of a software developer the hardware devices are abstracted by the device drivers, which encapsulate the specific protocols and characteristics of each device and the communication channel. This is an important and essential simplification, because the term *hardware device* can now be one by one replaced by the term *device driver*, which reduces the complexity of the data flow inside the RTE. Instead of device specific protocols, the software developer can use the *runtime variables* for the communication with the hardware. Furthermore each device driver can serve as a buffer and data synchronization point.

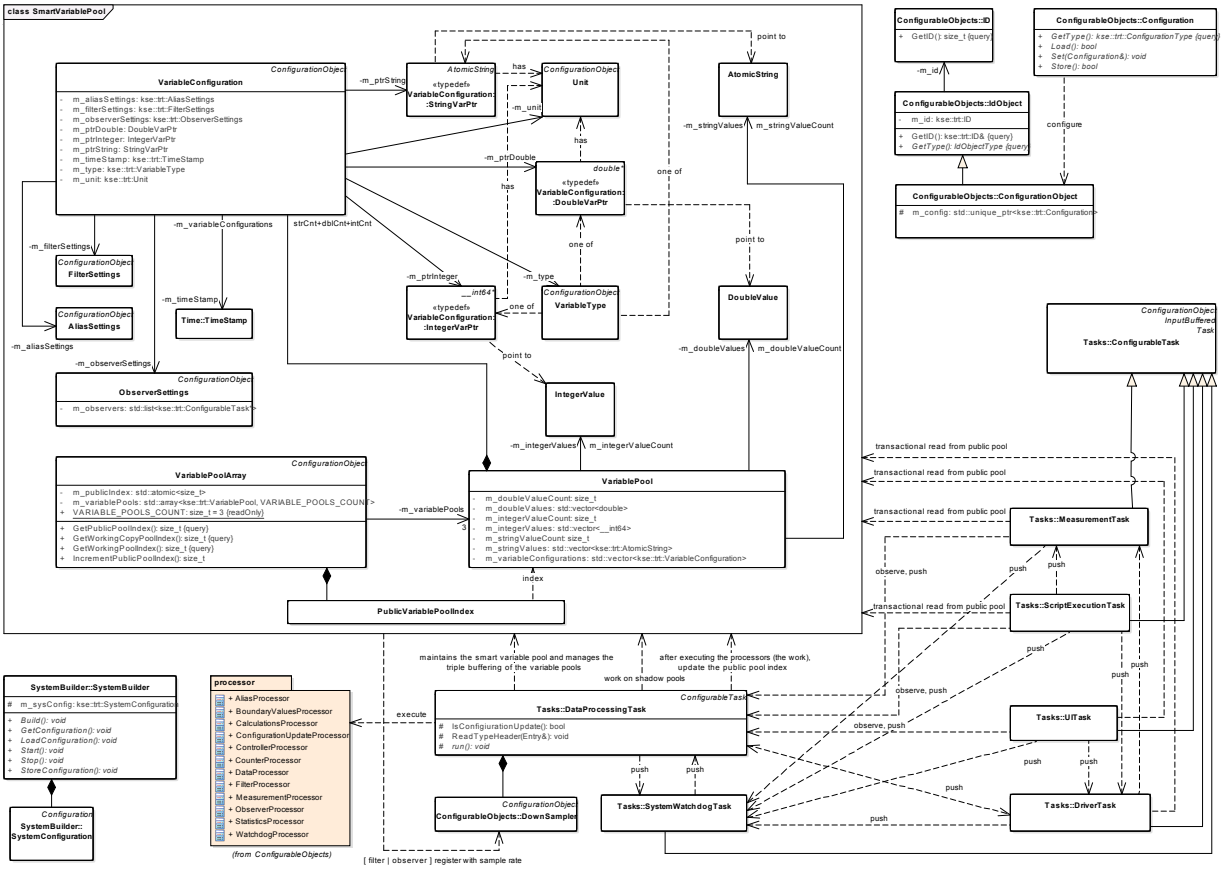

Figure 3.4: **Class diagram of the runtime environment.** The diagram depicts the software design and data flow dependencies of the runtime environment (RTE). An automation system heavily relies on the provided sensor data and thus the RTE is designed to be data flow oriented.
The *Data Processing Task* maintains the *Smart Variable Pool* and manages all system state transitions based on the data samples and commands received from the other tasks. The *Driver Tasks* establish the connection to the hardware devices at the test bench. The *Measurement Task* generates the test and measurement protocols. The *Script Execution Tasks* control the test execution and test bench automation. The *System Watchdog Task* monitors the overall system. In the case of a software or a hardware failure the *System Watchdog Task* and the *Data Processing Task* are able to trigger an emergency stop independently. If the runtime environment fails, several hardware devices can also trigger an emergency stop.
All tasks are configurable. The configuration is allowed to be changed during a test run by the user. At the beginning of a test run the *System Builder* loads the system configuration and initializes the software, establishes the communication channels and configures the hardware devices located at the test bench.

Interposing FIFO queues for the data and task synchronization seems to be the optimal solution. As the class diagram reflects, *input buffers* are already provided as a technique for latency hiding, which is especially important if IO-operations are required. Of course, the *input buffers* are also used for latency hiding between different threads of execution to facilitate parallel data processing. A FIFO queue combines the required buffering and synchronization capabilities and thus is well suited for the inter-thread communication in the presented software design. Hence, the next chapter introduces an intrusive bounded

lock-free causal FIFO queue algorithm, which was especially designed to best support the data flow oriented software design presented in this chapter.

4 A universal intrusive bounded lock-free causal FIFO queue

The software design presented in chapter 3 depends on FIFO queues for the task and the data synchronization. For the efficient implementation of this design, in this chapter, we introduce a universal fine-grained queue interface and an intrusive bounded lock-free causal FIFO queue algorithm. The developed queue emerged to be a fast and universally applicable synchronization mechanism, which is suitable for intra-process and inter-process communication, even for the communication between 32-bit and 64-bit processes. Both, the interface and the algorithm, are especially designed to best support the data flow oriented software design described in chapter 3.

4.1 A universal fine-grained queue interface design

The universal queue interface design, illustrated in listing 4.1, provides fine grained access to the queue's synchronization mechanism and the internal memory buffer. A producer, making use of this interface, calls the *acquire* method at first, asking for exclusive access to a queue buffer region of the desired size. If the call succeeds, the producer continues with writing its data into the queue buffer utilizing the interface provided by the *Entry* data type. After production the data is made visible for consumption by calling the *enqueue* method.

As soon as an entry is available for consumption, a call to the *dequeue* method succeeds and the caller is granted exclusive access to the data encapsulated by the entry. Calling the *release* method frees the reserved buffer region for subsequent data production. To enable the consumption of multiple consecutive entries the bulk reading operations are supplied.

Depending on the underlying synchronization algorithm the methods that are responsible for the data transfer (*acquire*, *enqueue*, *dequeue* and *release*) could arbitrarily fail. Hence, these methods return a boolean value, which indicates the success of the operation. The *empty* method is allowed to be called by producers and consumers concurrently and returns *true*, if no entries are available for consumption.

Listing 4.1: Universal fine grained queue interface

```
1 class UniversalQueue {
2 public:
```

```
3   using Entry          = my::Entry;          ▷ see listing 4.3
4   using Bulk           = my::Bulk;           ▷ see listing 4.4
5
6   using value_type     = Entry::value_type;  ▷ smallest accessible buffer region
7   using value_ptr      = value_type*;        ▷ pointer type to a buffer region
8
9   UniversalQueue(const unsigned int32 bufferSize, const unsigned int32
        sequenceArraySize);
10
11  bool empty() const;
12
13  bool acquire(const unsigned int32 requiredMemSize, Entry& entry);
14  bool enqueue(Entry& entry);
15
16  bool dequeue(Entry& entry);
17  bool release(Entry& entry);
18
19  bool dequeue(const unsigned int32 maxBulkSize, Bulk& bulk);
20  bool release(Bulk& bulk);
21 };
```

For most software applications the *UniversalQueue* interface might be too general, which would result in unnecessary lines of code. For this reason listing 4.2 exemplifies the implementation of the commonly used push and pop interface by wrapping the *UniversalQueue* implementation. The *try_push* method, in line 18, starts with calling the *acquire* method. If the return value is *false*, a counter is incremented and another attempt is made until the maximum counter value is reached.

If the call to *acquire* succeeds, exclusive access to a queue buffer region is granted. The program then can continue with writing data ⓐ into the buffer. Afterwards the *enqueue* method is called to release the buffer region and to make the data visible for consumption.

When building a push/pop wrapper around the *UniversalQueue* class, the desired wait-strategy, which fits best your application's needs, can be chosen. The shown wait-strategies ⓑ ⓒ ⓓ are all busy-wait implementations that are common for lock-free algorithms. Of course, blocking wait-strategies can also be applied. Providing a generic interface ⓔ moves the selection of the wait-strategy to the application developer. For this purpose mechanism such as function objects or lambda functions can be used in C++, or the interface pattern in Java. The other method implementations, not shown in this listing, follow the same pattern as the *try_push* and *push* methods and can be realized in a similar way.

The *acquire* and *enqueue* methods, used in the implementation of *try_push*, could be redirections to functions like *mutex.lock()* and *mutex.unlock()*, or they could implement some other synchronization mechanism, which guaranties the exclusive queue access. The important observation here is, it does not matter. The entire implementation details are encapsulated by the *UniversalQueue* interface. Summing up, the *UniversalQueue* interface implementation yields the following benefits:

- A flexible and stable interface early in the development process, which makes parallel front-end and back-end development possible.
- The implemented synchronization algorithm is transparent to the client code
  - Blocking, lock-free and wait-free synchronization algorithms are applicable
  - Various use cases can be supported:
    - single-producer single-consumer (SPSC)
    - multiple-producer single-consumer (MPSC)
    - single-producer multiple-consumer (SPMC)
    - multiple-producer multiple-consumer (MPMC)
  - Algorithmic optimizations can be implemented later in the development process. Hence, an executable synchronization mechanism can be provided quickly, if a well known or simple algorithm is implemented at first. The algorithm may later be changed, extended or improved. This yields an early executable implementation, which can not only be tested and debugged, but also benchmarks can be executed to check, if further algorithmic improvements are required.

Of course, one drawback of the *UniversalQueue* interface is the overhead due to the additional method-calls, compared to the push/pop interface. Nevertheless, section 4.1.1 explains the positive performance impacts of the fine grained interface, which outweigh the influence of the additional function calls.

Listing 4.2: Push and pop queue interface

```
1  class PushPopQueue : private UniversalQueue {
2  public:
3    PushPopQueue(const unsigned int32 bufferSize);
4
5    bool try_push(Entry& entry);
6    bool try_push(const size_t maxRetries, Entry& entry);
7    void push(Entry& entry);
8
9    bool try_pop(Entry& entry);
10   bool try_pop(const size_t maxRetries, Entry& entry);
11   void pop(Entry& entry);
12
13   template <typename T>
14   bool try_push(T& waitStrategy, Entry& entry);   ▷ (e)
15   ...
16 };
17
18 bool try_push(const size_t maxRetries, Entry& entry) {
19   Entry e;                                  ▷ provides access to the queue buffer
20   size_t retryCount = 0;
21   while(!this->acquire(entry.GetSize(), e)) {   ▷ acquire exclusive access
22     ++retryCount;
23     if (retryCount > maxRetries) {
24       return false;
25     }
26     std::this_thread::yield();              ▷ (b) yielding-wait
```

```
27    }
28    e = entry;                                  ▷ ⓐ copy data into the queue buffer
29    while(!this->enqueue(e)) {                  ▷ release exclusive access and make the data
          visible for consumption
30      ;                                         ▷ ⓒ busy-wait
31    }
32    return true;
33  }
34
35  bool try_push(Entry& entry) {
36    return try_push(0, entry);
37  }
38
39  void push(Entry& entry) {
40    while(!this->try_push(entry)) {
41      std::this_thread::sleep_for(1ms);         ▷ ⓓ sleeping-wait
42    }
43  }
44
45  ...
```

The *Entry* data type, declared in listing 4.3, encapsulates the queue buffer implementation and is initialized by calling the *acquire* or *dequeue* method. The given implementation provides direct access to the queue's buffer memory via the *GetPtr()* method. The *GetSize()* method returns the size of the encapsulated buffer region. The private member variables may be utilized for transporting information ⓐ from the *acquire* method to the *enqueue* method, or from *dequeue* to *release*. Such a data transfer could be required if a non-blocking synchronization algorithm is implemented, or if the queue buffer should be exclusively reserved for client data only.

Listing 4.3: Encapsulation of the queue buffer memory

```
1  namespace my {
2
3  class Entry {
4  public:
5    using value_type  = unsigned char;
6    using value_ptr   = value_type*;
7
8    virtual unsigned int32 GetSize() const {
9      return pSeqEntry->size;
10   }
11
12   value_ptr GetPtr() {
13     return &(pBuffer[pSeqEntry->startIndex]);
14   }
15
16 protected:
17   value_ptr        pBuffer;     ▷ pointer to the queue memory buffer
18   SequencePtr      pSeqEntry;   ▷ pointer to a sequence array entry
19   unsigned int32   seqNum;      ▷ ⓐ absolut sequence position
20 };
```

```
21
22 }
```

Listing 4.4 defines the *Bulk* data type, which is designed for bulk reading operations only. The *Bulk* class publicly inherits from the *Entry* class and overrides the *GetSize()* method. A successful call of the *dequeue* method initializes a bulk. The bulk then provides exclusive access to the buffer memory of one or more consecutively enqueued entries. The presented implementation provides access to a memory bulk, but not to a single entry within the bulk. Obviously, the class interface can also be extended to provide access to each single entry's memory region. Calling the *release* method releases the exclusive access to the encapsulated buffer regions and makes the queue memory available for data production again. The implementations of the *bulk-dequeue* and the *bulk-release* operations are given in listing 4.9.

Listing 4.4: Consumer interface for accessing multiple consecutive queue entries

```
1  namespace my {
2
3  class Bulk : public Entry {
4  public:
5    virtual unsigned int32 GetSize() const override {
6      return bulkSize;
7    }
8
9  protected:
10   unsigned int32  bulkSize;   ▷ buffer size occupied by the encapsulated consecutive entries
11 };
12
13 }
```

4.1.1 Comparison of queue interfaces

The sequence diagrams, in figure 4.1, illustrate the communication between two producers and one consumer that utilize the queue interfaces explained above. The sequence diagram A applies the fine grained interface declared in listing 4.1, while the sequence diagram B uses the push/pop interface presented in listing 4.2. In both interleavings all queue accesses are strictly sequential, which in practice could result from implementing a blocking synchronization mechanism. Furthermore, the initial and the final state of the queues, Q_A and Q_B, are identical for both program executions. At the beginning the queues are empty. At the end the producers have enqueued their entries, and the consumers, C_A and C_B, have consumed the entries produced by P_{1A} and P_{1B}.

Interpreting the length of the lifelines as elapsed time, we see that P_{2A} finishes earlier in time than P_{2B}. This behavior results mainly due to two reasons:

- In scenario B the producers have to copy ❶ their previously produced data into the queue, while the producers in diagram A can directly access ❷ the queue buffer. Although the producers in A do more methods calls, they still have to execute less

work, compared to the additional copying overhead in scenario B. The same holds for the consumers reading the data.

- The second and more decisive reason is that the interface in scenario A provides a finer grained access to the queue's synchronization mechanism, which results in less contention on the queue. Thus allowing the program to execute a larger portion of its work in parallel.

The impact of the second argument is further emphasized in section 4.2, where a lock-free synchronization mechanism is implemented to preserve the queue's invariants, resulting in an even more parallel program execution.

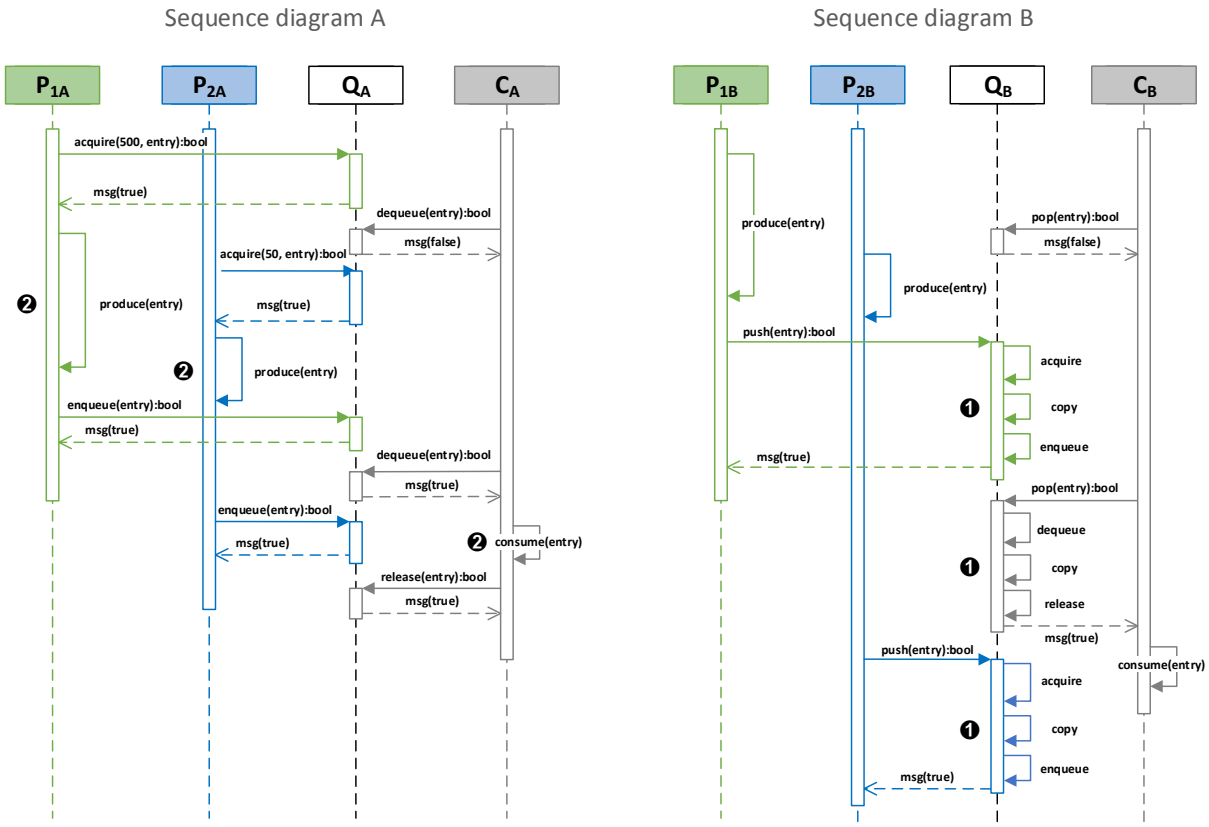

Figure 4.1: **Sequence diagram of a sequential MPSC run that compares the two queue-interface implementations.** Both sequence diagrams exemplify a possible interleaving of sequential queue accesses in a MPSC scenario. In the left diagram the interface of listing 4.1 is used, while the right diagram illustrates the interleaving using the push/pop interface of listing 4.2. The initial state and the final state of both queues, Q_A and Q_B, are identical, but scenario B requires more time, because of the additional copy overhead and higher contention on the queue Q_B.

4.2 An intrusive bounded lock-free causal FIFO queue algorithm

The queue algorithm shown in this section implements the *UniversalQueue* interface defined in section 4.1. All code listings are written in C++11 pseudo code style and are compilable after applying a few modifications. The presented algorithm has the following properties:

1. **Intrusive**: The queue can be used for side by side transfer of both, binary data and objects, without the necessity of dynamic memory allocation and additional node handling. There is also no garbage collection required. These properties yield less contention on the operating system's heap manager, which increases the parallel portion of the program.
2. **Bounded**: The buffer size of the queue is fixed at construction time and write attempts to a full queue buffer do fail. The buffer in the given implementation linearizes the enqueued data, resulting in prefetching-friendly and caching-friendly reading operations of the consumer.
3. **MPSC**: The queue can be concurrently accessed by multiple producers and a single consumer at the same time. For example, fife producers are writing into the queue, while the consumer is reading an already enqueued entry. The algorithm can be extended to support multiple consumers as well.
4. **Lock-free**: In order to further increase the parallel portion of the program, a lock-free synchronization algorithm is implemented and techniques, which reduce the contention between writes to the queue and reads from the queue, are applied.
5. **Causal FIFO**: The queue algorithm guaranties a causal FIFO ordering between all producers.
6. **ABA-free**: The presented algorithm does not suffer from the ABA problem.
7. **Ready for inter-process communication**: The properties 1, 2 and 4 guaranty that the memory regions, responsible for synchronization and data transfer, can be shared between processes. If the class member alignment is correctly adjusted, it is even possible to communicate between 32-bit and 64-bit processes (implementation not shown).

Additionally, the algorithm can be adjusted to support a lock-free MPMC communication and it can also be optimized for a wait-free SPSC scenario. The required modifications are not provided in this work. However, after reading this chapter, it should be possible to accomplish the necessary adjustments. The queue's performance properties are stated in the results chapter (see chapter 6).

The explanation of the algorithm is supported by the figures 4.2, 4.3 and 4.4, which cross-reference each other using the markers ❶ to ❿. The sequence diagram (figure 4.2) depicts the execution of a lock-free program interleaving, which covers the most important internal state changes of the queue. The current value assignment of all queue member variables is referred to as the queue state or the internal state of the queue. The state changes resulting from the method calls are illustrated in figure 4.3 and the

synchronization relations, responsible for preserving the invariants of the queue, are shown in figure 4.4. Beside the properties listed above, the queue has to maintain the following invariants:

- No two threads are allowed to have exclusive access to the same queue buffer region.
- A producer is only allowed to override a queue buffer region, which is uninitialized or was already read by the consumer.
- A consumer is only allowed to read a queue buffer region, which was initialized by a producer.

4.2.1 Queue attributes and member initialization

All queue attributes are declared in listing 4.5. The member variables in capital letters (*SN*, *PS*, *WR*, *EN*, *DE*, *RD*) are used for synchronization purposes. The explanation of the public data types can be found in section 4.1 and all other elements will be explained here.

The queue buffer ⓐ is a bounded array of *value_type*'s, which is allocated at construction time. The data generated by the producers is streamed through the queue buffer, without requiring dynamic memory allocation. The producer synchronization *PS* ⓑ manages the exclusive access to the buffer and guaranties the causal FIFO ordering among the producers. The values of *WR* ⓒ and *EN* ⓓ must be accessed within one single atomic operation.

To reduce the contention between *enqueue* and *dequeue* operations and to prevent cache ping-pong the *sequence array* ⓔ was introduced. The technique of implementing a *sequence array* was first seen on the web page of Dmitry Vyukov in his multiple-producer multiple-consumer queue (Vyukov, 2014). The size of the sequence array ⓕ determines the maximum number of simultaneously enqueued entries. Together with the *enqueue position* ⓓ and the *dequeue position* ⓖ, the *sequence array* synchronizes the data transfer between the producers and the consumer.

The *read position* ⓗ points to the buffer position that the consumer is reading next, so the algorithm utilizes the positions stored in *RD* ⓗ and *WR* ⓒ to obtain the free queue buffer region. Cache line pads are inserted between the variables that are responsible for the inter-thread synchronization and around the queue buffer. The cache line pads prevent false sharing between different threads of execution.

Listing 4.5: Attributes of the intrusive bounded lock-free MPSC causal FIFO queue

```
1 class LFBoundedMPSCQueue {
2 public:
3   using Entry          = my::Entry;          ▷ provides access to the queue buffer
        memory of an entry
4   using Bulk           = my::Bulk;           ▷ provides access to consecutive entries
5
```

```
6   using value_type     = Entry::value_type;     ▷ type of the smalltest accessible buffer
        region
7   using value_ptr      = value_type*;           ▷ pointer type to a buffer position
8
9 private:
10  using ProducerSync  = struct {
11    unsigned int32 WR;                          ▷ ⓒ next free buffer position
12    unsigned int32 EN;                          ▷ ⓓ next enqueue position
13  };
14
15  using Sequence       = struct {
16    std::atomic<unsigned int32> SN;             ▷ sequence number
17    unsigned int32              size;           ▷ size of the buffer region
18    unsigned int32              startIndex;     ▷ start index of the buffer region
19  };
20
21  using SequencePtr    = Sequence*;
22
23  CACHELINE_PAD
24    const unsigned int32        bufferSize;     ▷ queue buffer size
25    value_ptr                   buffer;         ▷ pointer to the buffer start position
26    const unsigned int32        slotCount;      ▷ ⓕ sequence array size
27    const unsigned int32        slotMask;       ▷ avoids modulo calculations
28    SequencePtr                 seqArr;         ▷ ⓔ sequence array
29  CACHELINE_PAD
30    std::atomic<ProducerSync>   PS;             ▷ ⓑ producer synchronization
31  CACHELINE_PAD
32    std::atomic<unsigned int32> DE;             ▷ ⓖ next dequeue position
33  CACHELINE_PAD
34    std::atomic<unsigned int32> RD;             ▷ ⓗ next read position
35  CACHELINE_PAD
36    my::PaddedMemory            paddedMem;      ▷ ⓐ queue buffer memory, located at the
          heap and enclosed by cache line pads
37 };
```

The queue construction is illustrated in listing 4.6. The *buffer size* ⓐ defines the size of the memory region that is available for inter-thread communication and can be initialized to an arbitrary value. The *sequence array size* ⓑ controls the maximum number of simultaneously enqueued entries and must be a power of two, which yields the following benefits:

- The sequence counters (enqueue position *EN* and dequeue position *DE*) can be implemented as ever increasing values, which are allowed to overflow. This reduces the amount of branches in the algorithm and instead of using a modulo operation for accessing the sequence array entries, a simple masking operation can be applied.
- The enqueue position *EN* can serve as an alongside counter variable, next to the write index *WR* to prevent the ABA problem that is common in lock-free algorithms (Decheva, Pirkelbauer, and Stroustrup, 2010).

When the queue buffer is allocated the cache line pads are added to preventing false sharing ⓒ. To enable full operating speed already at the first access of the queue, each

memory page of the queue buffer is accessed once ⓓ, which forces the operating system to initialize the memory pages at construction time. After calling the constructor all members are set to their initial values.

The initial queue state is depicted in figure 4.3 at marker *Init*. The *Sequence Array* entries in that diagram reveal only the sequence number *SN*. The pseudo code in figure 4.4 shows the usage of the *size* and *startIndex* values, while all further algorithmic details can be found in section 4.2.4.

Listing 4.6: Queue initialization

```
1  LFBoundedMPSCQueue::LFBoundedMPSCQueue(const unsigned int32 bufSize, const
       unsigned int32 seqArrSize) {
2    bufferSize = bufSize;          ▷ ⓐ
3
4    slotCount = NextPowerOfTwo(seqArrSize);        ▷ ⓑ
5    slotMask = slotCount - 1;
6    seqArr = new Sequence[slotCount];
7
8    ProducerSync prodSync;
9    prodSync.EN = 0;
10   prodSync.WR = 0;
11   PS.store(prodSync, std::memory_order_relaxed);
12
13   DE.store(0, std::memory_order_relaxed);
14   RD.store(0, std::memory_order_relaxed);
15
16   for (unsigned int32 i = 0; i < slotCount; ++i) {
17     seqArr[i].SN.store(i, std::memory_order_relaxed);
18     seqArr[i].size = 0;
19     seqArr[i].startIndex = 0;
20   }
21
22   paddedMem = my::PaddedMemory(
23     bufferSize * sizeof(value_type), CACHE_LINE_SIZE);        ▷ ⓒ
24   buffer = paddedMem.GetPtr<value_type>();
25   for (unsigned int32 i = 0; i < bufferSize; i += PAGE_SIZE) {
26     buffer[i];          ▷ ⓓ forces a memory page initialization
27   }
28 }
```

4.2.2 Sending and receiving data via the queue

In figure 4.2 all three producers and the consumer start to access the queue nearly simultaneously. Producer P_1 returns first from its call to acquire ❶ and is granted exclusive access to the green shaded queue buffer region, which is illustrated in figure 4.3 at marker ❶. When calling the acquire method P_1 atomically updates *WR* and *EN*, which causes the state change from *Init* to ❶ as depicted in figure 4.3. The atomic variable updates are highlighted in red color.

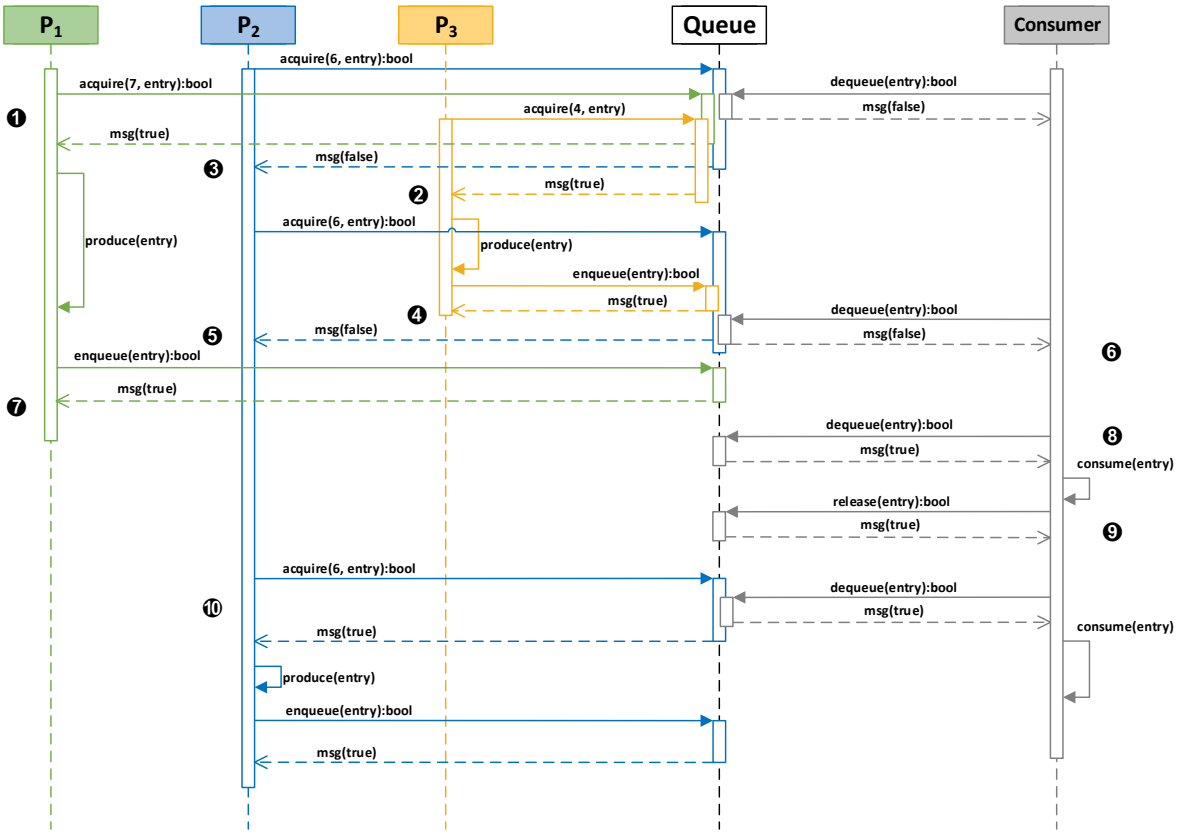

Figure 4.2: **Sequence diagram of a lock-free MPSC scenario.** The diagram shows the interleaving of multiple producers and a single consumer communicating via the lock-free queue. The numbered markers refer to markers in figure 4.3, where the corresponding state changes of the queue are illustrated and to the markers in figure 4.4, which depicts the synchronization relations.

P_1 continues with writing data into the queue buffer. In the meanwhile P_3 successfully acquired a buffer region ❷ for production. Producer P_2 was unlucky with its call to acquire ❸, because earlier in the program order P_1 updated PS, which yields P_2's call to the CAS operation, shown in figure 4.4 at marker ❸, to fail. If the CAS operation would not fail, the queue's invariants were broken, because P_2 would have got exclusive access to the memory region already owned by P_1. Furthermore P_2 could also overwrite the data already produced by P_1.

P_3 finishes with its data production and makes the entry visible for consumption by calling enqueue ❹, which changes the sequence number of the second entry from 1 to 2. Concurrently P_2 failed a second time to acquire a buffer region, because the queue ran out of space ❺. The consumer flops to dequeue P_3's entry ❻, because the FIFO ordering requires that the entry of P_1, which is currently not enqueued, must be consumed before the entry of P_3. After P_1 has enqueued its entry ❼, the consumer can dequeue the entry ❽ and read it. Afterwards the consumer releases both, the memory region encapsulated by the entry ❾ via updating the read position RD. The sequence array slot is released via increasing the initial sequence number of the first sequence array entry by the sequence

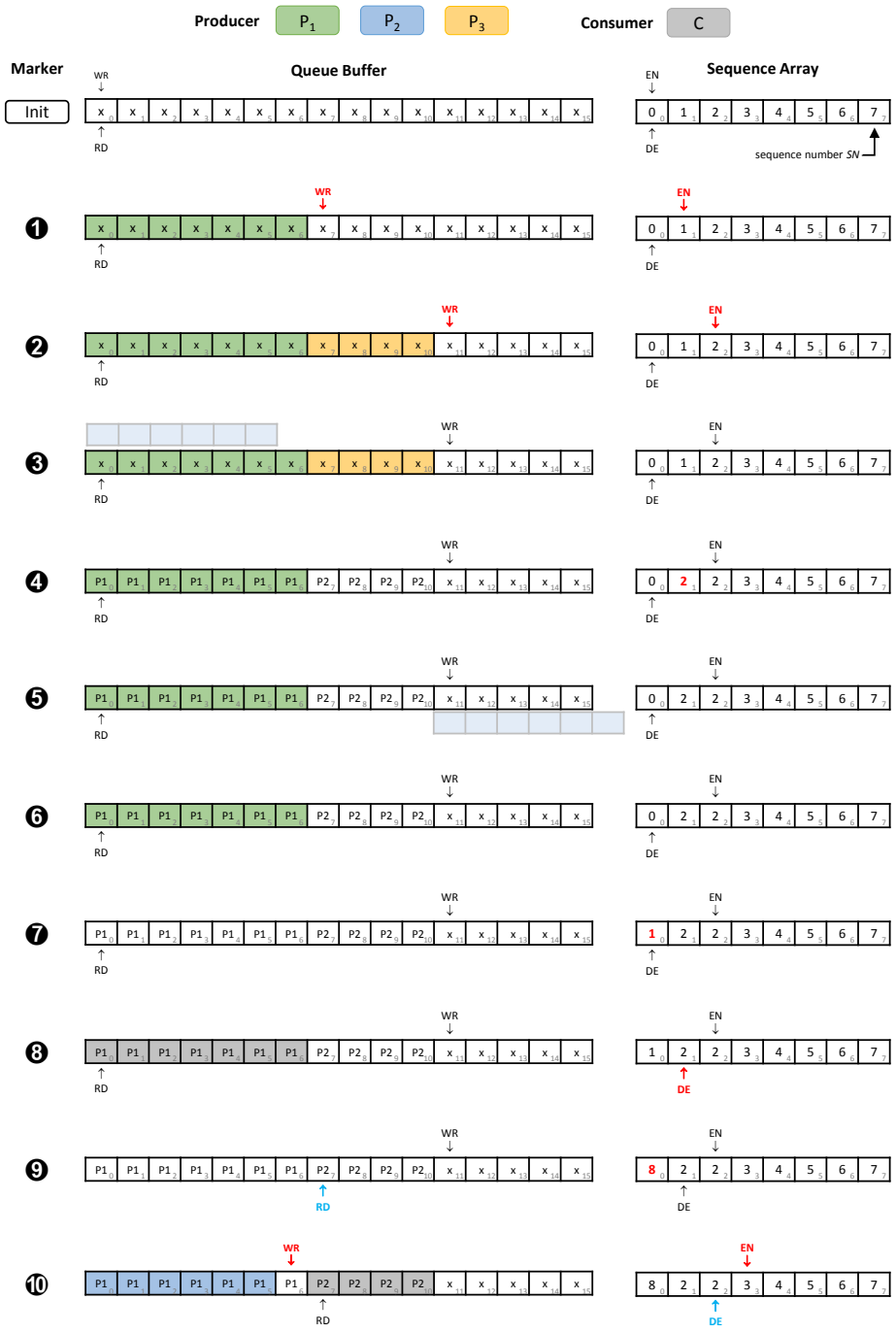

Figure 4.3: **Illustration of the internal queue states during the program execution shown in figure 4.2.** The shaded queue buffer regions indicate the exclusive ownership by the producer or the consumer. Font colors that are different from black, signal that the value has changed form the previous step to the current step. The same font color of two different values in one step points out that those values where updated within a single atomic operation. The markers on the left side refer to the markers in figure 4.2 tagging the corresponding method calls that yield the queue state changes. Furthermore the markers also refer to those in figure 4.4, which illustrates the synchronization relations of the lock-free algorithm.

array size. At that point in time no participant has exclusive access to a queue buffer region, as shown in figure 4.3 at marker ❾, but there is still the entry of P_2 ready for consumption.

At marker ❿ the consumer and the producer simultaneously access the queue. However, both successfully accomplish their operation and also do not influence each other during the execution, because the dequeuing and enqueuing operations are completely decoupled. Producer P_2 acquires exclusive access to the blue shaded queue buffer region and the consumer obtains exclusive access to the gray shaded queue region. During the execution of the *acquire* method the producer P_2 twice modifies the *write position*. The first modification of the *WR* is not shown in figure 4.3, but the pseudo code reveals the step in figure 4.4 ❿ at line 14.. After setting the write position to zero, another attempt to acquire a buffer region is launched, which updates *WR* in line 6. and then successfully returns, yielding the final queue state illustrated in figure 4.3 at marker ❿.

4.2.3 The queue invariants and the synchronization relations

The pseudo code in figure 4.4 illustrates the synchronization relations of the lock-free algorithm and the code further depicts the usage of the sequence array members. Each atomic-store-release operation synchronizes-with its corresponding atomic-load-acquire operation, as indicated by the dashed lines.

The write position *WR* and the enqueue position *EN*, encapsulated by the producer synchronization *PS*, are updated only within the *acquire* method (line 6. and line 14.). In order to allow the concurrent access of multiple producers it must be ensured that the queue invariants hold between the load operation in line 1. and the update operations in line 6. or 14.. The desired invariants can only be guarantied if *WR* and *EN* are updated in a single atomic operation and if *WR* and *EN* were not changed between the load operation and the update operation. Thus a CAS instruction is required for updating *PS*. A CAS instruction atomically executes the following three steps:

1. Load the most recent value of the variable
2. Compare the currently loaded value with the expected value. (The expected values passed to the CAS operation, are those loaded at the beginning of the *acquire* method.)
3. Update the current variable, if the current value is equal to the expected value, else the operation fails.

The read position *RD* is only updated by the single consumer inside the release method. Thus a simple store-release operation is sufficient. The store instruction then synchronizes-with the load operation in the acquire method (line 2.). The read position *RD* together with the write position *WR* is utilized to obtain the free queue buffer region.

- If there is sufficient buffer space left (line 3.) the sequence number *SN* of the current sequence array entry is loaded. This sequence number is required to check, if the obtained sequence array entry is allowed to be accessed. If access is granted, the

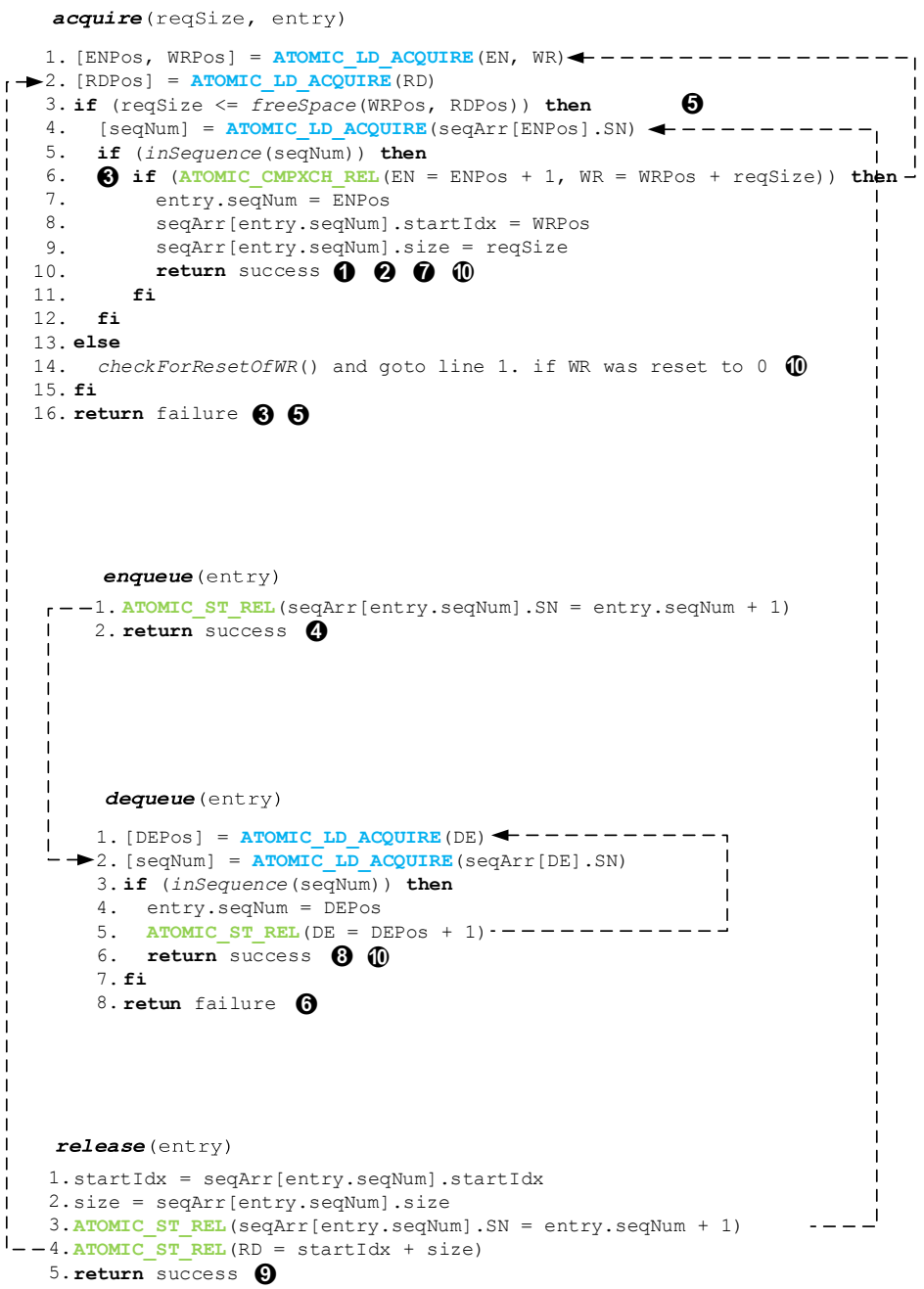

Figure 4.4: **Pseudo code demonstrating the lock-free synchronization relations.** The pseudo code demonstrates the information transport between producer(s) and consumer and the implemented lock-free synchronization algorithm. The dashed lines illustrate synchronizes-with relations. Operations with acquire-semantics are highlighted in blue, operations with release-semantics are highlighted in green. Release operations synchronize-with their corresponding acquire operations. The markers refer to the markers in the figures 4.2 and 4.3.

CAS operation in line 6. can be executed and on success the caller is awarded with exclusive access to the requested queue buffer and a sequence array entry.
- If the queue buffer runs out of space, it is necessary to reset the write position *WR* to the queue buffer's front (line 14.). The reset of *WR* is only allowed under certain conditions, which are depicted in the slow path ⓑ of the code in listing 4.7. If the reset of the write position succeeds, a new attempt to acquire exclusive access to a queue buffer region is started, else the acquire method returns without success.

To release the exclusive access obtained by calling acquire, the enqueue method updates the sequence number of the sequence array entry. This atomic store operation also synchronizes all write operations done by the producer before. Exactly this happened-before relation is exploited in the dequeue method by the consumer to synchronize the produced data for consumption. Indeed, there is only one consumer updating the dequeue position *DE*, but the *empty* method (see listing 4.8), which reads *DE*, might be called concurrently. Thus *DE* still must be atomically accessed.

To release the queue buffer and the sequence array entry after consumption, the consumer must call the release method. The release method updates the sequence number of the sequence array entry and the read position. Both update operations synchronize-with their corresponding load operations at the beginning of the acquire method.

To sum up, in all four methods a single atomic update operation serves as the methods' *linearization point* that makes the modifications visible. The concept of *linearization points* is explained in section 2.4.2.

Correctness of the algorithm

To explicitly specify the synchronization relations between the method calls the next listing states a simple scenario, where one producer *P* communicates with one consumer *C*. The queue buffer as well as the sequence array, consist only of a single entry: The queue buffer entry *QBE* and the sequence array entry *SAE*. The listed scenario is cyclic and shows that the production and the consumption of data can be safely executed based on the synchronization enforced by the atomic operations. The following scenario is the only possible sequential interleaving that is allowed by the algorithm. Of course, this is only a SPSC example, but as already explained, the CAS operations in the *acquire* method guaranty that the synchronization also works for multiple producers.

S1. `Setup`: Setup the queue invariants
S2. `P.acquire(QBE, SAE)`:
   a) *P* loads *EN* and *WR* within a single atomic operation
   b) *P* loads *RD*
   c) *P* checks if there is enough space available
   d) *P* loads *SAE.EN*
   e) *P* checks if the *SAE* is allowed to be accessed and if the entry is in sequence
   f) *P* acquires exclusive access to *QBE* (update *WR*) and *SAE* (update *EN*) by updating both values within a single atomic operation (*linearization point*)

S3. `P.write(QBE)`: *P* writes data into *QBE*
S4. `P.enqueue(QBE, SAE)`: *P* signals *C* that some data is ready for consumption by updating *SAE.SN* (*linearization point*)
S5. `C.dequeue(QBE, SAE)`:

a) *C* loads *DE*
b) *C* loads *SAE.SN*
c) *C* checks if some data is ready
d) *C* acquires exclusive access to *QBE* and *SAE* by updating *DE* (*linearization point*)

S6. `C.read(QBE)`: *C* reads the data produced by *P* in step S3.
S7. `C.release(QBE, SAE)`:

a) *C* releases *SAE* by incrementing *SAE.SN* (*linearization point*$_1$)
b) *C* releases *QBE* by increasing *RD* (*linearization point*$_2$)

S8. Continue with step S2.

Based on the given scenario we can deduce the synchronization relations, which further show that the invariants stated on the beginning of section 4.2 are preserved by the presented queue algorithm. Be aware that all synchronizes-with relations result from the execution of atomic operations (eg. updating *SAE.SN* in step [S4.] synchronizes-with loading *SAE.SN* in step [S5.b]) and not from the method calls per se. Instead, the method calls are used to illustrate the happens-before relations (eg. step [S6.] happens-before step [S7.] or step [S7.a] happens-before [S7.b]) in the program flow.

The parent items in the following listing informally state the invariant and the ordering requirements, which must be preserved by the queue algorithm. The child items reveal the ordering and synchronization relations which ensure that the data production and data consumption is correctly linearized between the threads and that the invariants are preserved.

- A producer is only allowed to override a queue **buffer region**, which is **uninitialized** or was already **read by the consumer**: This requires `Setup` or `C.read(QBE)` to inter-thread happen-before `P.write(QBE)`:
  - **Uninitialized region**: `Setup` [S1.] inter-thread happens-before `P.write(QBE)` [S3.]
  - **Consumed region**: `C.read(QBE)` [S6.] inter-thread happens-before `P.write(QBE)` [S3.]
    * `C.read(QBE)` [S6.] happens-before `C.release(QBE, SAE)` [S7.]
    * `C.release(QBE, SAE)` [S7.] inter-thread happens-before `P.acquire(QBE, SAE)` [S2.]
    * `P.acquire(QBE, SAE)` [S2.] happens-before `P.write(QBE)` [S3.]
      + `SAE.SN.store_rel` [S7.a] synchronizes-with `SAE.SN.load_acq` [S2.d]
      + `RD.store_rel` [S7.b] synchronizes-with `RD.load_acq` [S2.b]
      + The checks [S2.c] and [S2.e] guaranty that *P* does not access/overwrite a buffer region exclusively acquired by another thread
      + *P* acquires exclusive access [S2.f]

+ *P* can not write into the queue buffer, without initializing an *entry* by successfully calling *acquire*. The *acquire* method only returns successfully, if the queue is new or if the consumer has released the buffer region.

- A consumer is only allowed to **read** a queue **buffer region**, which was **initialized by** a **producer**: This requires `P.write(QBE)` to inter-thread happen-before `C.read(QBE)`:
  - `P.write(QBE)` [S3.] happens-before `P.enqueue(QBE, SAE)` [S4.]
  - `P.enqueue(QBE, SAE)` [S4.] inter-thread happens-before `C.dequeue(QBE, SAE)` [S5.]
  - `C.dequeue(QBE, SAE)` [S5.] happens-before `C.read(QBE)` [S6.]
    * Updating *SAE.SN* [S4.] synchronize-with loading *SAE.SN* [S5.b], which also guaranties the FIFO ordering and ensures that the data previously written by *P* is correctly synchronized between all cores and processors. Hence, a subsequent read operation of *QBE* always sees the data produced by *P*.
- No two threads are allowed to have exclusive access to the same queue buffer region.
  - The precondition for the given algorithm is that there must be exactly one consumer, thus the consumer side does not require additional synchronization to prevent two threads from reading the same queue buffer region.
  - On the producer side the *acquire* method checks in step [S2.c], if a free buffer region is available and the method uses a CAS operation in step [S2.f] to update *EN* and *WR* within a single atomic operation. Thus multiple threads may call the *acquire* method concurrently without violating the invariants.

4.2.4 Implementation details

This section reveals the C++11 implementations of the intrusive bounded lock-free causal FIFO queue algorithm. The code of the *acquire* and *enqueue* methods is shown in listing 4.7, listing 4.8 depicts the implementation of the *dequeue* and *release* methods and the bulk reading operations are stated in listing 4.9.

Listing 4.7: Implementation of the producer interface methods

```
1  bool LFBoundedMPSCQueue::acquire(const unsigned int32 requiredMemSize, Entry&
       entry) {
2    ProducerSync prodSync = PS.load(std::memory_order_acquire);
3    while (true) {
4      unsigned int32 RDPos = RD.load(std::memory_order_acquire);
5      unsigned int32 WRPos = prodSync.WR;
6
7      unsigned int32 freeMemory = 0;
8      if (WRPos < RDPos) {
9        freeMemory = RDPos - WRPos - 1;
10     } else {
```

```
11        freeMemory = bufferSize - WRPos;
12      }
13
14      if (requiredMemSize <= freeMemory) {   ▷ (c) fast path: acquire exclusive access
15        ProducerSync prodSyncUpdate;
16        prodSyncUpdate.EN = prodSync.EN + 1;
17        prodSyncUpdate.WR = WRPos + requiredMemSize;
18
19        SequencePtr pSeqEntry = &(seqArr[prodSync.EN & slotMask]);
20        unsigned int32 seq = pSeqEntry->SN.load(std::memory_order_acquire);
21        int32 diff = static_cast<int32>(seq - prodSync.EN);
22        if (o == diff) {
23          if (PS.compare_exchange_weak(prodSync, prodSyncUpdate, std::
              memory_order_rel)) {
24            entry.SetData(pSeqEntry, WRPos, requiredMemSize, prodSync.EN,
                buffer);
25            break;    ▷ caller is granted exclusive access to pSeqEntry and to the buffer from
                buffer[WRPos] to buffer[WRPos + requiredMemSize - 1]
26          }
27        } else if (o > diff) {
28          return false;    ▷ all sequence array entries are in use
29        } else {
30          prodSync = PS.load(std::memory_order_acquire);    ▷ the entry pSeqEntry is
              already in use, retry to acquire another entry
31        }
32      } else {   ▷ (b) slow path: check for a buffer overflow
33        if (WRPos < RDPos) {
34          return false;   ▷ the consumer is ahead of the producers, thus WR is not allowed to be
                reset
35        } else if (o == RDPos) {
36          return false;    ▷ the buffer is full, but the reader did not start
37        }
38
39         ▷ If the current thread passed the if-conditions above some other thread may have already
            acquired a new entry. If we would now succeed to reset WR to zero, the program would
            end up in undefined behavior. Of course, the other thread must have updated PS, which
            results in a failing CAS operation in line 44. and the queue invariants are correctly
            preserved.
40
41        ProducerSync prodSyncUpdate;
42        prodSyncUpdate.EN = prodSync.EN;
43        prodSyncUpdate.WR = 0;
44        if (PS.compare_exchange_weak(prodSync, prodSyncUpdate, std::
              memory_order_rel)) {
45          prodSync.WR = 0;   ▷ reset succeeded, retry to acquire a buffer region
46        }
47      }
48    }
49    return true;    ▷ acquired exclusive access to a buffer region of the desired size
50  }
51
52  bool LFBoundedMPSCQueue::enqueue(Entry& entry) {
53    entry.pSeqEntry->SN.store(entry.seqNum + 1, std::memory_order_release);
        ▷ Signals the consumer that a new entry is ready for consumption.
```

```
54    entry.ResetData();
55    return true;
56  }
```

Listing 4.8: Implementation of the consumer interface methods

```
1  bool LFBoundedMPSCQueue::dequeue(Entry& entry) {
2    const unsigned int32 DEPos = DE.load(std::memory_order_acquire);
3    SequencePtr pSeqEntry = &(seqArr[DEPos & m_slotMask]);
4    unsigned int32 seq = pSeqEntry->SN.load(std::memory_order_acquire);
5    const int32 diff = static_cast<int32>(seq - (DEPos + 1));
6    if (0 == diff) {
7      entry.SetData(pSeqEntry, DEPos, buffer);
8      DE.store(DEPos + 1, std::memory_order_release);
9      return true;     ▷ caller dequeued and acquired exclusive access to the entry
10   }
11   return false;     ▷ no entries to dequeue
12 }
13
14 bool LFBoundedMPSCQueue::release(Entry& entry) {
15   unsigned int32 RDUpdate = entry.start_index() + entry.GetSize();
16   entry.pSeqEntry->SN.store(entry.seqNum + slotCount, std::
         memory_order_release);     ▷ Signal the producers that the entry is
         free.
17   RD.store(RDUpdate, std::memory_order_release);     ▷ Signal the producers that the
         buffer region is consumed and free.
18   return true;
19 }
20
21 bool empty() const {   ▷ The empty method may be concurrently accessed by producers and
       consumers.
22   const unsigned int32 DEPos = DE.load(std::memory_order_acquire); {   ▷ Due to
         the concurrent access DE must be atomically loaded (and updated).
23   SequencePtr pSeqEntry = &(seqArr[DEPos & m_slotMask]);
24   unsigned int32 seq = pSeqEntry->SN.load(std::memory_order_acquire);
25   const int32 diff = static_cast<int32>(seq - (DEPos + 1));
26   return 0 > diff;
27 }
```

Listing 4.9: Implementation of the bulk reading interface methods

```
1  bool LFBoundedMPSCQueue::dequeue(const unsigned int32 maxBulkSize, Bulk& bulk
       ) {
2    unsigned int32 entryCount = 0;
3    unsigned int32 bulkSize = 0;
4
5    while (bulkSize < maxBulkSize) {    ▷ stop if the bulk becomes too large
6      const unsigned int32 DEPos = DE.load(std::memory_order_acquire);
7      SequencePtr pSeqEntry = &(seqArr[DEPos & m_slotMask]);
8      unsigned int32 seq = pSeqEntry->SN.load(std::memory_order_acquire);
9      int32 diff = static_cast<int32>(seq - (DEPos + 1));
10     if (0 == diff) {
11       if (pSeqEntry->m_startIndex == 0 && entryCount != 0) {
12         break;    ▷ the memory would no longer be consecutive
13       }
14       if (entryCount == 0) {
15         bulk.SetData(pSeqEntry, DEPos, m_pBuffer);    ▷ initialize the bulk
16       }
17       DE.store(DEPos + 1, std::memory_order_release);
18       bulkSize = bulkSize + pSeqEntry->size;
19       ++entryCount;
20     } else if (0 > diff) {
21       break;    ▷ no (more) entries to dequeue
22     }
23   }
24
25   bulk.bulkSize = bulkSize;
26   return bulk.IsValid();    ▷ true if line 15. was previously executed
27 }
28
29 bool LFBoundedMPSCQueue::release(Bulk& bulk) {
30   unsigned int32 nextReadPos = bulk.start_index() + bulk.GetSize();
31
32   unsigned int32 seqNum = bulk.seqNum;
33   unsigned int32 loopCondBulkSize = bulk.GetSize();
34
35   while (loopCondBulkSize != 0) {
36     SequencePtr pSeqEntry = &(seqArr[seqNum & slotMask]);
37     loopCondBulkSize -= pSeqEntry->size;
38     pSeqEntry->SN.store(seqNum + slotCount, std::memory_order_release);
           ▷ Signal the producers that the entry is free.
39     ++seqNum;
40   }
41
42   bulk.ResetData();
43   RD.store(nextReadPos, std::memory_order_release);    ▷ Signal the producers that
         the buffer region is consumed and free.
44   return true;
45 }
```

4.2.5 An alternative lock-free algorithm

During development a second lock-free queue algorithm evolved, because the previously explained algorithm may end up in a livelock in some situations. Although, there is a simple solution to prevent livelock situations, we decided to develop an algorithm that is free of livelocks. The livelock situations, the livelock prevention and the alternative algorithm are shortly explained in this section.

The second algorithm is quite similar to the first one, but additionally the so called *dynamic buffer size* value is maintained (see listing 4.10 at marker ⓐ). The dynamic buffer size *DS* is used to prevent livelocks by storing the size of the queue buffer at the point in time, when the queue buffer overflows. An overflow occurs if the size required by an entry exceeds the end of the queue's memory buffer. Such a case was already explained in section 4.2.2 and depicted in the figures 4.2, 4.3 and 4.4 at marker ⑩. The state transitions from marker ❷ to marker ❸ in the figures 4.5 and 4.6 also illustrate two examples of a buffer overflow. The last two figures directly compare the behavior of the two queue algorithms.

In the case of an overflow the previously presented algorithm, denoted as the standard algorithm or the standard version, resets the write position *WR* inside the *acquire* method, if some conditions are met (see listing 4.7 and start at marker ⓑ). Instead, the alternative algorithm only has to check if the consumer has started, if yes, then the producer can immediately try to reset the write position *WR* to zero followed by an update of the dynamic buffer size value. The *dynamic buffer size DS* is also used by the producers to reset the read position *RD* to zero, if no more entries can follow. In the standard algorithm only the consumer is allowed to modify the read position within the *release* method (see listing 4.8 at line 17.).

In short, if the queue buffer overflows, the alternative algorithm sets $DS = WR$ and $WR = 0$ and may also update $RD = 0$, if no more entries can follow. The standard algorithm instead sets $WR = 0$, if some conditions are met and *RD* is only updated by the consumer within the *release* method.

Listing 4.10: Member declaration of the alternative queue implementation

```
1  class AlternativeLFBoundedMPSCQueue {
2  public:
3    using Entry          = my::Entry;          ▷ provides access to the queue buffer
         memory of an entry
4    using Bulk           = my::Bulk;           ▷ provides access to consecutive entries
5
6    using value_type     = Entry::value_type;  ▷ type of the smalltest memory buffer
         region
7    using value_ptr      = value_type*;        ▷ pointer type to a buffer position
8
9  private:
10   using ProducerSync   = struct {
11     unsigned int32 WR;                       ▷ next free buffer position
12     unsigned int32 EN;                       ▷ next enqueue position
```

```
13    };
14
15    using ProdConsSync   = struct {
16      unsigned int32 RD;                          ▷ next read position
17      unsigned int32 DS;                          ▷ ⓐ dynamic buffer size
18    };
19
20    using Sequence       = struct {
21      std::atomic<unsigned int32> SN;             ▷ sequence number
22      unsigned int32              size;           ▷ size of the buffer region
23      unsigned int32              startIndex;     ▷ start index of the buffer region
24    };
25
26    using SequencePtr    = Sequence*;
27
28    CACHELINE_PAD
29      const unsigned int32        bufferSize;     ▷ queue buffer size
30      value_ptr                   buffer;         ▷ pointer to the buffer start position
31      const unsigned int32        slotCount;      ▷ sequence array size
32      const unsigned int32        slotMask;       ▷ avoids modulo calculations
33      SequencePtr                 seqArr;         ▷ sequence array
34    CACHELINE_PAD
35      std::atomic<ProducerSync>   PS;             ▷ producer synchronization
36    CACHELINE_PAD
37      std::atomic<unsigned int32> DE;             ▷ next dequeue position
38    CACHELINE_PAD
39      std::atomic<ProdConsSync>   PCS;            ▷ ⓑ producer-consumer synchronization
40    CACHELINE_PAD
41      my::PaddedMemory            paddedMem;      ▷ queue buffer memory, located at the
            heap and enclosed by cache line pads
42  };
```

Comparison of the two algorithms

The basic usage of the dynamic buffer size value is illustrated in figure 4.5. This figure also directly compares the behavior of the standard and the alternative algorithm. The main difference between the algorithms can be seen at the end ❺ of the illustrated scenarios, where the producer P_1 succeeds to acquire exclusive access to the green shaded buffer region, if the alternative algorithm is used. However, when using the standard algorithm the producer P_1 fails to acquire the same buffer region, although there would have been enough buffer space left in step ❺. If the standard algorithm is used, the producer P_1 must wait on the consumer to update the read position *RD*, when reading the yellow shaded entry, because of the standard algorithm's convention that only the consumer is allowed to update the read position. Exactly this convention may lead to a live lock, which is explained next.

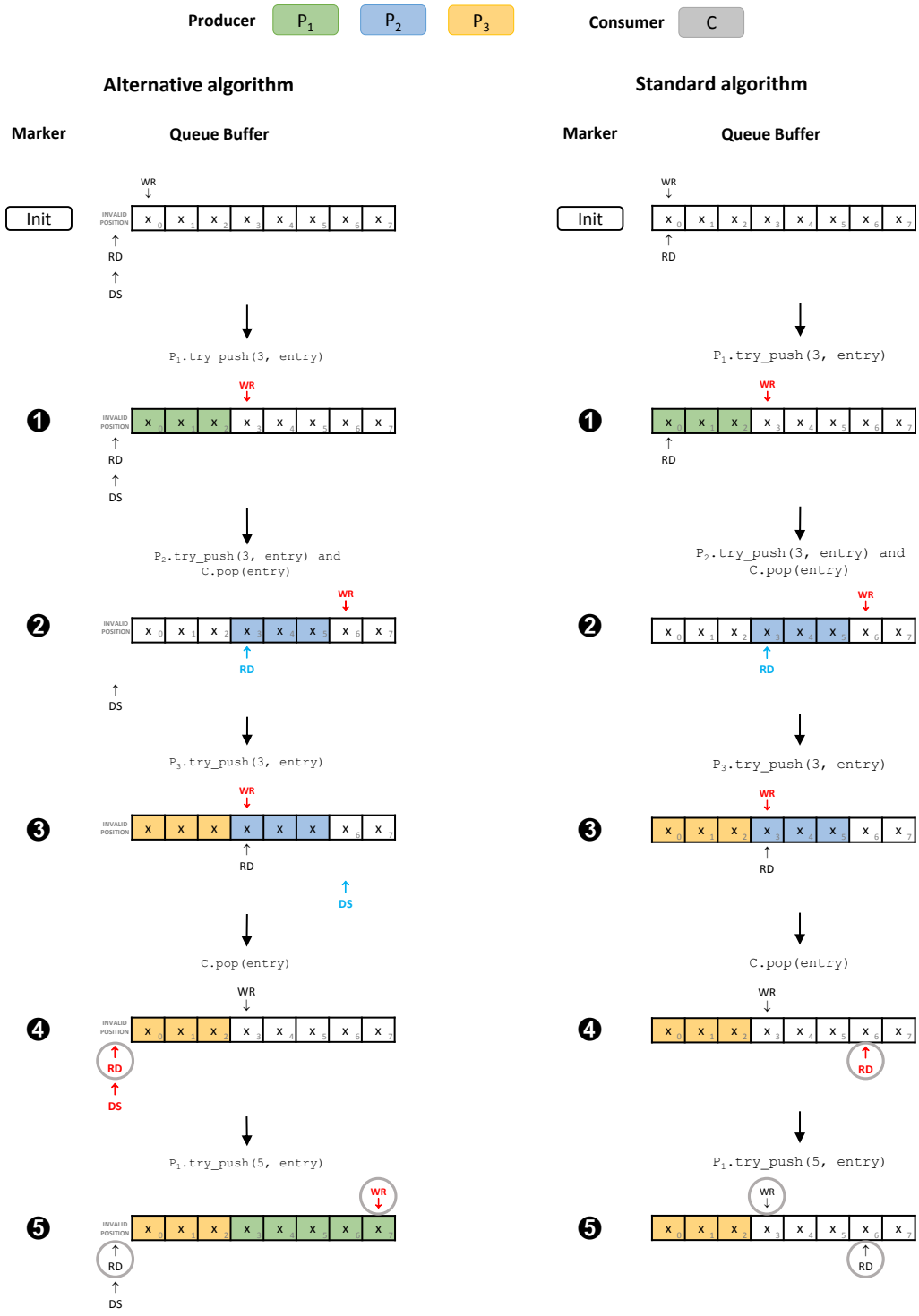

Figure 4.5: **Comparison of the standard algorithm and the alternative implementation.** Illustration of the queue state changes conducted by the standard algorithm and the alternative implementation. The alternative algorithm additionally maintains the dynamic buffer size value *DS*. The producer P_3 sets the dynamic buffer size value in the state transition from marker ❷ to ❸, which results in a successful *acquire* operation of P_1 in step ❺. In the standard algorithm instead, the producer P_1 fails to acquire an entry of size five, because *RD* was not updated by the consumer yet. P_1 must wait for the next consume operation to push its data into the queue.

Livelock and livelock prevention

The source of the livelock in the standard algorithm is the convention that the read position *RD* is only allowed to be updated by the consumer inside the *release* method. The scenario in figure 4.6 depicts a program flow where the standard algorithm ends up in a livelock, while the alternative solution can cope with the situation. Both programs again execute the same method calls, but only P_1 using the alternative algorithm succeeds to acquire an entry in step ❹. This is possible, because the alternative algorithm does not only reset the write position *WR* in the case of an overflow ❸, but it also recognizes that no entries are left for consumption and thus P_1 can also immediately reset the read position *RD* in step ❸.

A simple solution to prevent the livelock in the standard algorithm is the increase of the queue buffer size. For the depicted and all other scenarios it would be sufficient to setup a queue buffer that is two times as large as the maximum entry size. So, in the presented example the maximum entry size is seven, and if the buffer size would be fourteen, then there would never occur a livelock in the standard algorithm. In programs where multiple producers are active, it is recommended to make the queue buffer even larger to obtain a good data transfer performance. The results in chapter 6 depict not only the influence of the queue buffer size and the sequence array size on the message transfer rate and the data throughput, but also compares the performance of the standard and the alternative algorithm.

Summing up, independent of the buffer size the alternative algorithm can not end up in a livelock, while there are situations where the standard algorithm may. In the standard algorithm a livelock can be reliably prevented by acquiring a queue buffer that is twice as big as the size of the largest entry. From an algorithmic point of view the contention between producer and consumer as well as the complexity of the alternative algorithm is much higher compared to the standard version. Furthermore the results chapter shows that the standard algorithm has better scalability properties and allows higher throughput ratios. The measurement methods and the measurement results, to evaluate the performance of the queue algorithms, are presented in the following two chapters 5 and 6.

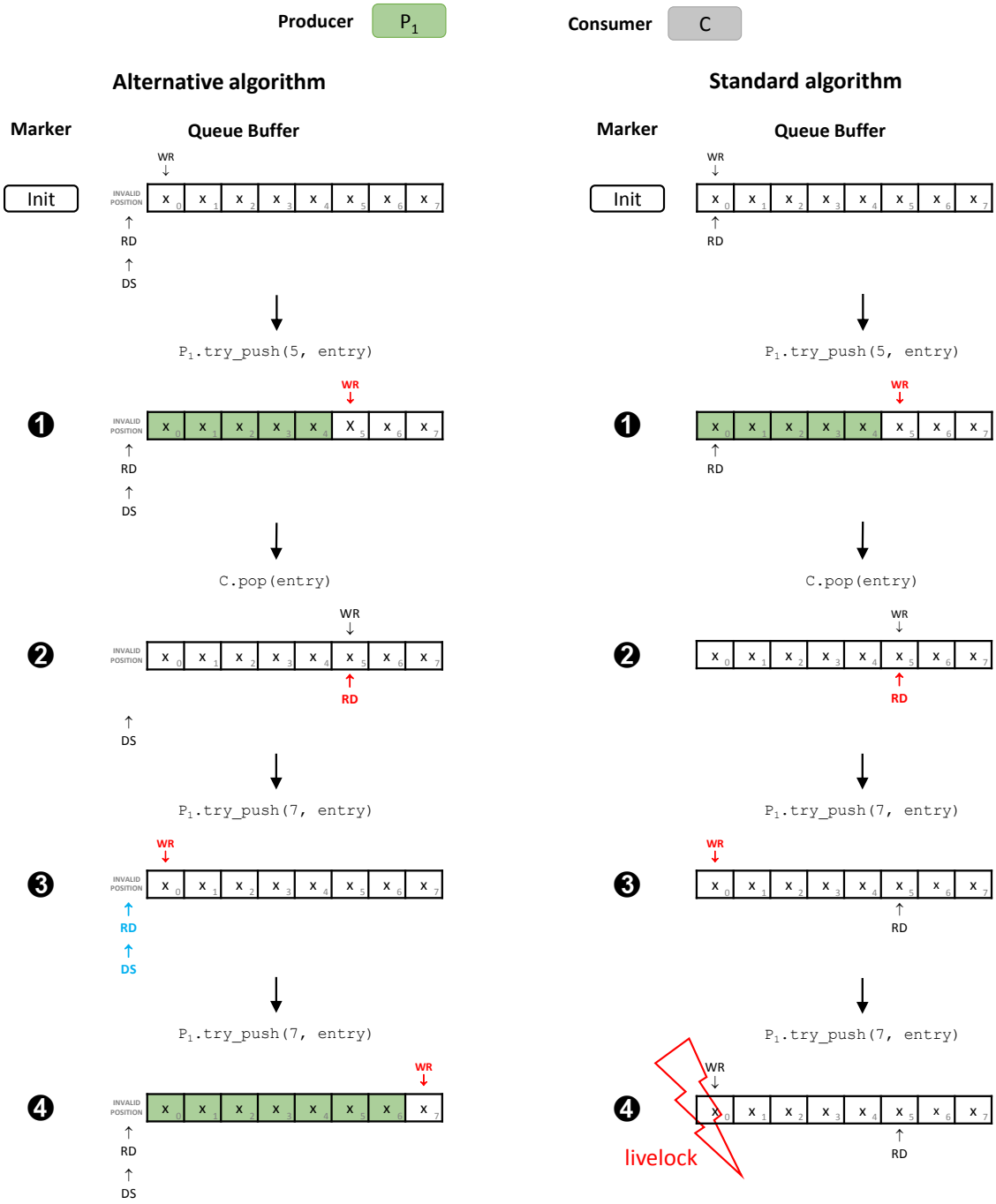

Figure 4.6: **Scenario where the standard algorithm ends up in a livelock.** Illustration of the queue state changes conducted by the standard algorithm and the alternative implementation. The alternative algorithm additionally maintains the dynamic buffer size value *DS*.
The scenario depicts an interleaving, which leads to a live lock of the standard algorithm. Although there would have been enough buffer space left, but the read position *RD* limits the buffer size to five and *RD* is only updated by the consumer within the *release* method. Of course, there is no entry ready to be dequeued, so there is no entry to be released, which results in the illustrated livelock situation.
In the alternative version of the algorithm the producer is also allowed to update the read position *RD* in the case of an overflow. Thus no livelock occurs. The livelock in the standard algorithm can be prevented by allocating a queue buffer with a size two times as large as the largest entry. To benefit from a queue as a synchronization mechanism the buffer size should be chosen to be a multiple of the entry's sizes anyhow.

4.3 Use case examples

This section depicts two examples where the presented queue is used for task synchronization and data transport. The first example in listing 4.11 shows the implementation of a class for asynchronous file IO. The asynchronous file IO class allows the efficient separation of the *runtime layer* data flow from the slow input/output operations on the *persistence layer*, as required by the data flow DF8 and DF12 in figure 3.3. Instead of executing a file output or database operation, a *runtime layer* tasks can simply write ⓐ its data into the queue. In a second step the data is moved to disk ⓒ by another task, which decouples the slow output operation from the *runtime layer*. The bulk reading operations further reduce the amount of file accesses, because several messages/entries might be combined within one bulk.

Asynchronous read-requests ⓓ are stored in the command queue ⓔ and can be issued before the data is required. This allows the requester to continue with its execution, while the other task completes the IO-operation. The result of the request can be obtained at any time via the future.

Listing 4.11: Example - Using the queue for asynchronous file IO

```
1  class AsynchronousFile : public Task {
2  private:
3    LFBoundedMPSCQueue   m_outputQueue;     ▷ ⓑ
4    LFBoundedMPSCQueue   m_commandQueue;    ▷ ⓔ
5    FILE*                m_file;            ▷ a FILE is faster than a std::fstream
6
7  public:
8    std::future<CommandResult> OpenFile(const std::string& file);
9    std::future<CommandResult> CloseFile();
10   bool IsFileOpen();
11
12   bool Print(char* pFormat, ...);    ▷ ⓐ lock-free write into queue buffer
13   bool PrintLine(char* pFormat, ...);
14
15   std::future<CommandResult> Read(const size_t length);          ▷ ⓓ read
         length bytes and store result into future
16   std::future<CommandResult> ReadLine(const size_t lineNumber);  ▷ read the line
         and store the content in the future
17   std::future<CommandResult> SeekReadLine(const size_t pos, const size_t
         length);
18
19   // ...
20
21 protected:
22   void Run() override {
23     ...
24     bIsWorkLeft = true;
25     while(Task::Continue() || bIsWorkLeft) {
26       if (!bIsWorkLeft) {
27         std::this_thread::sleep_for(20ms);
28       }
```

```
29        bIsWorkLeft = ExecuteCommands();
30        bIsWorkLeft |= WriteBufferToFile();
31
32        if (this->Task::Abort()) {
33          break;
34        }
35      }
36      ...
37    }
38
39    bool ExecuteCommands() {
40      my::CommandEntry entry;
41      if (m_commandQueue.dequeue(entry)) {
42        this->ExecuteCommand(entry);     ▷ executes the command and if required, the result
              is stored in the std::future
43        while (!m_commandQueue.release(entry)) {
44          ;
45        }
46        return true;
47      }
48      return false;
49    }
50
51    bool WriteBufferToFile() {
52      const unsigned int32 maxBulkSize = 4096;
53      LFBoundedMPSCQueue::Bulk bulk;
54      if (this->IsFileOpen()) {
55        if (m_outputQueue.dequeue(maxBulkSize, bulk)) {   ▷ ⓒ combining messages
              into bulks reduces the number of slow file accesses
56          fwrite(bulk.GetPtr(), sizeof(LFBoundedMPSCQueue::Bulk::value_type),
                bulk.GetSize(), m_file);
57          fflush(m_file);
58          while (!m_outputQueue.release(bulk)) {
59            ;
60          }
61          return true;
62        }
63      }
64      return false;
65    }
66  };
```

The second example in listing 4.12 illustrates the transfer of objects via the queue. The universal queue interface is extended ⓐ and the new methods now accept generic *ObjectEntry<...>* types. The *ObjectEntry<...>* class inherits from the *Entry* type ⓑ and provides object-like access to the encapsulated queue buffer memory via the *GetObject()* methods ⓒ ⓓ.

The placement new operator allows the constructions of objects at user defined memory locations and is thus used to create objects within the queue buffer memory ⓕ after successfully acquiring an entry ⓔ of the desired size. The destructor of the object is then explicitly called in the *release* method ⓖ to correctly perform all clean-up operations. The

whole life-cycle of the object can be implemented without requiring new allocated heap memory. This reduces the contention on the heap. Additionally the copy-overhead of passing objects between threads/tasks is eliminated by the possibility to directly work on the object that was constructed inside the queue.

The usage of the *ObjectQueue* is exemplified in listing 4.12 and starts at line 69. with the declaration of the *BaseObject* type (h). All objects that should be transfered via the *ObjectQueue* must inherit from the *BaseObject*, such as the *SpecialObject* (i). The *SpecialObject* is constructed and initialized in the functions starting at line 94. and at line 107.. The first function directly constructs the object in the queue buffer (j), because the initialization phase (k) is short. The initialization phase of the second function (L) instead, requires a long time or may even fail with an exception. Thus the function does not exclusively acquire a queue buffer region before the initialization is finished and executes its work on a local object. Afterwards the data is cloned into the queue (m).

A consumer task must use the *BaseObject* type for dequeuing. After dequeuing the (virtual) methods of the objects (l) can be called and a dynamic-cast operation (o) may be used to make decisions based on the object's type. As always, after consumption the entry must be released. The *release* method explicitly calls the destructor of the object and releases the exclusive access to the buffer region and the sequence array entry.

Listing 4.12: Example - Transferring objects via the queue

```
1  template <typename TObject, typename TQueue = LFBoundedMPSCQueue>
2  class ObjectEntry : public TQueue::Entry {   ▷ (b)
3  public:
4    using object_type = TObject;
5    using object_ptr = object_type*;
6
7    template<typename TDerivedObject>
8    TDerivedObject* GetObject() const {              ▷ (c)
9      return static_cast<TDerivedObject*>(m_pObject);
10   }
11
12   object_ptr GetObject() const {                   ▷ (d)
13     return m_pObject;
14   }
15
16 protected:
17   friend class ObjectQueue<TObject>;
18   object_ptr m_pObject;
19 };
20
21 template <typename TBaseObject, typename TQueue = LFBoundedMPSCQueue>
22 class ObjectQueue : public TQueue {   ▷ (a) interface extension to allow the transfer of
       objects
23 public:
24   typedef typename TQueue                    BaseClass_t;
25
26   typedef typename BaseClass_t::value_type   value_type;
27   typedef typename BaseClass_t::value_ptr    value_ptr;
```

```
28
29    typedef typename BaseClass_t::Entry          Entry;
30    typedef typename BaseClass_t::Bulk           Bulk;
31
32    typedef typename ObjectEntry<TBaseObject> ObjectEntryBase;
33    typedef typename ObjectEntryBase*         ObjectEntryBase_ptr;
34
35    template<typename T>
36    bool acquire(ObjectEntry<T>& entry) {
37      const unsigned int32 requiredSize = sizeof(T);
38      if (!m_queue.acquire(requiredSize, entry)) {   ▷ (e) obtain a memory region in the
            queue
39        return false;
40      }
41      value_ptr pBuffer = entry.GetPtr();
42      ::new((void*)(pBuffer)) T();   ▷ (f) placement new: constructs the object T at the
            memory position pointed to by pBuffer
43      entry.m_pObject = reinterpret_cast<T*>(pBuffer);   ▷ initializes the entry
44      return true;
45    }
46
47    bool enqueue(Entry& entry) {
48      return BaseClass_t::enqueue(entry);
49    }
50
51    bool dequeue(ObjectEntryBase& entry) {
52      if (!BaseClass_t::dequeue(entry)) {
53        return false;
54      }
55      typename Entry::value_ptr pBuffer = entry.GetPtr();
56      entry.m_pObject = reinterpret_cast<TObject*>(pBuffer);
57      return true;
58    }
59
60    inline bool release(ObjectEntryBase& entry) {
61      if (entry.m_pObject) {
62        entry.m_pObject->~TObject();   ▷ (g) explicit call of the destructor
63        entry.m_pObject = nullptr;
64      }
65      return BaseClass_t::try_release(entry);
66    }
67  };
68
69  class BaseObject {   ▷ (h)
70  protected:
71    // some data
72
73  public:
74    void CloneInto(void* pTargetBuffer) const {
75      ::new(pTargetBuffer) BaseObject(*this);
76    }
77
78    virtual void Foo() { ... }
```

```
79    virtual void Bar() { ... }
80  };
81
82  class SpecialObject : public BaseObject {   ▷ (i)
83  protected:
84    // some more data
85
86  public:
87    void CloneInto(void* pTargetBuffer) const {
88      ::new(pTargetBuffer) SpecialObject(*this);
89    }
90
91    virtual void Bar() override { ... }
92  };
93
94  bool ShortProductionTime_WriteDirectlyIntoQueue(ObjectQueue<BaseObject>&
        objectQueue) {
95    ObjectEntry<SpecialObject> mySpecialObject;
96    if (!objectQueue.acquire(mySpecialObject)) {   ▷ (j) construct an object of type
          SpecialObject directly in the queue's memory buffer)
97      return false;
98    }
99    SpecialObject* pSpecObj = mySpecialObject.GetObject();
100   DoWorkOn(pSpecObj);   ▷ (k) do some short work on the object
101   while (!objectQueue.enqueue(mySpecialObject)) {
102     // the wait strategy goes here
103   }
104   return true;
105 }
106
107 bool LongProductionTime_CopyIntoQueueAfterProduction(ObjectQueue<BaseObject>&
        objectQueue) {
108   SpecialObject myStackObject;
109   DoVeryLongWorkOn(myStackObject);   ▷ (L) do some long lasting work on the object
110
111   ObjectEntry<SpecialObject> mySpecialObject;
112   if (!objectQueue.acquire(mySpecialObject)) {
113     return false;
114   }
115   myStackObject.CloneInto(mySpecialObject.GetPtr());   ▷ (m) clone the object into the
          queue buffer
116   while (!objectQueue.enqueue(mySpecialObject)) {
117     // the wait strategy goes here
118   }
119   return true;
120 }
121
122
123 bool ConsumeObject(ObjectQueue<BaseObject>& objectQueue) {
124   ObjectEntry<BaseObject> baseObjectEntry;   ▷ (n) The BaseObject must be utilized for
          reading
125   if (!objectQueue.dequeue(baseObjectEntry)) {
126     return false;
```

```
127    }
128
129    baseObjectEntry.GetObject()->Bar();    ▷ Ⓞ Execute a (virtual) method
130
131    if (dynamic_cast<SpecialObject*>(baseObjectEntry.GetObject())) {
132        ▷ Ⓜ Do operations based on the specific object type
133    }
134
135    while (!objectQueue.try_release(baseObjectEntry)) {
136      // the wait strategy goes here
137    }
138    return true;
139 }
```

5 Test setup for the queue performance evaluation

5.1 Test system specifications

The tests were executed on an Intel Core i7-4710MQ CPU codename Haswell which implements the Intel 64-bit ISA. The Intel Core i7 had 4 cores with 2 threads per core if Intel-Hyperthreading Technology was enabled. The cores operated at a base frequency of 2,5 GHz and had a maximum core speed of 3,5 GHz. The CPU had

- a 6 MB 12-way L3 cache
- 4x 256 KB 8-way L2 cache
- 4x 32 KB 8-way L1 instruction cache and
- 4x 32 KB 8-way L1 data cache

The test machine was a laptop with an integrated 16 GB DDR3 dual channel main memory that operated at a frequency of 800 MHz. Main-memory benchmarks measured an average mixed (read/write) performance of $\sim 11,8\ GB/s$ [1] [2] [3] [4]. The test machine ran Windows 7 SP2 in the maximum performance mode, swapping was deactivated and Intel-Hyperthreading Technology was enabled. During the test execution the cores operated at their maximum speed.

The executables were generated with Visual Studio 2015 using the MSVC 19.0 compiler. The generated code was optimized for speed (compiler option /Ob2) and compiled for a x64 machine.

[1] https://www.novabench.com/ Measured main memory bandwidth: 10,3 GB/s
[2] http://www.sisoftware.co.uk/ Measured main memory bandwidth: 13 GB/s
[3] http://www.userbenchmark.com/ Measured main memory bandwidth: 12,5 GB/s
[4] http://www.passmark.com/ Measured main memory bandwidth: 11,3 GB/s

5.2 Test scenarios

5.2.1 Single threaded tests

A single threaded test evaluated the runtime of a specific function f by measuring the time that was required for executing the function f one million times. The final result depicts the average of five test runs and based on the five test runs also the standard deviation from the mean was calculated.

Synchronization primitives

The first single threaded test scenario evaluated the runtime of the synchronization cycle of several synchronization primitives. A synchronization cycle is the phase of acquiring exclusive access and releasing it again. For a mutex m a synchronization cycle is a *m.lock()* followed by a *m.unlock()*. For an atomic operation a synchronization cycle is the atomic operation itself. The following primitives were tested:

- **Mutex**: A windows mutex that is created by the *WIN-API* function *CreateMutex*. A windows mutex switches to the kernel to execute a lock or unlock operation.
- **Critical Section**: A critical section is initialized by a call to the *WIN-API* function *InitializeCriticalSection*. A critical section is similar to a mutex, but instead of entering the kernel the critical section used a CAS operation to acquire exclusive access. If a thread fails to enter a critical section, the thread calls a wait-function on an associated semaphore (behavior on multiprocessor systems).
- **CMPXCH instruction**: The CAS operation supported by the Intel 64-bit ISA with the LOCK signal set.
- **Atomic load with sequential consistency**: A sequential consistent atomic load operation, which is equivalent to a *MOV* instruction on the Intel Core architecture.
- **Atomic load with acquire semantics**: An atomic load operation that implements the acquire semantics on the given processor. This is equivalent to a *MOV* instruction on the Intel Core architecture.
- **Atomic store with sequential consistency using CMPXCH**: A sequentially consistent store operation that uses a *CMPXCH* operation inside a loop.
- **Atomic store with sequential consistency using a memory fence**: A sequentially consistent store operation that uses a *mfence* instruction to guaranty the sequential consistency. This is equivalent to a *MOV* instruction followed by a *mfence* instruction that enforces the memory synchronization.
- **Atomic store with release semantics**: A store operation that implements the release semantics. This is equivalent to a *MOV* instruction on the Intel Core architecture.

Method performance

The second single threaded test scenario evaluated the method runtime (*acquire*, *enqueue*, *dequeue* and *release*) of four different queue implementations:

1. **CritSec_Q**: Implements the alternative algorithm (maintains the dynamic buffer size) explained in section 4.2.5 and uses a single critical section to synchronize the access to the queue methods. (blocking algorithm)
2. **Mixed_Q**: Implements the alternative algorithm (maintains the dynamic buffer size) explained in section 4.2.5 and uses a single critical section to synchronize the access to the *acquire* and *release* methods, while a wait-free synchronization for the *enqueue* and *dequeue* methods is implemented. (blocking algorithm)
3. **DynSize_Q**: Implements the alternative algorithm (maintains the dynamic buffer size) explained in section 4.2.5 and uses atomic operations for the entire synchronization. (lock-free algorithm)
4. **NoDynSize_Q**: Implements the standard algorithm (no dynamic buffer size) explained in section 4.2 and uses atomic operations for the entire synchronization. (lock-free algorithm)

5.2.2 Multi threaded tests

A multi threaded test evaluated the synchronization performance of a queue implementation for different operating points. An operating point is determined by the number of active producers and consumers, the message size, the queue's buffer and sequence array size and the number of messages that have to be consumed. The data *throughput* and the data *latency* were evaluated by measuring the time required for sending a fixed number of messages in different operating points. The final results depict the average of five test runs and based on the five test runs also the standard deviation from the mean was calculated for every operating point.

The multi threaded tests integrated the *Intel Performance Counter Monitor* framework (Willhalm, Dementiev, and Fay, 2012) and several *WIN-API* calls to obtain detailed performance informations:

- Core utilization per producer- and per consumer-thread
- Data throughput and data latency per producer- and per consumer-thread
- The following parameters were measured over a full test run and depict the accumulated average of one operating point. The values were measured per core, per socket and per system. (The test machine integrates only one processor, thus the per sockt and per system measurements are identical):
  - Instructions per cycle
  - Cycles per instruction
  - Average CPU frequency
  - L3 cache misses
  - L3 cache hit ratio

- Cycles lost due to L3 cache misses
- L2 cache misses
- L2 cache hit ratio
- Cycles lost due to L2 cache misses
- System core C states
- System core C states residency
- Bytes read from memory controller
- Bytes written to memory controller
- and more ...

6 Results

This chapter reveals the performance results of the queue algorithms presented in chapter 4. Chapter 5 specifies the properties of the test platform and describes the test setup.

6.1 Single-threaded test results

The main part of this work is about synchronization. Synchronization is only required, if more than one thread participates in an algorithm. Nevertheless, the first two figures (6.1 and 6.2) present the results of single-threaded experiments. These experiments determine the best case performance of the test platform's capability to execute the synchronization instructions, because if only one thread is active no additional inter-thread synchronization is needed. The data in figure 6.1 shows the required time for a synchronization cycle. For a blocking primitive a synchronization cycle includes a lock operation followed by a unlock operation and for a non-blocking primitive a single atomic access is considered as the synchronization cycle.

The two left most bars are the results obtained from testing the blocking synchronization primitives. The other bars show the results of the non-blocking primitives. Both, a *Mutex* and a *Critical Section*, are kernel objects. A *Mutex* always performs its *lock* and *unlock* operations in the kernel, while the *Critical Section* also implements a lock-free user space synchronization mechanism and the kernel is only entered to suspend the thread, if the *Critical Section* can not be acquired. In the single-threaded experiment the locking must always succeed and the *Critical Section* never has to enter the kernel. Hence, the *Critical Section* requires less time for synchronization compared to the *Mutex*.

The performance of the non-blocking primitives heavily depends on the progress guarantee they ensure. In general, **a)** lock-free primitives are cheaper than wait-free primitives, **b)** load operations require less CPU cycles than store operations and **c)** relaxing the instructions memory ordering reduces the amount of required CPU cycles. (Sutter, 2012a) (Herlihy and Shavit, 2008). This general view is also confirmed by the measurement results shown in figure 6.1. Of course, the results obtained on other platforms could be completely different form the results presented here, because the performance of non-blocking instructions entirely depends on the hardware of the test platform.

On the given test platform the following three operations are equally fast and can be executed in a single CPU cycle:

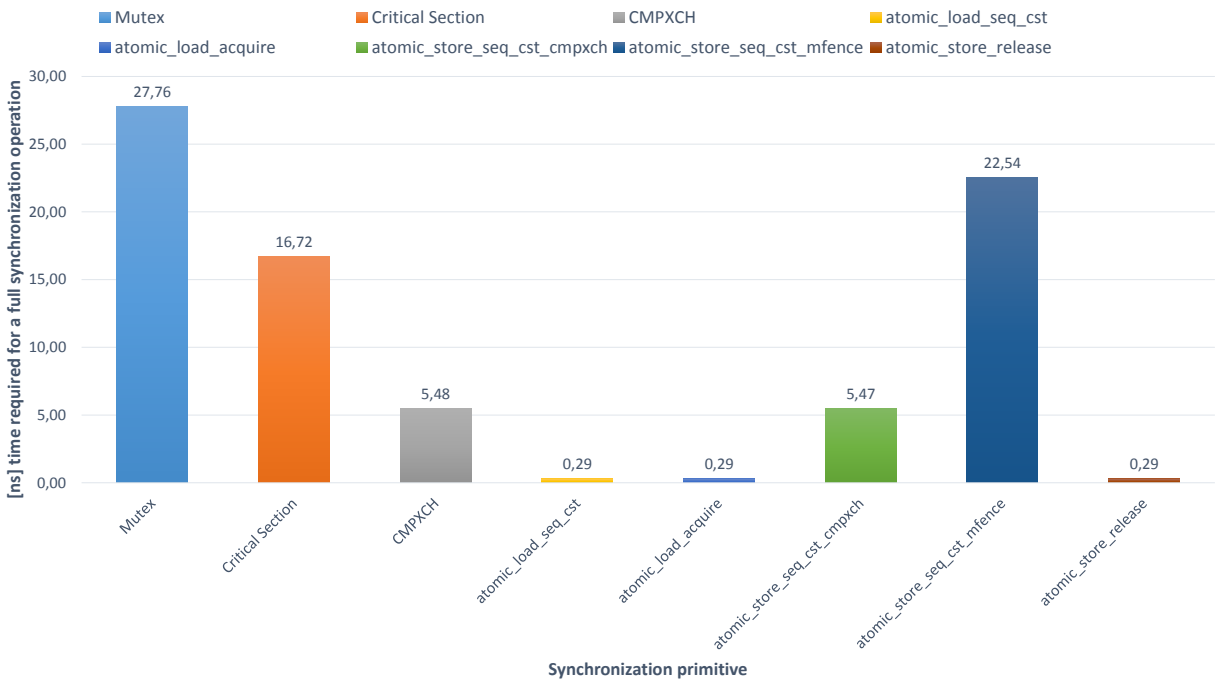

Figure 6.1: **Comparisoin of the single-threaded performance of blocking, lock-free and wait-free synchronization primitives.** The test setup is described in chapter 5. The values represent the average latency (no payload) of 5 test executions, where 1 million single-threaded synchronizations were executed. For all tests the, standard deviation from the mean is negligibly small (data not shown).

- *atomic_load_seq_cst*: sequentially consistent atomic load
- *atomic_load_acquire*: atomic load with acquire semantics
- *atomic_store_release*: atomic store with release semantics

All three operations are implemented using a simple *MOV* instruction and the memory ordering completely relies on the test platform's memory model and its cache coherence protocol. Thus only a single instruction is sufficient to load or store a value and the memory ordering semantics is automatically ensured by the hardware. An introduction about memory ordering and cache coherency protocols can be found in the related work section (see section 2), but for detailed information please consider your hardware vendor's software developer's manual.

The results in figure 6.1 reveal that store operations with a stronger memory ordering require significantly more time for synchronization than the store operations with release semantics. However, there is also a significant difference between the performance of the two sequentially consistent store operations (*atomic_store_seq_cst_cmpxch* and *atomic_store_seq_cst_mfence*). The difference between these operations results from the underlying synchronization primitives they use. The green bar shows the performance of a store operation, which uses a compare-exchange instruction (*CMPXCH*) inside a loop to achieve the sequentially consistent memory ordering. This type of implementation results in a program with a lock-free progress guaranty. The other sequentially consistent

store operation utilizes a simple *MOV* instruction followed by an *mfence* instruction. The memory fence (*mfence* instruction) is responsible for ensuring the sequential consistency by serializing all preceding load-from-memory and store-to-memory instructions, which requires significantly more time than the *CMPXCH* instruction on the test platform. However, the *MOV-mfence* implementation results in a program with a wait-free progress guaranty.

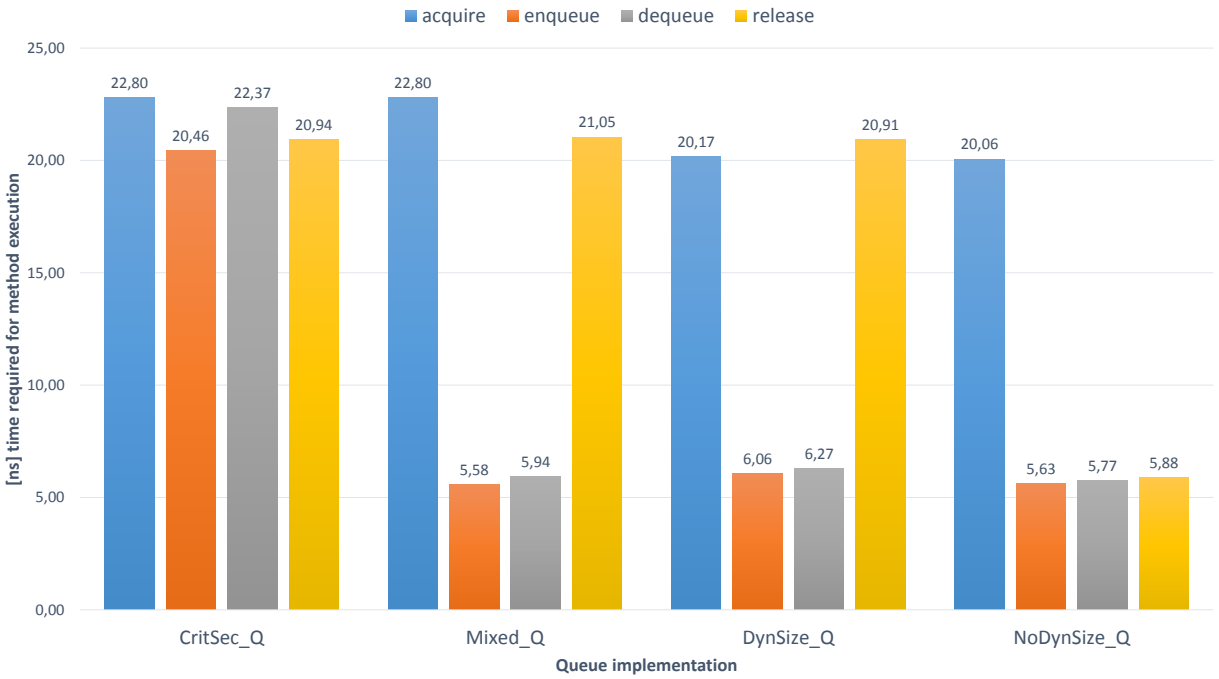

Figure 6.2: **Comparison of the single-threaded performance of blocking, mixed and lock-free queue implementations.**The test setup is described in chapter 5. The values represent the average latency (no payload) of 5 test executions of 16 million single-threaded method calls with a requested message size of 50 bytes. For all tests the standard deviation from the mean is negligibly small (data not shown).
The *CritSec_Q* uses a *critical section* for data synchronization. The *Mixed_Q* only utilizes a *critical section* within the *acquire* and the *release* methods, while the other two methods are implemented wait-free. The *DynSize_Q* and the *NoDynSize_Q* are both implemented fully lock-free. The *DynSize_Q* additionally maintains the dynamic buffer size value (see section 4.2.5) and must use a CAS operation in its *release* method.

The bars in figure 6.2 show the required execution time of each queue method for four different algorithms. The four implementations use diverse synchronization primitives and/or algorithms. The *NoDynSize_Q* queue is the only queue which implements the standard algorithm described in section 4.2, while all other queue implementations maintain the dynamic buffer size value as explained in section 4.2.5. The *CritSec_Q* queue uses a critical section to synchronize the queue access. The *Mixed_Q* queue combines the usage of a critical section and a lock-free approach. The *DynSize_Q* queue and the *NoDynSize_Q* queue are implemented entirely lock-free. The implementation of the *enqueue* and *dequeue* methods are similar for all four tested queues, except the *CritSec_Q* queue utilizes a critical section for synchronization, which explains the performance

differences compared to the other algorithms. The implementations of the *acquire* and *release* methods for the *CritSec_Q* queue and the *Mixed_Q* queue are identical, which results in an identical execution time. The fast code path of the *DynSize_Q* queue's *acquire* method differs only in a single line from the *NoDynSize_Q* queue's *acquire* method, thus the measured values also reveal almost identical execution times. The *DynSize_Q* queue algorithm may reset the read position in its *acquire* method. Hence the *DynSize_Q* queue must use a CAS operation to update the read position in its *release* method, which explains the significant better performance of the *NoDynSize_Q* queue's *release* implementation.

6.2 Multi-threaded test results

This section presents the multi-threaded test results, which show the scalability properties as well as the message and data transfer rations of the different queue algorithms. All figures (see figures 6.2 to 6.8) illustrate the performance of the four different queue implementations (*DynSize_Q*, *NoDynSize_Q*, *CritSec_Q*, *Mixed_Q*) and all figures consist of two diagrams. The diagram on the left hand side shows the experiment results with sent payload, while the diagram on the right hand side depicts the queue's performance without payload, namely the latency. The diagrams have in common that two variables are plotted on the x-axis. The bottom x-variable always illustrates the amount of active producers and the upper variable on the x-axis is changed from experiment to experiment.

6.2.1 Influence of the queue parameters

Sequence array size

Figure 6.3 illustrates the impact of the *sequence array size*. The *sequence array size* determines the maximum number of entries that can be simultaneously enqueued. The diagrams in figure 6.3 have in common that the number of messages per seconds decreases with the amount of producers, because of several reasons:

- There is only one consumer to process the entries created by multiple producers. Thus the read performance of the queue is a crucial factor to obtain a good MPSC performance.
- With the number of producers also the contention on the producer side increases. The write performance of the queue is the second factor, which heavily influences the overall efficiency of the data transfer.
- The limited number of available hardware threads further degrades the performance of the queue, if the amount of active producers increases. Especially the variation of the graph between 8 producers and 16 producers outlines the effects of oversubscription. In the case of oversubscription the graphs of the lock-free queues

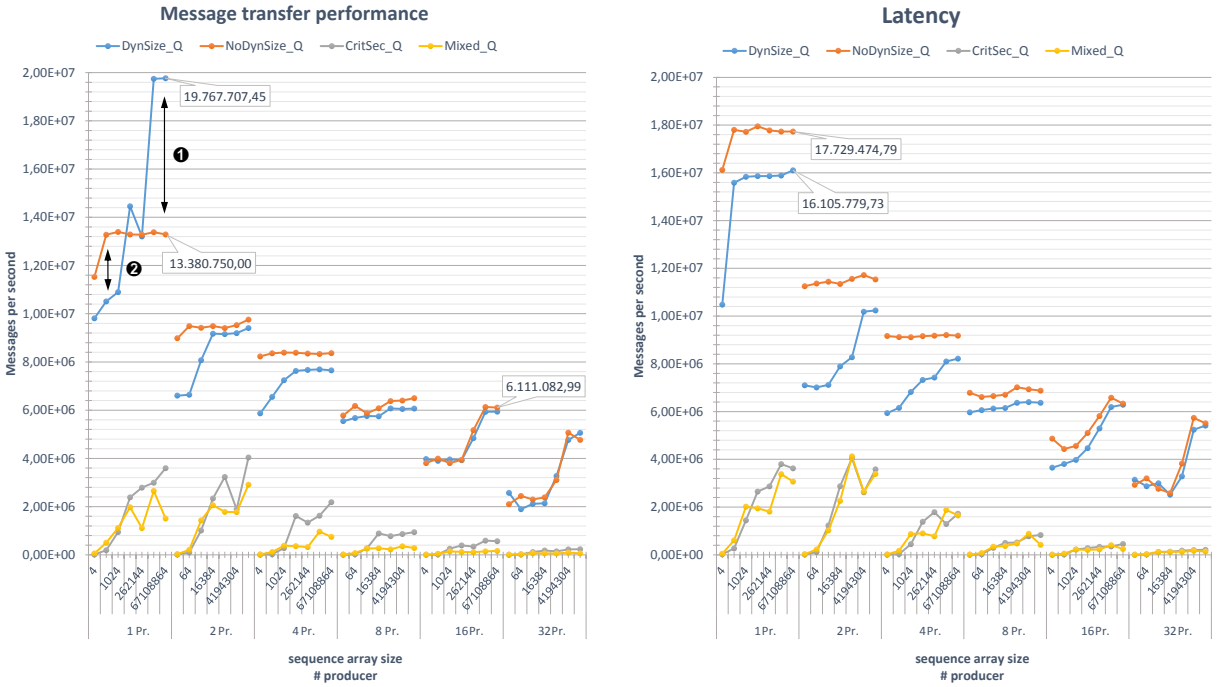

Figure 6.3: **Multi-threaded message transfer performance depending on the size of the sequence array.** The test setup is described in chapter 5. The results in the left diagram represent the average number of messages per second of 5 test runs, where 17 million messages with a payload of 64 bytes were sent per test. The diagram on the right side illustrates the results without payload. The sequence array size and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

(*DynSize_Q*, *NoDynSize_Q*) have the same characteristics, while in the experiments with less producers, the influence of the *sequence array size* on the *NoDynSize_Q* queue's message latency and message transfer rate was negligible.

An upper bound of the lock-free queues' message transfer rate is reached for a sufficiently large *sequence array size*, and remains constant even if the size of the sequence array is further increased. The *sequence array size* has very little influence on the *NoDynSize_Q*, as long as there are enough hardware resources available, which is a hint towards good scalability properties. Hence, the *NoDynSize_Q* queue may be better suited for programs where lots of producers are active.

The single-producer and single-consumer experiment emphasizes that the *DynSize_Q* is well suited for programs where mainly one producer is active. The *DynSize_Q* queue's message transfer rate is significantly better for a sufficiently large *sequence array size* ❶ compared to the transfer rate of the *NoDynSize_Q* queue. For smaller sequence arrays the *NoDynSize_Q* queue performs better ❷. The phenomenon can be also observed in the remaining SPSC results presented in this section and is labeled with the marker ❶ in the figures 6.3, 6.4, 6.5 and 6.6.

The above described behavior in the SPSC experiments emerges from the algorithmic advantages of the *DynSize_Q* queue implementation, which can take effect if there is only

little contention on the producer side. When the *DynSize_Q* algorithm is executed, the single producer can immediately reset the read position during the overflow handling. Thus the producer thread does not have to synchronize with the consumer thread in the case of an overflow, which is the limiting factor of the *NoDynSize_Q* in the SPSC scenario. The algorithmic details are described in section 4.2.5.

The influence of the *dynamic buffer size* value is only visible in the experiments with sent payload, where the contention on the queue's memory buffer has obviously a significant impact on the message transfer rate. The additional time required for writing and reading the payload data reduces the contention on the queue in general, because the time interval between the method calls, which execute the data synchronization, is increased. Of course, the reduction of the contention between producer and consumer in the SPSC scenario results in a notable speed up, if the *DynSize_Q* algorithm is used. Comparing the SPSC experiments with and without payload we see that the message transfer rate for a sufficiently large sequence array with sent payload is about $\sim 20\%$ higher than the maximum message transfer rate without payload.

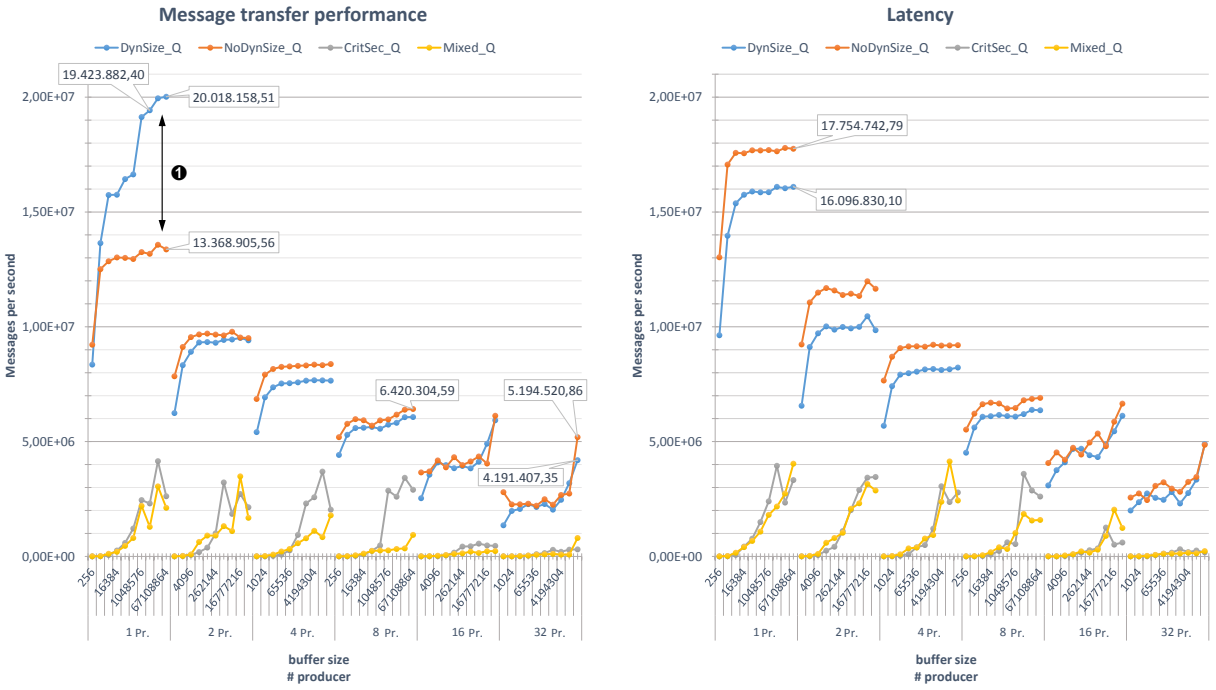

Figure 6.4: **Multi-threaded message transfer performance depending on the size of the queue buffer.** The test setup is described in chapter 5. The results in the left diagram represent the average number of messages per second of 5 test runs, where 17 million messages with a payload of 64 bytes were sent per test. The diagram on the right side illustrates the results without payload. The queue buffer size and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

Surprisingly (see section 6.1), the *CritSec_Q* queue performs equally good or even better than the *Mixed_Q* queue at most measuring points. This observation is common for all diagrams in this section and may be the consequence of a lower contention and a reduced over-subscription. If a thread is not able to acquire exclusive access to an entry,

the *CritSec_Q* queue suspends this thread. The *Mixed_Q* queue algorithm only suspend threads within the *acquire* and *release* method. Especially the consumer thread is not suspended when trying to dequeue data. Thus the consumer thread always requires its full time slice, even if no data is dequeued.

Queue buffer size

The graphs shown in figure 6.4 illustrate the influence of the *queue buffer size* on the message transfer performance. The *queue buffer size* determines the maximum number of entries that can be simultaneously enqueued depending on the sizes of each entry. Thus the observable effects are approximately equal compared to those in figure 6.3, where the *sequence array size* limits the maximum number of simultaneously enqueued entries. Both figures show, that an infinite huge buffer (sequence array and memory buffer) would not result in an infinitely high message transfer rate, because the time required for the data synchronization overhead is the limiting factor.

The single producer results in figure 6.4 at marker ❶ further outline the advantages of the *dynamic buffer size* value for small queue memory buffers. Using the *DynSize_Q* algorithm together with a 1024 byte queue buffer, already yields a higher message transfer rate than the *NoDynSize_Q* algorithm can achieve with a significantly larger buffer. Again, if multiple producers are active and the contention on the producer side of the queue is high, the *NoDynSize_Q* queue performs better.

Number of messages to consume

The graphs in figure 6.5 depict the queue's message transfer performance in different operating points by varying the number of messages the consumer must successfully read.

- In the first operating point, where the queue is mostly empty and only a couple of messages must be consumed, the producers are able to write all their messages into the queue, which can then be instantaneously consumed. So neither the producers, nor the consumer have to wait. This results in transfer rates up to 37,7 million messages per second including payload, or in up to 44,2 million messages per second without payload. Of course, if the number of producers is increased, the contention on the producer side also increases, which yields to failing CAS operations in the *acquire* method and degrades the performance. The single-threaded experiments, shown in figure 6.2, predict a message transfer performance without payload of approximately $1/(20,06ns + 5,63ns) = 38,92 mega\ Msg/s$ for the *NoDynSize_Q*, which is in accordance with the results presented in figure 6.5.
- The second operating point is reached, if about 4,2 million messages have to be consumed. At this point the graph starts to flatten and remains constant, even if the number of messages is further increases. In this operating point the queue is mainly full, which results in high contention on the producer side. The message

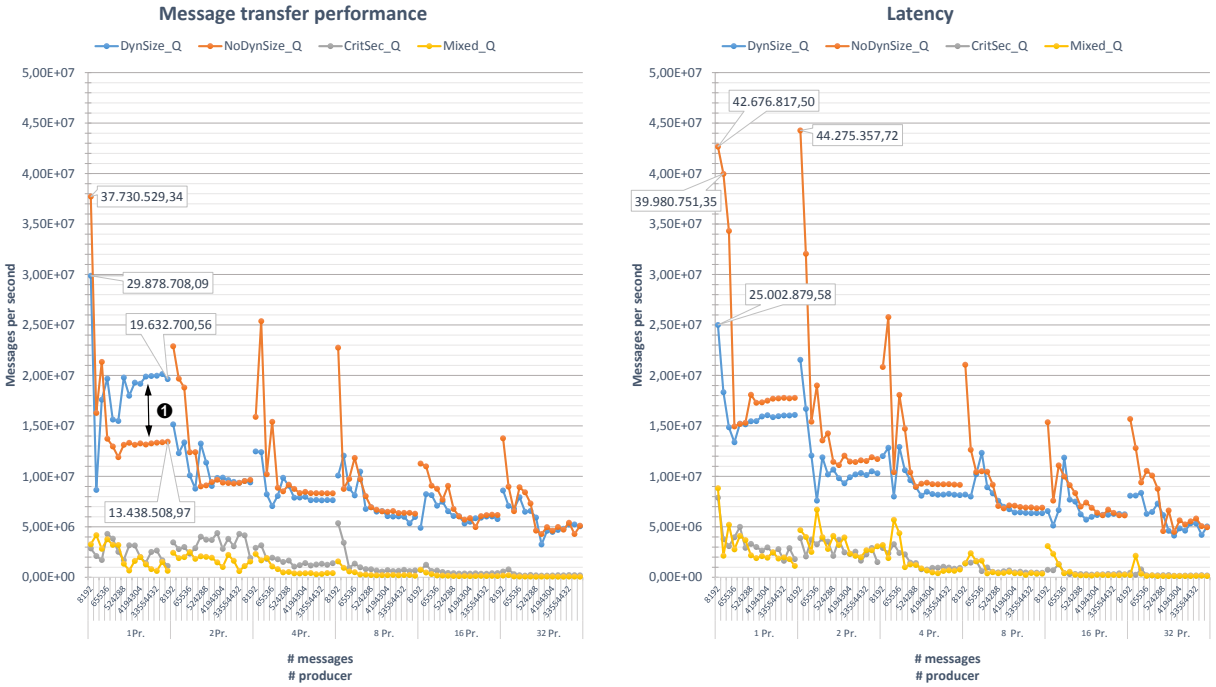

Figure 6.5: **Multi-threaded message transfer performance depending on the amount of send messages.** The test setup is described in chapter 5. The results in the left diagram represent the average number of messages per second of 5 test runs, with a payload of 64 bytes per message and a varying number of transfered messages per test. The diagram on the right side illustrates the results without payload. The number of send messages and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

transfer performance is limited by the capability of the consumer to read the entries. This operating point also determines the minimum number of required messages to obtain reproducible results in the other experiments.

6.2.2 The point of maximum performance

So far, the experiments determined the upper bounds of the message transfer rate in dependence of the *queue buffer size* and the *sequence array size*. Also the number of required messages for obtaining reproducible results was evaluated. In the next experimental setup the *queue buffer size* and the *sequence array size* are chosen to not limit the maximum performance of the algorithms. This test setup ($sequence\ array\ size = queue\ buffer\ size = 67.108.864$) will reveal the maximum achievable messages transfer performance of the developed queue algorithms in dependence of the message size.

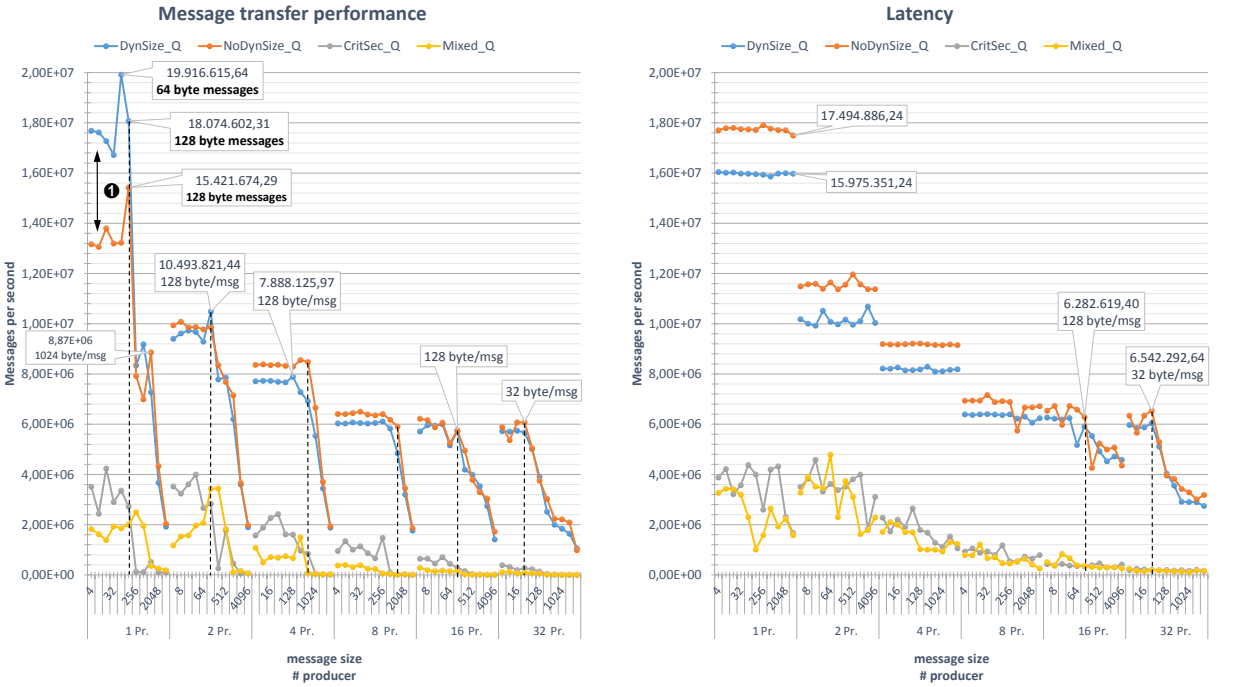

Figure 6.6: **Multi-threaded message transfer performance depending on the producers' message size.** The test setup is described in chapter 5. The results in the left diagram represent the average number of messages per second of 5 test runs, where 17 million messages with a payload of 64 bytes were sent per test. The diagram on the right side illustrates the results without payload. The message size (the payload) and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

Message size

The figures 6.6, 6.7 and 6.8 present the performance values obtained by altering the message size. In figure 6.6 we can see, that the maximum amount of messages per second can be transfered by the *DynSize_Q* queue, if the message size is 64 bytes. 64 bytes is also the size of a cache line on the test platform.

The *NoDynSize_Q* queue reaches its maximum message transfer rate at a message size of 128 bytes. If the message size of 128 bytes is exceeded, the L3 cache hit ratio starts to dramatically decrease, which is depicted in figure 6.7. The 128 bytes boarder results from the *adjacent cache-line prefetch* mechanism (Hegde, 2008) of the test platform. The mechanism automatically fetches two adjacent 64-byte cache lines, regardless of whether the additional cache line has been requested or not. This property is exploited by the producer and the consumer, because the queue memory buffer is continuously fetched into the L3 cache. Especially the consumer profits from the *adjacent cache-line prefetch* mechanism, because the linear queue buffer and the FIFO ordering, guaranty that the data from multiple producers is linearized and thus can be efficiently streamed during reading. However, the only exceptional case, where the L3 cache hit ratio increases for messages larger than 128 bytes, is the cache hit ratio measured in the SPSC scenario, when testing the *NoDynSize_Q* queue, the cache performance increases again and only

starts to degrade at a message size of 1024 bytes.

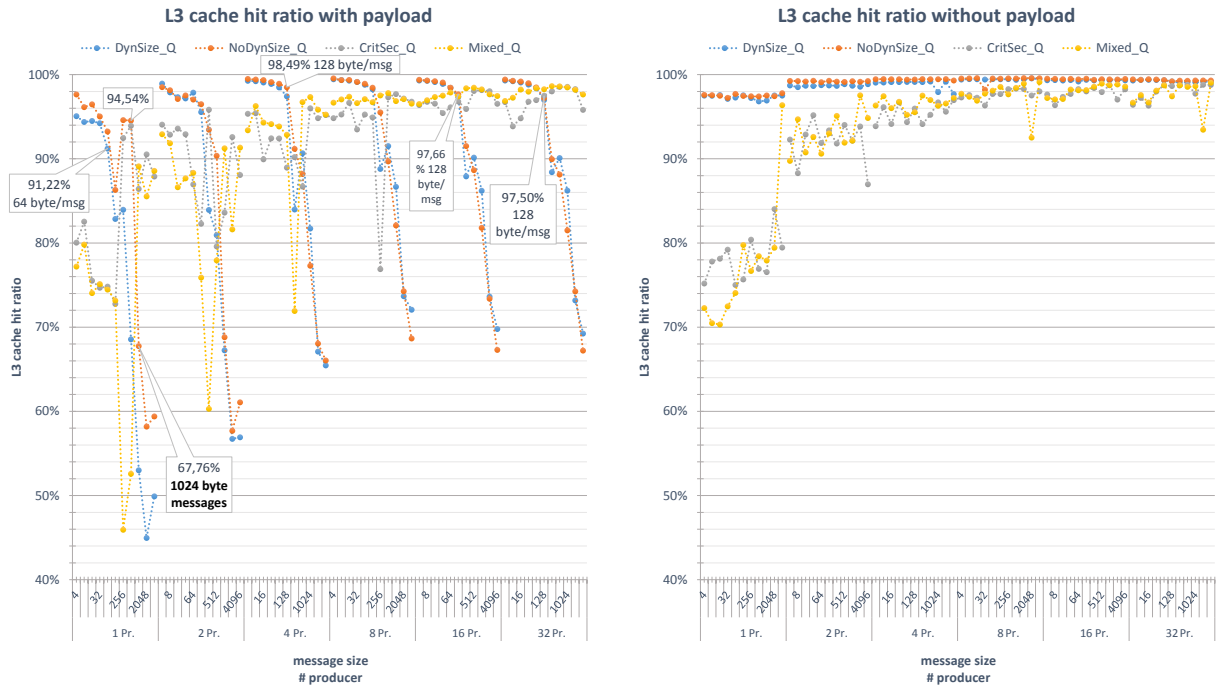

Figure 6.7: **L3 cache hit ratio depending on the producers' message size.** The test setup is the same as in figure 6.6. The results in the left diagram represent the average L3 cache hit ratio of 5 test runs, where 17 million messages with a variable payload were sent per test. The diagram on the right hand side illustrates the results without payload. The message size (the payload) and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

The latency graph shows that the increase of the message size only influences the message transfer rate if more than eight producers are active and the performance suffers due to the over subscription and the high contention on the producer side. This observation implies that the performance of the queue is not limited by the queue's buffering capability and the queue size and sequence array size were correctly chosen for this experiment.

However, in the experiments with sent payload the message transfer rate decreases already in the SPSC test case. Although the message transfer rate starts to decrease at a message size larger than 128 bytes, the data throughput still linearly increases until an upper bound is reached. The upper bound on the test platform is a throughput of approximately $9,1 GB/s$, which is achieved by the *NoDynSize_Q* algorithm in the SPSC scenario with a message size of 1024 bytes. Though the message transfer rate in this operating point is only about $8,87 mega\ Msg/s$, which is $\sim 42,5\%$ less than the *NoDynSize_Q* queue's maximum transfer rate of $15,42 mega\ Msg/s$.

The data latency (throughput without payload) is not influenced by the message size and increases linearly, even for the blocking queue algorithms and also in the case of oversubscription (see figure 6.8). This observations confirm that not the queue buffer

size, but the main memory bandwidth and the data synchronization overhead limits the maximum message transfer rate and the maximum throughput of the queue. Hence, an infinitely large queue buffer would not result in infinitely high data throughput.

Figure 6.8: **Throughput depending on the producers' message size (logarithmic scale).** The test setup is equal to the setup in figure 6.6. The results in the left diagram represent the average throughput of 5 test runs, where 17 million messages with a variable payload were sent per test. The diagram on the right hand side illustrates the results without payload. Both diagrams use a logarithmic scale on the y-axis. The message size (the payload) and the number of producers are systematically modified. For all tests the standard deviation from the mean is negligibly small (data not shown).

7 Conclusion and discussion

The test automation and the measurement tasks in the field of automotive testing are challenging problems. The high performance demands on the hardware devices and the software tools are driven forward by the technological change and the evolutions in the automotive industry. To be well-equipped for testing the forthcoming vehicle generations, we developed and evaluated a task-parallel and data flow oriented measurement and automation software. The evolved software design scales well with the proceeding increase of the CPU core count and the applied implementation techniques best exploit the features of modern processor architectures. We have yet also developed an intrusive bounded lock-free causal FIFO queue algorithm to efficiently digest the massive amount of data that is generated during vehicle-testing. The queue is universally applicable and is especially designed to increases the parallel portion of data flow oriented software applications.

7.1 The software design

The design decisions

The class diagram in figure 3.4 illustrates the final design of the software system. Here we discuss the design decisions we took based on the created domain views.

The *data flow diagram* (figure 3.3) outlined that the fast data traffic on the *runtime layer* must be decoupled from the slow IO-operations of the *persistence layer*. Otherwise the data throughput on the *runtime layer* would be restricted by waiting for the completion of file or database operations. The decoupling of the fast and slow data traffic was addressed by the introduction of **queues**.

The *data flow diagram* (figure 3.3) and *the architectural stack* (figure 3.2) revealed that software drivers are needed to establish the connection to the hardware devices. Each driver then encapsulates the protocols, the hardware specific timing behavior and the eventually slow communication channel to the hardware device. Such a design facilitates a homogeneous data flow between all *runtime layer* components and thus reduces the complexity of the software design and the implementation. Again, a **queue** is required to decouple the IO-operations.

The *domain/component model* (figure 3.1) disclosed the time dependency between the *test bench time* and the *runtime system time* and that a time-synchronization is necessary. This

observation also dictates ordering constraints on the data processing steps, because the control signals and the measurement data processing have to be accurately coordinated to obtain consistent results and a stable system.

In the final software design (figure 3.4) various threads (script/driver/user-interface thread) frequently issue variable or configuration update requests. If a variable is referenced for example by calculations or controllers, then a variable update generally triggers several other variable updates. In order to preserve a consistent system state it is not allowed to access the affected variables until the update operation is completed. The required amount of time to complete the update operation depends on the complexity of its relations and may be significant. The complexity is further increased by the coordination of the time dependences between the variable updates.

Locking the whole variable pool during the update process was not an option, because all threads that want to read from the pool would be blocked or suspended, even if the required variable is not affected by the update in progress. Thus a lot of processing time would be waisted due to task switching or busy-waiting.

The triple buffered variable pool provides the possibility to execute variable updates without blocking the reader threads. The *data processing task* executes the update steps and modifies the two shadow pools. While the update is in progress, the reader threads have transactional access to the public variable pool. A completed update operation is made visible by the wait-free modification of the *public variable pool index*.

The *data processing task* is the only thread that is allowed to change variable values or the system configuration. All other threads have to order modification requests. The requests are buffered again by applying a **queue**. This design simplifies the complexity of an update operation and further easies the software development, because

- the *data processing task* knows the complete update history. Thus no additional data synchronization or complex history management is required to correctly organize the update steps.
- the *data processing task* is the only thread that can directly modify the variable pool. Hence, no complex locking strategy is required and the wait-free update operation of the *public variable pool index* always succeeds. If multiple threads could modify the variable pool at least a lock-free strategy would be required. The lock-free index update would be very likely to fail for complex and long lasting update operations. This is expected to drastically increase the system work load or might even result in starvation.
- the *data processing task* triggers all system state changes, which easies the software debugging and software maintenance.

The linearization of all update requests via the *data processing task* yields a lot of benefits. However, the linearization is also identified as a potential bottleneck of the presented design. If the execution of update requests determines the main sequential portion of the data flow on the *runtime layer*, then this might also determine the maximum performance of the entire application. At the point of writing we believe that the maximum performance will be determined by the speed of the IO operations. However, we also believe

to our best knowledge that the presented system design, with its entirely lock-free data and access synchronization, will be sufficiently fast to cope with todays and future data traffic in automotive testing and test automation ($\sim 800\ \frac{kSamples}{s}$, for further information see section 3.1.3). Nevertheless, it might be that the sample rate and/or the sample size would drastically increase in the future and hence, it may turn out that the *data processing task* becomes a real bottleneck.

How could that bottleneck be eliminated? To address the problem we have to increase the parallel portion of the update operation by either parallelizing a *single update operation* or parallelizing *multiple update operations*. A *single update* would be hard to parallelize, because the update steps have a strict sequential order that heavily depends on the user defined configuration. Restricting the configuration flexibility to easy the update parallelization is not an option. If the parallelization of a single update operation is not possible, we have to execute *multiple updates* in parallel. However, previously we linearized the updates to reduce the synchronization overhead. The linearization of the update requests, with its associated benefits, is absolutely worthwhile and should not be sacrificed. Instead, we can use the linear stream of requests to implement speculative updating. Speculative updating sounds somewhat strange, but it is very likely that subsequent update requests do not interfere. Thus subsequent update requests can be processed in parallel and are made visible in sequential order again. Before an update is made visible, it must be checked if the update interferes with the previous update. In the case of interference a reprocessing of the request is necessary.

The synchronization of the parallel updates could be established by splitting the *data processing task* into a three staged software pipeline. The first stage linearizes and dispatches the update requests to N data processing threads of the second stage. The data processing threads speculatively execute the update request on their shadow pools and forward the result to the third and last stage. The last stage joins the processing results and runs the interference check. The N data processing threads parallelize the sequential portion that we have previously identified as the main bottleneck.

The number of threads N also scales well with the increasing core count of future processor architectures. Unfortunately each additional thread requires its own shadow variable pool, which might become a scalability issue even in 64 bit applications and machines with a huge main memory. Moreover, the pipeline approach is only suitable, if enough hardware threads are available, so that all other software threads do not lack resources. At the point of writing four to eight core processor, with two hardware threads per core, were typically installed. Waisting cores for speculative updating seemed to be a bad idea at that point.

7.2 The impact of the queue

During the evolution of the software design the efficient buffering and transfer of data between threads turned out the be a crucial performance factor. For this purpose the

intrusive bounded lock-free causal FIFO queue was developed. Here we want to discuss the impact of that queue on the final application design.

The queue interface is fine grained. Thus one more method call per write/read operation is required compared to the common push/pop interface. So why did we choose a fine grained interface? The separation of requesting and releasing the exclusive queue access into two method calls allows the caller to directly access the queue's buffer memory between these calls. Hence, the data no longer has to be copied to and from the queue's buffer memory, instead the data can be instantaneously accessed. This drastically reduces the work load of the whole application. The copy operation is just overhead and would never contribute to the intended target of the application. Additionally, copying the data into and out of the buffer is not simply unnecessary work, but it also requires memory and cache synchronization. In the given application the data traffic and hence the contention on the main memory and the caches is high anyway. Thus eliminating the copy operations reduces the contention on the main memory and the caches.

The causal FIFO ordering guaranty of the queue algorithm is required by the *data processing task* to ensure a fair treatment of all requests and to reduce the synchronization overhead due to out-of-order (with respect to time stamps) update requests. The FIFO ordering also enables the *measurement task* to write multiple consecutive measurement lines within a single file access by using the bulk operations. The *asynchronous file* implementation (see listing 4.11) exemplifies how the queue can increase the parallel portion of the program by delegating the expensive string formating to the data sources (the writers/producers). The data sink (the asynchronous file writer task) only has to copy the already formatted data from the queue buffer into the file. The examples in section 4.3 further showed that the queue is universally applicable for the side-by-side transfer of binary data and objects. The transfer of objects facilitates a homogeneous and object oriented message transfer between the software components, while the transfer of binary data allows the parallel creation of binary content (formated measurement strings) that can be efficiently transfered to files.

7.2.1 Evaluation of the queue algorithms

During the presentation of the queue test results in chapter 6 some phenomenas and observations were already explained and discussed. Here we outline the correlation between the test results.

The single threaded experiments in figure 6.2 showed that the implementation of the *NoDynSize_Q* algorithm is the fastest one among the four tested algorithms, if the execution time of all four methods is summed up. The multi-threaded experiments then confirmed that the *NoDynSize_Q* queue scales best with the increasing number of active producer. Additionally the consumer interface methods of the *NoDynSize_Q* queue are wait-free and the fastest among the four algorithms. This minimizes the synchronization workload, which is especially important for the *data processing task* that already has a high default workload. The maximum data throughput was also obtained by the

NoDynSize_Q queue. For these reasons we decided to use the *NoDynSize_Q* queue in the future implementation of the presented software design.

The *DynSize_Q* queue on the other hand also scales well with the increasing number of active producer. However, the tests revealed that the *DynSize_Q* queue's strengths are situated in scenarios with low contention and/or small queue buffers. All experiments showed that the *DynSize_Q* queue performs best, if only a single producer is active. In such cases the *DynSize_Q* queue outperforms the *NoDynSize_Q* queue.

In the multiple-producer experiments the *NoDynSize_Q* queue achieves its peak message transfer rate already with small queue buffers, while the *DynSize_Q* queue instead requires significantly larger buffers (see figure 6.3). The memory requirements of the *DynSize_Q* queue might become a scalability issue if a lot of queues are required.

	1x producer / 1x consumer						*32x producers / 1x consumer*	
	64 Byte/Msg		**128 Byte/Msg**		**1024 Byte/Msg**		**2048 Byte/Msg**	
	Msg/s	**Byte/s**	**Msg/s**	**Byte/s**	**Msg/s**	**Byte/s**	**Msg/s**	**Byte/s**
DynSize_Q	19,9 mega	1,27 giga	18,1 mega	2,32 giga	7,26 mega	7,43 giga	1,64 mega	3,35 giga
NoDynSize_Q	13,4 mega	0,86 giga	15,4 mega	1,97 giga	8,87 mega	9,08 giga	2,08 mega	4,26 giga

Figure 7.1: **Extract of operating points of the lock-free queues.** The table depicts interesting operating points of the two lock-free queue algorithms extracted from the results presented in section 6.2.2. An operating point is determined by the number of active producers and consumers, the message size, the queue's buffer and sequence array size, and the number of messages that have to be consumed. The last three parameters are identical for all operating points in the table. (*Msg/s*:Messages per second, *Byte/s*:Byte per second, *Byte/Msg*: Byte per message)

The table in figure 7.1 summarizes the main differences between the lock-free queue implementations in terms of numbers. If only one producer is active the *DynSize_Q* generally performs better than the *NoDynSize_Q*. Nevertheless, the maximum data throughput of $\sim 9,08\ GB/s$ was achieved by the *NoDynSize_Q* in the single-producer scenario. The *L3 cache hit ratio* in this operating point is only 67,76%. The graph in figure 6.7 depicts that the cache hit ratio decreases with the increase of the message size. Thus more data has to be transfered between the L3 cache and the random access memory (RAM). This data transfer bounds the maximum achievable throughput on the test platform in the given operating point. The average main memory mixed (read and write) performance of the test machine is $\sim 11,8\ GB/s$, which was evaluated out of four different main memory benchmarks (see section 5 for details). Molka et. al. obtained a main memory bandwidth of 11,7 GB/s on their test machine with an Intel Sandy Bridge processor in their survey "*Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer*" (Molka, Hackenberg, and Schoene, 2014). Thus the measured bandwidth results on the given test machine are trustworthy. The main memory benchmarks, of course, simply stream data between the CPU and the RAM. The queue also synchronizes the data access and guaranties a stronger memory ordering between the streamed data. Thus the maximum data throughput of the queue is lower than the machines maximum average throughput. However, with a data throughput of $\sim 9,08\ GB/s$ the queue achieves results near to the platform's maximum performance. Even in the case of

heavy oversubscription (32x producers / 1x consumer) a data throughput of $4,26\ GB/s$ was achieved.

7.3 Summary and future work

To sum up, we have developed an entirely lock-free task parallel and data flow oriented measurement and automation software, which uses lock-free bounded FIFO queues for the data and task synchronization. The developed queue algorithm increases the parallel program portion by minimizing the heap contention and the copy overhead during the data transfer. The queue was especially designed to reduce the cache and main memory contention, which is confirmed by the measurement results in chapter 6. The results show that the queue algorithm allows data throughput ratios close to the test platform's maximum main memory bandwidth. Additionally the queue facilitates intra-process as well as inter-process communication, even between 32-bit and 64-bit processes.

We also identified a potential scalability bottleneck in the system design and suggested speculative updating as a possible solution. However, the development of an efficient update interference recognition algorithm is relinquished to future work. The invented queue algorithm may also be improved by utilizing processors that support hardware transactional memory. Furthermore, the presented lock-free system design may be advanced to a wait-free system, which then can be reliably used in applications with real-time requirements.

Bibliography

Amdahl, Gene M. (1967). "Validity of the single processor approach to achieving large scale computing capabilitiesd." In: *AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485*. Web accessed 2016.11.23. URL: http://dl.acm.org/citation.cfm?id=1465560 (cit. on pp. 5, 7).

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt (2007). *Pattern-Oriented Software Architecture - A Pattern Language for Distributed Computing*. Vol. 4. Wiley (cit. on pp. 12, 29).

Decheva, Damian, Peter Pirkelbauer, and Bjarne Stroustrup (2010). "Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs." In: *13th IEEE Computer Society ISORC 2010 Symposium* (cit. on p. 41).

Esmaeilzadeh, Hadi et al. (2011). "Dark Silicon and the End of Multicore Scaling." In: *Proceedings of the 38th International Symposium on Computer Architecture (ISCA 11)*. Web accessed 2016.11.25. URL: ftp://ftp.cs.utexas.edu/pub/dburger/papers/ISCA11.pdf (cit. on p. 6).

Gamma, Erich et al. (1994). *Design Patterns*. Addison Wesley (cit. on p. 29).

Gustafson, John L. (1988). "Reevaluating Amdahl's law." In: *Communications of the ACM* 31.5. Web accessed 2016.11.23, pp. 532–533. URL: http://dl.acm.org/citation.cfm?id=42415 (cit. on p. 7).

Haberfellner, Reinhard (2015). *System Engineering - Grundlagen und Anwendungen*. Orell Fuessli Verlag (cit. on p. 2).

Hegde, Ravi (2008). *Optimizing Application Performance on Intel Core Microarchitecture Using Hardware-Implemented Prefetchers*. Web accessed 2016.12.02. Intel. URL: https://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers (cit. on p. 77).

Henkel, Joerg et al. (2015). "New Trends in Dark Silicon." In: *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. Web accessed 2016.11.25. URL: http://ieeexplore.ieee.org/document/7167304/?arnumber=7167304&tag=1 (cit. on p. 6).

Herlihy, Maurice and Nir Shavit (2008). *The Art of Multiprocessor Programming*. Ed. by Tiffany Gasbarrini. Elsevier Inc (cit. on pp. 19–21, 69).

Herlihy, Maurice and Jeannette Wing (1990). "Linearizability: a correctness condition for concurrent objects." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3. Web accessed 2016.11.30, pp. 463–492. URL: http://dl.acm.org/citation.cfm?id=78972 (cit. on p. 19).

Hill, Mark D. and Michael R. Marty (2008). "Amdahl's Law in the Multicore Era." In: *IEEE Computer* 41. Web accessed 2016.11.23, pp. 33–38. URL: http://research.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf (cit. on pp. 8, 9).

Iancu, Costin et al. (2010). "Oversubscription on Multicore Processors." In: *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. Web accessed 2016.11.28. URL: http://ieeexplore.ieee.org/document/5470434/ (cit. on p. 11).

Intel 64 and IA-32 Architectures Software Developer's Manual (2016). Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D. Web accessed 2016.11.29. Intel. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (cit. on pp. 16–18, 21).

Lamport, Leslie (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." In: *IEEE Transactions on Computers* C-28. Web accessed 2016.11.30. URL: http://ieeexplore.ieee.org/document/1675439/?arnumber=1675439 (cit. on p. 17).

Mattson, Timothy G., Beverly A. Sanders, and Berna L. Massingill (2004). *Patterns for Parallel Programming (Software Patterns)*. Addison Wesley (cit. on pp. 10–12).

Molka, Daniel, Daniel Hackenberg, and Robert Schoene (2014). "Main memory and cache performance of intel sandy bridge and AMD bulldozer." In: *MSPC '14 Proceedings of the workshop on Memory Systems Performance and Correctness*. Web accessed 2016.12.03. URL: http://dl.acm.org/citation.cfm?id=2618129 (cit. on p. 85).

Moore, Gordon (1965). "Cramming more components onto integrated circuits." In: *Electronics* 38.8. Web accessed 2016.11.23. URL: http://web.eng.fiu.edu/npala/eee6397ex/gordon_moore_1965_article.pdf (cit. on p. 5).

Patterson, David (2004). *Why Latency Lags Bandwidth, and What it Means to Computing*. Tech. rep. Web accessed 2016.11.29. URL: http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf (cit. on p. 16).

References (2016). Web accessed 2016.11.30. KS Engineers. URL: http://www.ksengineers.at/en/References (cit. on p. 2).

Schmidt, Douglas et al. (2000). *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Vol. 2. Wiley (cit. on p. 12).

Shi, Yuan (1996). *Reevaluating Amdahl's Law and Gustafson's Law*. Web accessed 2016.11.23. URL: http://cgvr.informatik.uni-bremen.de/teaching/mpar_literatur/Reevaluating%20Amdahl's%20Law%20and%20Gustafson's%20Law.pdf (cit. on pp. 7, 8).

Sutter, Herb (2005). *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Web accessed 2016.11.23. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (cit. on p. 5).

Sutter, Herb (2012a). *C++ and Beyond 2012: Herb Sutter - atomic<> Weapons*. Web accessed 2016.12.02. URL: https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2 (cit. on pp. 21, 69).

Sutter, Herb (2012b). *Welcome to the Jungle*. Web accessed 2016.11.23. URL: https://herbsutter.com/welcome-to-the-jungle/?blogsub=confirming#blog_subscription-3 (cit. on pp. 6, 15, 21).

Tanenbaum, Andrew S. (2001). *Modern Operating Systems*. 2nd ed. Prentice Hall International (cit. on pp. 12, 21).

Tornado System (2016). Web accessed 2015.07.30. KS Engineers. URL: http://www.ksengineers.at/en/Automotive-Testing (cit. on p. 2).

Vajda, A. (2011). *Programming Many-Core Chips*. Springer. Chap. Multi-core and Many-core Processor Architectures (cit. on pp. 6, 8, 10, 21).

Vyukov, Dmitry (2014). *Bounded MPMC queue*. Web accessed 2016.11.23. URL: http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue (cit. on p. 40).

Wang, Yan, Kenli Li, and Keqin Li (2016). "Partition Scheduling on Heterogeneous Multicore Processors for Multi-dimensional Loops Applications." In: *International Journal of Parallel Programming*. Web accessed 2016.11.25. URL: http://link.springer.com/article/10.1007/s10766-016-0445-2 (cit. on p. 6).

Willhalm, Thomas, Roman Dementiev, and Patrick Fay (2012). *Intel Performance Counter Monitor - A better way to measure CPU utilization*. Web accessed 2016.12.04. Intel. URL: https://software.intel.com/en-us/articles/intel-performance-counter-monitor (cit. on p. 67).

Williams, Anthony (2012). *C++ Concurrency in Action : Practical Multithreading*. Ed. by Cynthia Kane. Manning Publications Company (cit. on p. 21).

Wrinn, Michael (2007a). *Is the free lunch really over? Scalability in Many-core Systems: Part 1 in a Series*. Tech. rep. Web accessed 2016.11.23. Intel Corporation. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Wrinn_Free_Lunch_part_1_Scalability.pdf (cit. on p. 21).

Wrinn, Michael (2007b). *Is the Free Lunch Really Over? Scalability in Manycore Systems Part 2: Using Locks Efficiently*. Tech. rep. Web accessed 2016.11.23. Intel Corporation. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Wrinn_Scalability_Part2_EfficientLocks.rh.pdf (cit. on p. 21).