Time-Triggered Architecture for Online Diagnosis in Resource-Constrained Systems with Compressed Data Streams

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von
Dipl.-Ing. Simon Julius Meckel

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2021

Betreuer und erster Gutachter
Prof. Dr. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr. Uwe Brinkschulte
Goethe-Universität Frankfurt am Main

Vorsitzender der Promotionskommission
Prof. Dr. Günter Schröder
Universität Siegen

Tag der mündlichen Prüfung
27. September 2021

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Abstract

Coping with limited communication resources in distributed real-time systems is a major challenge nowadays. With regard to high-dependable and safety-critical systems (e.g., flight control systems and advanced driver assistance systems) the use of a time-triggered architecture is advantageous, since the periodic task executions and message exchanges according to a static schedule maximize the predictability compared to event-triggered systems. These systems efficiently realize fault tolerance by means of online and active fault diagnosis that enables redundancy-based fault-specific recovery or degradation strategies, e.g., system reconfigurations. In this way, they are able to overcome a failure or malfunction of some of their constituent components and continue to operate. As many systems are becoming more and more complex, there is an ever-growing amount of network traffic for monitoring and diagnostic purposes. In many cases, this causes the communication network to become the limiting resource, which can negatively impact the lengths of schedules, i.e., lead to longer overall service times, and reduce the maximum level of integration of different services in one system.

The use of data compression can help to reduce network traffic. However, due to the specific requirements and constraints of both time-triggered systems and diagnostic applications, some of which are contradictory, classical data compression algorithms cannot be straightforwardly applied. The difficulty lies in reconciling the needs for guaranteed data quality as well as temporal guarantees regarding information delivery. Lossless data compression does not support a certain worst-case compression ratio on short input sequences, but produces variable-length outputs. Therefore, it is difficult to guarantee the amount of information that will be encoded into a time-triggered message or, in turn, to guarantee the number of messages needed to transmit a certain amount of information. In lossy data compression, fixed-length outputs can be produced, but the quality of the reconstructed data might degrade. With respect to fault diagnosis, both incomplete or delayed data transmissions and reduced data quality lead to inaccurate fault identifications, which subsequently affects the fault recovery capabilities.

This thesis addresses the challenge of handling an increasing amount of network traffic within distributed time-triggered systems, with a particular focus on the data needed for diagnostic analyses. The thesis first presents a time-triggered architecture definition, including a compression model, that enables a systemwide coordination of data compression. Algorithms are then presented that go beyond the state of the art and realize online data compression in time-triggered systems. Specifically, the design of the novel algorithms allows the redundancy between multiple data streams to be exploited for compression by encoding the data simultaneously as tuples in a dynamic multidimensional product space. Correlated data streams are found as characteristic patterns in a low-dimensional subspace, which the compression algorithm covers by dynamically maintained dictionaries. This design also enables compressed data streams to be directly merged and split at arbitrary nodes within distributed systems, thus providing superior compression performance when the sources (or destinations) of multiple data streams are different. As key features, the presented data compression algorithms provide real-time capability through short worst-case compression and decompression times and, in addition, a worst-case compression ratio on short input sequences and a mechanism that guarantees a desired data quality with respect to the application.

The experimental evaluations show the advantages of compressed messages in resource-constrained real-time systems. Considering the trade-off between the reduction of communication times and the overhead in terms of computation times for compression, the analyses highlight the reduction potential in the response times of a time-triggered system. Compression allows to guarantee shorter deadlines and a higher integration of different services in one system. Providing a worst-case compression ratio on short input sequences, the online compression algorithms are not completely lossless, so the impact of degraded data quality is evaluated using a fault diagnosis use case and multiple test datasets that allow generalized conclusions. Especially with respect to the ability of the algorithms to compress multiple correlated data streams simultaneously, the evaluations show the scalability of the compression benefits for large systems. Savings from 40 % up to 56 % of the bits are observed in the analyzed scenarios. A comparison with other compression techniques highlights the advantages of the developed online data compression algorithms.

Zusammenfassung

Der Umgang mit begrenzten Kommunikationsressourcen in verteilten Echtzeitsystemen ist heutzutage eine große Herausforderung. Im Hinblick auf hochzuverlässige und sicherheitskritische Systeme (z. B. Flugsteuerungssysteme und moderne Fahrerassistenzsysteme) ist die Verwendung einer zeitgesteuerten Architektur von Vorteil, da durch die periodischen Task-Ausführungen und den Nachrichtenaustausch nach einem statischen Zeitplan die Vorhersagbarkeit im Vergleich zu ereignisgesteuerten Systemen maximiert ist. Fehlertoleranz wird in diesen Systemen effizient durch Online- und aktive Fehlerdiagnose realisiert, die redundanzbasierte fehlerspezifische Wiederherstellungs- oder Degradationsstrategien, z. B. Systemrekonfigurationen, ermöglicht. Somit sind sie in der Lage, einen Ausfall oder eine Fehlfunktion einiger ihrer Komponenten zu kompensieren und weiterzuarbeiten. In immer komplexer werdenden Systemen entsteht ein ständig wachsender Netzwerkverkehr für Überwachungs- und Diagnosezwecke. Dies führt in vielen Fällen dazu, dass das Kommunikationsnetzwerk zur begrenzenden Ressource wird, was sich negativ auf die zeitlichen Ablaufpläne auswirken kann, d. h. zu längeren Ausführungszeiten der Dienste insgesamt führt, sowie die maximal mögliche Integration verschiedener Dienste in einem System verringert.

Der Einsatz von Datenkompression kann zur Reduzierung des Netzwerkverkehrs beitragen. Aufgrund der spezifischen Anforderungen von zeitgesteuerten Systemen einerseits sowie Diagnoseanwendungen andererseits, die sich zum Teil widersprechen, können klassische Datenkompressionsalgorithmen nicht ohne Weiteres angewendet werden. Die Schwierigkeit liegt in der Vereinbarung der Anforderungen an eine garantierte Datenqualität mit zeitlichen Garantien bezüglich der Informationsübermittlung. Verlustfreie Datenkompression unterstützt kein bestimmtes Worst-Case-Kompressionsverhältnis auf kurzen Eingangssequenzen, sondern erzeugt Ausgaben variabler Länge. Daher ist es schwierig zu garantieren, wie viel Information in eine zeitgesteuerte Nachricht kodiert werden kann, bzw. die Anzahl an Nachrichten zu garantieren, die zur Übermittlung einer bestimmten Informationsmenge benötigt werden. Bei der verlustbehafteten Datenkompression können Ausgaben mit fester Länge erzeugt werden, jedoch kann die Qualität der rekonstruierten Daten vermindert sein. Im Hinblick auf Fehlerdiagnose führen sowohl unvollständige oder verzögerte Datenübertragungen als auch eine verminderte Datenqualität zu ungenauen Fehleridentifikationen, was sich wiederum auf die Möglichkeiten zur Fehlerbehandlung auswirkt.

Diese Arbeit befasst sich mit der Herausforderung, den zunehmenden Netzwerkverkehr in verteilten, zeitgesteuerten Systemen zu bewältigen. Ein besonderer Schwerpunkt liegt dabei auf für Diagnoseanalysen benötigten Daten. Die Arbeit stellt zunächst eine zeitgesteuerte Architekturdefinition, einschließlich eines Kompressionsmodells, vor, die eine systemweite Koordination von Datenkompression ermöglicht. Anschließend werden über den Stand der Technik hinausgehende Algorithmen vorgestellt, die eine Online-Datenkompression in zeitgesteuerten Systemen realisieren. Insbesondere ermöglicht das Design der neuartigen Algorithmen die Ausnutzung von Redundanz zwischen mehreren Datenströmen für die Kompression, indem die Daten in einem dynamischen multidimensionalen Produktraum gleichzeitig als Tupel kodiert werden. Korrelierte Datenströme werden als charakteristische Muster in einem niedrigdimensionalen Unterraum erkannt, die der Kompressionsalgorithmus durch dynamisch verwaltete Wörterbücher abdeckt. Dieses Design ermöglicht es zudem, komprimierte Datenströme an beliebigen Knoten innerhalb eines verteilten Systems direkt zusammenzuführen und aufzuteilen, was eine überlegene Kompressionsperformance ermöglicht, wenn die Quellen (bzw. Senken) mehrerer Datenströme unterschiedlich sind. Als wesentliche Merkmale bieten die vorgestellten Datenkompressionsalgorithmen Echtzeitfähigkeit durch kurze Worst-Case-Kompressions- und Dekompressionszeiten sowie zusätzlich eine Worst-Case-Kompressionsrate auf kurzen Eingangssequenzen und einen Mechanismus, der eine gewünschte Datenqualität im Hinblick auf die Anwendung garantiert.

Die experimentellen Auswertungen zeigen die Vorteile von komprimierten Nachrichten in ressourcenbeschränkten Echtzeitsystemen. Unter Berücksichtigung des Trade-offs zwischen der Reduktion von Kommunikationszeiten und dem Overhead in Form von Rechenzeiten für die Kompression verdeutlichen die Analysen das Reduktionspotenzial bei den Reaktionszeiten eines zeitgesteuerten Systems. Durch eine Datenkompression lassen sich kürzere Antwortzeiten garantieren sowie eine höhere Integration von verschiedenen Diensten in einem System realisieren. Da die Online-Kompressionsalgorithmen eine Worst-Case-Kompressionsrate auf kurzen Eingabesequenzen aufweisen, sind sie nicht vollständig verlustfrei, sodass die Auswirkungen einer verminderten Datenqualität anhand eines Diagnose-Anwendungsfalls und mehrerer Testdatensätze evaluiert werden, die verallgemeinernde Schlussfolgerungen erlauben. Insbesondere im Hinblick auf die Fähigkeit der Algorithmen, mehrere korrelierte Datenströme gleichzeitig zu komprimieren, zeigen die Auswertungen die Skalierbarkeit der Kompressionsvorteile für große Systeme. Einsparungen von 40 % bis zu 56 % der Bits werden in den analysierten Szenarien erreicht. Ein Vergleich mit anderen Kompressionsverfahren unterstreicht die Vorteile der entwickelten Online-Datenkompressionsalgorithmen.

Acknowledgements

This thesis was created during my employment at the Chair for Embedded Systems at the University of Siegen in the context of the research project *DAKODIS* funded by the German Research Foundation (DFG) under the project number 275601549.

First and foremost, I would like to express my sincere gratitude to the head of the chair and my advisor Prof. Dr. Roman Obermaisser for his continuous support and the numerous valuable discussions which greatly influenced the outcomes of my research. He opened excellent opportunities for me to participate in various prestigious research and development projects, where I got to know great personalities of the national and international industrial and scientific community.

I would also like to express my special appreciation to Prof. Dr. Markus Lohrey from the Chair for Theoretical Computer Science at the University of Siegen, who was our research partner in the DAKODIS project. I am very grateful for our collaboration, the interesting discussions and contributions that have led to valuable publications. Moreover, I thank Prof. Lohrey for his participation in the examination commission.

My sincere thanks go to Prof. Dr. Uwe Brinkschulte who agreed to be the second reviewer for my thesis. I would also like to thank Prof. Dr. Günter Schröder who agreed to chair the examination commission.

My special thank-you goes to my colleagues who contributed to the DAKODIS research project, Seungbum Jo, Damian Ludwig, Setareh Majidi, Ali Behravan and Simon Plasger. I would especially like to thank Jochen Saßmannshausen, with whom I discussed many project-related ideas and who provided me with great support during the preparation of this thesis. Special thanks go to all students who prepared their student theses with me and contributed to our research. In particular, I would like to mention Pravin Kumar Jaisawal and Jie-Uei Yang. I would also like to thank all current and former colleagues from the chair who have made office life a pleasure, especially Stefan Otterbach, Michael Schmidt, Veit Wiese, Daniel Onwuchekwa, and Hamidreza Ahmadian.

I am grateful to my family, to Vera and to all my friends who supported and motivated me throughout my studies.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AAC	Advanced Audio Coding
ADC	Analog-to-Digital-Conversion
ASCII	American Standard Code for Information Interchange
ASIL	Automotive Safety Integrity Level
ATRAC	Adaptive Transform Acoustic Coding
AVC	Advanced Video Coding
BCH	Bose-Chaudhuri-Hocquenghem
BWT	Burrows-Wheeler Transform
CBR	Constant Bit Rate
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DC	Direct Current
DCT	Discrete Cosine Transform
DFG	Deutsche Forschungsgemeinschaft
DIN	Deutsches Institut für Normung
DPCM	Differential Pulse-Code Modulation
DSC	Distributed Source Coding
ETA	Event Tree Analysis
FEC	Forward Error Correction
FIT	Failure in Time

FLAC	Free Lossless Audio Codec
FPGA	Field-Programmable Gate Array
FTA	Fault Tree Analysis
GIF	Graphics Interchange Format
GPU	Graphics Processing Unit
HEV	Hybrid Electric Vehicle
HTTP	Hypertext Transfer Protocol
ICE	Internal Combustion Engine
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
IFAC	International Federation of Automatic Control
IoT	Internet of Things
ISO	International Standards Organization
ITG	Informationstechnische Gesellschaft
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
JPEG	Joint Photographic Experts Group
LDPC	Low-Density Parity-Check
LFU	Least Frequently Used
LRU	Least Recently Used
LZW	Lempel-Ziv-Welch
MILP	Mixed-Integer Linear Programming
MOM	Message-Oriented Middleware
MPEG	Moving Picture Experts Group
MSB	Most Significant Bit
MTTF	Mean Time to Failure
MTTR	Mean Time to Repair

NTG	Nachrichtentechnische Gesellschaft
PCA	Principal Component Analysis
PGF	Progressive Graphics File
PNG	Portable Network Graphics
PPM	Prediction with Partial Match
RAID	Redundant Arrays of Independent (or Inexpensive) Disks
RPC	Remote Procedure Call
SIL	Safety Integrity Level
TDMA	Time-Division Multiple Access
TMR	Triple Modular Redundancy
TSN	Time-Sensitive Networking
VBR	Variable Bit Rate
VDE	Verein Deutscher Elektroingenieure
VDI	Verein Deutscher Ingenieure
WCCR	Worst-Case Compression Ratio
WCCT	Worst-Case Compression Time
WCDT	Worst-Case Decompression Time
WCET	Worst-Case Execution Time
WLTC	World Harmonized Light-Duty Vehicles Test Cycle
WMA	Windows Media Audio
WSN	Wireless Sensor Network

Symbols

0^n	n many 0-bits
1^n	n many 1-bits
D	Dictionary
$\mathrm{bin}_\ell(i)$	The ℓ-bit binary expansion of a value $i \in \mathbb{N} : i < 2^\ell$
d	Number of data streams
r	Length of dictionary index in bits
s	Length of value head in bits
t	Length of value tail in bits
u	Value head
v	Value tail
δ	Offset
τ	Time
$\mathbb{N}$	Natural numbers (nonnegative integers)
$\mathcal{P}(X)$	Power set of a set X
$\{0,1\}^n$	n many symbols of a given alphabet
$\lceil\cdot\rceil$	Ceil function
$\lfloor\cdot\rfloor$	Floor function

Chapter 1

Introduction

1.1 Motivation

Systems are referred to as *safety-critical systems* when a failure or malfunction potentially results in serious injury or death of people, severe damage or complete loss of (parts of) the system, or when severe environmental harm is expected. To prevent this, safety-critical systems must be fault-tolerant, i.e., they must be able to overcome a failure of some of their constituent components and continue to provide the designated service. This thesis focuses on distributed embedded real-time systems in safety-critical and high-dependability applications for which a *time-triggered system architecture* is advantageous. In time-triggered systems, the periodic task executions and message exchanges according to a static schedule maximize the predictability compared to event-triggered systems. The prior knowledge of the permitted temporal behavior enables the detection and containment of errors and failures [Obe11, p. 2].

There is a causal relationship between faults, errors, and failures [ALR+01]. From a system-level perspective, a fault can be the failure of a node in a distributed system, with the consequence that an incorrect service is provided to the environment. At the component level, in turn, a fault can be a defective electrical contact (e.g., of a microcontroller) which potentially leads to a failure in terms of the transmission of incorrect messages (e.g., value or timing failures). Fault tolerance is mainly concerned with the system level, whereas validation and maintenance primarily address the component level. In general, a fault as a cause of an error and eventually of a (total system) failure can originate from various reasons (e.g., wearout or overstress, or external influences such as voltage peaks) and exhibit different temporal behavior (e.g., permanent, transient).

Several reliability regimes are defined for safety-critical systems and classify systems according to their capabilities to handle fault situations. *Fail-safe* systems enter a safe system state if a misbehavior is detected. This prevents further danger but also stops the system from providing its desired services. For instance, a railway signaling system turns all relevant signals to red to avert the danger for train collisions until the fault situation is resolved. *Fail-operational* systems handle fault situations by strategies which keep alive at least a minimum guaranteed level of service for a certain time. For instance, if an autonomously driving vehicle experiences a fault in a relevant driver assistance system (depending on the Automotive Safety Integrity Level (ASIL) according to ISO 26262), it is supposed to at least maneuver away from the driving lane to not endanger other road users. *Fault-tolerant* systems even tolerate faults and continue to work without loss of the quality of services to be provided. An example for such a system is a control system for nuclear reactors or a flight control system aboard an aircraft.

Fault-Diagnosis-Based Fault Tolerance and Its System Implication

Redundancy is a key element in fault-tolerant system design. The four main forms of redundancy are hardware redundancy, software redundancy, information redundancy, and time redundancy [KK07, p. 3]. In short, redundancy refers to an addition (or replication) of critical resources (e.g., computations and information) with the aim of increasing the reliability of a system. With the additional resources, the system is equipped beyond what is needed for its regular operation, such that the redundancy compensates for failed parts (e.g., hardware, computations, and information) within the system, thus enabling it to continue operating instead of failing completely.

There are, naturally, implications on systems which include additional costs and space requirements with regard to hardware redundancy. Particularly in distributed systems, information and time redundancy directly increase the amount of data to be stored and transmitted. The additional network traffic can furthermore lead to additional communication delays and task execution delays.

Active redundancy strategies overcome fault situations by reconfigurations. In fault-tolerant computer systems, triple modular redundancy (TMR) with majority voting is commonly employed. By comparing the results of a triplicated process (potentially executed on different computation nodes), one faulty result can be masked without human intervention. Beyond that, a system's reliability, availability, and the tolerance against faults can be further enhanced and efficiently realized, respectively, by utilizing online and active *fault diagnosis*. Its ability to detect, identify, and potentially even predict faults in a system enables application-specific recovery or degradation strategies, e.g., system reconfigurations after a node failure including rescheduling or service degradation. Naturally, fault diagnosis with corresponding recovery strategies also utilizes redundancy, but in many situations the use of resources is more efficient, especially with a view to hardware savings [Ise06, pp. 348–349]. As an example, a triplication of components can be overcome with fault diagnosis, as active reconfiguration possibilities based on (diagnostic) information supersede simple fault masking (as used in TMR).

For this, fault diagnosis monitors the system's behavior according to its specification and if a deviation is detected, the faulty behavior is narrowed down from a first symptom to the root cause. Common fault diagnosis techniques are based on inference and classification [Ise06, p. 7]. A primary source of information for diagnosis are sensor measurements (e.g., currents, voltages, and temperatures) acquired at various locations in a distributed system, e.g., assume a scenario where multiple interacting robots perform an overall service in a smart factory [KOK14].

In the same way as the regular system service is subdivided into tasks, the analysis steps involved in fault diagnosis are task-based and executed on different nodes. The tasks include trend observations of signals, correlation and plausibility analyses, as well as classifications all of which involve features extracted from different signals and locations in the distributed system, respectively. In safety-critical systems with active fault diagnosis, system reconfigurations in the case of a component failure depend on the results of the diagnosis. In order to meet the stringent deadlines of those systems, it makes sense to execute the fault diagnosis at run-time in a time-triggered fashion. The diagnostic tasks and the required data exchange of the diagnostic inference process then compete with the tasks and messages of the system's regular service, since the diagnosis typically runs on the same (distributed) system.

In general, many such systems are becoming more complex and the computation nodes more powerful to keep up with computationally intense tasks, including diagnostic tasks. The communication capacity of the communication network often becomes the limiting resource, as data communication is typically considered to be more expensive than data processing or data storage [DFG16]. The ever-growing amount of network traffic for monitoring and diagnostic purposes is also subject-matter of many survey papers, e.g., [Meh+18]. Some of the analyzed works consider data compression, e.g., [Kol+12], however, the discussed application domains are non-safety-critical and the main objectives lie on energy savings and improved data transmission protocols, respectively (see Section 3.2).

Online Data Compression for Time-Triggered Communication

To tackle the challenge of handling an increasing amount of network traffic within distributed time-triggered systems, in particular with a focus on the data needed for fault diagnosis, this thesis presents algorithms to utilize online data compression within time-triggered systems.

The research gap arises from the special requirements and constraints of time-triggered systems and online diagnosis applications, which are partially contradictory. In particular, the impossibility of a lossless online compressor to guarantee a certain worst-case compression ratio (WCCR) on short input sequences and consequently its variable-length output symbols make it challenging to encode a guaranteed amount of information into a time-triggered message or, in turn, to guarantee the number of messages needed to communicate a certain amount of information. However, diagnosis applications in safety-critical systems require both guaranteed data quality and temporal guarantees regarding information delivery. On the other hand, lossy online compression is capable of producing fixed-length outputs, but it has the drawback that the compressed data might not be fully reconstructable and thus might be of degraded quality. This, in turn, might lead to inappropriate fault identifications and consequently thwart the essential fault recovery capabilities in safety-critical systems. For this reason, the main application area of lossy data compression is the multimedia domain, where the recipients of the data are humans, who accept a certain loss of quality or transmission delays.

This thesis was prepared in the context of the research project *DAKODIS*, funded by the German Research Foundation (DFG), which had the primary objective of establishing the ability to cope with limited communication resources while providing temporal guarantees for online diagnosis in distributed embedded systems using data compression. The thesis presents online data compression algorithms developed as part of a time-triggered architecture called the DAKODIS architecture. The designed data compression algorithms provide WCCRs, short worst-case compression times (WCCTs) and worst-case decompression times (WCDTs), as well as guaranteed data quality, and are thus able to accommodate the needs of both time-triggered systems and active diagnosis applications. In addition, the compression algorithms exploit the fact that data for fault diagnosis often have special characteristics, such as high correlations between different sensor data of related measurements [VAA04]. This leads to an improved compression performance in terms of the achievable compression ratio. As a performance measure, this is reflected in a reduced makespan (i.e., the length of a time-triggered system's schedule) and ensures that the deadlines of the applications are met.

1.2 Research Scope

Data compression is deployed in various application domains with considerable advantages when it comes to the storage and particularly to the communication of large amounts of data.

The main objective of this work is to develop and implement beyond state-of-the-art online data compression algorithms applicable for time-triggered communication in distributed systems, where the communication slots are established with respect to a global time base. Such a deterministic communication system is common for safety-critical systems to provide real-time guarantees, fault containment, and certification. Since the reliability and availability of such high-dependable and safety-critical systems is further realized with online diagnosis techniques in combination with recovery strategies, the focus of this thesis is on compressing data gathered from sensors as a foundation for determining the health status of a system.

In order to realize the goals, this thesis presents several main contributions that go beyond the state of the art:

- **The definition of a compression model** as part of the DAKODIS architecture enables a scheduling algorithm to incorporate knowledge regarding achievable WCCRs of individual as well as combined data stream compressions. This contributes to a systemwide coordination of data compression and helps to decrease the makespan, i.e., the length of the schedule.
- **Online data compression algorithms** that reconcile the requirements of both time-triggered communication and fault diagnosis applications. This is realized by lossless data compression with bounded omissions for sensor data, i.e., the data is compressed predominantly lossless, and with a low probability omissions of data values might occur. Yet not the entire information is lost in this case, since these data values are recovered with only having slightly reduced accuracy.
- **Utilizing redundancy between multiple diagnostic sensor data streams** for a superior combined compression of multiple data streams. Following the general goal of data compression to remove unwanted redundancy from input data, the newly designed algorithms continuously analyze multiple data streams for their potential to be compressed together. They are thus able to achieve a better compression ratio compared to individual data stream compressions.
- **Data compression for branching data streams in distributed time-triggered systems.** The ability of the data compression algorithms to perform real-time merging and splitting of compressed sensor data streams at arbitrary nodes of a distributed time-triggered system allows a scheduler to plan task allocations and executions as well as message injections and message paths more efficiently, e.g., with a view to time or energy savings.

1.3 Document Structure

After highlighting the motivation and the research scope of this work in this chapter, **Chapter 2** introduces the basic concepts and terminologies. It covers the three major fields of distributed real-time embedded systems, fault diagnosis, and data compression.

Chapter 3 defines the requirements for the research and reviews the state of the art. The main focus throughout this chapter lies on existing data compression techniques and their utilization possibilities in the context of time-triggered communication. The comparison of this work with related works in an overview table emphasizes the research gap.

Chapter 4 presents the system architecture including the formalization of the physical model, the logical model, the compression model, and the scheduling model. With the logical model describing an application in terms of a directed acyclic graph (DAG), the primary focus is on fault diagnosis applications. This is why the chapter takes a closer look at the specific characteristics and needs of such applications.

Chapter 5 is the core of this thesis and deals with the newly designed online data compression algorithms for individual and simultaneous compression of diagnostic sensor data streams within distributed time-triggered systems. The working principles and the specific properties of the algorithms are discussed in detail and supported with meaningful examples.

Chapter 6 evaluates the data compression algorithms in the context of a fault diagnosis use case from the automotive domain. Based on test datasets, the performance of the compression algorithms is first evaluated in terms of the achievable compression ratios with respect to the omission of data values. Next, the impact of compressed messages on time-triggered communication is investigated and finally the effect of compressed communication is evaluated with respect to applications for which the diagnostic use case is used.

Chapter 7 concludes this thesis by summarizing the achievements and providing an outlook for future work.

Chapter 2

Basic Concepts

This chapter introduces the basic concepts and terminologies and provides a foundation for the remainder of this work. It addresses the three fields of (1) embedded systems with a special focus on distributed and time-sensitive systems, (2) fault-tolerant concepts and fault diagnosis techniques applied in reliable embedded real-time systems, and (3) data compression techniques in the context of time-triggered systems.

2.1 Embedded Real-Time Systems

A computer system that is embedded in a technical context and fulfills a dedicated function is called an embedded system. As an example, a modern car incorporates many embedded devices for various tasks, some of which are highly time-sensitive, e.g., controlling a combustion engine requires precise timings for petrol injection and petrol ignition. An *embedded real-time system* unites real-time capabilities in the form of an embedded computer system. In order for such a system to show a correct behavior, its computation results must be logically correct and, in addition, produced before a predefined physical time [Sch05, p. 39; Kop11, p. 3]. The system behavior refers to the sequence of all its outputs in time. A *deadline* refers to the time instant when the system must yield the result. It is not the goal in embedded real-time computing to produce computation results as fast as possible but rather exactly within the given temporal boundaries (not too early and not too late), which are defined by the environment. The plethora of nowadays embedded real-time systems can be roughly structured according to their characteristics and capabilities.

(1) A first categorization addresses the severity of consequences in case a real-time system does not meet a deadline. In this regard, one classifies the deadlines and consequently real-time systems into *soft*, *firm*, and *hard*. This classification is driven by the application.

The output of a soft real-time system (with only soft deadlines) is still somehow useful in case of a missed deadline, since the definition of the soft deadline can be seen as a statistical criterion with a certain tolerance range [Sch05, p. 40]. Such a system shows acceptable average-response times for noncritical events. For many Internet or multimedia applications, for example, it is desirable that content is delivered on time, whereas in case of a delay, the negative impact is limited to the user experience.

The output of a real-time system after missing a firm deadline becomes inaccurate or inappropriate and is useless for the application. Besides this, there is no immediate damage and the consequences are not catastrophic. An example is an outdated transmission packet in a telecommunication system, which might lower the audio quality for a limited time and in a worst case would require the communication channel to be reestablished.

Violating the timing constraints in a hard real-time system can result in a catastrophe, such as severe harm to humans, loss of live, or vast environmental damage, among others. When a real-time computer system must meet at least one hard deadline, it is called a hard (or *safety-critical*) real-time computer system. Such systems guarantee to provide correct results within the predefined timing constraints [Kop98; Kop11, p. 3]. Systems that control safety-critical processes, e.g., chemical processes, nuclear power plants and aircraft flight control systems are examples for hard real-time systems.

(2) A second categorization addresses the system behavior in the case of a failure. Also driven by the application, one distinguishes between *fail-safe* and *fail-operational* systems. While the former have the ability to enter a safe state in order to prevent further harm or damage at the cost of stopping their designated service, the latter type of systems are designed to provide at least an acceptable quality of service despite the occurrence of a failure [Koh+16]. Examples for the two categories are a train signaling system which can transition into a safe state by setting all signaling lights to red, whereas a flight control system in an aircraft must remain operational at all times. Further details in this regard are given in Section 2.6.

(3) When it comes to the design of real-time systems, there are several possibilities how to implement the requirements and specifications. Contrary approaches are *event-triggered* and *time-triggered* real-time systems, where the latter class is often preferred to realize hard real-time systems [Obe11, p. 2]. Detailed considerations are presented in Section 2.3

(4) There are further categorizations such as *guaranteed response* versus *best effort* or *resource-adequate* versus *resource-inadequate* [Kop11, p. 16]. They all refer to different design approaches and are strongly associated with soft and hard real-time systems, respectively (see Section 2.3).

2.2 Distributed Systems

2.2.1 Characteristics of Distributed Systems

Many of the currently existing real-time computer systems are *distributed*. A so-called distributed system is built from a set of autonomous computation nodes (in the literature sometimes also referred to as computation units, processing units, or simply nodes) which are interconnected by a communication network. Typically, the independent nodes are physically distributed at different locations and the coordination of data for the tasks executed on the nodes is managed by messages via the network. There is no access to a shared memory. Each node has its own local memory and processor as illustrated in Figure 2.1. In this way, the nodes interact with each other and perform an overall service [VT02].

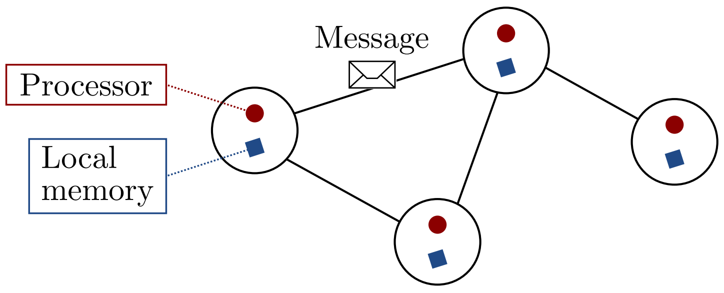


Figure 2.1: Distributed system with four nodes that cooperate with each other by exchanging messages over a communication network.

According to Van Steen and Tanenbaum [VT02], three main characteristics of distributed systems are the concurrency of components, the lack of a global clock, and an independent failure of components. However, nowadays there are special system architectures that are based on a synchronized global time, e.g., consider Time-Sensitive Networking (TSN). Moreover, even autonomous processes that run on a same physical machine, but whose communication is handled by message exchange, are sometimes referred to as distributed systems [And00, pp. 291–292]. This means that the term is used in a much broader sense these days.

In distributed computing, the goal is to encapsulate an overall service to be provided into multiple tasks that can then be scheduled to different networking computation nodes [God12]. The execution and communication of such tasks is defined by a logical model and the entire process becomes temporally and spatially decomposed. This offers great potential for parallel computation of tasks, as it is often cost-effective to favor multiple low-end computers over one high-end computer. It can also lead to a significantly increased system reliability, as a failure of an independent node can be compensated by other nodes via task redundancy or task rescheduling, see e.g., [Kri14; Erc19]. Another advantage is the scalability of a distributed system over a single computer system due to the possibility of adding new computation nodes and extending the communication network with additional routers. In general, the structure of such a system might be initially unknown and can potentially change during the execution of a distributed process, e.g., in dynamic distributed systems [Bal+07]. The usage of a distributed system may arise from the nature of an application in combination with a given physical infrastructure. For instance, if (sensor) data is generated at a certain physical location (or multiple locations) and subsequent data processing including data fusion needs to take place at another physical location, a communication network is used to transport data between multiple nodes. The message passing procedures in distributed systems can be implemented using various protocols. Examples are the hypertext transfer protocol (HTTP), remote procedure calls (RPCs), or a message-oriented middleware (MOM), see e.g., [Mag15]. Due to the message-based approach, each of the contributing nodes is only aware of a part of the overall process.

There are many examples of distributed systems, e.g., telecommunication networks including wireless sensor networks, computer networks (e.g., the Internet, intranets for industrial applications), as well as network applications like peer-to-peer networks or network file systems. Another broad field is parallel computation including cloud computing or grid computing.

The hardware that forms the physical infrastructure of a distributed system must be chosen according to the needs of the designated applications. For example, time-sensitive applications might demand specialized hardware (e.g., field-programmable gate arrays (FPGAs)) in order to comply with strict timing constraints. In this regard, distributed systems can be composed of multiple identical computation nodes and routers, respectively, such that the system is considered *homogeneous*. However, in numerous real-world scenarios, for instance in the context of the Internet of things (IoT), one finds given physical infrastructures consisting of *heterogeneous* nodes, which are getting loaded with more and more processes to be executed. As an example for such a system, consider modern industrial plants in the context of the Fourth Industrial Revolution (Industry 4.0): The hardware infrastructure is built from many different computers (e.g., embedded devices, CPUs, FPGAs, and GPUs) and the applications typically include control tasks, classification tasks, and diagnostic tasks.

For the special field of *distributed real-time systems*, the same definitions as given in Section 2.1 apply. In such a constellation, a common goal is achieved within specified time boundaries through the cooperation of the nodes [Erc19, p. 41]. For instance, real-time process control requires strict timing guarantees. Other examples are industrial control systems or aircraft control systems.

2.2.2 Resource Constraints

In many application domains, the resources of embedded systems are constrained. In this context, the term *resource* stands for different factors or circumstances, but typically addresses hardware limitations. In the last decades, the computation speed of computer systems has in general increased approximately as predicted by Moore's law. This law describes an observation formulated by Gordon Moore in 1965 [Moo+65], where the author predicted that the number of components per integrated circuit would roughly double every year for at least a decade. Consequently, he expected an increase in the computation performance. This ratio was later revised to a double every two years in 1975 [Moo+75]. Due to miniaturization and mass production, the cost of computer chips did not increase to the same extent. Despite this technical evolution, computation speed can be seen as a limitation of computer systems as there will always be applications for which the computation speed is insufficient. There are other constraints that come into play with growing processor and system speeds. An increased energy consumption causes higher costs and can lead to difficulties in the heat dissipation of the thermal loss. In consequence, these factors influence the maximum achievable system performance and play a major role in today's intentions for energy efficiency.

Considering the plethora of smart devices on the market today, many of which are battery powered (e.g., wireless sensors, wearables, or communication devices), it becomes apparent that an optimal utilization of available resources is of greatest importance. In particular, bottlenecks (e.g., weak connection/communication interfaces) between system components must be avoided, as they might impair the overall system performance. There are also space and weight constraints for many systems, especially when they are portable or mobile. Again, this might affect the available power and performance. Many existing systems were originally designed for specific purposes but have since been extended as newer applications demanded an increased performance. As a consequence, one finds disadvantageous situations where a (distributed) system has only been partly upgraded, e.g., new high-performance nodes are connected via a slow communication network.

The demanded system reliability can also be seen as a constraint, as it might entail the need for redundant system hardware, which in turn might be limited by the allowed space, weight, or power consumptions [KK07, p. 11].

In resource-constrained systems, the goal is always to make the best possible use of the available resources. In distributed real-time systems, this can only be achieved by considering the allocation of tasks to appropriate computation nodes in combination with the required message exchanges over the network. Scheduling algorithms help to solve such an optimization problem (see Section 2.3).

2.3 Time-Triggered Systems and Scheduling

A *time-triggered system* is a design choice (primarily for distributed real-time systems) where the allocation of tasks to processing nodes, the execution times of tasks and the message injection times and paths for the data exchange between the nodes via a communication network are predetermined by a schedule [Obe11, pp. 1–2]. Such a schedule is computed under consideration of a logical model (i.e., dependencies between tasks) and a physical model (i.e., resources in terms of a network topology that can execute those tasks and realize their communication), and potentially additional models (e.g., a compression model). These models provide information such as worst-case execution times (WCETs) for all tasks as well as constraints, for instance, to which processing nodes certain tasks can be allotted. Scheduling requirements furthermore describe the scheduling goal, e.g., optimization according to certain performance metrics [LM16]. In such a time-triggered system, every contributing node of the network (processing nodes and routers) is aware of the schedule. The executions of tasks at processing nodes and the subsequent injection of messages are then internally *triggered*, i.e., initiated in a periodical manner (according to the schedule) just by the progression of physical time, as illustrated in Figure 2.2. Therefore, the internal clocks of all nodes must be time-synchronized such that a global time becomes available at every node to which all events are related. This is contrary to *event-triggered* control approaches, where all processing and communication activities are initiated by (external) events, independent from any physical time. No global time base is required and to realize this kind of processing, standard interrupt mechanisms can be applied [Kop11, p. 17].

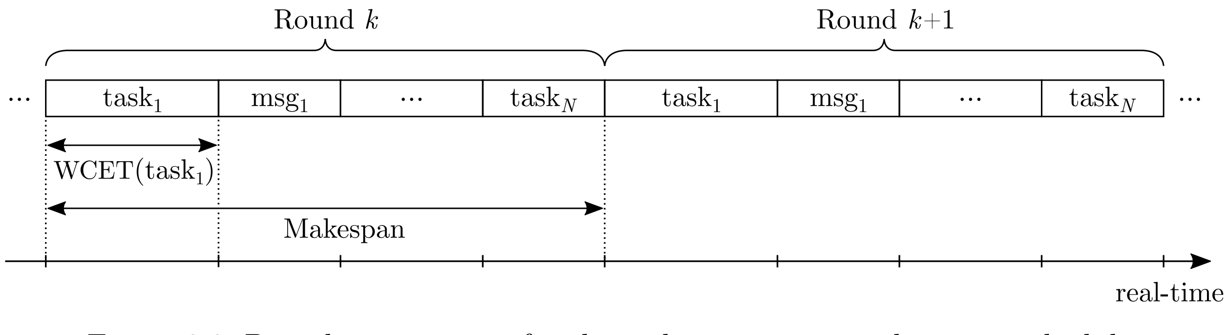


Figure 2.2: Periodic execution of tasks and messages according to a schedule.

The length of a task schedule is denoted as the *makespan*. Many scientific works and research projects pursue the goal of minimizing makespans to enable a shorter period for the execution of services (e.g., [Tab20]) or to exploit idle times during schedule executions in order to reduce the system's energy consumption, e.g., through dynamic voltage and frequency scaling [CK07].

Recalling hard and soft real-time requirements, the time-triggered approach is ideal to realize hard real-time systems due to its temporal predictability (under consideration of properly determined WCETs). Moreover, time-triggered systems are suited to accomplish *guaranteed responses* and *resource adequacy.* These attributes refer to peak load situations of fault scenarios. For hard real-time systems, a fault and load hypothesis must be specified such that the system is able to guarantee correct behavior in all situations covered by the hypotheses. Fault-tolerant scheduling particularly provides schedules according to these specifications [Kri14], possibly also alternative schedules that can be invoked in fault situations [Pop+07]. A design following the guaranteed response paradigm, again, requires the availability of adequate resources. To achieve these characteristics, careful planning and extensive analyses during the design phase of the system are of utmost importance.

[LM16] provides a general taxonomy for scheduling problems. By clustering those problems into ten groups, it provides an overview of typical up-to-date scheduling problems. Based on their analysis of numerous research papers, the authors found that solving scheduling problems for cloud computing infrastructures has recently gained considerable attention. The most frequently addressed scheduling problems are (1) scheduling of workflow tasks in dedicated computing clusters and (2) scheduling of bags of tasks in heterogeneous, widely distributed federated resources.

2.4 Dependability

In their well-respected 2004 paper [Avi+04], Avižienis et al. stated that computer (computing and communication) systems are characterized by the four fundamental properties functionality, performance, cost, and dependability. In the context of this thesis, this section takes a closer look at the last-mentioned property, the dependability. According to the dependability tree [ALR+01], the main threats to dependability are faults, errors, and failures (see Section 2.5). On the other hand, one finds the means to attain dependability, which aim at lowering the probability for the above threats through fault prevention, fault tolerance, fault removal, and fault forecasting. The four most relevant attributes of dependability, namely reliability, availability, maintainability, and safety [Mar11, p. 5] will be addressed in the following. They all relate to the quality of service that a system is able to provide during a certain time interval [Avi+04].

2.4.1 Reliability

The reliability of a system refers to probability that a system will not fail to correctly provide its designated service before a certain time. Taking as the counterpart the probability of failure, i.e., the probability that a system will fail within a given interval of time, then $\textit{reliability} = 1 - \textit{probability of failure}$. To denote this probability of failure, the *failure rate* is applied, which measures the average number of failures of devices within a given time frame. A special measure in this context is failures in time (FIT); more specifically, 1 FIT corresponds to one failure per 10^9 (i.e., 1 billion) device operating hours. For example, in the automotive domain, the norm ISO 26262 [ISO11] defines tolerable FIT rates of safety-relevant electronic systems to achieve a respective Automotive Safety Integrity Level (ASIL). According to the SN 29500 from Siemens [SN09], an example for an electronic component that achieves 1 FIT is a Universal-Diode. Assuming a constant failure rate, its reciprocal describes the average life-time of a device in hours, i.e., the mean time to failure (MTTF). The reliability is an important measure for systems where even a short outage can have a catastrophic impact [KK07, p. 5]. Systems which are required to offer a failure rate of about 1 FIT are considered to be ultrahigh reliable systems [Kop11, p. 11] or, alternatively, ultrahigh dependable systems [SWH94].

2.4.2 Maintainability

The maintainability refers to the time and effort that needs to be spent to repair or maintain a system and bring it back to correct functioning after a (noncritical) failure. According to Kopetz [Kop11, p. 12], there is a fundamental conflict between a high maintainability and a high reliability, as easily maintainable systems are often composed of field replaceable units and serviceable interfaces, which in turn come with much higher physical failure rates compared with fully integrated systems or components. As a quantifiable measure of maintainability, the mean time to repair (MTTR) refers to the average time it takes to reestablish a system's delivery of correct service.

2.4.3 Availability

The availability is seen as the fraction of time in which a system is ready to provide its designated service correctly with respect to a certain observation time interval. If one expects constant failure and repair rates, the (long-term) availability is derived as $\mathit{MTTF}/(\mathit{MTTF} + \mathit{MTTR})$ [Kop11, p. 12]. This ratio means that a system is available before a failure occurs and is unavailable until this failure is repaired. A system that is demanded to offer high availability may therefore show a long mean time to failure or a short mean time to repair. The availability is an important quality of service measure for systems where downtimes are undesirable and potentially costly, but not immediately vital [KK07, p. 5]. For instance, an online store should be highly available, since downtimes can annoy customers and decrease sales, yet short system outages can be tolerated.

2.4.4 Safety

Failures can be classified into critical and noncritical failure modes. Recall Section 2.1, in safety-critical real-time systems the harm or damage in case of a failure can be extremely high. The safety can thus be seen as the reliability regarding critical failure modes. Depending on the safety integrity level (SIL), systems in these application domains (e.g., an airplane control system) must be certified by an independent certification agency, which performs a critical review of the complete system design and can attest the system a so-called ultrahigh reliability with a failure rate (regarding critical failures) of as low as about $10^{-9}\,\mathit{failures}/h$ [Kop11, p. 11].

2.5 Faults, Errors, and Failures

2.5.1 Terminologies

The terms *faults* and *failures* are widely used in various technological fields, especially when reliability and fault tolerance play a role. However, the terminology and the definitions throughout the literature are not always consistent. As early as 1988, Omdahl et al. published the RAM (reliability, availability, maintainability) dictionary [Omd88] as an approach to standardize the terminology. Later works, e.g., from the International Federation of Automatic Control (IFAC), the German DIN (e.g., DIN 40041), the VDI/VDE (e.g., VDI 4001/2, NTG 3004 (now ITG)), and other works (e.g., [Lap92; IB97]) made efforts to come to commonly accepted definitions. Still, often depending on the application domain, different authors use slightly different terminology and definitions. In particular the term *error* has a strong relation to the field of computer systems and refers, for instance, to corrupted h-state data in memory [Kop11, pp. 138–139]. For mechanical systems (e.g., an axle) such an error state is inappropriate [Ise06].

In Section 2.4, the threats to dependability were identified as faults, errors, and failures. They stand in a causal relationship as depicted in Figure 2.3. A fault is a cause of an error and eventually of a failure or a malfunction. Faults and errors are states within a system, whereas malfunctions and failures are events [Ise06, pp. 20–21; Kop11, pp. 136–141].

$$\begin{tikzcd}
\cdots \arrow[r] & \text{Fault} \arrow[r, "\text{Activation}"] & \text{Error} \arrow[r, "\text{Propagation}"] & \text{Failure} \arrow[r, "\text{Causation}"] & \text{Fault} \arrow[r] & \cdots
\end{tikzcd}$$

Figure 2.3: The fundamental chain of dependability threats [ALR+01].

The following sections define the terminology used throughout this thesis, respecting the relevant literature in the field. Detailed remarks are provided for each of the terms.

2.5.2 Faults

According to Isermann [Ise06, p. 20], a fault is an unpermitted deviation of at least one characteristic property of a system from the acceptable, usual, standard condition. Such an unpermitted deviation refers to a discrepancy between the fault value and the usual value respecting a certain tolerance range. The IEC 61508 furthermore states that a fault may ultimately reduce the capability of a functional unit, and consequently of the system, to fulfill the intended service [IEC10]. A fault might not directly have an impact on the system. Especially in mechanical systems, faults such as small cracks (e.g., in a gear wheel) might stay undiscovered for a long time. Numerous faults originate from different reasons, in different situations, with a different temporal behavior and affect different domains of a system. Figure 2.4 shows five elementary fault classes and provides the terminology to properly describe faults.

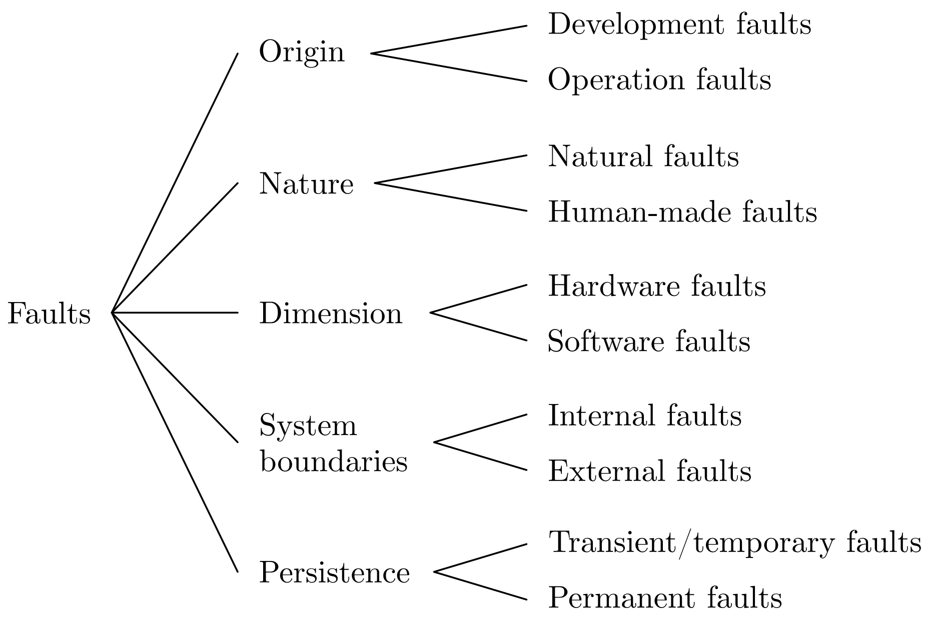


Figure 2.4: Fault classes.

Firstly, a fault can be classified regarding its origin. This viewpoint refers to the phase in which a fault is created or occurs and separates development faults from operation faults. Faults that originate during the development phase of a system, but also those which originate during maintenance situations, belong to this class. If, on the other hand, a fault occurs during the runtime of a system, it is identified as an operational fault.

In the second category, the nature of a fault is considered and a distinction is made between natural faults and human-made faults. The former category indicates that a fault is caused by natural phenomena without any human involvement. The human-made faults include all actions of humans and, in addition, the absence of actions when actions should have been performed (omission). One further characterizes the underlying objective of the involved human, which can be malicious or nonmalicious. Characterizing a fault to be human-made and malicious implies a human's intent to deliberately effectuate the system behavior in his or her favor (e.g., access confidential information) or to negatively influence the functioning (e.g., disrupt the service). Human-made, nonmalicious faults might still result from deliberate harmful decisions, however, the intention is of different kind, e.g., a certain trade-off decision might turn into a fault at a later time. Of course, such faults might also arise without any special awareness. Then it is typically considered to be an accidental fault. However, if it turns out that a human-made, nonmalicious fault is introduced without awareness, but results from an inadequate qualification of humans (e.g., operators, managers), it might be more appropriately described as an incompetence fault [Avi+04].

The third category identifies a fault in terms of its dimension, meaning that it either affects the system's hardware or the software.

The fourth category describes a fault related to the system boundaries. A fault can be internal and originate from hardware (internal physical fault) or software (internal design fault). In contrast, external faults arise from either environmental circumstances that effect the hardware (e.g., lightning stroke) or the software (e.g., inappropriate inputs).

Lastly, faults exhibit a certain temporal behavior (persistence). Permanent faults are assumed to remain continuously present after their initial occurrence. The presence of transient faults is bounded in time, their presence might be of periodic or sporadic appearance.

2.5.3 Errors

An error (in a computer system) is an unintended state and it is the consequence of a fault [Kop11, p. 136]. It might be inactive for a while until, if activated, it leads to a failure. On the other hand, it can also remain inactive without any impact on the system. The state *error*, unlike electrical systems, cannot be appropriately applied to many mechanical systems [Ise06]. In various examples in the literature, a fault is directly referred to be the cause of a failure or malfunction, see e.g., [Ise11, p. 20].

2.5.4 Failures

A malfunction as well as a failure is characterized by a deviation of a currently provided service of a system from the intended service, where a malfunction refers to a temporary interruption and a failure means a permanent interruption of a system's ability to provide its intended service [Ise06, pp. 20–21]. Malfunctions and failures result from one or more faults. Figure 2.5 provides the terminology to characterize a failure according to four elementary aspects.

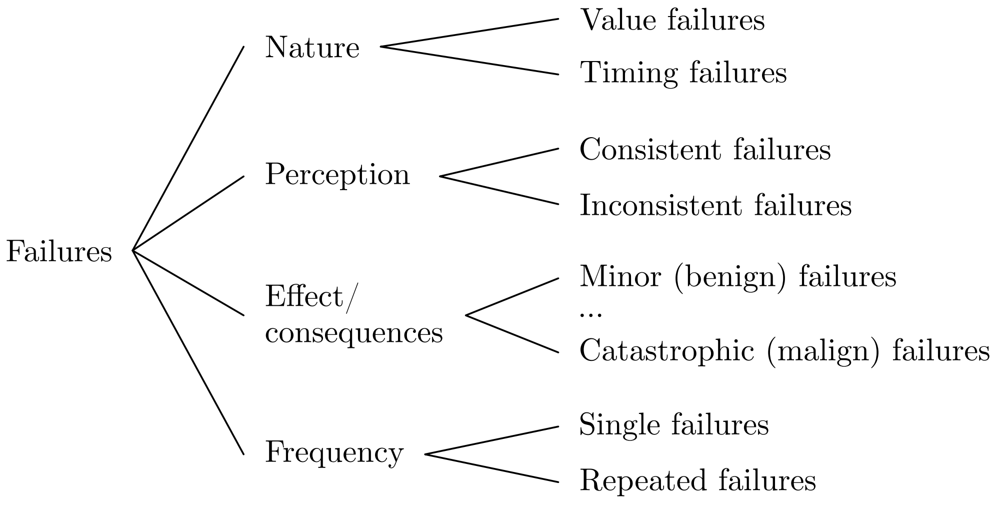


Figure 2.5: Failure classes.

Firstly, a failure has a certain nature, i.e., it occurs in a specific domain, where a distinction is made between value (i.e., content) and timing failures. A value failure (of a component) refers to an incorrect output value at the component interface. Failures of a system in the temporal domain only exist if an intended temporal behavior is specified (recall Section 2.3). They refer to a violation of the defined temporal behavior when a value is produced outside the intended real-time interval. One can further distinguish early temporal failures and late temporal failures [Kop11, pp. 139–141]. Omission failures are a special case of late failures. Of course, there might also be cases where both the content and the timing is incorrect. Then one describes the failure either as halt failure (i.e., the service to be provided is halted), or as erratic failure, where the system is performing some wrong activities (e.g., babbling) [Avi+04].

In the second category, perception, a failure is designated as either consistent or inconsistent. This category is considered if a system has more than two users (a user in this context can be, for example, an interconnected system or a node in a distributed system). If a failure is consistent, all users observe the same failing behavior. An inconsistent failure manifests itself by exhibiting a different failing behavior for different users. Other terms for an inconsistent failure in the literature are two-faced failure or Byzantine failure, see e.g., [Dri+04].

The third category addresses the effect or the consequences of failures. The two most extreme cases are minor (benign) failures and catastrophic (malign) failures. Other severity levels in between the aforementioned two can be defined according to criteria such as tolerable outage duration, possibility of human lives being endangered, or consequences in case of loss or disclosure of (confidential) data [Avi+04]. Systems whose failure potentially has disastrous effects on the environment, leads to severe harm to humans, or causes significant economic loss are referred to as safety-critical systems (recall Section 2.1).

The fourth category describes a failure in terms of its occurrence within a certain time interval. If it occurs only once, it is denoted as single failure. A further distinction can be made as to whether a system is able to resume operation after a single failure, in which case the failure is called a transient failure. Otherwise, the failure is called permanent and requires repair of the system. In the given time interval, a failure might occur frequently; then it is called a repeated failure.

In the literature, one might find slightly different failure mode classifications or the level of detail of the categorization might vary, however, the four failure domains presented in this work conform to most of the available literature, see e.g., [Ise06; Vac06].

2.6 Fault-Tolerant Systems

2.6.1 Fault-Tolerant System Design

An important step towards a high system reliability and safety is an improvement of the design and a high quality of all involved components. However, in many (safety-critical) application domains, the system reliability is demanded to be higher than the reliabilities of the constituent subsystems or components.

Fault tolerance refers to the capability of a system to tolerate faults by exploiting redundancy, thus preventing an overall system failure. Recalling the causal relationship between faults, errors, and failures (Section 2.5.1) and taking a distributed system as an example, the failure of a node (which might result in providing an incorrect service to the environment) is considered to be a fault from the system-level perspective. Fault tolerance demands redundant system hardware (e.g., redundant network nodes, redundant mechanical structures) and also involves information redundancy, software redundancy, as well as time redundancy [KK07, p. 3]. Figure 2.6 illustrates the four major forms of redundancy in computer science.

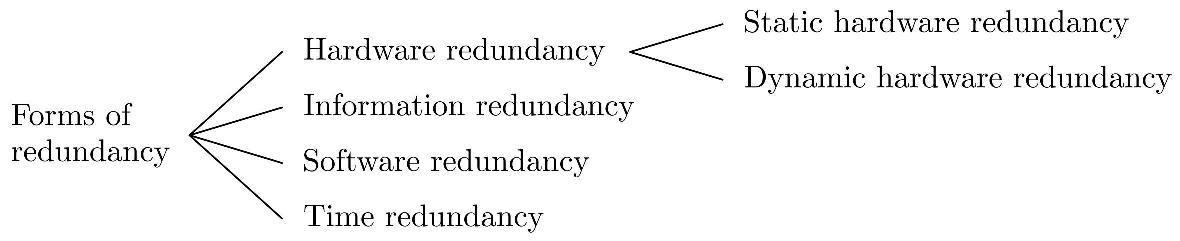


Figure 2.6: Forms of redundancy.

2.6.1.1 Hardware Redundancy

The most obvious way to deal with hardware faults is to include extra hardware into the system design to detect and possibly compensate the effects of failed hardware. Besides this, information or time redundancy can also help to tolerate hardware faults. With respect to extra hardware, one basically categorizes fault-tolerant system realizations into those with *static redundancy* and those with *dynamic redundancy* [Ise06; Alf17].

Static Redundancy

Triple modular redundancy (TMR) in electronic systems is an example for static hardware redundancy, where the outputs of three redundant modules (e.g., computer hardware), which work on the same inputs, are compared via a voter so that one faulty module can be overcome by this constellation without any efforts on fault detection. To be consistent with the common literature, the term *module* is used here and in the following to refer to what is tripled in a TMR configuration. It is used as a general term and covers computer hardware and computations in the same way as electric or mechanic components.

In general, in an n-modular redundant system with an odd number n of redundant modules, $F = (n-1)/2$ faults can be tolerated since a majority voter overwrites faulty outputs. On the other hand, if, for example, $n = 3$ and two modules were faulty, a majority voter would pass through the faulty signal.

Figure 2.7 exemplarily shows an n-modular redundancy structure with a majority voter. Obviously, every extra module produces additional cost, consumes energy, needs space and

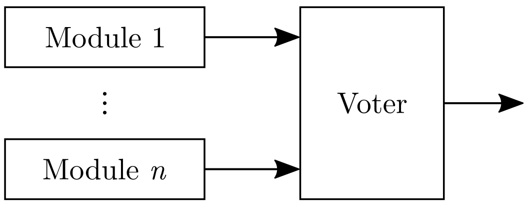


Figure 2.7: An n-modular redundancy structure with majority voting.

makes a system more complex, which is a disadvantage of the strategy of massive static redundancy [KK07, p. 11].

Sift-out modular redundancy, originally introduced in [SM78] for digital systems, is a variation of the above static configuration. Instead of a majority voter, it uses a comparator, a disagreement detector, and a collector, and is thus capable of switching out faulty modules. With those no longer contributing to the system output, the constellation is able to achieve a higher fault tolerance of $F = n - 2$.

Dynamic Redundancy

Dynamic hardware redundancy comes along with fault detection capabilities. In this way, a minimal configuration utilizes two redundant modules (recall, the minimum number of modules in static redundancy is three), whereof one is the active module and the other one is the back-up module. The active module is continuously monitored for faults and the fault detection in combination with a reconfiguration capability decides when to switch to the back-up module.

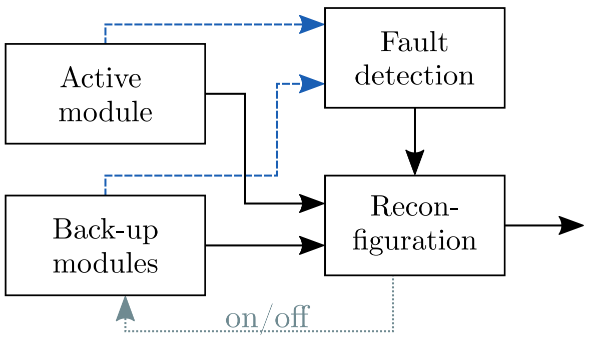


Figure 2.8: Dynamic hardware redundancy.

Figure 2.8 illustrates dynamic hardware redundancy. The dashed blue lines indicate diagnostic information gathered from the modules. Utilizing the dynamic redundancy strategy, the back-up module (potentially more than one) can be either kept in cold standby, i.e., turned off until needed, or in hot standby, i.e., running without using the output. A hot standby offers a fast transition between modules, but is associated with higher operational costs and wearout of the back-up modules (vice versa for a cold standby configuration). For the cold standby configuration, the dotted gray line in Figure 2.8 indicates communication with the back-up modules.

Hybrid Redundancy

It is also possible to combine static and dynamic hardware redundancy to hybrid redundant schemes. Assume an n-modular redundant system which has k additional back-up modules. By comparing the output of the majority voter with each output of the n primary modules, the faulty one is determined and the reconfiguration unit replaces the faulty module with a back-up module. According to Koren [KK07, pp. 25–26], such a configuration significantly increases the system reliability.

2.6.1.2 Information Redundancy

When information in terms of bits is stored, processed or transmitted, it is prone to errors. In order not to lose information due to such errors (e.g., bit flips), error detection and correction methods are widely applied. In this regard, coding refers to the strategy of adding extra bits to the data which allow to verify data correctness and, in some cases, even to restore corrupted data bits.

A common method for error detection is a cyclic redundancy check, where data is separated into chunks from which a check value is processed and attached to the chunk. Repeating the calculation of the check value before usage of the data reveals information about the data correctness. A class of cyclic error-correcting codes is named after their inventors Bose-Chaudhuri-Hocquenghem (BCH). As a key feature, these codes allow to precisely determine the desired number of correctable bits. In general, such an information redundancy strategy is called forward error correction (FEC) and a main field of application is data transmission over noisy communication channels, e.g., mobile communications or satellite communications.

For example, a BCH code with the parameters $m = 4$ ($m \in \mathbb{N} : m \geq 3$) and $t = 2$ ($t \in \mathbb{N} : t < 2^{m-1}$) produces code words consisting of $n = 2^m - 1 = 15$ bits with $k = n - mt = 7$ bits holding the information and $mt = 8$ bits serving as redundancy (i.e., parity-check digits). The minimum Hamming distance of the code is $2t + 1$. It can detect $2t$ single bit errors and correct up to t single bit errors. Considering burst errors, up to k successive bit errors can be detected [RL09, pp. 111–113]. Other examples of FEC codes are low-density parity-check (LDPC) codes and turbo codes [Mac05; RL09].

Contrary approaches to obtain information redundancy include retransmissions of data in case of data loss, which can be a suitable way to overcome transient link failures. Such retransmissions are also a kind of time redundancy (see Section 2.6.1.4), but to deal with permanent link failures, information redundancy is better established by redundant communication links.

Resilient disk systems based on a data storage virtualization technology called RAID (redundant arrays of independent (or inexpensive) disks) are another way to realize information redundancy through coding. Depending on the required level of redundancy, data is distributed among multiple disk drives according to different schemes.

One example of the use of data redundancy in distributed systems is data replication, which aims to consistently provide identical data at different nodes to be able to compensate for node failures.

Data corruption does not only occur during transmission and storage, but also when it is processed. In the context of efficient data processing, algorithm-based fault tolerance was designed to protect the computations of large data arrays (e.g., matrix multiplication, matrix inversion, and Fast Fourier Transform) [KK07, pp. 99–101].

2.6.1.3 Software Redundancy

Software redundancy has the goal to prevent software failures. A well-known approach to achieve this is multiversion programming [CA78]. Roughly speaking, since large pieces of software are expected to contain faults (here also referred to as bugs), the strategy is to independently create multiple versions of software from an unambiguous specification in different programming languages by different teams which do not interact with each other. In this way, the risk of these multiple software versions failing on the same inputs or at the same time is reduced.

Software redundancy can be employed similarly to hardware redundancy, i.e., concurrently, or on demand (in combination with fault detection capabilities), and it is also possible to realize hybrid software redundancy schemes [Sto96].

2.6.1.4 Time Redundancy

A common strategy for establishing time redundancy is to repeat the execution of a program or an operation on the same data multiple times on the same computer hardware. In this way, short-lived, transient hardware faults can be overcome. Time redundancy imposes few demands on additional hardware or software, but introduces temporal overhead, especially if such multiple results need to be compared before further processing is possible. To reduce this overhead, error detection mechanisms can help to limit reexecutions to certain instructions of an overall program (recall algorithm-based fault tolerance).

In distributed systems, a retransmission of lost or corrupted messages (e.g., after a link failure) can also be viewed as a form of time redundancy.

2.6.2 Degradation Steps

Safety-critical systems or systems where a high availability is demanded are usually designed to be fault-tolerant. The required degree of fault tolerance as well as the appropriate types of redundancy are defined by the application and might vary for different parts of a system. The term *degradation* refers to the intended behavior of systems after a fault or failure appeared and is closely linked to fault-tolerant design decisions. In many application domains, a system is required to be either *fail-operational* or *fail-safe*. According to Isermann [Ise06, p. 352], the two degradation steps are characterized as follows:

- Fail-operational systems tolerate one failure, i.e., they are able to provide the desired service even after one failure. In safety-critical applications, a system must be fail-operational if no safe state exists which a system could enter after a component failure.
- Fail-safe systems have a safe state that is entered after a failure occurs. The desired system service is then interrupted such that further negative impacts on the system and the environment are prevented.

For instance, a flight control system aboard an aircraft must be fail-operational as there is no safe state for an aircraft to enter. For such live-critical systems, all four introduced forms of redundancy must be employed. In this domain, triplex redundant hardware realizations are essential for many involved components, so that up to two failures can be tolerated. In any case, if the transition to the back-up strategy (e.g., switching to a back-up component or entering a safe system state) has external dependencies, such as electric power, this must be replicated as well.

In distributed systems where the main functionalities are implemented in terms of software, a failure of a computation node can often be overcome by migrating the service (i.e., the execution of the software) to a different node (rescheduling).

Fail-operational systems might show a reduced performance after component failures. According to González [Gon+97], a behavior where a system's performance quality decreases in relation to the severity of failures is referred to as *graceful degradation*. Referring to the above example, in case the computing capacity on remaining nodes is insufficient to keep the full service alive, graceful degradation addresses the desire to keep at least critical services alive and to interrupt noncritical services only.

2.7 Fault Diagnosis

2.7.1 Introduction to Fault Diagnosis

For more than four decades, many researchers and engineers have been dedicated to the field of technical fault diagnosis. Based on the demand for an optimization of technical processes and improved product quality, computer systems were introduced and enabled a higher degree of automatization and the handling of complex control processes. In many industrial domains, human workers evolved from producers of products to operators of automated production processes. This required supervision functionalities, so sensors and indicator lights were introduced first. In combination with precise process descriptions, strategies for *fault detection* and *fault diagnosis* evolved.

In comparison, the detection of a failure, i.e., the deviation of a system's behavior from its specification, is easier than identifying the fault as the root cause of the failure. Strictly speaking, fault *detection* refers to an analysis of system or process outputs (e.g., sensor measurements) based on generated features that capture the system behavior. In a further step, fault *diagnosis* analyzes these features in order to classify a fault according to its type, size, location, and time, among others.

In the literature, the detection steps are often included in the diagnosis steps, so that the whole process of determining the details about an occurred fault is simply referred to as fault diagnosis. The following sections provide an overview of widely applied fault detection and diagnosis methods.

2.7.2 Fault Detection Methods

In order to guarantee an early and accurate detection of faults in technical systems, the detection mechanisms must be application-specific. The continuous developments in the diagnosis field over the last decades yielded a variety of strategies and approaches, which can be categorized according to their inputs and working principles. In the literature, different categorization approaches can be found, however, a classification into (1) methods that work on single signals (e.g., sensor measurements) and (2) those which are based on multiple input signals and models, is widely-used [Ise06; CP12; Alf17]. Based thereupon, Figure 2.9 gives an overview of typical fault detection methods.

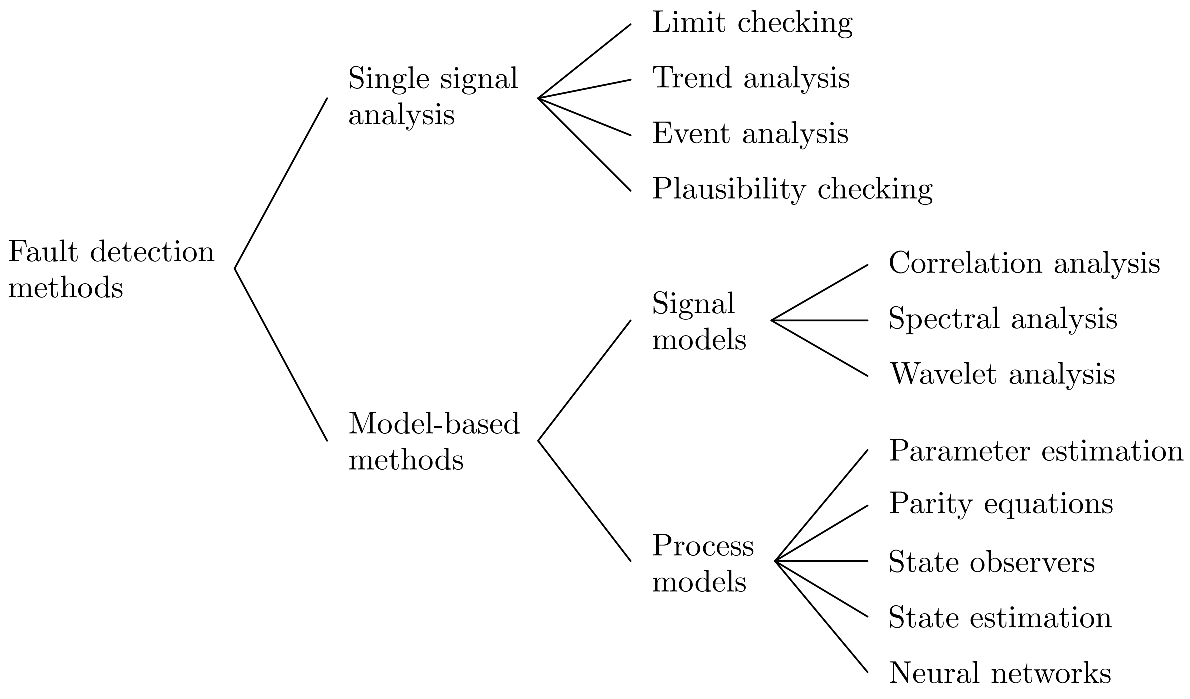


Figure 2.9: Fault detection methods.

2.7.2.1 Single Signal Analysis

The upper branch in Figure 2.9 refers to fault detection strategies which evaluate single signals from the underlying process. These methods make low demands on computation resources and system knowledge. Since they are inexpensive and easy to be implemented, they are widely applied in many industrial domains and processes. In limit and trend evaluations, the measured signals from the process are independently compared to a predefined (adaptive) threshold that marks a tolerance area. A violation indicates an unpermitted deviation from the usual condition and triggers an alarm, for example, for an operator to take care of the situation.

2.7.2.2 Signal Models

Model-based fault detection methods are subdivided into those which use signal models and those which apply process models (see Figure 2.9). The former class brings along the potential to extract specific information from signals (i.e., features). For example, spectral analysis (Fourier analysis) is suitable for detecting a faulty behavior of periodic signals by evaluating them in the frequency domain (e.g., vibration analysis of rotating parts of mechanical systems). A signal analysis with wavelets is suitable for acyclic signals, whereas meaningful information from stochastic signals can be extracted with correlation as well as with spectral analyses. For an efficient processing, certain *features* are defined and derived from the measurements. They form the basis for the analysis. If, for instance, a periodic signal (e.g., vibration analysis) is evaluated in the frequency domain, special interest might lie only on a small set of frequencies and their amplitudes, respectively. The so-called *symptoms* then capture the discrepancies for further interpretation (i.e., the subsequent fault diagnosis process). They should be robust against small, unavoidable variations and disturbances in the process.

2.7.2.3 Process Models

Another model-based fault detection method utilizes process models. This means that the inputs and outputs of the process to be diagnosed are compared with an analytically or experimentally derived mathematical model that serves as a replica of the process. These model-based fault detection methods make use of the fact that faults might change the behavior of the processes between the inputs and outputs in such a way that a comparison with the fault-free replica is able to reveal faults which would directly not be measurable. Obtaining an accurate model of the process is the prerequisite for making sound decisions on faults. Depending on the available knowledge or data about the process, the two most extreme modeling cases are (1) white-box models, where the underlying physical relationships as well as all parameters are known, so that the model is set up via linear or nonlinear differential equations, and (2) black-box models, which purely arise from measurable input/output signals or assumptions about the process (e.g., neural networks). In between, gray-box models are established by a combination of these input possibilities [Ise06, p. 73].

The difference between the model-produced output and the process output is called a residual. Ideally, in case of a very accurate model, the residual should be zero in a fault-free scenario. However, due to disturbances or noise, a certain tolerance interval must be introduced. The goal is to establish characteristic residuals for particular faults that show maximized fault sensitivity. Besides, the residuals should be robust against modeling errors [Ise06, p. 210]. With several different sensors collecting information about the process, potentially multiple residuals lead to a set of parity equations that form the basis for the inference on faults. On the other hand, a set of state observers can check the residuals for discrepancies and is able to reveal a faulty process behavior. In this regard, the features in model-based fault detection methods are residuals, parameters, or state variables. Analytical fault symptoms are derived for the subsequent fault diagnosis. The aspired robustness of these symptoms ensures a fault detection despite inevitably occurring small process variations and disturbances.

In order to obtain the necessary model accuracy for this kind of fault detection methods, so-called process identification methods are typically applied before the actual fault detection, e.g., parameter estimation (see e.g., [Jua94; Ise97; IM10]) or neural networks are particularly used for black-box models (see e.g., [Ise06]).

2.7.3 Active and Passive Fault Detection

Fault detection primarily pursues the goal of detecting faults while the system is running. In such cases, one speaks of *online* fault detection or even *online* fault diagnosis. The detection mechanisms are executed during the runtime of the system in parallel to the actual system service and do not affect or conflict with it. The opposite approach is *offline* fault detection, where process data is only recorded for a subsequent analysis. Both of these strategies fall into the category of *passive* fault detection, where all the information necessary to detect faults is obtained only passively, e.g., via sensors.

There is an extended approach that combines these system observations with interactions with the system. It is called *active* fault detection and is typically accompanied by the mechanisms of fault diagnosis (Section 2.7.4). In this strategy, the fault detection and diagnosis procedures can influence and potentially interfere with the process, e.g., by (periodically) executing specifically designed input patterns for a fault detection or by running diagnosis routines for fault identifications after a fault has been detected [PŠ18]. In this regard, such an active approach needs to be planned along with the normal system operation.

In the literature, one also finds examples where active diagnosis addresses the ability of a system to utilize fault diagnosis results at runtime for fault-specific fault tolerance or recovery strategies involving system interactions or reconfigurations, see e.g., [WW11] for a fault-tolerant control example using a four-wheel independently powered electric ground vehicle.

2.7.4 Fault Diagnosis Methods

According to the definition of SAFEPROCESS [IB97], the goal of fault diagnosis is to identify an occurred fault in terms of its type, size, location, and time of occurrence from the symptoms provided by the fault detection. A plethora of fault diagnosis methods is presented in the literature, often with slightly different terminology. An overview of widely used diagnosis techniques is presented in Figure 2.10, where the two major areas *inference methods* and *classification methods* are emphasized [Ise06; Alf17]. The former incorporate structural knowledge about the process to be diagnosed (i.e., a causal relationship between symptoms and faults), whereas the latter do not rely on such knowledge.

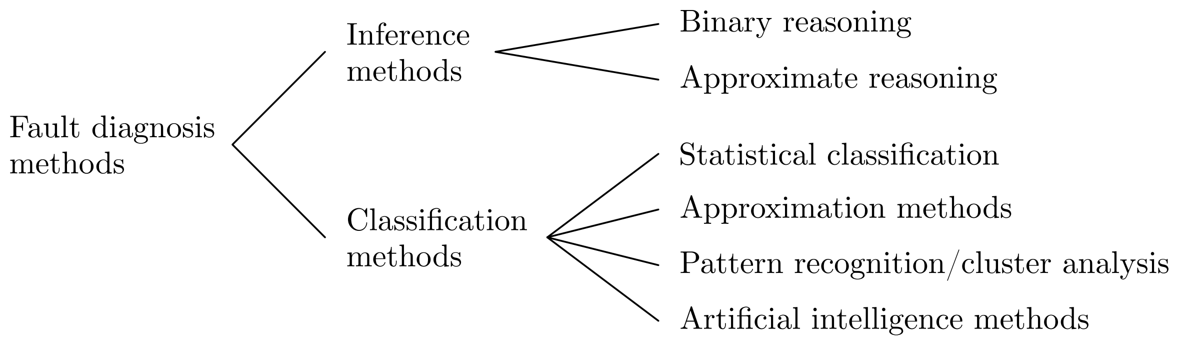


Figure 2.10: Fault diagnosis methods.

2.7.4.1 Inference Methods

In inference methods, structural knowledge in the form of causal relations can be implemented in the form of rules, e.g., *IF symptom* 1 *AND symptom* 2 *THEN fault* 1, (binary reasoning). Here, the symptoms represent a certain condition, whereas the fault (or an event) represents a certain conclusion. Fault trees or event trees as results of a fault tree analysis (FTA) or event tree analysis (ETA) visualize such causalities. Approximate reasoning can be seen as an enhancement of binary reasoning. Apparently, if symptom values are continuous in nature, their distance from a given decision threshold can be used to weight the conclusion and assign a higher or lower confidence, respectively (e.g., fuzzy logic).

2.7.4.2 Classification Methods

In fault diagnosis with classification methods, the aim is to distinguish a defined number of different faults from each other by evaluating a certain number of symptoms. The application of classification methods is particularly useful when no structural knowledge describing the symptom–fault relation is available. Instead, the required knowledge for a functional mapping of symptoms to faults is determined experimentally by a training and validation procedure.

In supervised learning, a classifier sees labeled data samples, i.e., the symptoms of certain situations with corresponding information about the faults. If the number of samples (in a dataset) for training the classifier is sufficiently large and covers the possible faults in various situations, the classifier is able to generalize and becomes capable of properly classifying previously unseen data in the so-called inference phase. In unsupervised learning, the training dataset comes without fault labels so that a cluster analysis can be performed. Recently, artificial intelligence methods (e.g., (deep) neural networks) gained much attention due to their impressive classification capabilities.

Statistical classification methods include, for example, Bayes classifiers and decision trees. Apart from that, polynomial classifiers can be mentioned as an example for an approximation method.

2.8 Data Compression

2.8.1 Introduction to Data Compression

Data compression is subfield of signal processing (here signal processing is synonymous with data processing). The term *data* in this context refers to information of arbitrary kind, such as an image or a text. An initial step in making data processable by a computer system is the encoding of the data. This refers to the representation of data in the form of binary sequences. The set of these binary sequences is called a *code*, with each individual member of the set being a *code word*. Every such unique bit sequence is then assigned to a unique element of an input alphabet, which is a collection of symbols, also called letters (or a finite range of values) [Say17, p. 27]. As an example, consider the American Standard Code for Information Interchange (ASCII), where an alphabet of letters, numbers, and special characters is encoded by 7-bit binary sequences.

In many cases, data originates from sensors that measure continuous physical quantities such as temperatures or voltages. The initial processing step is then quantization, meaning digitization of an analog signal. This is often realized by sample and hold circuits, where the measurement range is covered by a finite range of numbers, so that for every (periodic) measurement of the input signal a code word is produced by rounding the measurement to the nearest value of the given range.

Assuming that data is represented in terms of bits, data compression is the process of encoding the information with fewer bits than the original representation. Thus, both the memory space required for the data and the transmission time for the data is reduced. These properties qualify data compression algorithms to be employed in many data communication systems nowadays, e.g., audio and video streaming, mobile communications. Realizing data compression refers to the general idea of removing redundant information from the input by finding a different representation of the data. The opposite processing steps of retrieving the original representation are called data decompression.

In the 19th century, Samuel Morse developed the Morse code, which can be seen as an early example of data compression. The letters of an alphabet are encoded with dots and dashes (i.e., two different signal durations) and are prepared for transmission by sound or light, for example. In many written languages (e.g., English), certain letters occur more frequently than others. Thus, assigning shorter sequences of dots and dashes to more frequently occurring letters (e.g., *e* (·), or *t* (-)) and longer sequences to less frequently occurring letters (e.g., *p* (· - - ·), or *q* (- - · -)), the average time to transmit messages (i.e., words or sentences) is reduced. This concept also forms the basis of more advanced compression schemes like Huffman coding, which is discussed in Section 2.8.3.

There are two fundamentally different approaches for compressing data, namely *lossy* and *lossless* data compression. In lossy data compression, the original data cannot be exactly reconstructed from the compressed representation. In lossless compression, an exact data reconstruction is possible. In application domains where some loss of information can be tolerated, e.g., image or audio data compression, lossy data compression is often preferred. In contrast, if an exact reconstruction is necessary, e.g., for text compression, lossless compression is applied. Numerous compression algorithms are available for both strategies, some of which support both lossy and lossless compression (e.g., compressors for the Progressive Graphics File (PGF) format). Generally, the benefit of a reduced data size of the compressed data is associated with additional effort for the compression and decompression, respectively, e.g., time or energy consumption for the computations. The available compression algorithms exhibit particular characteristics which qualify or disqualify them for certain applications. Some of these are the time required for compression and decompression, the achievable compression ratio, or special properties of the storage format.

Two examples from different application domains highlight potential criteria to be considered when choosing a compression scheme and the corresponding parameters: (1) For live video broadcasting, a fast lossy compression and decompression is often preferred, even if this comes with a lower video quality. The priority is on instantaneous content delivery. (2) When delivering a large amount of program data (e.g., download of operating system updates by many users), the focus is on maximum lossless data reduction to save memory and transmission time rather than on the speed of the data compression (resp., decompression) algorithm.

The following sections give an overview of relevant lossy as well as lossless compression techniques, as highlighted in Figure 2.11. Many alternative categorizations of data compression schemes are possible (e.g., based on data types or applications), see e.g., [UVD18]. With respect to the diagnostic architecture of this thesis and the specific requirements for the data compression algorithms (see Section 3.1), further considerations on the suitability of particular data compression algorithms are provided in Chapter 3 (related work).

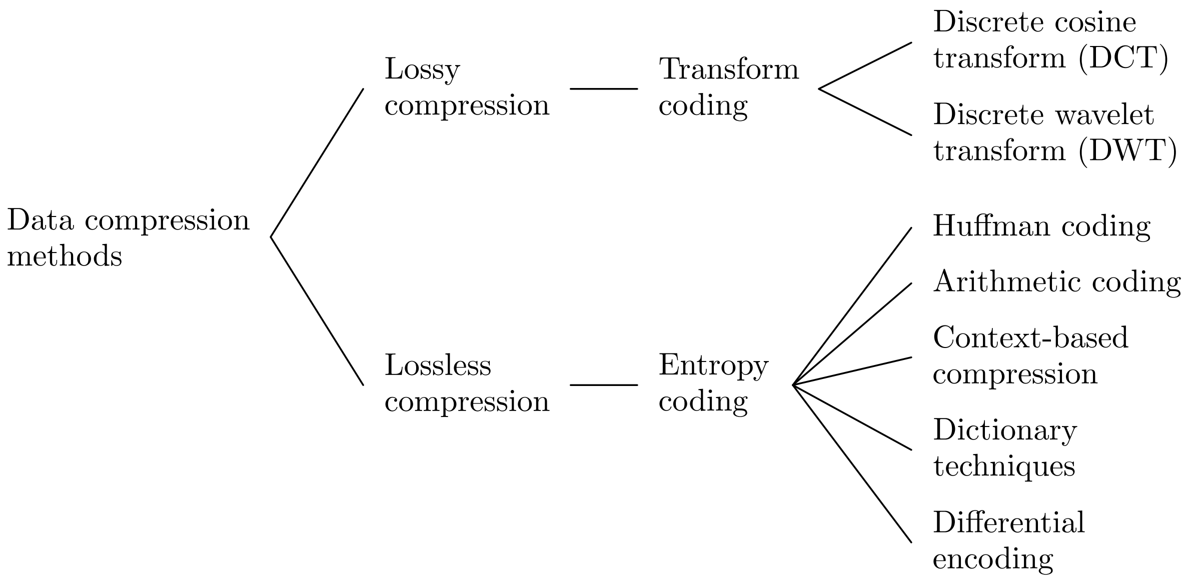


Figure 2.11: Data compression methods.

2.8.2 Lossy Compression

2.8.2.1 Introduction to Lossy Compression

It is the nature of lossy compression that part of the original data is lost after compression and subsequent decompression. If some information in the original data is irrelevant with respect to a recipient, losing this information is insignificant, e.g., high frequencies in audio data that a human cannot hear anyway. Lossy data compression is always possible. Of course, the data reduction potential is application-specific and requires knowledge about the relevance of the data in order to obtain acceptable results. If one assumes that a lossy compression algorithm orders some input data by relevance, then the reduction in data size can be controlled by a threshold that decides whether data is kept or lost.

In lossy compression, there are two basic compression schemes: (1) transform coding and (2) predictive coding. The main application area for lossy compression is multimedia data (images, video, or audio). For this type of data, the recipients are typically humans. A compression algorithm then takes advantage of several facts: Recorded input data exhibits irrelevant information for humans, e.g., the absence of very high frequencies in audio data has a marginal impact on the recipient. Moreover, a human brain is a specialist in reconstructing missing information. It compensates for some missing words in compressed speech (e.g., in a telephone call) and still captures the meaning of a sentence.

Two widely-used transform-based compression methods are discrete cosine transform (DCT) coding and wavelet compression (which is actually a variant of DCT coding). The former is suitable for compressing images, video data, audio data, and speech. The latter is particularly well suited for representing transients in the data to be compressed. This refers to sudden distinct changes in the data, for example, audio with percussion sounds or an image of a night sky with stars.

The DCT is briefly introduced in the following section. In addition, one finds other lossy compression techniques particularly designed or qualified for certain multimedia applications. Examples for compressing images are Cartesian Perceptual Compression, fractal compression, and S3TC texture compression. The Sorenson codec is an example for video compression. For compressing general audio, adaptive differential pulse-code modulation and aptX/aptX-HD are widely-used, and particularly for speech compression, linear predictive coding with many variants has become well-established.

2.8.2.2 Discrete Cosine Transform

Using the discrete cosine transform, a finite set of input data points (e.g., audio samples or pixels of an images) is represented as a weighted sum of cosine functions with different frequencies. The mathematical equations of the transform were originally introduced in [ANR74] and the DCT has since become highly important for lossy data compression. After transforming the data into the so-called frequency domain, DCT coefficients are obtained, the first of which is called the DC coefficient and the others are called AC coefficients, respectively. These coefficients are real numbers and they can be positive or negative. Under ideal conditions (no precision loss during processing), the original data can be completely recovered via an inverse DCT. Audio and image data in particular consist of many correlated quantities. In the frequency domain, most of the input information is then concentrated in the lower frequency range, meaning that typically only the first few DCT coefficients are large. The other coefficients, which hold higher frequency information, are typically low or even zero, and therefore do not contain much information. A reduction of data is done in the frequency domain by quantizing the DCT coefficients. The small ones are coarsely quantized or even set to zero, whereas the larger ones are typically rounded to their nearest integer value, which is then stored using fixed- or variable-size codes. A decompressor performs an inverse DCT based on these coefficients and obtains a slightly quality-reduced version of the original input information.

Depending on various parameters (e.g., threshold for setting DCT coefficients to zero, coarseness of the quantization) the quality loss of the input data can be controlled. DCT coding is the basis for many lossy image compression formats (e.g., JPEG) or for video compression (e.g., advanced video coding (AVC/H.264/MPEG-4 AVC)). In a modified version (MDCT), it is also used for compression of general audio (e.g., Dolby Digital, MPEG Layer III (MP3), advanced audio coding (AAC), windows media audio (WMA)) and for speech compression (e.g., AAC-LD, or Opus).

More technical details to the transform, practical realization examples, and variations of the basic scheme can be found in [Mac05; SM10; RY14; Say17]. They are not discussed here as DCT-based compression only plays a minor role in the compression of data relevant for control systems and diagnosis processes considered in this thesis.

2.8.3 Lossless Compression

2.8.3.1 Introduction to Lossless Compression

Lossless compression is not always possible and requires the data to exhibit some kind of redundancy. If this is not the case, e.g., if the data is entirely random, the Kolmogorov complexity [Kol63] precludes lossless compression as there is no shorter description that is able to produce the data as an outcome, see e.g., [LV+08]. Besides, the pigeonhole principle can be used to explain why certain data cannot be compressed. It states that if n objects are to be distributed among m sets ($n, m > 0$) and if $n > m$, then there must be at least one set containing more than one object [Aig+18, Chapter 28]. With respect to data compression, an analogy can be made to data stored in a memory of size n. Assume an arbitrary sequence of n bits ($n > 1$) which should be reduced to at least $n - 1$ bits. There are 2^n unique binary input sequences, but only 2^{n-1} unique output sequences, i.e., half as many output sequences as input sequences. Here, at least in one case, more than one input sequence maps to the same output sequence. Since lossless compression has to be seen as a reversible mapping, it is impossible to reconstruct the correct input from the output in the above case.

Lossless data compression is the preferred technique to be applied in this work. The diagnostic data to be communicated among the nodes of a distributed system is mainly raw data originating from sensors or preprocessed intermediate results, i.e., numbers of certain limited intervals.

In the following, a selection of widely used lossless compression techniques will be presented in order to lay the foundation for the further chapters of this thesis. Firstly, a look at statistical compression methods is taken: Huffman coding as well as arithmetic coding are two of the most common entropy encoding schemes. Moreover, context-based (text) compression is highlighted. Dictionary-based methods are introduced thereafter, with a special focus on the compression techniques by Lempel and Ziv with their many variations. Finally, differential encoding and predictive encoding, which can be used for both lossless and lossy data compression, are briefly discussed.

2.8.3.2 Huffman Coding

Huffman coding is named after David Huffman, who introduced the coding scheme in 1952 [Huf+52]. Like most entropy codes, the Huffman code assigns variable-length output symbols (i.e., code words) to a defined number of fixed-length input symbols. According to an estimate of the probability of occurrence of the input symbols in the input data (e.g., how frequently do letters in a certain written language occur), the lengths of the output code words vary. Similar to the Morse code, the Huffman code assigns shorter code words to symbols that occur more frequently and vice versa, and thus, reduces the average number of bits per symbol. Ideally, the (binary) output code word length for a symbol is $-\log_2 P$, where P is the probability of occurrence of the input symbol (see Shannon's source coding theorem [Sha48]). This results in shorter code words for common input symbols and larger code words for rarely occurring input symbols. To be uniquely decodable, the Huffman code must satisfy the Kraft-McMillan inequality, a necessary and sufficient condition for the existence of a prefix code [Kra49]. A prefix code (i.e., a prefix free code) requires that no whole output code word is a prefix (i.e., the initial segment) of any other output code word.

As an example, assume an alphabet $A = \{a_1, a_2, a_3, a_4, a_5\}$ with the probabilities of occurrence of its symbols being $P(a_1) = 0.4$, $P(a_2) = P(a_3) = 0.2$, and $P(a_4) = P(a_5) = 0.1$. Since $P(a_1)$ is largest, one expects its code word $c(a_1)$ to be the shortest. An optimal solution for assigning code words (i.e., bit sequences) to the symbols could then be $c(a_1) = 1$, $c(a_2) = 01$, $c(a_3) = 000$, $c(a_4) = 0010$, and $c(a_5) = 0011$. Being prefix free, Huffman codes can be represented as binary trees, where the leaves correspond to the symbols, as exemplarily shown in Figure 2.12.

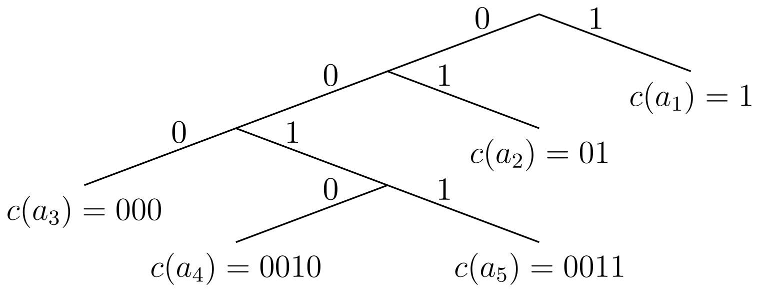


Figure 2.12: Huffman coding example.

The efficiency of the code can be measured in terms of its redundancy, i.e., the difference between the entropy and the average length for the code, which is desired to be close to zero. In the above example, the entropy according to Shannon [Sha48] is $H = -\sum_{i=1}^{5} P(i) \log_2 P(i) = 2.122$ bits/symbol and the average length, calculated from the probabilities and the number of assigned bits to each symbol, is $0.4 \cdot 1 + 0.2 \cdot 2 + 0.2 \cdot 3 +$

$0.1 \cdot 4 + 0.1 \cdot 4 = 2.2$ bits/symbol. The redundancy then is 0.078 bits/symbol. For detailed information regarding the design of the code, see [Say17, Chapter 3].

If the probability of occurrence of the input symbols is initially unknown, an adaptive Huffman strategy can be applied. Then, a two-pass procedure first collects the statistics from the input data before the actual encoding step.

Many variants of the original approach have been developed based on Huffman's work [Huf+52], e.g., nonbinary Huffman codes, adaptive Huffman codes, Golomb codes, Rice codes, and Tunstall codes. The original Huffman code and its variants pair also well with other compression algorithms. Huffman codes are applied in many different applications, such as lossless image compression, text compression, and audio compression.

2.8.3.3 Arithmetic Coding

Arithmetic coding is also a form of entropy encoding. Unlike other implementations, such as classical Huffman coding, it encodes an entire input message (consisting of multiple input symbols) into a single, arbitrarily precise number x from the interval $[0, 1) = \{x \in \mathbb{R} | 0 \leq x < 1\}$ (which is known to the encoder and decoder). The encoding scheme is based on the probabilities of the occurrences of the input symbols. Starting with the given interval $[0, 1)$, the encoding scheme forms sub-intervals according to the probabilities of occurrence of the symbols. This process is repeated in the sense that the sub-interval that corresponds to the currently encoded symbol becomes the next interval to be divided into sub-intervals according to the same model. The order (e.g., an alphabetical order of the input symbols) in which those sub-intervals are formed, must be known to the encoder and decoder. This iterative process is terminated by the end of the sequence to be encoded. The number x is then chosen as a number from the last interval which can be expressed with as few bits as possible.

As an example, let the symbols of an alphabet $A = \{a_1, a_2, a_3\}$ have the probabilities of occurrence of $P(a_1) = 0.7$, $P(a_2) = 0.2$, and $P(a_3) = 0.1$. Let a_1, a_3, a_2 be the input sequence. Starting with the initial interval $[0, 1)$, one obtains the three sub-intervals $[0, 0.7)$, $[0.7, 0.9)$, and $[0.9, 1)$ according to the probabilities. As the first symbol of the input sequence is a_1, its corresponding interval $[0, 0.7)$ is now divided in the same manner as before, so the next sub-intervals are $[0, 0.49)$, $[0.49, 0.63)$ and $[0.63, 0.7)$, see Figure 2.13. The next symbol is a_3, so the sub-intervals are now $[0.63, 0.679)$, $[0.679, 0.693)$, and $[0.693, 0.7)$. As the last symbol of the input sequence is a_2, x must lie in the interval $[0.679, 0.693)$, which can be seen as the current range of information (marked in blue in Figure 2.13). A good choice for x would be 0.68 (or 0.69). In addition to x, the decoder needs to know the length of the sequence, or else, an end-of-data symbol has to be transmitted (i.e., additionally encoded). Based on the same probability model, the decoding process forms sub-intervals

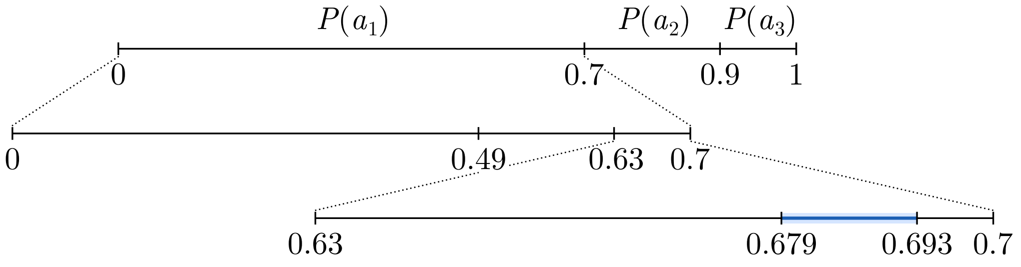


Figure 2.13: Arithmetic coding example.

and iteratively extracts the current symbol by checking in which sub-interval x lies. For the storage or transmission of the encoded data, one needs to generate a unique binary representation of x in an efficient manner. From the above example, where both numbers 0.68 and 0.69 represent the encoded information equally well, it is apparent that the use of decimal numbers is somehow inefficient. Better results can be obtained by storing a different fraction x from the interval, which allows a shorter binary representation than a decimal to binary conversion of either one of the above values. Further details on the generation of such binary codes can be found in [Say17, Chapter 4].

There are many variations and extensions of classical arithmetic coding, such as adaptive arithmetic coding, where the symbol's probabilities of occurrence can change over time, or where an adaptive alphabet is used. Arithmetic coding can be found in many standards of the multimedia domain, e.g., published by the International Standards Organization (ISO) or the IEC and is applied in image compression, audio compression, and video compression, among others.

Arithmetic coders are capable of producing near-optimal outputs for arbitrary sets of symbols and probabilities. For a fair comparison with Huffman codes, one must consider the alphabet size, the distribution of the probabilities, and the message sizes. Arithmetic coding requires longer input sequences to work efficiently. For a symbol-wise encoding, Huffman codes typically perform better [Say17, pp. 125–127]. Encoding longer sequences of input symbols is also possible, but the codebook for all possible sequences would grow drastically. When it comes to changing input statistics, arithmetic coding scales better. Compared with Huffman coding, arithmetic coding is more computationally complex but, depending on the parameters, for most sources arithmetic coding is able to produce better results in terms of an ideal code word output length ($-\log_2 P$). As an example, consider facsimile, where the alphabet size is rather small and the probabilities are highly unbalanced. In such cases, the use of arithmetic coding is generally favored despite its additional complexity [Say17, Chapter 4].

2.8.3.4 Context-Based Compression

Context-based compression techniques are mainly applied for text compression and take advantage of the fact that letters (of an alphabet) typically occur in certain contexts. The probabilities of the letters' occurrences are important information for an encoder. It was mentioned in Section 2.8.3.2 (Huffman coding) that it is beneficial for compression if those probabilities are skewed. This means that certain letters occur with higher probabilities than others. This information generally depends on many factors, e.g., the language and the type of text. Context-based compression utilizes the fact that the letters' probabilities of occurrence highly depend on their preceding letters (this is referred to as the *context*). As an example, in a typical English text, the probability for the letter u to appear is about 3 % (depending on the analysis; see e.g., the Concise Oxford Dictionary). However, if the preceding letter was a q, then the probability for a u increases drastically.

Algorithms such as the PPM (prediction with partial match), originally introduced by Cleary and Witten [CW84], constantly prepare an estimate of the current input letter's probability of occurrence based on a preferably long context, in order to obtain an appropriate probability set for a subsequent arithmetic encoder (recall Section 2.8.3.3). In detail, the PPM encoder maintains a data structure to remember the contexts in which the letters have occurred. The length of the considered context is assumed to be fixed. An order-N context includes the last N symbols processed. For every new input letter, the algorithm checks its data structure for the current order-N context and determines the probability for the current letter to follow that context, e.g., from counts of how often certain letters followed that specific context. If the search was successful, the relevant information is forwarded to an adaptive arithmetic encoder. If there was no match, the algorithm looks for shorter contexts by reducing the order step by step and, in a success case, it behaves as above. Determining an order-0 context means that the letter was never seen in any context, but appeared individually in the text. In the special situation where a letter has never appeared before, the algorithm uses fixed probabilities delivered with the alphabet and indicates this via a reduction of the context order to -1. The information about such a context size reduction needs to be encoded too (e.g., as a special letter of the alphabet), and its probability has to be properly assigned. The symbol used for the indication is referred to as *escape symbol*.

Many variants of the PPM algorithm, e.g., PPMC or PPMX, tackle the problem of determining precise probabilities of the escape symbol. The former implements a counting strategy, whereas the latter is based on a Poisson distribution.

The Burrows-Wheeler Transform (BWT), developed in 1994 [BW94], can also be advantageously applied in context-based compression. It utilizes the fact that in many languages words and syllables start with a limited set of letters, e.g., in English many words start with the letters *s* or *e* (see the Concise Oxford Dictionary). The BWT is part of the *bzip2* application for general purpose lossless data compression. In a block-wise manner, the transform permutes the letters of the input, thus establishing a new sorting in which those frequently occurring letters or sequences of letters lie close to each other. A subsequently applied classical context-based compressor such as the PPM is then able to achieve a better compression. Of course, the decompression requires the indices of the permutation to recover the original order of the letters.

2.8.3.5 Dictionary Techniques

Dictionary-based compression utilizes the fact that the outputs of a source (i.e., sequences of symbols) often show recurring patterns. Assume a written text where the words are formed from the symbols (i.e., letters) of an alphabet. One typically finds both frequently recurring sequences of letters within words (e.g., *th* (in English texts)) and recurring sequences of multiple words that belong together (e.g., *a lot of*). Dictionary-based compression aims to save bits by storing frequently recurring patterns in a dictionary. Then, if the output of the source matches such a stored pattern, it can be encoded with the respective dictionary index instead. This strategy is efficient if the size of the dictionary is smaller than the number of all possible patterns.

As an example, assume a text consisting of four-letter words based on an alphabet with 16 letters. Each letter should have the same probability of occurrence. There are $16^4 = 2^{16} = 65536$ possible combinations of such words, and if all of them were equally likely to occur, one could straightforwardly encode each letter with $\lceil \log_2 16 \rceil = 4$ bits, and a 4-letter word with 16 bits. Now assume a text where some of the words are more likely to occur than others. Then, for instance, a dictionary which holds the 32 most likely 4-letter words addresses each of its entries and consequently the word with $\lceil \log_2 32 \rceil = 5$ bits (compared to 16 bits for the word if stored as above). Of course, when encoded like this, the decoder must have the same dictionary and it must know in which way the received bits are to be interpreted. For this purpose, an indication bit can be added to each stored bit sequence. So if the source output is found in the dictionary, the encoding requires 6 bits and 17 bits otherwise. To benefit from the strategy, the average number of bits required per word must be less than 16 in the example. It depends on the probability P that a word is found in the dictionary: $6P + 17(1 - P) < 16$, i.e., $P > 0.091$. One concludes that a large number for P is targeted, which means that the dictionary size must be properly chosen as well as the knowledge about the source must be well-founded. More details on the analysis of the probability P can be found in [Say17, Chapter 5].

In many works, researchers introduced various dictionary-based compression strategies, some of which maintain static dictionaries and others utilize adaptive dictionaries. The two well-known dictionary-based strategies *LZ77* and *LZ78*, which will be discussed in the following sections, can be seen as foundations for many up-to-date approaches.

LZ77

In their 1977 paper [ZL77] Jacob Ziv and Abraham Lempel introduced a dictionary-based compression approach which is referred to as *LZ77* or *LZ1*. It is based on a dynamic dictionary, realized as a fixed size sliding window consisting of two parts (also denoted as buffers). On the one hand, the sliding window stores a certain number of previously seen symbols of an input sequence in the *search part* of the window, e.g., previously seen letters of a text. Let its size be S bits (with $S > 0$). On the other hand, it holds some of the next symbols of the sequence to be encoded in the *look-ahead* part of the window. Let its size be L bits ($L > 0$). In the encoding procedure, the search window is scanned for the longest matching sequence of symbols which begins with the first symbol of the look-ahead window. If such a match exists, it is indicated via an *offset* value o and a *length* value l, otherwise both values are set to zero. The encoded data is then stored as triples, each containing the offset, the length, and lastly the code word corresponding to the first symbol in the look-ahead window that is not part of the matching sequence. This is graphically highlighted for an example alphabet $A = \{a, b, c\}$ in Figure 2.14. The encoded triple is $(6, 4, C(b))$, where $C(b)$ corresponds to the code word of the symbol b.

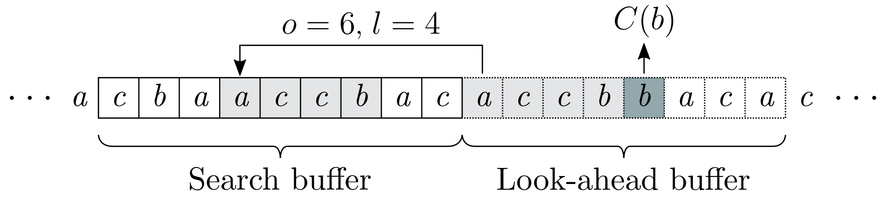

$o = 6, l = 4$
$C(b)$
$\cdots$ a c b a a c c b a c a c c b b a c a c $\cdots$
Search buffer
Look-ahead buffer

Figure 2.14: LZ77 encoding example.

The encoding process allows the length of the matching sequence to be greater than the length of the search window; the sequence only has to start in the search window. This refers to a case where the relevant content of the search window shows a repetitive behavior in the look-ahead part of the window. For instance, if a triple holds $o = 4$ and $l = 7$, then after decoding the first four symbols via a copy pointer, the pointer is continued to be moved by three more symbols until the denoted length value is reached. That is, the first three symbols are decoded again. Consequently, for an alphabet A of size $|A|$ (with $|A| > 0$), $\lceil \log_2 S \rceil + \lceil \log_2 (S + L) \rceil + \lceil \log_2 |A| \rceil$ bits are needed to store a triple using fixed-length codes [Say17, p. 136].

The working principle of the LZ77 scheme is rather simple on the encoder and decoder side. No prior knowledge of the source is required, however, the scheme works under the premise that recurring patterns of symbols lie close to each other. In a worst-case scenario, the use of LZ77 might even lead to an expansion rather than a compression of the original data size. The LZ77 scheme can be straightforwardly combined with other compression techniques, e.g., further encoding of the triples with a variable-length code such as a Huffman code is possible. Variants of the original LZ77 scheme are part of many up-to-date compression techniques, e.g., for the Zip format or PNG image compression (see the *Deflate* compression algorithm introduced by Phil Katz).

A well-known variation called LZSS increases the efficiency of the original scheme by eliminating the need for the triples [SS82]. It only stores the offset and length information for the dictionary access and concatenates a bit as a flag to each encoding in order to differentiate between an individually encoded symbol and a dictionary access. Other variants of LZ77 have been published by Williams; in [Wil91] he proposes to use hash tables to find string matches. His ideas also led to the LZRW4 approach, where he added prediction capabilities to the compression scheme [SM10, p. 364]. This idea was also used in the LZP variant by Bloom [Blo96].

LZ78

For many sources, the assumption of the previously introduced LZ77 scheme that repeating sequences of symbols appear close to each other does not hold. In 1978, it was again Jacob Ziv and Abraham Lempel who introduced another dictionary-based compression approach, the *LZ78* (also named *LZ2*) [ZL78]. This compression scheme maintains an explicit dictionary that stores previously seen sequences of symbols during the encoding process and basically keeps them until further dictionary growing is limited by the available memory or other constraints.

The working principle of LZ78 is as follows: Initially there is an empty dictionary. The input symbols to be encoded are stored as a double with two entries. The first entry is an index of the dictionary and the second entry is a code word of an input symbol. For every new input symbol, the compression procedure searches the dictionary for a match and, if successful, the next input symbol is appended to the current symbol. Then the dictionary is searched for a match for this sequence of two symbols. These steps are repeated until there is no longer a match. This sequence is then encoded by the dictionary index corresponding to the longest match and the latest symbol that led to the unsuccessful search. In addition, the sequence is inserted at a new entry in the dictionary. In this way, the decoder builds the same dictionary as the encoder. In case of no match, the encoded double contains 0 at its first entry and the code word of the current symbol at the second entry.

As an example, consider the sequence of symbols $a\,b\,b\,c\,b\,c\,a\,b\,a$ formed from an alphabet $A = \{a, b, c\}$. Assuming an empty dictionary D, the first encoded double is $(0, C(a))$, with

$C(a)$ denoting the code word of the symbol a, and the first dictionary entry is consequently set to a. Table 2.1 illustrates the construction of the dictionary according to the encoder output for each symbol of the input sequence. The second symbol, b, is encoded in the same way. Since b gets stored at $D[2]$, the next two input symbols, bc, are encoded as $(2, C(c))$ and stored at $D[3]$. For the following symbols, the longest match is found at $D[3]$, so bca are encoded as $(3, C(a))$.

Table 2.1: Development of the dictionary in the LZ78 approach.

Encoder		**Dictionary**	
Position	Output	Index	Entry
1	$(0, C(a))$	1	a
2	$(0, C(b))$	2	b
3	$(2, C(c))$	3	bc
5	$(3, C(a))$	4	bca
8	$(2, C(a))$	5	ba

For practical implementations, an unlimited growth of the dictionary is not useful and can be prevented, e.g., by replacing some entries with newer ones or by treating the dictionary as a static dictionary once a certain size is reached. An efficient strategy for achieving optimized code word lengths using online dynamic dictionaries is proposed in [Cho+08]. In any case, the applied strategy of the encoding process must be known to the decoding process. Many variations of the LZ78 have been proposed in the literature, of which LZW is probably the most commonly applied variation (see next section). Besides, hybrid approaches of LZ77 and LZ78 have been investigated, e.g., by Fiala and Greene and are known by the acronym LZFG [FG89].

LZW

A well-known modification of LZ78 was introduced by Terry Welch in 1984 [Wel84] and is referred to as LZW. In his proposed strategy, the dictionary is initialized with all the symbols of the source alphabet, so that in any case a symbol can be encoded with a dictionary entry. This eliminates the need to encode a symbol as a double (as it is done in LZ78). Further dictionary entries holding longer sequences of symbols are established during the encoding process.

The LZW compression procedure works as follows: Since every individual symbol has a match in the dictionary, the algorithm forms a longer sequence from the input symbols, comparable to LZ78. After every concatenation of a further symbol, the growing sequence is searched in the dictionary and this process is repeated until there is no match anymore. In the latter case, the sequence gets stored at a new dictionary entry. The last appended symbol cannot be encoded in this step, as a decoder does not yet know the new dictionary entry. Instead, this symbol forms the start of the next sequence, while all other symbols of the matching sequence are encoded with one dictionary entry. A decoder builds its dictionary from the decoded symbols analogously to the encoder dictionary.

Similar to LZ78, practical implementations of LZW require a limitation of the dictionary. As an application example, the Graphics Interchange Format (GIF) utilizes the LZW compression scheme. Several variants of LZW aim for a faster dictionary adaptation compared to the original approach. For instance, the LZMW strategy assigns sequences of multiple symbols to a new dictionary entry at once, which however comes at the cost of a more complex dictionary search procedure [MW85]. The LZAP and LZY variants introduced by Storer and Bernstein, respectively, modify multiple dictionary entries at once, e.g., by additionally storing some prefixes of the added sequence. This allows a faster search for matches (compared to LZMW), but typically requires larger dictionaries, and thus more bits are needed per dictionary index [SM10; Sto88].

2.8.3.6 Differential Encoding

Differential encoding, also called delta encoding, is a strategy for storing data in terms of differences with respect to a reference. This is especially useful in case sequential data needs to be stored. The strategy can be applied in both lossy and lossless forms to many different kinds of data, such as text files, audio samples, or images. An intuitive example of differential encoding is revision control software for text files, such as programming code. Once a complete version is initially stored, it serves as a reference. Future modifications to the text, which usually affect only a minor part of the entire text, are represented as difference information from the reference. This difference representation offers great potential to be reduced in terms of required disk space, e.g., by applying one of the above-mentioned lossless compression strategies. Sampled sensor data of measured physical quantities as well as audio data, among others, often show a strong correlation between consecutive data samples. This means that the differences between these samples are relatively small and thus the dynamic range as well as the variance of these differences are smaller than those of the source data [Say17, p. 352]. This leads to a smaller entropy of the difference data, which can then be advantageously exploited by an entropy-based encoder. In this regard, (adaptive) differential pulse-code modulation (DPCM) is widely applied. It makes use of the correlation between consecutive samples and (in a complex version) is able to produce small difference values with a minimized quantization noise by utilizing some n previous samples to predict the value of the next sample for the difference computation [Pro+02, p. 309].

2.8.3.7 Other Lossless Compression Techniques

One straightforward lossless compression technique is run-length encoding. This is advantageous if data exhibits many sequences in which consecutive repetitions of the same symbol occur. Roughly speaking, the algorithm then encodes the symbol only once and additionally stores the count value.

Many lossless compression algorithms are particularly designed or qualified for multimedia applications: for raster graphics, e.g., lossless discrete cosine transform, JPEG 2000, or PNG; for video, e.g., H.264/H.265 lossless, Motion JPEG 2000 lossless; for audio e.g., Adaptive Transform Acoustic Coding (ATRAC), or the Free Lossless Audio Codec (FLAC).

Distributed source coding can be used for both lossy and lossless data compression and is further addressed in Section 3.2.4 (related work). Based on the theoretical foundations of [SW73] and [Cov75], practical implementations (especially for wireless sensor network (WSN)) exploit the fact that correlated data sources that cannot communicate with each other can compress their data advantageously if a common decoder is used that has knowledge about the correlation [ANH16].

The above sections introduced many well-established lossy as well as lossless compression techniques. Chapter 3 builds upon this knowledge to filter out those candidates of data compression techniques that can serve as a foundation for the time-triggered architecture of this thesis. These candidates are reviewed in detail for their capabilities and limitations in the context of the related work.

For further details on general working principles, variants, or other data compression algorithms, the reader is referred to the literature, e.g., [Mac05; SM10; Say17].

Chapter 3

Related Work

3.1 Requirements

Time-triggered systems play an important role in high-dependability and safety-critical systems. They allow predictable system behavior even under high-load and in fault situations, since all task executions and message transmissions are timely scheduled with respect to a global time base. The prior knowledge about the permitted temporal behavior offers error detection and fault isolation [Obe11, p. 2], and thus enables fault tolerance in terms of subsequent system reconfigurations, e.g., rescheduling or service degradation (with regard to nonfunctional properties).

Redundancy (e.g., replication of hardware, computations, and information) is necessary to realize fault tolerance. Moreover, active fault diagnosis in combination with system interactions is an efficient way to implement fault tolerance and avoid excessive hardware redundancy.

Sensor measurements of physical quantities (e.g., currents, voltages, and temperatures) are a primary source of information for fault diagnosis. Large-scale and complex distributed systems have numerous sensors placed at different locations (see e.g., [KOK14]) such that their data needs to be made available to the tasks of the diagnostic process via the communication network. A diagnostic process, which is modeled according to Section 4.3, involves different kinds of tasks such as data preprocessing, signal trend analyses, correlation and plausibility analyses, amongst others. These tasks produce diagnostic (intermediate) results which in turn serve as inputs to other tasks of the process which has the overall goal to narrow down fault symptoms (i.e., a detected deviation of a system from its specification) to their root cause. This is additional data that would not be needed without diagnosis.

Since system resources, i.e., computation nodes and the communication network are typically shared, the network traffic of diagnosis applications competes with the regular communication of the system. As today's systems continue to evolve, more and more tasks are demanded to be integrated. As a consequence, communication resources become bottlenecks and limit the services that can be executed as well as real-time support. Considering further that transmitting messages between nodes in a distributed system is generally more expensive (e.g., in terms of energy consumption or data protection) than computing tasks at the nodes or storing data, compressing data for transmission is an excellent strategy to achieve better utilization of available communication resources. Efficient usage of communication resources is more significant than optimization of computation resources [DFG16].

In detail, the following reasons support the goal of this work to utilize online data compression for time-triggered communication with a special focus on sensor data for diagnosis applications. The points apply both to newly designed distributed time-triggered systems, where a key focus today is low energy consumption, as well as to existing time-triggered systems, where new (additional) services are to be executed as the system evolves. It also applies to open distributed systems, where new services can join and leave a system at runtime to contribute to the overall service.

- Compressed messages in time-triggered systems help to optimize schedules, in particular to decrease the makespan (i.e., the length of a schedule), thus allowing tighter deadlines to be met and supporting shorter overall service times, e.g., for diagnostic services.
- Shorter messages contribute to a reduction of network traffic and enable a higher level of integration and the combination of multiple services in one system, including online diagnosis.
- Especially during transmissions, data is prone to corruption. By removing unwanted redundancy, data compression helps to reduce the amount of data to be transmitted. The space thereby freed up can consequently be used to increase the fault tolerance of a system, e.g., by protecting such data with deliberately added redundancy (e.g., consider forward error correction (FEC)).
- Diagnostic information (e.g., sensor data) often contains redundant parts due to functional or stochastic dependencies between the information. These signal characteristics can be utilized beneficially for data compression, leading to more efficient use of available system resources (e.g., memory and communication capacities).

An essential source of diagnostic information is data from sensor measurements of physical quantities, such as voltages, currents, and temperatures. This work assumes that data values for online fault diagnosis are initially obtained by physical sensors that produce data samples of real numbers (or tuples of real numbers). These real-valued data samples are mapped to a finite range of N values identified by the numbers $0, \dots, N-1$. Standard quantization techniques such as analog-to-digital-conversion (ADC) using sample and hold circuits are applied for this purpose [SM10; Say17]. Any number from the range $0, \dots, N-1$ can be encoded with $\ell := \lceil \log_2 N \rceil$ bits.

In order to realize the above benefits and fulfill the objectives of this thesis, appropriate online data compression algorithms beyond the state of the art are required. As listed in the following, specific requirements arise for the data compression and imply the compression algorithms to exhibit certain features. They must incorporate the needs of both time-triggered systems and online diagnosis applications.

R1: In safety-critical real-time systems where the reliability depends on online diagnosis, the violation of a certain deadline might render a diagnostic result unusable and subsequently lead to a system failure, e.g., if a system reconfiguration or a back-up strategy is initiated too late in the event of a fault. Therefore, data compression and decompression have **hard real-time constraints**. With respect to time-triggered systems, the compression algorithms must **guarantee short worst-case compression times (WCCTs) and worst-case decompression times (WCDTs)**, respectively.

R2: To utilize online data compression for a scheduling process, the compression algorithms must guarantee a certain **worst-case compression ratio (WCCR) on short input data sequences** (see also point R3). Furthermore, this WCCR should be less than 1 for the compression to be useful, i.e., the encoded data has fewer bits than the input data.

R3: With the primary goal of compressing data for communication, only **online (i.e., one-pass) data compression** techniques must be used. An incoming data value (or a short (fixed-length) sequence of input data values) must be compressed before the next one arrives to account for real-time capabilities of online diagnosis. In order to take into account the temporal dynamics of data streams, the ability to buffer certain parts of data must be ensured, e.g., to exploit the temporal redundancy of multiple data streams (see also point R7; for the definition of a data stream, see Section 4.3.2).

R4: **Guaranteed data quality with respect to time-triggered messages**: Due to the impossibility of a lossless compressor to produce fixed-size outputs from fixed-size inputs, a lossless compression algorithm with bounded omissions is needed (neglecting an initial quantization phase for sensor data, which is lossy). In this regard, allowing bounded omissions refers to the strategy of either encoding a data value into a compressed representation that can be exactly recovered, or encoding it into an indication value that tells the decoder (resp., receiver) that the original data value is lost and can only be reconstructed with a lower accuracy.

R5: To strictly adhere to the scheduled communication, there is **no additional channel for exchanging overhead information** between a sender and receiver. All potential overhead information, such as synchronization information, must be extracted from the main data communication channel.

R6: The compression algorithms must be able to dynamically adapt to signals if their behavior changes over time without compromising compression performance. To achieve this, the compression algorithms must be able to process signals even if **no prior statistical information** about the probability distribution of their data values is available.

R7: To achieve an optimized compression performance, the algorithms must be able to **exploit redundancies among multiple data streams** in terms of their functional, stochastic, or temporal dependencies, e.g., signal correlations.

R8: The compression algorithms must support efficient **merging and splitting for branching data streams** at the nodes of a distributed system (without full decoding and re-encoding). Such capabilities enable a scheduling algorithm to make optimal use of data compression on the system level.

3.2 Related Work

The choice of a suitable compression algorithm for a specific application depends on many considerations. [UVD18] reviews various data compression techniques based on different categories such as the underlying data type or the application. In the following, an overview of state-of-the-art data compression techniques is presented and reviewed with respect to the requirements from Section 3.1.

3.2.1 Transform Coding

Classical lossy compression techniques based on transform coding (e.g., discrete cosine transform (DCT) [RY14] or wavelet transform [SKR14]) are inherently lossy; the original data cannot be recovered without some error. These techniques are mainly applied in multimedia applications (e.g., storing or transmitting video or audio data). This type of data addresses human recipients, and the compression techniques exploit the insensitivity of humans for certain parts of the data and also make use of the fact that a human brain is to some extent able to overcome missing information (e.g., from the context or from past experiences) while stilling capturing the main information (recall Section 2.8.2). Moreover, humans tolerate a certain decoding delay which might occur from data buffering, e.g., in (online) video broadcasting. This might not be true for technical processes, which often rely on strict timing constraints. [RP06] analyzes the decoding times of MPEG-1/2 and 4 video and concludes that ungraceful quality degradation can arise during the decoding process due to improper resource management, e.g., caused by the fact that accurate worst-case decompression times can hardly be determined, since the decoder's resource usage highly depends on the video data itself. [Tek12, p. 7] highlights that an MPEG encoder might fail if the assumed redundancy between successive images (i.e., temporal compression) is nonexistent, e.g., recording a video of an event where a lot of flash photography is used.

Apart from that, some lossy compression schemes allow to guarantee fixed compression ratios below one, which is an important requirement for this work. Consider, for example, the Opus audio coding format (defined in the standards RFC 6716 [VVT12] and RFC 8251 [VV17] of the Internet Engineering Task Force (IETF)), which works with linear predictive coding and modified discrete cosine transform. If such multimedia data needs to be communicated in real-time via a channel with limited capacity, constant bit rate (CBR) encoding is advantageous as it can constantly utilize all available channel capacity.

A contrary strategy is variable bit rate (VBR) coding where complex signal parts are encoded with higher bit rates and simple parts with lower bit rates. This adaptation capability leads to an average bit rate (e.g., of a stored file) and, overall, to better data quality through more optimal utilization of memory capacity. In combination with buffering and a maximum bit rate, streaming with a variable bit rate becomes possible (e.g., consider the audio compression format Vorbis). For special applications such as medical imaging, where only a certain region of an image might be of interest for diagnostics, a variable image quality can be explicitly maintained within one image [Men06]. Yet, encoding data with VBR, the output file size (e.g., of an audio file) is not predictable. If one aims for a specific file size, a feasible way would be two-pass encoding, where data is analyzed before the actual encoding takes place.

Besides multimedia, there are safety-critical applications which require audio or video data to be encoded and transmitted in real-time with guaranteed quality and limited delay, e.g., for remote controlled vehicles, or for pilot to ground communication. In this context, evaluations on audio quality with respect to required bit rates and data delay have been conducted, e.g., in [Gay+04]. Such analyses yielded many compression algorithms that exploit the special signal characteristics of video and audio data (e.g., the use of key frames), which however, do not suit the type of data (i.e., time series of sensor measurements) that this work addresses. Yet, some researchers applied DCT to sensor signals in the context of data compression [Spa+17], however, the main objective was noise reduction before data was compressed by an arithmetic coder, which produces variable length outputs.

The above characteristics disqualify data compression based on transform coding in this work. It is mainly the trade-off between a more constant data quality (VBR encoding) and a constant bit rate with degraded data quality (CBR encoding), as well as the fact that technical processes (in particular safety-critical applications), as addressed in this thesis, do not tolerate and overcome degraded data quality and high decoding time delays as human recipients do. For a comprehensive overview including lossless video compression see e.g., [HCS07; SM10; Say17].

3.2.2 Entropy Coding and Dictionary Techniques

Considering the requirements from Section 3.1 regarding time-triggered systems, lossy data compression techniques are generally not suitable for encoding symbolic data (i.e., letters and numbers) in real-time. Looking at lossless data compression, a variable bit rate is mandatory. Otherwise, using a constant bit rate would lead to a contradiction; since the bit rate would then have to be as high as the source bit rate for error-free reconstruction, there would be no compression.

Entropy-based data compression schemes, such as Huffman codes, Golomb codes or arithmetic coding, assign variable-length output sequences to each (fixed-size) input by exploiting the fact that the input symbols show different probabilities of occurrence (recall Sections 2.8.3.2 and 2.8.3.3). These schemes work well for various types of data, e.g., images or text, especially if data gets compressed as a whole (i.e., a data file) and the main focus lies on storing, such that an average compression ratio is preferred. If the symbol occurrence probabilities are furthermore (at least initially) unknown, which is assumed in this thesis, only adaptive versions of the above entropy encoders can be potentially applied. For some techniques, like adaptive Huffman coding, this increases the computational complexity significantly, especially if the probability distribution changes over time [Vit87]. With respect to time-triggered systems with fixed-length periodic messages, predetermined transmission slots and the demanded guarantee for information to reach the receiver in bounded time, variable-length output compressors raise some problems.

Yoshida et al. [YK15] address the unclear boundary problem of variable-length codes for text compression, highlighting the difficulty to directly work on the data when the code word lengths are not constant. They propose to use Tunstall codes, which work similarly to Huffman codes but map variable-lengths input sequences to fixed-length code words. In this case, however, like in the dictionary approach LZ77 (Section 2.8.3.5), the number of required fixed-length output code words to represent a certain information is unpredictable. This conflicts with the requirement R4 from Section 3.1 that safety-critical applications demand guaranteed information transmission in real-time.

Many variations and optimizations of the original LZ77, LZ78, and LZW data compression schemes have been developed; [Lan13] gives an overview. In [KN10] the authors introduce LZ-End, an improvement to LZ77 which enables to decompress arbitrary phrases independent from knowing the beginning of the decoded data. The LZ78 and LZW dictionary-based data compression techniques exploit redundancy in data in terms of encoding repeating sequences of symbols with dictionary keys (i.e., indices) that consume fewer bits than the original symbol sequence would need (recall Section 2.8.3.5).

In general, entropy encoding techniques as well as dictionary-based approaches aim for an average compression ratio, but in a time-triggered system there would be no direct benefit from occasionally occurring shorter code words, and for safety-critical applications, inconstant information delivery is not acceptable. So, in addition to R4, there is also a clear contradiction with the requirement R2, which demands that the compression algorithms support a WCCR on short input sequences.

3.2.3 Differential Encoding and Predictive Encoding

Delta encoding, as a form of differential compression, exploits the fact that some target data can often be stored with less bits in the form of differences with respect to source data (i.e., a reference). This is very useful in incremental text savings, e.g., for revision control. With a view to the data to be compressed in this work, i.e., numbers of finite intervals, delta encoding is a feasible way to make use of the expected small differences between consecutive data values. The representation of such correlated data values in terms of differences leads to a reduced variance and dynamic range [Kol+12; Say17, p. 325], an advantageous condition for subsequent lossy or lossless encoding strategies. It needs to be considered that if a compressor must guarantee to encode all potential differences between any two consecutive values with a fixed number of bits for error-free value reconstructions, no bits can be saved. Moreover, delta encoding must ensure to always have a reliable reference to which a difference is related, e.g., by periodically transmitting synchronization sequences to prevent a potentially infinite number of corrupted data values. In the context of wireless sensor network (WSN), the idea of delta encoding is implemented in the LEC-compressor from [MV09] and [VGM14]. Contrary to the requirements from Section 3.1, the LEC-compressor is a lossless entropy-based encoder, which makes use of the highly skewed probabilities of occurrence of the difference values and obviously does not show a fixed compression ratio. Nevertheless, WSN show some similarities to the systems and data addressed in this thesis, mainly the distributed character and the limited resources, which lead to the common objective to decrease message sizes for the communication of a similar kind of data. The above LEC-compressor from Marcelloni and Vecchio can be seen as a representative for many other implementations where differential encoding is combined with entropy encoding. In many other works that address data compression techniques in WSN, the special focus mostly lies on the energy constraints of these systems, in particular on the communications unit [SM06; Kol+12]. So the goal of those works is to compress data, which was gathered at the sensor nodes over a certain time interval, in order to decrease the amount of bits (or the number of required messages) for a subsequent transmission [Sri+12; Kol+13; Kol+15]. This implies that time-sensitivity only plays a minor role, which is significantly different to the requirements R1 and R3 from Section 3.1, so that the presented strategies can only be partly adopted.

Differential pulse-code modulation (DPCM) can be seen as an extension to simple delta encoding between consecutive samples as described above. It employs a local model of the decoder process and predicts every next sample from some previous samples. The difference value to be encoded is then calculated from the prediction and the real sample, which is often smaller than the difference between two samples, and thus helps to minimize the quantization error [Pro+02, pp. 308–309]. The adaptive variant (ADPCM) offers additional scaling of the quantization levels with respect to the signal. The goal is to have finer quantization for smaller differences and coarser quantization for larger differences, respectively, see e.g., the companding algorithms A-law and μ-law used in digital communication systems defined in the ITU-T G.711 standard [ITU-T00].

The above difference coding strategies work well for individual sampled signals where consecutive samples show small differences, or where such differences can be well predicted and a potential error can be tolerated for some time, e.g., in speech coding. They have also been implemented for other domains, such as prediction-based image compression, see e.g., [Kum+13]. They can be easily combined with additional data compression strategies and consequently produce fixed-length or variable-length code words and show small quantization errors if input sequences behave as expected. Lossless as well as lossy data compression is supported. The cost for saving bits is that either not all possible difference values can be encoded, or that the uncertainty for a correctly reconstructed value in case of large sample differences is high. This conflicts with the requirement R4 from Section 3.1 of accurate value reconstructions for safety-critical applications. The combination of lossless difference coding with an entropy encoder obviously conflicts with the same requirements as explained in Section 3.2.2.

Predictive encoding is strongly related to difference encoding. Approaches such as linear or adaptive predictive coding maintain a statistical model to predict future observations (e.g., of sensors) and save data by only storing or transmitting new data if it deviates from the prediction, typically in an event-triggered manner, see e.g., [Des+04]. Predictive encoding works especially well in speech coding since the underlying prediction models are able to predict human voice with low error [DO03, p. 41]. Predictive coding approaches are frequently combined with transform coding, e.g., in [HSR10] (lossy compression).

The adverse possibility that an (adaptive) predictor does not properly adapt at some point, which might lead to dangerous situations in safety-critical applications due to large errors in subsequent values, in combination with the compression gain based on an event-triggered transmission strategy, contradicts the requirements from Section 3.1 that data samples need to be made available to the respective tasks reliably in short and bounded time.

3.2.4 Distributed Source Coding

In [Sri+12], the authors present a survey on practical data compression in WSN and distinguish between distributed data compression approaches and local data compression approaches to exploit spatial and temporal correlation, respectively. Besides an evaluation of achievable compression rates, algorithm complexity, and energy consumption with a special focus on low-power mobile devices, the problem of compressing correlated data streams from multiple sources which do not communicate with each other is highlighted. In the field of information theory, this is known as distributed source coding (DSC) [DG09]. DSC can be used for lossy as well as for lossless data compression. In the discussed application scenario of [Sri+12], the technique helps to shift computational complexity for compression from low-power encoders to a joint decoder. [RBD13] reviews DSC to be a feasible tool to make use of multiple spatially correlated data streams in sensor networks for data reduction purposes. According to the theoretical foundations of [SW73] and [Cov75], multiple correlated data sources (e.g., sensors at different locations in a network) that cannot directly communication with each other can encode their data advantageously if they are jointly decoded at one sink that has knowledge about the sources' statistical dependence.

Depending on the implementation, DSC is capable of providing real-time guarantees and a WCCR below one for short input data sequences. Although this approach is mentioned to be promising for many application scenarios, limitations include an often insufficient prior knowledge of the data correlations at the different sensors in real systems, which lowers its effectiveness, as well as a lack of robustness and scalability [RBD13]. Furthermore, the fact that the correlation between data sources typically varies over time implies an iterative exchange of the relevant information between the sources and the decoder [ANH16]. With a view to the architecture addressed in this theses and the requirements from Section 3.1, a frequently used side channel would contradict requirement R5 and the possibility that improper statistical knowledge results in degraded data quality conflicts with R4. Another particular drawback of DSC is the need for a joint decoder.

3.2.5 Off-the-Shelf Algorithms Versus Application-Specific Compression Algorithms

In the context of compressing sensor data in distributed systems, Sadler et al. [SM06] state a general trade-off between using off-the-shelf algorithms such as LZW, which are however not designed for special applications, signals, or hardware, and developing better performing application-specific compression algorithms, which is usually time-consuming and costly. The authors came up with a dictionary-based compression approach called S-LZW (i.e., LZW for sensor nodes) and an improved version S-LZW-MC, which maintains a mini cache and is optimized for sensor data characteristics like repetitive sample sequences over short intervals. It is evaluated in terms of energy consumption and compared with other compression schemes. However, none of the algorithms presented supports a fixed compression ratio as required according to Section 3.1 (requirement R2).

Another approach is called coding by ordering and was originally introduced in [Pet+03] for distributed data gathering networks. It reduces the amount of communication needed to transmit sensor readings from multiple sources to a common data base by aggregating data of multiple sources within certain regions of interest before those data is combined and forwarded via a lossless reordering and compression scheme. Pipelined in-network compression [Ari+03] is strongly related to the above approach. It specifically exploits signal commonalities to reduce redundancy in the temporarily buffered data of multiple sensors in order to minimize the amount of wireless communication for energy saving purposes. In their compression scheme, higher latency is traded for lower energy consumption, which does not comply with the requirements from Section 3.1, particularly with respect to the demanded real-time capabilities (R1).

3.2.6 Data Compression and Scheduling

Luo et al. [Luo+18] bring together the topics data compression and communication scheduling in data gathering networks. Their work is based on theoretical foundations of [Ber15]. The assumed network topology in both papers consists of many data sources (e.g., sensors) and one data sink (i.e., a data base), a typical constellation of WSNs. A time-division multiple access (TDMA) scheme is applied to organize the data traffic. The authors aim for minimizing the length of the schedule (i.e., the makespan), which refers to the periodic time needed to transfer all data from the sources to the sink. In [Luo+18], an optimization problem is formulated; data compression and decompression, respectively, need additional processing time at the sender and receiver, but allow for shorter messages, which helps to reduce transmission time.

None of the above two papers presents applicable data compression schemes. All considerations that influence the cost-value ratio (e.g., compression and decompression times, compression ratios) are treated as parameters. It needs to be further analyzed whether available compressors can fulfill these conditions. [LGL18] extends the above evaluations for heterogeneous scenarios where different sensors have different data compression ratios, compression times, and compression costs in terms of energy consumption.

A similar evaluation metric considering the benefits and costs of data compression with respect to time savings in time-triggered systems is used in [MO19]. The work assumes a more sophisticated network topology where data is communicated multidirectional among several computation nodes according to a logical task dependency model. The authors presume that pairs of jobs which produce correlated data are best allocated to the same computation nodes so that their outputs can be advantageously combined (i.e., compressed to shorter messages). For their evaluations, they assume variable parameters for compression ratios and show that their approach helps to minimize the length of a schedule. In their model, however, combining messages is only possible if the individual data streams have the same source node and destination node, respectively. The paper does not present any compression scheme which meets the stated stringent real-time requirements with respect to WCCT, WCDT, and WCCR (Section 3.1). Nevertheless, the scheduling model and the evaluation metric are valuable and can be adopted and extended for this thesis.

3.3 Summary of the Requirements and Related Work

In summary, none of the reviewed works presents data compression techniques that satisfy all requirements from Section 3.1. Communicating symbolic data in time-triggered systems primarily demands lossless data compression techniques which offer WCCRs on short input sequences or constant bit rate compression. Since these are contradicting goals, a major challenge lies in maximizing the overall benefit in the context of relevant application scenarios. This thesis addresses the lack of applicable online data compression techniques from the state of the art and presents an online data compression algorithm (including several extensions) that is in accordance with the requirements from Section 3.1.

Table 3.1 shows the capabilities of the newly designed online data compression algorithms compared to related works. The table abstracts over individual papers and summarizes the state of research and implementations by their underlying data compression techniques. An in-depth evaluation of the related works is presented in Section 3.2. In the table, a check mark reflects the ability of the compression technique (in at least one implementation possibility) to satisfy the respective requirement, and indicates that corresponding related works are discussed in Section 3.2. A check mark in parentheses indicates that while it might be technically possible for the data compression technique to support the requirement, no related work applies the compression in the way needed within the research scope of this thesis. A dash indicates that the particular data compression technique is disqualified for the requirement for one or more reasons as discussed in Section 3.2. The requirements in the first column are labeled from R1 to R8, each with a short form description. The complete definition of the requirements can be found in Section 3.1.

	Transform coding	Entropy coding	Dictionary techniques	Differential encoding	Predictive encoding	Distributed source coding	This work
R1: Real-time capabilities (WCCT, WCDT)	(✓)	✓	✓	✓	✓	✓	✓
R2: WCCR below 1 on short input sequences	✓	-	-	(✓)	(✓)	✓	✓
R3: Online (one-pass) compression (i.e., short input sequences)	-	✓	✓	✓	✓	✓	✓
R4: Guaranteed data quality with respect to time-triggered messages	-	-	-	(✓)	(✓)	-	✓
R5: Overhead information included in data stream	✓	✓	✓	✓	✓	-	✓
R6: No prior statistical information necessary	✓	-	✓	✓	✓	-	✓
R7: Combined compression of multiple data streams	-	-	-	-	-	(✓)	✓
R8: Support for branching data streams in distributed systems	-	-	-	-	-	-	✓

Table 3.1: Overview of the state of the art.

Chapter 4

DAKODIS Architecture

4.1 Architecture Overview

This chapter defines a time-triggered architecture which supports compressed communication. It is named *DAKODIS architecture*.

In various application domains, electronic and mechatronics systems are implemented as distributed systems with numerous nodes, e.g., computation nodes, sensors, and actuators. The communication among all these nodes is realized via messages through a network of routers or via data bus structures. Particularly in safety-critical application domains, where online diagnosis techniques with fault tolerance strategies often contribute to an increased system reliability and availability, the data exchange between different nodes of such a network is time-sensitive and often a bottleneck. In this regard, online data compression helps to reduce the data size for transmission. However, major challenges become apparent when considering the contradictions of lossless and lossy online data compression techniques in the context of the time-triggered paradigm. One-pass lossless data compression does not provide guaranteed output sizes. Considering the timed messages, this might lead to incomplete information delivery, which conflicts with safety-critical applications. In contrast, lossy data compression achieves fixed-size outputs, but has the disadvantage that data quality cannot be guaranteed.

Time-triggered systems for safety-critical, high-dependable, or mixed criticality systems have been investigated by many researchers and numerous research projects. For instance, [Obe11] provides a detailed foundation of time-triggered communication and [BD19] reviews up-to-date mixed criticality systems.

With support for compressed communications, the DAKODIS architecture is designed to integrate the online data compression algorithms presented in Chapter 5, and is thus the first step towards tackling the open research challenge. This chapter formalizes the different models that are part of the DAKODIS architecture, namely the physical model, the logical model, the compression model, and the scheduling model. The physical model defines the physical resources of the system, i.e., it describes the nodes as well as the links between these nodes. The logical model specifies the application, e.g., the number of tasks with relevant information such as worst-case execution times (WCETs) and task dependencies in terms of messages with size, source and destination information. In the literature, some authors specifically define and distinguish the terms *task* and *job*, e.g., [CK07] denotes a job as an instance of signal processing, computation, or the like, and a task as a sequence of jobs with similar characteristics and timing requirements. In contrast, in [LM16] a job means a collection of computational tasks. In many other cases in the literature, such a hierarchy is omitted and the two terms are used synonymously and imply a sequence of processing steps. In this thesis, only the term *task* is used. The compression model holds information about worst-case compression times (WCCTs), worst-case decompression times (WCDTs), and worst-case compression ratios (WCCRs) for the messages.

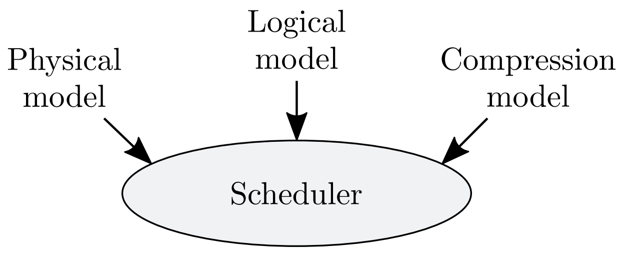


Figure 4.1: Architecture overview.

As illustrated in Figure 4.1, all those models are input to a scheduler (i.e., a scheduling algorithm), which computes a scheduling model.

4.2 Physical Model

In Time-Sensitive Networking (TSN) prominently Ethernet-based networks are employed. TSN is gaining more and more importance for applications that require low latency and high availability for data transmissions, and thus plays an important role in industrial communication and automation systems [LS19]. This thesis therefore assumes a general network topology based on routers. Special topologies such as bus-based networks are not in the focus.

Let C be a set of computation nodes and let R be a set of routers with $C \cap R = \emptyset$. A network is then represented by a *simple graph* $Net = (V, L)$, with $V = C \cup R$ being the vertices (i.e., *nodes*) of the graph and L being a subset of $\mathcal{P}_2(V)$, called the edges (also referred to as *links*) of the graph. This means L is a set of two-element subsets of V and is an undirected edge relation. Such a simple graph does not allow multiple edges to have the same pair of endpoints (the endpoints refer to the vertices of an edge) and it is also not allowed to contain loops (i.e., edges that connect a vertex to itself) [BW10, pp. 148–150]. It is further defined that $L \subseteq \mathcal{P}_2(V) \setminus \mathcal{P}_2(C)$. This definition states that computation nodes can be directly connected to one or more routers but not to any other computation node. Routers can be connected to multiple other routers. An edge (i.e., a pair of nodes) is in L if and only if there is a connection between them. It is defined that only computation nodes can execute tasks and only routers are able to forward messages. Routers can additionally perform specific operations to merge and split compressed data streams while these are communicated through the network.

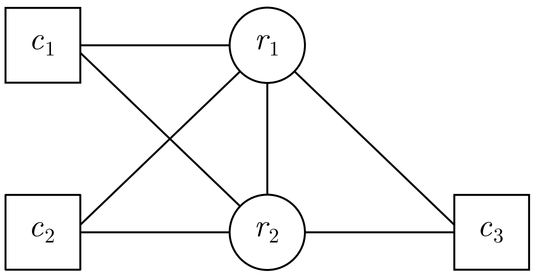


Figure 4.2: Distributed system with three computation nodes and two routers.

Figure 4.2 exemplarily shows a network with three computation nodes (c_1, c_2, and c_3) and two routers (r_1 and r_2). A computation node is equipped with all necessary hardware resources (e.g., memory, processor) and operating software to execute tasks as defined by the logical model. The network can consist of different kinds of specialized computational hardware such as field-programmable gate arrays (FPGAs) or graphics processing units (GPUs). Different capabilities of computation nodes are reflected in the logical model, e.g., by indicating for each task to which nodes it can be allocated.

4.3 Logical Model

4.3.1 Directed Acyclic Graphs

A service of a system is accomplished through an interaction of multiple tasks. The main elements of the logical model are tasks and messages. The logical model formally specifies a set of tasks $T = \{t_1, t_2, \ldots, t_n\}$ (with $n \in \mathbb{N}$) and models their dependencies by means of a directed acyclic graph (DAG). Let this graph be defined as $G = (T, E)$ with

$E \subseteq \{(t,t')|(t,t') \in T^2 \text{ and } t \neq t')\}$ being a set of ordered pairs of vertices which model a precedence relation between two tasks. The directed edges $e \in E$ are called (*logical*) *channels*. According to the definition of G (without an incidence function), multiple edges with the same pair of endpoints are not possible [BW10, p. 148]. For the graph to not contain cycles, one requires that for a sequence of edges $e_1, e_2, \ldots, e_{n-1}$ (with $n \in \mathbb{N}$), there is a sequence of distinct vertices $t_1, t_2, \ldots, t_n$ so that $e_i = (t_i, t_{i+1})$ for $i = 1, 2, \ldots, n-1$ [BW10, p. 162].

The length of a message in bits that is sent via a channel e is denoted as $\ell_e \in \mathbb{N}$. According to the requirements from Section 3.1, the compression scheme can act on individual data values. In an ideal scenario, the length of a data value is equal to the length of a message. Then a stream of data values is sent over a channel e and every data value is encoded using ℓ_e bits. In a DAG, all predecessors and successors of any task are clearly identifiable. Task t' directly depends on task t if and only if $(t,t') \in E$. A task may depend on multiple others and in turn may produce outputs as a prerequisite for other tasks. Each task from T has a unique identifier $i \in [1,n]$ and shows a WCET_i, which is the time it takes to complete in the worst case. Depending on the computation capabilities of a node, the WCET of a task can vary, respectively. A list of nodes on which a task can be executed is presented by the system designers with the graph.

The directed acyclic graph in Figure 4.3 exemplifies the dependencies between the tasks $t_1, \ldots, t_5$ via the messages $m_1, \ldots, m_6$. In this illustration, information such as the WCETs of the tasks and the lengths of the messages are neglected.

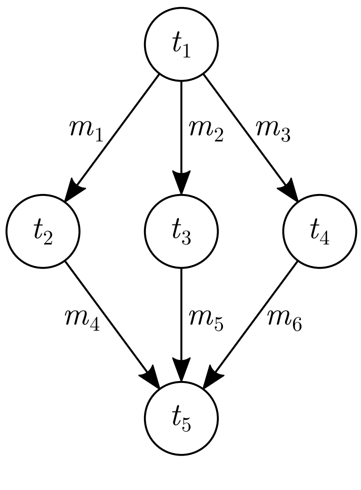


Figure 4.3: Directed acyclic graph with five tasks.

4.3.2 Data Streams

Throughout this thesis, the terminology *data stream* implies that data is constantly and periodically produced at a source, e.g., data samples from a sensor or other portions of numeric data at a node. According to the DAG from Section 4.3.1, such data needs to be communicated among computation nodes of a network where the relevant tasks are executed. Its transmission is actually realized in the form of time-triggered messages.

4.3.3 Applications

An application that provides a particular service can be viewed as a composition of tasks, with the tasks being instances of signal processing or computation. An application is formally described by a logical model, i.e., a DAG, as introduced in Section 4.3.1. Examples are control applications and diagnosis applications. This thesis has a focus on diagnosis applications, so some specific properties in this regard will be addressed in the following.

Diagnosis Applications

Fault diagnosis is a key element in fault-tolerant systems with dynamic redundancy (Section 2.6.1), as it significantly contributes to improving the system reliability. Online fault diagnosis implies that its associated tasks are executed in parallel with the tasks of the regular system application and might compete for computation and communication resources. The generation of a logical model for a diagnosis application is highly application-specific. For simplicity, here and in the following, let such a specific logical model be denoted as *diagnostic model.* A diagnostic model for a use case scenario is presented in Section 6.4.

Diagnosis Techniques

A variety of different fault diagnosis approaches and techniques exist (recall Section 2.7), which vary in their suitability for different applications. Essentially, fault diagnosis utilizes *explicit* or *implicit* knowledge about the system or the process being diagnosed in order to find the root cause of faulty system behavior. [Hen+14] and [Ise11] review diagnosis techniques for common industrial applications where the underlying diagnostic knowledge is explicitly provided by human system engineers in the form of signal thresholds and modeling equations. In [Lei+20], the recent trend towards preparing implicit knowledge about the process for fault classification with machine learning algorithms is reviewed. It is also possible to combine multiple fault diagnosis techniques within a DAG in a task-based manner [MO18].

Explicit Modeling

When explicitly modeling a diagnostic inference process by means of a DAG according to Section 4.3.1, the system infrastructure influences the encapsulation of processing steps into diagnostic tasks. For example, significant communication overhead between tasks can be avoided by taking into account the often different locations at which sensor data is gathered and naturally preprocessed in a distributed system. In explicit diagnostic modeling, the following guidelines help to design diagnostic tasks.

- A diagnostic task produces characteristic information for fault detection and fault inference by means of fault diagnosis methods (Sections 2.7.2 and 2.7.4).
- A task may comprise multiple processing steps.
- For the order of fault reasoning (i.e., searching for confirming or disconfirming information for a specific fault), several factors are considered, such as the probability of a fault's occurrence and the computational cost of the diagnostic operation required.
- Fault reasoning and intermediate conclusions are allowed in every subgraph.

Implicit Modeling

The generation of a diagnosis model from implicit knowledge requires the availability of a large amount of data of the system to be diagnosed, which reflects the system behavior in various faulty and fault-free situations. Machine learning techniques are then able to extract a model for fault classifications.

It is apparent that the use of implicit knowledge for designing diagnosis models for modern highly complex systems has significant advantages over human-driven approaches (e.g., in terms of time and cost), presupposing that sufficient process data is available for learning algorithms. For example, considering a large-scale scenario, a fault inference process might integrate multiple neural network classifiers as individual tasks.

In the context of the DAKODIS research project, [Mec+20b] presents the design of a diagnosis model from implicit knowledge for a use case scenario. Besides the well-known classification capabilities of the two different approaches, (i) decision tree model and (ii) convolutional neural networks, the authors highlight the possibility to derive a diagnostic DAG from a machine learning model.

4.4 Compression Model

The DAKODIS architecture supports different compression schemes. In order to provide real-time guarantees, the WCCT for compressing data values, the WCDT for decompressing compressed data values, and the WCCR must be known for all those schemes (e.g., compression of individual data streams, simultaneous compression of multiple data streams).

Formally, a data compression scheme is a tuple $Z = (\ell, k, \tau_c, \tau_d)$. With respect to the requirements from Section 3.1, the compression is performed on preferably short pieces of information, so $\ell \in \mathbb{N}$ is the length of an input data value in bits, $k \in \mathbb{N}$ is the maximal length of a compressed data value in bits, $\tau_c \in \mathbb{N}$ is the WCCT, and $\tau_d \in \mathbb{N}$ is the WCDT, both in terms of (clock) *ticks*. Here, a tick refers to a fraction of real time and is used to express a duration of time that abstracts over a real unit of time, such as microseconds. A compression scheme abstracts from a concrete compression algorithm that receives a sequence of ℓ-bit strings and transforms it into a sequence of bit strings of length at most k. The time needed to convert a single ℓ-bit string into its compressed output (a bit string of length at most k) is at most τ_c, and the time needed to recover the original ℓ-bit string from the output is at most τ_d. A compression scheme is applicable to a channel as defined in Section 4.3 if the bit length ℓ_e of the data values transmitted via the channel e is ℓ. The WCCR is k/ℓ and one aims for $k < \ell$ to benefit from compression. As this is clearly not possible using lossless compression (recall Section 2.8.3.1), this thesis implements compression algorithms with $k < \ell$ which allow error-free value reconstructions with a high probability. In rare cases, which depend on the signal characteristics, the accuracy of data values is lowered but still bounded.

For each message as defined by the logical model, the scheduling model prepares information about the achievable WCCR, WCCT, and WCDT, respectively. These times are added to a task's WCET if the scheduler decides to compress data.

Figure 4.4 highlights the benefits of compressed communication in terms of time savings. In the example, task_n transmits the message msg_m to task_{n+1}. The WCETs of the tasks and the transmission time of the message are visualized by means of a bar chart. From a theoretical viewpoint, the transmission time of a message depends on the path length (in the network) and the per-hop transmission time, where the latter is directly related to the message size (see Section 6.4). In order to generate time savings with respect to a schedule, the reduction of transmission times of compressed messages must be more significant than the extra times needed for data compressions and decompressions, respectively, see Figure 4.4. For practical implementations where specific time-triggered communication protocols are used, the benefits of compressing data become apparent when systems scale up and the data frames (according to the protocol) are filled with data from multiple tasks.

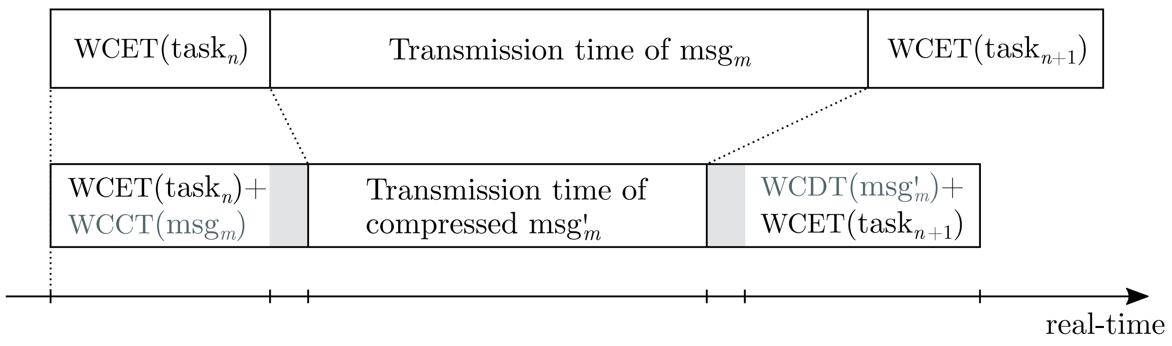


Figure 4.4: Compressed communication extends the WCETs of tasks and reduces communication times of messages.

Then compressed data contributes to an improved frame packing, which in turn helps to optimize schedules, particularly to decrease the makespan. In this way, tighter deadlines can be satisfied and shorter periods for the overall service can be provided. Moreover, a reduction of network traffic allows for a higher level of integration of services in a system.

4.5 Scheduling Model

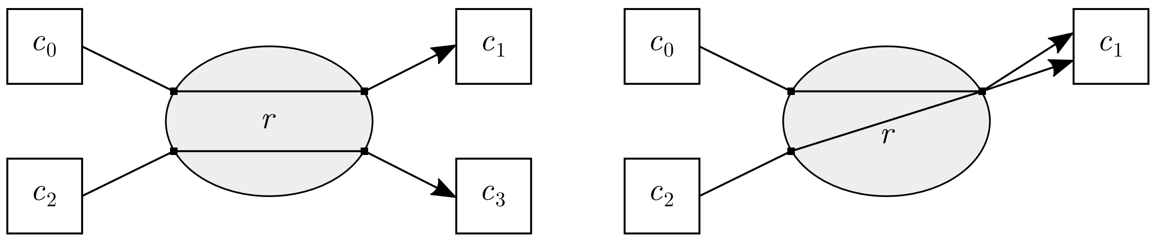
A schedule basically provides information about the allocation of tasks to computation nodes, the execution times of those tasks, the injection times of messages and the routes that those messages take through the network. All this has to be managed without violating resource restrictions and dependency relations. The scheduling model of this thesis is based on [MO19] and [LO17]. Both works were published in the context of the DAKODIS research project [ES21]. The former work introduces a scheduling model which also supports combined data compression of outputs of pairs of tasks using a genetic algorithm. The latter publication presents a simpler version of the scheduling model with only one compression algorithm and equal overheads for compression and decompression for mixed-integer linear programming (MILP) solvers.

A nonpreemptive static scheduling model is used for the proposed architecture. Such a scheduling model has advantages for multicore systems and distributed systems, as the overhead for managing tasks is easier to predict compared to preemptive scheduling [Gua16]. The disadvantage of nonpreemptive tasks is reduced system responsiveness, which is typically not relevant for multicore systems, since the natural parallelism of such a system can hide this latency [Gua16].

Tasks are mapped to computation nodes by means of an allocation function, which is a mapping $A : T \to C$. This also yields the distance of tasks in the network and has impact on the time needed for communicating messages between tasks, and consequently on the makespan. For every channel $e = (t, t') \in E$ and $A(t) \neq A(t')$, there must be a path of the form $A(t), r_1, \ldots, r_n, A(t')$ in the network *Net*, where $r_1, \ldots, r_n \in R$ with $n \geq 1$, $(A(t), r_1) \in L, (r_i, r_{i+1}) \in L$ for all $1 \leq i \leq n-1, (r_n, A(t')) \in L$, and $r_i \neq r_j$ for $i \neq j$. For further considerations, let such a path be fixed and denoted as $A(e)$. Every path $A(e)$ then has the following properties relevant for scheduling:

- it starts and ends with a computation node,
- only routers are allowed between computation nodes,
- it does not have cycles, i.e., a node does not appear multiple times on the path,
- its length (i.e., the number of nodes on the path) is between 3 and $|R| + 2$.

The use of time-division multiple access (TDMA) avoids conflicts of potentially intersecting paths in one or more nodes. In this regard, message buffering at the routers with respective tables or the use of multiple ports at routers can be beneficial. Possible strategies to avoid message collisions include temporal separation and spatial separation. In the former case, messages are intentionally delayed at a sender so that they arrive later at a shared resource, and in the latter case, a conflict free path is sought.



a) Nonconflicting communication   b) Conflicting communication

Figure 4.5: Communication at routers via ports.

An example for spatial separation is illustrated in Figure 4.5a. The communication between the nodes c_0 and c_1 does not conflict with the communication between c_2 and c_3 at the router r, even if two messages arrive at the same time, because the scheduler assigned independent sets of ports (as indicated by the filled black squares). In contrast, if the use of ports is not scheduled, conflicting communication might occur, as highlighted in Figure 4.5b [Jo+17].

The DAKODIS architecture adds the utilization of data compression to classical scheduling problems such as resource allocation and routing. As stated in Section 4.4, the compression schemes provide real-time guarantees in terms of WCCTs and WCDTs as well as a guarantee on the WCCR. If the compression scheme $Z = (\ell, k, \tau_c, \tau_d)$ is used for a channel e (with $\ell_e = \ell$), then the number of transmitted bits is reduced by $\ell - k$ for every single data value. On the other hand, the costs τ_c and τ_d have to be added to the execution times of the channel's end points. Thus, the usage of data compression is a trade-off between less communication and longer execution times (recall Figure 4.4).

Chapter 5

Online Data Compression for Time-Triggered Communication

This chapter focuses on online compression algorithms for the DAKODIS architecture and is based on the formal compression model from Section 4.4. The compression requirements arise from the time-triggered architecture and the diagnostic applications as discussed in Section 3.1. Specifically, newly designed dictionary-based data compression schemes are presented where the dictionary is implemented as a cache with common replacement strategies, e.g., least recently used (LRU). The developed algorithms are therefore referred to as cache-based compression algorithms in the following.

5.1 Compression of Individual Data Streams

5.1.1 Cache-Based Compression Algorithm for Individual Data Streams

In compressing an individual data stream (recall the definition of a data stream from Section 4.3.2), the goal is to compress a sequence of n-bit data values (for a fixed $n \in \mathbb{N}$) that are produced by a task running on a computation node. Such data originates from sensors measuring physical quantities and it is assumed that these data values exhibit some locality for limited time intervals. There are $N = 2^n$ possible data values. For the compression, every n-bit data value is split into s high-order bits (called *head*) and the remaining $t = n - s$ low-order bits (called *tail*) for some $s, t \geq 1$ with $s \leq t$. Then every value $i \in [0, N-1]$ can be encoded by a bit string of length $\ell := s + t$ by taking the ℓ-bit binary expansion of i, which is denoted $\text{bin}_\ell(i)$ in the following. Such a bit string is called a *code word*. Due to locality of consecutive data values, the heads of consecutive values are expected to have little variation in a limited time interval.

The bit strings of length s (the heads) can be identified with numbers from $[0, 2^s - 1]$, which allows to make arithmetic calculations on heads. For example, if $s = 4$ and $t = 8$, then the head and tail of the code word 1001 01110101 are 1001 and 01110101, respectively. The head $u = 1001$ corresponds to the number $2^3 + 2^0 = 9$ (the left-most bit is the most significant bit (MSB)) and the head $u - 1$ (resp., $u + 1$) is 1000 (resp., 1010). Note that for every fixed head $u \in \{0,1\}^s$, the set of code words $\{uv \mid v \in \{0,1\}^t\}$ corresponds to an interval $[u \cdot 2^t, (u+1) \cdot 2^t - 1] \subseteq [0, N-1]$ of data values. Referring to the aforementioned example ($s = 4, t = 8$), there are $2^{s+t} = 4096$ data values covered by 16 heads comprising 256 entries each.

The compression of the $n = s + t$ bits in a data value to $r + t$ bits (for some $r < s$) is realized by a dictionary D that stores the $2^r - 1$ most recently seen heads at dictionary entries $D[p]$. The dictionary index p is an r-bit code different from the reserved sequence 0^r (r 0-bits; this notation is used throughout this thesis), i.e., $p \in \{0,1\}^r$ with $p \neq 0^r$. The index p is also referred to as a *compressed head*. Initially, the dictionary is either empty or filled with some heads that are known to frequently occur in a data stream. The heads that belong to the dictionary are called *active heads*. Both the sender and the receiver store the same dictionary at every time instant. The parameters $N, s, t, \ell = s+t$ and $r < s$ are assumed to be fixed for further considerations.

Input data is now compressed as follows. The algorithm encodes the s bits from the head to r bits and transmits the t bits from the tail uncompressed. In detail, consider an n-bit input data value $i \in [0, N-1]$ and let $x = uv$ be the corresponding code word, i.e., the ℓ-bit binary expansion of i ($u \in \{0,1\}^s$ is the head and $v \in \{0,1\}^t$ is the tail, with $|u| = s$ and $|v| = t$). If the head u is stored in the dictionary at entry $D[p]$ (this can be checked in constant time by implementing the dictionary using a hash table; see Section 5.2.2 for an example implementation of a hash function), then the $(r+t)$-bit code pv is transmitted. In this way, the algorithm saves $s - r$ many bits. Otherwise, a so-called *miss* occurs, which is indicated to the receiver by the bit sequence 0^r (which is not part of the dictionary). The sender then transmits the bit sequence $0^r u$ (recall that $s \leq t$, so $0^r u$ fits into $r + t$ bits). The prefix 0^r tells the receiver that the next s many bits represent a new head u. Also, the sender and receiver update their dictionaries by computing a dictionary index $p = \text{fresh}(D)$ and setting $D[p] := u$. Here, $\text{fresh}(D)$ is the index of a free dictionary entry or, if the dictionary is fully populated, an index computed by a replacement strategy (the LRU strategy is used throughout this work; other strategies such as least frequently used (LFU) are also supported).

In a miss case, the data value is not completely lost, but the sender and receiver reconstruct a data value from the transmitted head u and the bit sequence $1\,0^{t-1}$ as the tail (starting with the MSB), which corresponds to the center value covered by the head. This means that the accuracy of the reconstructed value is only slightly lower, i.e., the uncertainty of the correct code word is half of its potential range of values ($2^t/2$) of the overall code word space of $N = 2^n$ values. This ensures that the main signal characteristics are always captured with the compression algorithm and that the information loss of a data value is limited.

The *compression ratio* is defined as the quotient of the length of the encoded bit sequence divided by the length of the input data value and is at most $(r+t)/(s+t)$. Hence, a smaller compression ratio means better compression. Due to locality in the data values, a small number of misses over time is expected.

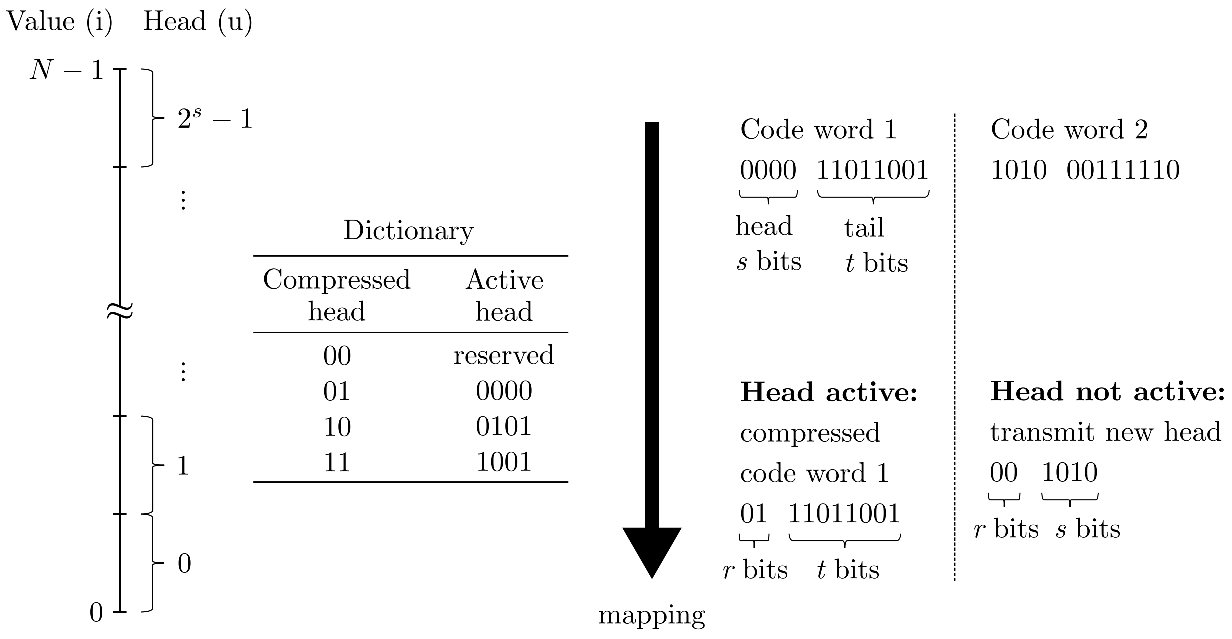


Figure 5.1: Cache-based algorithm procedure; example with $s = 4$, $t = 8$, $r = 2$.

Figure 5.1 graphically demonstrates the transmission procedure for two code words. The size of the dictionary is three and currently the heads 0000, 0101, and 1001 are active. The first code word 0000 11011001 is successfully encoded as its head 0000 is active at $D[01]$. Since the head 1010 of the second code word is not active (miss case), the head is communicated as the tail of the compressed code word starting with the prefix 00, i.e., the reserved compressed head. A more detailed example with more code words for the cache-based algorithm including some algorithm enhancements is covered in Section 5.1.3.

Algorithm 1 describes the pseudocode of the cache-based compression algorithm, see also [Mec+19b; Jo+18]. In a success case, the combination of the applicable dictionary index p and the uncompressed tail of the code word, v, is communicated (line 6). In a miss case, the compressed code word is formed from the prefix 0^r and the missed head u (line 8). Note that the sender and receiver keep their dictionaries synchronized purely based on the basis of the transmitted data values according to the LRU update strategy (lines 9–10). In accordance with the requirements from Section 3.1, no side channel is needed.

Algorithm 1 Cache-based compression algorithm for 1 data stream

```
1: input : data value $i \in [0, N-1]$
2: output : bit string of length at most $r + t$
3: initialize dictionary $D$ as empty hash table of size $2^r - 1$
4: let $x = uv = \mathrm{bin}_\ell(i)$ with $|u| = s$ and $|v| = t$
5: if there is $p$ with $D[p] = u$ then
6:     send $pv$ to the receiver
7: else
8:     send $0^r u$ to the receiver
9:     $p := \mathrm{fresh}(D)$ // use an index of a free dictionary entry
       or compute a new index based on the LRU strategy
10:    $D[p] := u$ // insert $u$ to $D$
11: end if
```

In summary, the main features of the compression algorithm are:

- For each data value, a fixed compression ratio of $(r + t)/(s + t) < 1$ is achieved. This is in contrast to classical lossless compression, where only statements about the average compression ratio are possible. A fixed compression ratio is important for time-triggered architectures as expressed in requirement R2 from Section 3.1.
- To allow for a fixed compression ratio < 1, occasional misses of data values must be accepted. In terms of the introduced compression algorithm, a miss means that the uncertainty of the transmitted data value is slightly increased, but bounded. A small number of such miss cases is acceptable for many (diagnosis) applications, since the essential signal characteristics are still captured and diagnostic results are typically not based on single samples. Furthermore, knowing the maximum uncertainty still makes a reconstructed value in a miss case useful for many applications.
- The cache-based compression algorithm, unlike classical lossy data compression, transmits those data values that are not lost without any degradation in accuracy. Overall, this complies with requirement R4 from Section 3.1 that the application can rely on guaranteed data quality.

5.1.2 Algorithm Enhancements – Reducing Uncertainty and Miss Rate

The requirements for the DAKODIS architecture stated in Section 3.1, particularly with regard to ensuring data quality for the diagnosis applications, allow only occasional misses. The goal is to keep these events as rare as possible, yet retaining as much information as possible about missed code words. The following sections present two improvements to the basic algorithm from Section 5.1.1. These improvements can be applied if $s < t$ (recall that $s \leq t$). This is a reasonable assumption: choosing t too small means that the heads only comprise as few as 2^t tails, which likely leads to a higher miss rate.

5.1.2.1 Reducing the Uncertainty

In Algorithm 1, in the case of a miss, the head $u = b_1 \cdots b_s$ is communicated, where $b_1 \cdots b_s b_{s+1} \cdots b_{s+t}$ is the current code word. This information is used to synchronously update the dictionaries of the sender and receiver. The remaining t bits (the tail) of the current code word are lost in this case. Thus, the receiver knows the head to which the current data value belongs. In other words, the potential range of the correct code word has the size 2^t.

This potential range (i.e., the uncertainty of the correct data value) can be further reduced. In a miss case, the bit sequence $0^r u$ of length $r + s$ is transmitted to the receiver (see Algorithm 1, line 8). Recall that a bit string of length $r + t > r + s$ is transmitted if there is no miss, resulting in a worst-case compression ratio (WCCR) of $(r + t)/(s + t)$. The DAKODIS architecture is time-triggered. Since each task is executed according to a fixed schedule, there is no benefit from potentially shorter messages in the event of a miss. Instead, the uncertainty of the correct value in this case can be reduced by transmitting the bit string $0^r b_1 \cdots b_t$, consisting of the head $u = b_1 \cdots b_s$ and the $(t - s)$ MSBs $b_{s+1} \cdots b_t$ of the tail without increasing the WCCR. This narrows the potential range of the correct data value to 2^s. The compression ratio then becomes $(r + t)/(s + t)$ in every step.

5.1.2.2 Reducing the Miss Rate

The $t - s$ unused bits in a miss case (line 8 of Algorithm 1) can also be used to reduce the miss rate in the following way. Recall that in the compression scheme, the prefix 0^r tells the receiver that the next s bits form the head of the current code word in case of a miss. Thus, the prefix 0^r has a reserved meaning and is excluded from the set of compressed heads (i.e., the entries of the dictionary). However, the bit sequence 0^r can still fulfill its purpose even if it is included in the set of compressed heads (then the dictionary has 2^r elements). This requires to reserve the first 2^s code words belonging to the head that is assigned to 0^r for the transmission of heads in the event of a miss. This leads to the following extension of the algorithm:

The sender and receiver reserve one of the 2^r many dictionary entries (i.e., compressed heads) for the miss case; let this entry be p_0. Initially, $p_0 = 0^r$. A miss occurs if (i) the head u of the current code word uv does not belong to the dictionary or (ii) the head u is stored in the dictionary entry p_0 (i.e., $D[p_0] = u$) but the tail v belongs to the first 2^s many bit sequences from $\{0,1\}^t$, i.e., $v \in 0^{t-s}\{0,1\}^s$. In both cases, the algorithm transmits $p_0 0^{t-s} u$. From the fact that the transmitted bit sequence starts with $p_0 0^{t-s}$, the receiver concludes that a miss occurred. Moreover, the dictionary entry where the least-recently-used head is stored becomes p_0 on both the sender and receiver sides.

Depending on the application, the priority might be either a lower uncertainty of the correct data value in a miss case or a reduced miss rate. As a compromise between the two improvements of the basic cache-based algorithm, one can choose some $m \in [0, t-s]$ and use the m most significant bits of the tail to be transmitted to reduce the uncertainty, while using $2^t - 2^{s+m}$ additional code words (to transmit data values) with the head that is currently used to indicate a miss.

5.1.3 Example of the Cache-Based Compression Algorithm

For the following example of the cache-based compression algorithm, assume that the samples come from an analog-to-digital-conversion (ADC) with a resolution of 12 bits, which means that every data value is represented by a 12-bit code word. Splitting these 12 bits into $s = 4$ bits for the head and $t = 8$ bits for the tail, there are $2^4 = 16$ heads, each comprising $2^8 = 256$ code words. Setting $r = 2$ one obtains $2^r - 1 = 3$ compressed heads (01, 10, and 11) and saves $s - r = 2$ bits for transmission. It is assumed that 3 code words have already been transmitted, resulting in the dictionary mapping shown in Table 5.1. The third column denotes the time instant in terms of a counting variable; the lowest number indicates that the active head was least recently used. The example includes the improvement from Section 5.1.2.1 (reducing the uncertainty). The following itemization goes through the algorithm with some code words, beginning at time step 4:

- Time step 4: The first code word is 0101 00001100. Its head is 0101 and its tails is 00001100. The head is active: $D[10] = 0101$, so 10 00001100 is sent and the time index corresponding to the dictionary entry is updated to the current time instant 4.

Table 5.1: Example codebook of size 3 for the cache-based compression algorithm at time instant 4.

Compressed head	Active head	Least recently used at time
01	0000	1
10	0101	2
11	1001	3

- Time step 5: The code word is 0000 11011001. As $D[01] = 0000$, the algorithm sends 01 11011001 and the least-recently-used time becomes 5.
- Time step 6: The code word is 1010 00111110 with the head 1010. This head is currently not active, so there is a miss. According to the current state of the dictionary, the head 1001 stored at entry 11 was least recently used, and consequently gets replaced, i.e., $D[11] = 1010$ is set. Since $t - s = 4$, the four most significant bits 0011 of the tail 00111110 are sent to reduce the uncertainty of the correct data value. The transmitted bit sequence is 00 1010 0011: 00 to indicate the miss, 1010 for the new active head, and 0011 for the four MSBs of the tail. The time index is updated to 6.
- Time step 7: The code word is 1010 00111110, i.e., the same as the previous one. This time the head 1010 is active and stored at dictionary entry 11. Thus, 11 00111110 is sent.
- Time step 8: The final code word is 1001 10101011. Its head 1001 is no longer active (it was removed from the dictionary at time step 6), so there is another miss. This time, the least-recently-used head is $D[10] = 0101$ (at time instant 3). The algorithm sends 00 10011010 and sets $D[10] = 1001$.

5.1.4 Probability of a Miss

This section analyzes the probability of a miss of the cache-based compression algorithm. For this, a sequence of code words $x_1 x_2 x_3 \cdots$ is modeled as a stochastic process. Recall that there are $N = 2^{s+t}$ different code words. For comparison, it is first assumed that successive code words are identically and independently distributed, resulting in what is called an i.i.d. process. This is described by a single probability distribution $(P[x] \in [0,1])_{x \in \{0,1\}^{s+t}}$ on the set of code words, where $P[x]$ is the probability that the code word x occurs (the letter P

is used to represent probabilities throughout this section). Whether a particular code word leads to a miss depends only on the head of the code word. From the probabilities $P[x]$, the probability p_u that a particular head $u \in \{0,1\}^s$ appears can be computed as

$$p_u = \sum_{v \in \{0,1\}^t} P[uv]. \tag{5.1}$$

For a uniform distribution (i.e., $p_u = p_{u'}$ for all heads u, u'), the probability of a miss is calculated with the number of heads in the dictionary and the total number of heads to $1 - (2^r - 1)/2^s$. Based on the analyses of [FGT92] from the field of caching, for given head probabilities p_u the miss probability can be calculated in principle with the following formula:

$$1 - \sum_{u \in \{0,1\}^s} p_u^2 \cdot \sum_{q=0}^{k-1} (-1)^{k-1-q} \binom{2^s - q - 2}{2^s - k - 1} \sum_{|J|=q, u \notin J} \frac{1}{1 - P_J}, \tag{5.2}$$

where $k = 2^r - 1$ is the size of the dictionary and $P_J = \sum_{v \in J} p_v$. As also noted in [FGT92], this formula is not accurate for practical calculations of the probability of a miss, since the introduced independent reference model simplifies that in reality the references (i.e., the items to be accessed) exhibit some locality. Franaszek and Wagner [FW74] consider the expected ratio $F_{\text{lru}}/F_{\text{opt}}$, where F_{lru} is the miss probability under the LRU strategy and F_{opt} is the miss probability of an optimal replacement strategy. The latter stores the $k - 1$ heads with the highest probabilities in the dictionary. The remaining dictionary entry is used for the miss indication. Note that the optimal strategy assumes knowledge about the above probabilities p_u. Using the relevant parameters of this thesis, the result from [FW74] states that

$$\frac{F_{\text{lru}}}{F_{\text{opt}}} \leq 1 + \frac{(2^r - 1)(1 - \beta)}{1 + (2^r - 2)\beta}, \tag{5.3}$$

where β is the sum of the smallest $2^s - 2^r + 1$ many head probabilities p_u. Again, the result assumes that the sequence of code words is produced by an i.i.d. process.

To compare with practice, in the measurement of physical quantities, consecutive data values are highly interdependent. In particular, the locality typically observed in physical processes implies that a data value is highly probable to be in a small neighborhood of its preceding value. In such a setting, the cache-based compression algorithms presented in this chapter show a much lower probability of a miss than in the above i.i.d. setting. This is demonstrated by the experimental data in Chapter 6.

5.1.5 Dynamic Cache-Based Compression Algorithm for Individual Data Streams

It is a characteristic of the cache-based compression algorithm from Section 5.1.1 that the dictionary is updated only after a miss occurred. Analyses of several signals of interest for this thesis show that consecutive data values (e.g., samples from a measurement of a physical quantity such as a voltage or current measurement) often rise or fall in one direction in the short term, especially when the sampling rate is relatively high [SM06] (see also the test signals used in the evaluation chapter). Such signal behavior leads to frequent misses, i.e., as soon as a data value is no longer covered by an active head, meaning that the head u of a required code word $x = uv$ is not in the dictionary. Utilizing this knowledge, a *dynamic* version of the online compression algorithm overcomes this issue by adding some flexibility using offsets in the dictionary entries. This unfixes the static relation between a value i (a quantization level) and its corresponding code word x.

Consider a sequence of values $S = a_1, a_2, \dots, a_k$ on a single data stream such that for all i with $1 \leq i < k$, $|a_i - a_{i+1}| < 2^{t-1}$ but $\lfloor a_i/2^t \rfloor \neq \lfloor a_{i+1}/2^t \rfloor$ (again s and t are the size of the head and tail, respectively). This defines the maximum difference between two consecutive values and states that they are covered by different heads. If S is compressed using the cache-based compression algorithm from Section 5.1.1 with a dictionary of size 1 (i.e., $r = 1$), this will result in k misses since there are no consecutive values in S with the same head. To handle this case, every dictionary entry $D[p]$ becomes a pair (u, δ) of a head $u \in \{0,1\}^s$ and an *offset* $\delta \in [-2^{t-1}, 2^{t-1} - 1]$ (initially, $\delta = 0$). The offset affects the mapping between data values and code words. Here and in the following, heads (resp., tails) are identified with numbers from the interval $[0, 2^s - 1]$ (resp., $[0, 2^t - 1]$) using their binary representation. Moreover, the corresponding interval $I[p] = [u2^t + \delta, u2^t + \delta + 2^t - 1]$ is defined. In this manner, p *covers* the data values in the interval $I[p]$. The cache-based compression algorithm from Section 5.1.5 can be considered as the special case where δ is always 0. Let $(u[p], \delta[p])$ stand for $D[p]$.

Consider an example with $s = 4$, $t = 8$, and the head $u = 0000$. If the corresponding δ at $D[p]$ is zero, then the data values in $I[p] = [0 + \delta[p], 0 + \delta[p] + 2^8 - 1] = [0, 255]$ are encoded by the first $2^t = 256$ code words (with the head $u = 0000$). If the offset is increased to $\delta[p] = 1$, then the data values from $[1, 256]$ are encoded by the code words with the head $u[p]$. Now, when a sender transmits a code word $x = uv$, where u (resp. v) is the head (resp., tail) of x, the sender first checks whether there is a dictionary index p covering x. Since $\delta[p] \in [-2^{t-1}, 2^{t-1} - 1]$, one must have $u[p] \in \{u - 1, u, u + 1\}$. Suppose that the dictionary contains an index p that covers x (otherwise, the sender transmits $0^r u$ and adds the pair $(u, 0)$ to the dictionary). The sender takes the smallest such p and transmits pv to the receiver (see line 8 of Algorithm 2). It then updates its dictionary such that x becomes the center of an interval $I[q]$ for some dictionary index q. For this, it first ensures that $u[q] = u$ holds for a unique index q. Then it sets the offset $\delta[q]$ to $v - 2^{t-1}$ (line 20–21). In this way, x becomes the center of the interval $I[q]$. The receiver reconstructs x from p and v and updates its dictionary analogously. This means that the covering head in the dictionary is potentially replaced by its neighboring head (i.e., by the head that covers the value by default (with zero offset)) to ensure that the maximum offset remains within the defined interval. It can be easily observed that in the above example, only a_1 is lost with this dynamic cache-based compression algorithm, while maintaining a dictionary of size 1.

The algorithm enhancements from Section 5.1.2 can be straightforwardly applied to the dynamic version of the cache-based compression algorithm. In particular, the strategy for reducing the uncertainty in a miss case can be extended to allow an offset value to be computed even in a miss case, provided $s < t$. The potential range of the correct data value is then narrowed down from 2^t to 2^s and it is a good choice to reconstruct the center value of this range. Let the tail v' correspond to this center value. The offset for the newly inserted head then becomes $v' - 2^{t-1}$ on the sender and receiver sides.

Algorithm 2 Dynamic cache-based compression algorithm for 1 data stream

```
1:  input : data value $i \in \{0,1\}^{n_i}$
2:  output : bit string of length at most $r + t$
3:  miss : variable for indicating the event of a miss
4:  initialize dictionary $D$ as empty hash table of size $2^r - 1$
5:  let $x = uv = \mathrm{bin}_\ell(i)$ with $|u| = s$ and $|v| = t$
6:  if there is $p \in \{0,1\}^r \setminus \{0^r\}$ covering $x$ then
7:      let $p$ be the smallest index covering $x$
8:      send $pv$ to the receiver
9:      miss := 0
10: else
11:     send $0^r u$ to the receiver
12:     miss := 1
13: end if
14: if there is no $q$ with $u[q] = u$ then
15:     $q$ := fresh($D$)
16:     $u[q] := u$
17:     $\delta[q] := 0$
18: end if
19: if miss = 0 then
20:     let $q$ be the unique index with $u[q] = u$
21:     $\delta[q] := v - 2^{t-1}$
22: end if
```

5.1.6 Example of the Dynamic Cache-Based Compression Algorithm

Suppose 12-bit code words, each consisting of a 4-bit head and an 8-bit tail, are to be compressed to at most 10 bits by maintaining a dictionary D with $2^2 - 1 = 3$ entries ($D[01], D[10]$, and $D[11]$) for active heads. As before, the bit string 00 indicates a miss. Each head can have an offset from $[-2^{8-1}, 2^{8-1} - 1] = [-128, 127]$. Let the dictionary be initially empty. Using the dynamic cache-based compression algorithm from Section 5.1.5, a sender compresses and transmits the three data values 384, 434, 534 as follows:

1. Transmit 384: Since $\text{bin}_{12}(384) = 0001\,10000000$, the head of this string is $u = 0001$. There is no active head in the current dictionary, so the data value is lost. The sender sends 00 0001 to the receiver and inserts 0001 with offset 0 into the dictionary. Let $D[01] = 0001$. The receiver then reconstructs a default value from the received head u and the bit sequence $1\,0^{t-1}$, which forms the tail. Clearly, at this point it would be a good strategy to also insert some adjacent heads of 0001 into the dictionary to increase the value coverage, especially since free entries are available. However, this approach is not followed in this example.

2. Transmit 434: Now $\text{bin}_{12}(434) = 0001\,10110010$. The head of this bit sequence is 0001, which is stored at dictionary entry 01 and currently has an offset of 0. The data value is covered by an active head, so the sender transmits the compressed string 01 10110010 to the receiver and the new offset of 0001 is calculated from the current tail v (v corresponds to the remainder of $434/2^8$, i.e., 178) to $v - 2^{t-1} = 178 - 2^7 = 50$.

3. Transmit 534: The head of the string $\text{bin}_{12}(534) = 0010\,00010110$ is 0010. It is not active in the current dictionary. With the nondynamic cache-based algorithm (Section 5.1.1) the data value would be lost in this case, but with the dynamic cache-based algorithm a successful data transmission is possible: The offset of the currently active head 0001 (at $D[01]$) is 50, so it covers all data values in $[2^8 + 50, 2 \cdot 2^8 + 49] = [306, 561]$. Since 534 belongs to this interval, there is a success in lines 7–9 of Algorithm 2 with $u = 0001$. Considering the offset of the covering head, the current tail v' is the remainder of $(534 - 50)/2^8$, i.e., 228. The new accumulated offset becomes $50 + (228 - 2^7) = 150$. Since 150 is not in $[-128, 127]$, the algorithm replaces the covering head with its neighboring head (in the respective direction), which is obviously the default head of $\text{bin}_{12}(534)$ (i.e., the head covering the value with zero offset). So it sets $D[01] = 0010$ and the offset $\delta[01]$ then refers to this head and is $v - 2^{t-1} = 22 - 128 = -106$. In this way, the data values in $[2 \cdot 2^8 - 106, 3 \cdot 2^8 - 107] = [406, 661]$ are now covered.

5.1.7 Difference Coding for Individual Data Streams

The DAKODIS architecture requires online data compression algorithms that guarantee low message delay. In addition, the time-triggered architecture relies on a WCCR. A fixed compression ratio optimally combines a compression benefit with the architectural requirements. The cache-based algorithm, in its basic and extended versions, satisfies these two prerequisites at the cost of sporadic losses of samples, which then have a lower accuracy. To deal with these events, the algorithms possess a robust strategy that allows a signal to be reconstructed to the greatest extent possible even in extremely volatile situations, such as high signal fluctuations due to faults in the system.

In the discussion of the related work in Chapter 3, differential encoding was introduced as a well-known technique for efficient compression of sensor data, where locality of consecutive data values can be assumed. Recall that sensor data in this context refers to measurements of physical quantities such as voltages and currents. This section presents a modified difference encoding algorithm that adopts some of the features of the cache-based compression algorithms. It is partially able to fulfill the requirements from Section 3.1, and thus serves for comparison purposes. The general idea of differential encoding applied to sensor data is to relate consecutive data values to each other and encode their difference, which is assumed to require fewer bits (see Section 3.2.3). Of course, if the number of data values to be encoded is N, one needs $n = \lceil \log_2 N \rceil$ bits per corresponding code word. In order to cover the difference between any two consecutive data values (i.e., code words), one would need N values for this as well, and consequently, n bits to express such a difference value. A compression scheme that is not demanded to produce compressed code words of a fixed size (and WCCR below one) directly benefits from a potentially larger number of shorter compressed code words (e.g., entropy codes). However, with fixed-size compressed code words, such as those required for the DAKODIS architecture, the number of available bits to encode a difference value of two consecutive data values might be insufficient, thus making a correct value reconstruction impossible. Such a case might lead to a series of incorrect value reconstructions due to the dependence on a correct reference value (e.g., the previous value).

In the following, the same notation as for the cache-based compression algorithm is used, but now $s < r < s + t$ for some $s, t, r \geq 1$. For each data value $i \in [0, N-1]$, there is a code word $x = uv$ with $|u| = s$ and $|v| = t$. The parameters s and t (in bits) determine the number of heads and the number of tails that each head comprises. A data value can be expressed by the difference from its predecessor value. For example, a sender wants to communicate a value i_k ($k \geq 0$) to a receiver. The sender transmits $d_k = i_k - i_{k-1} \in [1-N, N-1]$, i.e., the difference between the current value i_k and the previous value i_{k-1}. The previous value serves as a reference for calculating the difference.

The receiver reconstructs the current value i_k from the received difference d_k and the stored previous value i_{k-1}.

The strategy of the algorithm is as follows: A static dictionary D is defined. It contains 2^r entries, of which 2^s entries are used to store all heads u. The remaining $2^r - 2^s$ entries are used to store all differences $d \in [-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1] \subseteq [1 - N, N - 1]$. Both the sender and the receiver generate the same dictionary. All entries of the dictionary are mapped to code words w of length r. In this strategy, not all possible differences of two consecutive values are mapped to code words. If the current difference d_k is stored in $D[p]$ ($p \in \{0,1\}^r$), then the code word $w = p$ (the dictionary indices are interpreted as code words) is transmitted and the receiver successfully reconstructs the value from the received difference and the previously reconstructed value. If d_k is not stored in the dictionary, there is a miss. In this case, the head u of $\text{bin}_\ell(i_k)$ is computed and the unique p with $D[p] = u$ (every head u is stored in the dictionary) is sent. Because of the identical dictionaries of the sender and receiver, the latter knows that this code word stands for a head and not for a difference value. With this information, a new reference value for the difference calculation is established at both sides, specifically the center value $u2^t + 2^{t-1}$ of the interval $[u \cdot 2^t, (u+1) \cdot 2^t - 1]$ covered by the head. The compression ratio of the difference coding algorithm is $r/(s+t)$ in every step. The first value of a data transmission is inevitably a miss, since there is no reference value at the receiver. So the initial transmission is always the code word w, corresponding to the head u of the first data value $\text{bin}_\ell(i)$. Large tail sizes (i.e., few heads) may persistently prevent the algorithm from reconstructing values correctly. After a miss occurred and the new reference value is established as the center value of the reconstructed head, the differences of subsequent values to this center value may permanently lie outside the interval $[-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$. For the case $s = 1$, only two heads are available to indicate a miss. They tell the receiver that the required difference does not belong to $[-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$. In this case, a better reference value can be calculated in terms of the last successfully reconstructed data value plus $-(2^r - 2^s)/2$ or $(2^r - 2^s)/2 - 1$, depending on the received head. This may reduce the distance to the actual data value and hence the probability of many consecutive misses.

Difference Coding with Value Predictions

In general, differential encoding and predictive encoding are closely related (see Section 3.2.3). Value prediction can help to further reduce the difference values to be encoded in order to save bits or have fewer misses. Consider linear prediction as an example. Let a current value be denoted as i_k and its two predecessor values i_{k-1} and i_{k-2}. The corresponding differences are then $d_k = i_k - i_{k-1}$ and $d_{k-1} = i_{k-1} - i_{k-2}$. In the above difference coding algorithm, if $d_k \in [-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$, the corresponding code word is transmitted, otherwise there is a miss. Using linear prediction, $d'_k = d_k - d_{k-1} = i_k - 2i_{k-1} + i_{k-2}$ is computed. Since in many cases $|d'_k| < |d_k|$, e.g., for (linearly) increasing value sequences, transmitting d'_k (if possible) will lead to fewer misses. The receiver reconstructs the value i_k (in the non-miss case) from the previously reconstructed values and d'_k. The miss case is handled analogously to the difference coding algorithm presented above.

The adopted miss handling strategy allows to apply value prediction with the difference coding algorithm. As a general point, it should be noted that there is an adverse possibility that an (adaptive) predictor might eventually fail to adapt properly to the signals, thus producing large errors in subsequent values.

5.2 Simultaneous Compression of Multiple Data Streams

5.2.1 Preliminaries

The cache-based compression algorithm introduced in Section 5.1 in its static and dynamic version handles each data stream individually. For the compression, it utilizes the fact that measurements of physical quantities can often be covered with only a part of the total code word space for certain time intervals. With a view to the overall system architecture, much data for monitoring and diagnostic purposes (e.g., voltages, currents or vibration measurements) of complex mechatronic systems is gathered and processed at different locations and needs to be exchanged over a network. Often, many of these data streams are highly correlated due to redundant measurements or physical relations of the measured signals. The extended compression scheme presented in the following takes advantage of both facts, the locality assumption and the signal correlations. In time-triggered systems, all data traffic is scheduled so the transmission paths are predetermined. In particular, when the source and destination nodes of two or more data streams are located close to each other or are the same, the overall message sizes for transmitting the data can be reduced, leading to advantageous scheduling results (e.g., a shorter makespan) [MO19]. For the following considerations, a strong temporal dependency of the data streams is assumed, as this is the prerequisite for combining the samples of multiple data streams into one combined code word at a certain point in time. In time-triggered systems, this is a valid assumption.

5.2.2 Cache-Based Compression Algorithm for Multiple Data Streams

To exploit correlations between data streams, the cache-based compression scheme is adapted to encode multiple data streams at once, i.e., *simultaneously.* Assume some d data streams and let x_i ($1 \leq i \leq d$, $d \in \mathbb{N}$) be the current data value of the i-th stream. Further, x_i is assumed to be an n_i-bit data value. Analogous to Section 5.1.1, for each $i \in [1, d]$ a partition $n_i = s_i + t_i$ is fixed and x_i is split into $x_i = u_i v_i$ with $|u_i| = s_i$ and $|v_i| = t_i$. The bit sequence u_i (resp., v_i) is the current head (resp., tail) of the i-th stream. It is not assumed $s_i = s_j$ or $t_i = t_j$ for $i \neq j$. In the following, let $s = \sum_{i=1}^{d} s_i$ and $t = \sum_{i=1}^{d} t_i$. One could apply the compression scheme from Section 5.1.1 to each of the d data streams separately by choosing numbers $r_i < s_i$ and maintaining a dictionary of size $2^{r_i} - 1$ for every i with $1 \leq i \leq d$. This leads to an overall compression ratio of $(\sum_{i=1}^{d} r_i + t)/(s + t)$. On the other hand, due to correlations between the data streams, the tuples of data values $(x_1, \ldots, x_d)$ will be scattered around a low-dimensional subspace of the d-dimensional product space. For example, if $d = 2$ and $x_2 = f(x_1)$ for a function f, then all tuples belong to a one-dimensional curve in the two-dimensional plane. In such a case, a better compression ratio of $(r+t)/(s+t)$ can be achieved by using a single dictionary of size $2^r - 1$ for some $r < \sum_{i=1}^{d} r_i$ that stores tuples of heads $u = (u_1, \ldots, u_d)$, which need s bits. A compressed data value then consists of the dictionary index $p \in \{0, 1\}^r$ where u is stored and the concatenation $v_1 v_2 \cdots v_d$ of the current tails. In the case of a miss, 0^r is transmitted followed by u. This leaves $t - s$ unused bits. Similar to the extensions of the original algorithm (Section 5.1.2), these bits can be used to transmit parts of the tails. In the multidimensional case, there are several possibilities:

- Transmit as many tails as possible completely and fill the remaining bits with the MSBs of the next tail,
- transmit the same number of MSBs for each tail,
- transmit MSBs for each tail where the number of MSBs is weighted by the complete tail length.

Note that with this approach all d data values from $(x_1, \ldots, x_d)$ are lost in case of a miss. Consequently, this method is a trade-off between a better (i.e., lower) compression ratio and a higher number of lost data values. Yet, it has a major advantage in that it allows for a better allocation of available bits among the different streams through the shared dictionary. In contrast to compressing the data streams individually, the number of active heads for one data stream can vary. It is automatically determined by the LRU replacement strategy for the dictionary entries.

For example, in the two-dimensional case with $r = 3$, the dictionary stores seven 2-tuples of active heads. It is possible that all active heads corresponding to one of the streams are equal. If the seven dictionary indices are identified with numbers from $[1, 2^r - 1]$ and $u[p, i] = u_i$ is defined, then possibly $D[p, 1] = u$ for all $p = 1, \ldots, 2^r - 1$. That is, the number of currently active code words per data stream is flexible. Recall that in the compression of individual data streams, each stream i $(1 \leq i \leq d)$ has a corresponding dictionary with a fixed size $2^{r_i} - 1$, which is also the number of active heads and hence the number of active code words per stream. In a scenario where there is not much fluctuation in one data stream but there is in the other, the new method is expected to perform better. The t_i are fixed for both methods, of course, but do not affect the above considerations.

As mentioned in Section 5.1.1, the dictionary can be implemented using a hash table, which is a structure that maps keys to values. For this, every tuple of heads is translated into a unique key, which can then be checked in constant time. A naive way is to use the numbers from the interval $\left[0, \prod_{i=1}^{d} 2^{s_i} - 1\right]$. For an arbitrary combination of d heads the key is then calculated according to $\sum_{i=1}^{d} (2^{s_i})^{i-1} u_i$. This corresponds to the value of the concatenation of the s_i-bit-long binary representations of the heads $u_1, \ldots, u_d$. More information and other implementation possibilities of hash functions can be found in the relevant literature, e.g., [Cor+09].

Algorithm 3 Cache-based compression algorithm for d data streams

```
1: input : data values $x_i \in \{0,1\}^{n_i}$ $(1 \leq i \leq d)$
2: output : bit string of length at most $r + t$
3: initialize dictionary $D$ as empty hash table of size $2^r - 1$
4: let $x_i = u_i v_i$ with $|u_i| = s_i$ and $|v_i| = t_i$ for $1 \leq i \leq d$
5: **if** there is $p$ with $D[p] = (u_1, \ldots, u_d)$ **then**
6:     send $p v_1 v_2 \cdots v_d$ to the receiver
7: **else**
8:     send $0^r u_1 u_2 \cdots u_d$ to the receiver
9:     $p :=$ fresh$(D)$
10:    $D[p] := (u_1, \ldots, u_d)$
11: **end if**
```

5.2.3 Dynamic Cache-Based Compression Algorithm for Multiple Data Streams

For an efficient compression of highly correlated multiple data streams, when each stream has consecutive data values with small gaps, Algorithm 3 is made dynamic, analogous to Algorithm 2, see also [Mec+19b]. Each dictionary entry $D[p]$ stores a d-tuple $(u_1, \ldots, u_d)$

of heads ($u_i \in \{0,1\}^{s_i}$) and an offset vector $(\delta_1, \ldots, \delta_d)$ with $\delta_i \in [-2^{t_i-1}, 2^{t_i-1}-1]$. It is defined that $u[p,i] = u_i$, $\delta[p,i] = \delta_i$ and the interval

$$I[p,i] = [u_i 2^{t_i} + \delta_i, u_i 2^{t_i} + \delta_i + 2^{t_i} - 1].$$

It is said that p covers the data values in the d-dimensional *hypercube*

$$H[p] := \prod_{i=1}^{d} I[p,i]. \tag{5.4}$$

This hypercube is called an *active hypercube* and the union of all active hypercubes is called the set of *active data tuples*. Now consider the case where a sender sends a tuple $x = (x_1, \ldots, x_d)$ of data values from d data streams to a receiver. Let $x_i = u_i v_i$ be as in Section 5.2.2. Then the sender first checks if there is a dictionary index p covering x (otherwise it transmits $0^r u_1 \cdots u_d$ and adds $(u_1, \ldots, u_d)$ with the offset vector $(0, \ldots, 0)$ to the dictionary). For every i with $1 \le i \le d$, one must have $u[p,i] \in \{u_i - 1, u_i, u_i + 1\}$. The sender again takes the first such p and sends $p v_1 v_2 \cdots v_d$ to the receiver. Then it makes the same updates in each dimension as described above for the case $d = 1$. Algorithm 4 summarizes the procedure, see also [Mec+19b].

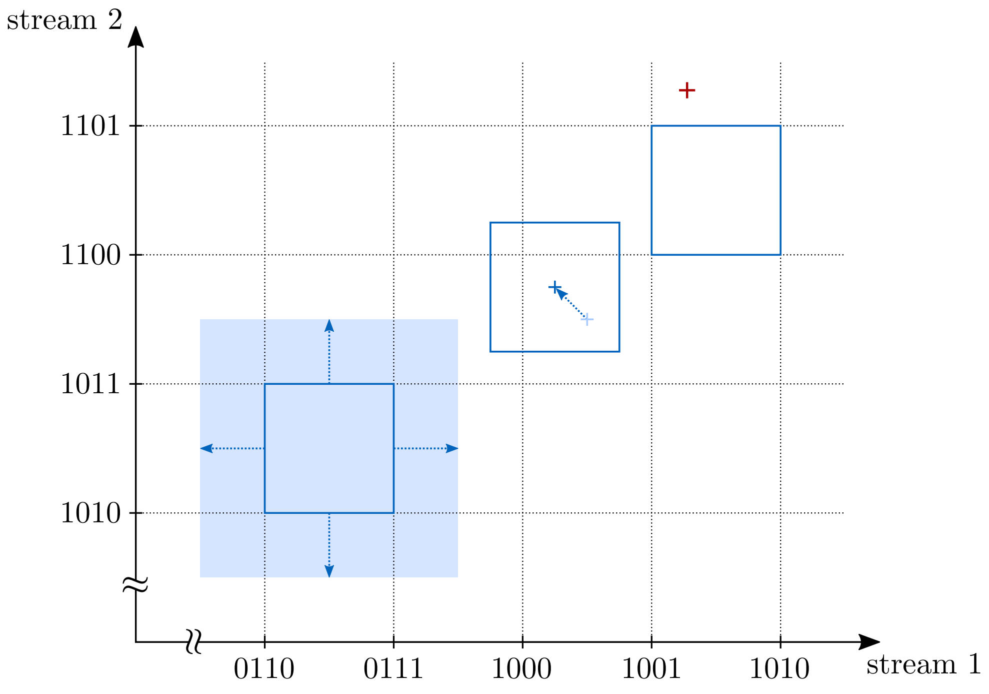


Figure 5.2: Simultaneous compression of two data streams.

Figure 5.2 illustrates a two-dimensional constellation of 3 active hypercubes (here squares) as blue squares and Table 5.2 represents the corresponding dictionary. For the hypercube at the bottom left (heads $u_1 = 0110$ and $u_2 = 1010$), the possible coverage range according to the maximum allowed offset values is exemplarily highlighted in light blue color. One of the hypercubes shown ($u_1 = 1000$, $u_2 = 1011$) is currently offset from its nominal position by some δ, indicated by the blue arrow pointing from the default center value towards the current center value. In the example, $-2^{t-1} < \delta_{2,1} < 0$ and $0 < \delta_{2,2} < 2^{t-1} - 1$, where the subscripts denote the dictionary index and the data stream index, respectively, see Table 5.2.

Table 5.2: Example codebook of size 3 for two-dimensional data stream compression.

Dictionary index p	$u[p,1]$	$u[p,2]$	$\delta[p,1]$	$\delta[p,2]$
01	0110	1010	0	0
10	1000	1011	$\delta_{2,1}$	$\delta_{2,2}$
11	1001	1100	0	0

Algorithm 4 Dynamic cache-based compression algorithm for d data streams

```
1: input : data values $x_i \in \{0,1\}^{n_i}$ $(1 \le i \le d)$
2: output : bit string of length at most $r + t$
3: miss : variable for indicating the event of a miss
4: initialize dictionary $D$ as empty hash table of size $2^r - 1$
5: let $x_i = u_i v_i$ with $|u_i| = s_i$ and $|v_i| = t_i$ for $1 \le i \le d$.
6: if there is $p \in \{0,1\}^r \setminus \{0^r\}$ covering $(x_1, \ldots, x_d)$ then
7:     let $p$ be the smallest index covering $(x_1, \ldots, x_d)$
8:     send $p v_1 v_2 \cdots v_d$ to receiver
9:     miss := 0
10: else
11:     send $0^r u_1 u_2 \cdots u_d$ to receiver
12:     miss := 1
13: end if
14: if there is no $q$ with $u[q] = (u_1, \ldots, u_d)$ then
15:     $q$ := fresh($D$)
16:     $u[q] := (u_1, \ldots, u_d)$
17:     $\delta[q] := (0, \ldots, 0)$
18: end if
19: if miss = 0 then
20:     let $q$ be the unique index with $u[q] = (u_1, \ldots, u_d)$
21:     $\delta[q] := (v_1 - 2^{t_1 - 1}, \ldots, v_d - 2^{t_d - 1})$
22: end if
```

5.2.4 Partial Misses

When compressing d data streams simultaneously, a miss occurs if there is no p covering x. Missing only one head u_i in the multidimensional product space leads to a miss in all data streams. Consider an example with $d = 2$ streams where $s_1 = s_2 = 4$, $t_1 = t_2 = 8$, and $r = 2$, i.e., the dictionary D is of size $2^r - 1 = 3$. Referring to Section 5.2.3, Table 5.2 is an example of such a dictionary. For now, all offsets are assumed to be zero. The dictionary index 00 is reserved for indicating the miss case and is not shown in the table. If the heads of the current data values x_1 and x_2 are assumed to be $u_1 = 1001$ and $u_2 = 1101$, respectively, there is a miss because the required concatenation of heads (u_1u_2) is not in the dictionary. Hence, the tail bits are used to send the new head combination u_1u_2 to the receiver. In Figure 5.2 the missing data value is marked as a red cross.

In the above situation, both data values x_1 and x_2 are lost. The strategy presented in the following overcomes this drawback of the algorithm in many cases and allows to successfully communicate the current data values corresponding to some of the streams. This is realized by reserving an additional dictionary index (i.e., the bit sequence $0^{r-1}1$) to indicate a so-called *partial miss* (in contrast to a *full miss* which is indicated by 0^r). This reduces the dictionary size to $2^r - 2$. The number of bits transmitted per time step is $r + t$ (i.e., the length of the dictionary index plus the length of all tails), as before. In the case of a partial miss, the bit sequence following $0^{r-1}1$ is interpreted differently than in the original algorithm. Table 5.3 shows the updated full dictionary for this example.

Table 5.3: Example codebook of size 4 for two-dimensional data stream compression.

Dictionary index p	$u[p, 1]$	$u[p, 2]$
00	reserved for *full miss*	
01	reserved for *partial miss*	
10	1000	1011
11	1001	1100

If there is no dictionary index p that covers the entire tuple of data values $(x_1, \ldots, x_d)$, the algorithm searches for a p that covers at least some of the data values. For this, the set $M[p] = \{i \in [1, d] \mid x_i \notin I[p, i]\}$ of those dimensions is defined where p does not cover the corresponding data value. In the example ($u_1 = 1001$ and $u_2 = 1101$), $M[10] = \{1, 2\}$ and $M[11] = \{2\}$. The algorithm tries to encode as many entries from the input tuple $(x_1, \ldots, x_d)$ as possible by looping over all dictionary indices $p \in \{0, 1\}^r \setminus \{0^r, 0^{r-1}1\}$. Consider a specific index p. Suppose the receiver is told p and the set $M[p]$. This leaves

$t-r-d$ bits from the initial $r+t$ bits (one needs r bits for the partial miss indicator $0^{r-1}1$, r bits for p, and d bits for $M[p]$). These $t-r-d$ bits can be used to send the following data to the receiver:

1. All tails v_i for $i \in [1,d] \setminus M[p]$. Note that for $i \in [1,d] \setminus M[p]$, the receiver can obtain the head u_i of x_i (and consequently the data value x_i) from the dictionary entry $D[p]$ as in Section 5.2.3. Note that $\sum_{i\in[1,d]\setminus M[p]} t_i$ bits are needed for these tails.
2. All differences $u_i - u[p,i]$ for every dimension $i \in M[p]$. Note that one bit is needed for each difference to encode the sign of the difference. Note also that the differences $u_i - u[p,i]$ ($i \in M[p]$) together with the index p allow to reconstruct the missed heads u_i ($i \in M[p]$) on the receiver side, which is necessary to update the dictionary.

For the differences in point 2 there are

$$h = \sum\nolimits_{i\in M[p]} t_i - r - d \tag{5.5}$$

bits available. If all differences from point 2 fit into these h bits, then p is a valid choice for the algorithm and all data values x_i with $i \in [1,d] \setminus M[p]$ are correctly transmitted. For more than 2 missed heads, the h many bits must be split among all missed heads according to a predefined scheme known to the sender and receiver.

Figure 5.3 illustrates the bit distribution of the available $r+t$ bits among the different segments and their order in the frame. After the partial miss indicator (p_{ind}), the index p_{hit} refers to the dictionary entry with at least one covering head. The set $M[p]$ takes d bits and is followed by all valid tails. The remaining bits are distributed among all head differences.

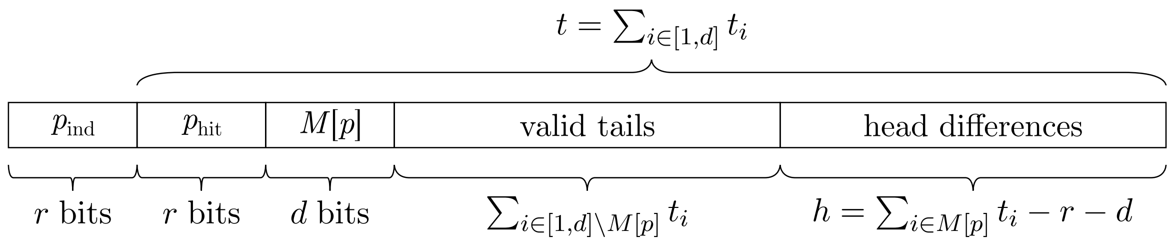

$t = \sum_{i\in[1,d]} t_i$
p_{ind}
p_{hit}
$M[p]$
valid tails
head differences
r bits
r bits
d bits
$\sum_{i\in[1,d]\setminus M[p]} t_i$
$h = \sum_{i\in M[p]} t_i - r - d$

Figure 5.3: Bit distribution in the partial misses strategy.

In the example, if $p = 11$, then $M[11] = \{2\}$ and the only difference needed is $u_2 - u[11,2] = 1$. Calculating $h = t_2 - r - d = 8 - 2 - 2 = 4$ and reserving one bit for encoding the sign of the difference value, one has a valid choice for p and can communicate the data value of stream 1 (via pv_1) without loss of accuracy and the head of the data value of stream 2 via its difference from the head currently stored at the transmitted dictionary index p. If it turns out that there is more than one valid choice for p, then one could choose that p that allows to transmit the maximum number of data values. There might also be scenarios where some of the data streams are more important; then one would give priority to those streams. If no valid index p is found, then a full miss is indicated via the bit sequence 0^r.

Especially for large dictionary sizes, where r is large, h becomes small. However, for a large dictionary (e.g., $r > 6$), one has the option of using more than one dictionary index to indicate a partial miss without significantly increasing the number of misses. If one decides to reserve 2^k dictionary indices (for some $k < r$) to indicate partial misses (resulting in a dictionary size $2^r - 1 - 2^k$), k additional bits can be assigned to h since each of the 2^k reserved indices now refers to a particular part of the dictionary which can be encoded with fewer bits.

The use of d bits to tell the receiver the set $M[p]$ allows all possible combinations of partial misses to be encoded. For example, if $d = 3$, there are 6 possible combinations, i.e., there could be a single head miss in any one of the data streams, or any combination of two heads could show misses. If the sender and receiver agree to allow only certain combinations of partial misses, e.g., only single head misses, the number of bits to communicate this information can be reduced from d to $\lceil \log_2 d \rceil$.

5.2.5 Grouping of Active Hypercubes

For the cache-based compression algorithm from Section 5.2.2 (where all offsets are zero), each active hypercube is maintained individually in terms of its corresponding head tuple in the dictionary, i.e., only in the case of a miss, the sender and receiver synchronously update their dictionaries by replacing the least-recently-used head tuple with the missed head tuple. The dynamic version of the algorithm from Section 5.2.3 uses an offset parameter to center an active hypercube around the transmitted sample to achieve a better coverage of the sample's neighborhood.

The evaluation results of the presented compression strategies show that for the same compression ratio, smaller dictionaries with larger hypercubes outperform larger dictionaries with smaller hypercubes, although the latter constellation manages more active values than the former, e.g., in Figure 6.5 (in the evaluation chapter) compare the constellation $d = 1$, $r = 3$, and $t = 8$ (i.e., $(2^3 - 1) \cdot 2^8 = 1792$ active values) against $r = 1$ and $t = 10$ (i.e., 1024 active values). In many cases, a newly inserted small active hypercube is not able to cover the next data value, especially when t is small compared to n (e.g., $t = n/2$). New strategies for combining multiple adjacent active hypercubes into larger active regions that cover more values around the last transmitted sample overcome this disadvantage. This can be applied both to individual data stream compression and to the simultaneous compression of multiple data streams. In the latter case, various possibilities exist for forming a so-called *region* of active hypercubes. Such a region is characterized by a subset of active hypercubes $H[p]$, one of which is defined as the *center hypercube*. An active region must be connected in the following sense: Take the graph whose vertices are the (d-dimensional) hypercubes from the region, and where two hypercubes are connected by an edge if and only if they intersect in a d'-dimensional sub-hypercube for some $0 \leq d' < d$. Then this graph must be connected. A region of active hypercubes is specified by a set $P \subseteq \{0,1\}^r \setminus \{0^r\}$ of (usually consecutive) dictionary indices, all having the same offset vector (which is called the offset of the region): $\delta[p] = \delta[p']$ for all $p, p \in P$ with $p \neq p'$. This means that if one hypercube is shifted (by changing the corresponding offset vector), then all other hypercubes from the region are shifted in the same way. Moreover, there is a distinguished $p_c \in P$, such that $H[p_c]$ is the *center hypercube* of the region and $u[p_c]$ is called the *center head* of the region. The active region corresponding to P is

$$R[P] = \bigcup_{p \in P} H[p]. \tag{5.6}$$

The same restriction as in Section 5.1.5 is imposed on the offset vectors, i.e., offset vectors must belong to $\prod_{i=1}^{d}[-2^{t_i-1}, 2^{t_i-1}-1]$. After every successful transmission (successful in the sense that no miss occurs), the heads and the common offset for the region $R[P]$ that covers the current tuple of data values $x = (x_1, \ldots, x_d)$ are updated in a way that x becomes the center of the center hypercube. Let $x_i = u_i v_i$, where u_i is the head of x_i and v_i is the tail of x_i. Assume that x belongs to the region $R[P]$, i.e., no miss occurs. Then every head $u[p]$ and offset $\delta[p]$ are updated for $p \in P$ by $u[p] := u[p] + (u_1, \ldots, u_d) - u[p_c]$ and $\delta[p] := (v_1 - 2^{t_1-1}, \ldots, v_d - 2^{t_d-1})$. Some care has to be taken in case a head $u[p]$ is out of the allowed range (e.g., contains a negative entry).

One may use only one active region (i.e., all dictionary indices contribute to the region) or multiple regions that can be moved independently. If the data values are clustered around several areas in the product space, then ideally one active region per cluster is used. In a miss case, there are several possibilities as to which region is moved to cover the current tuple of data values, such as the least-recently-used region or the nearest region.

In the one-dimensional case, a region is formed from a certain number (say k) of adjacent intervals, yielding a single interval $I = [u2^t + \delta, u'2^t + \delta - 1]$, where $u, u' \in \{0,1\}^s$, $k = u' - u$, and δ is the common offset for the region. The size of this interval is $k2^t \geq 2^t$. This large interval provides better coverage around the current data value and (under the locality assumption) leads to a higher probability that the next data value will also be covered by I.

The grouping technique shows its full potential in the compression of multiple correlated data streams. Multiple correlated signals often exhibit characteristic behavior in

Table 5.4: Head combinations for a two-dimensional transmission region of 7 squares.

Dictionary index p	$u[p,1]$	$u[p,2]$
001	$u_1 - 1$	$u_2 - 1$
010	$u_1 - 1$	u_2
011	u_1	$u_2 - 1$
100	u_1	u_2
101	u_1	$u_2 + 1$
110	$u_1 + 1$	u_2
111	$u_1 + 1$	$u_2 + 1$

the d-dimensional product space in the sense that they are concentrated around a low-dimensional subspace of the overall product space. This fact is exploited in multi-stream compression with grouping and allows to significantly reduce the number of misses. For example, the data values of two physically related signals might rise and fall in a similar manner. They are predominantly covered by regions that cover imaginary slopes in the two-dimensional product space. The algorithm supports predefined fixed d-dimensional regions with different number of hypercubes for different dictionary sizes. Symmetric regions are defined based on one center hypercube, e.g., a square or rectangle in the two-dimensional case, or a cuboid in the three-dimensional case, where the length of the hypercube in each dimension i is defined by the tail sizes t_i.

Table 5.4 shows an example for a dictionary defining a single active two-dimensional region consisting of 7 squares (the region corresponds to the 7 squares in Figure 6.6 with the black perimeter). Comparing the covered region of values with individual-stream compression, one would need to concatenate 3 intervals in each data stream to cover a square that encloses the defined region for the two-dimensional product space. For this, the individual-stream compression requires more bits (e.g., $r_1 = r_2 = 2$, $r = r_1 + r_2 = 4$ compared to $r = 3$ for the combined compression). This effect scales for larger regions, making multi-stream compression superior to single-stream compression due to its ability to save bits by limiting the value coverage to the specific region of interest. It should be noted that the grouping technique from this section can be combined with the partial miss strategy from Section 5.2.4.

5.2.6 Automatic Grouping of Active Hypercubes

The formation of a suitable region by grouping active hypercubes according to Section 5.2.5 requires knowledge about the expected behavior of some correlated signals in the product space. This can be obtained by analyzing the hit rate of hypercubes (e.g., see Figure 6.6 in the evaluation chapter). It is obvious that an unsuitable group constellation drastically decreases the performance of the algorithm. Furthermore, if the signal behavior changes over time, an initially optimal group constellation might not capture the signals well at a later time. The following sections present an extended strategy for the cache-based compression algorithm that automates the grouping of active hypercubes, thus eliminating the need for a manual signal analysis and design of group constellations.

5.2.6.1 Monitoring the Data Streams

By synchronously monitoring the hit rate of those hypercubes that cover the samples (hits and misses within a defined subspace of the overall product space) on the sender and receiver side, a region is automatically formed and maintained, and adapts to the signals as their characteristics change. To do this, the sender and receiver each maintain two dictionaries of different sizes, the smaller of which is referred to as the *transmission dictionary* and the larger as the *observation dictionary.* The former plays the role of the dictionary D used so far throughout Chapter 5, with the dictionary index p (an r-bit code) becoming part of the transmitted code word. In this extended strategy, however, the dictionary does not store the most recently seen heads at $D[p]$, but stores some indices that address entries of the observation dictionary. For clarity, from now on the transmission dictionary will be denoted by D_{trans} and the observation dictionary by D_{obs}.

The dictionary D_{obs} stores some heads that form a specially designed static region which is called *observation region.* The hypercubes of this region form a d-dimensional grid with an odd side length in each dimension. Neighboring hypercubes in the grid intersect in a $(d-1)$-dimensional hypercube. For simplicity, the same side length is chosen for the grid in each dimension (one could also choose different side lengths). Since this is an odd number, the center hypercube of the grid can be defined in the obvious way. The maximum side length of this grid, and hence the number of entries needed in the observation dictionary, depends on the size of the active region ($n_{\text{trans}} = 2^{r_{\text{trans}}} - 1$) maintained in the transmission dictionary. Let this region be called *transmission region.*

In an extreme case, all hypercubes of the transmission region are linked in a one-dimensional chain of length n_{trans}. This defines the maximum side length of the grid forming the observation region in each dimension. For example, if $d = 2$ and $r_{\text{trans}} = 2$, the number of hypercubes of the transmission region is $n_{\text{trans}} = 3$ and consequently the maximum number of hypercubes of the observation region is $n_{\text{trans}}^d = 9$. In practice, it turned out that a smaller observation region is sufficient. Experimental analyses showed that the number n_{obs} of hypercubes of the observation region should be chosen depending on d and satisfy $3^{d-1} \cdot n_{\text{trans}} \leq n_{\text{obs}} \leq n_{\text{trans}}^d$. Then the observation region is a d-dimensional grid of side length $\left\lceil n_{\text{obs}}^{1/d} \right\rceil$.

It should be pointed out that the size of the observation dictionary does not affect the transmitted code word lengths. Due to better adaptation capabilities of the active hypercubes to the signals, a larger n_{obs} might lead to fewer miss cases. Nevertheless, it is advantageous to limit the size for several reasons. The presented compression algorithms are specifically designed to compress correlated signals (see Section 3.1). Therefore, an alignment of many hypercubes in one dimension is unlikely. A limitation can help to prevent overfitting of the transmission region to the signals. This refers to the fact that short-term signal characteristics are well captured, but even the slightest signal changes lead to more misses. As a general rule, the performance of the algorithm decreases if overfitting occurs. Moreover, the time complexity of dictionary searches depends on the practical implementation and has to be considered with respect to the demanded real-time capability of the online data compression. Worst-case access times might drastically increase for larger dictionaries [Cor+09].

The strategy for an automatic composition of a transmission region aims to select $n_{\text{trans}} = 2^{r_{\text{trans}}} - 1$ indices from the observation dictionary. The hypercubes $H[p]$ (see Equation (5.4)) that are determined by the selected indices p define the transmission region; it is a subset of the observation region. The transmission dictionary stores the selected indices of the observation dictionary. The observation region behaves as described in Section 5.2.5, i.e., after every transmission, the tuple of input data values x lies in the center hypercube of the observation region. Moreover, the observation dictionary stores the offset vector. For the above selection procedure, a sliding window stores W (for a fixed constant W) observation dictionary indices. For each new tuple of input data values, the index p_{obs} of the observation dictionary that covers x is determined; p_{obs} is added to the sliding window and the oldest index is removed from the sliding window. If x does not belong to the observation region (and thus is not covered by an index of the observation dictionary), then the sliding window is not affected. These operations are performed synchronously on the sender and receiver sides. Note that it may happen that x does not belong to the transmission region (in which case a miss occurs), but does belong to the observation region. In this case, the sender and receiver modify the sliding window in the same way.

5.2.6.2 Static Transmission Dictionary Updates

After a predefined period of time, say τ_1 (e.g., some seconds), the sliding window is analyzed by counting for every observation dictionary index the number of its occurrences in the sliding window. The n_{trans} observation dictionary indices $p_1, \ldots, p_{n_{\text{trans}}}$ with the highest count are determined. The new transmission region is finally $\bigcup_{i=1}^{n_{\text{trans}}} H[p_i]$. Accordingly, the transmission dictionary is updated with the selected indices $p_1, \ldots, p_{n_{\text{trans}}}$. In this way, the transmission region reflects the temporal occurrence of data tuples with respect to their precursors. A predefined transmission region can be initially implemented on the sender and on the receiver side.

A successful data transmission is possible if there is a transmission dictionary index p_{trans} such that $D_{\text{trans}}[p_{\text{trans}}]$ (which is an index of the observation dictionary) covers the current tuple of data values x. If such an index p_{trans} does not exist, then there is a miss, which is indicated by the reserved entry of the transmission dictionary. Moreover, the observation region is centered around the approximation of x that is obtained from the transmitted tuple of heads (which is also known to the receiver).

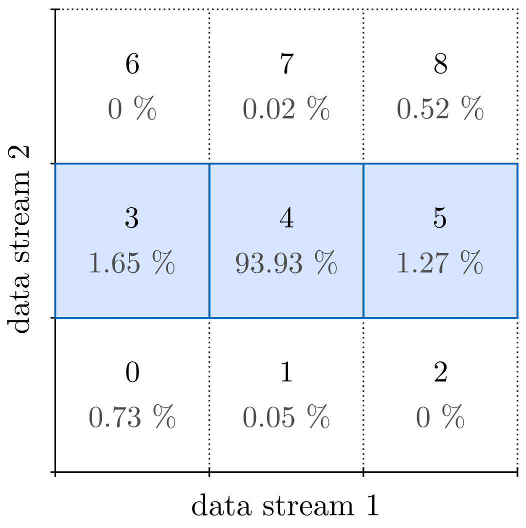


Figure 5.4: Two-dimensional observation region of 3×3 hypercubes for an active transmission region of 3 hypercubes.

Figure 5.4 illustrates an observation region for a two-dimensional case with $r_{\text{trans}} = 2$. Given the intended size of the transmission region (here $n_{\text{trans}} = 2^{r_{\text{trans}}} - 1 = 3$), the size of the observation region in each dimension is 3 and every index of the observation dictionary has $\lceil \log_2(3^2) \rceil = 4$ bits (i.e., $2^4 = 16$ dictionary entries, of which 9 are actually used). The squares in the figure are named with numbers from $0, \ldots, 8$ and represent the 9 entries (i.e., indices) of the observation dictionary. In the shown constellation, $H[4]$ forms the center hypercube. Moreover, the figure exemplarily shows hit counts for all observation dictionary indices as a result of a sliding window evaluation. It can be seen that in 93.93 % of the cases the data tuple to be transmitted was covered by the center hypercube. The active transmission region is consequently formed by the three hypercubes $H[3]$, $H[4]$ and $H[5]$ marked with the blue perimeter. The transmission dictionary entries are then set as follows: $D_{\text{trans}}[01] = 3$, $D_{\text{trans}}[10] = 4$, and $D_{\text{trans}}[11] = 5$. Adding up all the percentages shown reveals that a certain small portion of samples was not captured by the observation region.

The time interval τ_1 in which the transmission region is renewed influences the adaptability to the signals. If a change in the signal behavior occurs, a short time interval provides the possibility to adapt quickly. However, a typical problem then is a low count value for many of the observation dictionary indices in the sliding window. This might prevent a well-founded composition of the transmission region. For example, if the transmission region is supposed to consist of 31 hypercubes, but only 20 different indices occur in the sliding window, the strategy fails to compose the transmission region. Increasing the length of the sliding window might help, but a longer sliding window potentially contains more outdated information. The following paragraphs describe a strategy for filling the transmission region in the case there are not enough observation dictionary indices with a nonzero count in the sliding window.

The centers of the hypercubes $H[p]$, where p is an index of the observation dictionary (let them be called observation hypercubes in the following), form a set of points in a d-dimensional space. Each such point then corresponds to a unique observation dictionary index. The center of the center hypercube is declared to be the origin of the above point set. After an analysis of the sliding window, all such points are replicated with the number of occurrences of their corresponding observation dictionary index in the sliding window to weight their contribution. In order to determine the main direction of the signal behavior in the d-dimensional observation space that should be covered by the transmission region, a principal component analysis (PCA; see e.g., [Jol02, Chapter 8]) is performed. Roughly speaking, the principal components of such a set of points are a sequence of orthogonal basis vectors, which are determined as follows. First, a best fitting line (a vector) through the set of points is computed such that the average squared distance from the points to the line is minimized. A next best-fitting line must then have a direction perpendicular to the first, and in general this process is repeated so that the i-th vector that best fits the data is orthogonal to the first $i - 1$ vectors.

Figure 5.5 shows an example from two test signals from the evaluation chapter (see Section 6.2.1.1). The transmission region is supposed to be formed by 15 hypercubes ($r = 4$), but only 11 were selected from the sliding window analysis (marked in blue). In the two-dimensional observation space of size 9×9 hypercubes, the first principal component (*pc1*) highlights the main direction of the signal behavior (usually only one main signal direction is expected; especially when $d > 2$, the magnitudes of the principal components reveal the importance of adding hypercubes of the respective direction to the active region). It is a good choice to extend the transmission region in that direction.

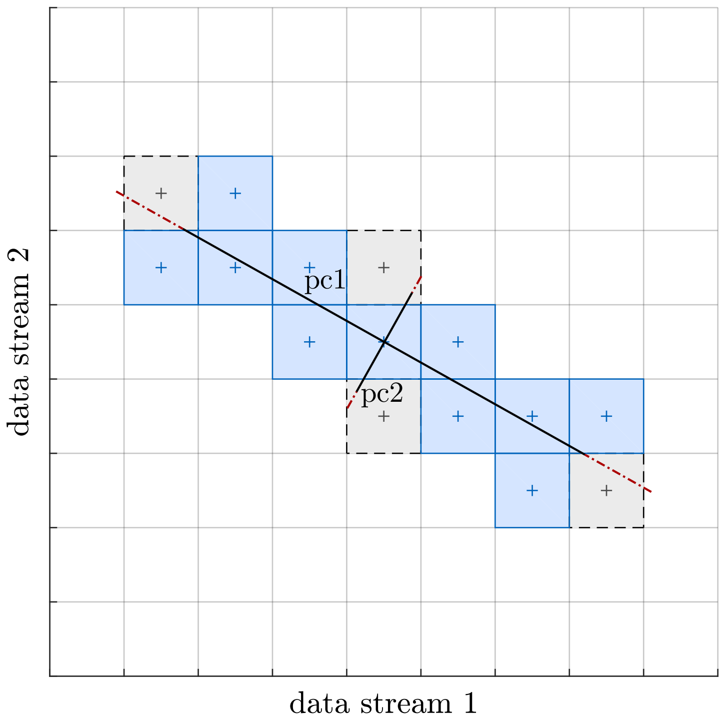


Figure 5.5: Forming a transmission region with the PCA strategy in case the number of different observation dictionary indices in the sliding window is insufficient.

To determine additional observation hypercubes (or equivalently, the corresponding indices of the observation dictionary) to be added to the transmission region, for each point the minimal distance to the origin of the principal components is calculated. In addition, a hypercube is formed whose edges are parallel to the principal components in the respective directions. The lengths of the edges are given by the maximum (resp., minimum) values of the points with respect to the new orthogonal basis. Let the center of this hypercube be the origin of the principal components, so that the hypercube is referred to as *enclosing hypercube*.

The distance values are sorted in ascending order, i.e., the point closest to the origin comes first. Recall that each point corresponds to a unique observation hypercube. Only if a point lies within the enclosing hypercube, its corresponding observation hypercube is a potential candidate to be added to the transmission region. The transmission region is then filled with observation hypercubes in the sorted order until the desired size is reached. Of course, only hypercubes that are not already part of the transmission region are selected. In the example in Figure 5.5, the transmission region is extended by the four hypercubes drawn in gray with a black dashed perimeter. To ensure that the transmission region always gets completely filled, the size of the enclosing hypercube is consistently and iteratively enlarged as needed. This strategy is indicated in the figure by the red dash-dotted lines that extend the principal components. When this strategy is applied, the shape of the transmission region also depends on the size of the observation region. Limiting the observation region to a reasonable size (recall that $3^{d-1} \cdot n_{\text{trans}} \leq n_{\text{obs}} \leq n_{\text{trans}}^{d}$) prevents the transmission region from becoming wide-stretched.

Furthermore, the following strategy is proposed to allow a fast adaptation in case the signal behavior changes drastically within a short time interval (consider also the case that a fault occurs in the system). A second time interval τ_2 is defined, which is shorter than the fixed renewal period τ_1 for the transmission region. Whenever possible (a certain number of different indices in the sliding window is required), the principal components are processed after τ_2 time steps. As long as the transmission region accurately reflects the signals, the direction of the first basis vector (pc1) of two consecutive τ_2-cycles does not vary much. However, an abrupt change in the direction indicates a changing signal behavior, so that the formation of a new region can be triggered in between the normal τ_1-cycles, which works well if the sender and receiver realize the strategy based on the same parameters.

5.2.6.3 Dynamic Transmission Dictionary Updates

The above strategy for updating the transmission region is based on fixed time intervals (τ_1 and τ_2). One can make this updating process more flexible once a transmission region has been formed with the above strategy. For this purpose, the misses that occur within the observation region are monitored. This implies cases where a data tuple is covered by an observation dictionary index which is, however, not stored in the transmission dictionary. These misses frequently occur when the shape of a transmission region does not cover the data tuples well and might indicate a change in the signal behavior. In each such miss case, the sliding window is evaluated (synchronously at the sender and receiver). The unique observation dictionary index with the lowest count that is part of the transmission dictionary is searched. Let this be p_1 stored at $D_{\text{trans}}[p']$. This count is then compared to the count of the observation dictionary index where the miss occurred, let this be p_2. If p_2 occurs more frequently in the sliding window than p_1, there is a high chance that upcoming data tuples will be covered by p_2. Therefore, p_1 is replaced by p_2 in the transmission dictionary by setting $D_{\text{trans}}[p'] = p_2$. Otherwise, the transmission dictionary remains unaffected.

This dynamic update process of the transmission region offers the potential to immediately and smoothly respond to changes in the signal behavior. However, since the updates are triggered by miss events and only refer to a single observation dictionary index at one time step, it does not provide the forward-looking character that the principal-component-based strategy offers. A combination of both strategies is possible. The transmission region is then completely renewed after the period τ_1. In between this time interval, the dynamic renewal process adjusts the transmission region to the signals as needed. In fact, the evaluation from Chapter 6 shows that without any prior analysis of the signal behavior in the d-dimensional product space over time, the strategy of automatic grouping of active hypercubes outperforms the other schemes due to its ability to automatically adapt to the signals and optimally form new regions.

5.3 Merging and Splitting Compressed Data Streams

5.3.1 Routing and Compression Scenarios

For a simultaneous compression of some values $x_1, \ldots, x_d$ of d different data streams into a combined code word, the presented cache-based compression algorithms require that all corresponding individual code words are present at *one* processing node, i.e., the source node. Analogously, the decompression of such a combined code word into all constituent data values is performed at one destination node.

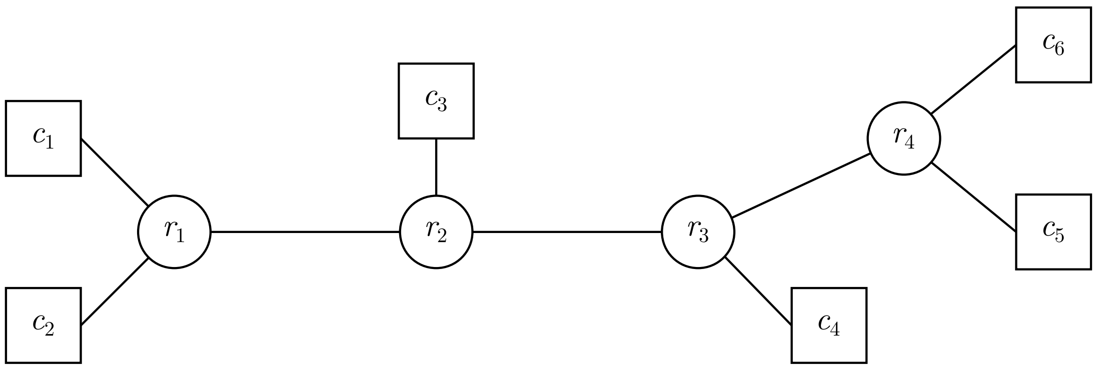


Figure 5.6: Distributed system with communication bottleneck.

Figure 5.6 shows a distributed system with six computation nodes ($c_1, \ldots, c_6$) and four routers ($r_1, \ldots, r_4$). Assume that two highly correlated data streams need to be communicated from c_1 to c_4. The only possible path for the messages is via the routers r_1, r_2, and r_3. Since the source and destination of the messages are identical, simultaneous compression of the two data streams can be accomplished using the cache-based algorithm. Note that the decision whether to actually combine some multiple data streams for compression is made by the DAKODIS scheduling algorithm as part of a systemwide organization of the data compression. The information about achievable (reduced) message sizes (with respect to the tolerable loss rate) for various scenarios (e.g., individual or simultaneous data stream compression) is stored in the compression model.

In Figure 5.6, assume a different scenario with two highly correlated data streams a_1 and a_2, both of which need to be communicated to computation node c_4, but this time one of them originates from c_1 and the other from c_2. According to the definition of the physical model from Section 4.2, only computation nodes can execute tasks. For the following considerations, routers are capable of performing compression operations on data streams. It is not assumed that routers can generally execute tasks similar to computation nodes. This reflects real-world scenarios where, for example, routers of a time-triggered system are implemented as field-programmable gate arrays (FPGAs), and it provides the ability to merge and split compressed data streams at routers. With the routers capable of executing the cache-based compression algorithms, an advantageous compression solution for the above example scenario could be an individual compression of the first data stream from c_1 to r_1 and an individual compression of the second data stream from c_2 to r_1, from where both data streams take the same path. The cache-based compression algorithm for multiple data streams then creates a combined code word from the two individual code words at r_1 and transmits it to c_4.

Since compression and decompression operations consume time and energy, the goal is to keep their number and computational cost as low as possible. The intended strategy combines (resp., splits) incoming data values at a router (r_1 in the above example) without first fully decompressing them. The process of creating combined code words from multiple input code words at a router is called merging data streams. Likewise, splitting a data stream at a router means decoding incoming combined code words into their constituent code words.

Referring to the above example, the code words of the two data streams a_1 and a_2 are supposed to arrive synchronously at router r_1 (at different ports) with the goal of forwarding them as a combined code word from r_1 to c_4. It is not straightforward to adopt the combined encoding strategy of the cache-based algorithm from Section 5.2.2 (with a joint dictionary), since a potential miss would occur only at the combining node, i.e., the router, without being known to the source nodes due to the absence of a side channel in the DAKODIS architecture. Dictionary synchronicity, and thus data consistency, can only be ensured if the router maintains multiple dictionaries and partially decodes incoming code words, as described in the following sections.

5.3.2 Merging and Splitting with the Static Cache-Based Algorithm

Let a router have k input ports and at each such port $j \in [1,k]$ arrive a compressed data stream. The incoming data is of the form $p_j V_j$, i.e., a dictionary index p_j of a dictionary D_j, where the respective heads $U_j = u_{j,1}, \ldots, u_{j,d_j}$ of the data tuple $X_j = x_{j,1}, \ldots, x_{j,d_j}$ are stored, and the tails $V_j = v_{j,1} \cdots v_{j,d_j}$ of some d_j compressed data streams, see Figure 5.7. In the following, let $d_j = d$, i.e., all incoming data streams are combinations of d data streams. The router maintains separate dictionaries for each port. To merge these k input data streams, the dictionary D_m (the subscript m denotes that it is the merging dictionary) of size $2^{r_\mathrm{m}} - 1$ with $r_\mathrm{m} < \sum_{j=1}^{k} r_j$ (to benefit from simultaneous compression) is used at the router (r refers to the number of bits per dictionary entry, recall Section 5.1.1).

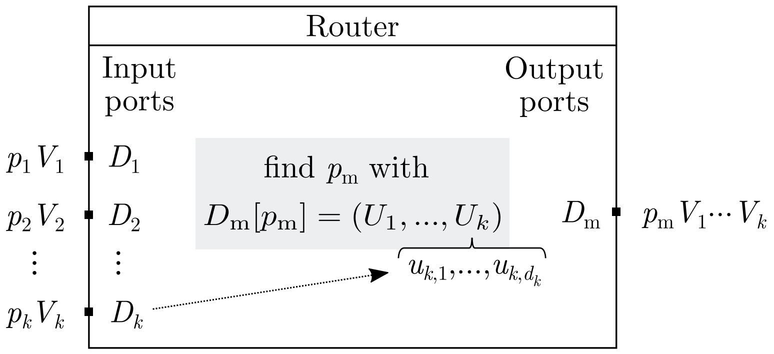


Figure 5.7: Merging of multiple data streams at a router.

Without fully reconstructing all data values arriving at the multiple ports, the merging procedure directly takes the tuple of the heads $U_1, \ldots, U_k$ of those data streams to be merged. In total, these are $\sum_{j=1}^{k} d_j$ heads. It is then checked whether there is a dictionary index $p_\mathrm{m} \in \{0,1\}^{r_\mathrm{m}} \setminus \{0^{r_\mathrm{m}}\}$ of D_m where such a combination of heads is stored, see Figure 5.7. If successful, the bit sequence $p_\mathrm{m} V_1 V_2 \cdots V_k$ is transmitted to the receiver, which synchronously maintains a dictionary of the same size, according to the original cache-based algorithm. On a miss, the strategy behaves as in Section 5.2.2 and sends $0^{r_\mathrm{m}} U_1 \cdots U_k$ to the receiver and updates its dictionary according to the LRU replacement strategy.

Splitting a combined code word into constituent code words (possibly formed from a subset of heads of the incoming combined code word) at a router works in an analogous manner. The router then serves as the sender and maintains dictionaries for each output port where the required new code words are composed. Misses are handled in the obvious way, i.e., by forming the code word from the reserved dictionary index and the missed heads.

5.3.3 Merging and Splitting with the Dynamic Cache-Based Algorithm

In the dynamic version, where the heads have offsets, merging and splitting data streams is more complex. Now all dictionaries D_j as well as D_m additionally maintain offset vectors. For the following considerations, the grouping strategy from Section 5.2.5 (with one active region) will be applied to all involved compressions. This means that the involved dictionaries each maintain only one offset vector.

The above strategy for checking D_m for the heads $U_1, \ldots, U_k$ of those data streams to be merged can be straightforwardly applied. It is extended such that D_m takes over the offsets belonging to the heads $U_1, \ldots, U_k$. This eliminates the need to fully reconstruct the input data values and check their coverage in D_m considering the offsets. However, consider the case where a miss occurs at a router during the merging process, i.e., the required combination of heads $U_1, \ldots, U_k$ is not in D_m. Due to the freshly inserted heads in D_m (with zero offsets), a mismatch occurs with the offsets stored in D_j, which invalidates further merging. In general, as long as there is no consistency between these offsets, one must fully decode all $X_j = x_{j,1} \cdots x_{j,d}$ and check the coverage of the combined data tuple in D_m including the offsets. As the evaluations from Chapter 6 show, it is likely that a value following a miss case is covered again. Then the offset vector in D_m gets resynchronized so that further checking can be avoided. In order to increase the chance of fast offset restoration, the compressors involved can agree to omit small offsets and set them to zero, or to periodically reset all offsets. Splitting a combined code word into constituent code words works in an analogous way.

Chapter 6

Evaluation and Results

The goal of this chapter is to evaluate the presented online data compression algorithms by means of test scenarios. The chapter is divided into several parts and demonstrates that the compression algorithms meet the requirements from Section 3.1. First, an automotive use case based on a Simulink model is introduced. The automotive domain is a typical example for the application of fault diagnosis. Especially the trend towards assisted and autonomous driving leads to more and more safety-critical systems where online fault diagnosis contributes to achieve fault tolerance, see e.g., [Gor19]. In the second part, the focus is on the performance of the data compression algorithms in terms of achievable compression ratios with respect to data value omissions (i.e., loss rates). In this context, both the compression of individual data streams and the simultaneous compression of multiple data streams are discussed. Next, the impact of compressed messages on scheduled communication is examined from a system-level perspective. In particular, the possibilities for reducing schedule lengths are investigated. Finally, the effect of compressed communication is evaluated in terms of applications, for which also a diagnostic use case based on real data is used.

6.1 Use Case – Hybrid Electric Vehicle

6.1.1 Hybrid Electric Vehicle Model in Simulink

This section introduces a Simulink model of a hybrid electric vehicle (HEV), which serves as a use case for the demonstration of fault diagnosis and the evaluation of the presented data compression algorithms. The base model was published by Miller [Mil20]. It offers a simulation of the vehicle's speed behavior according to a driving cycle input (i.e., a speed profile).

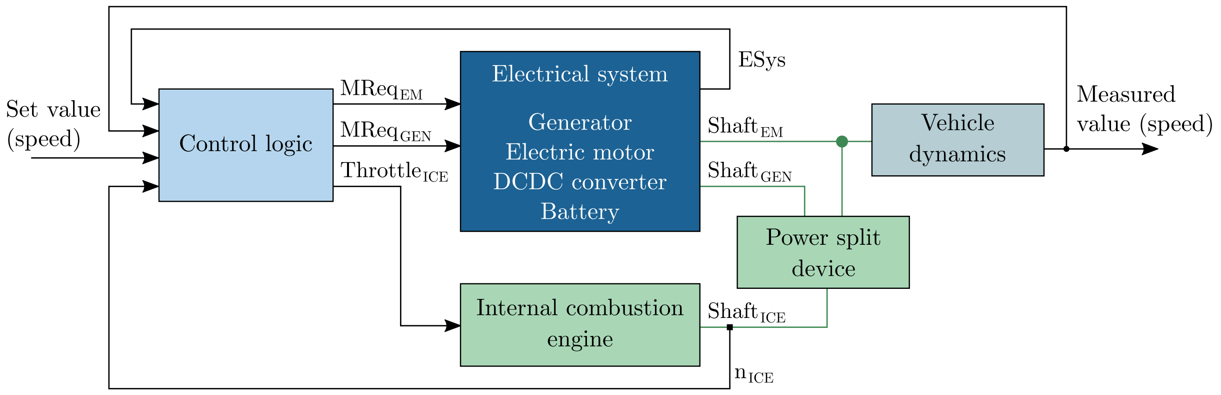

Set value (speed)
Control logic
MReq_{EM}
MReq_{GEN}
$\text{Throttle}_{\text{ICE}}$
Electrical system
Generator
Electric motor
DCDC converter
Battery
ESys
Shaft_{EM}
$\text{Shaft}_{\text{GEN}}$
Vehicle dynamics
Measured value (speed)
Power split device
Internal combustion engine
$\text{Shaft}_{\text{ICE}}$
n_{ICE}

Figure 6.1: Overview of the HEV model [MOY18].

As shown in Figure 6.1, the HEV model integrates electrical and mechanical parts (printed in blue and green, respectively) with various components in terms of a control application. The main components of the high-voltage electrical system include an electric motor and a generator in a parallel arrangement, which are connected to a battery via a voltage (i.e., DCDC) converter. The mechanical part comprises a power split device (i.e., a planetary gear) that combines an internal combustion engine (ICE) with the generator and the driveshaft, while the electric motor is directly connected to the driveshaft. Via the power split device, the combustion engine performs two tasks: supporting the electric motor to drive the vehicle and extending the vehicle's operating range by charging the battery via the generator. An operating control logic (printed in light blue) manages the interaction between these components according to the system input (i.e., set value) and the measurements that describe the current driving situation. The mode logic distinguishes two main modes, namely motion mode and brake mode. The former is divided into start mode and normal mode, where in start mode the combustion engine is switched off and the vehicle drives purely electrically. In normal mode, both the combustion engine and the electric motor operate, so the vehicle is either in the so-called accelerate mode or in cruise mode. In cruise mode, the generator is activated if the battery needs to be charged. This is managed by a battery controller. In brake mode, regenerative braking is performed to charge the battery. Many built-in sensors, such as voltage sensors, current sensors, torque sensors, or tachometers, allow the components to be monitored in various driving situations and conditions. The temperature of different components as well as the mode logic signals are also tracked. The HEV model is built from a broad range of predefined building blocks offered by the Simulink library. Thereby, certain components (e.g., the high-voltage battery) are compositions of multiple smaller units. Connections between the components can be mechanical (green) or electrical (blue, within the subsystem). A black line connection indicates the exchange of control signals and sensor measurements. Vehicle dynamics such as wind speed, road inclination and tire slip are also taken into account in the control loop.

In order to carry out fault diagnosis analyses, the Simulink base model was extended in the context of the DAKODIS research project. For this, fault injection capabilities were incorporated so that component failures can be simulated, as addressed in [MOY18; Mec+20b; ES21]. For a general introduction to HEVs, see e.g., [Hof14].

6.1.2 Fault Model and Fault Injection

Faults in a system originate from various causes, such as a faulty design of a component, wearout, overload, software faults, and erroneous user operations. As described in Section 2.5.2, faults are classified according to their origin, nature, dimension, system boundaries, and persistence. In the context of the simplified use case of this thesis, only permanent component failures will be considered. From the system-level perspective, a component failure is a fault. These failures can occur abruptly or incipiently at any point in time during run-time. The affected component then either fails suddenly and without prior indication within measurements, or its failure unfolds over a certain period of time (degrading performance).

The mechanism for simulating the above component failures is called *fault-injection*. Simulink modeling blocks that abstract real-world components (e.g., an electric drive) are typically not designed for failures. Rather, they simulate the ideal component behavior. To be more realistic, influences such as heat loss or noise must be additionally implemented. Consequently, the realization of fault injection requires component interventions in the sense of a targeted behavior manipulation, e.g., a deliberate power loss of an electric drive.

6.2 Evaluation of the Online Data Compression Algorithms

This section evaluates the performance of the presented compression algorithms in terms of the achievable loss rate with respect to the compression ratio.

6.2.1 Test Signals

6.2.1.1 Sensor Measurements from the HEV Model

The Simulink HEV model allows sensor measurements and state variables on the various electrical and mechanical components included to be realistically simulated as the vehicle travels the speed profile.

Figure 6.2 illustrates the voltage and current measurements at the DCDC converter from a simulation based on the World harmonized Light-duty vehicles Test Cycle (WLTC) class 3 driving cycle [Uni13]. The measurements are mapped to corresponding quantization levels

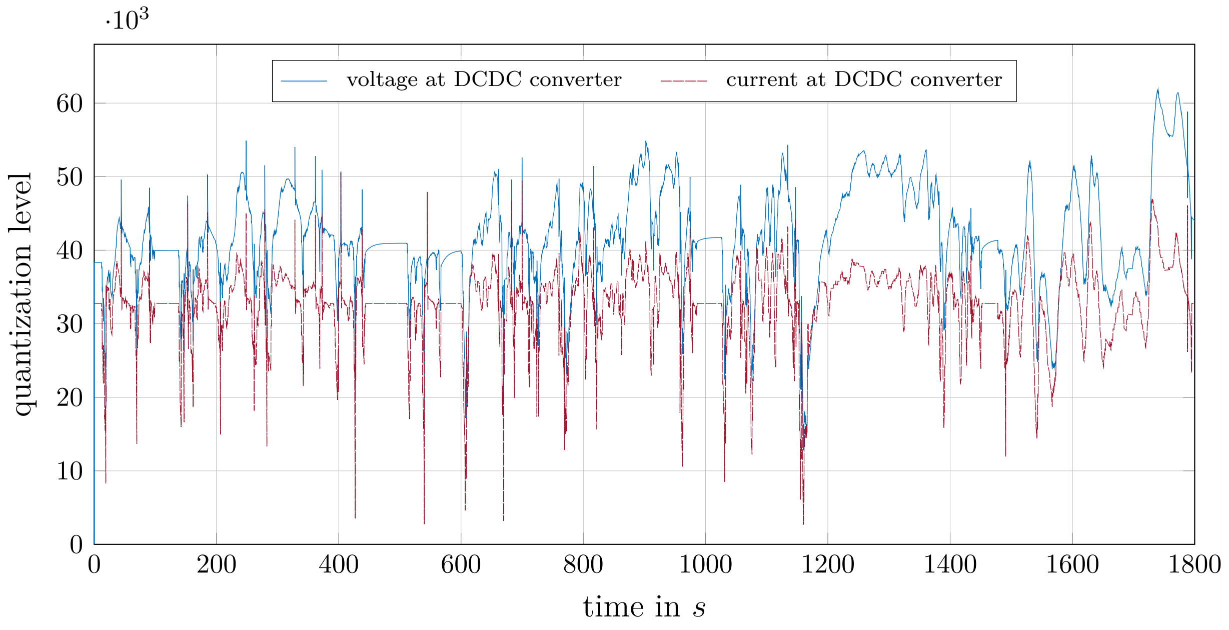


Figure 6.2: Quantized voltage and current signals at the DCDC converter of the HEV model.

(analog-to-digital conversion). The voltage signal is shown as a blue line and the current signal as a densely dashed red line. Each dataset contains $N_\mathrm{s} = 180100$ samples, quantized with $n = 16$ bits at a sampling frequency of 100 Hz. With regard to the data compression algorithms, the signals offer challenging signal behavior and can be seen as representatives of signals that are often to be analyzed in fault diagnosis. They include a broad coverage of quantization levels, both slowly varying and rapidly varying signal sequences. Moreover, the two signals exhibit a correlation based on their physical relationship, i.e., they rise and fall in a similar manner, often with different offsets.

This correlation can be quantified with the Pearson correlation coefficient $\rho \in [-1, 1]$, which is a suitable measure of the similarity of two signals, since it indicates a linear correlation. For example, a value of $\rho = 1$ expresses a completely positive linear correlation, 0 means no linear correlation, and -1 indicates a completely negative linear correlation. Let the two signals from Figure 6.2 be denoted by S_c (current) and S_v (voltage). The Pearson correlation coefficient for the two signals over the entire length is

$$\rho(S_\mathrm{c}, S_\mathrm{v}) = \frac{\mathrm{cov}(S_\mathrm{c}, S_\mathrm{v})}{\sigma(S_\mathrm{c}) \cdot \sigma(S_\mathrm{v})} \approx 0.813, \tag{6.1}$$

where cov is the covariance and σ is the standard deviation. The standard deviation of S_c is given by

$$\sigma(S_\mathrm{c}) = \sqrt{\frac{1}{N_\mathrm{s}} \sum_{i=1}^{N_\mathrm{s}} |S_\mathrm{c}[i] - \mu(S_\mathrm{c})|^2} \tag{6.2}$$

($\sigma(S_\mathrm{v})$ is calculated analogously) and the covariance of two signals is defined as

$$\mathrm{cov}(S_\mathrm{c}, S_\mathrm{v}) = \frac{1}{N_\mathrm{s}} \sum_{i=1}^{N_\mathrm{s}} (S_\mathrm{c}[i] - \mu(S_\mathrm{c})) \cdot (S_\mathrm{v}[i] - \mu(S_\mathrm{v})) \tag{6.3}$$

(in both equations Bessel's correction is not considered), where μ refers to the means of the signals:

$$\mu(S_\mathrm{c}) = \frac{1}{N_\mathrm{s}} \sum_{i=1}^{N_\mathrm{s}} S_\mathrm{c}[i] \quad \text{and} \quad \mu(S_\mathrm{v}) = \frac{1}{N_\mathrm{s}} \sum_{i=1}^{N_\mathrm{s}} S_\mathrm{v}[i]. \tag{6.4}$$

For further analyses, a dataset containing several measurements on different components of the vehicle was created based on a WLTC class 3 driving cycle simulation. The following list summarizes the most important components and measurements, some of which are used for the evaluations in Section 6.2.2 and Section 6.2.3.

- DC bus (voltage)
- Mode logic states
- Combustion engine (engine speed, torque)
- Generator (voltage, current, torque, rotational speed)
- Electric motor (voltage, current, torque, rotational speed)
- Car scopes (vehicle speed, accelerator, drive torque, power)
- Battery (voltages, currents, state of charge, cell temperatures)

6.2.1.2 Generation of Synthetic Test Signals

In addition to the simulated sensor measurements from the Simulink HEV model, this section presents a signal generator capable of synthesizing generic test signals from statistical parameters. In particular, the ability to produce a plethora of signals with specific correlation characteristics allows to evaluate the scalability of the developed compression algorithms. Using test datasets of real measurements published by companies would also be a good choice, but due to confidentiality and intellectual property protection, the availability of such test data is limited.

From the working principles of the presented data compression algorithms it is evident that losses tend to occur when the difference between consecutive data values is large.

Histograms are suitable for visualizing distributions of numerical data. Here, the distribution of differences between consecutive data values is of interest. To create a histogram from a signal, the entire range of input data values is divided into a series of consecutive, non-overlapping intervals. These so-called *bins* are adjacent to each other and of equal width, as they represent the steps between quantization levels. After counting for each bin how many of the input values belong to the corresponding interval, rectangles with a height proportional to the bin count (i.e., the frequency) can be erected over each bin for a graphical representation (assuming the bins are of equal width). In a normalized histogram, relative frequencies are plotted, i.e., the proportion of cases that fall into the respective bin. Then the sum of the heights of all rectangles is one.

In order to generate test signals for the evaluation of the data compression algorithms, a test signal generator was developed that applies the idea of a histogram in reverse. The goal is to synthesize desired signals based on difference values (i.e., differences between quantization levels), obtained using one or more Gaussian (i.e., normal) distributions defined by the parameters mean μ and standard deviation σ according to the probability density function

$$\frac{1}{\sigma\sqrt{2\pi}}\,\mathrm{e}^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}. \tag{6.5}$$

In case of using multiple (normalized) Gaussian distributions to specify difference values, one must have $\sum_{i=1}^{n}\mu_i = 0$, i.e., the sum of the respective mean values of n distributions must be zero so that the occurrence of positive and negative values is equal. This prevents the signal from drifting away from its initial value. Specifying a signal in this way enables both slowly as well as rapidly varying signal parts to be accounted on the basis of probabilities. The signal generation procedure then prepares an array of the desired signal length so that the value counts of the difference values reflect the above signal specifications. The permutation of the values in the array according to a discrete uniform distribution yields the final (difference) signal.

The signal generator also provides the possibility to add peaks to the final signal, which are modeled in the form of several successive rapidly increasing and decreasing (resp., decreasing and increasing) data values. These peaks reflect common signal behavior observed in many voltage and current signals. Other predefined signal patterns can also be incorporated in this way. Resampling can be performed as another postprocessing step. The generation of multiple correlated signals is supported by iteratively synthesizing signal parts that satisfy previously defined Pearson correlation coefficients. Overall, the signal generator also allows system designers to generate signals for modeling specific system behaviors for which real systems cannot be used, e.g., simulating signal behaviors in fault situations.

6.2.2 Loss Rates of Individually Compressed Data Streams

The data compression algorithms from Chapter 5 were developed to meet the requirements stated in Section 3.1. One of the main challenges was to reconcile the requirements of realizing a worst-case compression ratio (WCCR) below one on short input sequences (requirement R2) while ensuring data quality with respect to time-triggered messages (requirement R4). This led to a design of online data compression algorithms capable of producing fixed-size outputs from fixed-size inputs when a low probability of data values being reconstructed with slightly lower but bounded accuracy is accepted. The following analyses address these so-called loss cases and establish a relation between the statistics of input signals and expectable value losses. For preparing the figures in this section, the current measurement at the DCDC converter obtained from a WLTC simulation with the Simulink HEV model, as described in Section 6.2.1.1, is used as a test signal. Thus, although the specific values in the following figures are derived from the test signal, general conclusions about the behavior and performance of the compression algorithms can still be drawn. In all cases, the dictionary is initially empty.

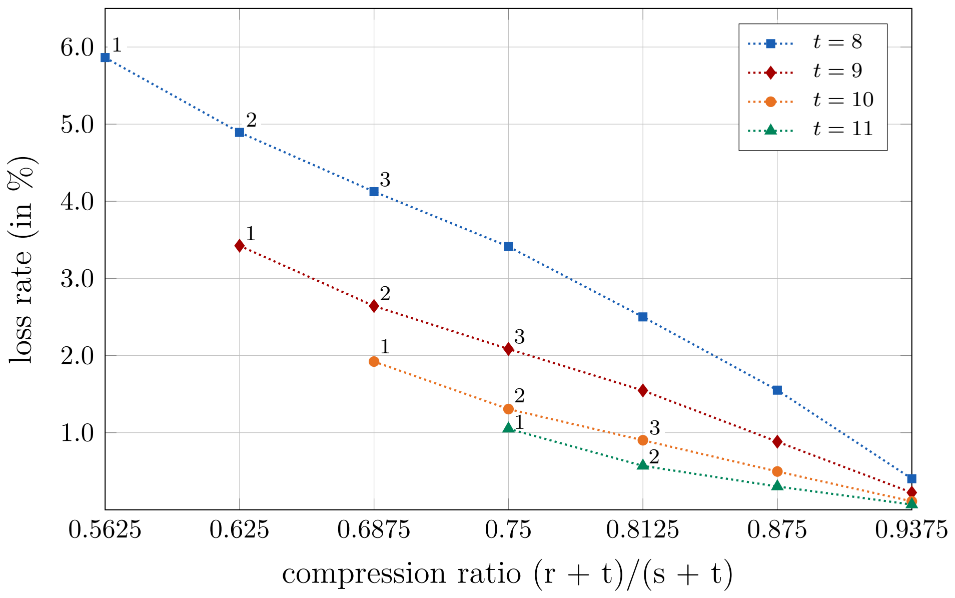


Figure 6.3: Individual compression of a data stream with the static cache-based algorithm. Comparison of different dictionary and tail sizes.

Figure 6.3 exemplarily examines the performance of individually compressed signals with the static cache-based algorithm from Section 5.1.1. It shows the relationship between the loss rate and the compression ratio $(r+t)/(s+t)$ for different settings of the parameter t. Recall that $n = s + t$; an n-bit input data value is split into a head (s bits) and a tail

(t bits). Since each line in the figure is computed for a constant tail size t, the corresponding head size s is also constant. Here and in Figure 6.4, the parameter r that determines the dictionary size (and consequently the compression ratio) is written to the first three data points of each line; r increases by 1 for every further data point. Since $t \geq s$, the lowest possible compression ratio (with $r = 1$) is $9/16 = 0.5625$ and the highest reasonable compression ratio is $15/16 = 0.9375$ in the example with $t = 8$. The dotted lines between the data points are shown for illustration only. For all specific compression ratios, the figure shows that combinations of smaller dictionary sizes and larger tail sizes result in lower loss rates, e.g., at a compression ratio of 0.75, the square (where $r = 4$, $t = 8$) marks the highest loss rate and the triangle (where $r = 1$, $t = 11$) indicates the lowest loss rate. This is due to the larger coverage of values in the neighborhood of the current sample (determined by t) while the signal traverses the entire range of values. Small tail sizes lead to an increased number of loss cases, especially if there are large gaps between successive data values of the input signal.

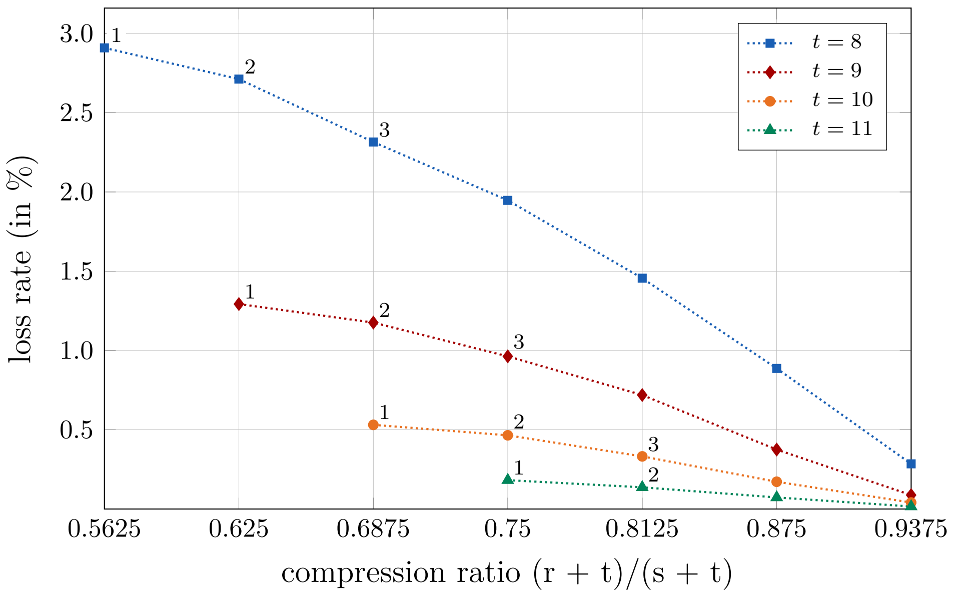


Figure 6.4: Individual compression of a data stream with the dynamic cache-based algorithm. Comparison of different dictionary and tail sizes.

The results in Figure 6.4 for the dynamic cache-based algorithm are based on the same test signal as used for Figure 6.3. Again, as indicated in the legend, t and consequently s are constant for each line, so the compression ratio is determined by the varying parameter r (see the annotations at the markers). In comparison, here the loss rate is lower for all compression ratios. This is firstly due to the dynamic adaptation of the value coverage of the covering head to the current sample, and secondly due to the mechanism that automatically activates a neighboring head when the covering head's offset exceeds a certain threshold. This enables dictionary updates without a prior value loss.

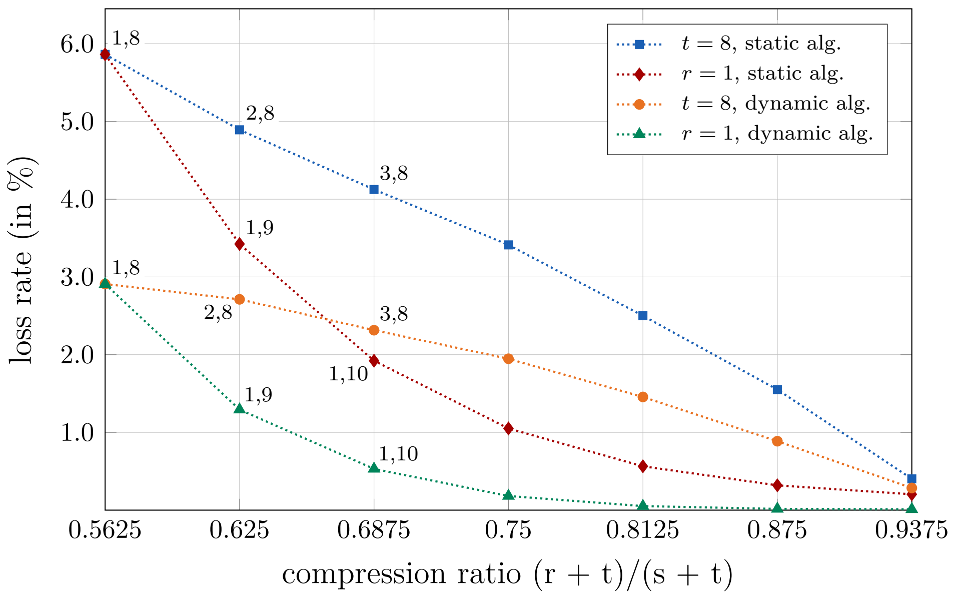


Figure 6.5: Comparison of the static and the dynamic version of the cache-based compression algorithm for different parameters of r and t.

Figure 6.5 compares the performance of the static cache-based algorithm with the dynamic cache-based algorithm for different parameters of r and t. According to the legend, the blue line with the square markers and the orange line with the circle markers are computed keeping the tail size t (resp., the head size s) constant, i.e., the increasing r determines the compression ratio in terms of an increasing dictionary size. For the red line with the diamond markers and the green line with the triangle markers, on the other hand, the dictionary size (i.e., r) is kept constant, while the compression ratio is determined by the increasing t (resp., decreasing s). In the figure, the first three data points of each line are annotated with r and t; for each additional data point, the variable parameter increases by 1, e.g., the first three squares of the blue line where r is variable are labeled $1, 8$, $2, 8$ and $3, 8$. From the figure it can be concluded that the static cache-based compression algorithm can outperform its dynamic version depending on the parameters, e.g., at a compression ratio of 0.75, the red line with the diamond markers $(r = 1, t = 11)$ lies below the orange line with the circle markers $(r = 4, t = 8)$. The reason for the lower performance of the dynamic cache-based algorithm in this case arises from the individually maintained offsets of the multiple active heads, which leads to an unfavorable coverage of neighboring data values around the current data value. In turn, the diamond marker at a compression ratio of 0.75 is produced maintaining only one active head $(r = 1)$ that consequently accounts for a broad coverage of values around the current data value.

The strategy of reducing the loss rate by using the $t - s$ unused bits of the reserved head in order to store more active values in the dictionary (Section 5.1.2.2) can be applied to the static as well as to the dynamic version of the cache-based compression algorithm.

Evaluations with test signals revealed that the loss rate can be approximately decreased up to 1 % at the different compression ratios for the test signals, where the impact of the algorithm improvement is more significant for the static cache-based algorithm.

To preserve as much information as possible in a loss case, the strategy of reducing the uncertainty from Section 5.1.2.1 can be applied here and also to the simultaneous compression of multiple data streams. It is directly related to requirement R4 (guaranteed data quality with respect to time triggered messages) from Section 3.1. Through bounded omissions, R4 is already satisfied by the design of the data compression algorithms, but the strategy improves the preservation of information.

6.2.3 Loss Rates of Multiple Simultaneously Compressed Data Streams

6.2.3.1 Transmission Regions

A key element of the algorithms for simultaneous compression of multiple correlated data streams is the strategy of grouping active hypercubes to form a transmission region. The performance is significantly affected by the definition of an appropriate shape of an active region in the sense of Section 5.2.5.

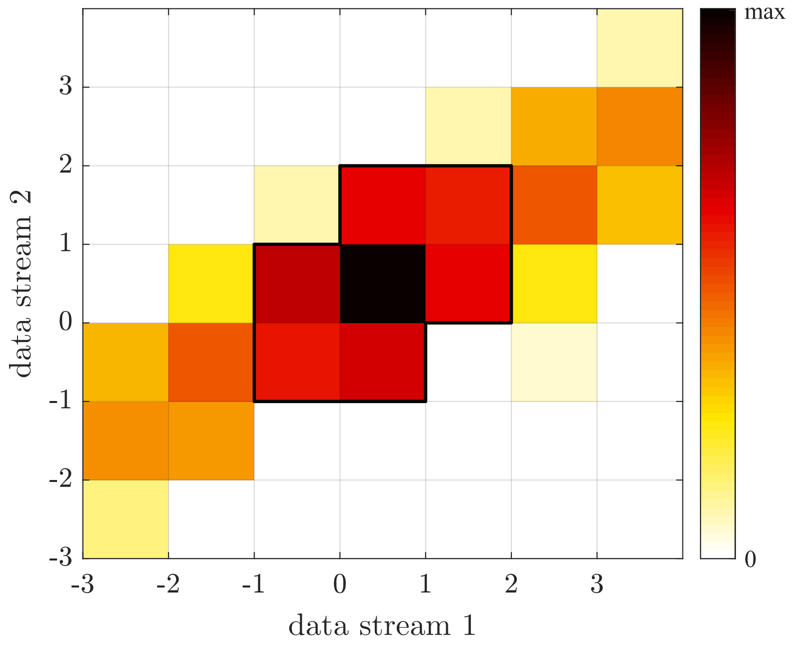


Figure 6.6: Heat map (logarithmic scale) of the hitting squares around the center. The active region consists of 49 hypercubes (here squares).

The heat map in Figure 6.6 shows a region with $7 \times 7 = 49$ squares and the relative center $u_1 = u_2 = 0$. The experiment uses the DC bus voltage and current signals from the Simulink HEV model as introduced in Section 6.2.1.1. For the analysis, $r = 6$ was set and only 49 out of the available 63 dictionary entries were used. According to Section 5.2.3, the region is always centered around the last transmitted data tuple. From darker to lighter colors, one sees that consecutive data tuples are covered by squares that lie on a diagonal of the region. This is to be expected if two signals rise and fall in a similar manner. Based on this analysis, a subset of the squares shown can be used for smaller dictionary sizes (see Table 5.4), or more squares can form larger regions. A white square indicates that no input sample was covered by that square, meaning that this square is dispensable and can be removed from the region without increasing the number of lost data values.

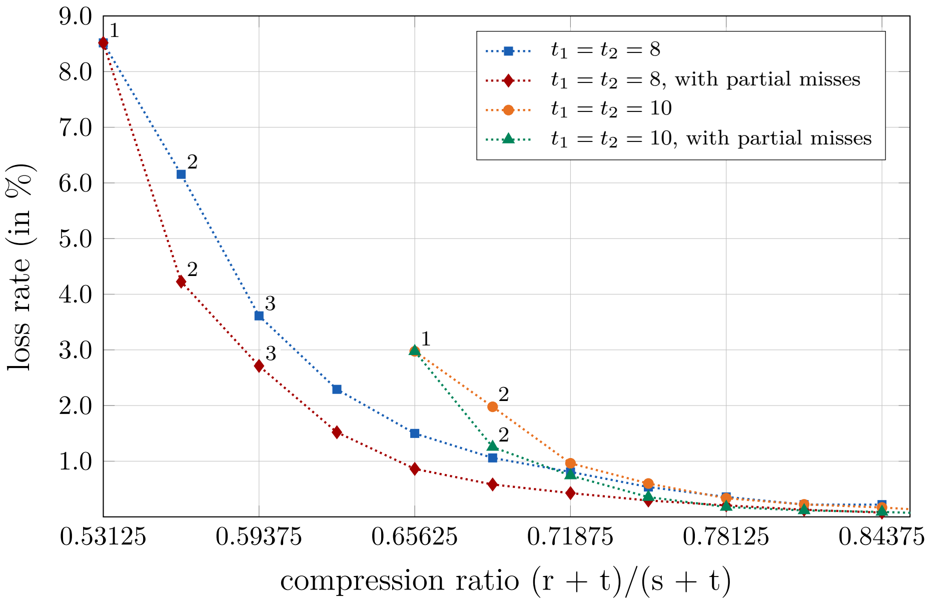


Figure 6.7: Simultaneous compression of two data streams. Comparison of the cache-based algorithm with its improved version supporting partial misses.

Figure 6.7 demonstrates a two-dimensional data compression scenario where the correlated DC bus signals from Section 6.2.1.1 are simultaneously compressed using the static cache-based algorithm (Section 5.2.2) extended by the grouping technique from Section 5.2.5. One predefined active region is maintained for all dictionary sizes. The figure presents the relationship between the loss rate (the total number of lost data values divided by the total number of data values in all streams) and the compression ratio for different settings for the parameters r and t. Again, the values for r are written to the first three data points of every line in Figure 6.7 and in all following figures that evaluate the loss rate; r increases by 1 for every further data point. The figure also highlights the strategy from Section 5.2.4 (partial misses), which reduces the loss rate for all compression ratios, e.g., compare the blue line with the square markers with the red line with the diamond markers. Since $s \leq t$, the lowest possible compression ratio for the two settings is $(r + t_1 + t_2)/(s_1 + s_2 + t_1 + t_2) = (1 + 16)/32 = 0.53125$ and $(1 + 20)/32 = 0.65625$, respectively. The benefit of the strategy becomes less significant at higher compression ratios, as a larger number of grouped active hypercubes inherently provides a better value coverage around the current data tuple. Further evaluations also showed that while the strategy can be applied with the dynamic version of the compression algorithm as well, the positive effect on the loss rate is less significant than with the static version of the algorithm.

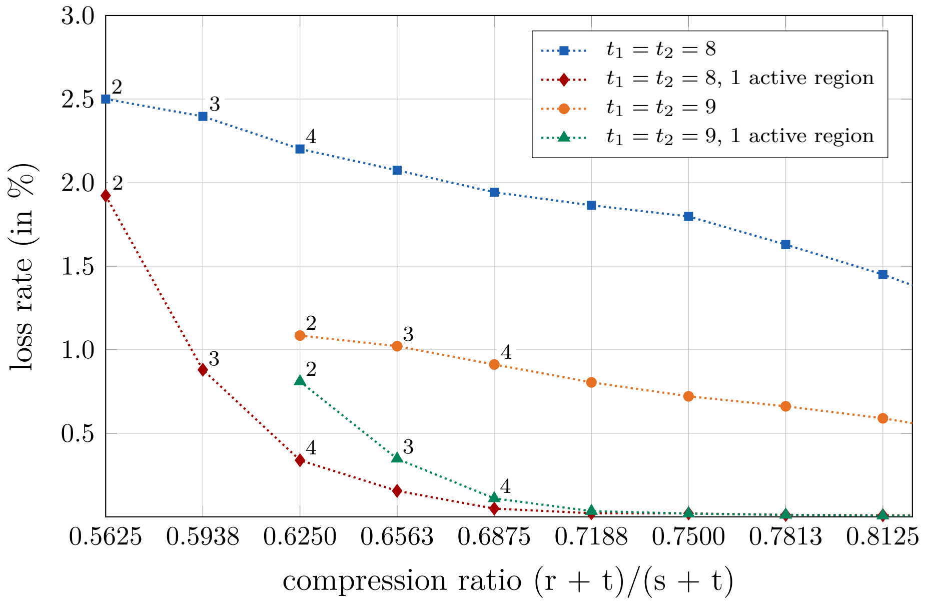


Figure 6.8: Simultaneous compression of two data streams. Comparison of individually maintained hypercubes and a region of hypercubes.

Figure 6.8 demonstrates the improvements of the dynamic simultaneous compression of two data streams (Section 5.2.3) when the hypercubes in the dictionary are maintained as one region according to Section 5.2.5. For the tail lengths $t_1 = t_2 = 8$ (compare the blue line with square markers with the red line with diamond markers), the loss rate drops from 2.2 % to 0.34 % at a compression ratio of 0.625. Moreover, one sees that larger dictionaries with smaller hypercubes (here squares) enable to better adjust them to the signal, e.g., at a compression ratio of 0.625, the red diamond marker is below the green triangle marker.

Figure 6.9 compares the loss rate of the data transmission of the described test signals for differently shaped fixed regions. In Figures 6.9 and 6.10, the values for r are written to the first three data points of the red lines with the diamond markers. They are valid for all lines at the corresponding compression ratio. Moreover, the side lengths of the grid forming the two-dimensional observation region (here $\sqrt{n_{\text{obs}}}$) are written in brackets and

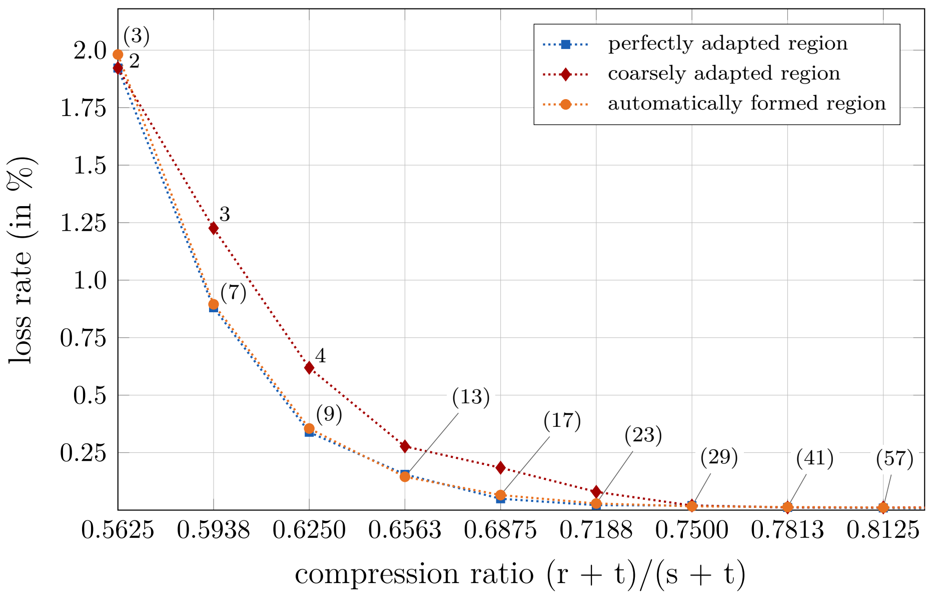


Figure 6.9: Comparison of different fixed regions with the automatic region forming strategy for the voltage and current test signals.

belong to the orange lines with the circle markers. For example, at a compression ratio of 0.5938, $r = 3$ and the region consists of at most $2^3 - 1 = 7$ squares in the two-dimensional product space.

Recalling Figure 6.6, the analysis of the hit rates reveals the 7 squares within the perimeter to be the best solution to form a region, whereas any other combination of 7 squares would lead to significantly higher loss rates. This procedure is analogously applied for all simulated region sizes. Based on these predefined regions, which show excellent sample coverage, the blue line with the square markers is processed according to the dynamic simultaneous compression (Section 5.2.3) and shows the lowest loss rate. The red line with the diamond markers represents a more realistic scenario, where more general shapes of regions are applied that better cover the sample's neighborhood in directions other than the predominant one. This has an advantage in terms of robustness, but comes at the cost of a higher loss rate. Still, certain knowledge is involved in the region formations, i.e., consecutive samples typically describe a movement on the diagonal from bottom left to top right, and so regions are set to exhibit symmetry around a line with the slope 1 in the two-dimensional product space. If no predominant direction of the samples in the product space can be expected within a certain time interval (i.e., no strong dependence between the signals), the compression strategies still work but will not be able to reach their full potential. The orange line with the circle markers is processed according to the strategy from Section 5.2.6.2 with the extension from Section 5.2.6.3 (automatic grouping of active hypercubes supporting dynamic transmission dictionary updates). This combination shows the best performance, as the dynamic transmission dictionary updates explicitly help to reduce the number of lost data values within the observation region. It is concluded that this method provides a loss rate basically as low as the manual strategy based on prior signal analysis, but here no prior knowledge of the expected signal behavior is needed.

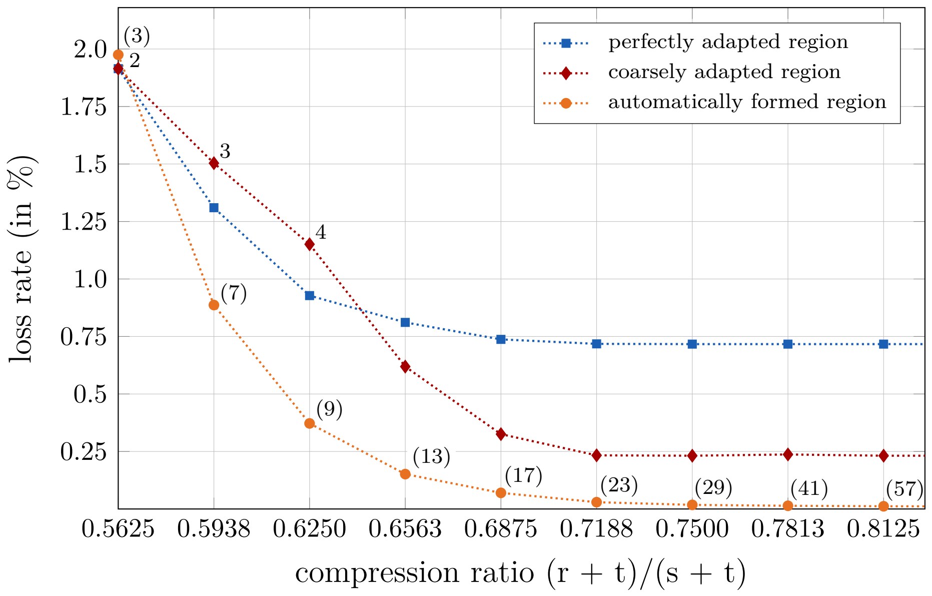


Figure 6.10: Comparison of different fixed regions with the automatic region forming strategy for the manipulated voltage and current test signals.

The major advantage of the automatic grouping strategy (Section 5.2.6) becomes visible in Figure 6.10. For demonstration purposes, one of the test signals was partly modified to force a change in behavior in the two-dimensional product space over time, i.e., the current signal (the densely dashed red line in Figure 6.2) is inverted for about half of the time so that the signal correlation is reversed. Such signal modifications exemplify changing signal behavior over time. Simulating a transient or a permanent fault in a sensor, e.g., by holding the values of a data stream for a certain time, is also a situation where the correlation between multiple signals varies and decreases. It is apparent that the previously introduced fixed regions perform substantially worse in this scenario. For instance, comparing Figures 6.9 and 6.10 at a compression ratio of 0.75, the loss rate of the blue line with the square markers increases from a value close to 0 % to about 0.75 %. The automatically formed regions are able to adapt to the signal changes, so that the orange line with the circles has about the same low loss rate for both scenarios.

Figure 6.11 visualizes an example of the procedure of forming an active region using the PCA strategy from Section 5.2.6.2 for $d = 3$ data streams. The active region is supposed to consist of 31 hypercubes. Only the hypercubes marked in blue showed hits, yielding an incomplete transmission region. The hypercubes marked in red then complete the transmission region. The transparency of the printed hypercubes allows to recognize the first three principal components and their origin. The lengths of the vectors are determined from the minimum and maximum values, respectively, of the point set representing the centers of the hit hypercubes after transformation into the principal component space (recall Section 5.2.6.2).

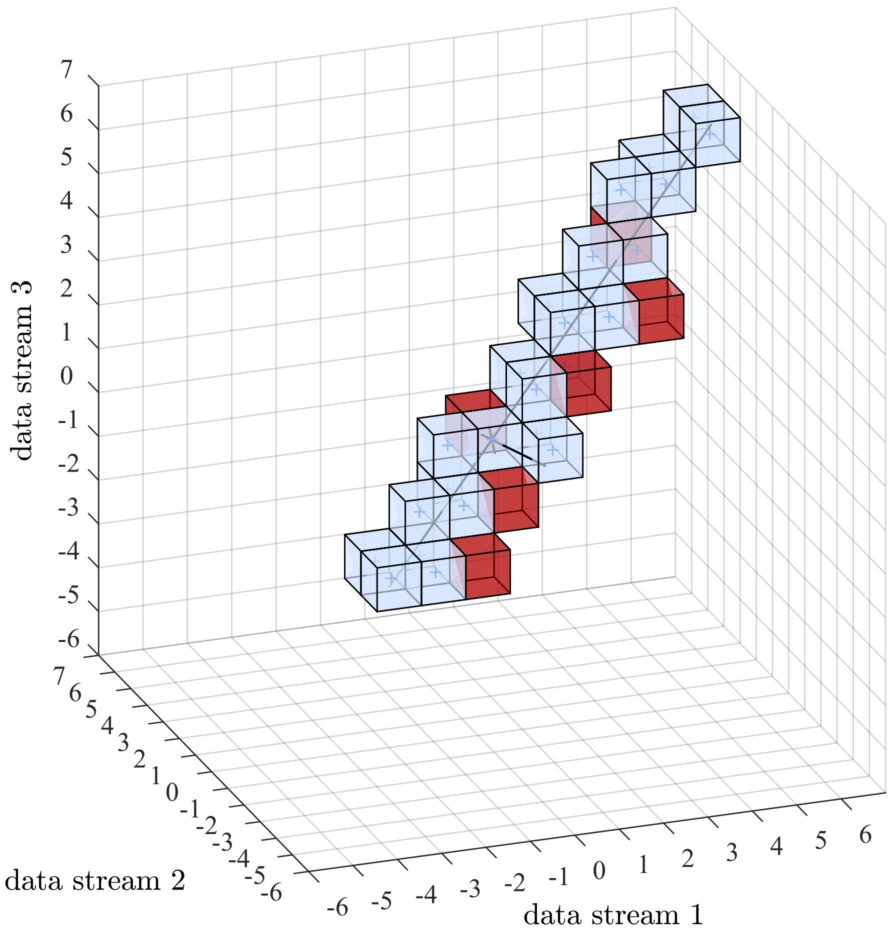


Figure 6.11: Formation process of an active region with the PCA strategy for $d = 3$.

The evaluations show that the algorithms handle the compression of data without prior statistical knowledge (requirement R6 from Section 3.1). The compression can be initialized with an empty dictionary, and as the signal behavior changes over time, the automatic grouping strategy adapts to the signal variations in the multidimensional product space.

6.2.3.2 Scalability of Simultaneous Data Compression

While the previous sections used one- or two-dimensional data compression scenarios to illustrate and evaluate specific working principles of the algorithms and their extensions, this section examines the scalability of the data compression for the case where also more than two data streams are simultaneously compressed.

By design, the data compression algorithms are able to provide a WCCR on short input sequences (requirement R2 from Section 3.1) for any number of simultaneously compressed data streams. Moreover, the working principle that maps fixed-size inputs to fixed-size outputs is consistent with R3, which requires one-pass data compression.

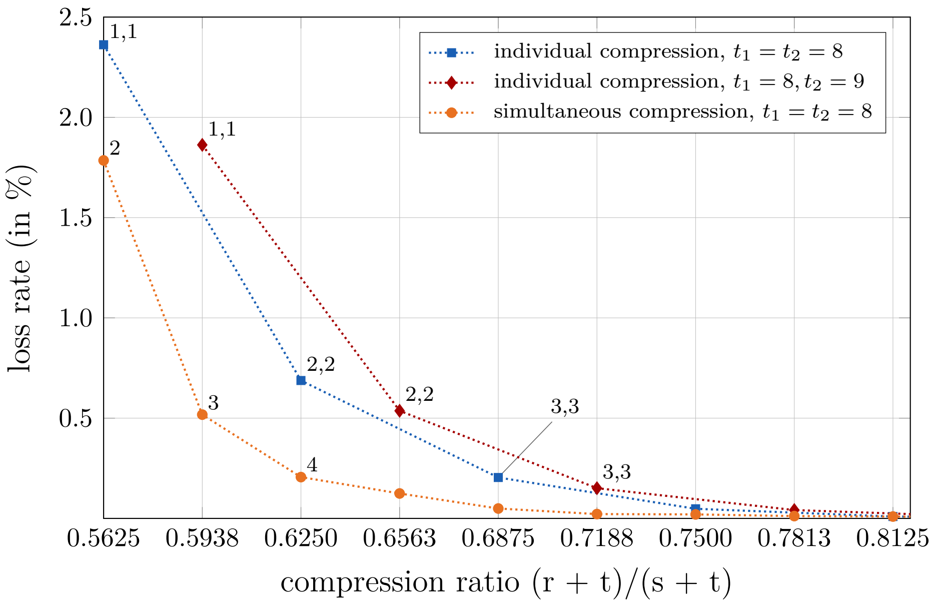


Figure 6.12: Comparison of the dynamic simultaneous compression of two data streams with the compression of the individual data streams.

Loss Rates

Firstly, Figure 6.12 compares the simultaneous compression of the two test data streams (according to the dynamic cache-based algorithm with the improvements from Section 5.2.4 and Section 5.2.5) with the two data streams compressed individually (dynamic cache-based algorithm with $d = 1$ and grouping according to Section 5.2.5), where the loss rate is calculated from the overall number of lost data values in both streams divided by the overall number of transmitted data values. To have comparisons for all compression ratios, two simulations were performed for the individual stream compressions, one where the tail sizes are $t_1 = t_2 = 8$ (blue line with square markers) and another with $t_1 = 8, t_2 = 9$ (red line with diamond markers). The results show that the simultaneous compression of the two data streams outperforms the individual data stream compression up to compression ratios of about 0.75, since the orange line with the circle markers is the lowest. Furthermore, the calculation of the loss rates for the two individually compressed data streams (square markers and diamond markers) actually hides the information that one data stream potentially suffers significantly more losses than the other. So another advantage of simultaneous data compression is that both data streams share the available bits in terms of the shape of the transmission region.

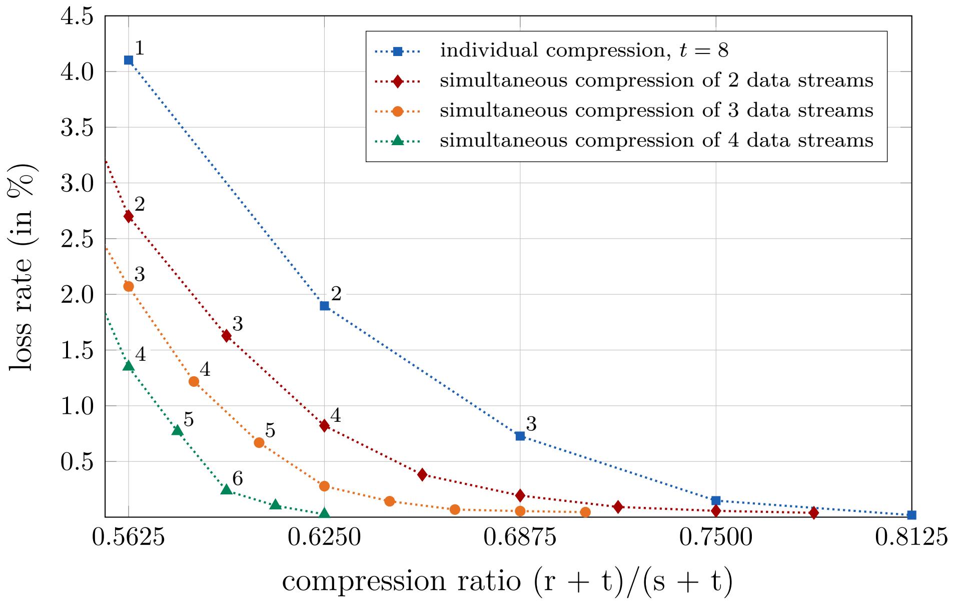


Figure 6.13: Analysis of the scalability of multiple simultaneously compressed data streams.

Figure 6.13 uses a scenario where $d = 4$ correlated data streams have approximately the same loss rate when individually compressed (i.e., $< \pm 0.25\,\%$ difference) and Pearson correlation coefficients (measured over the entire lengths of the input signals) of greater than 0.95 between any two data streams. Detailed information about the four test signals can be found in Appendix A.1. The blue line with the square markers exemplarily represents one of these signals. In the setup, $t_1 = t_2 = t_3 = t_4 = 8$ bits. The other signal parameters are as described above for the test signals, i.e., 16 bits per sample. It can be seen that a simultaneous compression of two of these data streams significantly reduces the loss rate for all compression ratios in the region of interest up to a compression ratio of 0.8125. For example, at a compression ratio of 0.5625, the loss rate decreases by 1.4 %. This improvement originates from the use of a common dictionary that enables hypercube grouping in combination with the fact that the correlated signals are scattered around a low-dimensional subspace of the d-dimensional product space of all the data streams. From the orange line with the circle markers and the green line with the triangle markers in the figure, it can be further concluded that this effect becomes more significant with more correlated data streams included in the simultaneous compression. All results are calculated using the dynamic cache-based compression algorithm with automatic grouping of active hypercubes.

One can also evaluate the benefits in terms of bit savings. Assuming a loss rate of below 1 % is demanded (i.e., the respective marker must be below 1), compressing the four data streams individually per sample is possible at a compression ratio of $(4 \cdot (3 + 8))/(4 \cdot 16) = 44/64 = 0.6875$. This means that over 30 % of bits can be saved compared to an uncompressed communication. Taking the red line with the diamond markers as an example and compressing two times two of the data streams simultaneously, only $2 \cdot (4 + 2 \cdot 8) = 40$ bits are needed, so a compression ratio of $40/64 = 0.625$ becomes possible. If all four data streams are simultaneously compressed, even more bits can be saved, enabling a compression ratio of $(5 + 4 \cdot 8)/64 = 0.578$. Compared to compressing the data streams individually, this is another saving of about 16 % of the bits.

Consecutive Losses

The loss rates shown in the previous figures consider the accumulated number of loss cases related to a certain number of samples. It does not reveal whether these losses were scattered or consecutive. Therefore, interpreting the number of consecutively lost data values in a data stream is an important evaluation. Recall that in a loss case, the uncertainty of the correct data value from the i-th stream is only half of its potential range of $2^{t_i}/2$ if the center value corresponding to the missed head is reconstructed. For many applications, this uncertainty is acceptable if it does not happen too many times in a row. Such losses might only indicate harsh and short-lived signal conditions, e.g., voltage spikes. In turn, if an online analysis of loss occurrences monitors multiple such incidences, the signal behavior might have permanently changed, such that the established transmission region became inappropriate. The strategy for automatic grouping active hypercubes from Section 5.2.6 tackles this case. In any case, no additional channel is needed to exchange overhead information in case of a loss, as demanded by requirement R5 (from Section 3.1) to be able to strictly adhere to scheduled communication.

The stacked bar charts in Figure 6.14 visualize the occurrence of consecutive losses with respect to all loss situations. Based on the compression scenario of Figure 6.13, the bar charts are exemplarily computed for the loss occurrences at a compression ratio of 0.625. It was concluded from Figure 6.13 that at a specific compression ratio, the loss rate decreases with more correlated data streams simultaneously compressed. Besides this positive effect, the color-coded percentages in Figure 6.14, which refer to the number of consecutive losses, emphasize another benefit. For instance, the dark blue portions of the stacked bars denote the percentage of loss cases where only *one* data tuple is lost before the active region again covers further data tuples, e.g., for $d = 1$, this happens only in 21 % of the loss cases. The percentages in each stacked bar add up to 100 %. The parameter d, which increases from top to bottom, indicates the number of simultaneously compressed data streams.

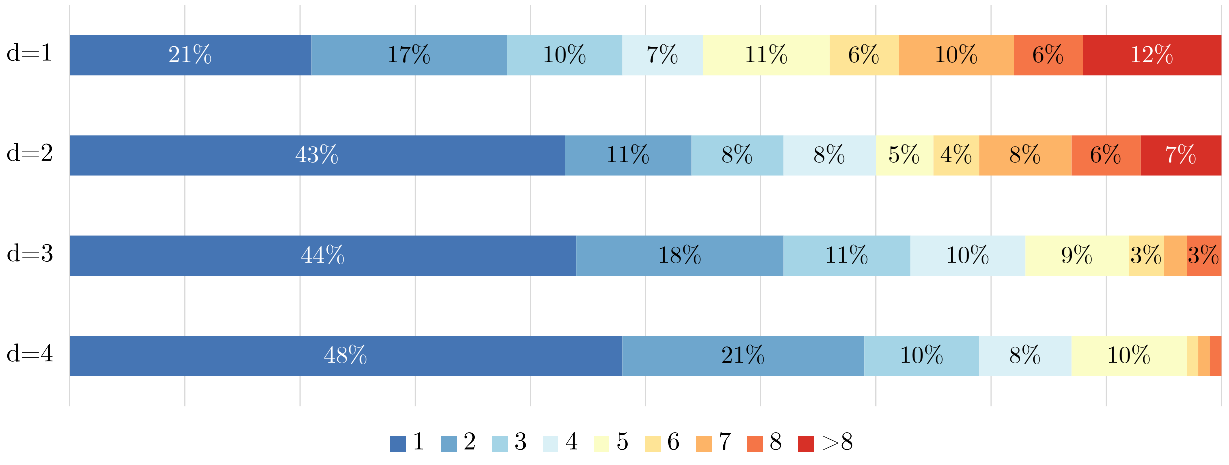


Figure 6.14: Consecutive losses of different numbers of simultaneously compressed data streams.

In the example with the tail sizes $t_1 = t_2 = t_3 = t_4 = 8$ bits and sample sizes $n_1 = n_2 = n_3 = n_4 = 16$ bits, the uncertainty of the correct code word in each of the d directions is only $2^{t_1}/2 = 128$ values of the overall code word range of $2^{n_1} = 65536$ values. With respect to the goal of preserving as much information as possible during data compression, fewer consecutive loss occurrences are targeted.

From the upper to the lower bar, it can be seen that, overall, the proportions of single loss events or few consecutive loss events increase, while at the same time the unfavorable cases with many consecutive loss events significantly decrease. Recalling Figure 6.13, as indicated by the different r values, the dictionary to achieve the same compression ratio is larger when more data streams are compressed simultaneously. When the input signals are correlated, this offers more hypercubes to better fit the data tuples in the multidimensional product space. Proportionally, a larger active coverage region capable of reducing situations with many consecutive losses can thus be provided.

6.2.3.3 Signal Selection for Simultaneous Data Compression

The strategy that automatically adapts the active region to the input signals in the d-dimensional product space helps to achieve beneficial compression results in terms of a minimized loss rate, even if the correlation between the involved signals is nonconstant or low.

Figure 6.15 analyzes the impact of signal selections for a simultaneous data compression with respect to their Pearson correlation coefficient. Four signals based on the test dataset from Section 6.2.1.1 are used for the analyses, two of which each are simultaneously compressed. Let the signals be denoted as S_1, S_2, S_3, and S_4. Again, the number of bits per

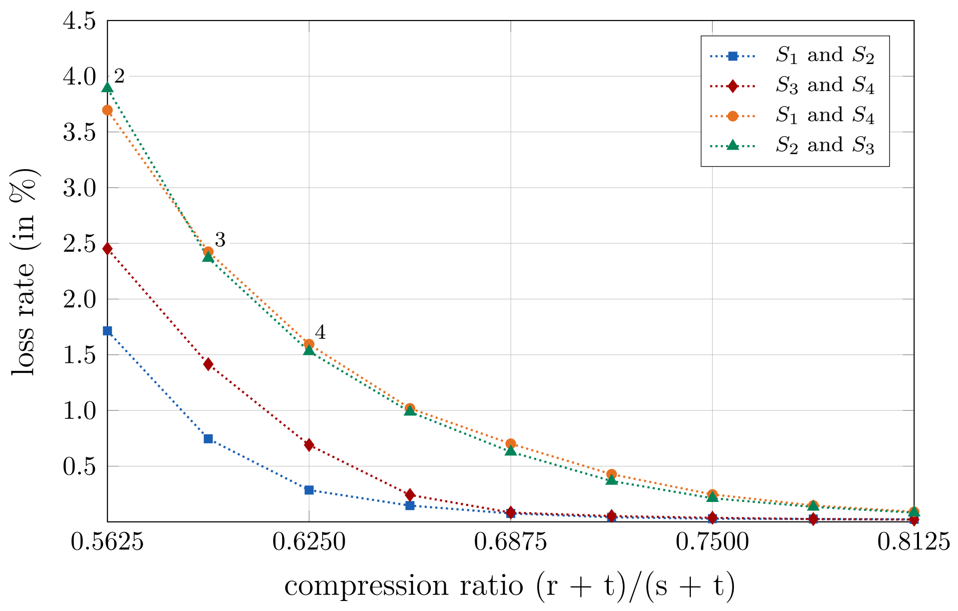


Figure 6.15: Selection of suitable signals for simultaneous data compression.

sample is 16 and for the compression, the tail lengths are $t_1 = t_2 = t_3 = t_4 = 8$ bits. The loss rates of these four individually compressed signals can be found in Appendix A.1. The Pearson correlation coefficients between S_1 and S_2 and between S_3 and S_4 are constantly high (> 0.95). For all other pairing possibilities, the absolute value of the Pearson correlation coefficient is less than 0.67. The blue line with the square markers and the red line with the diamond markers in Figure 6.15 show that the benefit of a simultaneous compression is significantly higher when the signals are more correlated. The loss rates in this case are lower for all compression ratios compared to the other two lines in the figure, which show the result for a different pairing setup. For a proper selection of advantageously compressible signals, prior analyses on achievable loss rates with respect to the compression ratio are important information for a scheduler, as described in Section 4.4. It is thus important for the online data compression algorithms to track the signal correlations and the loss rate over time. The analyses show that the data compression algorithms are able to exploit redundancies among multiple data streams to achieve an optimized compression performance, as demanded by requirement R7.

6.2.4 Time Considerations for Dictionary Searches

The simultaneous compression of more data streams naturally requires more computational effort for the involved processing steps. When implementing the dictionary using a hash table, finding a match takes on average constant time, i.e., $O(1)$ in big O notation [Cor+09, p. 253]. However, due to potential hash collisions, in the worst-case all dictionary entries must be checked, so the access time behavior becomes $O(n)$ when the number of entries is n [Cor+09, p. 253]. There is also a technique called perfect hashing. By using static keys, it is possible to achieve a worst-case access time behavior of $O(1)$, but this involves larger hash tables [Cor+09, pp. 277–278]. The choice of an implementation design must therefore be evaluated with consideration of the cost of communicating messages through a distributed system. In general, considering the time complexities of dictionary-based data compression algorithm implementations, the online compression algorithms developed in this thesis are capable of compressing and decompressing data in real-time, thus fulfilling requirement R1 from Section 3.1.

6.2.5 Comparison with Other Data Compression Techniques

6.2.5.1 Compression of Individual Data Streams

Given the related work and the requirements discussed in Chapter 3, differential encoding schemes are closest to the cache-based compression algorithm developed in this thesis and therefore serve for performance comparisons. Similar to the cache-based algorithm, the modified delta encoding scheme from Section 5.1.7 exploits the fact that differences between consecutive samples of sensor measurements of many physical quantities are often small in practice (considering a sufficiently high sampling rate) and can thus be encoded with fewer bits. As an improvement over classical delta encoding, it is able to automatically recover a reference value and it allows a limited uncertainty range in the case of a loss.

In a direct comparison of the achievable loss rates of individual data stream compressions, both the compression algorithms perform equally well for different tail sizes, i.e., with the same bounded uncertainty. However, the cache-based compression algorithm is more complex. To achieve equivalent results, grouping of active hypercubes (in one dimension) as well as the algorithm extension that increases the dictionary size to 2^r from Section 5.1.2.2 must be applied.

In any case, delta encoding requires a reference value to which the differences refer. If this is corrupted, or in the implementation of the modified difference coding algorithm from Section 5.1.7, if a next difference value cannot be encoded based on a reconstructed reference, multiple consecutive losses are to be expected. An advantage of the modified difference coding algorithm is that lower compression ratios can be realized, since the compression ratio does not depend on the choice of the parameter t (i.e., the tail size). This is different from the cache-based compression algorithm, where the minimum achievable compression ratio depends on t and is limited due to $t \geq s$. However, evaluations with small compression ratios showed that the loss rate increases drastically, limiting the potential usefulness of low compression ratios for practical implementations.

6.2.5.2 Simultaneous Compression of Multiple Data Streams

Due to the specific design of the dictionary of the cache-based compression algorithm, specific transmission regions can be formed according to Section 5.2.5. As the evaluations from the previous sections show, this significantly improves the compression performance in terms of reduced loss rates and fewer consecutive losses. This property is not possible to be realized with the modified difference coding algorithm. It can only form one large hypercube in the multidimensional product space whose side lengths correspond to the dictionary sizes, but this is basically the same as compressing multiple data streams individually.

In general, the following benefits of the cache-based compression algorithm compared to the modified difference coding algorithm emerge: (i) the cache-based algorithm shows better scalability for compressing multiple data streams simultaneously. (ii) due to the possibility of forming arbitrary transmission regions, multiple such regions can be simultaneously active and beneficially cover signals that frequently jump between different levels. (iii) the cache-based algorithm is more robust since it does not rely on a reference value to reconstruct an encoded data value.

It is also a unique feature of the cache-based algorithm that, by design, it is able to merge and split compressed data streams at arbitrary nodes in distributed systems, as described in Section 5.3. The dynamic dictionaries in combination with the automatic grouping of hypercubes into multidimensional transmission regions thus enable the algorithm to fulfill requirement R8.

6.3 Data Compression and Information Redundancy

As mentioned in Section 2.8.1, the general idea of data compression is to remove unwanted redundant information in terms of bits from some input data. On the other hand, in the context of fault-tolerant system design, information redundancy was described as the strategy of adding extra bits to some input data in order to be able to verify data correctness or even to restore corrupted data (i.e., wanted redundancy). By combining these strategies, information redundancy can be achieved without increasing the original data size. This is demonstrated in the following using the developed data compression algorithms.

Recall Section 2.6.1.2, linear block codes such as Bose-Chaudhuri-Hocquenghem (BCH) codes are often used in the context of forward error correction (FEC). From a certain fixed number k of input bits, they produce a larger fixed number n of output bits, thus providing error detection and correction capabilities. The code is said to act on *blocks* of data bits. The construction rules of a block code define the possible relation between the k information bits and the $n - k$ redundancy bits. For example, in a t-error-correcting BCH code, the relation is given by two integers m and t, where $n = 2^m - 1$ with $m \geq 3$ and $k = n - mt$ with $t < 2^{m-1}$ [RL09, p. 111].

The online data compression algorithms of this thesis are designed to work on short input data sequences in order to meet the requirements from Section 3.1, especially to enable real-time capabilities. Consequently, the output size n of an applied block code must not be large. Assuming a scenario where single bit errors (i.e., $t \geq 1$) in $k = 16$ bit data values must be tolerated using a BCH code, one must have $m \geq 5$ in which case $n = 31$. Then $t \leq \lfloor (n - k)/m \rfloor$ (here $t \leq 3$) single bit errors can be corrected, but at the cost of almost doubling the amount of input data bits. To achieve a smarter utilization of information bits with respect to the requirements (i.e., $t = 1$ in this example), consecutive input data values can be distributed among multiple output blocks or aggregated into one larger output block if a certain delay is acceptable. For example, if $n = 31$ and $t = 1$, there are $k = 26$ information bits available per block, so it is possible to transmit 3 input data values (i.e., 48 information bits) using $2 \cdot n = 62$ output bits (i.e., 2 blocks) of the BCH code.

Applying data compression before adding redundant bits via a block code generally enables smarter utilization of information bits (as long as the tolerable loss rate is not exceeded). By adjusting the WCCR, the compression algorithms allow the compressed data to better fit the available number of information bits per block (or across multiple blocks). In the above example, if data values are compressed from 16 to 11 bits prior to the FEC, a block output size of $n = 15$ including 4 redundancy bits is sufficient to correct a single bit error while a compression ratio below 1 is still achieved. The strategy can be straightforwardly applied to the simultaneous compression of multiple data streams.

The adjustment of the WCCR to optimally make use of the information bits of a block code can be done by either modifying the dictionary size or the tail sizes of the data streams, which has a different effect. As discussed in Section 5.1.2, whether to aim for a smaller uncertainty of the correct data value in a loss case or a reduced loss rate depends on the application.

It should be noted that there are other strategies besides block codes to achieve information redundancy. For example, convolutional codes can operate bitstream-based rather than block-based, so the above considerations play only a minor role. Nevertheless, according to the rate of a convolutional encoder, several output bits are generated for each input bit, so a benefit of compressed data is directly evident due to the reduced data size.

6.4 Influence of Compressed Communication on Schedules

The design of scheduling algorithms that aim for determining an optimal allocation of tasks to computation nodes, task execution times, message injection times and paths for the messages through the network, goes beyond the scope of this thesis. In the context of the DAKODIS research project, other researchers have addressed the field of scheduling, see e.g., [LO17; MO19]. The following considerations are based on examples for which no specific scheduling algorithm is required. Yet, the influence of compressed communication on schedules can be shown and generalized to some extent.

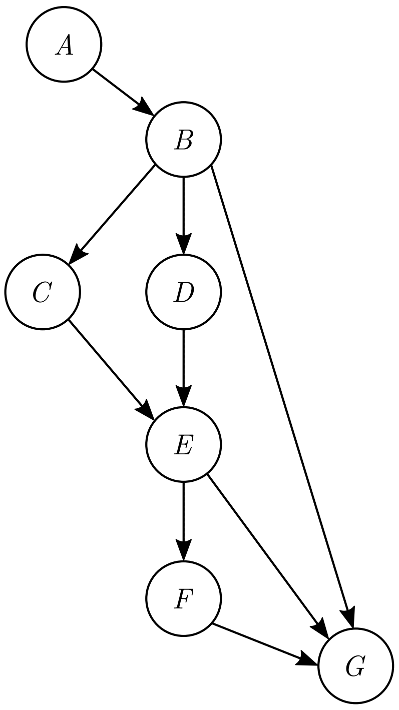


Figure 6.16: Logical model in terms of a directed acyclic graph with 7 tasks.

For the DAKODIS research project, several diagnosis use cases have been prepared using the fault injection capabilities of the extended Simulink HEV model introduced in Section 6.1, see [Jo+17; MOY18; Mec+20b]. The diagnostic directed acyclic graph (DAG) from Figure 6.16 is based on [Jo+17] and was modeled explicitly using expert domain knowledge. An approach to implicitly model a DAG for a fault diagnosis use case based on machine learning is presented in [Mec+20b]. Since the following considerations have a focus on scheduling, the explicit diagnostic purpose of the logical model from Figure 6.16 is not relevant. It serves as an example of a logical model which gets scheduled to a network in order to analyze the computation times and communication times of the tasks and messages, respectively.

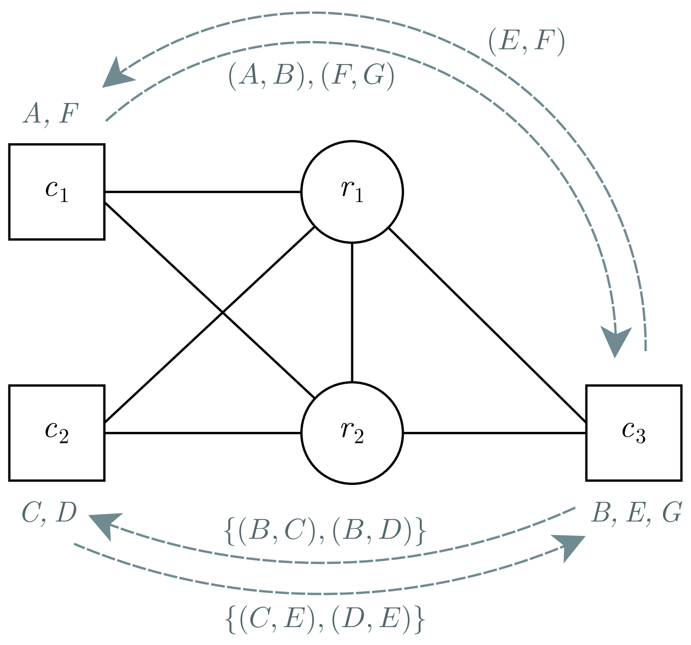


Figure 6.17: Network with allocated tasks and communication.

Figure 6.17 shows a network with the 7 tasks $A, B, \ldots, G$ of the logical model assigned to the three computation nodes (c_1, c_2, and c_3) and the communication links. For simplicity, the network is designed in a way that each channel can use two completely disjoint routes via the routers r_1 and r_2. Due to the network design and the allocation, there is no need to consider collisions. The stated allocation was chosen for demonstration purposes. Note that it does not exploit the concurrency of tasks C and D and is therefore not optimal. The dashed directed edges between the involved computation nodes indicate the communication pattern of the diagnostic DAG. The edges are labeled with channels (recall Section 4.3). A group of channels surrounded by curly braces means that these channels are mapped to disjoint communication paths (via the two routers). For the computation of the makespan, such a group can be consequently considered as one channel. Note that the two channels (B, G) and (E, G) are not present in Figure 6.17 because the three tasks B, E, G are mapped to the same computation node. In the following, the term *tick* again refers to a fraction of real time and is used to express a duration of time that abstracts over a real unit of time, e.g., microseconds. For example, assuming a worst-case execution time (WCET) of 6 ticks per task and a per-hop transmission time of 12 ticks per message, the makespan is $7 \cdot 6 + 5 \cdot (2 \cdot 12) = 162$ ticks (i.e., 7 tasks, each requiring 6 ticks, and 5 (groups of) channels, each mapped to a communication path of length 2). Note that the two topological orderings (which result from the two different orderings of C and D) have no influence on the makespan.

Whether the goal of decreasing the makespan can be achieved depends on the compression parameters. Assuming a $WCCR = 0.75$ for all communications and worst-case compression (resp., decompression) times $WCCT = WCDT = 1$ tick, this is possible. Being the first task, A only has to handle one compression for channel (A, B) and analogously G has to handle only one decompression for channel (F, G), so both tasks end up with accumulated WCETs of 7 ticks. Tasks C, D and F must each perform one compression *and* one decompression, so their WCETs become 8 ticks each. The remaining tasks B and E must process two compressions *and* one decompression (B) and one compression *and* two decompressions (E), respectively. So both tasks have a WCET of 9 ticks. The per-hop transmission time of the messages is reduced according to the WCCR to $12 \cdot 0.75 = 9$ ticks. The makespan of the DAG with compression becomes $2 \cdot 7 + 3 \cdot 8 + 2 \cdot 9 + 5 \cdot (2 \cdot 9) = 146$ ticks, which is a reduction of about 10 %.

In the above example, the specific choice of the compression parameters in relation to all other parameters (e.g., WCETs of tasks, message sizes) helped to reduce the makespan. In the following, the effect of the compression parameters on the makespan is analyzed from a general perspective. For this purpose, the involved parameters are brought into relation with each other.

The support for compressed communication is the main goal of the time-triggered DAKODIS architecture. System improvements from compressed messages can be of different kinds, e.g., (i) optimized (shorter) schedules allow tighter deadlines to be met and support shorter service times to be provided, (ii) reduced network traffic enables higher levels of integration and combining multiple services in one system, or (iii) saved data bits can be used for FEC to increase the system reliability and availability.

The following analyses have a focus on the first-mentioned point. In the DAKODIS architecture, time savings with respect to a schedule can only be expected if the reduction of transmission times is more significant than the extra times needed for data compressions and decompressions (recall Figure 4.4). Since data compression only affects messages, it seems natural that the higher the proportion of communication time relative to task computation time (in a schedule), the greater the potential for system improvements.

In order to generalize the influence of relevant system parameters, in the following experiment a logical model is assumed to be a directed simple graph with n nodes and $n - 1$ edges, such that the first edge directs from node 1 to node 2, the second edge directs from node 2 to node 3, and so on. In this way, no concurrency of task executions or message injections is possible, and the makespan can be calculated straightforwardly by considering the time needed for computations (of tasks) and the time needed for communicating messages (all in ticks). The total time for computations in the makespan is the product of the *number of tasks* and the *ticks per task* (i.e., the WCET of the task), which is increased by the WCCT and WCDT if messages are compressed and decompressed, respectively. The total time for communications is the product of the *number of messages*, the *number of hops per message* (in a network), and the *transmission time per hop*, which is weighted by the WCCR in the case of compressed messages.

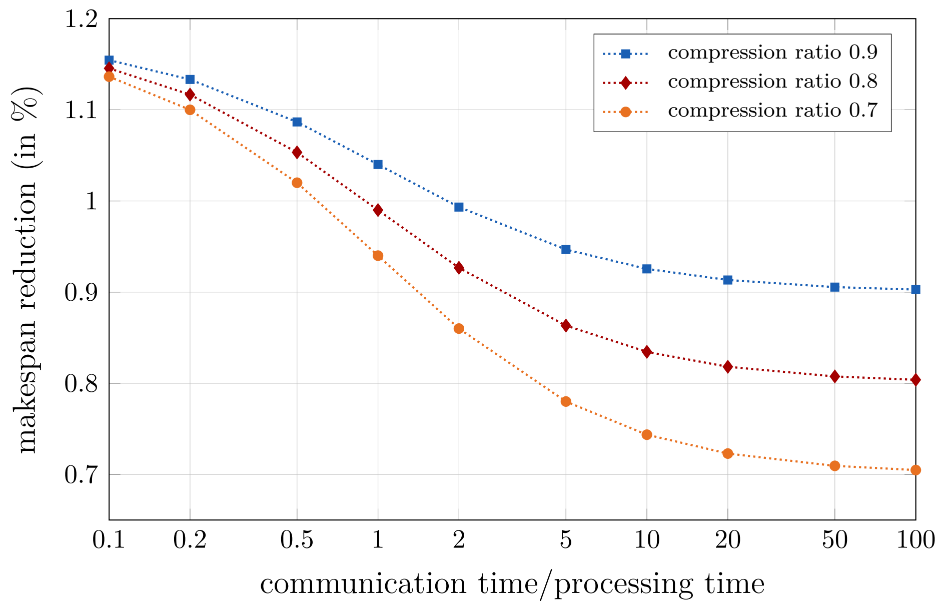


Figure 6.18: Effect of compressed communication on the length of a schedule.

Figure 6.18 shows the potential for reducing the makespan (in %) for different ratios of communication time to processing time in a schedule. Note the logarithmic scaling of the time ratios. According to the above descriptions, the following specific parameters to obtain the curves are assumed to be constant: $n = 10$ tasks, $n - 1 = 9$ messages, $WCET = 90$ ticks per task, $WCCT = WCDT = 9$ ticks (i.e., 10 % of the WCET). In this way, the total time for computations in the schedule is constant. The transmission time per hop is assumed to be 10 ticks. Increasing the number of hops per message changes the time effort towards additional time for communications. The figure shows the outcome for three different compression ratios. Generally, the higher the proportion of communication time in the schedule, the closer the makespan reduction converges towards its maximum value, i.e., the compression ratio. Only if a curve is below one, there is a benefit from compressed messages. For schedules where the proportion of communication time is small, the benefit from compressed messages is negated by the additional times needed for compressing and decompressing the data. The specific results of this plot apparently depend on the choice of the proportion of WCCT (resp., the WCDT) relative to the WCET. Due to the linear relationships in this example, the behavior of the graphs is not affected by the number of tasks.

It has to be pointed out that the logical models of real-world applications are certainly more complex. Scheduling algorithms will make use of task concurrency and decide when messages are compressed and in which cases an uncompressed transmission is advantageous. An estimation of potential makespan reductions then becomes more complex. Nevertheless, the above findings can positively influence system design decisions.

6.5 Impact of Data Compression on Fault Diagnosis

6.5.1 Classification-Based Fault Diagnosis

In fault diagnosis, classification methods such as statistical classification, pattern recognition, and artificial intelligence methods are commonly applied. As described in Section 2.7.4, classification methods aim to distinguish a certain number of faults from each other by evaluating fault *symptoms*. These symptoms refer to a change in an observable quantity from its normal behavior. The symptoms are often obtained from sensor measurements, where additional processing steps are involved. This is called *feature extraction* and refers to the steps necessary to produce meaningful values for the subsequent fault diagnosis, e.g., variances, amplitudes, and frequencies. The use of fault classification (as compared to fault inference) is particularly beneficial when sufficient process data is available to allow a classifier to extract a model of the process, i.e., to learn the behavior of a process (or a system) in all (fault) situations that should be classifiable from labeled data. In the context of machine learning, this is referred to as supervised learning. If the so-called mapping function has been accurately learned, a classification algorithm correctly infers the class labels for unseen data.

Typical performance metrics for classifiers include the classification *accuracy*, which is the ratio of the number of correct predictions to the total number of predictions made. In addition, the F_1 *score* (with $F_1 \in [0, 1]$) as the harmonic mean between the so-called *precision* and *recall* indicates how precise and how robust a classifier is. Precision is the fraction of the number of correct positive predictions and the number of all positive predictions by the classifier:

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Recall is the fraction of correct positive predictions and all relevant (i.e., positive) instances:

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

The F_1 score then is:

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

In general, a higher F_1 score implies a better performance of the classifier. The goal is to find a good balance between precision and recall.

6.5.2 Fault Diagnosis Use Case

6.5.2.1 Hybrid Electric Vehicle Model

In the context of the DAKODIS research project, the HEV model from Section 6.1 served as fault diagnosis use case in several publications, see e.g., [MO18; MOY18]. In the recent work [Mec+20b], an active fault diagnosis strategy based on a convolutional neural network was developed that achieved fault classification accuracies of over 98 %. In particular, the paper analyzed the temporal impact on the diagnostic performance. The results showed that less accurate conclusions about a faulty system behavior can be made immediately after a fault occurs, and that the decision becomes more confident as time progresses and more data is analyzed. This, in combination with the outcomes from Section 6.4, leads to the conclusion that shorter service times of diagnostic tasks based on compressed messages contribute to the improvement of the overall diagnostic performance.

6.5.2.2 Condition Monitoring of a Hydraulic System

To investigate the impact of data compression on fault diagnosis in detail, a more complex use case is needed. The use case for the following evaluations is based on a public dataset from the condition monitoring of a hydraulic system. The dataset was created by the Centre for Mechatronics and Automation Technology (ZeMA), Saarbrücken [HPS15a], and can be downloaded from the University of California Machine Learning Repository [UCI21]. The dataset contains raw process data from 17 sensors of a hydraulic test rig which consists of a primary working and secondary cooling filtration circuit connected via an oil tank [HPS15b]. Measurements (e.g., pressures, temperatures, volume flow rates, power) were obtained from repeated working cycles, each considering different load situations, while the condition of the hydraulic components was intentionally degraded over time, i.e., faults of different severity were introduced. The dataset considers faults on four involved components, i.e., a valve, an internal pump, a hydraulic accumulator, and a cooler. As an example, consider the valve. Its condition may show (1) optimal switching behavior, (2) small lag, (3) severe lag, (4) behavior close to total failure. In general, the task of a classifier with this dataset is to identify faulty system behavior by evaluating the symptoms derived from the measured values.

Overall, the dataset was chosen for several reasons. First, it is data from a real-world process with the purpose of performing fault classification. Second, the type of sensor data is suitable for the presented online data compression algorithms, especially since there is a high correlation between many of the recorded signals. And third, the dataset is predestined for classification-based fault diagnosis because it is possible to extract the classification model purely from the (labeled) data without the need for expert domain knowledge.

6.5.2.3 Classifier Implementation

The open source XGBoost library provides a high-performance implementation of gradient boosted decision trees. In contrast to the classical machine learning technique using single model decision trees, boosting is an iterative approach where multiple machine learning models are combined and trained successively. This means that each new model is trained to correct potential errors that previous models made.

In order to evaluate the impact of compressed data on fault classification, a model is first trained and tested with the original data using XGBoost. Here, using 12 out of the available 17 sensor measurements is sufficient to achieve an accuracy of nearly 100 % for the fault classification (using more features does not improve the results). Each sensor produces 12-bit data values. Assuming equal sampling rates, 144 bits are produced per time step. The measurements directly serve as input features for the training and inference process. In total, there are 2205 labeled instances of time-series data from each sensor. To train and test the classifier, the data was partitioned into sets following the usual split percentage of 80 to 20.

6.5.3 Constrained Communication Resources and Fault Diagnosis

For the following evaluations, a scenario is assumed where the sensor data must be transmitted from the hydraulic test rig to the classifier (which is a task executed on a computation node) via a network. Due to limited communication resources, it is further assumed that bits must be saved for the transmission. This reflects real-world scenarios where, for example, multiple data streams from different tasks compete with each other for communication resources, so that certain applications must be constrained in their communication load. With respect to the above example, a system designer obviously has several options to save data and consequently reduce network traffic in terms of bits, e.g., reducing the sampling rate of sensors, reducing the resolution of the quantizer. For the moment, however, these parameters are assumed to be fixed, since they come from the given dataset.

Another possibility to save bits is to reduce the number of features involved in the classification. Obviously, one would use the so-called predictor importance to omit the features with the least importance for the classification first. Figures 6.19 to 6.21 show the fault classification accuracy for three different components, i.e., the internal pump, the hydraulic accumulator, and the valve. In each plot, the accuracy is abstracted over the different severities of the respective faults. It is generally calculated based on all correct and incorrect classifications, respectively. The annotations at the markers indicate the number of features used for the classification. The complete classification report can be found in Appendix A.2.

For example, the blue line with the square markers in Figure 6.19 shows that the maximum fault classification accuracy is achieved when all 12 features are used. By saving bits in the sense of transmitting fewer features (i.e., fewer sensor signals), the classification accuracy continuously decreases, i.e., there is an increasing number of incorrect classifications of the system condition. This potentially results in dangerous situations where, for instance, an existing fault is not found at all or is incorrectly identified as a different fault. For comparison purposes, the red line with the diamond markers shows how the cache-based data compression algorithm allows to keep the number of transmitted features high (and consequently the fault classification accuracy) because it saves the bits by compressing the individual data values of each data stream. For a fair comparison, for each feature that is omitted in the blue curve, one bit per feature is saved for calculating the results in the red curve. This is done by splitting for each data stream the $n = s + t = 12$ bits per sample into $t = 6$ tail bits (resp., $s = 6$ head bits), and adjusting the dictionary size using $r \in [1, 5]$ so that the desired number of bits per sample is saved. Per definition, $t \geq s$, so saving on average more than 50 % of the bits per sample is only possible if the data values are simultaneously compressed with the presented data compression algorithms. In

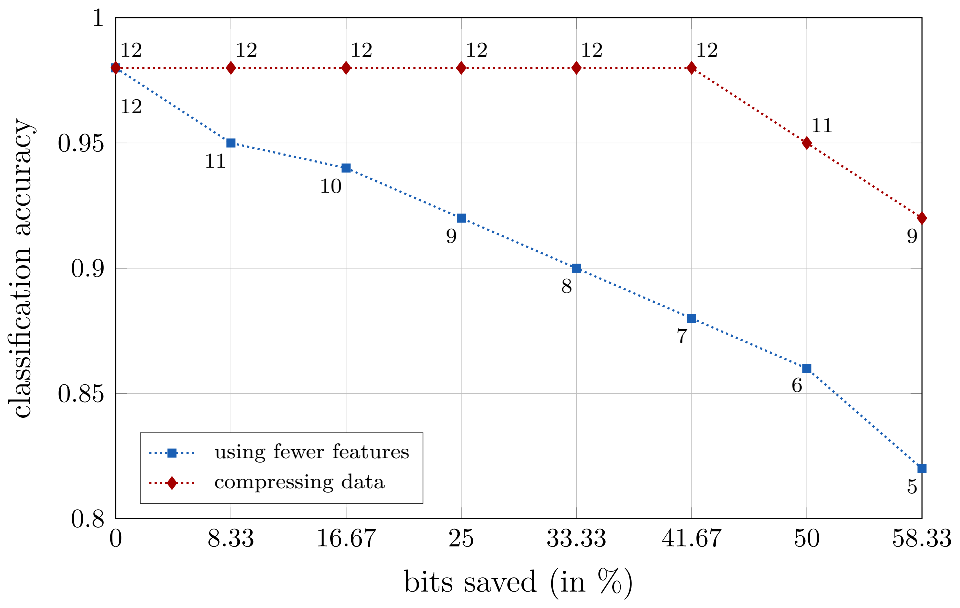


Figure 6.19: Classification accuracy of the internal pump leakage.

the figure, when using only 6 features (i.e., 72 bits, blue curve), an individual data stream compression that should not need more bits, would require to reduce the number of features to 10 (i.e., $10 \cdot 7$ bits, with $r = 1$ and $t = 6$). However, assuming high correlations among several data streams, a simultaneous compression of multiple data streams enables to transmit 11 features, i.e., using $11 \cdot 6$ bits for the tails and the remaining 6 bits for combined dictionaries. The same principle is true for the last comparison point. On the blue curve, 5 features consume 60 bits and on the red curve, the cache-based compression algorithm allows to transmit 9 features using combined dictionaries. Looking at the red line with the diamond markers, there is no obvious decrease in the fault classification accuracy for the first six data points although a certain number of omissions occurred during the compression, transmission, and decompression process. In general, when data is compressed using the cache-based algorithm, the loss rate and tolerable data recovery uncertainty of the respective signals must be considered when selecting the most important features for fault classification. Table A.3 in the appendix presents a complete overview of the loss rates of all sensor signals, compressed over their full lengths using the dynamic cache-based compression algorithm. Figures 6.20 and 6.21 present analogue analyses and refer to the fault classification accuracy of the hydraulic accumulator pressure and the valve condition, respectively.

While Figures 6.19 to 6.21 use the classification accuracy as the performance metric and abstract over the different severities of the respective component faults, Figure 6.22 exemplarily shows the F_1 score for the different valve conditions. In principle, the F_1 score is also a measure of the accuracy of a test, as described in Section 6.5.1. Assuming again

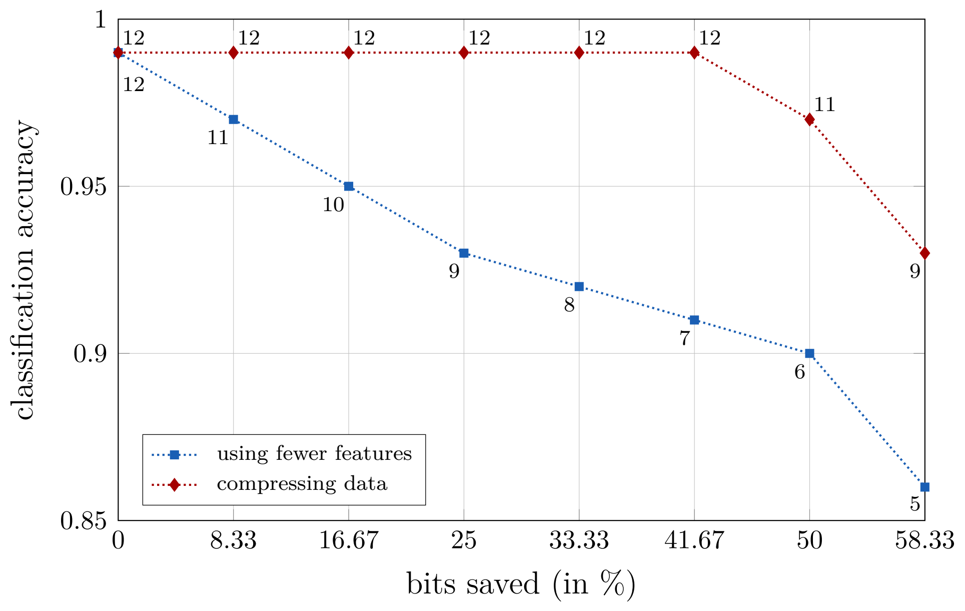


Figure 6.20: Classification accuracy of the hydraulic accumulator pressure.

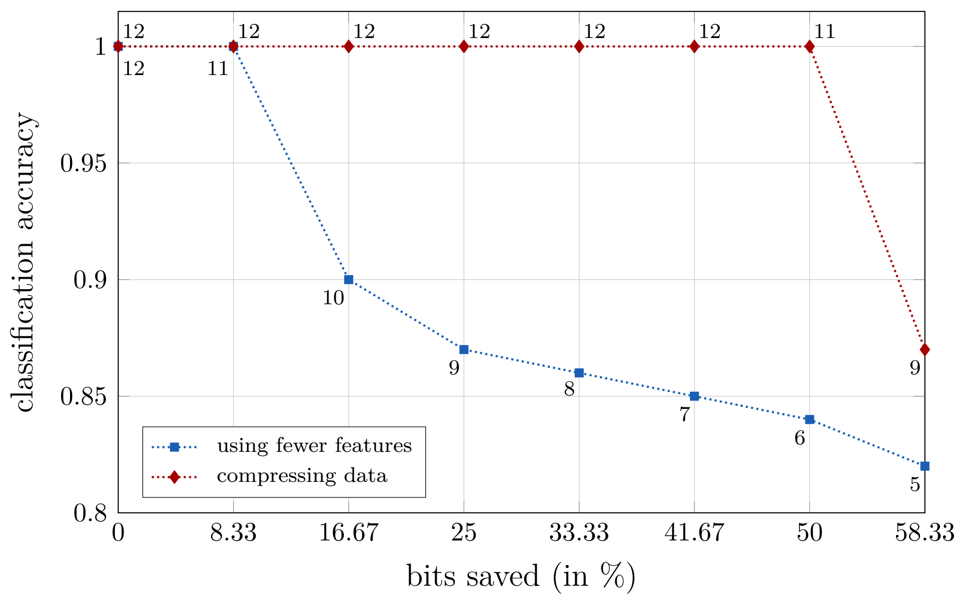


Figure 6.21: Classification accuracy of the valve condition.

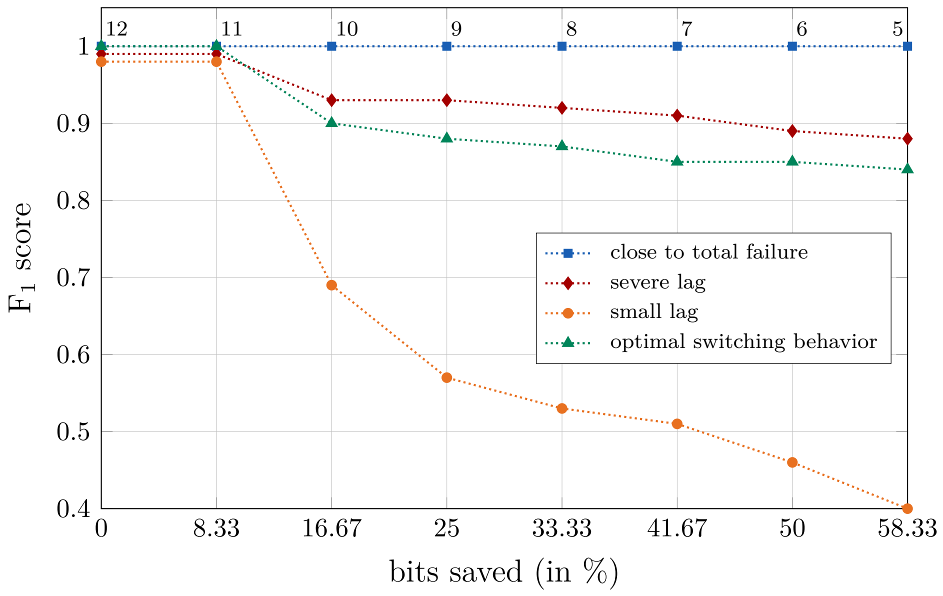


Figure 6.22: F_1 score of the valve condition. Bits are saved by using fewer features for the fault classifications.

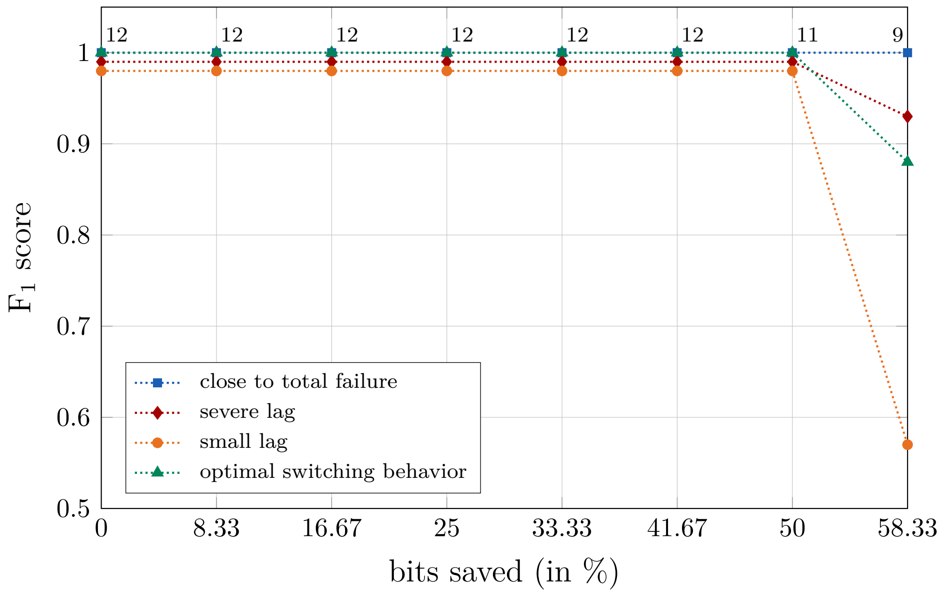


Figure 6.23: F_1 score of the valve condition. Bits are saved by using the cache-based compression algorithm.

the same scenario as above, bits are saved by transmitting fewer features for the fault classification, as indicated by the annotations at the data points. They are written only to the top line and apply to all lines at the respective compression ratios.

The figure shows that a total valve failure can be accurately identified even with few features, but especially the predictive character of the fault diagnosis in terms of classifying small lags decreases drastically. The comparison results are presented in Figure 6.23. With the cache-based data compression algorithm, the equivalent amount of bits is saved directly in the sensor data streams so that the F_1 score remains constantly high.

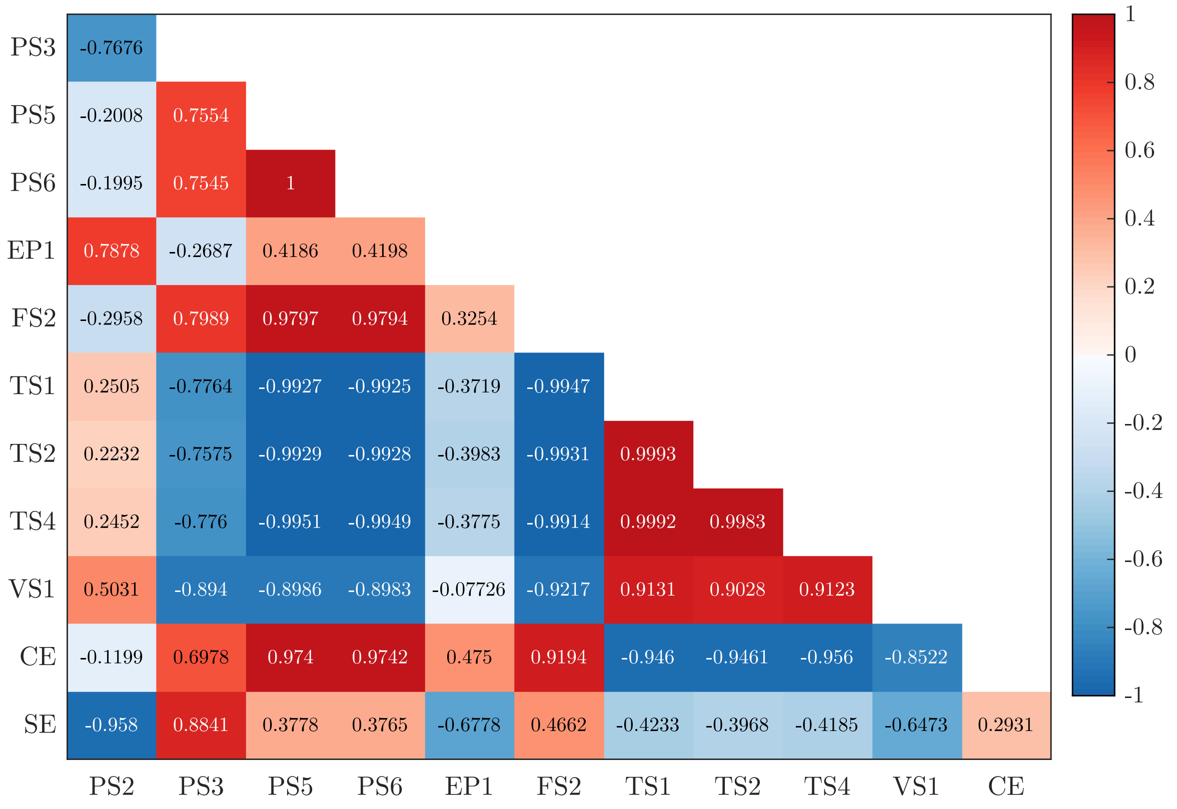


Figure 6.24: Correlation matrix of the sensor signals.

In order to find out which of the sensor signals are suitable to be simultaneously compressed (recall Section 6.2.3.3), Figure 6.24 shows the correlation matrix, i.e., the pairwise linear (Pearson) correlation coefficients of the 12 signals over the 2205 measured instances. The signals are denoted according to the measured quantity, e.g., PS represents a pressure, EP represents electric power, and FS stands for volume flow. As indicated by the colors, many of the signals show either high positive or negative linear correlations, while there are few signals with a low correlation coefficient. As described above, this fact is exploited in the experiment. The simultaneous compression and transmission of multiple data streams resulted in a better utilization of the communication resources and thus improved diagnostic results.

In summary, in resource-constrained systems where data traffic is the limiting factor, there might be a necessity for applications to operate with less data than desired. The above use case demonstrates such a scenario. The advantages of using the cache-based data compression algorithm to reduce the amount of data for communication compared to other reduction possibilities were highlighted by evaluating the fault classification accuracy. In particular, when a significant amount of data needs to be saved (e.g., more than 30 % of the bits), using the cache-based data compression algorithm prevents a significantly decreased fault classification accuracy.

Chapter 7

Conclusion

7.1 Summary and Contribution

This thesis was prepared in the context of the DFG-funded research project DAKODIS. As the main contribution, the thesis presents online data compression algorithms for sensor data streams that go beyond the state of the art and are applicable to time-triggered communication in distributed systems where the communication slots are established with respect to a global time base. Time-triggered systems play an important role in high-dependable and safety-critical systems. Their advantage is a maximized predictability compared to event-triggered systems, due to periodic task executions and messages exchanges according to a static schedule. In many such systems, fault tolerance is efficiently implemented through online fault diagnosis with active recovery strategies, which results in diagnostic tasks competing for resources with regular system service tasks. The need to utilize data compression for communicating data in high-dependable time-triggered systems arises from the ever-growing complexity with increasingly computationally intensive tasks that produce large amounts of data, causing communication resources to become bottlenecks.

First, this thesis defines the so-called DAKODIS architecture, which is a time-triggered architecture including a compression model that enables systemwide coordination of data compression (Chapter 4). The thesis then presents online data compression algorithms that enable short input data sequences to be compressed with a worst-case compression ratio below one while guaranteeing data quality for the applications (Chapter 5). In this way, the specific requirements of both time-triggered communication and the investigated (fault diagnosis) applications could be reconciled. A characteristic feature of the algorithms is the ability to exploit redundancy among multiple sensor data streams and compress them simultaneously in a multidimensional product space, thus achieving improved compression ratios with significantly reduced cases of data degradation. The design of the algorithms based on dynamically maintained dictionaries in combination with the introduced architectural models makes it possible to directly merge and split compressed data streams at arbitrary nodes of a distributed system. This is a prerequisite for a scheduling algorithm to optimize the (compressed) data communications in terms of reducing the lengths of schedules, which subsequently enables tighter deadlines to be met and shorter overall service times of applications to be supported. With a special focus on real-time capability, the novel algorithms compress data online (i.e., one pass) and are able to guarantee short worst-case compression and decompression times. This is realized by implementing the involved dictionaries in the form of a cache with common replacement strategies that allow dictionary operations in constant time. Another feature of the algorithms is that they are fully self-synchronizing concerning the dictionaries on the sender and receiver nodes, i.e., they do not rely on a side channel to exchange overhead information, which is an important issue in time-triggered systems. Also, the data compression can be performed even if no prior statistical knowledge is available. Yet, if it is available, it can be beneficially incorporated and support the formation of suitable transmission regions in the multidimensional product space (Section 5.2.5).

The presented online data compression algorithms are evaluated in Chapter 6 by means of two fault diagnosis use cases. As a typical example for the application of fault diagnosis, the first use case comes from the automotive domain and is based on a hybrid electric vehicle model in Simulink (Section 6.1). The various simulated sensor signals from this model serve as test signals for the evaluation of the online compression algorithms (Section 6.2), where also the advantages and the scalability of a simultaneous compression of multiple correlated data streams are demonstrated from a high-level perspective, i.e., the relationship between the reduction potential of the amount of data and the information loss in terms of loss of accuracy of data values (Section 6.2.3.2). A more complex fault diagnosis use case based on a public dataset of real-world hydraulic system condition monitoring data is used to evaluate the impact of compressed sensor data streams on the diagnostic performance (Section 6.5.2.2). Using a comparative scenario, it is shown that a significant amount of approximately 40 %–50 % of data bits can be saved while still achieving highly accurate diagnostic results. In turn, such saved network traffic can be used to improve the reliability of a system, e.g., by implementing information redundancy or, in other systems, by increasing fault tolerance through online fault diagnosis. The particular advantages of a simultaneous compression of multiple correlated data streams are also confirmed by this use case scenario. In general, the outcomes of the use case analyses underline the feasibility of the novel online data compression algorithms to be deployed in systems where the communication resources are limited (Section 6.5.3). Thereby, the analysis of the influence of compressed messages on the lengths of schedules exemplifies the time reduction potential that depends on the proportion of communication time relative to task computation time in a schedule.

7.2 Future Work

The plans for future work refer to two aspects. On the one hand it is about a direct extension of the achieved results of this thesis and on the other hand it is about ideas for future research projects, which have arisen in the context of this thesis.

The primary goal of this thesis was to establish the ability to cope with limited communication resources while providing temporal guarantees for online diagnosis applications in distributed real-time systems using data compression. The use cases used in the evaluation give an impression of how real systems can benefit from the developed data compression algorithms. However, since the algorithms were developed in the context of fundamental research, they are not yet optimized for real-world applications, but have only been prototypically implemented. Future considerations regarding the beneficial impact of compressed messages on schedules need to take real network protocols into account, e.g., FlexRay and Time-Triggered Ethernet. In addition, more complex application scenarios with task concurrency should be analyzed using advanced scheduling algorithms. An interesting question is also how the compression algorithms need to be modified or extended to manage other types of data. Of course, the measurements of physical quantities considered in thesis are a primary source of information for fault diagnosis, but symbolic data also plays a role.

The second aspect of future work relates to new ideas for future research projects that have emerged in the context of the DAKODIS project. To give two examples: Combining online fault diagnosis with organic computing to realize highly reliable self-organizing systems was first published in [Bri+19] and led to a successfully acquired DFG research project (project number 445555232). In addition, [Mec+19a] presents an approach to improve automotive fault diagnosis with knowledge extracted from web resources. The respective proposal AUTODIREKT was submitted and positively evaluated.

Appendix A

A.1 Loss Rates of Evaluation Test Signals

Table A.1: Loss rates of the individually compressed test signals (HEV dataset) used in Figure 6.13.

	Compression ratio					
	0.5625	0.625	0.6875	0.75	0.8125	0.875
	Loss rate (in %)					
S_1	4.2621	2.0122	0.8029	0.1793	0.0189	0.0022
S_2	4.1027	1.8962	0.7279	0.1483	0.0183	0.0023
S_3	3.9411	1.8123	0.6552	0.1160	0.0144	0.0017
S_4	3.9306	1.7951	0.6385	0.1110	0.0139	0.0017

Table A.2: Loss rates of the individually compressed test signals (HEV dataset) used in Figure 6.15.

	Compression ratio					
	0.5625	0.625	0.6875	0.75	0.8125	0.875
	Loss rate (in %)					
S_1	4.1643	1.9167	0.7507	0.1605	0.0183	0.0022
S_2	3.9306	1.7951	0.6385	0.1110	0.0139	0.0017
S_3	3.3453	1.0222	0.3132	0.0988	0.0122	0.0072
S_4	2.9878	0.9145	0.2760	0.0755	0.0111	0.0067

Table A.3: Loss rates of the individually compressed sensor measurements (condition monitoring dataset) used in Section 6.5.3.

	Compression ratio				
	0.5833	0.6667	0.75	0.8333	0.9167
	Loss rate (in %)				
PS2	0.450	0.233	0.167	0.100	0.033
PS3	0.617	0.083	0.067	0.033	0.017
PS5	0.017	0.017	0.017	0.017	0.017
PS6	0.017	0.017	0.017	0.017	0.017
EP1	0.183	0.033	0.017	0.017	0.017
FS2	0.167	0.167	0.167	0.167	0.167
TS1	1.667	1.667	1.667	1.667	1.667
TS2	1.667	1.667	1.667	1.667	1.667
TS4	1.667	1.667	1.667	1.667	1.667
VS1	3.333	1.667	1.667	1.667	1.667
CE1	1.667	1.667	1.667	1.667	1.667
SE1	21.67	20.00	16.67	8.333	5.000

A.2 Classification Report for the Fault Diagnosis Use Case

Table A.4: Classification report for internal pump faults.
Fault labels: (1) severe leakage, (2) weak leakage, (3) no leakage.

Features	Accuracy	Label	Precision	Recall	F_1-score
12	0.98	1	0.98	0.98	0.98
		2	0.98	0.93	0.95
		3	0.98	1.00	0.99
11	0.95	1	0.93	0.90	0.92
		2	0.94	0.89	0.91
		3	0.96	0.99	0.98
10	0.94	1	0.90	0.90	0.90
		2	0.92	0.85	0.88
		3	0.95	0.98	0.97
9	0.92	1	0.88	0.83	0.85
		2	0.88	0.85	0.87
		3	0.95	0.98	0.97
8	0.90	1	0.83	0.83	0.83
		2	0.90	0.80	0.84
		3	0.93	0.98	0.95
7	0.88	1	0.76	0.88	0.81
		2	0.85	0.72	0.78
		3	0.94	0.95	0.95
6	0.86	1	0.74	0.81	0.77
		2	0.77	0.74	0.75
		3	0.95	0.94	0.94
5	0.82	1	0.72	0.74	0.73
		2	0.71	0.69	0.70
		3	0.90	0.91	0.91

Table A.5: Classification report for hydraulic accumulator faults.
Fault labels: (4) close to total failure, (5) severely reduced pressure, (6) slightly reduced pressure, (7) optimal pressure.

Features	Accuracy	Label	Precision	Recall	F_1-score
12	0.99	4	1.00	0.99	0.99
		5	0.97	0.95	0.96
		6	0.95	1.00	0.97
		7	1.00	1.00	1.00
11	0.97	4	0.97	0.99	0.98
		5	0.95	0.95	0.95
		6	0.95	1.00	0.97
		7	1.00	0.96	0.98
10	0.95	4	0.96	0.97	0.97
		5	0.94	0.87	0.90
		6	0.86	1.00	0.92
		7	1.00	0.94	0.97
9	0.93	4	0.97	0.94	0.95
		5	0.94	0.84	0.89
		6	0.73	1.00	0.85
		7	1.00	0.91	0.95
8	0.92	4	0.97	0.94	0.95
		5	0.94	0.82	0.87
		6	0.75	1.00	0.86
		7	1.00	0.94	0.97
7	0.91	4	0.94	0.94	0.94
		5	0.97	0.82	0.89
		6	0.77	1.00	0.87
		7	1.00	0.93	0.96
6	0.90	4	0.92	0.92	0.92
		5	0.91	0.79	0.85
		6	0.78	1.00	0.88
		7	1.00	0.93	0.96
5	0.86	4	0.89	0.94	0.91
		5	0.75	0.63	0.69
		6	0.73	0.83	0.78
		7	0.95	0.91	0.93

Table A.6: Classification report for valve faults.
Fault labels: (8) close to total failure, (9) severe lag, (10) small lag, (11) optimal switching behavior.

Features	Accuracy	Label	Precision	Recall	F_1-score
12	1.00	8	1.00	1.00	1.00
		9	0.98	1.00	0.99
		10	1.00	0.97	0.98
		11	1.00	1.00	1.00
11	1.00	8	1.00	1.00	1.00
		9	0.98	1.00	0.99
		10	1.00	0.97	0.98
		11	1.00	1.00	1.00
10	0.90	8	1.00	1.00	1.00
		9	0.91	0.95	0.93
		10	0.74	0.65	0.69
		11	0.90	0.91	0.90
9	0.87	8	1.00	1.00	1.00
		9	0.91	0.95	0.93
		10	0.64	0.52	0.57
		11	0.86	0.89	0.88
8	0.86	8	1.00	1.00	1.00
		9	0.92	0.93	0.92
		10	0.58	0.48	0.53
		11	0.84	0.89	0.87
7	0.85	8	1.00	1.00	1.00
		9	0.93	0.90	0.91
		10	0.56	0.47	0.51
		11	0.83	0.87	0.85
6	0.84	8	1.00	1.00	1.00
		9	0.88	0.90	0.89
		10	0.52	0.42	0.46
		11	0.83	0.87	0.85
5	0.82	8	1.00	1.00	1.00
		9	0.88	0.88	0.88
		10	0.46	0.35	0.40
		11	0.81	0.86	0.84

Publications

This thesis uses content which was published in the following publications:

Journal Publications

[Mec+20a] S. Meckel, M. Lohrey, S. Jo, R. Obermaisser, and S. Plasger. “Combined compression of multiple correlated data streams for online-diagnosis systems”. In: *Microprocessors and Microsystems* 77 (2020).

[Mec+20b] S. Meckel, T. Schuessler, P. K. Jaisawal, J.-U. Yang, and R. Obermaisser. “Generation of a diagnosis model for hybrid-electric vehicles using machine learning”. In: *Microprocessors and Microsystems* 75 (2020).

[Jo+18] S. Jo, M. Lohrey, D. Ludwig, S. Meckel, R. Obermaisser, and S. Plasger. “An architecture for online-diagnosis systems supporting compressed communication”. In: *Microprocessors and Microsystems* 61 (2018).

Conference Publications

[Mec+19a] S. Meckel, J. Zenkert, C. Weber, R. Obermaisser, M. Fathi, and R. Sadat. “Optimized automotive fault-diagnosis based on knowledge extraction from web resources”. In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2019.

[Mec+19b] S. Meckel, M. Lohrey, S. Jo, R. Obermaisser, and S. Plasger. “Combined compression of multiple correlated data streams for online-diagnosis systems”. In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2019.

[BMO19] A. Behravan, S. Meckel, and R. Obermaisser. “Generic fault-diagnosis strategy based on diagnostic directed acyclic graphs using domain ontology in automotive applications”. In: *AmE 2019-Automotive meets Electronics; 10th GMM-Symposium*. VDE. 2019.

[Bri+19] U. Brinkschulte, R. Obermaisser, S. Meckel, and M. Pacher. “Online-diagnosis with organic computing based on artificial DNA”. In: *2019 First International Conference on Societal Automation (SA)*. IEEE. 2019.

[MO18] S. Meckel and R. Obermaisser. “Component-based combination of online-diagnosis methods using diagnostic directed acyclic graphs”. In: *2018 7th Mediterranean Conference on Embedded Computing (MECO)*. IEEE. 2018.

[MOY18] S. Meckel, R. Obermaisser, and J.-U. Yang. “Generation of a diagnosis model for hybrid-electric vehicles using machine learning”. In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018.

[Beh+18] A. Behravan, R. Obermaisser, D. H. Basavegowda, and S. Meckel. “Automatic model-based fault detection and diagnosis using diagnostic directed acyclic graph for a demand-controlled ventilation and heating system in Simulink”. In: *2018 Annual IEEE International Systems Conference (SysCon)*. IEEE. 2018.

[Jo+17] S. Jo, M. Lohrey, D. Ludwig, S. Meckel, R. Obermaisser, and S. Plasger. “An architecture for online-diagnosis systems supporting compressed communication”. In: *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE. 2017.

From these publications, the following figures, tables and algorithms are used in a similar or in an adapted version:

© 2018 Elsevier B.V.

Figures 4.5, 5.1, 6.16, 6.17. Table 5.1. Algorithms 1, 2.

© 2020 Elsevier B.V.

Figures 5.2, 5.4, 5.5, 6.1, 6.2, 6.6, 6.7, 6.8, 6.9, 6.10, 6.12. Table 5.4. Algorithms 3, 4.

Bibliography

[ANR74] N. Ahmed, T. Natarajan, and K. R. Rao. “Discrete cosine transform”. In: *IEEE Transactions on Computers* 100.1 (1974).

[Aig+18] M. Aigner, G. M. Ziegler, K. H. Hofmann, and P. Erdős. *Proofs from the book.* 6th ed. Springer, 2018.

[Alf17] S. Alfes. “Modell- und signalbasierte Fehlerdiagnose eines automatisierten Nutzfahrzeuggetriebes für den Off-Board und On-Board Einsatz”. PhD thesis. Technische Universität Darmstadt, 2017.

[ANH16] A. J. Aljohani, S. X. Ng, and L. Hanzo. “Distributed source coding and its applications in relaying-based transmission”. In: *IEEE Access* 4 (2016).

[And00] G. R. Andrews. *Foundations of multithreaded, parallel, and distributed programming.* Vol. 11. Addison-Wesley Reading, 2000.

[Ari+03] T. Arici, B. Gedik, Y. Altunbasak, and L. Liu. “PINCO: a pipelined in-network compression scheme for data collection in wireless sensor networks”. In: *Proceedings of the 12th International Conference on Computer Communications and Networks.* IEEE. 2003.

[Avi+04] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004).

[ALR+01] A. Avižienis, J.-C. Laprie, B. Randell, et al. *Fundamental concepts of dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[Bal+07] R. Baldoni, M. Bertier, M. Raynal, and S. Tucci-Piergiovanni. “Looking for a definition of dynamic distributed systems”. In: *International Conference on Parallel Computing Technologies.* Springer. 2007.

[BW10] E. A. Bender and S. G. Williamson. *Lists, decisions and graphs.* S. Gill Williamson, 2010.

[Ber15] J. Berlińska. “Scheduling for data gathering networks with data compression”. In: *European Journal of Operational Research* 246.3 (2015).

[Blo96] C. Bloom. “LZP: a new data compression algorithm”. In: *Data Compression Conference*. IEEE Computer Society. 1996.

[BD19] A. Burns and R. Davis. “Mixed criticality systems - a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2019). https://www-users.cs.york.ac.uk/burns/review.pdf, [12th ed.; online; accessed 04-Jan-2021].

[BW94] M. Burrows and D. Wheeler. “A block-sorting lossless data compression algorithm”. In: *Digital SRC Research Report*. Citeseer. 1994.

[CK07] J.-J. Chen and C.-F. Kuo. “Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms”. In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. IEEE. 2007.

[CP12] J. Chen and R. J. Patton. *Robust model-based fault diagnosis for dynamic systems*. Vol. 3. Springer Science & Business Media, 2012.

[CA78] L. Chen and A. Avižienis. “N-version programming: a fault-tolerance approach to reliability of software operation”. In: *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8)*. Vol. 1. 1978.

[Cho+08] S. Chowdhury, A. Chowdhury, S. R. Bhadra Chaudhuri, and C. T. Bhunia. “Data transmission using online dynamic dictionary based compression technique of fixed & variable length coding”. In: *2008 International Conference on Computer Science and Information Technology*. IEEE. 2008.

[CW84] J. Cleary and I. Witten. “Data compression using adaptive coding and partial string matching”. In: *IEEE Transactions on Communications* 32.4 (1984).

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. 3rd ed. MIT Press, 2009.

[Cov75] T. Cover. “A proof of the data compression theorem of Slepian and Wolf for ergodic sources”. In: *IEEE Transactions on Information Theory* 21.2 (1975).

[DO03] L. Deng and D. O’Shaughnessy. *Speech processing: a dynamic and optimization-oriented approach*. CRC Press, 2003.

[Des+04] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. “Model-driven data acquisition in sensor networks”. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. Vol. 30. 2004.

[DFG16] DFG Forschungsprojekt DAKODIS. *Data Compression for Active Diagnosis*. http://gepris.dfg.de/gepris/projekt/275601549. 2016–2021, [online; accessed 05-March-2021].

[DG09] P. L. Dragotti and M. Gastpar. *Distributed source coding: theory, algorithms and applications.* Academic Press, 2009.

[Dri+04] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona. “The real byzantine generals”. In: *The 23rd Digital Avionics Systems Conference (DASC).* Vol. 2. IEEE. 2004.

[ES21] Embedded Systems group, University of Siegen. *DFG research project - Data Compression for Active Diagnosis (DAKODIS).* https://networked-embedded.de/es/index.php/dakodis.html. 2021, [online; accessed 05-March-2021].

[Erc19] K. Erciyes. *Distributed real-time systems: theory and practice.* Springer, 2019.

[FG89] E. R. Fiala and D. H. Greene. “Data compression with finite windows”. In: *Communications of the ACM* 32.4 (1989).

[FGT92] P. Flajolet, D. Gardy, and L. Thimonier. “Birthday paradox, coupon collectors, caching algorithms and self-organizing search”. In: *Discrete Applied Mathematics* 39.3 (1992).

[FW74] P. A. Franaszek and T. J. Wagner. “Some distribution-free aspects of paging algorithm performance”. In: *Journal of the ACM (JACM)* 21.1 (1974).

[Gay+04] M. Gayer, M. Lutzky, G. Schuller, U. Krämer, and S. Wabnik. “A guideline to audio codec delay”. In: *Audio Engineering Society Convention 116.* Audio Engineering Society. 2004.

[God12] B. Godfrey. *A primer on distributed computing.* https://billpg.com/bacchae-co-uk/docs/dist.html. 2012, [online; accessed 19-Mar-2020].

[Gon+97] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. “Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling”. In: *Proceedings Real-Time Systems Symposium.* IEEE. 1997.

[Gor19] K. Gorelik. “Energy management system for automated driving”. PhD thesis. Universität Siegen, 2019.

[Gua16] N. Guan. *Techniques for building timing-predictable embedded systems.* Springer, 2016.

[HSR10] J. Han, A. Saxena, and K. Rose. “Towards jointly optimal spatial prediction and adaptive transform in video/image coding”. In: *2010 IEEE International Conference on Acoustics, Speech and Signal Processing.* IEEE. 2010.

[HCS07] L. Hanzo, P. Cherriman, and J. Streit. *Video compression and communications: from basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-style adaptive turbo-transceivers.* John Wiley & Sons, 2007.

[HPS15a] N. Helwig, E. Pignanelli, and A. Schütze. “Condition monitoring of a complex hydraulic system using multivariate statistics”. In: *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings*. IEEE. 2015.

[HPS15b] N. Helwig, E. Pignanelli, and A. Schütze. “Detecting and compensating sensor faults in a hydraulic condition monitoring system”. In: *Proceedings SENSOR 2015* (2015).

[Hen+14] H. Henao et al. “Trends in fault diagnosis for electrical machines: a review of diagnostic techniques”. In: *IEEE Industrial Electronics Magazine* 8.2 (2014).

[Hof14] P. Hofmann. *Hybridfahrzeuge: Ein alternatives Antriebssystem für die Zukunft*. 2nd ed. Springer, 2014.

[Huf+52] D. A. Huffman et al. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952).

[ITU-T00] International Telecommunication Union. *G.711: Pulse code modulation (PCM) of voice frequencies*. Tech. rep. https://www.itu.int/rec/T-REC-G.711. ITU-T, 2000.

[Ise97] R. Isermann. “Supervision, fault-detection and fault-diagnosis methods”. In: *Control Engineering Practice* 5.5 (1997).

[Ise06] R. Isermann. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006.

[Ise11] R. Isermann. *Fault-diagnosis applications: model-based condition monitoring: actuators, drives, machinery, plants, sensors, and fault-tolerant systems*. Springer Science & Business Media, 2011.

[IB97] R. Isermann and P. Ballé. “Trends in the application of model-based fault detection and diagnosis of technical processes”. In: *Control Engineering Practice* 5.5 (1997).

[IM10] R. Isermann and M. Münchhof. *Identification of dynamic systems*. Springer Science & Business Media, 2010.

[Jol02] I. T. Jolliffe. “Principal components in regression analysis”. In: *Principal Component Analysis, 2nd ed.* Springer, New York, NY, 2002.

[Jua94] J.-N. Juang. *Applied system identification*. Prentice Hall, Inc., 1994.

[Koh+16] A. Kohn, R. Schneider, A. Vilela, A. Roger, and U. Dannebaum. *Architectural concepts for fail-operational automotive systems*. Tech. rep. SAE Technical Paper, 2016.

[Kol63] A. N. Kolmogorov. “On tables of random numbers”. In: *Sankhyā: The Indian Journal of Statistics, Series A* (1963).

[Kol+13] J. G. Kolo, L.-M. Ang, S. A. Shanmugam, D. W. G. Lim, and K. P. Seng. “A simple data compression algorithm for wireless sensor networks”. In: *Soft Computing Models in Industrial and Environmental Applications.* Springer, 2013.

[Kol+15] J. G. Kolo, S. A. Shanmugam, D. W. G. Lim, and L.-M. Ang. “Fast and efficient lossless adaptive compression scheme for wireless sensor networks”. In: *Computers & Electrical Engineering* 41 (2015).

[Kol+12] J. G. Kolo, S. A. Shanmugam, D. W. G. Lim, L.-M. Ang, and K. P. Seng. “An adaptive lossless data compression scheme for wireless sensor networks”. In: *Journal of Sensors* 2012 (2012).

[Kop98] H. Kopetz. “The time-triggered architecture”. In: *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’98).* IEEE. 1998.

[Kop11] H. Kopetz. *Real-time systems: design principles for distributed embedded applications.* Springer Science & Business Media, 2011.

[KK07] I. Koren and C. M. Krishna. *Fault-tolerant systems.* Morgan Kaufmann, 2007.

[Kra49] L. G. Kraft. “A device for quantizing, grouping, and coding amplitude-modulated pulses”. PhD thesis. Massachusetts Institute of Technology, 1949.

[KN10] S. Kreft and G. Navarro. “LZ77-like compression with fast random access”. In: *2010 Data Compression Conference.* IEEE. 2010.

[Kri14] C. M. Krishna. “Fault-tolerant scheduling in homogeneous real-time systems”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014).

[Kum+13] A. Kumar, R. Kumaran, S. Paul, and S. Mehta. “A low complex ADPCM image compression technique with higher compression ratio”. In: *International Journal of Computer Engineering and Technology* 4.6 (2013).

[KOK14] A. A. Kumar S., K. Ovsthus, and L. M. Kristensen. “An industrial perspective on wireless sensor networks – a survey of requirements, protocols, and challenges”. In: *IEEE Communications Surveys & Tutorials* 16.3 (2014).

[Lan13] A. Langiu. “On parsing optimality for dictionary-based text compression – the Zip case”. In: *Journal of Discrete Algorithms* 20 (2013).

[Lap92] J.-C. Laprie. “Dependability: basic concepts and terminology”. In: *Dependability: Basic Concepts and Terminology.* Springer, 1992.

[Lei+20] Y. Lei, B. Yang, X. Jiang, F. Jia, N. Li, and A. K. Nandi. “Applications of machine learning to machine fault diagnosis: a review and roadmap”. In: *Mechanical Systems and Signal Processing* 138 (2020).

[LV+08] M. Li, P. Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer, 2008.

[LS19] L. Lo Bello and W. Steiner. “A perspective on IEEE time-sensitive networking for industrial communication and automation systems”. In: *Proceedings of the IEEE* 107.6 (2019).

[LM16] R. V. Lopes and D. Menascé. “A taxonomy of job scheduling on distributed computing systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.12 (2016).

[LO17] D. Ludwig and R. Obermaisser. “Scheduling of datacompression on distributed systems with time- and event-triggered messages”. In: *International Conference on Architecture of Computing Systems*. Springer. 2017.

[LGL18] W. Luo, B. Gu, and G. Lin. “Communication scheduling in data gathering networks of heterogeneous sensors with data compression: algorithms and empirical experiments”. In: *European Journal of Operational Research* 271.2 (2018).

[Luo+18] W. Luo, Y. Xu, B. Gu, W. Tong, R. Goebel, and G. Lin. “Algorithms for communication scheduling in data gathering network with data compression”. In: *Algorithmica* 80.11 (2018).

[Mac05] D. J. C. MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2005.

[Mag15] L. Magnoni. “Modern messaging for distributed sytems”. In: *Journal of Physics: Conference Series*. Vol. 608. 1. IOP Publishing. 2015.

[MO19] S. Majidi and R. Obermaisser. “Genetic Algorithm for Scheduling Time-Triggered Communication Networks with Data Compression”. In: *IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*. 2019.

[MV09] F. Marcelloni and M. Vecchio. “An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks”. In: *The Computer Journal* 52.8 (2009).

[Mar11] P. Marwedel. *Embedded System Design*. 2nd ed. Springer, 2011.

[Meh+18] A. Mehmood, N. Alrajeh, M. Mukherjee, S. Abdullah, and H. Song. “A survey on proactive, active and passive fault diagnosis protocols for WSNs: network operation perspective”. In: *Sensors* 18.6 (2018).

[Men06] G. Menegaz. “Trends in medical image compression”. In: *Current Medical Imaging* 2.2 (2006).

[Mil20] S. Miller. *Hybrid-Electric Vehicle Model in Simulink*. https://github.com/mathworks/Simscape-HEV-Series-Parallel. 2020, [online; accessed 05-March-2021].

[MW85] V. S. Miller and M. N. Wegman. “Variations on a theme by Ziv and Lempel”. In: *Combinatorial Algorithms on Words*. Springer, 1985.

[Moo+65] G. E. Moore et al. “Cramming more components onto integrated circuits”. In: *Electronics*. Vol. 38. 1965.

[Moo+75] G. E. Moore et al. “Progress in digital integrated electronics”. In: *Electron Devices Meeting*. Vol. 21. 1975.

[IEC10] *Norm IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission (IEC). 2010.

[ISO11] *Norm ISO 26262: Road vehicles – functional safety*. International Organization for Standardization (ISO). 2011.

[Obe11] R. Obermaisser. *Time-triggered communication*. CRC Press, 2011.

[Omd88] T. P. Omdahl. *Reliability, availability, and maintainability (RAM) dictionary*. American Society for Quality Control, 1988.

[Pet+03] D. Petrovic, R. C. Shah, K. Ramchandran, and J. Rabaey. “Data funneling: routing with aggregation and compression for wireless sensor networks”. In: *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*. IEEE. 2003.

[Pop+07] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. “Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems”. In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. 2007.

[Pro+02] J. G. Proakis, M. Salehi, N. Zhou, and X. Li. *Communication systems engineering*. 2nd ed. Vol. 2. Prentice Hall, Inc., 2002.

[PŠ18] I. Punčochář and J. Škach. “A survey of active fault diagnosis methods”. In: *IFAC-PapersOnLine* 51.24 (2018).

[RY14] K. R. Rao and P. Yip. *Discrete cosine transform: algorithms, advantages, applications*. Academic Press, 2014.

[RBD13] M. A. Razzaque, C. Bleakley, and S. Dobson. “Compression in wireless sensor networks: a survey and comparative evaluation”. In: *ACM Transactions on Sensor Networks (TOSN)* 10.1 (2013).

[RP06] M. Roitzsch and M. Pohlack. “Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video”. In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS’06)*. IEEE. 2006.

[RL09] W. Ryan and S. Lin. *Channel codes: classical and modern.* Cambridge University Press, 2009.

[SM06] C. M. Sadler and M. Martonosi. “Data compression algorithms for energy-constrained devices in delay tolerant networks”. In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. 2006.

[SM10] D. Salomon and G. Motta. *Handbook of data compression.* 5th ed. Springer Science & Business Media, 2010.

[Say17] K. Sayood. *Introduction to data compression.* 5th ed. Morgan Kaufmann, 2017.

[Sch05] P. Scholz. *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung.* Springer-Verlag Berlin Heidelberg, 2005.

[Sha48] C. E. Shannon. “A mathematical theory of communication”. In: *Bell System Technical Journal* 27.3 (1948).

[SN09] *Siemens Norm 29500: Ausfallraten Bauelemente.* Siemens AG. 2009.

[SW73] D. Slepian and J. Wolf. “Noiseless coding of correlated information sources”. In: *IEEE Transactions on Information Theory* 19.4 (1973).

[SM78] P. T. de Sousa and F. P. Mathur. “Sift-out modular redundancy”. In: *IEEE Transactions on Computers* 100.7 (1978).

[Spa+17] H. Sparka, R. Naumann, S. Dietzel, and B. Scheuermann. “Effective lossless compression of sensor information in manufacturing industry”. In: *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*. IEEE. 2017.

[SKR14] S. Sridhar, P. R. Kumar, and K. V. Ramanaiah. “Wavelet transform techniques for image compression – an evaluation”. In: *International Journal of Image, Graphics and Signal Processing* 6.2 (2014).

[Sri+12] T. Srisooksai, K. Keamarungsi, P. Lamsrichan, and K. Araki. “Practical data compression in wireless sensor networks: a survey”. In: *Journal of Network and Computer Applications* 35.1 (2012).

[Sto88] J. A. Storer. *Data compression: methods and theory.* Computer Science Press, Inc., 1988.

[SS82] J. A. Storer and T. G. Szymanski. "Data compression via textual substitution". In: *Journal of the ACM (JACM)* 29.4 (1982).

[Sto96] N. R. Storey. *Safety critical computer systems.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[SWH94] N. Suri, C. J. Walter, and M. M. Hugue. *Advances in ultra-dependable distributed systems.* IEEE Computer Society Press, 1994.

[Tab20] N. Tabassam. "Minimizing the makespan of diagnostic multi-query graphs in embedded real time systems". PhD thesis. Universität Siegen, 2020.

[Tek12] Tektronix. *Guide to MPEG fundamentals and protocol analysis.* https://download.tek.com/document/25W-11418-10.pdf, [online; accessed 09-Mar-2021]. 2012.

[Uni13] United Nations Economic Commission for Europe - World Forum for Harmonization of Vehicle Regulations. *Proposal for a new global technical regulation on the Worldwide harmonized Light vehicles Test Procedure (WLTP).* https://www.unece.org/fileadmin/DAM/trans/doc/2014/wp29/ECE-TRANS-WP29-2014-027e.pdf. 2013, [online; accessed 04-May-2021].

[UVD18] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan. "A survey on data compression techniques: from the perspective of data quality, coding schemes, data type and applications". In: *Journal of King Saud University - Computer and Information Sciences* (2018).

[Vac06] G. J. Vachtsevanos. *Intelligent fault diagnosis and prognosis for engineering systems.* Vol. 456. Wiley Hoboken, 2006.

[VV17] J.-M. Valin and K. Vos. *Updates to the Opus audio codec.* RFC 8251. https://rfc-editor.org/rfc/rfc8251.txt. RFC Editor, 2017.

[VVT12] J.-M. Valin, K. Vos, and T. Terriberry. *Definition of the Opus audio codec.* RFC 6716. https://rfc-editor.org/rfc/rfc6716.txt. RFC Editor, 2012.

[VT02] M. Van Steen and A. Tanenbaum. "Distributed systems principles and paradigms". In: *Network* 2 (2002).

[VGM14] M. Vecchio, R. Giaffreda, and F. Marcelloni. "Adaptive lossless entropy compressors for tiny IoT devices". In: *IEEE Transactions on Wireless Communications* 13.2 (2014).

[Vit87] J. S. Vitter. "Design and analysis of dynamic Huffman codes". In: *Journal of the ACM (JACM)* 34.4 (1987).

[VAA04] M. C. Vuran, Ö. B. Akan, and I. F. Akyildiz. "Spatio-temporal correlation: theory and applications for wireless sensor networks". In: *Computer Networks* 45.3 (2004).

[WW11] R. Wang and J. Wang. “Fault-tolerant control with active fault diagnosis for four-wheel independently driven electric ground vehicles”. In: *IEEE Transactions on Vehicular Technology* 60.9 (2011).

[Wel84] T. A. Welch. “A technique for high-performance data compression”. In: *Computer* 6 (1984).

[Wil91] R. N. Williams. “An extremely fast Ziv-Lempel data compression algorithm”. In: *1991 Data Compression Conference*. IEEE Computer Society. 1991.

[YK15] S. Yoshida and T. Kida. “An efficient variable-to-fixed length encoding using multiplexed parse trees”. In: *Journal of Discrete Algorithms* 32 (2015).

[UCI21] ZeMA gGmbH. *Condition monitoring of hydraulic systems data set*. https://archive.ics.uci.edu/ml/datasets/Condition+monitoring+of+hydraulic+systems. 2021, [online; accessed 14-April-2021].

[ZL77] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977).

[ZL78] J. Ziv and A. Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Transactions on Information Theory* 24.5 (1978).