An Architecture for Online-Diagnosis Systems supporting Compressed Communication

Seungbum Jo[a], Markus Lohrey[a], Damian Ludwig[a], Simon Meckel[a], Roman Obermaisser[a], Simon Plasger[a]

[a]*Department of Electrical Engineering and Computer Science, University of Siegen, Germany*

Abstract

With its ability to detect, identify and, if applicable, recover from occurred faults, online-diagnosis can help achieving fault-tolerant systems. A sound decision on an occurred fault is the foundation for fault-specific recovery actions. For this, typically a large amount of data has to be analyzed and evaluated. A diagnostic process implemented on a distributed system needs to communicate all those data among the network which is an expensive affair in terms of communication resources and time. In this paper we present an architecture for a distributed online-diagnosis system with real time constraints that supports data compression to reduce the communication time. We further present a library of lossy compression methods with guaranteed compression ratios that are suitable for real time purposes.

Keywords: Online-Diagnosis, Real-Time, Fault-Tolerance, Data Compression, Scheduling

1. Introduction

In safety-critical application domains such as avionics, health care or industrial automation, systems must provide their services with a high reliability. Even in the case of a fault in a hardware or software component or due to external influences, an acceptable quality of the services must be ensured. Typical faults are faults in the design of a component or a system, transient or permanent hardware faults or erroneous user operations amongst others. Applying fault-diagnosis methods to the system its reliability and stability can be significantly increased. In the widely-used *offline* fault-diagnosis process information, sensor data or fault indications exclusively get stored for later analysis. However, utilizing *online* fault-diagnosis with the ability to detect a fault, identify its location and potential effects at run time of the system, in combination with interactions with the system, e.g., reconfigurations, a fault-tolerant system can be formed [1].

Enabling the diagnostic algorithms to make correct and fast decisions on fault detection and identification, often a huge amount of raw data of a variety of sensors as well as system input and output parameters have to be processed, analyzed and stored during the diagnostic process. Considering that these tasks are computationally intensive in addition with the fact that the diagnostic process allows a concurrent processing of different tasks, parallelism can be exploited. By executing tasks on separate processing units within a distributed network, resources are planned meaningfully. According to the diagnostic process the calculated data must be consequently made available to the other units as messages via the network.

In [2], an inference-based system for active diagnosis in distributed embedded systems is presented, where diagnostic information is inferred using SPARQL[1] queries and stored in a distributed real-time database. The needed real-time guarantees are supported by a time-triggered schedule which controls the execution of diagnostic tasks and the replication of the database.

The quality of diagnostic decisions highly depends on the available amount of data. A limited storage capacity of the local real-time databases may therefore narrow the performance of diagnostic decisions, e.g., of long term trend analyses. Likewise, the time that is needed to conclude a specific fault from a first symptom can be seen as quality characteristic of the diagnostic architecture. It is the goal to minimize this time as much as possible. Since the diagnostic process requires both, the storage and the fast exchange of a huge amount of data within the distributed network, the DAKODIS[2] project deploys data compression for online-fault-diagnosis. Data compression (see [3] for a general introduction into this wide area) is a feasible instrument which can help to save bandwidth and consequently provide stronger real-time guarantees or save communication resources. Additionally, it might save storage which can be used to store more diagnostic information in the real-time databases to improve the quality of the fault-diagnosis in terms of a more precise and extensive fault detection and identification. Especially systems with limited resources may require data compression to actually make fault-diagnosis possible.

To support compression without loss of real-time guarantees, one needs to obtain strict upper bounds for the compression ratio (the quotient of the compressed data size and uncompressed data size) and the computation time needed for compression and decompression. These bounds can then be used to consider compression for resource planning. Unfortunately, traditional lossless compression techniques [3] cannot achieve a compression ratio strictly below one for every input, i.e., there must always exist an input which is not compressed. Moreover, offline data compression is often used in existing compression algorithms. In contrast we are aiming for a small sample-wise worst-case compression ratio for real-time algorithms to be applicable in online-diagnosis systems.

Let us briefly explain the idea of our main compression algorithm. The goal is to compress a sequence of n-bit data values (for a fixed n). Typically, these data values are produced by physical sensors and exhibit some locality. For compression, we split every n-bit data value into a block of s high-order bits (called the head) and the remaining $t = n - s$ low-order bits (called the tail). Due to locality of the data values, one can expect that the heads of the data values only show a small variation over time. The $n = s + t$ bits in a data value are compressed to $r + t$ bits (for some $r < s$). For this, we transmit the t bits from the tail uncompressed and compress the s bits from the head to r bits. To achieve the latter, we store in a dictionary the $2^r - 1$ most recently seen heads and assign an r-bit code different from the reserved sequence 0^r (r many 0-bits) to each of them. If the head from the next data value is in the dictionary then we transmit its r-bit code from the dictionary. Otherwise, a so called miss occurs which is indicated to the receiver by sending the bit sequence 0^r followed by the s bits from the new head (to fit this code into $r + t$ bits, we have to assume

[1] https://www.w3.org/TR/sparql11-query/

[2] DAKODIS (German acronym) - Data compression for online diagnosis systems. The primary objective of the research project is an increased efficiency and the reduction of overhead for online diagnosis in open embedded systems using data compression. For further information see: https://networked-embedded.de/es/index.php/dakodis.html

that $s \leq t$). Moreover, the new head is added to the dictionary. Due to locality in the data values we can expect a small number of misses over time. We also present several variations of the above basic compression algorithm.

The main features of our compression algorithm are:

- A fixed compression ratio of $(r + t)/(s + t) < 1$ is achieved for every data value. This is contrary to classical lossless compression, where only statements about the average compression ratio are possible.
- To make a fixed compression ratio < 1 possible, we have to accept occasional losses of data values. A small number of lost data values is acceptable for many online diagnosis applications.
- Those data values that are not lost are transmitted without any loss in accuracy, which is in contrast to classical lossy compression.

The idea of using locality in the data values for compression can be found in many works. One of the simplest ways of exploiting locality is delta-coding, where differences between consecutive data values are transmitted. These differences are typically small and can be further compressed with an entropy encoder. In the context of wireless sensor networks this idea is implemented in the LEC-compressor from [5]. In lossy compression, differential encoding [3, Chapter 11] exploits correlation between successive data values by transmitting the differences between a prediction of the next data value and the actual data value.

This paper is an extended version of our DSD 2017 conference paper [6]. We present an architecture for a distributed online-diagnosis system with real-time constraints that supports compression for communication and is controlled by a static time-triggered schedule. The new contributions include multiple improved compression schemes, interchangeable within our architecture to allow an optimal adaptation to different application scenarios. As limited communication resources are often the bottleneck in distributed online-diagnosis systems, the new online compression schemes allow a better utilization of the given resources.

2. Model

The DAKODIS architecture consists of three parts: the logical model describing diagnostic tasks, the physical model describing a network and the compression model, describing a set of different compression algorithms and schemes. These models are needed to define a fourth model for scheduling diagnostic tasks to the network using data compression for communication.

2.1. Logical Model

The prerequisite for fault-diagnosis, i.e., the detection and identification of faults, is knowledge about the process or the system in order to distinguish a faulty process behavior from a healthy one. Often, knowledge about the system is available from different sources. If a mathematical model of the system is available, observer based fault classifiers can be applied to evaluate the consistency between the measured outputs of the practical systems and the

model-predicted outputs. See [7] for an overview on model-based fault-diagnosis techniques or e.g., [8] for advanced observer techniques. In signal-based fault analysis the knowledge about the process is typically available in the form of a signal model (signal relations), process parameters (e.g., limits of a sensor signal) and corresponding digital signal processing techniques, see e.g., [9]. The system behavior can also be extracted from a large amount of historic data by utilizing digital signal and data processing techniques such as machine learning, amongst others, [10], [11]. A knowledge base combines the information about the healthy process behavior and information about fault indications.

Modern (mechatronic) systems achieve their functionalities through an interaction of multiple components. The process of fault-diagnosis from a detection of a first fault indication up to a distinguished identification of a fault type and its location requires to extract, analyze, interpret and merge a multitude of diagnostic information captured at the various components of the system. The diagnostic information is extracted in different manners. Analytical knowledge about the process, e.g., the signal model and corresponding processing techniques such as correlation or trend analyses, are utilized to extract characteristic information (features) from the measured signals which provides evidence of the current process state. Faults in the process are reflected in these signals and in the diagnostic features, accordingly. In a complex system, several potential faults (e.g., of different components) may lead to the same faulty signal behavior of a monitored signal. In such a scenario, the fault detection is followed by a process of evaluating confirmative and unsupportive diagnostic information, respectively, in order to include or exclude particular faults, eventually specifying the actually occurred fault. The diagnosis of different faults requires the execution of different signal processing and feature extraction tasks in a defined order.

The dependencies between the diagnostic operations are modeled in a diagnostic directed acyclic graph (DDAG), denoted as $G = (T, E, (\ell_e)_{e \in E})$, with T being a set of tasks t and E being a set of ordered pairs (t, t') modeling a precedence relation between two tasks. Edges from E are also called *logical channels* or just *channels*. For a channel e, the number $\ell_e \in \mathbb{N}$ specifies the bit length of the data values sent via channel e. Thus, over a channel e a stream of data values is sent and every data value is encoded using ℓ_e bits.

Task t' depends directly on t, iff $(t, t') \in E$. A task may depend on multiple others and in turn may work as a prerequisite for other tasks. In a directed acyclic graph we can clearly identify all predecessors and successors of any task.

The DDAG combines the knowledge about the process and potential faults, i.e., how far faults effect a change of measured signals. Furthermore it describes the signal processing techniques (at the nodes) to extract the diagnostic features. With the edges showing the necessary information to be exchanged, the fault inference process is modeled. In order to allow the node operations (tasks) to be scheduled to the processing units of a distributed network, the DDAG design underlies certain rules which are abstracted in the following:

- A diagnostic task produces characteristic information (no intermediate results).
- A task may comprise multiple processing steps.
- A task uses as little as possible input information.

- For the order of fault reasoning (seeking confirmative or unsupportive information for a specific fault) several factors (e.g., probability of a fault's occurrence, computational cost of the diagnostic operation required) are taken into account.
- Fault reasoning and intermediate conclusions are allowed in every subgraph.

In the DAKODIS architecture the processing steps are encapsulated into tasks which are executed at the processing units of a distributed network. According to its objective, a task utilizes modularized algorithms of the fields preprocessing of sensor or process data, diagnostic information extraction, further processing (e.g., combination) of diagnostic information and data compression.

All necessary data is made available to the dedicated nodes via messages through the network. As the diagnostic process is performed at run time on a distributed system the inference on faults is decomposed in the temporal and spatial domain. In case of a fault, the attained fault indications initially provide evidence for a certain fault but may yet hold inconclusive information at a certain node of the DDAG (e.g., confirmative information from other nodes is not yet available). The degree of confidence of a correct fault identification increases with more information merged.

2.2. Physical Model

Our network model consists of a set C of computation nodes and a set R of routers, where $C \cap R = \emptyset$. The graph representing the network is defined as $\text{Net} = (V, L)$, with $V = C \cup R$ and $L \subseteq (R \times R) \cup (C \times R) \cup (R \times C)$ being an undirected edge relation. The definition of L assures that a computation node can be directly connected to one or more routers but not to any other computation node. Only computation nodes can execute tasks and only routers are able to forward messages.

Figure 1 shows the architecture of the proposed system. Every computation node has access to a locally stored part of the distributed real time database that can be queried with SPARQL queries. This database holds all information needed to execute the diagnostic tasks located on this node and also stores the data produced by these tasks. The synchronization of the database among all nodes of the network is done by a dedicated synchronization unit available on each node. This unit also purges expired data from the storage as needed. As the communication between the nodes should use compression, the synchronization unit should compress outgoing messages, if the overhead is reasonable compared to the utility. For incoming compressed messages the unit needs to perform the decompression and write the newly arrived data to the database. Compute nodes may buffer outgoing messages. In contrast, routers are not able to do this. Instead, they must forward incoming messages directly.

The tasks running on a node do not have direct access to the database. Instead, they read from a task-exclusive input buffer and write their results to an also task-exclusive output buffer. These buffers are managed by a local database manager, which is aware of every tasks' needs. As the diagnostic tasks can operate on sliding time windows, the manager will only replace data which is not related to the current window, leaving all other data untouched. This

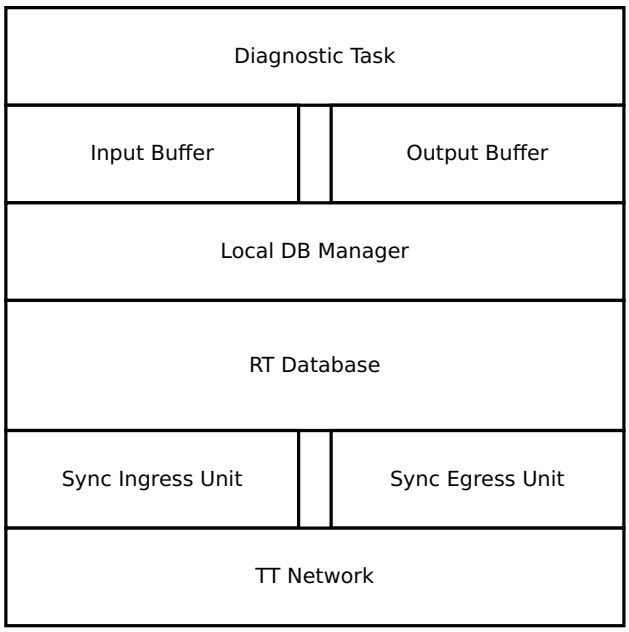

Figure 1: Physical model architecture

reduces read-operations on the database and write-operations on the input buffers. The buffer size and the SPARQL queries needed to collect the data from the database are design-time parameters of the diagnostic task.

To put everything together, the local database manager, the synchronization unit and the execution of diagnostic tasks are controlled by a time triggered schedule. For that purpose the scheduler knows about each tasks' fireing rate and the worst-case execution times of all involved processes. The schedule is calculated at design-time and does not change during runtime. It considers resource allocation, path- and slot-selection and compression, while trying to keep the makespan minimal.

2.3. Compression Model

The DAKODIS architecture is designed to work with different compression schemes. In order to provide real time guarantees the following parameters have to be known:

- the worst case compression time (WCCT) for a data value,
- the worst case decompression time (WCDT) for a compressed data value,
- and the worst case compression ratio (WCCR).

Formally, a compression scheme is a tuple $Z = (\ell, k, ct, dt)$, where $\ell \in \mathbb{N}$ is the bit length of an input data value, $k \in \mathbb{N}$ is the maximal bit length of a compressed data value, $ct \in \mathbb{N}$ is the WCCT, and $dt \in \mathbb{N}$ is the WCDT. Thus, 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 ct, and the time needed to recover the original ℓ-bit string from the output is at most dt. A compression scheme is applicable to a channel as defined in Section 2.1, if the bit length ℓ_e of the data values transmitted via channel e is ℓ. Note that the WCCR is k/ℓ. To profit from compression, we want to have $k < \ell$. This is clearly not possible using lossless compression. In Section 3 we propose a lossy compression algorithm with $k < \ell$. The algorithm fails to transmit data values with a small probability, but those data values that are transmitted can be perfectly recovered at the receiver.

2.4. Scheduling

Calculating a schedule is an important task when designing a real time embedded system. Basically, the schedule provides information about when and where a task can be executed without violating resource restrictions or dependency relations. For DAKODIS we choose a non-preemptive, static scheduling model similar to *partitioned scheduling*. As discussed in [12], a non-preemptive scheduling model has advantages on multi-core systems or distributed systems, as the overhead for migrating a task is more difficult to predict in case of preemptive scheduling. The disadvantage of non-preemptive tasks reducing the responsiveness of a system is not valid for multi-core systems, since the natural parallelism of such a system can hide this latency [12].

We need to consider task allocation, as the distance of two tasks in the network has an impact on the time needed for communication, thus on the makespan. An allocation function is a mapping $A : T \to C$ which maps tasks to computation nodes. We require that for every channel $e = (t, t') \in E$ and $A(t) \neq A(t')$ there exists a path of the form $A(t), r_1, \ldots, r_n, A(t')$ in the network Net, where $n \geq 1, r_1, \ldots, r_n \in R, (A(t), r_1) \in L, (r_i, r_{i+1}) \in L$ 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 consideration we fix such a path and denote it with $A(e)$.

Every path $A(e)$ has the following properties relevant for scheduling:

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

Obviously, a path $A(e)$ can intersect another path $A(e')$ in one or more computation nodes or routers. In order to avoid conflicts, one has to take a look at the ports being used at the shared resources.

In Figure 2 the communication between computation nodes c_0 and c_1 does not conflict with the communication between c_2 and c_3, because the paths use different sets of ports. Even if two data values arrive at the same time, no collision occurs. In contrast, Figure 3 shows conflicting communication, since both paths share the port that the router uses to forward data to c_1. To avoid conflicts and collisions of data values, the use of these ports has to be scheduled, too. This can either be managed by delaying values at the sender to make them arrive later at the shared resource

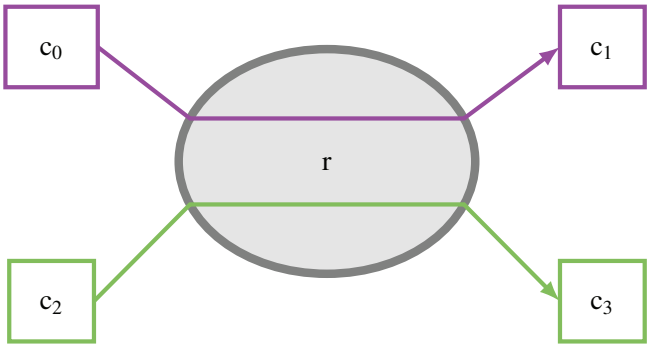

Figure 2: Non-conflicting communication

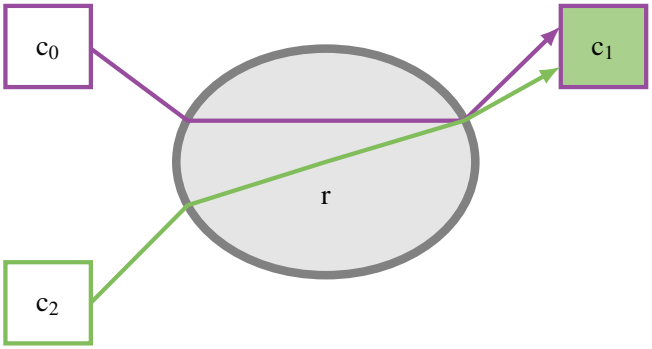

Figure 3: Conflicting communication

(temporal separation), or by trying to choose a conflict free path (spatial separation). Data compression eases the scheduling of conflicting paths, as it reduces the unavailability of involved resources.

The DAKODIS architecture adds the utilization of data compression to the classical scheduling problems, like resource allocation and routing. As known from Section 2.3 a compression scheme provides real time guarantees in terms of WCCT and WCDT as well as a guarantee on the WCCR. If the compression scheme $Z = (\ell, k, ct, dt)$ 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 ct and dt have to be added to the execution times of the channel's end points. Thus, the usage of data compression is always a trade-off between less communication and longer execution times.

A simpler version of this scheduling model with only one compression algorithm and equal overheads for compression and decompression has been formulated for MILP solvers in [13]. This model also includes a stricter policy for using the routers, so that even communication as shown in Figure 2 will be treated as a potential conflict and needs temporal separation.

3. Compression

Our compression method has to cope with different types of data values: Initially, data values are obtained by physical sensors that produce data samples of real numbers (or tuples of real numbers). We map these real-valued

data samples to a finite range of N values that we can identify with the numbers $0, \ldots, N-1$. For this we use standard quantization techniques, see e.g., [3]. We can encode every number from the range $0, \ldots, N-1$ with $\ell := \lceil \log_2 N \rceil$ bits. During the diagnostic process, the quantized values are transformed into symbolic data values.

Our specific requirements for the DAKODIS architecture imply the following features for the compression algorithm:

- Compression and decompression underlie hard real time constraints.
- Only online (one-pass) compression algorithms can be used that compress every arriving data value before the next data value arrives.
- No statistical information concerning the probability distribution of the data values is available.
- In order to utilize compression for the scheduling process, the compressor should guarantee a certain worst-case compression ratio below one.
- Compression should be lossless (after the initial quantization phase), with the exception that occasionally data values can be completely lost. This means that every data value is either transformed by the sender into a compressed representation that can be exactly recovered by the receiver, or it is transformed into a default value that tells the receiver that the original data value is lost. Loosing some data values is unavoidable if we want to guarantee a worst-case compression ratio below one. A small probability for loosing data values is tolerable in our context, since diagnosis is typically not dependent on single data values.

The above requirements rule out most of the classical compressors (see [3] for an overview on classical compression techniques):

- Lossy compressors based on transform coding (e.g., cosine transforms, wavelet transforms) are inherently lossy and do not allow to recover data values without error (which is a problem for symbolic data values that were produced during the diagnostic process). Moreover, these compressors typically exploit the limitations of human perception (e.g., for the compression of pictures, audio or video data), which is not relevant in our context.
- Lossless compressors (e.g., entropy-based coders like Huffman coding or arithmetic coding and the dictionary-based compressors from the Lempel-Ziv family) cannot guarantee a fixed compression ratio. A lossless compressor cannot properly compress every single input data value and compressing larger blocks of values would torpedo the real time constraints.

For our scenario we assume the data communication to be error free, i.e., a data sample is received exactly as it was sent. Under challenging conditions, forward error correction is a feasible instrument to ensure this.

3.1. Cache-Based Compression

In order to meet the above mentioned requirements for the compression algorithm, we developed a dictionary-based compression algorithm, where the dictionary is implemented as a cache with common replacement strategies. We therefore speak of a cache-based compression algorithm in the following. In order to ensure a small rate of lost data values, we have to assume some locality in the data values: If the current data value is $i \in [0, N-1]$ then with high probability the next data value should belong to a small neighborhood of i. This is a reasonable assumption if the data values $0, 1, \ldots, N-1$ represent neighboring quantization levels of real valued sensor data.

For the compression algorithm, we assume that the number N of data values is of the form $N = 2^{s+t}$ 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 we denote by $\mathrm{bin}_\ell(i)$ in the following. We call such a bit string a *code word* in the following. The first s bits (resp., last t bits) of a code word are its *head* (resp., *tail*). Heads, i.e., bits strings of length s are 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 100101110101 are 1001 and 01110101, respectively. The head $u = 1001$ corresponds to the number $2^0 + 2^3 = 9$, 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. We call each set $\{uv \mid v \in \{0,1\}^t\}$ a *block* or, more precisely, the block corresponding to the head u. Referring to the aforementioned example ($s = 4, t = 8$) we have $2^{s+t} = 4096$ data values, covered by 16 blocks each having 256 entries. Due to locality consecutive data values are likely to belong to the same block.

We also fix a number $r \leq s$ and construct a dictionary with at most $2^r - 1$ entries such that each entry stores a head, and every head is stored in at most one entry. Every dictionary entry is addressed with a bit string $u' \in \{0,1\}^r$ with $u' \neq 0^r$ (there are $2^r - 1$ such bit strings), which we call a *compressed head*. Initially, the dictionary is either empty or filled with some heads that are known to occur frequently in the data stream. The heads that belong to the dictionary are called the *active heads*, and the blocks corresponding to the active heads are the *active blocks*. Both, sender and receiver will store the same dictionary at every time instant. The parameters $N, s, t, \ell = s + t$ and $r \leq s$ will be fixed for the further considerations.

Input data is now compressed as follows. Consider an input value $i \in [0, N-1]$ and let $w = uv$ with $|u| = s$ and $|v| = t$ be the corresponding code word, i.e., the ℓ-bit binary expansion of i. If u is an active head (we can check this in constant time by implementing the dictionary using a hash table) and stored at position u' in the dictionary (where $u' \neq 0^r$ is a bit string of length r as described above), then we write the bit string $u'v$ of length $r + t$ on the communication channel. In this way, we save $s - r$ many bits. Otherwise, if u is not active, then we write the bit string $0^r u$ of length $r + s \leq r + t$ on the communication channel. Moreover, the sender inserts the new head u into the dictionary. If the dictionary is not yet fully filled (i.e., contains less than $2^r - 1$ heads) we assign a free entry to the new head u. If the dictionary is already full, we replace one of the old heads in the dictionary by u. For this we use the least-recently-used (LRU) strategy. Note that the code word $w = uv$ is lost in this situation and we call this event

Algorithm 1 Cache-based algorithm

```
Input : data value i ∈ [0, N − 1].
Output : bit string of length at most r + t.
Initialize the dictionary D as an empty hash table of size 2^r − 1.
Let uv = bin_ℓ(i) with |u| = s and |v| = t.
if u is stored in D[u'] then  // u' is an r-bit string.
    return(u'v)
else
    if D has a free entry then
        Insert u to D
    else
        Replace an old entry from D by u based
        on the LRU strategy.
    end if
    return(0^r u)
end if
```

a *miss*. If the receiver reads $0^r u$, the prefix 0^r indicates that the next s many bits represent a new head u. The receiver inserts u into its dictionary using the same strategy as the sender. Thus, the compression ratio of this algorithm is at most $(r + t)/(s + t)$. Note that we define the compression ratio as the quotient of the length of the transmitted bit sequence divided by the code word length $(s + t)$. Hence, a smaller compression ratio means better compression. Algorithm 1 describes the pseudo-code of the cache-based algorithm. Figure 4 graphically demonstrates the procedure for two code words. The dictionary has three entries and some three heads from the overall range of heads are active. The first code word 100101110101 can be successfully compressed as its head 1001 is active. Since the head 1011 of the second code word is not active (miss case) the head gets communicated as the tail of the compressed code word starting with the reserved compressed head 00. A more detailed example for the cache-based algorithm can be found in Section 3.4.

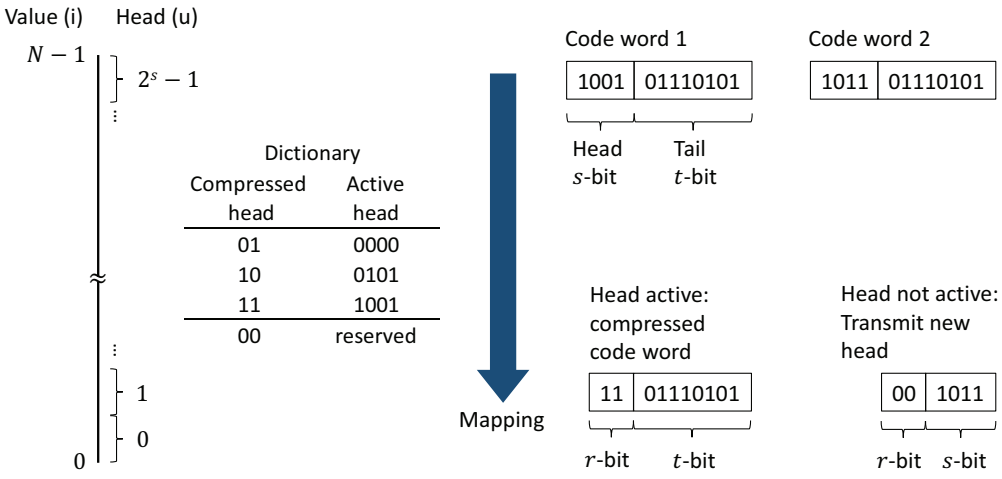

Figure 4: Cache-based algorithm procedure; example with $s = 4$, $t = 8$, $r = 2$

3.2. Analysis of the Probability of a Miss

Let us now consider the probability of a miss. For this we model the sequence of code words $w_1 w_2 w_3 \cdots$ as a stochastic process. Recall that we have $N = 2^{s+t}$ different code words. Under the (not always realistic) assumption that successive code words are identically and independently distributed (iid), we have a so called iid process, which is described by a single probability distribution $(P[w] \in [0,1])_{w\in\{0,1\}^{s+t}}$ on the set of code words, where $P[w]$ is the probability that code word w appears. Whether a certain code word leads to a miss only depends on the head of the code word. From the probabilities $P[w]$ we can compute the probability p_u that a certain head $u \in \{0,1\}^s$ appears as

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

For a uniform distribution (i.e., $p_u = p_{u'}$ for all heads u, u') the probability of a miss is $1 - (2^r - 1)/2^s$. Flajolet et al. [14] show that given the head probabilities p_u one can in principle compute the miss probability with the following formula, where $k = 2^r - 1$ is the size of the dictionary:

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

where $P_J = \sum_{v\in J} p_v$. As also noted in [14] this formula is not suitable for practical calculations of the probability of a miss.

Franaszek and Wagner [15] 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 the optimal replacement strategy. The latter stores the $k - 1$ heads with the highest probabilities in the dictionary. The remaining dictionary entry is used for replacement. Note that the optimal strategy assumes knowledge of the above probabilities p_u. The result from [15] (with our parameters) states that

$$F_{\text{lru}}/F_{\text{opt}} \le 1 + \frac{(2^r - 1)(1 - \beta)}{1 + (2^r - 2)\beta},$$

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

In practical situations, the next data value is highly dependent on the previous values. In particular locality, which is typically observed in physical processes, implies that the next data value is with a high probability in a small neighborhood of the previous data value. In such a setting, our cache based compressor will show a much smaller probability of a miss than in the above iid setting. This will be demonstrated by our experimental data in Section 4.3.

3.3. Reducing Uncertainty and Miss Rate

Our requirements for the DAKODIS architecture only allow occasional misses. Our goal is to keep these events as rare as possible, yet preserving as much information as possible about the missed code words.

In this section we present two improvements of our basic algorithm. These improvements can be done if $s < t$ (recall that we assume $s \leq t$). This is a reasonable assumption: making t too small means that the block size 2^t is small, but smaller blocks will lead to a higher miss rate.

3.3.1. Reducing the Uncertainty

In Algorithm 1, we communicate the head $u = b_1 \cdots b_s$ in the event of a miss, 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 the receiver. The remaining t bits (the tail) of the current code word are lost. Thus, the receiver knows the block to which the current data value belongs to. In other words: The potential range of the correct code word has size 2^t (the block size).

We can further reduce this potential range (i.e., the uncertainty of the correct value): In the miss case, the bit sequence $0^r u$ of length $r + s$ is transmitted to the receiver (see line 14 of Algorithm 1). Recall that a bit string of length $r + t > r + s$ is transmitted if there is no miss, which results in the worst-case compression ratio of $(r + t)/(s + t)$. The DAKODIS architecture is time triggered. As each task is executed according to a fixed schedule, there is no benefit from potentially shorter messages in the event of a miss. Instead, we can reduce the uncertainty of the correct value in this case by transmitting the bit string $0^r b_1 \cdots b_t$, consisting of the head $u = b_1 \cdots b_s$ and the $(t - s)$ most significant bits $b_{s+1} \cdots b_t$ of the tail, without increasing the worst-case compression ratio. This reduces the potential range of the correct data value to 2^s. The compression ratio then becomes $(r + t)/(s + t)$ in every step.

3.3.2. Reducing the Miss Rate

We can also use the $t - s$ unused bits in the miss case (line 14 of Algorithm 1) in order to reduce the miss rate in the following way. Recall that in our 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 (the entries of the dictionary). But, the bit sequence 0^r could still fulfill its purpose if we include it in the set of compressed heads (which then has 2^r elements) but reserve the first 2^s code words in the active block that is assigned to 0^r for the transmission of heads in the event of a miss. This leads to the following extension of our algorithm: Sender and receiver reserve one of the 2^r many dictionary entries (i.e., compressed heads) for the miss case; let us call this entry u'_0. Initially, $u'_0 = 0^r$. A miss occurs if (i) the head u of the current code word $w = uv$ does not belong to the dictionary or (ii) the head u is stored in the dictionary entry u'_0 (i.e., $D[u'_0] = u$) and 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, we transmit $u'_0 0^{t-s} u$. From the fact that the transmitted bit sequence starts with $u'_0 0^{t-s}$, the receiver can conclude that a miss occurs. Moreover, u'_0 is set to the dictionary entry that stores the least-recently-used block from the dictionary.

Depending on the application, the priority may be either a smaller uncertainty of the correct value in the miss case, or a reduced miss rate. As a compromise between the two improvements of the basic 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 values) from the active block that is currently used to indicate a miss.

3.4. Example for the Cache-Based Algorithm

Table 1: Codebook for the cache-based algorithm at time instant 4

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

For the following example of the cache-based algorithm we assume to get samples from an analog-to-digital converter (ADC) with a resolution of 12 bits, meaning that one sample 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, we obtain $2^4 = 16$ blocks with $2^8 = 256$ code words per block. In order to save 2 bits for the transmission we set $r = 2$, giving us $2^r - 1 = 3$ compressed heads: 01, 10 and 11. We assume that 3 code words have been already transmitted, resulting in the dictionary mapping shown in Table 1. We present the example with the improvement from Section 3.3.1. Now we go through our algorithm with some code words, beginning at time step 4:

- Time step 4: The first code word is 010100001100. Its head is 0101 and its tails is 00001100. Thus, the head is active: $D[10] = 0101$. So, we send 1000001100.
- Time step 5: The code word is 000011011001. As $D[01] = 0000$, we send 0111011001.
- Time step 6: The code word is 101000111110 with the head 1010. This head is currently not active, so we have 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., we set $D[11] = 1010$. Since $t - s = 4$ we send the four most significant bits 0011 of the tail 00111110 to improve the precision of the missed sample. The transmitted bit sequence is 0010100011: 00 to indicate the miss, 1010 for the new active head, and 0011 for the four MSBs of the tail.
- Time step 7: The code word is 101000111110, i.e., the same as the previous one. This time the head 1010 is active and stored at dictionary entry 11. Thus, we send 1100111110.
- Time step 8: The final code word is 100110101011. Its head 1001 is no longer active (it was removed from the dictionary at time step 6), so we have another miss. This time, the least recently used head is $D[10] = 0101$. We send 0010011010 and set $D[10] = 1001$.

3.5. Dynamic Cache-Based Algorithm

It is a characteristic of our compression algorithm (Section 3.1) that an update of the dictionary is only performed after a miss occurred. Analyses of several signals show that consecutive values often rise or fall in one direction in

Algorithm 2 Dynamic cache-based algorithm

$\delta : \{0,1\}^s \to [-2^{t-1}, 2^{t-1}]$: the offset mapping,
miss : variable for indicating the event of a miss.
Input : data value $i \in [0, N-1]$.
Output : a bit string of length at most $r + t$.
Initialize dictionary D as an empty hash table of size $2^r - 1$.
For every $x \in \{0,1\}^s$, set $\delta(x) := 0$
miss := 1
Let $uv = \text{bin}_\ell(i)$ with $|u| = s$ and $|v| = t$.
for $x \in \{u-1, u, u+1\}$ stored in D **do**
    if $i \in [x \cdot 2^t + \delta(x), (x+1) \cdot 2^t + \delta(x) - 1]$ **then**
        Let $xz = \text{bin}_\ell(i - \delta(x))$ with $|z| = t$
        Let u' such that $D[u'] = x$ // $|u'| = r$
        miss := 0
        $\delta(x) := i - c(x)$.
        if $\delta(x) < -2^{t-1}$ **then**
            $\delta(x) := 0$ // reset $\delta(x)$ if it is out of range.
            InsertHead $(x-1)$ // add neighboring head
        else if $\delta(x) > 2^{t-1}$ **then**
            $\delta(x) := 0$
            InsertHead $(x+1)$
        end if
        return$(u'z)$
        break
    end if
end for
if miss **then**
    InsertHead (u)
    return$(0^r u)$.
end if
function InsertHead(x)
    $\delta(x) := 0$
    if x is not in D **then**
        if D has a free entry **then**
            Insert x to D
        else
            Replace an old entry from D by x,
            based on the LRU strategy.
        end if
    end if
end function

the short term (e.g., samples from a measurement of a physical quantity such as a voltage or current measurement sampled at 100 Hz). This inevitably leads to a miss once a value is not covered by an active block, meaning that the according head u of the required code word $w = uv$ is not in the dictionary. Utilizing this knowledge a dynamic version of the online compression algorithm overcomes this matter by unfixing the static relation between a value i (a quantization level) and the corresponding code word w.

Initially, the mapping from data values (i.e., natural numbers from the interval $[0, N-1]$) to code words (i.e., bit strings of length $\ell = s + t$) is the standard binary expansion. The code words are then split into heads and tails. In this way, each head x (viewed as a natural number from $[0, 2^s - 1]$) refers to the fixed block $[x \cdot 2^t, (x+1) \cdot 2^t - 1]$ of data values. For each of these intervals we define the *default center* $c(x) = x \cdot 2^t + 2^{t-1}$. These values are not modified by the algorithm. Additionally, we introduce an *offset* parameter $\delta(x) \in [-2^{t-1}, 2^{t-1}]$ for each head (initially 0), which is modified by the algorithm. It influences the mapping between data values and code words. The offset $\delta(x)$ should be understood relative to the default center $c(x)$.

Consider the example with $s = 4$, $t = 8$ and the head $u = 0000$. If $\delta(u) = 0$, then the data values in $[0 + \delta(u), 255 + \delta(u)] = [0, 255]$ are encoded by the first $2^t = 256$ code words (with head $u = 0000$). If the offset of the head u becomes $\delta(u) = +1$, then the data values from $[1, 256]$ are encoded by the code words with head u.

The strategy of the dynamic cache-based algorithm is as follows (line numbers refer to Algorithm 2): As in the original algorithm (Section 3.1), we use a dictionary D to store the heads of some active blocks. A successful value transmission is possible if the dictionary contains a head x, whose corresponding block covers the current data value $i \in [0, N-1]$. The latter means that $i \in [x \cdot 2^t + \delta(x), (x+1) \cdot 2^t + \delta(x) - 1]$. Due to the limitation of the offset parameters to the interval $[-2^{t-1}, 2^{t-1}]$, x must be one of the three heads $u - 1$, u, and $u + 1$, where u is the head of $\mathrm{bin}_\ell(i)$. If we find an x with the above property, and x is stored at dictionary entry $u' \in \{0,1\}^r$ (lines 9, 10, 12) then we transmit the bit sequence $u'z$ (line 22), where z is the tail of $\mathrm{bin}_\ell(i - \delta(x))$ (line 11). The offset $\delta(x)$ then becomes the difference between the transmitted value and the default center of the used active block, i.e., $\delta(x) := i - c(x)$ (line 14). Hence, the new interval $[x \cdot 2^t + \delta(x), (x+1) \cdot 2^t + \delta(x) - 1] = [i - 2^{t-1}, i + 2^{t-t} - 1]$ is centered around i. If the distribution of the data values exhibits some locality, the next data value will belong to this interval with high probability. If after the update of $\delta(x)$, its value no longer belongs to $[-2^{t-1}, 2^{t-1}]$, we reset $\delta(x)$ to 0. The neighboring head where i occurs without an offset is additionally made active (in case $r > 1$), using the replacement strategy (lines 15–21).

A miss occurs if the data value is currently not covered by at least one code word in the dictionary. In this case we transmit 0^r followed by the default head u (i.e., the head of $\mathrm{bin}_\ell(i)$), reset the offset of u to zero and, in case u does not belong to the dictionary D, add u to D according to our replacement strategy (lines 27–28).

3.6. Example of the Dynamic Cache-Based Algorithm

Suppose we want to compress 12-bit code words, which are composed of a 4-bit head and an 8-bit tail, 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, we use the bit string 00 to indicate a miss. Each head can have an offset from $[-2^{8-1}, 2^{8-1}] = [-128, 128]$. Now

consider the case that the current dictionary is empty and the sender transmits the three data values 384, 434, 534 using the dynamic cache-based algorithm. The sender compresses these values as follows:

1. *Transmit* 384: We have $\text{bin}_{12}(384) = 0001\,10000000$ and the head of this string is 0001. There is no active head in the current dictionary. Therefore, the data value is lost and sender transmits 00 0001 to the receiver, and inserts 0001 with offset 0 to the dictionary. Let $D[01] = 0001$.
2. *Transmit* 434: We have $\text{bin}_{12}(434) = 0001\,10110010$. The head of this string is 0001, which has currently offset 0 and is stored at dictionary entry 01. Therefore, the sender transmits the compressed string 01 10110010 to the receiver, and changes the offset of 0001 to $434 - c(0001) = 434 - 2^8 - 2^7 = 50$.
3. *Transmit* 534: We have $\text{bin}_{12}(534) = 0010\,00010110$ and the head of this string is 0010, which is not active in the current dictionary. In the original cache-based algorithm, the data value is lost in this case, but in the dynamic cache-based algorithm, a successful data transmission is possible: The offset of the currently active head 0001 is 50 and 534 belongs to the interval $[2^8+50, 2\cdot 2^8+49] = [306, 561]$. Hence, we get a success in line 9 with $x = 0001$. Since $\text{bin}_{12}(534-50) = 0001\,11100100$ and $D[01] = 0001$, the sender transmits 01 11100100. Moreover, the offset of 0001 is changed to $534 - c(0001) = 534 - 2^8 - 2^7 = 150$. Since $150 > 128$, we reset $\delta(0001)$ to 0 (line 18 and 19) and add the neighboring head 0010 with offset 0 to the dictionary by setting $D[10] = 0010$.

3.7. Difference Coding Algorithm

For the DAKODIS architecture we require online compression algorithms that guarantee a low message delay. Furthermore, the time triggered architecture relies on a worst-case compression ratio. A fixed compression ratio optimally combines a compression benefit with the architecture requirements. The cache-based algorithm in its basic and extended version fulfills both of these prerequisites at the cost of sporadic misses of samples. To handle these rare events the algorithms possess a robust strategy, which allows to even reconstruct a signal to a greatest extent in extremely volatile situations, e.g., high signal fluctuations due to faults in the system.

In the following we introduce a complementary approach that utilizes data compression based on difference coding of values. It makes use of the fact that differences between consecutive values are typically small (and can be communicated with less bits), if locality of the data values can be assumed. We refer to the same notation as used for the cache-based algorithm, however, in the following $s < r < s + t = \ell$ for some $s, t, r \geq 1$. For every data value $i \in [0, N-1]$ there is a code word $w = uv$ with $|u| = s$ and $|v| = t$. The parameters s and t define the number and the size of the blocks. A data value can be expressed via the difference to its preceded value.

For example, we want to communicate a value i_k ($k \geq 0$) from the sender to the 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 the calculation of the difference. The receiver can reconstruct 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: We define a static dictionary D with 2^r entries from which we use 2^s entries to store all heads u. We use the remaining $2^r - 2^s$ entries 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. With this strategy not all potential differences of two consecutive values are mapped to code words. If the current difference d_k is stored in $D[d']$, then the code word d' is transmitted and the receiver reconstructs the value successfully from the received difference and the previously reconstructed value. If d_k is not stored in the dictionary we have a miss. In this case we compute the head u of $\text{bin}_\ell(i_k)$ and send the unique $u' \in \{0, 1\}^r$ with $D[u'] = u$ (every head u is stored in the dictionary). Due to 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, namely the center value $u \cdot 2^t + 2^{t-1}$ of the corresponding block $[u \cdot 2^t, (u + 1) \cdot 2^t - 1]$. The compression ratio of the difference coding algorithm is $r/(s + t)$ in every step.

Inevitably, the first value of a data transmission is a miss, as there is no reference value available at the receiver. The initial transmission is always the code word w' corresponding to the head u of the first data value $\text{bin}_\ell i$.

Large block sizes 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 block, the differences of subsequent values to this center value may be constantly out of the interval $[-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$. For the case $s = 1$, we have two heads to indicate a miss and we use them to inform the receiver whether 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 computed as 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 that many consecutive misses occur.

3.8. Biased Difference Coding

With the difference coding algorithm presented in Section 3.7 only differences from the limited range $[-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$ can be successfully transmitted, allowing an error free value reconstruction. Introducing an additional *bias* based on some previous values of the data stream, the difference values to be transmitted can often be reduced, thereby minimizing the probability of a miss. From the evaluations of multiple signals we know, that consecutive values of physical measurements often rise or fall in the same direction (see Section 2).

Consider the current value i_k and the two previous 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 normal difference coding algorithm, if $d_k \in [-(2^r - 2^s)/2, (2^r - 2^s)/2 - 1]$ we transmit the corresponding code word, or else, we have a miss. In the biased difference coding we compute $d'_k = d_k - d_{k-1} = i_k - 2i_{k-1} + i_{k-2}$. In many cases we have $|d'_k| < |d_k|$; this is for instance the case for a linearly increasing signal. Hence, transmitting the values d'_k (if possible) will reduce the miss rate in these cases. The receiver can reconstruct the value i_k (in the none-miss case) from the previous reconstructed values and d'_k. The miss case is handled analogous to the normal difference coding algorithm.

4. Examples and Evaluation

Recent developments in the automotive industry yielded a large number of driver assistance systems. Electronic stability control, assistance for parking and distance keeping, and especially autonomously driving vehicles (prototypes) require a multitude of different sensor information to be gathered, processed and evaluated. These systems demand a high reliability and with an increasing number of complex tasks taken over from the driver, safety-related aspects of the driver assistance systems become central concerns. In this context, online-diagnosis helps to detect, identify and manage faults occurred in the system to prevent damage, stay operational or increase the maintainability, amongst others.

4.1. Distributed Architecture of Modern Cars

Modern cars have many electronic control units, most of which have to deal with specific tasks, e.g., sensor data processing or providing control signals for other devices. However, these electronic control units may adopt diagnostic tasks as well which are related to their predestinated task. The overall electronic architecture of modern cars, can thus be seen as a distributed network where several processing units can execute diagnostic tasks leading to a distributed diagnostic process.

We introduce a Simulink model to exemplarily demonstrate the working principle of the diagnostic process on a distributed network by means of a hybrid-electric vehicle (HEV) model[3]. However, the aspects can be generalized for many other systems or processes.

The HEV-model offers an abstraction of a hybrid-electric car and allows to simulate the car's behavior according to a driving cycle input. The main components are the electrical part including an electric motor, a generator, a voltage converter and a battery. The electric motor is connected to the driveshaft. Besides, a power split device combines an internal combustion engine (ICE) with the generator and the driveshaft. Via the power split device the ICE fulfills two tasks: supporting the motor to drive the car and extending the car's operating range by charging the battery via the generator. A mode logic manages the interaction between these units (e.g., turning on or off the generator depending on the current battery state of charge and driving situation). The model is equipped with a variety of sensors, such as voltage sensors, current sensors, torque sensors or tachometers. The temperature of different components as well as the mode logic is also monitored. Since faults in the system (e.g., failure of a component) are reflected in the measured signals, a diagnostic decision can be conducted based on the analysis and processing of fault indications derived from the sensor signals.

4.2. Working Principles of a Distributed Fault-Diagnosis Process

In the following, we demonstrate a fault detection and diagnostic inference process by means of a DDAG. The DDAG in Figure 5 models the diagnostic dependencies for the faults specified in this example. It is not complete for

[3]Hybrid-Electric Vehicle Model in Simulink,
http://www.mathworks.com/matlabcentral/fileexchange/28441-hybrid-electric-vehicle-model-in-simulink

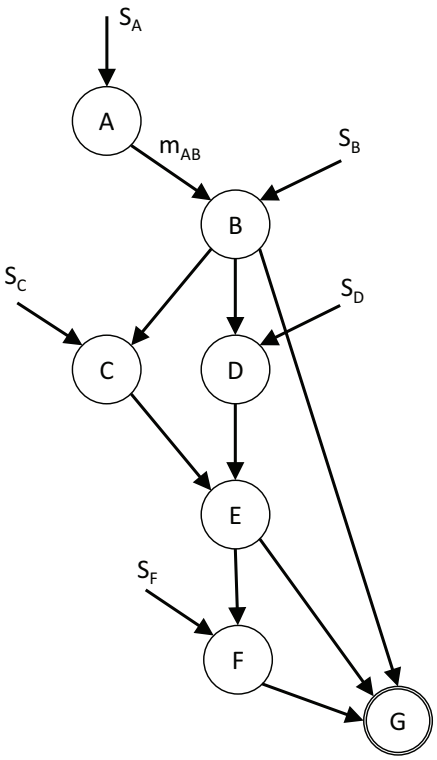

Figure 5: Diagnostic directed acyclic graph for the example model

the whole HEV-model yet suitable for demonstration purposes. Table 2 gives an overview of the diagnostic tasks and necessary input signals.

In the simplified scenario we assume that only one fault occurs at a time. Faults are limited to the following four events: wrong mode logic signal, failure of the generator, failure of the internal combustion engine or an electric line defect. Since all of these components are involved into the battery charging process, a failure of one of the components leads to an abnormal signal behavior of the battery measurements. Applying limit, trend, and plausibility observations on the monitored state of charge (SOC) signal S_A, the first fault indication is determined at node A (Figure 5), specifically, battery not charged. At this stage a fault is detected, however, it is not identified as it may have arisen from different component failures. During the fault-inference process more sensor data is evaluated to isolate the fault. The diagnostic information obtained at node A is communicated via the message m_{AB} to node B. Examining the signaling of the mode logic device (S_B), node B is able to reason about a fault in the mode logic. A faulty signaling immediately leads to a final fault decision bringing us to node G. In our simplified example, a deeper analysis for this type of fault is not performed, however, the DDAG could be extended to allow more accurate conclusions. A correct signaling excludes this potential fault and requires further analyses of the components generator and engine at the nodes C and D, respectively. Evaluating torque, rotational speed and electric measurements of these components (S_C and S_D) and taking energy conservation into account, their status is determined and the diagnostic features are

Table 2: Diagnostic tasks and signals

Node	**Task**
A	Evaluation of battery state of charge
B	Verification of mode signaling
C	Evaluation of generator functioning
D	Evaluation of ICE functioning
E	Intermediate fault decision
F	Electric circuit evaluation
G	Final fault decision

Signal	**Measurements**
S_A	Battery: state of charge
S_B	Mode logic: signaling
S_C	Generator: torque, generator speed, voltage, current
S_D	ICE: torque demand, torque, engine speed
S_F	Electrical system: voltages, currents

combined at node E, allowing a diagnostic intermediate decision based on plausibility relations. If a correct working of the generator and the ICE is stated, we proceed to node F. With S_F being voltage and current measurements in different locations of the electric circuit, the fault can be identified.

A stepwise fault-inference process matches real world systems. The successive generation, evaluation, and combination of the information for the fault-diagnosis process are often required. Especially when a system to be diagnosed consists of multiple independent processing units, not all relevant data may be available for the diagnosis at all times. Furthermore, the diagnostic feature extraction consumes computational resources. Likewise, the broadcasting of the information through the network impacts the overall transmission performance. In these cases, one may concentrate on monitoring fewer important signals continuously and ask for confirmative data after the fault detection step. The ability to adopt the diagnostic methods to a variety of applications may help to increase their reliability, availability, and especially the safety. When the compression model is additionally taken into account, online fault-diagnosis can be favorably established, in particular on systems with limited (communication or data storage) resources. By means of the scheduler, a (near) optimal distribution of diagnostic tasks to the given physical infrastructure is ensured, and thus the minimized makespan leads to fast diagnostic results.

4.3. Evaluation of the Compression Algorithms

To examine the performance of the cache-based algorithm (Section 3.1), the dynamic cache-based algorithm (Section 3.5), and the difference coding algorithm (Section 3.7), we implemented a simulation[4] that works on data sets from the HEV-model.

[4]Implementation of the cache-based code: https://networked-embedded.de/es/index.php/dakodis.html

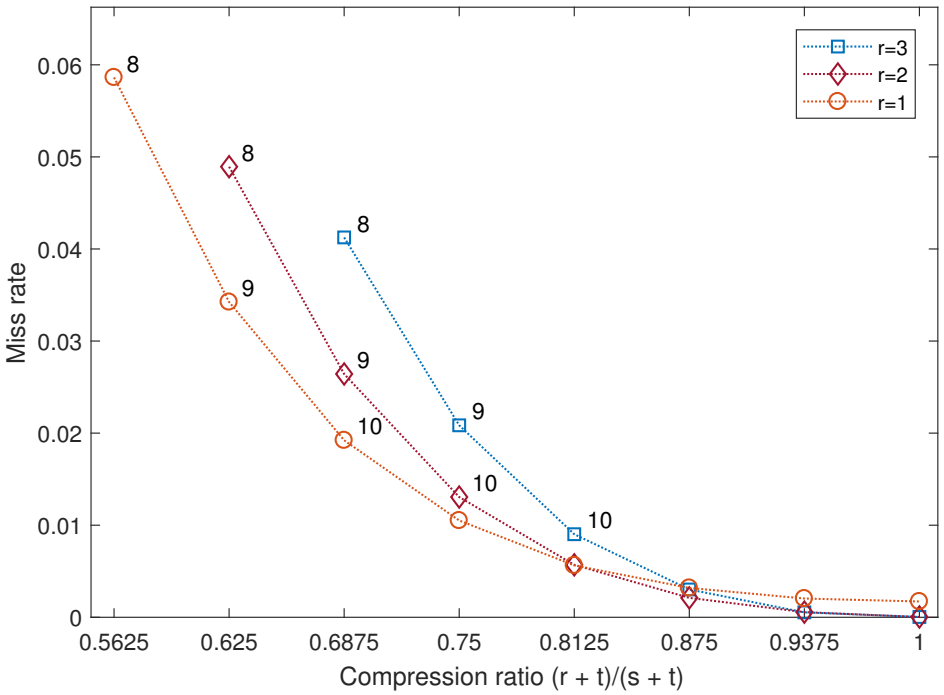

Figure 6: Cache-based algorithm simulation, current measurement at the DCDC converter, $r = 1$, $r = 2$, and $r = 3$ (i.e., dictionary sizes 1, 3, and 7). The tail length t is written to the data points. Lines between data points are shown for illustration only.

Figure 6 shows the relationship between the miss rate and the compression ratio $(r + t)/(s + t)$ with different settings for the parameters r and t. For the experiments we used the samples of the current measurement of the DCDC converter from our Simulink model during a WLTP[5]-Class 3 driving cycle simulation (the following plots are all based on this signal). The signal is suitable for the evaluations as if offers a challenging signal behavior and can be seen as a representative of signals often to be analyzed in diagnosis systems. It includes a broad coverage of quantization levels, slowly varying as well as rapidly varying signal sequences. The data set contains 180100 samples that were quantized with 16 bits and a sampling frequency of 100 Hz. In Figure 6 we see that up to a compression ratio of 0.75 smaller dictionary sizes (resp., larger block sizes) show lower miss rates. For compression ratios greater than 0.875 the dictionary size has less influence and the miss rates are significantly below 1 %.

Figure 7 highlights the improvement of the cache-based algorithm described in Section 3.3.2, where the $t - s$ unused bits are used in order to store more active heads. For $t > 8$ this indeed leads to a smaller miss rate (see the line with the diamond markers in Figure 7).

Figures 8 and 9 compare the performance of the cache-based algorithm with the dynamic cache-based algorithm for different dictionary sizes. For all compression ratios the dynamic cache-based algorithm shows a lower miss rate for $r = 1$ and $r = 3$. For a compression ratio of 0.6875 the miss rate is reduced from about 2 % to significantly below

[5]Worldwide Harmonized Light-Duty Vehicles Test Procedure; https://www.unece.org/fileadmin/DAM/trans/doc/2014/wp29/ECE-TRANS-WP29-2014-027e.pdf

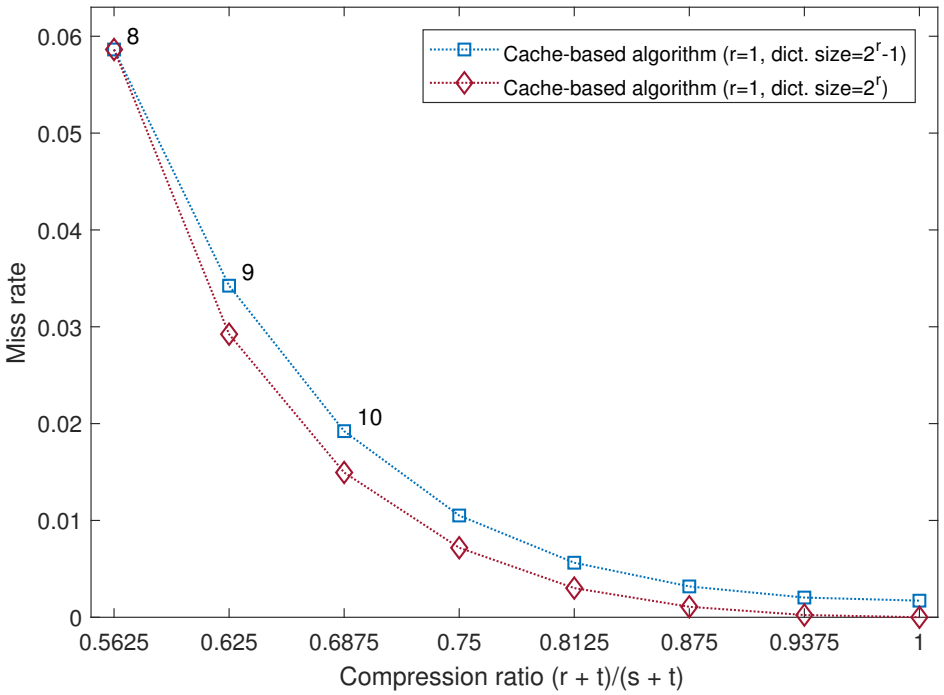

Figure 7: Comparison of the cache-based algorithm (current signal) with its improved version supporting an increased dictionary size of 2^r for $r = 1$. The tail length t is written to the data points. Lines between data points are shown for illustration only.

1 % (Figure 8).

A comparison of the cache-based algorithm and difference coding is shown in Figure 10. The line with the diamond markers refers to the case $s = 1$ where we arrange a new reference value at the sender and the receiver based on the largest (resp., smallest) transmittable difference in the case of a miss. This method can be explicitly employed as a fallback strategy for the case that a defined maximum number of consecutive misses occurred during operation with $s > 1$. With the new reference value approaching the data value typically within a few iterations, a successful value transmission can be reestablished. The line with the circle markers in Figure 10 shows the best performance, i.e., the lowest miss rate. In a miss case, the default head of the value is transmitted and the reconstructed value then is the center of the corresponding block. It maximally deviates by $\pm 2^{t-1}$ from the original value.

For the difference coding algorithm, Figure 11 visualizes the trade-off between the miss rate and the block size 2^t (resp., number of heads 2^s), which directly corresponds to the uncertainty about the reconstructed data values in the event of a miss. For a fixed compression ratio (determined by the choice of r) the minimum number of bits per block is given by $t_{min} = (s + t) - (r - 1)$. For instance the line with the square markers ($r = 9$, leading to a compression ratio of 0.5625) starts at $t_{min} = 16 - 9 + 1 = 8$.

Recall that our architecture only allows occasional misses and that in all other cases values are communicated error free. A higher accuracy for the reconstructed data values in the event of a miss is achieved with smaller block sizes, as the potential range of the correct value is narrowed. However, this increases the number of heads 2^s, meaning

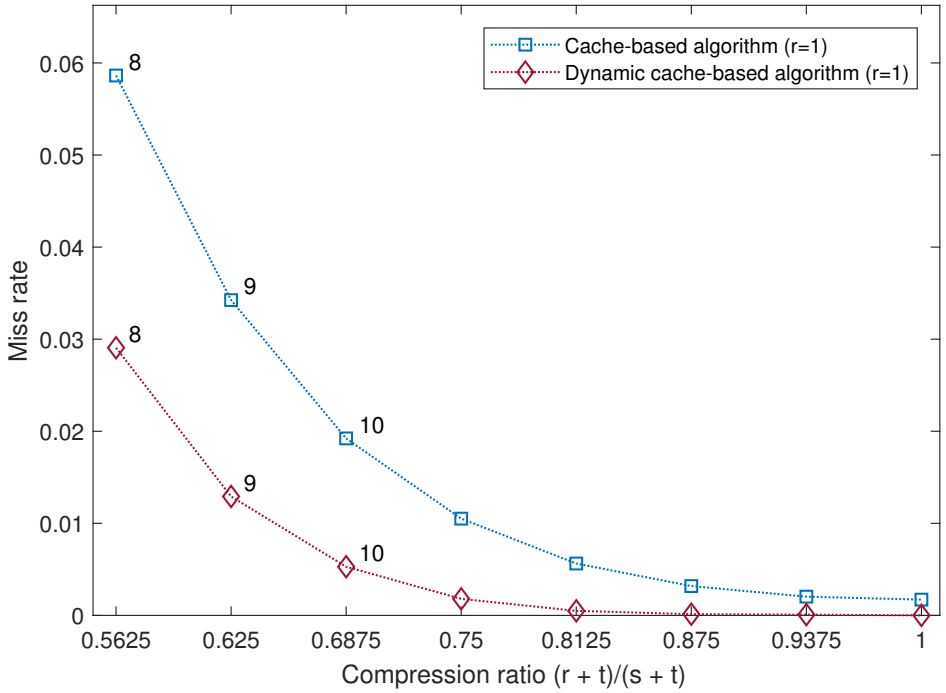

Figure 8: Comparison of the cache-based algorithm with the dynamic cache-based algorithm (current signal) for $r = 1$ (i.e., a dictionary size 1). The tail length t is written to the data points. Lines between data points are shown for illustration only.

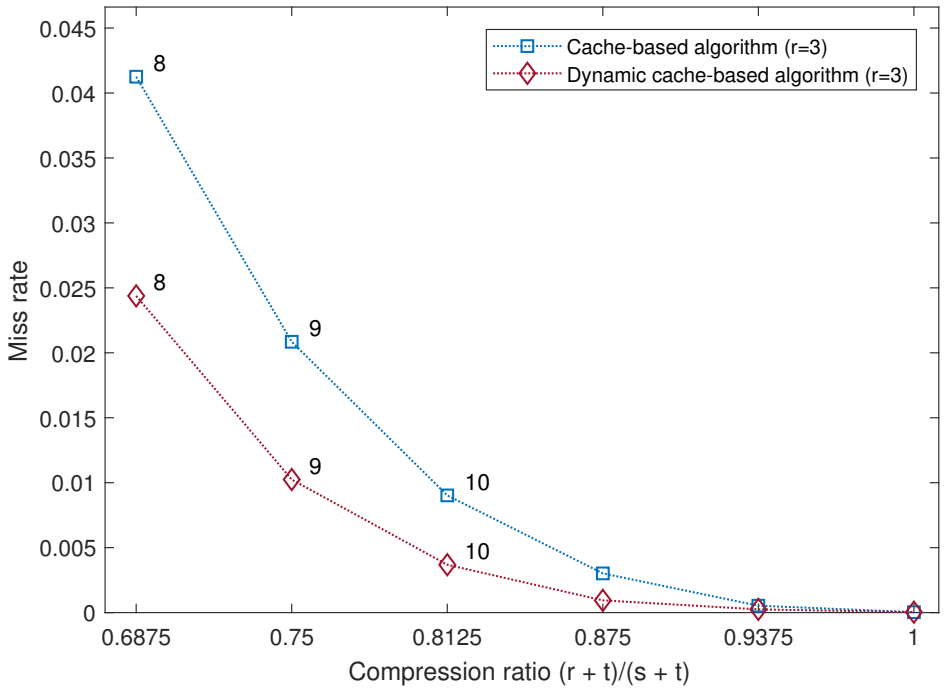

Figure 9: Comparison of the cache-based algorithm with the dynamic cache-based algorithm (current signal) for $r = 3$ (i.e., a dictionary size 7). The tail length t is written to the data points. Lines between data points are shown for illustration only.

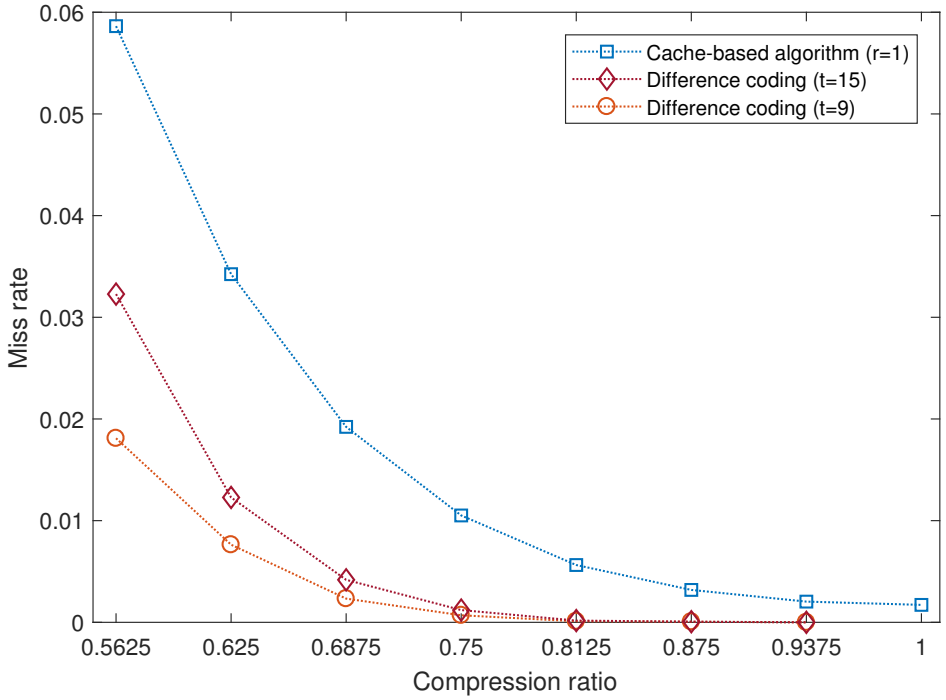

Figure 10: Comparison of the cache-based algorithm ($r = 1$) with difference coding schemes ($t = 9$ and $t = 15$). Lines between data points are shown for illustration only.

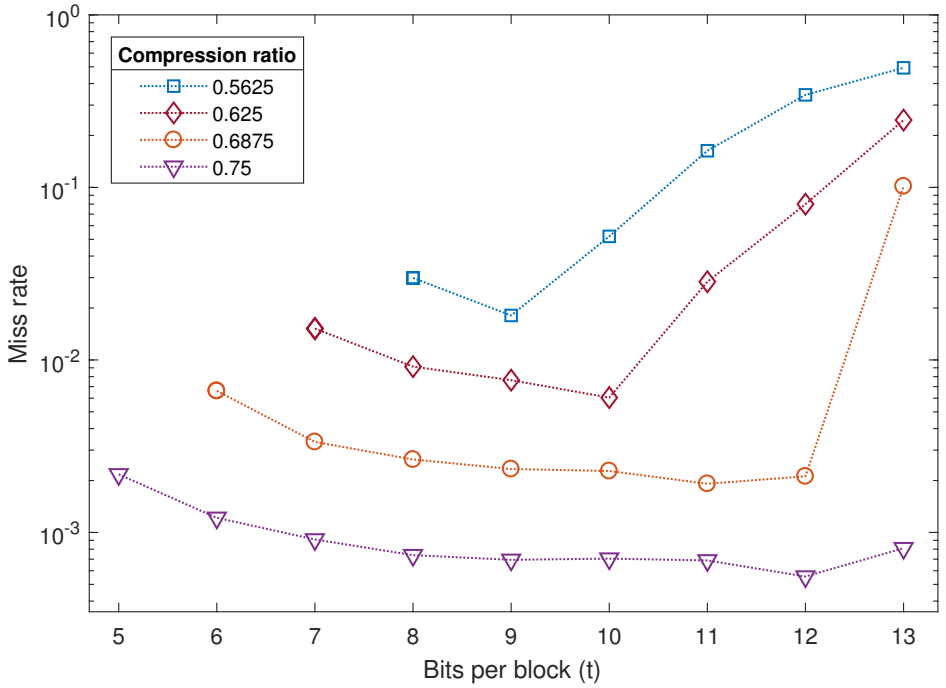

Figure 11: Comparison of different block sizes for the difference coding scheme with miss handling. Lines between data points are shown for illustration only.

less entries of the dictionary are available for encoding differences, which again may have a negative impact on the miss rate. At the same time, a lesser number of heads results in an increased block size. As a consequence, if there is a miss, the difference of the next value to the reference (center of the block) is more likely to be out of the range for differences, leading to another miss and generally to a higher miss rate. These two effects yield an optimal block size for each compression ratio, where the miss rate becomes minimum. For instance, using a compression ratio of 0.625 (the line with the diamond markers in Figure 11), within the range $7 \leq t \leq 10$ more dictionary entries for differences directly lead to a lower miss rate until the lowest miss rate is achieved at $t = 10$ bits per block. Within the range $11 \leq t \leq 13$ the rising miss rate is due to consecutive misses as a result of the increasing difference to the center of the block.

With the difference coding algorithm we introduce another version of our online compression algorithms. The delay due to the compression is reasonably small, as the difference calculation and mapping is typically little time consuming. Comparing the Figures 8 and 10 we see that the difference coding approach shows a slightly lower miss rate than the dynamic cache-based algorithm for all compression ratios. It is to be mentioned that the difference coding scheme shows the disadvantage of a basically unlimited number of consecutive erroneous value reconstructions, once the reference is wrong (e.g., through a transmission error, which is not a topic of this paper). A solution would be to periodically (and intentionally) transmit the head u of a value i instead of the difference d. This synchronizes the reference value, though, decreases the accuracy of that value. Another problem may occur if a signal is very volatile, i.e., consecutive data values have large differences. A proper signal reconstruction may become impossible with only the heads transmitted. Using the original cache-based (or dynamic cache-based) algorithm, this situation may be mastered if the required data values can be covered by few blocks.

Due to the typically lower miss rate but a possibly more difficult behavior in the event of misses (or in case transmission errors are assumed), both algorithms can be combined via a fallback strategy. For example, if the difference coding algorithm is applied and consecutive misses exceed a defined threshold (e.g., if the signal volatility is high), the sender and the receiver synchronously switch to the cache-based algorithm to potentially benefit from more active blocks covering independent value intervals. Analogously they switch back if only few misses occur to profit from an even lower miss rate of the difference coding algorithm.

There are several criteria for the selection of a compression scheme, some of which are the desired compression ratio, the expected signal behavior (e.g., slow or fast varying signals, volatility), tolerable miss rate, required accuracy of a data value in case of a miss, or the robustness of the communication in general.

Assuming a fixed compression ratio (in a time-triggered system) both compression schemes (cache-based approach and difference coding) have distinctive characteristics: The cache-based algorithm offers the possibility to change the relation between the number of active blocks and corresponding block size. Some signals can be better covered with fewer but larger blocks whereas for others more but smaller blocks work better. The difference coding algorithm, on the other hand, allows to change the relation between the number of entries of the dictionary used to encode differences and the corresponding block size. According to the example from Figure 11, for a compression

ratio of 0.625 a miss rate of 0.6 % can be achieved along with an uncertainty interval of $2^{10} = 1024$ ($t = 10$) values. Alternatively, with a miss rate of about 1.5 % the uncertainty interval decreases to $2^7 = 128$ ($t = 7$).

All of our introduced compression schemes utilize an (adaptive) dictionary to map input values of a fixed size to output values of a fixed size. This is in contrast to classical dictionary based approaches like [4], where more frequent symbols (or data values) are assigned to shorter bit strings and less frequent symbols to longer bit strings. In terms of diagnosis, longer messages (the scheduler always uses the worst-case message size) more likely slow down the overall execution of the logical model (the makespan, see Section 4.4) and since this time is consumed to come to a diagnostic decision a fixed-rate sample-wise compression optimally suits online fault-diagnosis in real-time systems. A fixed-size dictionary output is a huge advantage for time-triggered systems, as it accounts for the worst-case message size.

4.4. Minimizing Latency

We scheduled the DDAG from Figure 5 to a network and analyzed the communication times between the computation nodes. Tasks observing similar objects are allocated on the same node. For simplicity the network is designed in such a way that each channel can use two completely disjoint routes. Because of the network design and the allocation there is no need to consider collisions.

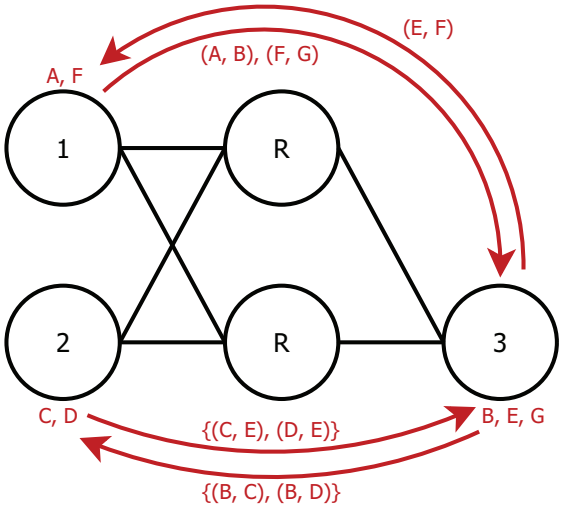

Figure 12: Network with allocated tasks and communication

Figure 12 shows the network and the 7 tasks $A, B, \ldots, G$ allocated to the three computation nodes as well as the communication links. Note, that the stated allocation is not optimal as the concurrency of C and D is not exploited. The communication pattern of the DDAG is shown by red directed egdes between the involved computation nodes. These edges are labeled with channels (see Section 2.1). A group of channels surrounded by curly braces means that these channels are mapped to disjoint communication paths (via the two routers). Hence, for the computation of the makespan, such a group can be considered as one channel. Note, that the two channels (B, G) and (E, G) are not present in Figure 12 because the three tasks B, E, G are mapped to the same computation node. In this example we

assume a worst-case execution time (WCET) of 4 ticks per task and a per-hop transmission time of 8 ticks per message. Hence, we can calculate the makespan as $7 \cdot 4 + 5 \cdot (2 \cdot 8) = 108$ ticks (7 tasks, each requiring 4 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. Assuming the compression parameters WCCR = 0.75 and WCCT = WCDT = 1, we are able to decrease the makespan. Task A must handle only one compression for channel (A, B) and G must handle only one decompression for channel (F, G). Therefore, both end up with an accumulated WCET of 5 ticks. Tasks C, D and F must each perform one compression *and* one decompression. They all end up with a WCET of 6 ticks. The remaining tasks B and E must handle two compressions and one decompression (B) or one compression and two decompressions (E). They both end up with a WCET of 7 ticks. The per-hop transmission time of the messages is reduced to $6 = 8 \cdot 0.75$ ticks. The makespan of the DDAG with compression is $2 \cdot 5 + 2 \cdot 7 + 3 \cdot 6 + 5 \cdot (2 \cdot 6) = 102$ ticks.

5. Future Work

In this section we provide an overview of research activities planned to improve the proposed architecture.

5.1. Extension of the Diagnostic Architecture

The diagnostic architecture is based on different atomic components. In order to extract features from different streams, these atomics can be combined to more powerful blocks. Complex processes may require different procedures. For this, a library of diagnostic feature extraction blocks along with interfaces and instruction sets will be established to allow an application specific and user-friendly integration of the diagnostic methods for many fields. Creating a DDAG is often complex, time consuming, and requires knowledge from human system experts. Machine-learning algorithms can help to extract DDAGs if knowledge about the system behavior can be obtained from a system model (e.g., simulation) or if adequate recorded data is available. Especially for the handling of huge and complex DDAGs we aspire an extension of the DDAG generation process based on machine-learning. A tool for timing analyses of the diagnostic feature extractions and all other processing blocks is also necessary, in order to provide worst case execution times and thus, time guarantees for the fault diagnosis.

5.2. Compression library

With the introduced compression schemes we established a library of different online compression algortihms, all suitable for the diagnostic architecture and providing real time guarantees. Depending on the expected signal behavior or the application, the best suitable compressor is integrated. The combination of several algortihms is also possible. Future work will concentrate on the utilization of correlations between multiple data streams. As first experiments show, the combined compression of correlated data streams can be efficiently used to achieve better compression results.

5.3. Scheduler

The scheduling model introduced in Section 2.4 is to be implemented. Our goal is to calculate static schedules considering different compression methods using genetic algorithms. Although these types of algorithms will not necessarily solve the problem optimally, experiments show that we are able to obtain near-optimal solutions in a feasible amount of computing time. Furthermore, the model needs to be generalized, so that a computation node can handle multiple jobs.

6. Conclusion

In this paper, we firstly motivated the usage of data compression in distributed online-diagnosis systems. We then presented an architecture for such a system by defining models for diagnostic graphs (DDAG), networks of computation nodes and routers, as well as a model for compression schemes that allows to provide real time guarantees. After discussing the scheduling model that now includes compression, a lossy cache-based compressor and a difference coding scheme with guaranteed compression ratios and low costs were proposed. With examples we showed that depending on the compression algorithm and the parameters more than one fourth of the bits (25 %) can be saved, while still not loosing more than about 1 % of the values (cache-based algorithm, $r = 1$, Figure 6). Considering the same signal and parameters, the dynamic cache-based algorithm even reduces the miss rate to 0.18 % (Figure 8). In turn, this means that about 99.8 % of the values are delivered correctly and without any loss, but with a significant reduction of bandwidth demands. Moreover, with the additional algorithm improvements (Section 3.3.1), the uncertainty of the correct data value in the case of a miss can be strongly reduced. The difference coding algorithm provides even lower miss rates (Figure 10). Its robustness, however, may not be as high in challenging situations (e.g., in the case of high signal fluctuations due to faults in the system). As fall-back strategies are supported, the algorithms complement each other. The small loss-rate, the low costs and the fixed compression ratio make the compressors suitable for real time purposes.

Acknowledgment

This work was supported by the DFG research grants LO748/11-1 and OB384/5-1.

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

[2] R. Obermaisser, R. I. Sadat, F. Weber, Active diagnosis in distributed embedded systems based on the time-triggered execution of semantic web queries, in: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on, IEEE, 2014, pp. 222–229.

[3] K. Sayood, Introduction to Data Compression, fifth edition, Morgan Kaufmann, 2018.

[4] S.-W. Seong, P. Mishra, Bitmask-based code compression for embedded systems, IEEE Transactions on computer-aided design of integrated circuits and systems 27 (4) (2008) 673–685.

[5] F. Marcelloni, M. Vecchio, An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks, The Computer Journal 52 (8) (2009) 969–987.

[6] S. Jo, M. Lohrey, D. Ludwig, S. Meckel, R. Obermaisser, S. Plasger, An architecture for online-diagnosis systems supporting compressed communication, in: Digital System Design (DSD), 2017 Euromicro Conference on, IEEE, 2017, pp. 62–69.

[7] S. X. Ding, Model-based fault diagnosis techniques: design schemes, algorithms, and tools, Springer Science & Business Media, 2008.

[8] Z. Gao, S. X. Ding, Y. Ma, Robust fault estimation approach and its application in vehicle lateral dynamic systems, Optimal Control Applications and Methods 28 (3) (2007) 143–156.

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

[10] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1) (1987) 57–95.

[11] H. Henao, G.-A. Capolino, M. Fernandez-Cabanas, F. Filippetti, C. Bruzzese, E. Strangas, R. Pusca, J. Estima, M. Riera-Guasp, S. Hedayati-Kia, Trends in fault diagnosis for electrical machines: A review of diagnostic techniques, IEEE industrial electronics magazine 8 (2) (2014) 31–42.

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

[13] D. Ludwig, R. Obermaisser, Scheduling of datacompression on distributed systems with time-and event-triggered messages, in: International Conference on Architecture of Computing Systems, Springer, 2017, pp. 193–204.

[14] P. Flajolet, D. Gardy, L. Thimonier, Birthday paradox, coupon collectors, caching algorithms and self-organizing search, Discrete Applied Mathematics 39 (3) (1992) 207–229.

[15] P. A. Franaszek, T. J. Wagner, Some distribution-free aspects of paging algorithm performance, Journal of the ACM 21 (1) (1974) 31–39.