Flexible In-Vehicle Stream Processing with Distributed Automotive Control Units for Engineering and Diagnosis

Hendrik Schweppe*†, Armin Zimmermann* and Daniel Grill†
*Technische Universität Berlin, Germany,
{schweppe,zimmermann}@cs.tu-berlin.de
†Mercedes-Benz Research and Development of North America Inc., Palo Alto, USA,
daniel.grill@daimler.com

***Abstract*— This paper introduces a method for selectively preprocessing and recording sensor data for engineering testing purposes in vehicles. In order to condense data, methodologies from the domain of sensor networks and stream processing are applied, which results in a reduction of the quantity of data, while maintaining information quality. A situation-dependent modification of recording parameters allows for a detailed profiling of vehicle-related errors. We developed a data-flow oriented model, in which data streams are connected by processing nodes. These nodes filter and aggregate the data and can be connected in nearly any order, which permits a successive composition of the aggregation and recording strategy. The integration with an event-condition-action model provides adaptability of the processing and recording, depending on the state of the vehicle. In a proof-of-concept system, which we implemented on top of the automotive diagnostic protocols KWP and UDS, the feasibility of the approach was shown. The target platform was an embedded on-board computer that is connected to the OBD-II[1] interface of the vehicle. As the scope of recording can be adjusted flexibly, the recording system can not only be used for diagnostic purposes, but also serves objectives in development, quality assurance, and even marketing.**

I. Introduction

The complexity of vehicles has increased in recent decades and will continue to increase significantly in future [1]. Until the late 1960s, cars were basically mechanical systems with a few electrical appliances, e.g., for engine spark and lighting. Modern vehicles are complex electro-mechanical systems with dozens of networked electronic control units (ECUs). ECUs enable or implement vehicle core functions such as power-train control, suspension control, safety, convenience functions, and infotainment. They are connected to a large number of sensors and actuators which they control. ECUs exchange information about their current sensor values over internal networks (for example a CAN bus), so that multiple redundant sensors are avoided. The kinds of sensor types used in the car are of a great diversity, ranging from pressure sensors over temperature to acceleration and contact sensors.

[1]OBD-II is the On-Board-Diagnostics interface standard. The diagnostic protocols KWP (Keyword Protocol) and UDS (Unified Diagnostic Services) operate on top of OBD-II.

Data resident and stored on ECUs and data exchanged between ECUs describe, from an IT-standpoint, the *state* of the vehicle at any time. To capture, analyze, and interpret this data are important activities of engineering testing and quality control processes. In recent years, the availability of cost-effective in-vehicle and off-board computing and communications systems enabled the systematic acquisition and processing of data from many vehicles over long periods of time. This is particularly important for the late engineering and early production phases of a vehicle's life cycle. Despite the improvement in computing infrastructure, however, since data volume generated from hundreds of sensors (see Fig. 1) delivering data at a high frequency is immense, it is still necessary to utilize filter and data aggregation mechanisms and only record detailed data for specific situations.

Another important application is the area of vehicle diagnosis. Current vehicle diagnosis relies on error codes (DTCs), that the corresponding ECU sets. These error codes may be retrieved via the on-board diagnostic socket (OBD-II), which became a mandatory standard for the United States in 1996. This diagnostic interface also allows to acquire the various sensor data from the ECUs of a vehicle.

Many approaches for data recording in the vehicle are rather

Fig. 1. ECUs and buses of Mercedes-Benz W211 vehicle. From DC-Media.

978-1-4244-1995-1/08/$25.00 ©2008 IEEE.

inflexible as far as *how*, *when* and *which* data are gathered, transformed and processed. This work aims at a flexible data aggregation system, that can dynamically adapt its behavior, depending on the specific state of certain subsystems of the vehicle (e.g. the engine). As a reaction to critical events, our system is able to read and process sensor values at a higher rate. The adaptiveness can also be employed in order to store data selectively.

The current state of the art of different topics in computer science, data stream processing, and lossy data storage, are combined with best-practice approaches for system design in the context of in-vehicle embedded systems and reusability. The approach described in this paper focuses on facilitating flexibility and maintainability of such a system. Processing elements and communication paths are part of a data flow architecture and may be adapted to a change of the state of the vehicle on-the-fly.

The main contributions of our work are:

- An *embedded and re-usable* platform, based on the OBD-II/CAN interface and KWP/UDS protocols.
- An *adaptable* stream-based recording system.
- *Situation-dependent* processing and recording of data.
- An event-system that allows *dynamic reconfiguration* on-the-fly.
- *Space-efficient* recording of multidimensional data.
- An on-board *prototype* system.

This paper is organized as follows. In Section II we briefly discuss related work and the state of the art of automotive logging systems. In Section III we introduce the conceptual data-flow architecture of the processing system and its adaptability. Section IV describes the implementation of the prototype system. An evaluation and selected real-world examples are demonstrated in Section V. We conclude our paper with an outlook on future work. For reasons of space, we do not discuss all technical details of model and implementation. They can be found in [2].

II. Related Work

Our work combines the industrial problem of remote vehicle diagnosis—keeping costs low while maintaining flexibility and quality of recorded data—and research aspects from different fields. The main field that caught our interest was stream processing of sensor data. Sensor networks are fundamentally different in their organization, i.e., the network consists of individual, rather autonomous nodes. On the other hand, their functionality, i.e., diagnosing the system state and aggregating raw data, is closely related to automotive problems.

A. Industry Practices

Existing systems for on-board data recording differ in the quality and quantity of the data they crop. Some systems aim at logging all available data for selected channels to a huge in-car storage, which requires heavy and expensive equipment. For example, an Australian transmission company developed a recording system in 2001, using tape recorders in combination with a Linux PC to record sensor-data at a high resolution [3]. The vehicles were test-driven for up to 100,000 kilometers, during which the tapes with gigabytes of data were regularly sent back to the company for evaluation. The same company, only a few years later (2004), has put effort into small-footprint logging systems for drive shaft analysis, where data are compressed and approximated prior to recording [4]. As opposed to data-intensive logging systems, there are a number of systems with a small footprint. They record only few data, for example DTCs. General Motors was the first company that introduced such a system: OnStar [5]. They coupled the ability of remote-diagnosis with convenience and security functions for market acceptance reasons. Since 1999, Mercedes-Benz offers a similar emergency and telematics system called TeleAid. The system integrates the vehicle's internal networks with a remote radio interface and demonstrates the significance of software development for safety and convenience systems [6]. Future remote vehicle diagnostic systems will have to provide a more detailed insight into the car's state and might even *"detect the need for preventive maintenance"* [1].

B. Stream Processing

Stream processing systems define a *computational pattern*. Data are processed on-the-fly, so that the original data, that may be of rather high volume, can be discarded after the computation and only aggregated and filtered data need to be saved or evaluated further. It is related to data flow architectures in a way, that data are processed immediately, if available at the inputs.

Requirements and demands for processing streams vary largely by the given constraints and the objectives, so that research in the field of stream processing spreads out rather widely. Golab and Özsu compiled a comprehensive overview of stream processing approaches [7]. On the one hand, there are systems to process large amounts of trading data or position data [8], [9] with real-time quality of service requirements [10] (Aurora, Borealis). On the other hand, stream processing solutions are influenced by traditional relational data base systems [11] (Stanford's STREAM), where so-called continuous queries are applied to data streams. In contrast, other approaches focus on distributed sensor and processing nodes [12], [13], that allow in-network aggregation [14].

Large stream processing systems, such as Aurora, Borealis, Stream and TelegraphCQ are considerably too large for an embedded deployment. The most promising work as in-vehicle system is probably VEDAS [15], which follows a distributed data-mining approach for automotive sensor data. Their approach also collects aggregated, statistical data, but does not allow an on-the-fly modification of the data processing.

III. System Design

The underlying model of our stream processing system is a directed acyclic graph (DAG), which consists of different kinds of processing nodes. These processing nodes *(operators)* are connected by data paths as edges *(streams)*. We call this graph a *Stream Processing Graph (SPG)*. The graph can be

adjusted to different situations to allow a varying scope of data processing and recording. The triggers for the adjustment of the graph are defined as part of the graph itself.

The extension of the stream processing graph by event-action pairs allows an adaptation of the graph at runtime. We introduce event triggers for the re-arrangement and re-configuration of the SPG. The two major problems of previous recording systems in the vehicle are addressed: The *customized* aggregation and filtering of data is now possible and actions may be taken upon *individually defined* events.

We will first present a formal model of the graph. The model and the terms used are inspired and adopted from operational stream processing research [16] and formal specification of data flow graphs [17] as well as data flow design principles. Our stream processing model is similar to, but weaker than the SPF (Stream Processing Function) algebra of Broy et al., which is a so-called Basic Network Algebra as discussed in [17]. Our model does not allow direct data feedback loops that explicitly are part of the SPF algebra. We use control inputs and an adaptation of the graph topology and operation parameters as feedback principle. We describe the purposeful modification of the graph in Section III-C.

A. Model of the SPG

The Stream Processing Graph is a DAG $G = (N, E)$ consisting of nodes N and edges E. Edges are a subset of all possible connections between the nodes $E \subset N \times N$[2].

The nodes N of the graph represent *operators*, which process *data streams*. Edges of the graph represent the flow of data between operators. An edge $e = (n_1, n_2)$ represents the flow of data between node n_1 and node n_2. We denote the space of all possible graphs by $\mathbb{G}$.

Nodes without incoming edges are *source nodes* and nodes with no outgoing edges are *sink nodes*. We say a node with in- and outputs is an *inner node*. As a matter of fact, data flows from source nodes to sink nodes. Source nodes produce data obtained from sensors or alike, while sink nodes take *action* on arriving data. The most common action is to *save* incoming data. Inner nodes of the graph process data from their incoming edges and output processed data to outbound edges.

Data within an SPG are encapsulated as data packets, called *tuples* $\mathfrak{t}$. They are called tuples, because they do not only contain the data but also meta data, such as a time-stamp of creation.

Definition: A *tuple* $\mathfrak{t} = (val, \tau, m) \in T$ consists of a value val, a time-stamp τ and a mileage stamp m, which are addressed with $\mathfrak{t}.val, \mathfrak{t}.\tau$ and $\mathfrak{t}.m$. The set T is the tuple space.

Definition: A *stream* $s = (\mathfrak{t}^*)$ consists of an ordered sequence of tuples $\mathfrak{t}^*$, so that for all tuples $\mathfrak{t}_1, \mathfrak{t}_2 \in s$ that fulfill $\mathfrak{t}_1 \leq \mathfrak{t}_2$, the tuple $\mathfrak{t}_1$ will be before $\mathfrak{t}_2$ in s. The relation $\leq$ reflects both time and mileage. S denotes the set of all streams. A label function $m_e : E \mapsto S$ maps all edges of the graph to corresponding streams. We have deliberately not included an absolute time t in this mapping for reasons of complexity.

[2]It is a real subset, because a complete graph is not a valid graph structure within the posted limitations.

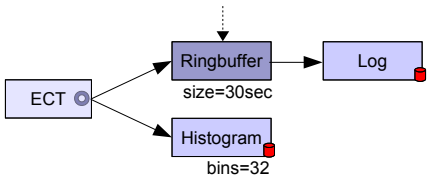


Fig. 2. Sensor data (ECT: Engine Coolant Temperature) are sent to a thirty-second ring buffer as well as to a histogram. The ring buffer's size may be adjusted through the control input, which can also be used to permanently save tuples from the ring buffer. The control input is used by a different end-operator (not shown).

Definition: An *operator* $o \in O, o = (f_k, \sigma, p)$ consists of a processing function f_k for k input streams, and a state $\sigma \in \Sigma$, where Σ is the set of all operator states. An operator is tuned by a parameter set p, where each parameter consists of an identifier id and the parameter value val, e.g., $p = \{(\mathfrak{p}, \top)\}$, where $\mathfrak{p}$ is the filter-predicate identifier and $\top$ the value 'true'. The function f_k is a mapping $S^k \times \Sigma \mapsto S^l \times \Sigma$. The number of inputs k of the corresponding graph node n depends on the number of input edges. All l output streams are supplied with identical output, which is an implicit branch of the stream, so that multiple successive operators retrieve the output tuples. In the Basic Network Algebra [17] such a branch is denoted by a separate *split* operator ($\bigwedge$).

Not every operator needs to keep a state. We distinguish between *stateful* and *stateless* operators. A state is, for example, a preliminary output result. A filter operator is a stateless operator, because the processing any tuple $\mathfrak{t}_i$ is independent from earlier tuples $\mathfrak{t}_{i-n}$. Whenever the previous input has an influence on the output of the operator, it is a stateful operator. The operator's state is updated, whenever a tuple is processed.

We have adopted the graphical representation of operators and streams as *boxes and arrows* that was suggested in [16], [18] and [9]. The data flows from source nodes (operator boxes on the left) to sink nodes on the right. We introduce another kind of arrow: We allow to change the parameters via a control input, that is depicted as a dashed arrow at the affected operator(s).

Let us give a comprehensive example of a small SPG, which is shown in Figure 2. The input sensor source ECT (Engine Coolant Temperature) is branched to two operators: A thirty second ring buffer and a histogram. There is a simple logging store connected to the ring buffer. The histogram and log store do not have outgoing edges (they are sink nodes) and record the incoming data. The ring buffer's data are not stored persistently, unless a special signal is given by the control input (denoted above the histogram operator with origin from an event-condition that is not shown). The operator's size parameter may also be changed by using the control input.

Here, the adjustable parameter for the ring buffer is a window-size, which is explained in the following section.

There may be several of these simple arrangements in the system, e.g. for every sensor source to be recorded. This demonstrates that the graph does not need to be connected.

B. Window Operators

Operations on data streams are typically performed on so-called *windows*. This allows for example to reduce a sequence of tuples from the input to only a single tuple as the output. Every operator with a given window size > 1 holds a state σ, which represents either the current preliminary result or buffered, unprocessed input tuples for a later batch processing, e.g. when the the calculation requires more than one pass over the data. The state is updated whenever the operator function processes incoming tuples or produces outgoing tuples.

A window consists of a finite number of consecutive tuples of a stream. The tuples of a stream are assigned to windows upon arrival. They are processed with respect to these windows. We denote the *size*[3] of a *window* by ω.

Windows can be defined over different attributes. They are typically defined over the *time* dimension: the window size is given as an interval size in time. However, a window size can also be defined by a *count*. Then the window of size ω consists of exactly ω tuples. In that case, the time difference of the tuples within this count-window is variable as opposed to the number of tuples, which is fixed. A third dimension, most important in the automotive application context, is *mileage*. For determining whether a tuple is within the boundaries of a specific window, their time or mileage stamp is compared to the window's start time or mileage. We call the dimension, over which a window is defined, the *windowing attribute* D.

The tuples of windows are less or equal than ω apart, so that for the first tuple t_f and the last tuple t_l of a time based window, the expression $\mathrm{t}_l.\tau - \mathrm{t}_f.\tau \leq \omega$ is always valid[4] .

An incoming tuple t falls into a window w if the tuple's time-stamp $\mathrm{t}.\tau \in [t_f.\tau, t_f.\tau + \omega]$, where t_f is the first tuple of window w. If the tuple does not fall into window w, w is *closed*, because the stream's metadata are monotonic increasing and thus the window size is reached. This means that the first tuple t_f of w is at least ω apart from the new tuple t, so that $\mathrm{t}.\tau - \mathrm{t}_f.\tau > \omega$. The window can now be processed by the operator's aggregate function, which may require more than one pass over the data. If only one pass over the data is required, an iterative calculation of the result may be performed, so that the tuples of the current window do not need to be saved[5]. The operator's function f calculates a new tuple, which is the output of the operator – for example the mean value within the given window. The output tuple's meta information (i.e., the time and mileage stamp) is always taken from the *first* tuple of each window.

We allow more than one window to be open at a time, resulting in overlapping windows. The parameter *window slide* δ indicates when (i.e., after how much time elapsed since the start of the last window) a new window shall be opened. For the simple case $\delta = \omega$, windows are non-overlapping, so that there will be exactly one open window for processing at a time. These windows are called *tumbling windows* [10] [19].

C. Actions and Dynamic Reconfiguration

We now describe how our concepts of graph adaptation and recording of data are modeled. Sink nodes (without outgoing edges) are so-called *end operators*. End operators trigger the recording of tuples or an adaptation of the processing and recording behavior of the system by altering other operator's parameters or the graph structure.

At the end operators of an SPG, data has already been filtered and processed and thus *"each path from a sensor input to an output can be viewed as computing the condition part of a complex trigger."* [18]. End operators represent event-condition-action (ECA) rules. The arrival of a tuple on the input stream may be regarded as a *basic event*. One event triggers one or more actions. In order to be more flexible, the actions taken upon a "tuple-arrival-event" may also depend on boolean conditions c_i.

Definition: An *end operator* o_e is defined by at least one input stream and a set of condition-action rules (c_i, a_i). A condition c_i is a predicate defined on tuples of the input stream. If $c_i(\mathrm{t})$[6] is true, action a_i of the pair (c_i, a_i) is executed.

In case of rules $r_i = (c, a_i)$ and $r_j = (c', a_j)$ with $c = c'$, both actions a_i and a_j are performed. This is abbreviated as $r = (c, \{a_i, a_j\})$. Action a_i of the rule $(\top, a_i)$ is executed unconditionally. This means that the basic event, the arrival of any tuple, causes the execution of the action in this case.

Actions: We distinguish three types of actions, that may be taken at an end operator:

- *Storage* actions a^S, which do not modify the processing pattern,
- *Parameter modification* actions a^P and
- *Topology modification* actions a^T, which both change the system's behavior.

Storage actions a^S conform to the signature $a^S(\mathrm{t}, S)$, where S is a store location (e.g. memory or a file). The content of the store is addressed by $[S]$.

A simple (log) storage operator only *saves* all input data. It consists of an empty condition (i.e., $c = \top$) and an action that saves the input tuple. The action saves the incoming tuples by appending them to the store. Formally, this means $a^S(\mathrm{t}, S) = ([S]\ concat\ \mathrm{t})$. The store S can be regarded as a log file. Another kind of storage operator, that was used in the earlier example, is an equi-width histogram. Parameters

[3]The window size is sometimes also called a window range [19].

[4]To simplify matters, we use the windowing attribute *time* in our examples. Our considerations also apply for the other windowing attributes.

[5]Even when the result can be calculated iteratively and an additional buffering within the operator is not needed, it can still be advantageous, because it allows to change the window size in operation without a loss of data, as discussed in III-C.

[6]For n input streams it would be $c_i(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ with tuples from the different streams.

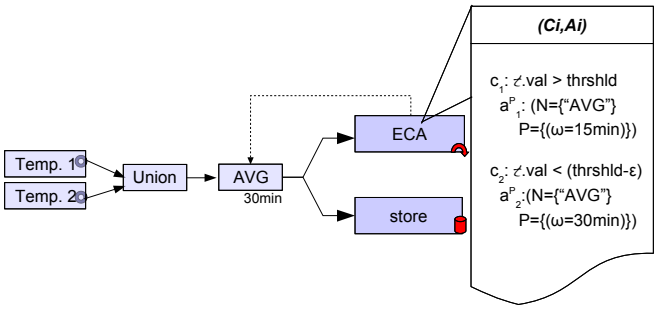


Fig. 3. Parameter Modification: Upon the conditions c_1 and c_2 the window size ω of the AVG operator is modified.

for this operator are the number of quantization levels, a minimum, and maximum value to.

An action $a^P(N, P)$ with $P = \{(p_1, \ldots, p_n)\}$ modifies, for every operator node $n \in N$, the parameters $p_i = (id, val)$ of the operator's parameter set to new values.

We want to give another example of an SPG, that makes use of parameter changes. Two temperature values are merged into one stream in pairs by a union operator and then averaged over a time window. This results in a stream that carries the average of both temperature sensors (e.g., left and right cylinder head) of the past thirty minutes. For a more detailed recording of this aggregate, the window size of the node $n = AVG$ can be reduced. Figure 3 shows the condition action rules: The parameter ω is changed to $15min$, if a threshold is exceeded ($c_1 : t.val > thrshld$). An opposite condition is shown as c_2: If the value drops under $thrshld - \epsilon$, the original window size of $\omega = 30min$ is restored[7]. The control input that was introduced earlier is depicted as a fine dashed line between the end operator and the affected operator node AVG.

If the parameters of a stateful operator are changed, its state σ has to be modified, too. It depends on the type of operator state (incremental or buffering), whether for example decreasing the window size ω is possible consistently. If the operator holds all previous tuples of all open windows in its state, a batch processing of the old tuples with the new window parameters can be performed. If the state of the operator is only a partial result of its computation, decreasing the window size is not possible consistently. There will be a gap in calculation to the next subsequent window result. To increase the window size and to change the parameters of stateless operators (e.g., filter predicate, map function) is always possible.

Actions a^T map $\mathbb{G} \mapsto \mathbb{G}$. The graph structure of the SPG is modified in a way that new nodes and edges are inserted and connected or specific nodes or edges are deleted. A modification of the graph also requires, for example when an inner node is deleted, an adequate modification of the graph by inserting and deleting edges. Actions a^T must always modify the SPG in a way that the graph stays consistent (i.e., inner nodes have a sufficient number of inbound edges and at least one outbound edge).

An intuitive example for modifying the graph is *adding* and *deleting* additional *storage nodes* for recording the streams at any source or inner node of the SPG upon condition and counter-condition. By this selective adding of additional recording points, specific situations, such as "at a high temperature", may be monitored selectively and in detail. In the example for parameter modification from Figure 3, an individual monitoring of the two temperature sensors for this situation is achieved by new actions to add (for condition c_1) and delete (for c_2) dedicated log operators to the sources.

[7]The ϵ is considered for hysteresis. Otherwise a bouncing between the two conditions would possibly occur and constantly change the window size between 15 and 30 minutes.

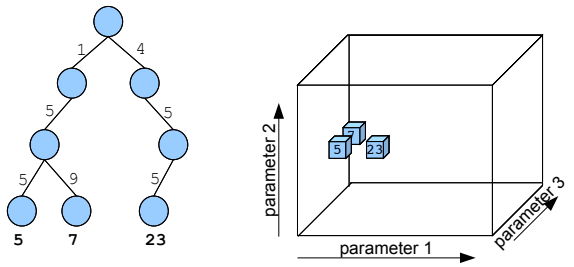
IV. Implementation

Fig. 4: On-Board Unit with OBD Interface and GSM/GPRS Uplink

We have implemented a prototype system on an embedded Linux telematic device, which is equipped with a CAN transceiver. We use the diagnostic protocols KWP and UDS to query the various ECUs over the OBD-II's diagnostic CAN bus. The ECUs are periodically queried for their sensor values and so produce the input streams of the SPG. The total volume of sensor data that can be acquired by the system depends on a) the bus topology, b) the load of the CAN buses, and c) the load of the individual ECUs, which may reject requests for sensor data when busy. Because of the arbitration mechanism of the CAN bus, security and safety relevant data are always prioritized before our requests.

The programming language C++ was used for the prototype. Inheritance allowed an encapsulation of communication and control flow structure of the SPG and thereby separating it from the operator functionality. All operators inherit functions that allow to connect them to any other operator of the SPG and safely operate on streamed data tuples. A "window operator" class encapsulates operations on sequences of tuples, such as keeping track of currently open and to-be-processed windows. Custom operators inherit these functions and only need to implement code for processing the input data accordingly. For concurrency, we use POSIX pthreads and follow a different approach for the execution for Linux kernel series 2.4 and 2.6. The 2.4 kernel does not support lightweight threads, so that we use threads conservatively. We use threads only for the sensor source operators, which trigger the execution of subsequent operators throughout the SPG. For 2.6, we employ threads for every operator. The event-system is always separated, so that a timely execution of actions is ensured.

Actions that change the stream processing graph's topology or parameters may be taken on event conditions, which have been defined as a path within the SPG. The afore mentioned "basic event" of a stream triggers actions. Every event is mapped to one or more actions, which themselves affect one or more operators of the graph. To give an example, an action to save time-series is applied to a set of ring buffers (default: all)



(a) Histogram as Trie (b) Histogram Space

Fig. 5. Trie as sparse data structure for multidimensional histograms. The count is saved within the leaves, the histogram's index as the key (the path to a leaf). Depicted is a three dimensional histogram with three non-zero entries.

for a defined window (default: 30 seconds). This action can be used in combination with different event trigger-conditions, so that time-series are triggered for different kinds of errors (e.g., drop of tire pressure or high lateral acceleration).

We implemented a number of processing operators, e.g., a filter operator with customizable predicate, a delta/gradient operator and window-aware versions of the standard functions *MIN, MAX* and *AVG*, along with a number of recording operators.

For complex technical problems such as engine management, it is often necessary to look at a *combination* of different sensor values (e.g., for engine RPM, throttle position, and intake manifold pressure). We implemented a *multi-dimensional histogram* storage operator, that allows multiple input streams with custom quantization attributes (min, max, num_{bins}). The input streams are synchronized within time windows. Because a multi-dimensional histogram is a very sparsely populated data structure, we had to store the data in an intelligent way, that a) allowed to save data space-efficiently, i.e., only to store non-zero elements and b) has an acceptable access time, i.e., does not depend on the size of the histogram. We used a trie, a search tree, to store the histogram. The index of the histogram is stored as key at the edges of the tree. One byte is used as quantization index of each input. The actual count of a histogram bin is saved in the leaf nodes of the tree. The access time of the trie only depends on the depth, i.e. the number of input streams. Figure 5 shows how the bins are saved in a trie for the three-dimensional case. The depicted histogram shows three non-zero bins `[1,5,5]=5`, `[1,5,9]=7` and `[4,5,5]=23`.

V. Experiments and Evaluation

We have conducted a number of experiments on the road, which showed that our approach is suitable for different scenarios of data collection. In the following, we will give an example, of how to apply our SPG model to a given problem. The data that we present as part of the evaluation is deliberately modified, so that confidentiality and proprietary rights are preserved.

The problem of qualifying intermittent and non-reproducible errors is inherent to the automotive industry. We have taken up cylinder misfires as an example for indeterministic events, that we want to investigate. Knocking and misfires, against the common perception, are still quite common for modern engines, as the combustion is usually most efficient when it happens close to the knocking limit [20].

Objective: Obtain a profile of the situation, where a specific error (i.e., misfire in our example) occurs. The profile needs to be of low volume on the one hand, but on the other hand informative enough to conduct leads to the root-cause of the error.

Rationale: A complete log of the vehicle's environment data is not sensible. Actually, even short time-series for every occurrence will produce large quantities of data for a high event-frequency, for many sources, and over a long time-span.

Approach: We use a multi-dimensional histogram as state-space, that represents a partial state of the vehicle. The non-zero states represent those in which an error situation has occurred. Ring buffers provide the input to every dimension of the state space. A ring buffer may be anywhere within the SPG, so that sources (e.g., speed) or pre-processed (e.g., average or maximum of speed within the last minute) streams can be used as input. For every dimension of the state-space, there are parameters for the minimal and maximal value and the number of quantization levels. One may, for example, only be interested in a rough quantization of the engine temperature (cold, warm, hot), but need a finer granularity for other dimensions.

Implementation: The engine control unit supplies a misfire-counter for every cylinder. We monitor the counters for changes and—upon an increment of a counter—we trigger an update of the state-space via an event/action pair.

Fig. 6. The event 'MISFIRE' is triggered, if misfire counter (MFC) of cylinder i was incremented.

Four consecutive operators are used per cylinder, to set off the event (Fig. 6). The SPG-arrangement is started by the source operator that supplies the misfire counter value, a delta, a filter and an event operator. Tuples periodically arrive at the delta operator, which outputs the difference between two following tuples to the filter. They are dropped, if their value is zero. Otherwise—this indicates the misfire—the tuple is forwarded to the event operator, which propagates the event "misfire" to the event handler. The event handler performs an action that releases the most recent data tuple from ring buffers via their control input, which are themselves the input to the multi-dimensional histogram store.

Interpretation: The evaluation of a multi-dimensional state-space reveals patterns, on *how* the vehicle was used when errors occurred. A clustering of only three states for a significant number of occurrences, as seen in Figure 5, points out a very specific cause of the problem.

Typically, we used a set of seven sensor sources for the state

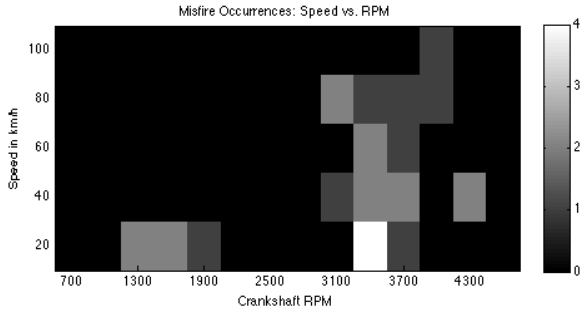


Fig. 7. 2D profile for speed and RPM of misfire occurrences: Every count (displayed as grey scale value) represents one cylinder misfire. Misfires seem to occur more frequently at around 3400 RPM.

space. In Figure 7, we show a plot of two dimensions (projected on this plane) of the error profile recorded for misfires. The profile shows a cluster of misfires around 3400 rpm for all speeds. The second cluster at low engine revolutions does not spread over all vehicle speeds, but concentrates on the low (0-30 km/h) speeds.

As subsequent question, one may ask *why* the misfires do occur mostly in the interval around 3400 rpm. An obvious guess for reasoning misfires is, that they occur at excessive changes of engine revolutions or load of the engine. To prove this guess right, one may add an additional action to the misfire event, which records a *time-series* from a buffered sensor-source (see Fig. 8 for a plot of the engine RPM). We can see, that before the misfire occurred, a rapid change of the engine speed happened. One may presume that the engine load also changed rapidly, during these thirty seconds before the misfire.

As far as a reduction of collected data is concerned, it may even be adequate not to record time-series, but only to record the maximal gradient for a recent time interval. The SPG-design easily allows to insert a delta operator with specified time-window (e.g., two seconds) and a maximum operator for the total buffer-length before the final log operator.

VI. Conclusions

The current automobile's electronic architecture is influenced by many different standards. There are ambitions within the automotive industry to standardize the diagnostic software interface in a way that abstract data sources such as "vehicle speed" are mapped to concrete sensors on the fly, which would ease data acquisition. The big advantage of using the vehicle's diagnostic infrastructure is that there is virtually no modification of the car necessary, so that the system is easy to install and completely reversible. Our system employs these benefits and combines them with advanced recording strategies.

The work shows that a given interface—OBD-II and diagnostic protocols—can be used to achieve a high-level goal by employing low-level tools and data. We show how a flexible and adaptable data recorder can be used in various vehicle-related contexts and for different purposes, by using one standardized interface and a small amount of processing power and

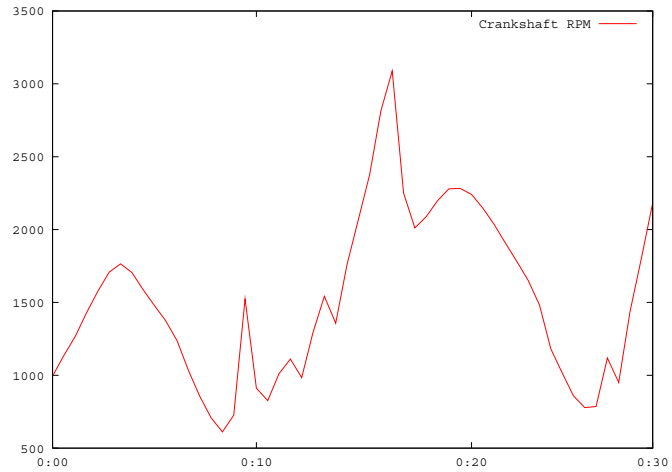


Fig. 8. Engine RPM, thirty seconds before a misfire occurs. The gradient changes abruptly.

memory. The situation dependent recording demonstrates a great improvement over existing systems and allows a selective reduction of data quantity while maintaining the adequate data quality. The engineer can start out with a generic configuration and subsequently add filter and aggregate operators, refine recording operations and actions on specific events.

We believe that this field of research will significantly gain attention as electronic automotive systems increasingly assist the classical mechanical domain. The electronic systems of a car provide valuable information about the vehicle's state to engineers, workshops and quality assurance—data only needs to be processed, preserved, and evaluated accordingly.

References

[1] J. Fröberg, K. Sandström, C. Norström, H. Hansson, J. Axelsson, and B. Villing, "Correlating bussines needs and network architectures in automotive applications - a comparative case study," in *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET)*. Aveiro, Portugal: IFAC, July 2003, pp. 219–228.

[2] H. Schweppe, "Optimization of an advanced automotive acquisition and aggregation system," Master's thesis, Technische Universität Berlin and DaimlerChrysler REDNA Inc., Palo Alto, July 2007.

[3] S. Warren, "In-vehicle data logging," *Embedded Linux Journal*, vol. 5, pp. 14–16, 18–19, Sep,Oct 2001.

[4] S. Ilic, J. Katupitiya, and M. Tordon, "In-vehicle data logging system for fatigue analysis of drive shaft," in *International Workshop on Robot Sensing, 2004. ROSE*, May 2004, pp. 30–34.

[5] V. Barabba, C. Huber, F. Cooke, N. Pudar, J. Smith, and M. Paich, "A multimethod approach for creating new business models: The General Motors OnStar project," *Interfaces*, vol. 32, no. 1, pp. 20–34, 2002.

[6] K. Grimm, "Software technology in an automotive company: major challenges," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 498–503.

[7] L. Golab and M. T. Özsu, "Issues in data stream management." *SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003.

[8] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik, "Retrospective on Aurora." *VLDB J.*, vol. 13, no. 4, pp. 370–383, 2004.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the Borealis stream processing engine." in *CIDR*, 2005, pp. 277–289.

[10] N. Tatbul and S. B. Zdonik, "Window-aware load shedding for aggregation queries over data streams." in *VLDB*, U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, Eds. ACM, 2006, pp. 799–810.

[11] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford stream data manager." *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003.

[12] S. Madden, R. Szewczyk, M. J. Franklin, and D. E. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *WMCSA*. IEEE Computer Society, 2002, pp. 49–58.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world." in *CIDR*, 2003.

[14] J. Considine, F. Li, G. Kollios, and J. W. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE*. IEEE Computer Society, 2004, pp. 449–460.

[15] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy, "Vedas: A mobile and distributed data stream mining system for real-time vehicle monitoring." in *SDM*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004.

[16] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Monitoring streams - a new class of data management applications." in *VLDB*. Morgan Kaufmann, 2002, pp. 215–226.

[17] M. Broy and G. Ştefănescu, "The algebra of stream processing functions," *Theoretical Computer Science*, vol. 258, no. 1–2, pp. 99–129, 2001.

[18] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: A new model and architecture for data stream management." *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[19] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation." in *SIGMOD Conference*, 2006, pp. 623–634.

[20] U. Kiencke and L. Nielsen, *Automotive Control Systems, for Engine, Driveline, and Vehicle*, 2nd ed. Springer Verlag, 2005.