Third Conference on Software Engineering and Information Management (SEIM-2018)  
(full papers) 

Saint Petersburg, April 14, 2018 

Yurii Litvinov, Marat Akhin, Boris Novikov, Vladimir Itsykson (editors)
This volume contains eight selected papers originally presented at the Third Conference on Software Engineering and Information Management (SEIM-2018), which was held in Saint Petersburg, Russia, on April 14, 2018. These papers were selected in thorough single-blind reviewing process.

ISBN 000-0-0000000-0-0

Originally published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073, volume 2135)

Copyright © 2018 for the individual papers by the papers’ authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.
# Table of Contents

Message from the Editors .......................... 4

SEIM 2018 Organization .......................... 5

Electromyography signals processing for gait phase recognition

*Georgiy Zhemelev* .................................... 6

Design and testing of LXC-based virtualization system for resource-constrained MIPS devices

*Maxim Menshchikov* .................................. 13

An optimistic approach to handle out-of-order events within analytical stream processing

*Submission: Igor Kuralenok, Nikita Marshalkin, Artem Trofimov, Boris Novikov* .................. 22

Keyword Extraction from Single Russian Document

*Mikhail Sandul, Elena Mikhailova* .................. 30

Data-based code synthesis in IntelliJ IDEA

*Vladislav Tankov, Timofey Bryksin* ................ 37

SMT-Based Analysis of Constraints on .NET Types

*Aleksandr Misonizhnik, Dmitry Mordvinov* ........ 44

On-the-Fly Filtering of Aggregation Results in Column-Stores

*Anastasia Tuchina, Valentin Grigorev, George Chernishev* ............................................ 53

Development of software package for data analysis of acoustic emission control

*Victoria Belousova, Anastasia Grigorieva* ........ 61
Message from the Editors

The Third Conference on Software Engineering and Information Management (SEIM-2018) opens its doors to young researchers and practitioners in different areas of computer science and software engineering, providing an opportunity to present their research, discuss state-of-the-art technology and engage in useful networking. As before, we consider SEIM to mainly focus on researchers who are just starting out their scientific careers, and hope to ease their introduction to the conference process. On the other hand, SEIM might also be of interest to more experienced researchers, who are aimed at sharing their research with a wider scientific community. The conference welcomes submissions on a wide range of topics, including but not limited to:

- Algorithms and data structures
- Big data
- Cloud systems
- Coding theory
- Compilers
- Crowdsourcing
- Data storage and processing
- Development management
- Digital signal processing
- Distributed systems
- E-commerce / e-government
- Empirical software engineering
- High-performance computing
- Information retrieval
- Information security
- Intelligent data analysis
- Internet of Things
- Machine learning
- Mobile systems
- Modelling
- Natural language processing
- Networks and telecommunications
- (Non-)relational databases
- Operating systems
- Programming languages
- Recommendation systems
- Robotics
- Semantic web
- Social networks
- Software analysis
- Software testing
- Software verification
- Software virtualization
- Software-defined networks
- Theoretical computer science
- Visual languages

This year we received 43 papers, each reviewed by at least 3 members of the Program Committee, of which 8 were selected for publication in CEUR-WS.org, 7 — for indexing in RSCI, and 1 was accepted as talk-only to allow the authors to receive valuable feedback about their research. We would like to thank the members of our Program Committee for their continuous work and contribution to the success of our conference.

These proceedings include the SEIM-2018 papers, which were selected by the Program Committee for publication in CEUR-WS.org. These papers passed not only the original review procedure, but also an additional round of post-review with the conference feedback. We thank the authors for their submissions to SEIM 2018 and hope to see them in the future.

Furthermore, we would also like to thank Tatiana Mironova, Sergey Zherevchuk and Svyatoslav Mikhailov for their great help in organizing the conference, Computer Science Center for hosting the event, and JetBrains Research for their overall support. The additional information about the SEIM conference series can be found on the conference website at:

http://2018.seim-conf.org/

Yuri Litvinov, Marat Akhin, Boris Novikov, Vladimir Itsykson
Editors
SEIM 2018 Organization

The conference was organized jointly with Computer Science Center and supported by JetBrains Research.

Steering Committee

Dmitry Bulychev / St. Petersburg State University
Vladimir Itsykson / St. Petersburg Polytechnic University
Andrey Ivanov / JetBrains
Iakov Kirilenko / St. Petersburg State University
Kirill Krinkin / St. Petersburg Electrotechnical University
Boris Novikov / St. Petersburg State University

Program Committee Chairs

Yurii Litvinov / St. Petersburg State University
Marat Akhin / St. Petersburg Polytechnic University

Program Committee

Mikhail Belyaev / St. Petersburg Polytechnic University
Daniil Berezun / JetBrains, St. Petersburg State University
Natalia Bogach / St. Petersburg Polytechnic University
Timofey Bryksin / St. Petersburg State University
Dmitry Bugaichenko / St. Petersburg State University
Dmitry Boulytchev / St. Petersburg State University
Ekaterina Verbistskaya / St. Petersburg State University
Nikolay Vyakhirev / St. Petersburg State University
Nikolay Glukhikh / JetBrains
Natalia Grafeeva / St. Petersburg State University
Dmitry Grigorev / St. Petersburg State University
Semyon Grigorev / St. Petersburg State University
Victoria Erofeeva / St. Petersburg State University
Yuriy Ivanskiy / Institute of Problems of ME RAS
Vladimir Itsykson / St. Petersburg Polytechnic University
Alekandr Kapitonov / ITMO University
Iakov Kirilenko / St. Petersburg State University
Dmitry Kosarev / St. Petersburg State University
Kirill Krinkin / Open Source and Linux Lab
Svetlana Lazareva / Raidix
Dmitry Luciv / St. Petersburg State University
Igor Malyshev / St. Petersburg Polytechnic University
Elena Mikhailova / St. Petersburg State University
Mikhail Moiseev / Intel Corporation
Dmitry Mordvinov / St. Petersburg State University
Vyacheslav Nesterov / Dell EMC
Boris Novikov / St. Petersburg State University
Gennady Pekhimenko / Microsoft Research
Oleg Pless / Oracle
Anton Podkopaev / St. Petersburg State University
Sergey Salishev / St. Petersburg State University
Elena Sivogolovko / Arcadia
Kirill Smirnov / St. Petersburg State University
Darja Solodovnikova / University of Latvia
Sergey Stupnikov / Institute of Informatics Problems RAS
Maksim Tkatchenko / Singapore Management University
Peter Trifonov / St. Petersburg Polytechnic University
Arthur Khanov / St. Petersburg State University
Dmitry Tsitelov / Devexperts
Mikhail Zymbler / South Ural State University
George Chernishev / St. Petersburg State University
Anna Yarygina / St. Petersburg State University
Abstract—Gait phase recognition systems are widely used in medicine to control devices aimed at restoration of patients’ motor functions and have an increasing interest in scientific society. Use of electromyography as a source of information for such systems gives considerable advantages in comparison with other data sources at the cost of complex signal processing needed. In this paper, the author outlines these advantages and suggests a method that uses discrete wavelet transform to retrieve muscle activity shape and a novel double-threshold detector to find the regions of activity. Then a robust statistical treatment is performed following a dimensionality reduction. As a result, a set of classification objects is retrieved that are suitable for further use in various clustering and classification techniques. The introduced method was tested in the Movement Physiology Laboratory of I. P. Pavlov Institute of Physiology, Russian Academy of Sciences and proved its applicability on real electromyography data.

I. INTRODUCTION

Gait phase can be used as an integral time characteristic of complex body and limb movements during the walk. Therefore, in medicine gait phase information can be used to control exoskeletons [1]–[4] or provide stimuli for spinal cord or muscles so as to restore patient’s motor functions [3], [5]–[7]. Nowadays, there are various techniques for gait phase recognition that differ mainly in the data source they are based on. This paper addresses the use of electromyography (EMG) signals for this purpose in the context of developing a gait phase recognition software.

The author suggests a method that uses discrete wavelet transform to retrieve muscle activity shape and a novel double-threshold detector tolerable to amplitude hopping to find the regions of activity. Then a robust statistical treatment based on interquartile range calculation is performed following a dimensionality reduction. This results in a set of classification objects that are suitable for further use in various clustering and classification techniques. In his bachelor thesis [8], the author used end-to-end approach which included fuzzy C-means clustering and classification using adaptive neuro-fuzzy inference system (ANFIS). Although fuzzy techniques are a reasonable choice for EMG signals since their nonstationary nature, the method suggested in this paper does not set any limitations that would impede other classification techniques that are typically used for EMG-related tasks (support vector machines (SVM), linear discriminant analysis (LDA), artificial neural networks (ANN) etc) [9], [10].

A. EMG as Data Source for Gait Phase Recognition

The process of gait phase recognition is built around sensors that are used to retrieve the data during walk. According to the overview [11] by Muro-de-la-Herran et al. sensors for gait analysis can be divided into wearable and non-wearable groups of devices. The latter “require the use of controlled research facilities where the sensors are located and capture data on the gait while the subject walks on a clearly marked walkway” [11]. In contrast, wearable sensors make it possible to capture gait information during the person’s everyday activities. Thus the systems that are based on wearable sensors can be used outside the laboratory which is the crucial advantage of the wearable approach.

When capturing the data during walk, different physical quantities can be measured. Based on such quantities one can divide wearable sensors into lots of categories: accelerometers, gyroscopic sensors, magnetometers, force sensors, extensometers, goniometers, EMG sensors etc. [1], [11].

This paper addresses the use of electromyography to retrieve the information about gait phase. The benefits provided by this approach include the following:

- the least response time to the start of muscle activity [12] since the electrical activity is measured by EMG sensors directly whereas other sensors measure the physical quantities that change as a consequence of movements caused by that activity (according to Wentik et al. the use of EMG allows to predict movements “up to 138ms in advance in comparison to inertial sensors”);
- the ability to detect movement intention [13], [14] as electrical activity in muscles exists even if its power is insufficient to initiate the movement (e.g. as a result of nervous connection injury);
- the possibility to use EMG sensors for amputees because, in most cases, EMG signals of large muscles can be measured from patient’s stump [12].

The main disadvantage of using EMG is the need for high-quality instrumentation as EMG signals “are invariably very small (in the order of 0.00001 to 0.005 of a Volt)” [11] and thus are strongly influenced by noise.

Considering all the pros and cons, electromyography signals appear to be a reliable and rich data source for gait phase recognition systems. To prove this concept the software pack-
age was developed and tested [8] on real electromyography data showing promising results which are discussed later in this paper.

Some researchers have already used EMG for solving gait phase recognition [15], [16], movement pattern classification [17], [18] or locomotion mode identification problems [9], [19]. Typical techniques incorporate extraction of such aggregate features as mean absolute value, root mean square, number of zero crossings etc in a moving window and forming a feature vector via combination of these features extracted from several EMG-channels. Some authors [17] have successfully used wavelet transform to extract numerical features related to wavelet-coefficients. The resulted classification objects were then used in conjunction with ANN, SVM, LDA and other classification techniques.

EMG processing technique proposed by the author produces essentially different classification objects. Not only a combination of some aggregate values calculated from EMG samples are they but also an image in the feature space that describes EMG fragments containing muscle activity by their shape in the time-domain. The basis of this approach is stated in the next sections of the paper.

B. Paper Structure

The paper structure is as follows:

- Gait phase representation.
- Experimental setup and raw data analysis.
- Preparation of classification objects:
  1) retrieval of muscle activity shape;
  2) muscle activity detection;
  3) statistical analysis;
  4) dimensionality reduction.

- Results and discussion.

II. MATERIALS AND METHODS

A. Gait Phase Representation

Within the limits of a gait cycle gait phase can be represented as a continuous monotonically increasing function of time. It is important to fix the transition from stance to swing hence the representation showed on Fig. 1 was chosen: gait phase is measured in conventional units from 0 to 200 where range 0-99 applies to the stance phase and the range of 100-200 applies to the swing phase.

Such a representation is very useful to detect transitions between steps and between stance and swing phases within a step, as well as it can be easily constructed and efficiently processed since its linear nature.

B. Experimental Setup and Raw Data Analysis

Electromyography data used in the research was acquired in the Movement Physiology Laboratory of I. P. Pavlov Institute of Physiology, Russian Academy of Sciences in the course of acute experiments on healthy and decerebrate cats. Their locomotion was aroused by epidural stimulation of the spinal cord (dorsal surface) with the optimum frequency (5-10 Hz) [20] for stepping pattern.
C. Preparation of Classification Objects

According the suggested concept, muscle activity batches should be divided into some clusters. Each cluster of batches is supposed to correspond to some gait phase value that is derived from EMG signal during the learning stage of a gait phase recognition system.

Therefore, in order to be used as an inputs for a classifier, i.e. classification objects, muscle activity batches must undergo some preparation. Steps of this preparation are described below.

1) Retrieval of Muscle Activity Shape: The first step of the suggested gait phase recognition method consists in retrieval of muscle activity shape meaning some raw EMG signal processing that will facilitate the following steps of muscle activity detection and dimensionality reduction.

This paper suggests using discrete wavelet transform (DWT) [22] to expose muscle activity shape. At the beginning of the process the EMG signal is rectified. Then the exposure is done in the following way:

1) perform DWT to a high level of decomposition so as to extract large-scale components of the signal;
2) discard wavelet-coefficients of the lower levels;
3) reconstruct the signal using the remaining wavelet-coefficients.

As a result, there will be constructed a smooth curve – an envelope – that correctly [23] describes the shape of a batch preserving the most powerful peaks while not containing the high-frequency spectrum region. In this research the best results were achieved using the 3rd Coiflet (coif3) as the mother wavelet and decomposition was performed at level 7. The low-frequency nature of the envelope makes it possible to perform decimation on the subsequent steps of the gait phase recognition method.

In author’s previous work [8] the comparison was performed between DWT and other methods that can be used to retrieve muscle activity shape. As a result of this comparison, the approach based on DWT was considered as the most flexible and accurate.
2) Muscle Activity Detection: In order to perform clustering and classification of the muscle activity batches it is essential to detect these batches in the EMG signal. Constructing an envelope on the previous step makes detection much simpler than if a raw EMG signal was used as an input for a detector.

According to Reaz et al. one should use double-threshold methods to detect motor-related events in EMG signals [23]. Single-threshold approach was shown to produce generally unsatisfactory results [24]. Moreover, using a double-threshold method one “can tune the detector according to different optimal criteria, thus, adapting its performances to the characteristics of each specific signal and application” [23]. This tuning ability is especially useful as EMG channels have decent differences in muscle activity shape and power.

In this research a novel implementation of a double-threshold detector is introduced: the author suggests to define the second threshold in the time domain (instead of amplitude) and make it use two possibly different values - one to detect beginning of a batch, and another one to detect the ending.

The use of a time-domain threshold as described above makes the detector much more tolerable to amplitude hopping in the middle of a batch and hence its low probability of false detections. A flow chart of the detecting algorithm is shown at Fig. 4.

Fig. 5 illustrates a comparative test of three detectors performed on a sample of raw EMG signal: a single-threshold (with \( th = 0.1 \) V), a simple double-threshold (\( th_1 = 0.1 \) V, \( th_2 = 0.2 \) V) and the one suggested by the author (\( th_1 = 0.15 \) V, \( T_{on} = 1 \) ms, \( T_{off} = 100 \) ms). The latter has detected all activity regions without false positives while others have given unsatisfactory results: instead of five continuous regions they have produced hundreds of narrow intervals. Despite the fact that, being applied to an envelope instead of raw EMG, the simple detectors are likely to produce acceptable results, the advantage of the novel detector shown in the unfruitful conditions makes it much more suitable for gait phase recognition systems based on EMG signals.

3) Statistical Analysis: Detected batches of muscle activity need to be adjusted to the single length (in samples). This requirement follows from the fact that all classification objects must be described as a feature vectors that belong to the feature space of a fixed dimensionality. This is not required for all classification algorithms but facilitates the application of the typically used ones (SVM, LDA, ANN, ANFIS etc) [9], [10].

After the previous step, there is a selection of batches that form a statistical sample that needs to be conditioned before performing adjustment to the single length. The conditioning implies omitting outliers that are usually incorrectly detected if present in the sample, differ greatly from the majority of batches in their length even considering natural variation in
the duration of step and its phases \[8\].

A common method of omitting outliers is estimation of the standard deviation across the sample and using the three sigma rule to set the acceptable parameter bounds (length of a batch). However, the author suggests using interquartile range (IQR) as a robust alternative to the three sigma rule because the latter rule is based on the assumption that the sample is distributed normally. In contrast, using IQR makes it possible to find and discard outliers even in case the sample does not comply with the normal law.

When the IQR is computed, the acceptable value bounds are defined as:

\[
(Q_1 - 1.5 \times IQR, Q_3 + 1.5 \times IQR),
\]

where \(Q_1\) and \(Q_3\) are the estimates of the 1st and the 3rd quartiles respectively. This rule is illustrated at the Fig. 6.

After omitting the outliers, the single length of batches is chosen as the maximum across the remaining elements in the sample. Then the adjustment of all remaining batches to that length is performed.

### D. Dimensionality Reduction

The final step in the process of preparation of classification objects is their decimation in order to reduce dimensionality of the feature space. The decimation is possible because batches are represented by the envelope that does not contain high-frequency spectrum region.

Raw data used in this research was captured with sampling frequency of 1 kHz and the batch length was 500 samples on average. An envelope constructed after DWT in II-C1 had only low-frequency components of 10-20 Hz so the decimation factor was chosen to be 20. As a result, a feature space dimensionality was reduced to the value of 25 that is acceptable for further use in clustering and classification.

### III. Results and Discussion

Gait phase recognition systems which are based on electromyography require complex processing of EMG signals. This paper has covered steps of such processing that result in a set of classification objects derived from the EMG and suitable for the following use in clustering and classification.

Fig. 7 shows an example of clusters that were found in the EMG of the gluteus muscle. One can see the shapes of this muscle activity fragments (batches) that were retrieved via the method suggested in II-C1 before decimation. As a consequence of natural nonstationarity, these batches keep substantial variations in shape even inside a cluster so there is a need for wise decisions on classification methods to use \[25\].

In \[8\] a gait phase recognition software based on EMG was developed utilizing adaptive neuro-fuzzy inference system (ANFIS) as classifier. The results of classification, matching phase value and its derivative, were used to construct an approximate phase line in accordance with an ad-hoc algorithm. Fig. 8 shows a comparison between real (blue) and approximate (red) phase lines on a time interval spanning three consecutive steps. Data from 6 EMG-channels (coinciding with the muscles listed in II-B) and a sample consisting of 35 locomotor cycles were used there.

The accuracy of the developed system was estimated using the normalized integral criterion \((2)\) and a set of qualitative measures \((3)-(5)\) of time misalignment:

\[
\varepsilon = \frac{\sum_{i=1}^{N} (P_{\text{real}}(i) - P_{\text{approx}}(i))^2}{\sum_{i=1}^{N} P_{\text{real}}^2(i)},
\]

where \(P_{\text{real}}\) – samples of the real phase line, \(P_{\text{approx}}\) – samples of the approximate phase line, \(N\) – total number of samples;
... was equal to mean duration of a gait cycle (for the examined experimental subject) which was equal to $a = 1000 \pm 40$ ms (with the confidence probability of $q = 0.90$). The results obtained by the author in [8] are presented in Table I.

One can see that all mean estimates by their absolute value are not greater than 60 ms which is 4–8% of the mean gait cycle duration. The integral criterion value that is equal to 11.6% also proves good approximation of the gait phase.

The above-mentioned results were achieved to a considerable degree with the help of the EMG processing method suggested in this paper. The constructed classification objects enabled the system to train on and analyse real EMG data which resulted in successful gait phase recognition [8].

Since assessment of the proposed technique with respect to the state of the art can be of interest, the author has made a comparison between the results achieved in his own work [8] and that presented in four other papers [9], [15]–[17] considering mean classification error as the quality estimate that can be compared with the normalized integral criterion (2). In general, one cannot directly compare results achieved by different researchers since they use dissimilar gait phase representations and sometimes solve problems which are closely related to gait phase recognition but are not exactly the same. However, use of classification makes it possible to compare EMG processing techniques indirectly via comparison of mean classification errors. The results of the comparison are presented in Table II.

In sum, the suggested method gives accuracy at the level of other state-of-the-art techniques but it has some crucial advantages over them, viz. automatic moving window size choice (as a result of statistical analysis), flexible and reliable detector, robustness to artefacts in EMG and potential to utilize classification techniques used in image recognition since the structure of feature vectors produced by the suggested method.

### IV. Conclusion

As a result of the research covered by this paper, an EMG signal processing method was developed that facilitates the use of electromyography for gait phase recognition systems. The suggested method makes it possible to retrieve classification objects from raw EMG data and can be used during learning stage of the gait phase recognition system as well as the stage of its real-time functioning.

![Fig. 8. Gait phase line construction in the system based on the EMG processing method suggested in this paper. Circles point out gait phase values calculated by the classifier and arrows represent values of phase derivatives at those points. These values were used to construct the approximate phase line (red) using an ad-hoc algorithm. The blue line representing the real gait phase is quite close to the approximate proving the functionality of the system and applicability of the suggested method.](image_url)
TABLE II
COMPARISON WITH OTHER GAIT RECOGNITION SYSTEMS

<table>
<thead>
<tr>
<th>Authors</th>
<th>Features extracted from EMG</th>
<th>Classification method used</th>
<th>Number of used EMG channels</th>
<th>Mean classification error, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huang et al.</td>
<td>MAV, ZC, WL, SSC, RMS, AR and other</td>
<td>Linear discriminant analysis</td>
<td>6*</td>
<td>14.0</td>
</tr>
<tr>
<td>Li et al.</td>
<td>MAV, Variance</td>
<td>Support vector machine</td>
<td>4</td>
<td>11.2</td>
</tr>
<tr>
<td>Meng et al.</td>
<td>MAV, WL†</td>
<td>Hidden Markov model</td>
<td>8</td>
<td>8.5</td>
</tr>
<tr>
<td>Yu et al.</td>
<td>DWT-based aggregates</td>
<td>Artificial neural network</td>
<td>9</td>
<td>10.0</td>
</tr>
<tr>
<td>G. Zhemelov</td>
<td>DWT-based activity shape</td>
<td>Adaptive neuro-fuzzy inference system</td>
<td>6</td>
<td>ε = 11.6</td>
</tr>
</tbody>
</table>

The abbreviations are defined as follows: MAV – mean absolute value, ZC – zero crossings, WL – waveform length, SSC – slope sign changes, RMS – root mean square, AR – autoregression coefficients, DWT – discrete wavelet transform. In the research of Huang et al. experiments with 6, 8, 10 and 16 EMG channels were conducted. Here the result for 6 channels is shown to match the number used by the author. † Meng et al. studied different combinations of features, here presented the one that led to the best results.

ACKNOWLEDGMENT

Raw experimental data used in the research was acquired during 2012 in the Movement Physiology Laboratory, I. P. Pavlov Institute of Physiology, Russian Academy of Sciences. The author would like to thank P. Musienko, O. Gorskiy and other members of the laboratory staff that contributed to the research covered in this paper. The author also expresses gratitude to N. Bogach, Peter the Great Saint-Petersburg Polytechnic University for scientific guidance and her advice given during writing of this paper.

REFERENCES

Design and testing of LXC-based virtualization system for resource-constrained MIPS devices

Maxim Menshchikov
Department of Applied Mathematics and Control Processes
Saint Petersburg State University
Saint Petersburg, Russia
maximmenschchikov@gmail.com

Abstract—Container-based virtualization became a standard de-facto for many environments, but it’s applicability to resource-constrained systems is not as trivial question as it seems to be at first glance. Despite many projects aimed at bringing lightweight containers, such as Linux Containers (LXC), Docker, Kubernetes, etc, there are numerous problems in real-world development that might dramatically increase the impact on resources or render device unstable. In the paper we focus on Linux-based resource-constrained systems, studying the impact of LXC on MIPS CPU. Problems observed during the implementation of our system were studied and solutions for some of them, specifically slab overflow, provided. Our interpretation of obtained CPU, file system and other measures implies a negligible difference in comparison between host and isolated environment with only TCP performance reduced by up to 8-9%.

Index Terms—virtualization, containers, lxc, linux, resource-constrained devices, embedded devices, internet of things, mips.

I. INTRODUCTION

Embedded devices are used worldwide. They include routers, IoT/Smart Home devices, smartphones, players, set-top boxes etc. The way to add functionality to such devices hasn’t changed over years: there is always a possibility to write a new module and build a firmware with it. It is assumed that such module is written by firmware developer and therefore a right place is taken in internal device infrastructure: it doesn’t affect stability, performance, and security. The latter statement can only be taken with a pinch of salt: how can developers guarantee that there are no unforeseen outcomes of such functionality?

Mentioned product release schema is quite common, but what if software company decides to delegate some functionality to another company? Trusted partners can make a package nearly independent upon the generic firmware. Printing server (such as CUPS1), File server (e.g. Samba2) serve as good examples of almost independent components. However, vulnerabilities such as SambaCry3 (a Samba vulnerability discovered in 2017) leave device open to new threats just due to external software.

Virtualization helps avoid these risks. Resource-constrained devices usually have too little raw performance to run a full-fledged virtualization and containers have a remarkable advantage over traditional virtualization methods. Container word relates to the environment made of multiple technologies inside the operating system kernel: PID namespaces [1], user namespaces [2], network namespaces [3], mount namespaces, etc. It is an isolated environment in which applications that are running inside are locked from accessing external resources unless explicitly or implicitly allowed doing so. Containers are running on top of main system kernel heavily exploiting isolation provided by it, which implies better performance and lower resource consumption.

However, even lightweight virtualization provided by kernel has serious limitations and consequences which must be addressed. In this paper we try to build a package virtualization supporting system and study performance, stability, security, networking and other effects of LXC (Linux Containers) virtualization on a specific MIPS-based embedded device and describe found problems with solutions to them.

A. Existing solutions

The first and the most obvious solution is Docker4. It is performing very well on desktop-class operating systems and more widespread embedded devices. Main disadvantages are in its binary size, the tendency to use Go runtime and compilers (that raises big questions about used toolchains, binary size, “maturity” of compilers) and its official incompatibility with MIPS processors.

Project balena5 takes Moby project (the foundation for Docker) and tries to improve its weakest points. It features smaller single binary, multi-architecture support and more effective use of network bandwidth. The latter is quite exceptional and interesting, but suggested binary size is still quite big. MIPS architecture is also not supported officially.

If full-fledged virtualization is taken into account, then it is reasonable to check officially recommended [4] hypervisors for MIPS, such as prplHypervisor [5] (ex. Hellfire), SELTECH Fexerox [6], OmniShield [7]. However, these solutions tend to require more RAM, disk space and computational power for running isolated kernel alongside with operating system files.

1https://www.cups.org
2https://www.samba.org
4https://www.docker.com
5https://github.com/resin-os/balena
B. Motivation

Existing solutions are more focused on more resource-rich desktop-class platforms. Our main intention is a creation of a system maintaining low overhead for boards used for more specific purposes and therefore lacking resources. We would like to run third-party packages of varying complexity while having those applications integrate to the device platform in a natural way. At the same time, the system we build is quite minimalistic and straight-forward, which enables quicker development of features.

While we test only MIPS devices, our expectation is that overhead measurement result will be quite generic among recent CPUs of various architectures. Still it requires an additional verification.

C. Positioning

Our system is positioned at a higher level than LXC itself since it deals with its setup rather than internal details. It shares the level with Docker/balena, which are far more user-friendly applications, but unlike containerd-based applications, we don't necessarily set up namespaces by our own means besides quotas and access restriction details. Virtualization by hardware or operating system level hypervisors is definitely on a lower level than our system.

II. System Profile

The key components of our system are:

1) CPU: MIPS architecture 600 MHz dual core (in CPU parallel test similar single core CPU was used to ensure no core load differences).
2) Kernel: Non-preemptive SMP Linux (exact version can’t be disclosed).
3) RAM: 256MB.
4) ROM: 128MB.
5) 2.4GHz 802.11bg [8] and 5GHz 802.11ac [9] Wi-Fi antennas, traffic offloading hardware, cryptography acceleration, USB ports.

III. Classification of Points of Interest

1) **Kernel.** Operating system kernel is a key component for OS-level virtualization provider. We checked how it may generally prevent developers from developing embedded solutions running containers.
2) **Latency.** A device has to remain responsive no matter how many applications are running and how much do they exploit the CPU. Third-party applications mustn’t spoil the main functionality.
3) **Raw performance.** Containers shouldn’t suffer from decreased performance to extent set by quotas.
4) **Network performance.** Download and upload speeds shouldn’t reduce significantly compared to host environment.
5) **RAM usage.** Main device’s activity shouldn’t be spoiled dramatically by the memory-consuming applications.

6) **File system.** A device should handle a reasonable number of containers with bound packages. There must be a possibility to avoid redundant copies of container files.
7) **Fault tolerance.** A device must be tolerant to boot failures and must be able to survive major panic situations.
8) **Security problems (throughout the paper).** Containers shouldn’t affect host’s security.
9) **Board temperature.** Containers shouldn’t have any influence on board temperature, nor they should lead to overheating. We consider it dependent upon processor load and power consumption. Since there is no intention to perform any serious computations on our CPU, board temperature wasn’t a target of our research.
10) **Power consumption.** Use of applications in containers shouldn’t lead to increased power consumption compared to runs within host namespace.
11) **Container activation time** This criterion is about time to start the container, which might be crucial in case of deeply distributed real-time computations. Our use case didn’t imply any strict requirement for it, so it wasn’t explored much. It will be indeed mentioned in file system overview.

In the next section we present the solutions on these aspects, show observed issues and suggest solutions.

IV. Aspects and solutions

A. Kernel

1) **Embedded devices often can’t do their job without hardware support:** Operating system kernel has the largest influence on performance, stability, memory usage and other aspects. The main problem with embedded devices is in their dependency upon hardware supplier. Many manufacturers provide their own kernel for System-on-a-Chip (SoC) or some special hardware. With resource-constrained devices the situation is even more serious: often the device has no technical possibility to do its job without the help of hardware. In our case networking and cryptography accelerators required deep changes throughout the kernel to support shorter packet flow paths altering networking and netfilter cores very heavily. Justifying the need, CPU alone would have provided much less network bandwidth. In our experiments, 5GHz Wi-Fi antenna supporting IEEE802.11ac standard [9] was generally providing gigabit speeds, but could only demonstrate 40 mbit/s without overflowing the slab caches (we’ll add on this problem later in RAM section). Summing up, resource-constrained embedded devices not only depend on the kernel for performance and correct operability but also do not have any workarounds if driver functionality does not work as expected.

2) **Embedded devices have kernel version lag due to hardware supply:** Generally, there is nothing to worry about regarding virtualization if patched kernel is based on recent enough kernel and tested properly, however, in our case the kernel was many versions behind the mainline: it was using not the latest LTS kernel, but LTS with far enough End Of Life time without any possibility to upgrade. Consequently,
any desire to use container functionality was hitting at rough
implementation in the kernel. To be specific, this version
didn’t have all required bits of user namespace support, there-
fore the UID/GID mapping was largely unsupported, mak-
ing containers “privileged”. Such containers have ‘root’
user identical to the ‘root’ outside the container, and while it
had both advantages and disadvantages, it involved changing
the policy for packages to “trust only in specific cases”. Of
course, making a package out of Samba and other less-than-
trusted software was no longer an option until the update
from a hardware supplier. Migrating to the latest LTS was
not an option as well due to development time constraints.
We must also point out that the similar problem is common
with Android smartphones [10].

B. Raw performance

Raw performance is worth a special check on resource-
constrained embedded devices, a significant number of which
stick to low-end CPUs. The wide range of provided services
might require a reasonable computational power: VPN services
using packet encryption and decryption without hardware
cryptography acceleration, HTTPS servers (the most popular
use case in our opinion), and quite a few other possible needs.

1) CPU quotas: CPU limits set up using cgroups, allowing
containers to consume up to 50% of CPU, yet if responses
from a main program were getting too rare, software watchdog
was proactively decreasing the CPU share to some lower value
until reasonable response time was achieved. This scheme has
proven to be working well, however, a number of packages
required almost none computational power, so the scheme
wasn’t widely adopted.

Our formula (1) is mathematically trivial. Consider \( R_n \)
a normal response rate without additional load and \( R_c \) to be a
current rate (both in responses per second). The upper-thres-
hold \( T_u \) for containers CPU share was 0.5 (which corresponds
to 50% of CPU time and to \( I_u \) response interval), the lower
threshold \( T_l \) was 0.05 (5% of CPU time).

\[
s = \min(T_l + \frac{R_c}{R_n} \cdot (T_u - T_l), T_u)
\]

Without \( \min \), \( s \) can potentially grow more than \( T_u \), which
never happened in our experience, but in testing 50% was
always a reasonable upper limit to keep main device’s activity
running well.

2) Benchmarking methodology: \texttt{nbench} (the BYTE Maga-
azine’s BYTEmark) was used for benchmarking. When search-
ing for properly supported utility, other utilities such as
\texttt{sysbench} were tested, but most of them were problematic
to compile on MIPS due to extensive use of assembler inlines.
We consider \texttt{nbench} utility a reasonable choice because of its
good cross compilation ability and its focus on single-threaded
raw performance tests which fit our use case well.

CPU limits were disabled and the \texttt{nbench} executed 100
times in a row. The device was rebooted between tests.

Tests of CPU performance when running multiple bench-
mark processes were performed, but our prior investigation had
shown that \texttt{nbench} test isn’t much influenced by the number
of processes (this fact is indeed good for the sanity of the
first test). It was decided to measure time to complete the test.
Two or more processes were run in a row, an average time was
recorded. To ensure no core load differences, another single
core board with similar characteristics was used. The same
software package was flashed to it.

C. Latency

Latency is the “interval between stimulation and response”.
When evaluating operating system latency, we consider latency

Scheduler latency is a time interval taken by an operating
system before performing scheduling. Scheduler’s response
delay is based on two main parameters: minimal granularity
and latency. Minimal granularity is a minimal time given to
a process to execute. Latency is a period in which all tasks
should run at least once. Consider \( T \) a number of executable
tasks, \( L \) is a defined latency, \( G \) is a minimal granularity.
Therefore scheduler period is based on the following formula.

\[
Period = \begin{cases} L & \text{if } T < \frac{L}{G} \\ T \cdot G & \text{if } T \geq \frac{L}{G} \end{cases}
\]

However, running non-preemptive kernel adds some com-
plexities to the scheduler. The kernel cannot interrupt the pro-
cess executing a kernel call until the return to user mode [11].
Therefore the absence of preemption brings additional security
consideration: there must not be any serious way to interact
with kernel besides regular kernel API. For example, if usage
of some device driver can lead to infinite loops or long
computations, such driver should not be allowed. \texttt{libusb} was
determined to be causing a loop when using DWC2 (Design-
Ware USB2 Core driver) driver\(^6\). The latter was actually fixed
in the upstream kernel.

Also supplied kernel might have binaries precompiled for a
preemptive kernel, but even if provided modules are recompil-
able, they still might have interlocking errors like missing spin
lock or dead lock. They might render the kernel unstable in
preemptive mode. So we had to stick with non-preemptive
mode and provided kernel. Instead, \texttt{libseccomp}\(^7\) was
used in order to close unneeded kernel API and \texttt{AppArmor} [12]
helped restrict access to all devices except really needed ones.

Interrupt latency is constructed of the time elapsed since the
interrupt is first fired and interrupt is started servicing. The
main contributors to this latency type are drivers disabling
interrupts for a long time. It means containers generally
contribute to interrupt latency in exactly the same way as the
host does, mostly by initiating driver activity via networking,
peripheral or any other kind of access. We haven’t compared
the difference between containers and host because there is
almost no differentiation between namespaces on driver level.

\(^6\)DWC2 source code: https://git.kernel.org/pub/scm/linux/kernel/git/
torvalds/linux.git/tree/drivers/usb/dwc2

\(^7\)https://github.com/seccomp
D. Networking

1) Virtualization approaches: There are multiple ways to get access to the network within the container. Methods were generalized to avoid discussing unnecessary details.

- **Shared networking.** Applications in container deal with the same set of network interfaces, routes, and rules. In certain circumstances, container can control device from networking stack. Guest can create netfilter rules to redirect traffic intended for another application or take control over internal APIs by manipulating IP addresses. Depending on the purpose, it might be desired or undesired behavior, but a right practice would be to avoid such networking type. To name few, one working scheme to take over the device is to perform Man-in-the-Middle [13] (MITM) attack against secure management protocol unless certificate pinning is used. Another possibility coming from MITM is an interception of HTTP/HTTPS Web Interface traffic involving stealing of end-user login/password.

- **Virtual Ethernet (VETH)** [14] devices might have performance penalty mostly due to implementation-specific problems of network namespaces. In our tests containers take a range of private IP addresses (such as 192.168.2.0/24) and DHCP server manages IP leasing, although for rarely changing environment static IP approach can be even more effective.

Hardware traffic processing engines often have fast paths for forwarded traffic, yet for Virtual Ethernet it will still go through the main CPU, which will dramatically decrease possible performance.

2) Benchmarking methodology: The overhead of containers networking was measured accordingly. The existence of overhead for containers and host in shared networking scenario was checked. The test had been performed by using Raspberry Pi 3 connected to our MIPS device by Ethernet cable. **iperf** was used in server mode on Pi side and in client mode on a device side. Shared networking setup for both host and guest is demonstrated in Fig. 1.

For isolated networking, setup was harder to accomplish. The main idea of whole VETH setup is to have a separate networking namespace, so we couldn’t just repeat the first scenario. With VETH interface added to the same bridge as the generic LAN interface (Fig. 2) results were recorded again. However, in our case packet acceleration hardware, more specifically its driver, effectively broke TCP communication for the scenario. Once driver had been disabled we measured performance again, even though the host side was left essentially the same as in shared network test.

For TCP the following commands were issued:

```
iperf -s -p 5000 #server
iperf -c 192.168.2.3 -p 5000 -t 30 \
   -b 100MB -w 10000KB #client
```

For UDP, the bandwidth parameter was reduced down to 100 mbit/s, and socket size set to 10000KB, automatically limited to 352KB. There was 5-second interval between attempts to avoid band overflow. Only server’s report was actually evaluated.

E. RAM usage

During experiment RAM usage was generally good and to some extent couldn’t be any better. The process running in container consumed exactly the same amount of memory as the one which was running outside the container. However, measuring overhead in a kernel space is an additional problem which requires a separate study.

1) RAM quotas: One of the interesting questions was what the system would do when the generic RAM usage would have risen above the bar so that it wouldn’t be possible to reclaim any memory for host OS. RAM limit through **cgroups** was generally a good way to ensure stability of the host in such situation. Unlike CPU, RAM quota wasn’t something we could change dynamically as swap partition was unused. If some container would have used e.g. 30% of all RAM, we wouldn’t have any way to squeeze it to 15% in the runtime. The solution we used was simple: leave 30% of RAM to containers. However, if generic free RAM counter would go below some threshold (e.g. 30MB) for some reason, the kill command was issued to the process in a container using the most of memory. Obviously, this scheme is only correct when containers aren’t ought to do any sensitive work. As our applications were using a fair minimum of memory, this quota rule wasn’t widely adopted.
2) Slab injection: In a quest for predictable and stable operability of a device, yet another issue was faced. Linux kernel uses slab subsystem [15] to decrease memory fragmentation when allocating objects of the same size. The outcome of it is barely visible, but if container gets a possibility to cause a kernel to allocate virtual memory, it might get problematic because the kernel cannot reclaim memory for slab in atomic context (reclaiming is done for cached pages — those used for file caching or other purposes yet essentially free). Drivers sending a lot of packets in short time frames might be unable to allocate memory, and further attempts to do so will lead to hardware queue growth. One of such examples was found in the Wi-Fi driver which under certain circumstances, such as build configuration or due to technical need, could be unable to pass traffic through fast path, and would have had to allocate regular sockets (note that IEEE802.11ac [9] standard extends the bandwidth to at least 433mbps \(\approx 54\text{MB/s}\)). The exploitation method is shown in Fig. 3.

It was investigated that having a lot of empty slabs was barely possible to achieve. Virtual memory tuning through procsf means, such as \( /\text{proc/sy}\text{s/vm/min\_free\_kbytes} \) [16] and other entries, didn’t have any influence — Wi-Fi driver could still overfill buffers and get caught in an infinite loop, which would have caused system reboot after watchdog detected the situation. Our solution included a technique called slab injection, which was used to fulfill caches with objects of the same size, selectively leaving one of the objects per page. This resulted in increased capacity for selected types of objects which couldn’t be gracefully controlled any other way.

Slab injection must be performed in the kernel mode. Consider \( s \) an object size, \( p \) to be page size. To make the algorithm more precise, it is desired to know the number of active and the total number of objects in the cache (total—active shows the number of free objects within existing slabs which can make algorithm ineffective if real active \( \neq 0 \) and total \( \neq 0 \)). The method is shown in Fig. 4. We have also created a GitHub project for it\(^8\).

This technique was proven to be working well on small size objects, which were of primary interest, however it couldn’t inject pages to slab caches with object size close to \( P \), i.e. when page can only fit one object of that size. For such cases a patch disabling shrinking for certain user-defined caches was created.

RAM (and to some extent ROM) usage was actually one of reasons why Docker hadn’t been used. At the moment of writing Docker was a multi-binary application written in Go.

\(^8\)https://github.com/mmenschchikov/slab_inject

```c
1: function PREFILLCACHE(cache, active, total)
2:   \( c \leftarrow 1 \)
3: for \( i \leftarrow \text{active}, \text{total} \) do
4:   \( O_i \leftarrow \text{KMEM\_CACHE\_ALLOC}(cache) \)
5:   \( c \leftarrow c + 1 \)
6: end for
7: return \( O \)

8: end function

9: procedure FREEPREFILLED(cache, \( O \))
10: \( c \leftarrow 1 \)
11: while \( \exists O_i \) do
12:   \( \text{KMEM\_CACHE\_FREE}(cache, O_i) \)
13: \( c \leftarrow c + 1 \)
14: end while
15: end procedure

16: procedure INJECTPAGE(cache, active, total, \( s, p \))
17: \( P \leftarrow \text{PREFILLCACHE}(cache, \text{active}, \text{total}) \)
18: \( n \leftarrow p/s \)
19: for \( i \leftarrow 1, n \) do
20:   \( O_i \leftarrow \text{KMEM\_CACHE\_ALLOC}(cache) \)
21: end for
22: for \( i \leftarrow 1, n - 1 \) do
23:   \( \text{KFREE}(O_i) \)
24: end for
25: \( \text{FREEPREFILLED}(cache, P) \)
26: end procedure
```

Docker didn’t support MIPS in its build system and source files. While the build support was fixed (it involved rewriting of make files, adding .mips.go files, adding 32 bit support whenever applicable, and fixing gccgo/go selector to choose target compiler more properly), shared libraries couldn’t be generated, so we ended up with Docker binaries bigger than flash size. In our case, even if shared libraries could work, Go runtime potentially could still be too big, leaving too large working set for the running process. That were the reasons Docker porting had been stopped.

### F: Fault tolerance and debugging

The important part of any consumer device is a failure tolerance. While our main software goes through very intensive testing proactively detecting both kernel crashes and userland problems, third-party packages might be less tested and, what’s even harder to check, built without proper knowledge of hardware- or software-specific weaknesses leading to kernel panics.

Our measures against runtime loops employ a watchdog which operates at kernel and userland levels and proactively reboots the device if any long-standing lockup is detected.

The other measure is a safe mode. If a device fails booting for 5 times, the partition with external packages is not loaded at all.

In all failure cases, the software collects configuration, core files, and logs, compresses them to a single archive and makes
it available for an investigation through service provider’s means.

G. File system

1) Single repository policy: Lack of disk space was the largest problem in our system. The task was to get a file system (FS) configurator spawning FS with the following properties:

- Consistent.
- Allows no duplicates.
- Free of “DLL hell” and versioning problems.

Following qualities can only be achieved by using a single repository policy. The software can not actually receive a new software package without getting update availability status. Once the new software is available and loaded, it is assumed that the package is compatible with other packages installed to the system. In our case, the presence of the Internet on the device was essential to functionality and user was not given an ability to install packages without it, so the system was quite stable in that aspect.

2) System file security: Another important point is how to deal with system files: there must be an ability to load additional packages with libraries, yet packages must have no capability to write to files they don’t own. A read-only bind mount sounds like an adequate response to the problem.

Container file system is built from read-only bind mounts of package content (allowing containers share packages), generic file systems such as devfs, procfs, sysfs and tmpfs. Whole container file system is physically located in a separate folder inside package partition, so free space is directly taken from it. The scheme of container file system sources is shown on the Fig. 5. No problem had been found maintaining such a system, nor any sign of security problems it may bring (besides the case when such mount is read/write) was detected.

3) Performance and benchmarking methodology: Our file system is an OverlayFS [17] built on top of squashfs [18] (read-only part) and ubifs [19]. The dd utility was used to measure the impact of containers on file system performance:

```bash
for i in `seq 1 100` ; do
   /tmp/busybox dd if=/dev/zero \ 
   of=/test/file bs=4096 \ 
   count=10000 2> write$i.txt
done
```

The following script was used for testing read performance.

```bash
for i in `seq 1 100` ; do
   echo 3 > /proc/sys/vm/drop_caches
   /tmp/busybox dd if=/test/file \ 
   of=/dev/null bs=4096 \ 
   count=10000 2> read$i.txt
done
```

Cache dropping was used to prevent the operating system from filling file system caches, significantly improving performance in consequent tests.

4) Container activation time: When the container is not enabled and some application is about to start inside it, activation is fairly quick: it is essentially the same as few mount calls and fork with exec. However, if the container is running and some package is ought to be bound to it, the only option is to use shared folders to transfer bind mounts from host to container. This process is indeed not that trivial, big packages required up to few seconds to bind. Fortunately, the whole scheme in which container can’t be shared among packages however greatly minimizes the possibility of such a long delay down to zero.

In case of Docker that time could be seriously different: as docker images tend to include operating system templates, mount time would be less proper.

H. Container API

Software packages should have an ability to perform system tasks such as “stop container” request, cron-like application start management, listing of computers in LAN and so on. With container networking it could become troublesome to distinguish the container from others and define access control rules. Consider two use cases.

- Shared networking was determined to provide no real possibility to determine the container, granted that container always has a possibility to change its network configuration. Spoofing at container’s side is generally too trivial for real-world usage, it can be done through iptables or ip.
- Isolated networking (through Virtual Ethernet (VETH) devices) provides a full-fledged IP management. With VETH the determination of IP spoofing within the container is trivial just by comparing of IP addresses within the container with predefined/leased value and specific netfilter rules. Any newly created networking interface won’t let the traffic flow outside the container.

Consequently, due to the usage of isolated networking, the container API server had been set up to listen at specific VETH interface.
TABLE I
CPU PERFORMANCE MEASURES FOR HOST AND GUEST

<table>
<thead>
<tr>
<th>Test</th>
<th>Host Avg.</th>
<th>Guest Avg.</th>
<th>Diff.</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric Sort</td>
<td>157.89</td>
<td>157.94</td>
<td>0.054</td>
<td>0.033</td>
</tr>
<tr>
<td>String Sort</td>
<td>6.145</td>
<td>6.121</td>
<td>0.024</td>
<td>0.368</td>
</tr>
<tr>
<td>Bitfield</td>
<td>0.0020270</td>
<td>0.015896</td>
<td>-1.275</td>
<td>-0.201</td>
</tr>
<tr>
<td>FP Emulation</td>
<td>39.715</td>
<td>39.746</td>
<td>-0.031</td>
<td>-0.096</td>
</tr>
<tr>
<td>Fourier</td>
<td>51.074</td>
<td>51.146</td>
<td>-0.073</td>
<td>-0.139</td>
</tr>
<tr>
<td>Assignment</td>
<td>2.8199</td>
<td>2.8622</td>
<td>-0.042</td>
<td>-1.5</td>
</tr>
<tr>
<td>IDEA</td>
<td>755.0928</td>
<td>753.0852</td>
<td>2.006</td>
<td>0.269</td>
</tr>
<tr>
<td>Huffman</td>
<td>171.1702</td>
<td>171.0864</td>
<td>0.083</td>
<td>0.049</td>
</tr>
<tr>
<td>Neural Net</td>
<td>0.0635</td>
<td>0.0636</td>
<td>-0.00095</td>
<td>-0.149</td>
</tr>
<tr>
<td>LU Decomp.</td>
<td>2.0483</td>
<td>2.0688</td>
<td>0.0205</td>
<td>0.4616</td>
</tr>
</tbody>
</table>

TABLE II
TEST TIME MEASURES IN PARALLEL MODE

<table>
<thead>
<tr>
<th>Test</th>
<th>Host (sec)</th>
<th>Guest (sec)</th>
<th>Diff. (sec)</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 proc</td>
<td>533.15</td>
<td>535.89</td>
<td>-2.74</td>
<td>-0.662</td>
</tr>
<tr>
<td>2 proc</td>
<td>680.86</td>
<td>680.345</td>
<td>0.015</td>
<td>0.026</td>
</tr>
<tr>
<td>4 proc</td>
<td>1338.323</td>
<td>1337.288</td>
<td>1.035</td>
<td>0.077</td>
</tr>
<tr>
<td>8 proc</td>
<td>2676.929</td>
<td>2681.940</td>
<td>-5.011</td>
<td>-0.187</td>
</tr>
</tbody>
</table>

I. Power consumption

Power consumption was measured during parallel runs of nbench utility for both dual core and single core variants of MIPS CPU. The biggest power consumers such as Wi-Fi antenna were disabled, power consumption in an idle state was measured. Absolute watt numbers are irrelevant: since our hardware is not publicly available, it can’t serve as a baseline for future tests except ours, the difference between power consumption in an active state and the idle state is provided. Of course, since idle state boundary might be floating, it was measured before every test.

V. RESULTS

A. CPU performance

Average results (calculated using bytemark_counter utility), are shown in Table I and Table II. All results were uploaded to GitHub for convenient interpretation.

In 60% of tests the guest is faster than the host (we give our explanation of this fact later). One test shows difference more than 1% (1.5%), while in 90% of tests measures in absolute numbers are well lower than 0.5%. 30% of numbers are even lower than 0.3. The maximum difference is 1.5%, it is reached in assignment test.

In the second test results seem to be more sporadic, ranging from 0.662% win of the host over the guest to 0.226% win of the guest. It is notable that there is no generic trend.

B. Network performance

Raw results had been uploaded to GitHub. Average results are shown in Table III.

TCP tests for the guest show 91.29% and 91.8% of host’s performance. In UDP tests host gets 99.68%/99.39% of guest’s speed. While not outlined, the impact on CPU was comparable.

C. File system performance

Average results are demonstrated in Table IV. The difference is low: 0.13 MB/s, 0.022 MB/s and 0.034 MB/s. Therefore the maximum difference is about 130 KB/s (which is alone not quite small, but incomparable to generic performance).

D. Power consumption

Average results are shown in Table V. Note that we show results for single core and dual core counterparts in form single core result / dual core result. iperf was tested in both shared and isolated modes, but results were mixed because they don’t differ.

Overall, no measurable difference in power consumption between host and guest was found. Two and more processes indeed had shown the rise of dual core processor consumption by 0.9W.

VI. DISCUSSION

A. Benchmark

Our interpretation of results led us to a conclusion that the impact of container-based virtualization is low. Raw CPU performance is better for a guest than for a host, which contradicts to common sense (although the similar statistics...
is provided in another research [20]). In our understanding, it is scheduling which is to blame since scheduler’s choice of quantum might substantially differ for the process in another PID namespace, and we tend to believe the implicit CPU share between namespaces is quite fair. Another possible reason might come from a difference between device runs, as the one added by varying service start time, but it shouldn’t have any serious impact.

While generally incomparable, dd write test (unsynchronized) outperforms read test solely due to writeback mechanism. A synchronized test is slower due to writethrough, the direct write to flash. The filesystem cache was dropped before every reading test, so the result can be considered “pure”. Since the difference is too small, we assume that containers don’t add any serious overhead for the file system in our use case.

As for network performance, UDP protocol tests demonstrate guest outperforming host — it looks like an effect of scheduling. At the same time TCP protocol is generally more consuming and it adds some overhead to container environment (about 8-9% of host speed).

Power consumption for host and containers was equal. It didn’t come as a surprise, our theoretical understanding also implied this. It is very notable that for single core processor the power consumption is even less for nbench testing. Scheduling and different computational power needed for specific purposes seem to be the reasons for it. Dual core processor faced the rise of power consumption when second core woke up. In general, this result shows there is no any difference in energy consumption between host and lightweight containers, granted they don’t bring any additional computational costs besides ones introduced by kernel namespaces.

Overall we can conclude that advantages of containers outweigh negative impact of additional namespaces in the kernel. File system performance and raw CPU performance are almost unaffected by it. Network performance went down, however, 8-9% is a reasonable price for extra security and comfort. Power consumption doesn’t rise. Therefore Linux Containers technology is mature enough for use on resources constrained devices according to our tests.

Generic latency, network latency, temperature and other important points were overlooked due to low interest. Those are definitely worth examination in the future.

B. System design

The suggested system design works well for our use cases, however, there are problems in each core aspect which might be crucial for specific purposes. The system is not well-protected against kernel lockups: maintainers must be extra careful when allowing specific kernel APIs or access to kernel drivers. That’s mostly vendor-specific limitation than the design flaw. CPU quotas are operating properly, but a stricter control over container’s quotas might be needed: otherwise limits might get reduced too much based on a very imprecise measure of “application response time”. At the same time, RAM quotas aren’t flexible at all, and the solution for this problem is unclear.

File system architecture is providing low overhead (as seen in tests), however, it is still important to audit overlays for possible node problems [21]: there is a strong need to know it for sure if there is an intention to build reliable platform.

Container API needs strict enforcement of IP addresses inside the container. Our system doesn’t have an access restriction for certain containers, so this is again not a problem. If it was the case, we could have employed some other mechanism for communication, e.g. UNIX sockets.

VII. RELATED WORK

The container-based virtualization was first explored with [22] work, continued to embedded devices with Cells [23], the virtualization system for Android smartphones.

Service-hosting gateways [24] were one of the first attempts to define requirements and build a proof-of-concept of a more constrained device with containers running. A. Krylovskiy [20] examines containers on Raspberry Pi, the embedded device built upon ARM CPU. We expand the topic further by using MIPS CPU with even fewer resources available. The notable thing is that a lot of studies focus on proof-of-concepts, while our approach was tested in production. One of the other examples of a system used in real-world applications is resinOS [25], but it is still tailored for more resource-rich devices, most notably due to more RAM and disk space.

ParaDrop [26] is a multi-tenant platform for third-party services on wireless gateways. While ParaDrop’s goal is to “push services from data centers to the network edge” [27], our goal differs in a way to allow partners further extend device’s software package with the ease and adequate level of security.

Paper [28] provides a high-level overview of lightweight container problematics on IoT devices, which we partially cover.

The framework for block-streaming application execution [29] demonstrates a method of loading software binary chunks to resource-constrained hardware. Our device is not that limited by RAM and ROM, it is running Linux and full-fledged executable files, yet it provides a foundation for even more constrained devices.

Aimed at performance, another research [30] is concentrating on Raspberry Pi and Odroid (ARM) single-board computers running Docker images. Generally, our study differs from it as we are not focused on performance but rather building a system from scratch. Also, all tested systems are considerably faster and have a lot more built-in RAM. Obtained results are indeed similar, showing a negligible impact of container virtualization on these systems.

As for full-fledged virtualization, many efforts [31]–[33] had been put into making it work for embedded MIPS CPUs in various ways. Moratelli, Zampiva, and Hessel [31] presented an embedded hypervisor for real-time execution of applications. Low overhead was observed for single processor platforms, and a certain penalty detected for multiprocessor systems. There were updates to the research, e.g. [34], dealing
with hardware-assisted interrupt delivery. The approach looks promising, but just as any virtualization requires slightly more disk space and computational power. KVM-Loongson [33] authors develop processors based upon MIPS with an efficient hypervisor, with only I/O being a serious bottleneck. Still it is a hardware solution, which is not what most users can integrate. Hypervisor OmniShield [7] is CPU vendor-backed, so it is assumed to have a good support in recent MIPS processors. Also, it brings improvements to hardware-based domain isolation. All it makes it a good alternative once sufficient resources are present.

VIII. CONCLUSION AND FUTURE WORK

We managed to create a system for running containers on resource-constrained embedded devices. Working design for different aspects was suggested. Major problems found during the development process, namely slab overflow, packet acceleration engine troubles, container API considerations, were explained. We compared the performance of host and containers and found the difference in most of the tests quite negligible with host winning only in TCP networking tests. Speed test result is confirming that LXC environment is ready for container-based third-party applications. Power consumption measurement doesn’t show any difference between containerized and native environments.

Our future work would be concentrated on further examination of container capabilities on embedded devices, as well as the practical work towards improving disk space and RAM usage. Additionally, we have an intention to make a security audit of the described system.

REFERENCES


An optimistic approach to handle out-of-order events within analytical stream processing

Igor E. Kuralenok #1, Nikita Marshalkin #2, Artem Trofimov #3, Boris Novikov #4

# JetBrains Research
Saint Petersburg, Russia

1ikuralenok@gmail.com 2marnikitta@gmail.com 3trofimov9artem@gmail.com 4borisnov@acm.org

Abstract—In recent years, there has been a growth in research and industrial solutions in the field of distributed stream processing. However, even state-of-the-art stream processing systems experience difficulties with out-of-order data arrival. The most common solution to this issue is buffering. Its main problem is the extra cost for blocking before each order-sensitive operation. The goal of this paper is to propose and evaluate an optimistic approach to handle out-of-order events. We introduce a method that is suitable for any stateful operation and needs a single buffer for the complete computational pipeline. Such technique requires extra network transfers and re-computations, but the experiments demonstrate that a prototype of our approach is able to create low overhead while ensuring the correct ordering.

I. INTRODUCTION

Nowadays, a lot of real-life applications use stream processing for network monitoring, financial analytics, training machine learning models, etc. State-of-the-art industrial stream processing systems, such as Flink [1], Samza [2], Storm [3], Heron [4], are able to provide low-latency and high-throughput in distributed environment for this kind of problems. However, providers of the order-sensitive computations remain suboptimal. Most of these systems assume that events are fed to the system with monotonically increasing timestamps or with minor inversions. Often, such timestamps can be assigned at system’s entry. Nevertheless, even if input items arrive monotonically, they can be reordered because of the subsequent parallel and asynchronous processing. In this case, order-sensitive operations located deeper in data flow pipeline can be broken. Figure 1 shows the example of common distributed stream processing pipeline that breaks the input order of the operation 2, even if inputs are monotonic and links between operations guarantee FIFO order.

The typical way to achieve in-order processing is to set up a special buffer in front of operation. This buffer collects all input items until some user-provided condition is satisfied. Then the contents of the buffer are sorted and fed to the operation. The main disadvantage of such technique is latency growth. This issue becomes even more significant if the processing pipeline contains several operations that require ordered input. The problem here is that each buffer increases latency, because of the additional waiting time.

The alternative is to develop business logic tolerant to out-of-order items. However, this approach is suitable only for a limited set of tasks. Moreover, it may dramatically complicate the business-logic, which can lead to the maintenance cost increase and is error-prone.

In this paper we introduce an optimistic approach to handle out-of-order items. Our evaluation demonstrates the low overhead of our method. The contributions of this paper are the following:

- Definition of new optimistic technique to handle out-of-order items in stateful operations that requires single buffer per computational pipeline
- Analysis of properties of this approach
- Demonstration of working example that applies proposed method

The rest of the paper is structured as follows: in section II we formalize the preliminaries of stream processing, the examples of tasks that require ordered input are described in section III, the typical approaches for handling out-of-order events are discussed in IV, our optimistic technique is detailed in V and its performance is demonstrated in VI, the main differences between proposed method and existing ones are shown in VII, finally we discuss the results and our plans in VIII.
II. STREAM PROCESSING CONCEPTS

In this section we define some preliminaries for distributed stream processing. It allows us to unify required terms and to introduce definitions, which are needed for the subsequent statements.

In this paper a stream processing system is considered as a shared-nothing distributed runtime. It handles input items and processes them one-by-one according to user-provided logic. It is able to handle a potentially unlimited number of items. The main requirement of this kind of data processing systems is to provide low latency between event occurrence and its processing under a fixed load. The term distributed implies that user’s procedures can be partitioned into distinct computational units or shards. The following subsections detail the properties of such systems more precisely.

A. Data flow

The basic data flow abstraction is a stream. The stream is an infinite sequence of heterogeneous data items. Each data item contains a payload defined by a user. Besides, it can be accompanied by meta-information. Typically, meta-information is used to define an order on data items. For instance, it can be represented as a UNIX timestamp in milliseconds.

B. Computation flow

Commonly, the computational pipeline is defined in the form of logical graph. The vertices of the logical graph are operations, and the edges are links between them. Logical graph defines only relations between operations, but it does not describe the physical deployment. The logical graph for the pipeline shown in Figure 1 is presented in Figure 2.

C. Operations

There are two main types of streaming operations: stateless and stateful. Stateless operations do not need any knowledge about past inputs to process current one correctly. A simple illustration is a map operation that multiplies by two any numeric input item’s payload. On the other hand, stateful operations are able to keep some aggregations or summaries of received events. In such case, the output of the operation depends not only on the input but also on its current state. As an example, one can define an operation that sums all previous items with numerical payloads.

D. Physical deployment

As mentioned above, each operation can be partitioned between multiple computational units. Data items can be balanced between partitions by key extracted from an item’s payload for stateful operations. For stateless operations items can be balanced randomly. The schema of physical partitioning of operations is sometimes called physical graph. Regarding physical links between operations, in the most cases, it is assumed that they guarantee FIFO order.

E. Guarantees

Recognized important property of stream processing systems is the type of guarantees it provides in case of failures. There are three main types of such guarantees. At most once semantics states that each input event is processed once or not processed at all. At least once guarantees that each input item is processed, but possibly multiple times, that can lead to result duplication. Exactly once semantics guarantee that each input event is processed exactly one time.

III. TASKS THAT REQUIRE IN-ORDER INPUT

In this section we outline common problems that require the order persistence of input items and describe a couple of computation scenarios, which can be found in many real-life projects.

A. Tasks requiring complete event retrieval

The processing of the single event could be split into multiple independent parts that are executed in parallel. After execution finishes, the results must be combined into a single cumulative item. This task could be naturally implemented using order guarantees: the final part of the task could be flagged and receiving the flagged result guarantees that the rest of the operation is completed. Unfortunately, as it was shown in Figure 1, independent processing via different paths can lead to reordering.

As an example, we can mention the computation of inverted index. Pipeline shown in Figure 1 can be applied for the task. In this case, operation 1 accepts documents from the input and for each word produces corresponding positions. Operation 2 receives pairs of word and positions and computes changelog of the inverted index for each word. In order to produce changes for each document in the form of single update, there is a need for retrieval all changelogs regarding the document.

B. Tasks that depend on the order of input events

This class includes all non-commutative operations. Such tasks strictly require the order of input items, because there are no any other methods to compute a valid result.

Generally, this class of examples includes all windowed aggregations. Moving average calculation over a numerical stream is a typical case. Even if values inside window could be arbitrary reordered, the order between windows is required to ensure that incomplete windows are not produced.
IV. EXISTING SOLUTIONS

There are two most common methods that are used to implement order-sensitive operators: in-order processing (IOP) [5], [6], [7] and out-of-order processing (OOP) [8].

A. In-order processing

According to IOP approach, each operation must enforce the total order on output elements that can be violated due to asynchronous nature of execution. Buffering is usually used to fulfill this requirement. Figure 3 shows the union operation that combines multiple streams into the single one. Both input streams are ordered, as predecessors must meet ordering constraint. Nevertheless, if there is arrival time skew between input streams, the union must buffer the earlier stream to produce ordered output. It is known that IOP is memory demanding and has unpredictable latencies and limited scalability [8].

![Fig. 3. IOP union operation. Due to delay of the upper stream operation must buffer elements](image)

B. Out-of-order processing

OOP is an approach that does not require order maintenance if it is not needed. In the case of ordering requirements, OOP buffers input items until a special condition is satisfied. This condition is supported by progress indicators such as punctuations [9], low watermarks [10], or heartbeats [11]. They go along the stream as ordinal items, but do not trigger business logic of the operations. Each progress indicator carries meta-information and promises that there are no any elements with lesser meta-information. Therefore, indicators must be monotonic, but data items between two consecutive indicators can be arbitrarily reordered. Data sources periodically yield them.

A timed window operation can be mentioned as an example of OOP approach. A window operation buffers partial results until a progress indicator arrives. After that, the window flushes corresponding buffers and propagates the indicator to the next operation down the stream.

OOP addresses the downsides of IOP, but the direct implementation has flaws too. Even if the input stream is totally ordered, the operation must wait for the progress indicator. Figure 4 illustrates such case. Bottom window is complete but must be blocked until the indicator for the item 11 arrives. Another issue of OOP is that periodical flushes can result in load bursts and an increase in latency.

![Fig. 4. OOP sliding window, range=3, slide=1. Operation must block lower window until next punctuation arrival](image)

V. OPTIMISTIC APPROACH

The main idea of our method is to represent stateful transformations as a sequence of a map and windowed grouping operations and handle out-of-order items within them.

Following our approach, we make some assumptions about stream processing system, that is suitable for applying it. Such system must support meta-information on data items, allow cycles in the logical graph, and its set of predefined operations must be sufficient to express map and windowed grouping operations. Additionally, OOP indicators should be supported.

The ordering model of data items is defined at the beginning of this section. Then, we show that any stateful transformation can be implemented using the combination of windowed grouping and map operations. After that, we demonstrate an optimistic approach to handle out-of-order items within these operations. At the end of the section, the limitations of such technique are discussed.

A. Ordering model

We assume that there is a total order on data items. If there are multiple data sources and there is no natural ordering between them, the source id can be used in addition to the locally assigned timestamp. The items are compared lexicographically: timestamps first, then source id. Ordering is preserved when an item is going through the operations. More precisely, the order of output items is the same as the order of corresponding input items. If more than one item is generated, they are inserted in output stream sequentially. Moreover, the output follows corresponding input but precedes the next item. The ordering model is shown in Figure 5. $F(x)$ is an arbitrary transformation. Replication and union are used to inject original unmodified items into the resulting stream to show the order between items.

B. Semantics of map and windowed grouping operations

1) Map: Map transforms input item into a sequence of its derivatives, according to a user-provided function $f$. This sequence can consist of any number of items or even be empty.

2) Windowed grouping: Windowed grouping is a stateful operation with a numeric parameter window. It is supposed that payloads of input items of grouping operation have key-value form. The state of this operation is represented by a set of buckets, one for each key.

Windowed grouping has the following semantics:
Each input item is appended to the corresponding bucket.

The output of grouping operation is a window-sized tuple of the last items in the corresponding bucket. If bucket size is less than window, the output contains full bucket of the last items in the corresponding bucket. If bucket size is greater than window, the output contains full bucket of the last items in the corresponding bucket.

The pseudocode is presented in Algorithm 1. **Emit** function is called to send new items downstream.

```
Algorithm 1 Grouping semantics
1: function INSERT(item, bucket)
2:  APPEND(bucket, item)
3:  left ← max(0, bucket_length − window)
4:  right ← bucket_length
5:  EMIT(new DataItem(bucket[left, right]))
6: end function
```

The following example illustrates the semantics of the windowed grouping operation. In this example, payloads of input items are represented as natural numbers: 1, 2, 3, etc. The hash function returns 1 if the number is even and 0 otherwise. If the window is set to 3, the output is:

(1), (2), (1|3), (2|4), (1|3|5), (2|4|6), (3|5|7), (4|6|8)...

The special case of grouping with window = 2 in conjunction with a stateless map is used to implement arbitrary stateful transformation.

**C. Stateful transformations using defined operations**

Figure 6 shows the part of the logical pipeline, that can be used for stateful transformation. The input of windowed grouping operation is supposed to be ordered. There are several functional steps to produce output and update state. There are two cases of these steps:

- When the first item arrives at grouping, it is inserted into the empty bucket. The grouping outputs single-element tuple, and then it is sent to the combined map. Combined map generates state object and sends it back to the grouping in the form of an ordinal key-value data item. The key of the state item is the same as in the item in tuple and value is the state. Combined map can generate some additional output and send it further down the stream.

- When the state item arrives at grouping, it is inserted into the tail of the corresponding bucket after the item that triggers state updating. The ordering model guarantees that the state item would be processed before the next item. Grouping outputs tuple with this item and the state. However, combine map filters out such tuple, because its last element is the state. This fact implies that the state has been already combined with the first item in the tuple.

- When new regular input item arrives at windowed grouping, it is inserted into the corresponding bucket’s tail, because of the order assumptions. Additionally, the right ordering guarantees that input item is grouped into the tuple with previously generated state item. The next map operation combines new item and previous state into the new state item. After that, the new state item is returned to the grouping through the cycle. As in the first case, combined map can generate some additional output.

The example of square matrix multiplication within proposed approach is shown in Figure 7. In this example, input items are represented as key-value pairs, where the key is the dimension of a matrix, and the value is the matrix itself. The reaction on three input matrices are the following:

- When the first matrix A arrives at grouping, it is put into the empty bucket for 3x3 matrices. After that, the single-element tuple with matrix A is sent to combine map operation. Combine map creates state object for matrix A, which is just A itself. In the last step, state item is sent back to grouping, and it is inserted right after item for matrix A.

- Matrix B is arrived and inserted into the bucket right after state item. The tuple containing state item and item for matrix B is sent to combine map. Combine map multiplies matrix in the state by matrix B. The result of this operation is matrix AB. New state item for matrix AB is created and sent back to the grouping. It is inserted in bucket right after item with matrix A.

- Matrix C is arrived and went through the pipeline in a similar way as matrix B.

**D. Handling out-of-order events**

When we introduce the model for stateful operations, we assume that all items arrive at grouping in the right order.
However, as it was shown above, it is not possible in practice without additional buffering. We propose the following approach to handle events in grouping:

- If an item arrives in order, it is processed as usual
- If two items are out-of-order, and the grouping observes the second one then it is inserted into the corresponding bucket at the position defined by the meta-information. After that, tuples, which contain new item are generated and sent further down the stream. At the same time, for tuples, that has been produced but became invalid, tombstones are sent.

Tombstones are ordinal data items but with a special flag in meta-information. This flag means that tuples with such meta-information are invalid, and they should not leave the system. Tombstones have the same payloads as invalid items in order to go through the same path in the computational pipeline.

The pseudocode of the grouping is shown in Algorithm 2.

The functions accept new element, depending on its type. They also receive a bucket for element’s hash. **Emit** function is called to send new items downstream.

This technique guarantees that all correct tuples are eventually produced. However, invalid ones are also generated. Therefore, there is a need for **barrier** at the pipeline’s sink, that filters invalid items when corresponding tombstones arrive. The barrier is partially flushed for some meta-information interval when there is a guarantee that there are no any out-of-order items and tombstones further up the stream for this range. This guarantee can be provided by punctuations or low watermarks, as it is implemented in the most stream processing systems. The fundamental idea behind this approach is to shift blocking as far as possible down the stream. Notably, this is the only buffer in the whole system, unlike existing solutions.

The pseudocode for the barrier is shown in Algorithm 3.

The function **Insert** is called by the system on the new item’s arrival. **Punctuation** is called when there is a guarantee that there are no tombostones up the stream with the specified time. The OOP architecture [8] can be employed to provide such guarantee.

### E. Advantages and limitations

The proposed architecture’s performance depends on how often reorderings are observed during the runtime. In the case when the order naturally preserved there is almost no overhead: when the watermark arrives, all computations are already done. The probability of reordering could be managed on a business-logic level and optimized by the developer. In experiments section it is shown that the computational nodes count is one of such parameters.
Algorithm 3 Barrier
1: \( buffer \leftarrow {} \)
2: 
3: function \textsc{Insert}(item)
4: \hspace{1em} position \leftarrow \text{lowerBound}(item, buffer)
5: \hspace{1em} if \text{isTombstone}(item) then
6: \hspace{2em} \textsc{Remove}(buffer, position)
7: \hspace{1em} else
8: \hspace{2em} \textsc{Insert}(buffer, position)
9: \hspace{1em} end if
10: end function
11: 
12: function \textsc{Punctuation}(time)
13: \hspace{1em} for all item : item \in buffer \& item_{ts} < time do
14: \hspace{2em} \textsc{Emit}(item)
15: \hspace{2em} \textsc{Remove}(item, buffer)
16: \hspace{1em} end for
17: end function

Regarding the weaknesses, this method can generate additional items, which lead to extra network traffic and computations. Experiments, which are shown in the section VI demonstrate that the number of extra items is low.

VI. EXPERIMENTS

A. Setup

We performed the series of experiments to estimate the performance of our system prototype. As a stream processing task, we apply building an inverted index. This task is chosen because it has the following properties:

1) Task requires stateful operations. It allows us to check the performance of the proposed stateful pipeline
2) Computational flow of the task contains network shuffle that can violate the ordering constraints of some operations. Therefore, inverted index task can verify the performance of our optimistic approach
3) The load distribution is skewed, because of Zipf’s law

These properties make the task sufficient to comprehensively analyze the performance of the proposed solution. Building inverted index can be considered as the halfway task between documents generation and searching. In the real-world, such scenario can be found in freshness-aware systems, e.g., news processing engines.

The logical pipeline of this computation is shown in Figure 8. First map operation accepts Wikipedia documents and outputs pairs of words and corresponding positions. The next part of the pipeline accepts pairs of word and positions and computes updated posting list and the actual changelog.

This stateful transformation is implemented in the form of grouping and map operation with a cycle, as it was shown in the previous section. Regarding the physical deployment, the full logical graph is deployed on each computational unit or worker. Documents are randomly shuffled before the first map operation. Word positions are partitioned by word before grouping. The other links are implemented as simple chain calls.

Our experiments were performed on clusters of 10 nodes. Each node is an AWS EC2 micro instance with 1GB RAM and 1 core CPU.

B. Overhead and scalability

As a key metric in our experiment, we take the ratio of arrived at the barrier items count to the number of the valid items among them. This value clearly represents the overhead of our approach, as it was mentioned at the end of the previous section.

The relation between the number of workers, the delay between input documents and the proposed ratio is shown in Figure 9. As expected, the peak of the ratio is achieved when the document per second rate is high, and the number of the nodes is low. This behavior can be explained by the fact that a few workers cannot effectively deal with such intensive load. Nevertheless, the proportion of invalid items reduces with the increase of workers number. Under non-extreme load, the total overhead of the optimistic approach is under 10% for all considered number of workers. These results confirm that the ratio does not increase with the growth of the number of nodes.

Therefore, the most important conclusions of the experiments are: the proposed method is scalable, the overhead could be optimized by system setup.

VII. RELATED WORK

Research works on this topic analyze different methods of handling out-of-order items. Most of them are based on buffering.

K-slack technique can be applied, if network delay is predictable [12], [13]. The key idea of the method is the assumption that an event can be delayed for at most K time units. Such assumption can reduce the size of the buffer. However, in the real-life applications, it is very uncommon to have any reliable predictions about the network delay.
IOP and OOP architectures, that are mentioned in the section IV, are popular within research works and industrial applications. IOP architecture is applied in [6], [14], [5], [15], [16], [17]. OOP approach is introduced in [8] and it is widely used in the industrial stream processing systems, for instance, Flink [1] and Millwheel [10].

Optimistic techniques are less covered in literature. In [18] so-called aggressive approach is proposed. This approach uses the idea that operation can immediately output insertion message on the first input. After that, if that message became invalid, because of the arrival of out-of-order items, an operation can send deletion message to remove the previous result and then send new insertion item. The idea of deletion messages is very similar to our tombstone items. However, authors describe their idea in an abstract way and do not provide any techniques to apply their method for arbitrary operations.

Yet another optimistic strategy is detailed in [19]. This method is probabilistic: it guarantees the right order with some probability. Besides, it supports only the limited number of query operators.

VIII. CONCLUSION

In this paper we introduce an optimistic approach for handling out-of-order events. Our technique has the following key properties:

- It does not require buffering before each order-sensitive operation
- The method handles properly any stateful operation
- The overhead of the proposed approach is low (under 10% in most of our experiments)
- The total overhead could be managed by optimization of the computational layout

The optimistic nature of this method is able to help to reduce the cost of waiting for punctuations or watermarks. It is implied by the fact, that at the moment when the watermark arrives all computations are already done.

The experiments show that the number of the extra items does not increase with the growth of the number of the computational units. Therefore, this approach can potentially provide lower latency in stream processing systems.

REFERENCES


Fig. 9. The relation between the number of workers, the delay between input documents and the replay ratio


Keyword Extraction from Single Russian Document

Mikhail Vadimovich Sandul  
ITMO University  
St. Petersburg, Russia  
sandulmv@yandex.ru

Elena Georgievna Mikhailova  
Saint Petersburg State University, ITMO University  
St. Petersburg, Russia  
e.mikhaylova@spbu.ru

Abstract—The problem of automatic keyword and phrases extraction from a text occurs in different tasks of information retrieval and text mining. The task is the identification of terms that best describe the subject of a document. Currently there are a lot of research to solve this problem. Basically, algorithms are developed for texts in English. The possibility of applying these algorithms to the Russian texts are not sufficiently investigated. One of the most known algorithms for solving the problem of keyword extraction is RAKE. This article examines the effectiveness of RAKE algorithm for texts in Russian. The work also applies the hybrid method, which uses the $\Gamma$-index metric for phrases weighting, which were obtained using the algorithm RAKE. The article shows that this algorithm is more accurate than PAKE while reducing the number of selected phrases.

I. INTRODUCTION

Organizing data into a database requires expenses on database designing, data organizing, etc. These expenses are not necessarily related to the analysis of the data itself. So, we spend additional time and effort on preprocessing to get new knowledge from a data. The worst part is that organizing some types of a data into a database can lead to a loss of the content. Usually it’s impossible to convert text documents into a table (in case of relational DB) without loss of its meanings. Thus, texts are usually stored unchanged as BLOB-fields. As a result, we can say that using databases for analysing texts is not efficient.

As we can see, it is hard to bring structure into a text for analysis. However, we would like to extract useful information from it anyway. The branch of science which deals with such problems is called Text Mining[1].

Nowadays a lot of real-world problems are subjects of study of Text Mining starting from typical Data Mining problems such as classification and clustering to exclusively Text Mining problems like automatic keyword extraction and annotation[1].

Currently, amount of unstructured text information constantly grows. Keywords can help to get new knowledge from it because they let us understand the meaning of a text without reading it. Automatic keyword extraction is the subject of study of Text Mining.

Keywords can be applied to improve the functionality of information retrieval systems. For example, Phrasier[8] system uses keywords to find documents related to the primary one and words themselves serve as links between documents. That allows a user to navigate within documents much faster. Another example is Keyphind[9]. It is the search engine for digital libraries. Keywords are used there as the main source for building index. They are also used to enrich the presentation o search results.

Despite the wide range of applications, most documents do not have assigned keywords. Most approaches are focused on manual assignment of keywords. Usually, this procedure is done by specialists in the relevant field. In their work, they may use a fixed taxonomy or rely on author’s judgment to provide a representative list[2]. The main goal of further investigations is automatization of this process.

Early approaches to automatic keyword extraction focus on corpus-oriented statistics of individual words. However, they have some flaws. For instance, while some words might be evaluated as keywords within the whole corpus, keywords within a single document or several documents might not[2]. Also, such methods can not help us in finding keywords consists of two or more words. To avoid these drawbacks, we focus on approaches that operate on a single document.

We will describe keyword as a sequence of one or more words which provide a representation of the document’s content. An algorithm should return us a list of such sequences. Ideally, this list represents in condense for the essential content of a document. However, there is no need for these sequences to form coherent text. To avoid ambiguity, further in the article we will refer to such sequences as key phrases to highlight that they may contain more than one word. We will use the term "keyword" only to emphasize that we are dealing with a single-word sequence.

To evaluate the efficiency of an algorithm we use Precision and Recall metrics. Recall is the fraction of correctly extracted keywords among all keywords in a text:

\[
\text{Recall} = \frac{\text{correctly extracted}}{\text{all keywords in a text}}
\]

Precision is the fraction of correctly extracted keywords to a count among all extracted keywords:

\[
\text{Precision} = \frac{\text{correctly extracted}}{\text{all extracted}}
\]

Different approaches to automatic keyword extraction exist: supervised and unsupervised machine learning algorithms, statistical approaches, approaches based on linguistic features. They all can be used to solve our problem. These methods can be divided into groups if they domain-dependent or domain-independent, corpus-oriented or made for single documents, require training set or not. In this particular work will be considered domain-independent methods which do not require training set.
II. RELATED WORK AND BACKGROUND

A. TextRank

TextRank[3] is the agile graph-based method that can be used not only for key phrase extraction but also for creating annotation for a text.

Representing data as a graph is one of the approaches to retrieve information. The basic idea implemented by a graph-based ranking model is that of "voting" or "recommendations". When there an edge exists from one vertex to another, it is basically casting a vote for that other vertex. The more votes a vertex gets, the more important it becomes. Moreover, an importance of the vertex determines the importance of its "votes", and this information is also taken into account by ranking model.

Formally, let \( G = (V, E) \) be a directed graph with the set of vertexes \( V \) and the set of edges \( E \), where \( E \) is the subset of \( V \times V \). Let \( In(V_i) \) be a set of vertexes which has an edge that goes to \( V_i \). Let \( Out(V_j) \) be a set of vertexes to which leads an edge from \( V_j \). We will define a score of a vertex \( V_i \) as follows:

\[
S(V_i) = (1 - d) + d \sum_{V_j \in In(V_i)} S(V_j)
\]

where \( d \in [0; 1] \) is a dumping factor, which has a role of integrating into the model the probability to jump from the given vertex to another random one in the graph. The algorithm starts with arbitrary values of \( S(V_i) \) for each node, the computation iterates until convergence below given threshold is achieved.

To apply such method to a text, first, we need to introduce some rules according to which we will build a graph. We need to define units which will be treated as nodes and relations between them. We can distinguish the following steps:

1) Identify text units that best define the task at hand, and add them as vertexes in the graph
2) Identify relations that connect such text units, and add them as edges between respective vertexes
3) Iterate graph-based algorithm until it converges
4) Sort vertexes by their score and use this score for selection decisions

Such definition allows us to apply this method to a wide range of problems. In particular, we will show how to apply this algorithm to key phrase extraction.

First, we have to decide what to use as units when building the graph. Authors suggested to this role nouns and adjectives. They tried different part-of-speech filters but this one showed the highest performance. Two lexical units were related if they co-occur within a window of maximum N words, where N can be set anywhere from 2 to 10. As a result, authors constructed an unweighted and undirected graph. The initial score for each vertex was set to 1. After this, authors ran the algorithm described earlier. Once a final score was obtained for each vertex in the graph, first T vertexes with the highest scores were selected. T may be set to any fixed value or may depend on various factors for instance, on a text size. Then, sequences of adjacent keywords are collapsed into a key phrases.

The dataset used in the experiments is a collection of 500 abstracts from the Inspec database, and the corresponding manually assigned key phrases. Results are shown on Tab. 1.

It should be noted, that due to algorithm definition, the number of vertexes in a graph depends on the number of distinct words in a text. So, for large texts algorithm can be very long to run.

| Precision | 32.1% |
| Recall | 36.2% |

Tab. 1. TextRank evaluations

B. PageRank on Synonym Networks

The problem of a graph growth can be partially solved by merging words into groups. This can also improve the efficiency of the algorithm. One way to form groups is to define equivalence classes. Authors suggest merging words by their meaning. PageRank on Synonym Networks[7] implements this idea. They need a thesaurus for that matter. Each class can be represented by a unique number. Every word in a text is replaced by a unique number that corresponds to the equivalence class of the word. Then the algorithm described in the previous paragraph is applied to the modified text.

C. RAKE

RAKE[2] (Rapid Automatic Keyword Extraction) is a relatively effective algorithm that operates on individual documents, easily applied to new domains and does not depend on the structure of a document or specific grammar rules. RAKE is based on the observation that key phrases frequently contain multiple words but rarely contain standard punctuation or stop-words. As a stop-words, we denote function words like "and", "of", "the", or other words with minimal lexical meaning. It is worth to mention that such words are also ignored in building an index in information retrieval systems. The algorithm requires the list of such words.

In addition to a stop-word list, sets of phrase- and word-delimiters needed. RAKE uses stop-words and phrase-delimiters to partition a document into a set of candidate key phrases. Word delimiters are used to split candidates into individual words.

RAKE begins key phrase extraction with partitioning a text into candidate key phrases. On the next step, the set of unique words is obtained by splitting candidates into individual words with word-delimiters. After this, scores are calculated for each word. Authors propose metrics based on word frequency and word degree. Word frequency \( f_{\text{req}}(w) \) is a number of occurrences of the word in candidates. Word degree \( d_{\text{eg}}(w) \) is a total length of all candidates which contain the word. The score of a word is defined as

\[
s(w) = \frac{d_{\text{eg}}(w)}{f_{\text{req}}(w)}
\]

Then a score is calculated for each candidate key phrase and defined as the sum of its member word scores.

Authors evaluated the efficiency of the algorithm on the Inspec set, the set of 500 abstracts with manually assigned key phrases. The same dataset was used in testing TextRank effectiveness. They also compared performance of the algorithm with different stop-word lists. The results are shown on the Tab. 2.
As we can see, the difference in the Precision with generated stop-word list based on keyword adjacency (KA stoplist) and with Fox’s stop-word list is quite noticeable. This means that RAKE is strongly depends on the provided set of stop-words.

<table>
<thead>
<tr>
<th>stop-word list</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>KA stop-word list</td>
<td>33.7%</td>
<td>41.5%</td>
</tr>
<tr>
<td>Fox stop-word list</td>
<td>26.0%</td>
<td>42.2%</td>
</tr>
</tbody>
</table>

Tab. 2. RAKE evaluations

D. Position Weighing

Position Weighing[6] is based on the fact that a word’s position plays an important role in linguistics. Words in different positions carry different entropy. For example, if a word appears in an introduction or in a conclusion it usually carries more information. This observation also applies to words within the same paragraph: words appeared in a leading and summarization sentences are often more important than those in other positions of the same paragraph.

Authors propose their approach called Position Weight. For each word position they suggest computing a score which strongly depends on a paragraph and a sentence where the word occurrence is found and on a form of the word. Here is the formula:

\[ pw(t_i) = pw(t_i, p_j) * pw(t_i, s_k) * pw(t_i, s_r) \]

Where \( pw(t_i, p_j) \) represents the score of an occurrence \( t_i \) within the paragraph \( p \); \( pw(t_i, s_k) \) represents the score of an occurrence \( t_i \) within the sentence \( s \); \( pw(t_i, s_r) \) represents the score of an occurrence \( t_i \) as a word for \( r \). Then the total weight of a term \( t \) in a document \( d \) computed as the sum of all position scores:

\[ PW(t, d) = \sum_{i=1}^{m} pw(t_i) \]

We can conclude from the definition of the algorithm that it relies on the structure of a document. Also, we should notice when assumptions about a document structure are violated (for example, there are no clues to determine a type of sentences and paragraphs), the algorithm is reduced to simply calculating the frequency of a word.

E. Statistical Approach

1) Weighing of Individual Words: In addition to the frequency of a word, one can also obtain information about the distribution of a word. The authors of the approach suggest using this information to determine the significance of individual words. The metrics proposed in [5] are based on the phenomenon of the attraction of keywords in the text. So, for example, the authors give the distribution of distances to the nearest neighbor for the word "NATURE" in the book "The Origin of Species" by Charles Darwin (Fig. 1). On the graph, you can see that the distribution of distances resembles an exponential distribution. In the case of noise words, this property is much weaker. Thus, keywords tend to form clusters in a text unlike noise words.

2) \( \sigma \)-index: To study the spatial distribution of a given word in a text, first, we denote by \( t_i \) the absolute position in the corpus of the \( i \)-th occurrence of a word. Thus we obtain a sequence of such positions: \( \{t_0, t_1, \ldots, t_{n+1}\} \), assuming there are \( n+2 \) occurrences. Next, we need to compute the average distance between two successive words:

\[ \mu = \frac{1}{n+1} \sum_{i=0}^{n} (t_{i+1} - t_i) \]

The following step is to obtain the standard deviation:

\[ s = \sqrt{\frac{1}{n+1} \sum_{i=0}^{n} ((t_{i+1} - t_i) - \mu)^2} \]

To eliminate the dependence on the frequency of occurrence for different words, authors suggest normalizing the token spacing, thus they define \( \sigma \)-index as follows:

\[ s/\mu \]

Given that standard deviation grows rapidly when inhomogeneity of a distribution spacing \( t_{i+1} - t_i \) increases. However, a value of sigma can be strongly affected by the change of a single occurrence position. Also, high values of sigma do not always imply a cluster concentration.

3) \( \Gamma \)-index: To solve such problems, authors suggest considering the average separation around the occurrence at \( t_i \). They define it as follows:

\[ d(t_i) = \frac{(t_{i+1} - t_i) - (t_i - t_{i-1})}{2} = \frac{t_{i+1} - t_{i-1}}{2} \]

. For each position \( t_i \) the cluster index \( \gamma \) is computed:

\[ \gamma(t_i) = \begin{cases} \frac{\mu - d(t_i)}{\mu}, & \text{if } d(t_i) < \mu \\ 0, & \text{else} \end{cases} \]

Finally, the score of a word is obtained from the average of all its cluster indexes:

\[ \Gamma = \frac{1}{n} \sum_{i=1}^{n} \gamma(t_i) \]

\( \Gamma \)-index is more stable than \( \sigma \)-index. However, it is more time-consuming to evaluate than \( \sigma \).
4) Extracting of Key Phrases: The metrics described above makes it possible to rank only individual words. In [4], proposed several ways to generalize metrics to two-word phrases:

First Way: Rank whole word sequences. The weight of the sequence is calculated similarly to a single word. The problem is that the frequency of the desired phrases may be insufficient for analysis.

Second way: Make up all possible unique phrases of a given length. To compute the score of a phrase we need to sum up all score of individual words from which the phrase consists. The problem with this approach is that the number of phrases grows exponentially with the given length.

### III. Experiments

#### A. Data Description

We used the book "Abel Theorem in Solutions Problems" written by Alekseev V.B. as a test data. The text consists of 39519 words 1576 of which are unique. The alphabetical index at the end of the book was used to evaluate the retrieval capabilities of different key phrase extractors. The choice of the test data is explained, firstly, by a large volume of the text, which allows us to give more accurate estimates of the efficiency of algorithms, and secondly, by the requirements of the statistical approach to a number of occurrences of analyzed words (for small texts this condition may not be fulfilled even for one word).

#### B. Getting Evaluations

We measured the efficiency of algorithms in terms of Recall and Precision. The result of each algorithm is a set of phrases. To obtain the Precision estimate, we must determine how many key phrases are in the result set. We will assume that the phrase from the result set is a key phrase if it fully contains any key phrase from the reference set and the order of words does not count. To obtain the Recall estimate, we must determine how many key phrases from the reference set are in the result set. Also, we will assume that the phrase from the reference set is in the result set if it is fully contained in any phrase in the result set without regard to the order of words. In addition, all the words in all phrases in both sets were stemmed, which solves the problem with comparing different forms of a word.

#### C. RAKE

The algorithm requires a set of stop-words. We used publicly available one from ranks.nl site. To avoid the problem with different forms of words We used SnowballStemmer from the nltk library for Python. Every word in every phrase was stemmed. For each stemmed phrase was kept the count of unstemmed ones from which it could be obtained. These numbers were used in computing scores for individual words. Results are shown on Tab. 3:

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31.8%</td>
<td>74.2%</td>
</tr>
</tbody>
</table>

Tab. 3. Evaluations for RAKE without any modification

The result set contained 4320 phrases and all these phrases were unique in terms of word forms. However, we faced the problem: the output contained long phrases, for example, the longest one consisted of 10 words. We would like to have shorter phrases. It was decided to set the limit for the maximum phrase length. All phrases with length above the limit were split into shorter ones. From words that make up the long phrase, all possible phrases of a given length were made without taking into account the word order. Also, the limit was set for the maximum degree of each word to support this changes.

Next, We studied the dependency of efficiency on the restriction on the length of a phrase. As we can see on the Fig.2 the highest precision was reached when the maximal phrase length was limited to 4 or 5 words. On Tab. 5 are shown exact evaluations for RAKE with phrase length limitations.

The following table shows the results:

<table>
<thead>
<tr>
<th>Max. length</th>
<th>Precision</th>
<th>Recall</th>
<th>NWords</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>40.8%</td>
<td>73.0%</td>
<td>9327</td>
</tr>
<tr>
<td>5</td>
<td>41.5%</td>
<td>73.6%</td>
<td>7293</td>
</tr>
</tbody>
</table>

Tab. 4. RAKE evaluations with phrase length limitations

It makes sense to consider only such restrictions on length, since at these limitations Precision reaches its highest values, with Recall close to maximum. It can be seen that the number of words extracted by the algorithm has increased dramatically. At the same time, the number of correctly extracted words also increased, because there is an increase in Precision.

This way of generating candidates, in theory, can cause an exponential growth of their number. By the length of a phrase we will understand the number of words it consists of. Let \( k \) be a restriction on the maximum length of a phrase, \( m \) - the maximum length of a phrase that hit the derivation of the algorithm. By the definition of the algorithm, phrases obtained after partitioning do not have intersections in the text, i.e. for any two different phrases, their positions in the text will not
overlap. Let \( N \) be the number of words in a text. Thus, the number of such phrases is not greater than \( \frac{N}{m} \). We assume that \( m > k \). When generating new phrases, the word order is not taken into account, so the number of new phrases can be calculated as the number of combinations without repetitions:

\[
C_{m}^{k} = \frac{m!}{(m-k)! \cdot k!}
\]

Thus, we can give the following estimate of the new number of phrases:

\[
\frac{N}{m} \cdot C_{m}^{k}
\]

As we can see, the growth is proportional to the number of combinations. By definition of the algorithm, the length of the key phrase in average can not be more than the average sentence length. The average sentence consists of 10-11 words[10], which means that the average value of \( m \) will not exceed this number. In this way, we can conclude that the increase in the number of candidates will be within reasonable limits for random texts.

Limiting the size of the result set, we can improve Precision of an algorithm. But discarding part of the results may lead to the decreasing of Recall. Fig. 3 shows the dependence of Precision and Recall on the number of selected candidates.

\[\text{Fig. 3. Comparison of estimates for RAKE with 4 and 5 words phrase length limitations. Graph shows dependence on the number of words with the highest ranks extracted}\]

\[\text{Fig. 4. Dependence of estimates for statistical approach on the number of words with the highest ranks extracted when phrase length is limited to two words}\]

\[\text{Fig. 5. Dependence of estimates for statistical approach on the number of words with the highest ranks extracted when phrase length is limited to three words}\]

D. Statistical Approach

The statistical approach provides the way of measuring the significance of individual words. Nevertheless, key phrases can consist of more than one word. In this connection, the question arises how to form a set of candidates. The solutions proposed in [4] showed good results, but they require a large amount of memory to store all candidates. For example, the number of all possible key phrases formed of two words (without taking into account the order) will be equal to the number of combinations. So, for the text used in the work, the number of unique words is around 1500, thus there will be around \( 10^6 \) phrases of two words and around \( 10^9 \) phrases of three words. Actually, there is no need to store all candidates as we only need first \( T \) of them with the maximal score.

Selection of candidates: We will consider the case of selecting phrases from two words. This problem can be addressed to the next. Given two sorted arrays \( A[1 \ldots n] \) and \( B[1 \ldots n] \). We want to print all \( n^2 \) sums \( A[i] + B[j] \) of two elements from different arrays in descending order. There is the solution to the problem that performs in \( O(n^2 \log(n)) \) time and requires \( O(n) \) space.
**Description of the solution of the problem of arrays:** We will use max-heap to store tuples consist of a sum \( A[i] + B[j] \) and a couple of indexes \((i, j)\) which defines such sum. Elements in the heap are ordered by the first element of a tuple.

At the beginning, heap stores all pairs such as: \( \forall j = 0 \ldots n : (A[0] + B[j], (0, j)) \).

The step of the algorithm is that the first element is extracted from the heap - this is the element with the largest sum. Suppose that indexes \((i, j)\) were associated with that sum. We print extracted sum and then put a new sum in the heap \( A[i + 1] + B[j] \) with the pair of respective indexes. When the elements of the arrays are finished, the remaining content of the heap is displayed. From the definition of the solution, the validity of estimates of time and space consumption is obvious.

**Correctness of the solution:** To prove the correctness of the solution it is handy to consider square matrix formed from all possible sums. We denote such matrix as \( C \) and it forms as follows: \( C[i, j] = A[i] + B[j] \). Since the elements of arrays \( A \) and \( B \) are sorted, we can notice that the elements of the matrix are decreasing from left to right and from top to bottom. In other words: \( \forall i, j : c[i, j] \geq c[i + 1, j] \) and \( c[i, j] \geq c[i, j + 1] \).

At each step, the following invariant is supported: the heap stores the maximal element of each column that have not been printed yet. Since the maximal element that should get into the output is in one of the columns, this proves the correctness.

**Efficiency of the algorithm:** Before calculating scores, each word was stemmed with SnowballStemmer taken from nltk version 3.2.2. Thus, words that differ only by their forms were merged into equivalence classes. We used Gamma-index in the experiments. Also, at the stage of selecting words for weighing, it was decided to use a filter along the length of the word: those words which length was less than the specified threshold did not participate in the weighing and in the construction of candidates. In the experiments with filtering the threshold value for the word length was set to 3. Experiments were carried out for phrases of two and three words.

On Fig. 4 and Fig. 5 is shown that the use of the filter have almost no impact on Recall and Precision of the algorithm.

The considered algorithm shows high Precision. This is due to the construction of candidates, Gamma-index accurately ranks individual words, hence top-scored ones are most likely to be actual key phrases or to be contained in multi-word key phrases. Because of the way of constructing candidates and due to the definition of efficiency metrics top-scored candidates are most likely to be treated as properly extracted key phrases.

Based on the results of the experiments, it can be concluded that the metrics Gamma-index correctly ranks individual words. Nevertheless, artificially constructed phrases can not ensure the high Recall of the algorithm.

**E. Hybrid approach**

The advantage of the statistical approach is the high Precision of the ranking of single words. And the main drawback is the complexity of constructing candidates and incapability to ensure high Recall with those candidates. On the other hand, the RAKE algorithm allows us to create phrases that provide a relatively high Recall, but proposed ranking formula does not allow to achieve high Precision while limiting the number of candidates. In this connection, the idea of using Gamma-index for ranking of a single word, but use phrases that are obtained as a result of the RAKE algorithm. Fig. 6 and Fig. 7 show a comparison of corresponding methods.
IV. CONCLUSION

In the work it was shown that, with certain additions, the RAKE algorithm and the statistical approach can be used to extract key phrases from Russian texts. In addition, the paper proposed new Hybrid method that uses $\Gamma$-index for weighing phrases that are obtained as a result of RAKE algorithm. And it was shown that the algorithm allows to achieve higher Precision and Recall comparing to other algorithms considered in the paper.

V. REFERENCES

Abstract—Automatic code synthesis has been attracting more attention lately. Some recent papers in this traditionally academic field even present results that could be applicable for industrial programmers. This paper provides an overview of Bayesian Sketch Learning (BSL) approach, describes basic concepts and workflow of a BSL synthesizer. Based on this we discuss an architecture of a configurable BSL synthesizer that could work as a part of an integrated development environment. We describe the implementation of such synthesizer for JVM platform and its integration with IntelliJ IDEA as a plugin. Two approaches to implement user interaction in a plugin like this are presented: method annotations and a domain-specific language. The paper concludes with an evaluation and a discussion on limitations of selected approach for industrial programmers.

INTRODUCTION

Automatic code synthesis has been a field of interest of computer science and software engineering for decades. It is defined as a task of synthesizing an algorithm in certain programming language based on incomplete specifications. Depending on the approach, this specification can be a definition of an algorithm in some domain specific language (DSL), a set of inputs and outputs of the algorithm, a set of system calls that occur when the algorithm is executed, etc.

Recently, more and more researchers have been addressing this task, and even first industrial code synthesizers begin to appear on the market [1]. However, most modern code synthesis tools are poorly applicable, since they usually require programmers to master some new formalism, almost always alienated from programmers' main field of knowledge. For example, study a separate specification language.

In this regard, Bayesian Sketch Learning approach (BSL approach), proposed by a group of researchers from the University of Rice in [2], is fundamentally different. BSL approach allows you to use identifiers of a programming language and its libraries as domain specific language. It narrows the gap between a specification language used by a code synthesizer and a synthesized code used by programmers. This approach has a significant impact: since the synthesizer works with nothing else than the programming language and its libraries, it can be used within an integrated development environment (IDE) and fit in the usual patterns of work with this IDE. Moreover, using such a synthesizer, an IDE could offer more intelligent code completion and even try to synthesize parts of developed systems.

This paper describes the implementation of a BSL synthesizer for JVM platform and its integration with IntelliJ IDEA. The reminder of this paper is structured as follows. Section I provides an overview of BSL synthesizer’s concepts, describes several alternative approaches to code synthesis and examines the architecture of Bayou which is the baseline implementation of the BSL approach. Section II presents an architecture of a configurable BSL synthesizer that could work as a part of an IDE. Section III describes implementation of such a synthesizer and its compatibility with Bayou models. Section IV explores challenges arising while implementing the synthesizer as an IntelliJ IDEA’s plugin. Section V provides evaluation of suggested implementation.

I. OVERVIEW

A. Overview of BSL synthesizer’s concepts

BSL synthesizer is a code synthesizer based on Bayesian Learning approach [3]. The essence of Bayesian approach in this case boils down to the following: a synthesizer is trained on a corpus of programs and so called “evidence” (some values associated with every program). After that, getting some set of evidence, it tries to synthesize code that most likely satisfies this evidence in context of the whole corpus. So, on the learning step an a priori distribution of programs is calculated for a given set of evidence, and on the synthesis step an a posteriori distribution is calculated for a specific evidence.

Evidence for a BSL synthesizer is usually method calls and classes used in the synthesized function. Experiments in [2] showed high effectiveness of BSL synthesizers for API-heavy code (code with a large number of API calls). Efficiency of BSL synthesizers in arbitrary code synthesis tasks has not been investigated yet.

BSL synthesizer uses sketches [4] as internal representation of programs. A sketch is a simplified representation of a program that consists only of basic language constructs (such as control flow and, in case of BSL synthesizers, API calls). Sketches do not preserve semantics of the program (which, nevertheless, can be recovered using probabilistic methods, see [2] for details), but they represent programs with similar intent in a uniform manner. Sketches are
associated with evidence using Bayesian encoder-decoder (BED) techniques [2] (fairly close to variational autoencoders [5]). An evidence is converted into an element of intent latent variable space (“encoded”), and this element is converted (“decoded”) into a sketch corresponding to the given evidence.

Thus, from the probability theory’s point of view, following calculations take place.

1) Let X be a collection of evidence, Z is an intent latent variable, corresponding to X, Y is a sketch corresponding to Z.
2) Calculate $P(Z|X)$ — a distribution of the hidden variable by X.
3) Sample Z in accordance with the distribution obtained.
4) Calculate $P(Y|Z)$ — a sketch distribution by Z.
5) Sample Y in accordance with the distribution obtained.

By combining Bayesian approach with representation of programs as sketches, BSL synthesizers can correctly synthesize fairly complex API-heavy methods.

B. Related works

Currently there are several tools that are capable of synthesizing API-heavy methods provided with evidence.

1) Bing Developer Assistant (BDA): Bing Developer Assistant [6] is a system for searching code samples or even entire projects corresponding to natural language queries. BDA consists of a Visual Studio plugin that consists of a user interface (frontend), and a cloud-based platform that provides search capabilities (backend). The frontend part uses natural language as a way of communicating with its users (for example, queries like “how to save png image”). After receiving response from the backend, BDA is able to insert code into current user’s project, open a separate window with a code sample, or even offer GitHub projects matching with the current request.

To search code using natural language queries the backend part uses Bing1, restricting itself to a limited set of sites. A framework proposed in [6] extracts code pieces from Bing search result pages and ranks them. Then code samples are passed to the frontend.

BDA is a successful tool to search code with 670 thousand downloads according to [6]. However BSL synthesizers are not just code search systems. BDA is not capable of generalizing and synthesizing code not known to it, but deducible from already known data.

2) Synthesizing API usage examples (SAU): Another interesting alternative is proposed in [7] — an algorithm for synthesizing code samples for Java Standard Library API. This algorithm takes an input type T (so called target type) for which it is required to synthesize usage examples, and code corpus where this type is used in some way. Synthesis is performed in several steps.

Initially existing methods using type T are enumerated, and via symbolic execution [8] a graph model is constructed, representing different ways of working with type T. Then, obtained use cases of type T are clustered. Clustering is performed using k-medoids method [9] based on the metric proposed in [7]. For each of the clusters obtained, an abstract element is synthesized — a representative of this entire cluster. Finally, all these abstract representatives are concretized into final code. This code might not be compilable: catching of checked exceptions or initialization of variables may be omitted, if exception handling in code corpus is usually performed outside of functions using type T or initialization is not important for the sake of synthesized examples.

SAU shows excellent results of synthesizing examples (82% of respondents did not see any difference between SAU generated code and human written code or even would prefer SAU examples) and has a decent generalizing capability. However, it is obvious that this algorithm is highly specialized for code examples synthesis. Unlike BSL synthesizers, SAU is not able to synthesize code that uses several types, and such support would require fundamental rework of the entire algorithm.

C. Bayou

Researchers from the Rice University also proposed their implementation of the BSL synthesizer, Bayou [2]. Bayou is a complex machine learning system employing several algorithms. It work in two modes: learning mode, when a statistical model is obtained, adjusting to a training sample, and synthesis mode, when code synthesis is performed in accordance with the statistical model.

Input is a set of sets of evidence — a set (possibly empty) for each type of evidence. The implementation supports three types of evidence:

- ApiCall — call of some API method (for example, “readLine”);
- ApiType — usage of some library class (for example, “BufferedReader”);
- ContextType — usage of some class as method argument.

During synthesis evidence is being passed through the following transformation layers:

1) Evidence Embedding Layer: evidence sets (for each evidence type separately) are converted into a numeric vector;
2) Evidence Encoder Layer: obtained set of vectors is encoded into an element of intent latent variable space (also a numeric vector);
3) Intent Decoder Layer: obtained element is converted into a sketch;
4) Combinatorial Concretization Layer: the sketch is turned into code using random walk technique.

Evidence Encoder and Intent Decoder layers together form Bayesian Encoder-Decoder (BED).

---

1Bing: a web search engine by Microsoft, URL: https://www.bing.com
These layers are only a general description of the synthesizer’s architecture. Depending on the task and its domain, different algorithms can be used within Evidence Embedding and Combinatorial Concretization layers, acceptable evidence types also might differ.

We will call a set of algorithms and acceptable evidence types a metamodel of the BSL synthesizer, and the result of learning a particular metamodel is a model of the BSL synthesizer. Currently there are two metamodels, for Android SDK and for Java STDlib, and a fairly large number of models for each of them.

Android SDK metamodel uses three types of evidence: ApiCall, ApiType and ContextType. This metamodel is optimized for models trained on Android SDK library and is described in detail in [2]. Java STDlib metamodel uses two types of evidence: ApiCall and ApiType. This metamodel is optimized for models trained on Java STDlib and is under development by researchers at this moment.

Now let us consider Bayou’s implementation. Bayou is implemented as two web servers: a Java server and a machine learning (ML) server. Input code is passed to the Java server (a servlet within a Tomcat web server), where it is parsed using Eclipse JDT and a set of evidence is retrieved. Then this set is passed to the ML server, which is implemented in Python. It runs embedding (using Scikit-learn), encoding and decoding steps (neural networks are implemented with Tensorflow). The resulting sketch is serialized and passed back to the Java server, where Combinatorial Concretization via Eclipse JDT is performed, and synthesized code is returned to the user.

Thus, reference implementation of Bayou’s BSL synthesizer is a ready-to-use web service for code synthesis, but it can not be integrated into IntelliJ IDEA as a plugin without significant rework. IntelliJ IDEA does not require installation of a Python interpreter, so most users might not have it installed. All layers implemented in Python must be re-implemented on the JVM platform. Moreover, current Bayou architecture does not support configuration of metamodels, each metamodel is implemented as a separate application.

II. ARCHITECTURE OF THE CONFIGURABLE BSL SYNTHESIZER

Unlike Bayou’s implementation, we want to build a configurable BSL synthesizer — a BSL synthesizer capable of running synthesis according to different metamodels. We provide a mapping of existing abstract BSL synthesizer’s layers to layers of our implementation. Each layer of our implementation can be parameterized with different algorithms, and such set of parameterizations of all layers, as we defined above, forms a metamodel.

Complete Data Flow diagram of the suggested configurable BSL synthesizer is shown in Fig. 1.

A. Input Layer

The purpose of Input Layer is to handle input data (some code with evidence): extract evidence from the code and create code’s intermediate representation for Concretization Layer. This intermediate representation will be used as context for synthesized code. There is no such layer in the abstract BSL synthesizer, its role here is strictly technical. Having input data processed in a separate layer, we can pass evidence into the synthesizer in different ways: embed it into code, or pass along with code and a position to insert synthesized code into.

B. Embedding Layer

Embedding Layer handles embedding the evidence received from Input Layer. Embedding Layer supports several algorithms required by different metamodels: TF-IDF and LDA for Android SDK and k-hot encoding for Java STDlib. This layer corresponds to Evidence Embedding Layer in the abstract BSL synthesizer’s architecture. It is parameterized by evidence types accepted by the metamodel and embedding algorithms for each evidence type. In case embedding requires some additional data (an LDA

Figure 1. Data flow diagram of the configurable BSL synthesizer

3Scikit-learn: a machine learning library for Python, URL: http://scikit-learn.org
3Tensorflow: a machine learning library for Python, URL: https://www.tensorflow.org/
model, a k-hot encoding dictionary, etc.) they are also added to layer’s configuration.

C. AST Synthesis Layer

AST Synthesis Layer performs synthesis of sketches’ ASTs by the embedded evidence. This layer can be parameterized with a number of hidden layers of encoder and decoder, however this is not used currently for existing metamodels.

This layer corresponds to both Evidence Encoder Layer and Intent Decoder Layer. Within the AST Synthesis Layer evidence is encoded into an intent latent variable element and than decoded from this element into a sketch. Two layers are combined into one because the change of parameters or models of one layer is impossible without corresponding changes to the other layer. AST Synthesis Layer is parameterized by the BED model.

D. Quality Measurement Layer

Quality Measurement Layer measures quality of synthesized sketches, deduplicates and ranks them. There is also no such layer in the abstract BSL synthesizer, here it is used to isolate quality measuring algorithms into individual abstractions. Thus, we can use different ranking algorithms for different metamodels, or even let our users choose which quality metrics to use.

E. Concretization Layer

Concretization Layer turns sketches obtained from Quality Measurement Layer and code’s intermediate representation received from Input Layer into Java code. This layer corresponds to Combinatorial Concretization Layer in the abstract BSL synthesizer’s architecture. Concretization algorithm has several parameters (for example, recursive search depth while synthesizing method arguments), however all existing metamodels use default settings.

It is worth noting that concretization is performed taking surrounding code in account, so Concretization Layer accepts not only an AST, but also an intermediate representation of the input source code, and uses this representation as context of synthesized code.

The result of Concretization Layer is a source code fragment with synthesized code block inserted into it.

III. IMPLEMENTATION OF THE CONFIGURABLE BSL SYNTHESIZER

Now let us consider implementation of the architecture presented above. We start with the way models for the configurable BSL-synthesizer were obtained.

A. Exporting the models

As mentioned earlier, researchers from the Rice University continue to develop Bayou and release improved models regularly. Therefore, it seems reasonable to re-use existing Bayou models and maintain compatibility with the new ones.

Depending on the metamodel, a model contains different data. Bayou’s source code was instrumented to export all this data in runtime. For Android SDK version of the Embedding Layer it was required to export LDA and TF-IDF models from Scikit-learn objects for each type of evidence. In case of TF-IDF, we export the dictionary and IDF matrix [10]. In case of LDA, it is \( \alpha, \eta \) values and \( \phi \) matrix [11]. For Java STDbil version of the Embedding Layer only k-hot vector dictionary was exported.

For the AST Synthesis Layer both metamodels required export of Tensorflow models. To achieve this, we name all variables and output tensors of the Tensorflow model (otherwise it will be difficult to access them when executing the model) and save the model using Tensorflow’s export capabilities.

B. Implementation of layers

Now we discuss implementation of each synthesizer’s layer.

1) Input Layer: This layer currently has two implementations. Both implementations use Eclipse JDT to parse input code and create its intermediate representation (as JDT’s CompilationUnits).

First implementation is quite similar to the one proposed in [2]. Evidence is passed directly within the code as library functions calls. Using Eclipse JDT, the code is parsed, evidence is extracted, and its position is marked for subsequent code insertion. The evidence itself is removed from the code.

Second implementation separately takes input code, an evidence and a position to insert synthesized code into. It has a slightly more convenient API, and this implementation is a little bit more efficient, since it does not need to extract evidence from input code.

2) Embedding Layer: This layer also has two implementations: k-hot encoding for Java STDbil metamodel and TF-IDF + LDA for Android SDK metamodel. The layer even allows to use different embedding algorithms for each type of evidence, but existing metamodels don’t use this at the moment. All embeddings are parameterized with Bayou’s exported models and satisfy specifications of corresponding Scikit-learn algorithms.

The number of output dimensions for each embedding is defined by embedding’s model (for example, k-hot vector’s length will be equal to the dictionary’s length).

3) AST Synthesis Layer: This layer has a single implementation. It is parameterized with the exported Tensorflow model. The model itself is loaded and executed using Tensorflow for Java.

At the encoding step, evidence vectors received from the Embedding Layer are passed to the corresponding inputs of BED encoder and the Tensorflow model is executed. The result is the element of intent latent variable space (\( \psi \)).

At the decoding step, this \( \psi \) element is passed to the BED decoder (which is again a Tensorflow model). As a
response, the decoder returns a vertex — start of a production path, which is described in [2] in detail. Then, in accordance with the algorithm, an entire production path is constructed. On each step of this algorithm, depending on the given node, the construction of the production path can branch out in several directions. For example, “DBranch” node (the “if” node) will build paths for a predicate, “then” and “else” branches. At the moment all models support only “if”, “while” and “try ... catch” control flow constructs.

AST Synthesis Layer’s result is a set of sketches’ ASTs, their desired number can be specified in layer’s configuration.

4) Quality Measurement Layer: This layer has several implementations and their number continues to grow. Currently following algorithms are used for all metamodels: presence verification for all evidence, deduplication and ranking ASTs based on their occurrence frequencies. Evidence presence verification walks through an AST and collects API calls, types and contexts (types of method arguments) present in the current sketch candidate. The result is compared with the original evidence set. If some evidence is missing, this candidate sketch is removed from the result. Deduplication removes duplicate ASTs and counts the number of occurrences for every individual AST. Occurrences frequency is used for ranking: more often an AST is met, more likely it will fit a given query.

However, this naive algorithm is not the only option. We can also rank ASTs using standard code metrics, such as LOC (lines of code), cyclomatic complexity [12], or more complex ones. Depending on task’s domain, these metrics can be selected to provide more relevant results (for example, LOC could be used for generating examples, which are preferred to be short according to [7]).

5) Concretization Layer: Finally, Concretization Layer performs synthesis of the program. This layer has a single implementation based on Eclipse JDT.

Concretization Layer takes an AST and code’s intermediate representation as input. In our case the intermediate representation is an Eclipse JDT CompilationUnit object. The concretization algorithm described in [2] is executed on this CompilationUnit object. Then, combinatorial search is performed guided by several heuristics. For example, functions without arguments are examined earlier than functions with arguments, since functions with arguments produce further search.

It is worth noting that current implementation is able to synthesize a method, some arguments of which can not be synthesized within current context. In such cases, a variable of an appropriate type is created and initialized with a null value. It is assumed that the programmer will replace it with an appropriate initialization.

Finally, unreachable code or code that does not affect the result of the function is removed (i.e. Dead Code Elimination is performed), the code is formatted, fully qualified class names are converted into simple class names and import expressions. Resulting code fragments are returned according to corresponding rank order of their ASTs.

IV. IMPLEMENTATION OF THE INTELLIJ IDEA PLUGIN

This section describes an IntelliJ IDEA plugin providing user interface to the implemented BSL synthesizer. Firstly, we mention several issues arising from such integration that need to be resolved to make the synthesizer work in IDEA’s environment.

A. Integrating BSL synthesizer with IntelliJ IDEA

There were two main challenges integrating implemented BSL synthesizer into an IntelliJ IDEA’s plugin: obtaining models and progress indication.

As mentioned before, a BSL synthesizer needs models to perform. Unfortunately these files are quite large (~100 Mb for the Android SDK model and ~200 Mb for the Java STDLlib model), so we can’t distribute them with the plugin directly. Currently, these models are stored in an Amazon S3 repository along with a descriptor file listing all supported models, paths to corresponding files and MD5 hashes of these files. Each instance of the plugin creates a directory for local repository. If the requested model is missing on disk or corrupted it will be downloaded from the remote repository.

Another issue is that code synthesis can take up to 10 seconds, and downloading the models could take even longer, so the lack of progress indication during such tasks will result in quite negative user experience. To handle progress indication the synthesizer subsystem accepts a special object: it is a data class object which fields are used for storing currently executed process (e.g. “Generating Sketches”, “Downloading TF-IDF Model”, etc.) and its progress as a double from 0.0 to 1.0. Based on this object’s state the UI thread shows and updates a progress indicator when time-consuming operations take place.

B. User interface

User interface for IntelliJ IDEA’s plugins could be implemented in a number of ways: using special window dialogs, with a separate DSL to use in comments, or as some language elements within Java code itself. We decided to create two alternatives: an approach based on method annotations defining required evidence (the same approach is employed by numerous popular Java libraries, for example, Project Lombok) and an approach based on a comment-based DSL.

4Project Lombok: a library for generation of utility methods in Java classes. URL: https://projectlombok.org/
1) Method annotations: We have implemented a Java library containing following annotations:

- **BayouSynthesizer** — annotation defining a metamodel to use;
- **ApiCall** — annotation defining an API call evidence;
- **ApiType** — annotation defining an API type evidence;
- **ContextType** — annotation, defining a context type evidence (type of an API call argument).

Let us consider a request to the synthesizer, which contains an “ApiCall” evidence with a value “readLine” and an “ApiType” evidence with a value “FileReader”. Using method annotations approach this query looks like this:

```java
import tanvd.annotations.*;
import java.io.File;
import java.io.FileReader;
public class TestIO {
    @BayouSynthesizer(type = SynthesizerType.StdLib)
    @ApiCall(name = "readLine")
    @ApiType(name = FileReader.class)
    void read(File file) {
    }
}
```

ApiType and ContextType annotations accept an object of type `Class<T>` — class of a Java class and for them IntelliJ IDEA automatically performs code completion. But because of Java constraints, ApiCall annotation cannot accept objects of type `Function<T>`. Hence, ApiCall should accept either a String or an enumeration, previously created and describing all possible API calls of the current model. For Java STDLib, this kind of enumeration can not be created because the number of available API calls exceeds maximum enumeration size. And obviously, there is no auto-completion for String values in IntelliJ IDEA.

Another issue with this approach is the lack of strict typing. For example, it is possible to specify BayouSynthesizer as StdLib (Java STDLib metamodel is chosen), and code completion for all of evidence types values is active. Moreover, IntelliJ IDEA allows to extend language’s grammar to show custom messages for some specific grammar errors.

The resulting language is quite simple. Strict typing and error prompts make it easy to learn even without additional documentation.

2) Domain Specific Language: The DSL was implemented using Grammar Kit\(^7\). It contains the following identifiers:

- **STDLIB** or **ANDROID** — initial identifier, defines a metamodel to use;
- **API** — identifier for an API call evidence;
- **TYPE** — identifier for an API type evidence;
- **CONTEXT** — identifier for a context type evidence (type of an API call argument).

The same query for a “readLine” call and a “FileReader” type evidence will look like this using DSL approach:

```java
import java.io.File;
import java.io.FileReader;
public class TestIO {
    /*
    STDLIB
    API:=readLine
    TYPE:=FileReader
    */
    void read(File file) {
    }
}
```

The domain specific language integrates well with IntelliJ IDEA. Strict typing is available (for example, CONTEXT can not be used when Java STDLib metamodel is chosen), and code completion for all of evidence types values is active. Moreover, IntelliJ IDEA allows to extend language’s grammar to show custom messages for some specific grammar errors.

The resulting language is quite simple. Strict typing and error prompts make it easy to learn even without additional documentation.

V. Evaluation

Our plugin was tested on a number of generation tasks mentioned in [2] for Android SDK and on a set of examples similar to those that are presented on http://www.askbayou.com/ for Java STDLib. In total we have tried 20 generation tasks for Android SDK (e.g. code that works with “BluetoothSocket”, code that works with “File”, etc.), and 20 generation tasks for Java STDLib (code working with Collection classes, “File” and different “Readers”). All tasks were executed on latest builds of corresponding Bayou applications and on our system. The results show that our implementation is equivalent to Bayou’s: in all cases both synthesizers produced the same result.

For instance, the following is a piece of code generated from the example above that used “readLine” and “FileReader” as evidence:

```java
import java.io.File;
import java.io.FileReader;
public class TestIO {
    void read(File file) {
        FileReader fr1;
        String s1;
        BufferedReader br1;
        try {
            fr1 = new FileReader(file);
            br1 = new BufferedReader(fr1);
        }
    }
}
```

\(^7\)Grammar Kit: a tool for custom language support for IntelliJ IDEA, URL: https://github.com/JetBrains/Grammar-Kit
while ((s1 = br1.readLine()) != null) {}
br1.close();
} catch (FileNotFoundException _e) {
} catch (IOException _e) {
} return;
}

This example shows that the synthesizer is able to generate code with non-trivial logic and could be applied to generate supplementary functions working with libraries.

A. Known limitations

Surely, BSL synthesizers have their limitations. First of all, they are not capable of generating code using two metamodels simultaneously. For example, it is not possible to generate code using both methods existing only in Android SDK and methods that exist only in Java STDLib. The only way to do this is to create a new metamodel including the other two. As far as we understand, there is also no research available showing how synthesis’ quality depends on the growth of generated programs’ space. An educated guess is that the quality will drop.

Secondly, BSL synthesizers are currently applied to generate only parts of methods or whole methods. BSL synthesizers could not be used to generate a whole class or even a whole project. As far as we know, there is no published research on generating pieces of code larger than methods using BSL synthesizers. However, they are a suitable tool to generate API-heavy code, which is confirmed by evaluation in [2] and our experiments.

CONCLUSION

In this paper we present a configurable BSL synthesizer compatible with Bayou models. It was implemented as a plugin for IntelliJ IDEA providing two types of user interface: Java annotations and built-in DSL. Our evaluation shows that qualitatively and quantitatively our implementation complies with Bayou’s experimental results presented in [2].

As future work we plan to improve plugin’s user interface, prepare models for some other libraries (e.g. IntelliJ Platform SDK) and improve IDEA’s code completion subsystem using synthesizer’s results providing users with a tool to generate chains of API method calls. Furthermore, we plan to integrate our BSL-synthesizer with DeepAPI tool [13] to create a system capable of synthesizing code from natural language queries.

REFERENCES

Использование SMT-решателей для анализа систем ограничений на .NET-типы

А.В. Мисонижник
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: misonijnik@gmail.com

Д.А. Мордвинов
JetBrains Research
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: dmitry.mordvinov@se.math.spbu.ru

Аннотация — Точный анализ поведения объектно-ориентированного кода в изоляции от точек его вызова может потребовать рассуждений о взаимоотношениях открытых типов. В работах Кеннеди и Пирса подробно изучены номинальные системы типов с вариантомностью, показана неразрешимость подтипования закрытых типов и предложены ограничения на систему типов, делающие задачу разрешимой. Открытые типы, тем не менее, остаются неизученными, и все еще не существует разрешающей процедуры для задачи выполнимости системы ограничений на открытые типы. В данной работе представлен алгоритм решения систем ограничений на типы в платформе .NET. Алгоритм системы ограничений 𝑇 𝑖𝑠 𝐴 к задаче выполнимости предложений логики первого порядка. Исследована его завершаемость и показано, что результатирующее логическое кодирование принадлежит к разрешимому фрагменту логики первого порядка.

I. Введение

При статическом анализе функций, написанных на объектно-ориентированных языках программирования, нередко возникает необходимость проверки выполнимости некоторой системы ограничений на типы и типовые переменные. К примеру, если в некоторой точке исполнения программы известно, что типовая переменная 𝑇 — подтип класса A, и тип 𝐵 — брат 𝐴 в иерархии наследования, анализатор может определить недостижимость из нее вектор, защищенного условием 𝑇 𝑖𝑠 𝐵; для этого достаточно обнаружить противоречивость системы ограничений 𝑇 𝑖𝑠 𝐴 ∧ 𝑇 𝑖𝑠 𝐵.

Для современных объектно-ориентированных языков (таких как C# и JAVA) задача определения выполнимости системы ограничений на типы является особенно нетривиальной. Во-первых, во время анализа большинства программ могут возникать объемные системы ограничений, являющиеся достаточно хитрой комбинацией элементарных ограничений и их отрицаний, соединенных логическими связками. Во-вторых, ситуация довольно сильно усугубляется сложностью самих систем типов: как известно, в номинальных системах типов с вариантомностью даже задача определения, является ли один закрытый тип подтипом другого, неразрешима [10], а обобщения в JAVA тюринг-полны [7]. Следует также заметить, что в случае платформы .NET известно разрешимое сужение системы типов.

Хороший алгоритм решения систем ограничений также должен гарантированно останавливаться и выдавать корректный ответ, если все типы в системе ограничений относятся к этому сужению.

Наконец, во время анализа библиотечной функции в изоляции от проекта, ее использующего, может быть известна только часть таблицы классов. К примеру, предположим, что класс Base, как и его потомок Derived и все классы в иерархии между Base и Derived, не реализуют интерфейс IComparable; предположим также, что в известной анализатору части таблицы классов не существует типов, наследующих класс Base и реализующих интерфейс IComparable одновременно. Рассмотрим теперь систему ограничений T is Base ∧ T is IComparable. Очевидно, что несмотря на отсутствие классов-кандидатов на роль T в известной таблице типов, такая система ограничений в системе ограничений все еще выполнима — достаточно ввести новый тип, наследующий Base и реализующий IComparable. Полезной в таком случае может служить возможность автоматического конструирования объявлений типов-кандидатов на роль T: к примеру, верификатор может использовать такие объявления для автоматической генерации исполнимого файла, воспроизводящего ошибочную трассу в библиотеке. Тем не менее, при добавлении ограничения Derived is T, система становится противоречивой, но единственный способ доказать это — перебрать все базовые классы класса Derived.

Необходимость учета всех подобных деталей вместе со сложностью системы типов анализируемого языка делает задачу анализа ограничений на типы крайне громоздкой и сложной. К тому же удобно было бы иметь возможность комбинировать ограничения на типы с другими условиями (например, арифметическими). К счастью, существуют SMT-решатели — инструменты эффективного поиска моделей формул логики первого порядка, в которой некоторым символам приписаны особые интерпретации [1, 4]. Современные SMT-решатели поддерживают большое количество теорий, включая линейную арифметику, алгебраические типы данных, непереопределяемые функции и т. д.

В данной работе представлен алгоритм решения системы ограничений на типы в платформе .NET. Ал-
горит кодирует систему ограничений в предложение логики первого порядка и использует SMT-решатель для поиска ее моделей. Показано, что для разрешимого фрагмента системы типов .NET алгоритм кодирования всегда останавливается и выдает предложение из разрешимого фрагмента логики первого порядка. Модели, получаемые от SMT-решателя, содержат логическую интерпретацию для отношения подтипирования и могут быть использованы для конструирования объявлений типов, явно не присутствующих в части таблицы классов, отправленной в решатель. Насколькo известно авторам, это первая работа, обсуждающая применение SMT-решателей для анализа системы типов объектно-ориентированного языка во всей ее полноте. Несмотря на заточенность данной работы под систему типов .NET, подход может быть адаптирован для других объектно-ориентированных языков со сложной системой типов (таких как Java и Scala).

II. Обзор существующих работ

Существует большое количество работ, посвященных статическому анализу различий систем типов. Среди них есть и такие, в которых анализ осуществляется путем кодирования типов в формулы языка логики первого порядка и анализа их выполнимости при помощи SMT-решателей [2, 3, 6, 8, 11, 12, 15, 16].

Например, в работе [11] подробно описывается кодирование неамортных обобщенных типов в логику первого порядка с использованием логического представления алгебраических типов данных. Но в ней никак не затрагивается кодирование отношения подтипирования. Напротив, в работе [16] предложен способ аксиоматизации логики первого порядка с номинальной типизацией, в которой есть аналоги типовых операторов (is, as, оператор преобразования типа). Однако её недостаточно, чтобы задекорировать систему типов .NET с её структурными элементами.

Важной вехой является статья Кеннеди и Пирса, которая исследует подтипирование в номинальных системах типов и систему типов .NET в частности [10]. В работе показано, что отношение подтипирования в таких системах неразрешимо даже для закрытых типов и предъявлены несколько разрешимых фрагментов. Однако задача подтипирования для открытых типов является более общей, и к ней нужно применить подход, описанный в [10].

III. Система типов .NET

В своей основе система типов .NET является номинальной (см. ниже), однако в ней имеются и элементы структурной системы типов: к примеру, обобщение интерфейсов и делегатов могут явно определять вариантовность своих типовых параметров, а массивы ковариантны по типу своего элемента.

Часто обозначения и правила вывода для отношения :: используются в данной главе, взяты из работы [10].

Типы (обозначим их как T, U, V и W) можно разделить на два вида: типовые переменные X, Y и Z и сконструированные типы C<T>, где C — это конструктор типа, а T — упорядоченный список аргументов типов. Закрытые будут называть типы, которые в листьях дерева конструкторов не содержат типовые переменные. Открытые назовём всё типы, которые не являются закрытыми. Высотой типа будем называть высоту дерева его конструкторов. Обозначим высоту функцией height.

В рамках данной работы можно считать, что все типы являются подтипами System.Object: типы указательей здесь рассматриваться не будут из-за тривиальности их подтипирования.

Номинальность системы типов означает, что отношение подтипирования явно определяется в коде программы и может быть представлено в виде конечной таблицы классов. Каждая запись в таблице классов для системы типов .NET имеет следующий вид:

\[ C^*<X> :: T_1, \ldots, T_n \]

Для каждого типового параметра имеется запись, которая ограничивает его:

\[ C^#i^* :: U_1, \ldots, U_k \]  \hspace{1cm} (1)

Без потери общности можно считать, что в таблице классов нет двух типовых переменных с одним названием (иначе переменные их).

Левая часть каждой записи в таблице классов содержит аннотацию типа, помещаемую вместо \( * \). Аннотация указывает, является ли тип интерфейсом (в таком случае будет записана буква \( I \)), типом значения (\( V \)), ссылочным типом (аннотация взята на работу \( S \)), оператор преобразования типа (аннотация взята на работу \( C \)), запечатанным типом (аннотация взята на работу \( C \)).

Например, для System.Object, System.IComparable и структуры S (не реализующей ни одного интерфейса), записи будут выглядеть следующим образом:

\[
\begin{align*}
\text{System.Object} & \quad C() :: \quad \text{System.Object} \\
\text{System.IComparable}^{i} & \quad :: \quad \text{System.Object} \\
S & \quad :: \quad \text{System.ValueType}
\end{align*}
\]

Символ :: обозначает отношение номинального подтипирования. Дополнительно также \( C<T> :: [U/X]T \). Через :: обозначим его транзитивное замыкание. Отношение :: обозначает номинальность системы типов, но его не достаточно, чтобы выразить структурные элементы. Для отражения вариантовности, потребуем от записей таблицы классов иметь вид

\[ C^*<v_1X_1, \ldots, v_mX_m> :: T_1, \ldots, T_n, \]

где \( v_i \) определяет вариантовость по \( i \)-тому типовому параметру и может принимать следующие значения:
(инвариантность), + (ковариантность), − (контравариантность). Положим \( C#i = X_i \) и \( \text{var}(C#i) = v_i \).

Заметим, что в записях вида (1) содержится символ подтипирования \(<:\) вместо символа номинального подтипирования. Отношение подтипирования в .NET будет строго определено ниже.

Примеры 1, 2 и 3 демонстрируют таблицы классов для трех фрагментов кода на языке C#.

Пример 1.

```csharp
interface IBuilder<out Z> where Z : IComponent {}
interface IComponent {}
sealed class Symbol {  
    IComponent, IBuilder<Symbol> {}
}
abstract class Line:  
    IComponent, IBuilder<Line> {}
class Combo<X, Y>  
    where X : IComponent, IBuilder<X>  
    where Y : IComponent, IBuilder<X> {}
```

```
Z <: IComponent
IBuilder<+ Z>: System.Object
Symbol C() <: IComponent, IBuilder<Symbol>
Line C <: IComponent, IBuilder<Line>
Combo C()<X, Y>: System.Object
X <: IComponent, IBuilder<X>
Y <: IComponent, IBuilder<X>
```

Пример 2.

```csharp
interface IBox<in Z> {}
class Thing: IBox<IBox<Thing>> {}
```

```
Z <: System.Object
Thing C(): IBox<IBox<Thing>>
```

Пример 3.

```csharp
class Rec<Z> {}
class MutualRec<X, Y> where X: Rec<Y>  
    where Y: Rec<X> {}
```

```
Z <: System.Object
MutualRec C()<X, Y>: System.Object
X C: Rec<Y>
Y C: Rec<X>
```

Наложим ограничения на таблицу классов:
1) отношение \(<::\) должно быть ацикличным;
2) записи в таблице должны быть корректными относительно вариантовности: например, запрещено
   \( B^<1 < - X> <:: ... \)
   \( A^<1 + X> <:: B^<1 + X> \)
3) запечатанному типу (в т.ч. делегату или массиву) разрешено пользоваться в правой части записи только в качестве типового аргумента;
4) в правой части любой записи может стоять максимум один тип с аннотацией \( C \) (множественное наследование запрещено);
5) аннотации не должны противоречить естественными ограничениями .NET (интерфейс не может быть запечатанным или иметь конструктор без аргументов, типы значений обязательны запечатаны и имеют конструктор без аргументов и т.д.).

Определение 1. Расширением таблица классов \( CT \) называется таблица классов, в которой содержатся все записи из \( CT \), а все левые части прочих записей содержат лишь конструкторы и их типовые переменные, не входящие ни в одну из левых частей \( CT \).

Наконец, можно определить отношение подтипирования с учетом вариантовности.

Определение 2. Отношение подтипирования для закрытых типов \(<:\) задается следующими правилами:

```
(Var) for each \( i \) \( T_i <:: \text{var}(C#i) U_i \)
(C<>) \( C<X> <:: V \) \( [T/X] V <:: D<U> \) \( C \neq D \)
```

```
(Super) \( C<T> <:: D<U> \)
```

```
\( \begin{align*}
    T <:: U & \quad \frac{T <:: U}{T <::+ U} \\
    T <::< D & \quad \frac{T <:: D}{T <::- U} \\
    U <:: T & \quad \frac{T <:: D}{U <:: T}
\end{align*} \)
```

Предложение 1. Отношение подтипирования \(<::\) разрешено для закрытых типов, при условии, что таблица классов будет нерасширяемой.

За определением нерасширяемости таблицы классов и доказательством Предложения 1 читатель отсылается к работе [10]. Из этого следует разрешимость подтипирования для закрытых типов .NET, т.к. по спецификации любая расширяемая таблица классов отклоняется средой выполнения [5].

Другими словами, работа [10] доказывает существование алгоритма, принимающего на вход любую таблицу классов (прошедшую валидацию средой выполнения .NET) и утверждение вида \( T <:: U \), где \( T \) и \( U \) — закрытые типы. Данная работа решает аналогичную задачу, но для открытых типов.

Очевидно, вопрос \( T <:: U \) в случае открытых типов сам по себе не имеет смысла: уже для двух типовых переменных на вопрос \( X <:: Y \) можно дать как утвердительный, так и отрицательный ответ; такая постановка имеет смысл лишь при связывании типовых переменных кванторами. Естественной, поэтому, будет формулировка задачи в терминах логики первого порядка.
IV. Алгоритм
На протяжении всей главы предполагается, что имеется и зафиксирована таблица классов $C_T$.
Рассмотрим язык первого порядка $L_4$ над сигнатурой $(\varphi_1, \varphi_2)$. Здесь

- $\varphi_1$ — множество функциональных символов, отождествляемое с множеством конструкторов типов.
- Для удобства применение функционального символа $C$ к аргументам $T$ будет по-прежнему записываться как $C\langle T \rangle$.
- $\varphi_2 = \{<:\}$, где $<:$ является двухместным предикатным символом и будет записываться в инфиксном стиле. Для удобства, формулы вида $\neg(T <: U)$ и $\neg(T = U)$ будут записываться как $T :: U$ и $T = U$.

Данная глава описывает алгоритм решения следующей задачи. По натуральному числу $h$ и данной формуле $\psi_1 \in L_4$ без кванторов необходимо построить такую модель $M = \langle \varphi_1, \varphi_2 \rangle$, что $M \models c_3(\psi_1)$, либо доказать, что такой модели не существует. Однако на $M$ должен накладываться набор ограничений:

- носитель $D_1$ должен быть подмножеством множества замкнутых типов;
- для каждого $C \in \varphi_1$, $\sigma(C)(T)$ должен конструировать замкнутый тип $C\langle T \rangle$;
- $\sigma(C) <: D_1 \times D_2$ — отношение подтипирования из Определения 2, заданное некоторым расширением таблицы классов $C_T$, на $c_3(\psi_1)$ — экстенсивное замыкание формул $\psi_1$ (т.е. формулы $\exists x_1, \ldots, x_n. \psi_1$, где $x_1, \ldots, x_n$ — свободные переменные в $\psi_1$);
- $\text{val}_t$ — оценка переменных $f(v(\psi_1))$ и $\max_{T \in \text{val}_t(f(v(\psi_1)))} \text{height}(T) \leq h$;
- семантика равенства стандартна.

Пример 4. Рассмотрим системы ограничений (формулы $\psi_1$) для таблиц классов из примеров 1—3:

1) $X <: IBUILDER<Symbol> \land X <: Line \land Line <: Y$ (выполняется, если $X$ — новый тип, а $Y$ — Line)

2) $\text{Thing} <: IBox<\text{Thing}>$ (невыполнимо из-за зацикливания правил вывода)

3) $X :: Y \land Y :: X$ (выполняется, если $X$ и $Y$ — новые типы)

Основным препятствием для применения решателей логики первого порядка к данной задаче является наличие ограничений (и притом нетривиального) на искомую модель. Алгоритм решает эту проблему, переводя формулу $\psi_1$ на другой язык первого порядка $L_m$, для которого уже может быть применен решатель. Перевод осуществляется в несколько шагов: (1) замыкание множества типов, (2) расширение отношения подтипирования, (3) формирование аксиом частичного порядка, (4) запрет множественного наследования, (5) формирование ограничений на замещаемые типы, (6) формирование ограничений на ссылающуюся, (7) формирование ограничений на конструкторы без параметров, (8) формирование ограничений на типовые параметры и (9) формирование окончательной формулы. Шаг 2 предполагает наличие предиката-оракула groundSubtype, который для пары замкнутых типов определяет: является ли первый подтипом второго, используя таблицу классов $C_T$.

Далее будет дано общее описание языка $L_m$ и детальное описание каждого шага алгоритма, проанализирована разрешимость выполнимости формул языка $L_m$, выдаваемых алгоритмом, а также представлен способ преобразования модели $M_m$ обратно в модель $M$. Доказательство корректности сведения, хоть и представляет интерес, опущено из-за недостатка места.

A. Язык $L_m$

Язык $L_m$ описывается сигнатурой $(\varnothing, \varphi_m)$, где $\varphi_m = \{<:, \text{subclass}, \text{con}, \text{interface}, \text{hple}\}$. Здесь $<:$, subclass, con — двухместные предикатные символы, a interface и hple — одноместные. За $\psi$ обозначим счетное множество предметных переменных, в идентификаторах которых разрешим использовать «<» и «>».

B. Замыкание множества типов

Определение 3. Множество типов $S$ будем называть замкнутым относительно декомпозиции, наследования и ограничений (далее просто замкнутым), если

(a) $C\langle T \rangle \in S 
\Rightarrow \forall i, T_i \in S$,

(b) $T \in S \land T <: V_1, \ldots, V_m \Rightarrow \forall i, V_i \in S$,

(c) $X \in S \Rightarrow \forall i, U_i \in S$, где $U_i$ — элементы правой части в записи таблицы классов $X <: U_1, \ldots, U_n$. Замыканием множества $S$ (обозначается $c(S)$) будем называть минимальное замкнутое множество, содержащее $S$.

Теорема 1. Для нерасширяющихся таблиц классов замыкание конечного множества типов конечно.

Доказательство. В доказательстве Теоремы 9 в работе [10] показан аналогичный факт для замыкания относительно декомпозиции и наследования (т.е. без учета $\psi$ Определения 3). Обозначим за $c^{a,b}(A)$ замыкание $A$ относительно декомпозиции и наследования.

Пусть теперь дано конечное множество $S$. Заметим, что взятие замыкания $c(S)$ аналогично выполнению двух шагов до достижения неподвижной точки:

1) вычисление $S' = c^{a,b}(S)$;

2) вычисление нового $S = S' \cup U$, где $U$ — множество всех элементов всех правых частей записей таблицы классов вида $X <: U_1, \ldots, U_n$, где $X \in S'$. Итак, если $S$ — конечное множество, то $S'$ — также конечное множество, что влечет конечность нового множества $S$ (к $S'$ добавляется конечное множество записей $U$, т.к. количество типовых переменных в $S'$ конечно). Наконец, заметим, что каждая итерация алгоритма добавляет к $S$ лишь типовые переменные, явно содержащиеся в таблице классов. Но т.к. таблица
классов конечно, количество итераций также конечно, что и дает нам конечность $c(S)$.

Определим множество $S'$, как множество всех тер- 

Далее надо определить функцию newVars, зависи-

Для этого введём новые обозначения:

- $CS$ — множество не пульарных конструкторов из
таблицы классов;
- $R$ — максимальная арность конструкторов из таб-
лицы классов $CT$;
- `newVar(C, n)` — функция, возвращающая множе-
ство типов, определенная следующим образом:

\[ newVar(C, n) \overset{\text{def}}{=} \{ C \times X^i \}_{i=1}^n, \]

где все типовые переменные $X_j^i$ различны;
- $count(h, R)$ — функция, возвращающая макси-
мальное число раз, которое можно использовать
один и тот же конструктор арности $R$ из множе-
ства $CS$ для построения типа высоты $h$:

\[ count(h, R) \overset{\text{def}}{=} \begin{cases} 
0, & h = 1 \land R \neq 0 \\
1, & h = 1 \land R = 0 \\
\frac{R^h - 1}{R - 1}, & \text{otherwise}
\end{cases} \]

Теперь можно определить функцию newVars и ре-
зультат её применения к выходным данным $CT$ и $h$:

\[ newVars(CT, h) \overset{\text{def}}{=} \bigcup_{C \in CS} \text{newVar}(C, count(h, R)) \]

\[ NV \overset{\text{def}}{=} \text{newVars}(CT, h) \]

Обозначим за $S$ объединение множеств $S'$ и $NV$. Так как множество $S$ будет конечным, по Теореме 1 его замыкание $cl(S)$ также будет конечным. Выберем произвольное инъективное отображение $\tau : cl(S) \rightarrow \mathcal{V}$. Обозначим за $FV$ образ $\tau(cl(S))$. Очевидно, $\tau$ задает взаимно-однозначное соответствие между $cl(S)$ и $FV$.

Пусть предикат $inCT(x)$ истинен, если главный кон-
структор аргумента принадлежит оригинальной таб-
лице классов, и ложен иначе. Другими словами, для 
в $FV$, $inCT(x)$ истинен, если $x = v$ и $\tau^{-1}(v)$ является применением какого-либо функционального символа в $\mathcal{L}_t$, и ложен в ином случае:

\[ inCT(x) \overset{\text{def}}{=} \bigvee_{T \in \text{cl}(S)} x = \tau(T) \]

Фактически, такое определение не гарантирует, что $inCT$ охватывает все конструкторы из таблицы классов $CT$, а только те, что встретились в замыкании $S$. Однако, не умоляя общности, мы можем считать, что $CT$ содержит только релевантные для $\phi_t$ записи.

Пример 5. Для примера 4.1 и $h = 1$, 

\[ S = \{ X, Y, \text{Line}, \text{IBuilder<Symbol>, Line} \} \]

\[ cl(S) = \{ X, Y, \text{Line, IBuilder<Symbol>, Symbol}, \text{IBuilder<Line>, IBuilder<Line>, Symbol, IComponent, IBuilder<X>, IBuilder<Y>, System.Object} \} \]

\[ \text{inCT}(x) = (x = \text{Line}) \lor (x = \text{Symbol}) \lor (x = \text{IBuilder<Symbol>}) \lor (x = \text{IBuilder<Line>>) \lor (x = \text{IBuilder<X>>) \lor (x = \text{System.Object}) \]

Для примера 4.2 и $h = 2$, 

\[ S = \{ \text{Thing, IBox<T>, IBox<Z>} \} \]

\[ cl(S) = \{ \text{Thing, IBox<T>, IBox<Z>, IBox<IBox<T>>, IBox<IBox<T>>, System.Object, Z} \} \]

\[ \text{inCT}(x) = (x = \text{Thing}) \lor (x = \text{IBox<IBox<T>>}) \lor (x = \text{IBox<T>>) \lor (x = \text{IBox<Z>)} \lor (x = \text{System.Object}) \]

C. Расширение отношения подтипирования

После построения замыкания множества типов $cl(S)$ и сопоставления им переменных $FV$ алгоритм распространяет отношение подтипирования на пары типов в $cl(S)$, вводя предикат

\[ st \overset{\text{def}}{=} \bigwedge_{A,B \in \text{cl}(S)} (A,B) \neq A \nlop B \]

где $st_{A,B}$ определяется одним из четырех способов.

- Если $A$, и $B$ — закрытые типы, то решение об их подтипировании принимает оракул groundSubtype:

\[ st_{A,B} \overset{\text{def}}{=} \tau(A) <: \tau(B) \iff \text{groundSubtype}(A, B) \]

Пример 6. Для примера 4.1,

\[ \text{Symbol} <: \text{IComponent} \iff \top \]

\[ \text{IBuilder<Symbol} <: \text{IBuilder<Line} \iff \bot \]

- Если $A$ открыт, но ни один из них не является 
типов переменной. В таком случае можно за-
писать ограничения в виде эквивалентных преоб-
разованиях, которые задаются правилами вывода 
Var и Super. Введем вспомогательное множество

\[ supertype(C<T>, D<U>) \overset{\text{def}}{=} \{ D<\overline{W} > | C<\overline{T} > <: \overline{T} < D<\overline{W} > \} \]

Тогда

\[ \tau(C<\overline{T} >) <: \tau(D<\overline{U} >) \iff \bigvee_{V \in \text{supertype}(C<T>, D<U>)} \tau(V) <: \tau(D<\overline{U} >) \]

отображает применение детерминированной вер-
сии правила Super, а

\[ \tau(C<\overline{T} >) <: \tau(C<\overline{U} >) \iff \bigwedge_{i} \tau(T_1) <: \text{var}(C\#i) \tau(U_i) \]

(3)
отображает применение правила Var. $st_{A,B}$ в этом случае является конъюнкцией формул (2) и (3).

Пример 7. Для примера 4.1,

$$IBuilder<X> <: IComponent$$

$$IBuilder<X> <: IBox<IBuilder<X>>$$

В процессе применения правил вывода может произойти зацикливание (occurs check). В тексте спецификации CLI [5] отсутствует явное указание на поведение среды в ситуации occurs check, однако в разделе I.8.7.1 сказано, что отношение подтиповирования есть наименьшее отношение, замкнутое относительно набора некоторых правил. Из этого следует, что в случае occurs check отвергается подтиповирование одного типа другим. Итак, в случае зацикливания вывода добавляется правило

$$st_{A,B} \overset{def}{=} C<T> \not\vdash D<\mathcal{U}>$$

Пример 8. Для примера 4.2,

$$Thing <: IBox<Thing>$$

$$IBox<IBox<Thing>> <: IBox<Thing>$$

$$IBox<IBox<Thing>> <: IBox<Thing>$$

$$Thing \not\vdash IBox<Thing>$$

- Если $A = C<T>$ — конструированный тип, а $B = X$ — типовая переменная, то необходимо ввести ограничение, показывающее, что типовая переменная $X$ должна принадлежать к изначальной табличе классов CT (в противном случае, решатель может «включить» новый класс в иерархию $C<T>$). Один из способов сделать это — добавить правило следующего вида:

$$st_{A,B} \overset{def}{=} C<T> \not\vdash \text{inCT}(X)$$

- Наконец, случай, когда $A$ — типовая переменная, будет покрыт на последующих шагах. На данном шаге все такие пары игнорируются: $st_{A,B} \overset{def}{=} \top$.

D. Аксиомы частичного порядка

Исходя из того, что отношение подтиповирования является рефлексивным и транзитивным [10], а таблица классов аксиоматична, можно доказать, что отношение антисимметрично. Важно ясным образом добавить аксиомы частичного порядка, так как это гарантирует, что отношение частичного порядка будет выполняться не только для закрытых типов, но и для открытых.

$$po \overset{def}{=} \forall x. x <: x \land$$

$$\forall x, y. x <: y \land y <: x \Rightarrow x = y \land$$

$$\forall x, y, z. x <: y \land y <: z \Rightarrow x <: z$$

E. Запрет множественного наследования

Множественное наследование в .NET разрешено только для интерфейсов. Если какой-то тип является подтипом не-интерфейса, то он не является интерфейсом, с одним исключением: тип System.Object является классом, но также является подтипом для интерфейсов. Для отражения этих свойств используются предикаты символы interface и subclass. Предикатный символ interface истинен для всех интерфейсов (т.е. типов, аннотированных в CT буквой I) и ложен для не-интерфейсов (т.е. типов, аннотированных в CT буквой C или V).

$$iface \overset{def}{=} \bigwedge \text{interface}(\tau(T)) \land \bigwedge \neg\text{interface}(\tau(T))$$

$$mi \overset{def}{=} iface \land (\forall x, y, z. \text{subclass}(x, y) \land \text{subclass}(x, z) \Rightarrow \text{subclass}(x, y, z))$$

$$\forall x, y. \neg\text{interface}(x) \land \neg\text{interface}(y) \Rightarrow$$

$$(x <: y \Rightarrow \text{subclass}(x, y)) \land$$

$$\forall x, y. \neg\text{interface}(y) \land x <: y \land y \neq \text{System.Object} \Rightarrow$$

$$\neg\text{interface}(x)$$

Пример 9. Для примера 4.2,

$$intr = \neg\text{interface}(\text{Thing}) \land \text{interface}(IBox<\text{IBox<Thing>>}) \land$$

$$\neg\text{interface}(\text{System.Object})$$

F. Запечатанные типы

На следующем шаге вводится ограничение seal для запрета наследования запечатанного типа:

$$seal \overset{def}{=} \bigwedge \forall x, x <: T \Rightarrow x = T \lor \text{inCT}(x)$$

Пример 10. Для примера 4.1:

$$seal = \forall x, x <: \text{Symbol} \Rightarrow x = \text{Symbol} \lor \text{inCT}(x)$$

G. Ссылочные типы и типы значений

Для каждого типового параметра можно указать, будет ли он типом значения или ссылочным типом. Это можно сделать, добавив аннотацию V или R соответственно. Наличие такой аннотации в таблице классов гарантирует, что в ней будет запись с конструктором System.ValueType. Эти ограничения будут представлены в виде формул

$$vl(x) = x <: \tau(\text{System.ValueType}) \land$$

$$x \neq \tau(\text{System.ValueType})$$

$$vrt \overset{def}{=} \bigwedge (\neg vl(\tau(T))) \land \bigwedge (vl(\tau(T)))$$

$T$ аннотирован «V»

$T$ аннотирован «R»
Интуитивно, разрешимость системы типовых ограничений возможна из-за ограничений на таблицу классов (она должна быть нерасширяемой) и из-за требований к отсутствию кванторов в $\phi_t$.

М. Преобразование модели

Пусть $cl_3(\phi_m)$ имеет модель $M_m = (D_m, \sigma_m)$ с конечным носителем $(2^D_m)$. Более того, как сказано в конце предыдущей секции, вместе с моделью можно получить оценку $val_m : FV \rightarrow D_m$ переменных под кванторами существования. Данная запись обсуждается, как перейти от $M_m$ к модели $M_t = (D_t, \sigma_t)$ формулы $cl_3(\phi_t)$ в $L_4$.

Лемма 1. Если для некоторого $a \in D_m$, $\tau(C<T>) \in val_{m-1}(\{a\})$ и $\tau(D<T>) \in val_{m-1}(\{a\})$, то конструкторы $C$ и $D$ совпадают и $\forall .val_m(\tau(T_1)) = val_m(\tau(U_1))$.

Утверждение Леммы 1 можно вывести из антисимметричности подтиповирования путем применения правил (2) и (3) к условиям леммы.

Лемма 2. Если существуют сконструированный тип $C<T>$ и тип $V$ такие, что для $\exists . val_m(\tau(T_1)) = val_m(\tau(V))$, тогда $val_m(\tau(C<T>)) = val_m(\tau(V))$.

Утверждение Леммы 2 можно вывести из набора аксиом 4.

Лемма 1 и Лемма 2 дают способ конструирования отображения $d : D_m \rightarrow D_t$, сопоставляющее элементы носителя $M_m$ замкнутым типам.

Возьмем отображение $fresh : D_m \rightarrow D_t$, сопоставляющее элементу из $D_m$ произвольный нуллярный (и потому замкнутый) конструктор типа, отсутствующий в таблице классов $CT$. Обозначим за $f v(\phi_t)$ множество свободных типовых переменных $\phi_t$. Теперь можно индуктивно определить отображение $d$.

$$d(a) \equiv \begin{cases} fresh(a), & \text{если } \tau^{-1}(val_{m-1}(\{a\})) \subseteq f v(\phi_t) \\ C<T>, & \text{если } \tau(C<T>) \in val_{m-1}(\{a\}) \end{cases}$$

По Лемме 1 отображение $d$ определено корректно, но по лемме 2 является определенным на всём $D_m$.

Отображение $d \circ val_m \circ \tau$ является оценкой на $f v(\phi_t)$. Модель же $M_t$ велико определяется расширением таблицы классов $CT'$. Для построения $CT'$ сначала зафиксируем множество новых типов, введенные отображением $fresh$ из носителя $D_m$:

$$N \equiv \{ fresh(a) \mid \tau^{-1}(val_{m-1}(\{a\})) \subseteq f v(\phi_t) \}$$

Теперь посмотрим «каркасы» отношений подтиповирования для элементов $N$, транзитивное замыкание которых даст всю их иерархию. Для этого для каждого $n \in N$ применяем алгоритм алгоритм выделения транзитивного остова [9]. Обозначим за $B_n$ множество вершин, соединенных в графе-каркасе с ребром $(n, \cdot)$. Таблица классов $CT'$ получается из $CT$ добавлением записи $fresh(n)^* ::= \{ d(b_{n_1}^1), \ldots, d(b_{n_k}^k) \}$ для каждого $n \in N$. Здесь $b_{n_1}^1, \ldots, b_{n_k}^k$ — элементы множества $B_n$. Аннотации для
Список литературы


Кеорема 2. ξ ∈ N и семейство формул в пренексной нормальной форме без кванторов существования \{\phi_i\}_{i=0}^m \subset L_m так что выполняются следующие утверждения

- Если \( cl_3(\phi_i) \) имеет модель, тогда \( cl_3(\phi_i) \) также имеет модель
- Если \( cl_3(\phi_i) \) имеет модель, \( val \) — оценка переменных \( f(v(\phi_i)) \) и max \( \text{height}(t) \leq h \), тогда \( cl_3(\phi_i) \) имеет модель

V. Эксперименты

Эксперименты проводились на SMT-решателе Z3. Всего было проведено 30 экспериментов, в таблицах классов было 10-20 записей. Во входных формулах было от 2 до 10 типов, количество типов в замыкании не превышало 50. Время работы решателя на всех запусках не превышало 0.1 секунды, все вердикты и модели были верными. Пример кодировки в формате SMT-LIB 2 представлен в Приложении A.

VI. Заключение

В работе было показано, как свести задачу выполнимости набора ограничений на открытые типы платформы .NET к разрешимому фрагменту логики первого порядка. Подход был реализован и показал свою работоспособность на практике.

Работа выполнена при финансовой поддержке компании JetBrains.

SMT-BASED ANALYSIS OF CONSTRAINTS ON .NET TYPES

Aleksandr V. Misonizhnik, Dmitry A. Mordvinov

A precise analysis of object-oriented code in isolation from its call points may require reasoning about the relationships of open types. A. Kennedy and B. Pierce extensively studied nominal type systems with variance, showed undecidability of subtyping relation between ground types and proposed the decidable fragments of the type system. Open types, however, remain unexplored, and still there is no decision procedure for the problem of satisfiability of constraints on the open types. In this paper we present an algorithm that reduces the problem of satisfiability of the constraints on the system of .NET-types to a satisfiability problem in first-order logic. Termination is investigated and the resulting logical encodings are shown to belong to a decidable fragment of the first-order logic.
Приложение A: Кодировка примера 4.1 в синтаксисе SMT-LIB2

(declare-sort Type)
(declare-fun subtype (Type Type) Bool)
(declare-fun isinterface (Type) Bool)
(declare-fun hplc (Type) Bool)
(declare-fun sealed (Type) Bool)
(declare-fun subclass (Type Type) Bool)
(declare-const Object Type)
(declare-const X Type)
(declare-const Y Type)
(declare-const Line Type)
(declare-const Symbol Type)
(declare-const IComponent Type)
(declare-const IBuilder<X> Type)
(declare-const IBuilder<Y> Type)
(declare-const IBuilder<Line> Type)
(declare-const IBuilder<Symbol> Type)

(assert (forall ((x Type) (y Type)) (=> (and (subtype x y) (= x y)))))
(assert (forall ((x Type) (y Type) (z Type)) (= (subclass x y) (subclass z y))))

(assert (forall ((x Type) (y Type)) (=> (and (isconstructed x) (subtype x y)) (isconstructed y)))

(assert (forall ((x Type) (y Type)) (=> (and (isinterface y) (= x y) Object) (not (isinterface y)))))

(assert (forall ((y Type) (x Type)) (=> (and (isinterface x) (not (isinterface y))) (or (subclass y x) (subclass x y))))

(assert (forall ((x Type) (y Type)) (=> (and (isinterface IBuilder<X>) (isinterface IBuilder<Y>) (isinterface IBuilder<Line>))
(isinterface IBuilder<Symbol>) (isinterface IComponent) (not (isinterface Object))
(not (isinterface Line)) (not (isinterface Symbol))
(not (hplc IBuilder<X>)) (not (hplc IBuilder<Y>)) (not (hplc IBuilder<Line>))
(not (sealed IBuilder<X>)) (not (sealed IBuilder<Y>)) (not (sealed IBuilder<Line>))
(not (sealed IBuilder<Symbol>)) (not (sealed IComponent)) (not (sealed Object))
(not (sealed Line)) (not (sealed Symbol)))

(assert (and (isinterface IBuilder<X>) (isinterface IBuilder<Y>) (isinterface IBuilder<Line>))
(isinterface IBuilder<Symbol>) (isinterface IComponent) (not (isinterface Object))
(not (isinterface Line)) (not (isinterface Symbol))
(not (hplc IBuilder<X>)) (not (hplc IBuilder<Y>)) (not (hplc IBuilder<Line>))
(not (sealed IBuilder<X>)) (not (sealed IBuilder<Y>)) (not (sealed IBuilder<Line>))
(not (sealed IBuilder<Symbol>)) (not (sealed IComponent)) (not (sealed Object))
(not (sealed Line)) (not (sealed Symbol)))

(define-fun isconstructed ((x Type)) Bool
(or (= x Line) (= x Symbol) (= x IComponent) (= x IBuilder<X>)
 (= x IBuilder<Line>) (= x IBuilder<Symbol>) (= x Object)))
(assert (forall ((x Type) (y Type)) (=> (and (isconstructed x) (subtype x y)) (isconstructed y)))

(assert (forall ((x Type) (y Type)) (=> (and (sealed x) (subtype y x)) (or (sealed y) (isconstructed y))))

(check-sat)
(get-model)
On-the-Fly Filtering of Aggregation Results in Column-Stores

Anastasia Tuchina
Saint Petersburg State University
Email: anastasia.i.tuchina@gmail.com

Valentin Grigorev
Saint Petersburg State University
Email: valentin.d.grigorev@gmail.com

George Chernishev
Saint Petersburg State University
JetBrains Research
Email: chernishev@gmail.com

Abstract—Aggregation is a database operation that aims to provide basic analytic capabilities by partitioning source data into several groups and computing some function on values belonging to the same group. Nowadays it is common in databases, and especially in the OLAP domain, which is a primary venue for column-stores.

In this paper we propose a novel approach to the design of an aggregation operator inside a column-store system. The core of our approach is an analysis of predicates in the HAVING-clause that allows the runtime pruning of groups. We employ monotonicity and codomain analysis in order to detect groups in which predicates would never be satisfied. Eventually, we aim to save I/O and CPU costs by discarding groups as early as possible.

We start by providing a high-level overview of our approach and describe its use-cases. Then, we provide a short introduction into our system and describe a straightforward implementation of an aggregation operator. Next, we provide theoretical foundations for our approach and present an improved algorithm. Finally, we present an experimental validation of our approach inside PosDB — a distributed, disk-based column-store engine that features late materialization and block-oriented processing. Experiments using an SSD drive show that our approach can provide up to 5 times improvement over the naive version.

I. INTRODUCTION

Aggregation is rather common in databases and, in fact, it forms the basis of OLAP. For example, the TPC-H benchmark [1] does not contain a single query which does not involve aggregation. Therefore, in the early 80’s the scientific community recognized the importance of efficient aggregation processing. While there are hundreds of studies concerning aggregation in row-stores, the column-store processing is less explored.

Column-stores are a relatively recent development [2]. Unlike classic approaches, where all attributes of every record are kept together, column-stores employ the opposite idea — they store each attribute separately. This leads to a number of challenges as well as opportunities in query processing.

In this paper we propose a novel technique intended for optimizing the processing of aggregation queries inside column-stores. Our approach concerns analysis of predicates in the HAVING-clause and relies on a simple idea: terminate processing of a group if, judging by the already processed data, the HAVING-predicate will never be satisfied. Thus, it should be possible to save I/O and CPU costs related to evaluation of aggregation functions located in the SELECT-clause.

In order to illustrate this, let us consider the following example (adapted from Q1 of TPC-H):

```sql
SELECT l_returnflag, l_linenumber, SUM(l_quantity) as sum_qty,
       SUM(l_extendedprice) as sum_base_price,
       SUM(l_extendedprice * (1 - l_discount)) as sum_disc_price,
       SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
       AVG(l_quantity) as avg_qty,
       AVG(l_extendedprice) as avg_price,
       AVG(l_discount) as avg_disc,
       COUNT(*) as count_order
FROM lineitem
WHERE l_quantity < 1000000
GROUP BY l_returnflag, l_linenumber
HAVING SUM(l_quantity) < 1000000
ORDER BY l_returnflag, l_linenumber
```

In this case naive processing can be organized as follows:

1) Rewrite query in the form

```sql
SELECT *
FROM (original query without having)
WHERE having-clause
```

2) Move aggregation expressions from the HAVING-clause to the SELECT-clause of the original query if they are not already present there, and perform aggregation as usual (e.g. hash-based aggregation);

3) Filter aggregation results by the HAVING-predicate that uses columns introduced in the step 2 and then eliminate extra columns by projection.

However, if during the second step the partial sum of l_quantity for some group exceeds 1000000, it is possible to discard this group immediately. In the column-store case, this approach will allow us to save some I/O and CPU costs.

At first glance, it may seem that the benefits of our approach are rather limited, because joins incur more significant costs than aggregation. However, there are aggregation queries which do not involve joins, e.g. Q1 and Q6 in TPC-H. More importantly, all queries in this benchmark try to mimic real-life scenarios, and, thus, they represent an actual business need. This indicates that similar queries can be encountered in real-life workloads.

Overall, there are several types of use-cases when our approach would be of use:
1) Aggregation queries that do not involve joins: aggregation running on a denormalized table, a materialized view, solely on a fact table and so on;

2) A case where join operators produce roughly as much data as they receive or even more, so aggregation requires comparable time for processing.

Unlike row-stores, column-stores offer an opportunity to reap the fruits of such optimization with ease. In this paper we consider this problem in the context of PosDB [3]–[5] — a distributed disk-based column-store engine that features late materialization and block-oriented processing. However, we think that the proposed solution is sufficiently universal — almost any column-store system can benefit from similar optimization, regardless of its underlying architecture. Also, despite the current columnar focus of our study, we believe that our approach is applicable to hybrid systems and in-memory DBMSes as well.

Overall, the contribution of this paper is the following:

1) Theoretical basis for on-the-fly pruning of groups during the evaluation of the aggregation operator.

2) An experimental study of our approach using PosDB.

This paper is organized as follows. In Section II we present a short introduction into PosDB and the aggregation operator. Section III provides definitions and presents formal grounds for our analysis. In Section IV we describe an optimized version of the aggregation algorithm. Next, in Section V we present an experimental evaluation and discuss current results. Then, we survey existing aggregation studies in Section VI. Finally, in Section VII we conclude our study and present future work.

II. QUERY PROCESSING IN PosDB

In this section we will give a minimally sufficient description of PosDB internals, and present the naive aggregation algorithm. For a comprehensive description of the whole system, see [3], [5]. Several surveys of column-store technology are presented in references [2], [6]–[8].

A. Basics

Query processing in PosDB is built upon the pull-based Volcano model [9] with block-oriented processing. According to this model, each query is represented by a tree whose nodes are physical operators and edges are data flows. Each operator supports an iterator interface, and data is passed between operators in blocks.

Currently, PosDB supports late materialization only. This is a query execution strategy for column-stores that aims to defer the tuple reconstruction and materialization until as late as possible. In order to implement this strategy, a special intermediate representation, based on the join index from the study [10], was introduced. This representation links records of different tables together. It is used to pass blocks of positional data between operators.

Some operators (e.g., joins) require not only positions, but also attribute values. Therefore, in order to obtain attribute values from the join index we employ a special entity — a reader. There are two types of readers in PosDB that are essential for understanding of our current study:

- ColumnReader is a reader for retrieving values of an individual attribute. In fact, there are several subtypes of ColumnReader, since our system supports data partitioning and data distribution.
- SyncReader is a reader for retrieving values of several attributes related to the same join index in a row-by-row manner. It is implemented as a wrapper for several ColumnReaders.

B. Baseline version of aggregation algorithm

Consider the following SQL query:

```
SELECT T.A, T.B, COUNT(*) as count ,
       MAX(T.C) - MIN(T.C) as diff
FROM T
GROUP BY T.A, T.B
HAVING count = 1
```

In this query, the three following components should be distinguished:

1) GROUP BY clause. It contains a list of attributes that are used to define groups. We will call these attributes grouping parameters.

2) SELECT clause. It contains a list of expressions that we will call aggregation expressions. In our study we assume that only grouping parameters are allowed on the top level. Other attributes should be wrapped in aggregate functions, such as MAX, SUM, etc. Currently, some database systems allow not only grouping parameters, but arbitrary attributes on the top level as well, e.g., MySQL and SQLite. However, such understanding of aggregation is unambiguous only if there is a functional dependency between grouping parameters and “raw” attributes on the top level of SELECT clause. Otherwise, it leads to uncertainty and, consequently, to implementation-dependent behavior. It should be noted that such understanding of aggregation is not prevalent in the database community since it contradicts the SQL’92 standard (see sections 6.5 and 7.9).

3) HAVING clause. It contains a predicate that is applied to the data regarding the whole group and is used to filter resulting rows.

Before describing the aggregation algorithm used in PosDB, we should formally describe the admissible aggregation expressions. They are inductively defined as follows.

Definition 1 A stateless expression is either:

- an identifier of an attribute belonging to a relation that was mentioned in a FROM-clause (free variable);
- a constant of a supported data type (constant);
- \( A + B, A - B, A \cdot B, A \div B, A^n \), where \( A \) and \( B \) are admissible numeric stateless expressions and \( n \) is a non-negative integer (arithmetic expressions);

---

1https://stackoverflow.com/questions/1023347/mysql-selecting-a-column-not-in-group-by
Definition 2 An aggregation expression is either:
- an identifier of any attribute belonging to grouping parameters;
- a constant of a supported data type;
- \( \text{COUNT}(E), \text{MAX}(E), \text{MIN}(E), \text{SUM}(E), \text{AVG}(E) \), where \( E \) is an admissible stateless expression. In this case, we will call such expressions aggregates.
- \( A + B, A - B, A \cdot B, A / B, A^n \), where \( A \) and \( B \) are admissible numeric aggregation expressions and \( n \) is a non-negative integer. In this case, we will call such expressions aggregation arithmetic expressions.

Other types of expressions can be added later in a similar manner.

Both stateless and aggregation expressions may appear in the \textit{SELECT} clause. Joint usage of these expressions on the top level of the \textit{SELECT} clause is prohibited: stateless expressions should be used if there is no aggregation, and aggregation expressions otherwise. As can be seen from the definition, stateless expressions are also used as subexpressions for aggregates.

Let us now consider the semantics of stateless expressions. Here, we consider an abstract processing scheme rather than focusing purely on row- or column-stores. Consider row-by-row scanning of a relation. A stateless expression is a “pure function”; it takes a row as an input and produces a result immediately. On the other hand, aggregation expression can produce a result only when the whole group is processed.

During group processing, values of aggregates that compose an aggregation expression should be updated for each incoming row. These updates lead to change of the “current” value of the aggregation expression involving these aggregates. Thus, during processing aggregation expressions pass through a sequence of intermediate states. Monotonicity analysis of such sequences is a central part of the current study.

Concerning implementation, we must emphasize that in case of a disk-based column-store, expressions should use an interface which allows on-demand column reading. This requirement is essential in order to reduce the number of disk accesses. In PosDB, \texttt{SyncReader} provides this functionality.

Let us now turn to the aggregation operator itself. Its inputs are grouping parameters and a list of aggregation expressions. The local state [9] of the aggregation operator is a hash table with tuples composed of grouping parameters used as keys and lists of aggregates extracted from aggregation expressions used as values.

In a general case, aggregation is an operation that requires full materialization of the corresponding relation, i.e. it should process all data in order to produce the first result. Thus, it can be decomposed into two stages: aggregate evaluation and result generation. The aggregate evaluation stage is the core of the operator. It is organized as a loop over logical “rows” (provided by the corresponding reader in case of PosDB). On each iteration of the loop a new key-tuple is created and the hash table is probed. If a corresponding record is not found, then the aggregates are cloned (they have a state, therefore, an individual copy should be maintained for each group) and a new entry is added to the hash table. Otherwise, it already exists in the hash table and, thus, the corresponding aggregates should be updated.

In the end of the aggregate evaluation stage, each individual entry of the hash table contains computed aggregates for a particular group. Next, at the result generation stage, the hash table is iterated through. During this process, the algorithm computes aggregation expressions, constructs full tuples and returns them to a parent operator.

If the considered query contains a \textit{HAVING}-clause then a parent operator performing filtering of results is added. In PosDB this operator is implemented as filtering on tuples.

III. PRUNING POSSIBILITY ANALYSIS

The core of the proposed approach is \textit{HAVING}-predicate analysis. It consists of two components: monotonicity analysis and codomain analysis. Let us begin with an inductive definition of admissible (correct in the context of a \textit{HAVING}-clause and currently supported) predicates.

Definition 3 The following predicates are admissible:
- \( A = B, A \neq B, A > B, A < B, A \leq x \leq B, \) where \( A, B \) and \( x \) are admissible aggregation expressions or constants (atomic predicates).
- \( P \land Q, P \lor Q, \) where \( P \) and \( Q \) are admissible predicates (compound predicates).

Other types of predicates can be added later in the same way. Currently, we support numeric types only.

A. Monotonicity analysis

Consider an aggregation expression from a fixed group. As we mentioned earlier, throughout the execution of the aggregation algorithm the “current” value of this expression changes several times and, thus, forms a sequence. Often we can guarantee monotonicity of these sequences by analyzing the properties of the corresponding aggregates. Therefore, we can talk about monotonicity of aggregation expressions itself.

Aggregates are monotonic in the following way:
- \( \text{COUNT}(E) \) is weakly increasing;
- \( \text{MAX}(E) \) is weakly increasing;
- \( \text{MIN}(E) \) is weakly decreasing;
- \( \text{SUM}(E) \) is
  - weakly increasing if \( E \geq 0 \),
  - weakly decreasing if \( E \leq 0 \),
  - constant, if \( E = 0 \),
  - not monotonic otherwise;
- \( \text{AVG}(E) \) is not monotonic.

Note that during algorithm execution values of grouping parameters belonging to a particular group do not change. Values of constant expressions behave in a similar manner. Thus, the former should be considered as constants as well.

If we denote weakly increasing aggregation expressions as \( E^+ \) and weakly decreasing ones as \( E^- \), then we can derive monotonicity of aggregation arithmetic expressions according to the following rules:
$E_1^+ + E_2^+, E_1^+ - E_2^-$ are weakly increasing;
$E_1^- + E_2^-, E_1^- - E_2^-$ are weakly decreasing;
if $E_1^+ \geq 0$ and $E_2^+ \geq 0$:
- $E_1^+ \cdot E_2^+, E_1^+ \div E_2^-$ are weakly increasing;
- $E_1^- \cdot E_2^-, E_1^- \div E_2^-$ are weakly decreasing;
if $E_1^- \geq 0$ and $E_2^- \geq 0$:
- $E_1^- \div E_2^-, E_1^- \cdot E_2^+$ are weakly decreasing;
- $E_1^+ \div E_2^+, E_1^+ \cdot E_2^-$ are weakly increasing;
if $E_1^- \leq 0$ and $E_2^- \leq 0$:
- $E_1^- \cdot E_2^-, E_1^- \div E_2^+$ are weakly decreasing;
- $E_1^+ \cdot E_2^+, E_1^+ \div E_2^-$ are weakly increasing;
for other cases, we cannot guarantee monotonicity of the result.

Proof of these statements obviously follows from the properties of arithmetic operations on inequalities.

Let us now turn to $\text{HAVING}$-predicate analysis. We start with atomic predicates and then generalize our approach to the compound ones.

The following predicate types allow to terminate processing of a particular group earlier if the corresponding termination condition is satisfied:

<table>
<thead>
<tr>
<th>Predicate type</th>
<th>Termination condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1^+ &gt; E_1^+$</td>
<td>$E_1^- \leq E_1^+$</td>
</tr>
<tr>
<td>$E_1^- &lt; E_1^-$</td>
<td>$E_1^+ \geq E_1^-$</td>
</tr>
<tr>
<td>$E_1^- = E_1^+$</td>
<td>$E_1^- &lt; E_1^+$</td>
</tr>
<tr>
<td>$E_1^- = E_1^-$</td>
<td>$E_1^+ &gt; E_1^+$</td>
</tr>
</tbody>
</table>

The first column contains the predicate type, which is essentially a predicate with free variables. In contrast, the second column contains an “implementation” of the corresponding predicate where intermediate values of aggregates are substituted instead of being free variables. In our study we refer to both of these entities as predicates. Thus,

**Definition 4** Potentially terminating predicate is a predicate that has a termination condition.

**Definition 5** Terminating predicate is a predicate that has its termination condition fulfilled by a particular instance of data.

The following statements hold for compound predicates:
- $P_1 \land P_2$ is potentially terminating, if $P_1$ or $P_2$ is potentially terminating.
- $P_1 \land P_2$ is terminating, if $P_1$ or $P_2$ is terminating.
- $P_1 \lor P_2$ is potentially terminating, if both $P_1$ and $P_2$ are potentially terminating.
- $P_1 \lor P_2$ is terminating, if both $P_1$ and $P_2$ are terminating.

**B. Codomain analysis**

As shown earlier, there are several important cases — aggregate $\text{SUM}(E)$, aggregation arithmetic expressions containing multiplication and division — that require additional analysis to assess their monotonicity. This analysis consists of checking whether $E \geq 0$, $E \leq 0$ or $E \equiv 0$. It can be carried out using codomain analysis of corresponding stateless numeric expressions.

Our codomain analysis is based on well-known interval analysis [11]. We support open, closed and half-closed intervals. Their endpoints can be infinite. Constants are represented by degenerate intervals.

The following operators of interval arithmetic are used to compute the codomain of an arithmetic expression:
- $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$
- $(x_1, x_2) - (y_1, y_2) = (x_1 - y_2, x_2 - y_1)$
- $(x_1, x_2) \cdot (y_1, y_2) = \left( \min(x_1 y_1, x_1 y_2, x_2 y_1, x_2 y_2), \max(x_1 y_1, x_1 y_2, x_2 y_1, x_2 y_2) \right)$
- \( \frac{1}{(x_1, x_2)} = \begin{cases} \left( \frac{1}{x_1}, \frac{1}{x_2} \right), & \text{if } x_1, x_2 > 0 \\ \left( -\infty, \frac{1}{x_2} \right), & \text{if } x_1 < 0, x_2 = 0 \\ \left( \frac{1}{x_1}, +\infty \right), & \text{if } x_1 = 0, x_2 > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \)
- \( (x_1, x_2) \div (y_1, y_2) = (x_1, x_2) \cdot \frac{1}{(y_1, y_2)} \left\{ \begin{array}{ll} [1, 1], & \text{if } n = 0 \\ (x_1^n, x_2^n), & \text{if } n \text{ is even, } x_1 \leq 0, x_2 \leq 0 \\ (x_2^n, x_1^n), & \text{if } n \text{ is even, } x_1 \leq 0, x_2 \leq 0 \\ [0, \max(x_1^n, x_2^n)), & \text{if } n \text{ is even, } x_1 \leq 0, x_2 \geq 0 \\ (x_1^n, x_2^n), & \text{if } n \text{ is even, } x_1 \geq 0, x_2 \geq 0 \\ (x_1^n, x_2^n), & \text{if } n \text{ is odd} \end{array} \right. \)

Concerning endpoint inclusion, it should be noted that inclusion is kept if and only if operation is applied to included endpoints. Otherwise, an endpoint is excluded.

Note that there is an issue with the interval arithmetics. Estimates obtained by its application strongly depend on the form of an expression. For example, consider the expression $x/(1-x)$, for which two different intervals can be obtained — one for $x/(1-x)$ and another for $1/(1-x)$. However, it is guaranteed that all intervals would contain the range of the analyzed function.

Currently, it is unclear whether it would be useful to employ more complicated (and more resource-consuming) approaches to get better estimates in case of our algorithm.

Another aspect that should be discussed is the input of operators. Codomain may be described not by a single interval, but by a union of several disjunctive intervals. Thus, operators may also produce unions of several intervals. In our study we restrict ourselves to supporting only a single interval per attribute.

Information about possible values of attributes is taken from the meta-information based on the CHECK-constraints stored in the catalog.
IV. OPTIMIZED AGGREGATION ALGORITHM

We are now ready to present an optimized version of the aggregation algorithm described in the section II-B. Our algorithm combines both aggregation and filtering into a single operator. Therefore, the interface of the aggregation operator was slightly changed — now it also receives a HAVING-predicate as a parameter. If there is no HAVING-clause in the query, then a non-optimized version should be run.

The optimized algorithm tracks the state of the predicate for each group, so several changes should be introduced into the hash table. Now this table contains entries that are structures with the following fields:

- A list of aggregates that should be evaluated eventually (as in the baseline version). Here we will call such aggregates primary aggregates.
- A copy of the HAVING-predicate to check whether it is necessary to further process the current group.
- A list of aggregates from the HAVING-predicate that are used to compute the state of the corresponding HAVING-predicate copy. We will call them auxiliary aggregates.

The optimized algorithm requires a new stage — a predicate analysis stage. At this stage, the HAVING-predicate is analyzed using the approaches described in the previous section. If it is potentially terminating, then it will be possible to perform earlier termination of processing of any group when the corresponding termination condition is satisfied for this group. Otherwise, the baseline version of aggregation algorithm should be used.

Optimization itself is mostly applied at the aggregate evaluation stage. As in the baseline version of algorithm, we iterate through the logical “rows” (provided by SyncReader in PosDB). Inside the loop we check if the corresponding group already exists in the hash table. However, now we need to check the state of the predicate copy as well. If there is no such group, we clone the predicate, extract the aggregates (auxiliary) from it, update their values and evaluate the predicate. If the predicate is not terminating yet, then we also clone the primary aggregates and add a record to the hash table. If the predicate is already terminating, then we delete the cloned predicate and add a “tombstone” into the hash table instead of a normal record.

If the corresponding group is already in the hash table, and it was not “tombstoned”, then we update the values of both primary and auxiliary aggregates from the predicate, and check the status of the predicate. If it becomes terminating, then we delete the found record and replace it with a “tombstone”. Otherwise, we just proceed to the next logical “row” without fetching values needed for computation of both primary and auxiliary aggregates. It is this step of the algorithm that makes I/O and CPU savings possible.

At the result generation stage, we iterate over the hash table, skip groups that have been “tombstoned” and check the HAVING-predicate for the remaining groups. Next, tuples constructed from records that satisfy the predicate are passed to the parent operator.

V. EXPERIMENTS

Experimental evaluation was performed on a PC with the following characteristics: 4-core Intel®Core™ i5-7300HQ CPU @ 2.50GHz, 8 Gb RAM, running Linux Ubuntu 16.04.1 LTS. 128GB KINGSTON RBU-SNS SSD was used as storage. Test queries are based on Q1 from TPC-H. The first four of them have the following form:

```
SELECT
    l_returnflag, l_linenumber,
    SUM(l_quantity) as sum_qty,
    SUM(l_extendedprice) as sum_base_price,
    SUM(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    AVG(l_quantity) as avg_qty,
    AVG(l_extendedprice) as avg_price,
    AVG(l_discount) as avg_disc,
    COUNT(*) as count_order
FROM lineitem
GROUP BY l_returnflag, l_linenumber
HAVING having-clause
```

where having-clause is

- `l_linenumber = 'O'` for Q1;
- `having count(*) < 100000` for Q2;
- `(l_returnflag = 'A') OR (l_linenumber = 'O')` AND `(MIN(l_tax) > MAX(l_discount))` for Q3;
- `SUM(l_quantity) < 1000000` for Q4.

These queries are designed for studying how optimization is affected by different kinds of predicates. All of them are supposed to demonstrate significant performance improvement due to avoidance of unnecessary I/O in the optimized algorithm.

We have also designed the Q5 query for a first, rough appraisal of the overhead introduced by our algorithm. Q5 is an example of a query where no performance improvement could be gained due to absence of I/O savings. In this query, all columns have to be read regardless of predicate values.

```
SELECT
    l_returnflag, l_linenumber
FROM lineitem
GROUP BY l_returnflag, l_linenumber
HAVING l_returnflag = 'A'
```

For each combination of an algorithm and a scale factor (SF) we have run all the aforementioned queries 10 times and calculated the 95% confidence intervals. The full results of the experiments are presented in the Table I. We have also visualized the results for SF = 50 in Fig. 1 to make it more illustrative.

As we can see from Table I, there is no considerable performance dependency on the predicate complexity — queries Q2–Q4 show very close results on all scale factors. On the other side, as expected, optimization efficiency significantly depends
TABLE I
Experiments results (milliseconds)

<table>
<thead>
<tr>
<th>Query</th>
<th>Algorithm</th>
<th>SF = 1</th>
<th>SF = 5</th>
<th>SF = 10</th>
<th>SF = 25</th>
<th>SF = 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Optimized</td>
<td>2087 ms ± 2%</td>
<td>10515 ms ± 2%</td>
<td>20743 ms ± 1%</td>
<td>53145 ms ± 1%</td>
<td>104899 ms ± 1%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>3305 ms ± 2%</td>
<td>16589 ms ± 3%</td>
<td>32615 ms ± 1%</td>
<td>83419 ms ± 2%</td>
<td>159588 ms ± 2%</td>
</tr>
<tr>
<td>Q2</td>
<td>Optimized</td>
<td>841 ms ± 1%</td>
<td>3830 ms ± 2%</td>
<td>8217 ms ± 1%</td>
<td>20150 ms ± 1%</td>
<td>45833 ms ± 2%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>3293 ms ± 3%</td>
<td>16564 ms ± 3%</td>
<td>32389 ms ± 1%</td>
<td>83336 ms ± 2%</td>
<td>158751 ms ± 2%</td>
</tr>
<tr>
<td>Q3</td>
<td>Optimized</td>
<td>685 ms ± 1%</td>
<td>3634 ms ± 1%</td>
<td>8163 ms ± 1%</td>
<td>20086 ms ± 1%</td>
<td>45734 ms ± 1%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>3458 ms ± 2%</td>
<td>17276 ms ± 3%</td>
<td>33843 ms ± 1%</td>
<td>87288 ms ± 2%</td>
<td>164593 ms ± 2%</td>
</tr>
<tr>
<td>Q4</td>
<td>Optimized</td>
<td>766 ms ± 1%</td>
<td>3717 ms ± 1%</td>
<td>8217 ms ± 1%</td>
<td>20078 ms ± 1%</td>
<td>45764 ms ± 1%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>3511 ms ± 2%</td>
<td>17651 ms ± 3%</td>
<td>34626 ms ± 1%</td>
<td>88972 ms ± 2%</td>
<td>168037 ms ± 2%</td>
</tr>
<tr>
<td>Q5</td>
<td>Optimized</td>
<td>510 ms ± 1%</td>
<td>2590 ms ± 1%</td>
<td>4980 ms ± 1%</td>
<td>12693 ms ± 1%</td>
<td>25994 ms ± 1%</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>504 ms ± 1%</td>
<td>2439 ms ± 1%</td>
<td>4873 ms ± 1%</td>
<td>12465 ms ± 1%</td>
<td>25666 ms ± 1%</td>
</tr>
</tbody>
</table>

Fig. 1. Comparison of algorithm performance for SF=50

on the predicate selectivity and on the time when pruning can be performed. We suppose that the difference between Q1 and Q2–Q4 should be explained by this fact. Detailed analysis of these factors was postponed for the future.

Evaluation of Q5 shows that the overhead introduced by our algorithm is rather small (about 2–3%) and does not depend on the scale factor.

Concerning the dependency on the scale factor, it is looking close to linear for all the considered queries.

VI. RELATED WORK

According to reference [9], the processing of aggregation queries has been studied at least since the end of the 70’s [12]. Nowadays, it is a mature area of research which features hundreds of papers [13].

These studies can be roughly classified into the following groups:

1) Optimization of aggregation queries. In these papers authors study how to efficiently process aggregation queries mostly by using various plan transformations. In the study [14], [15], the authors propose two transformations: eager aggregation (moving a GROUP BY down through join) and lazy aggregation (moving a GROUP BY up). The former allows to reduce the number of entries that need to be processed by the join operator, and, consequently, to improve the overall query processing performance. The latter is of interest when the query operates on a view containing a GROUP BY. A similar transformation is proposed in study [15]. The core of this approach is to pre-aggregate data using any column set that functionally determines the table being aggregated. In study [16] authors integrate subquery and aggregation processing techniques by proposing a set of shared primitives. Then these primitives are used to generate optimized plans. The problem of aggregation query optimization in the OLAP environment was considered in reference [17]. In this paper, an analysis of an approach called Hierarchical Pre-Grouping is performed and a number of transformations is proposed and analyzed.

2) Optimization of evaluation of an individual aggregation operator. These studies aim to organize processing in the most efficient way. There are two basic methods for performing an aggregation [9] — hashing and sorting. According to reference [9] sorting was considered in study [12]. Among contemporary studies it is essential to note Blink [18], where authors consider aggregating compressed data and reference [19] where hardware-efficient multi-threaded aggregation in column-store was presented. However, none of these studies analyzed aggregation predicates in order to improve individual operator performance.

3) A relatively recent approach called online aggregation. Its goal is to provide the database user with means to control the execution of aggregation. The proposed use-case is the following: while processing data, progressively refine an approximate answer and provide its running confidence intervals. Unlike plain aggregation, where the user passively waits for the answer, this approach allows the user to terminate a query early if they deem the approximate results acceptable. Originally, this approach was proposed in reference [20], and later, many studies have followed. For example, the study [21] extends this idea onto nested queries that contain aggregation and also proposes a multi-threaded model. Next, the paper [22] addresses parallel
aggregation on a cluster of computers.

4) Another group of papers studied approximate processing of aggregation queries. Unlike the previous approach, this type of studies does not imply user intervention during the processing. Approximate aggregation query answering using sampling was studied in reference [23]. The idea of this paper is to index outlying values in order to reduce the approximation error. The study [24] addresses the problem of approximate time-constrained query evaluation. The authors propose an algorithm for evaluating count-containing aggregation queries that can be stopped if the desired error range is obtained or the pre-specified time limit is exceeded.

5) Novel models of aggregation and novel aggregations operators. Horizontal aggregation is proposed in reference [25]. Its idea is to generate a new table, where a separate column for each unique value of columns belonging to aggregation expression is generated. At the same time, the rows of this table contain all unique values from the columns of GROUP BY list. The authors of this paper propose not only semantics of this operation, but also language extension and discuss query execution. Similarly, reference [26] describes two aggregation-like operators: PIVOT and UNPIVOT. The former transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to generate the new column for a row, and another source column is used as the data for that new column. The latter performs the inverse operation by removing a number of columns and creating additional rows. In studies [27], [28] novel aggregation operators are proposed. The former study considers embedding of grouping variables [29] into SQL queries. Grouping variables is a tool that allows to specify additional conditions for the desired groups. It is much more expressive than HAVING-clause. The latter study proposes a generalization of an aggregation operator that allows formation of aggregation groups without requiring an ordering of the data relation.

VII. CONCLUSION

In this paper we have proposed a novel approach to evaluation of aggregation in column-stores. We employ monotonicity and codomain analysis in order to invoke early termination that allows to save CPU and I/O costs. To validate our idea, we have implemented the designed algorithms inside a disk-based column-store query engine. Preliminary experiments show that our approach can improve query performance up to 5 times over the naive algorithm.

In our future studies we plan to evaluate performance dependency on the number of groups, data distribution, selectivity and complexity of the predicate. We also going to assess the overhead introduced by our algorithm in more detail. Other future studies may include combining proposed operator with other approaches to aggregation (e.g. partial aggregation approach), applying it for in-memory systems, and exploring opportunities offered by novel hardware.

REFERENCES


Development of a software package for acoustic emission control data analysis

Victoria Belousova
Saint Petersburg State University
Email: vsbelousova64@yandex.ru

Anastasia Grigorieva
Saint Petersburg State University
Email: a.v.grigorieva@spbu.ru

Abstract—The following article addresses a software package developed for working with data obtained during monitoring the detection of material defects via the acoustic emission (AE) method. Timely detection of cracks allows to prevent contingencies and accidents at early stages. This paper describes the architecture of this software, as well as the used calculation methods, provides visualized results of their work, and compares them with other analysis methods. The innovation of this work is the use of the moving window method for AE data analysis. Obtained results indicate the practical importance and relevance of our research in this area.

I. Introduction

Evaluation of the current condition of varying industrial and infrastructural objects is one of actual problems of modern material science. These objects include, but are not limited to the oil and gas and chemical industry equipment, thermal and nuclear power equipment, aerospace equipment, pipeline and railway transportation, bridge constructions, and concrete and reinforced concrete structures. The risks of equipment failure increase substantially after it has been in use for a long time under mechanical and thermal loads. The development of methods that allow to study the physical nature of material degradation processes is an important task within the field of technical diagnostics. Acoustic emission testing is prominent among these methods. It allows to identify the coordinates and estimate the danger level of defect-associated acoustic emission sources that appear in a loaded object.

In this work we present a software package intended for analyzing a large volume of specific data that is being studied by a considerable number of researchers all around the world. Our software is designed for analyzing different impulse responses of AE signals (amplitude, energy, length, etc) with the use of the moving window method. By employing this method, the software identifies the change dynamics of the arithmetic mean, median, standard deviation, and b-value of these responses.

II. Acoustic emission method

Acoustic emission testing is an efficient method of nondestructive testing. It is based on detecting elastic waves during deformation of stressed material. These waves travel from the source to sensors that transform them into electrical signals. From the standpoint of the AE method, a defect can produce its own signal [1]. The AE testing devices measure these signals, and then display data used for evaluating the condition and behavior of the entire structure of the tested object. The sensitivity of this method is sufficient to register even microscopic crack growth (by 0.001 mm), which allows to detect cracks in time. The AE method can be employed for testing of a wide variety of technological processes, as well as processes of changes in properties and condition of materials. This broad spectrum of tasks and the variety of control objects requires constant improvement of data processing tools.

III. Overview of existing solutions

There are several acoustic emission systems made by different manufacturers. A review of the characteristics of their software, technical parameters, capacity of their AE equipment, and certain abilities for the analysis of registered data is presented in study [2]. Unfortunately, the post-experimental analysis capabilities of reviewed systems are limited — mostly to creation of standard plots. For example, in AMSY-5, it is easy to create a number of impulses-amplitude histogram or an amplitude-time correlation plot, but it is not always possible to implement a custom user formula [3]. Moreover, the system itself is quite expensive.

Furthermore, a considerable number of studies dedicated to experimental and practical results of using AE exists, for example: [4], [5], [6]. The moving window method is widely applied in different scientific fields, such as economics [7], geophysics [1], social networks analysis [8], audio encryption, and so on.

Our software applies the moving window method to three statistical quantities and one composite parameter specifically during analyzing data of acoustic emission testing of loaded structures. This allows to filter peak values of these quantities and observe the trends in changes of process phases in general.

IV. Software architecture

Currently, our software allows to build the trend of a time series. We employ moving averaging for trend determination. The calculations are performed on impulse responses of acoustic signals — mainly amplitude values. This parameter is one of the most informative, because it indicates the detectability of a signal, which is why it is frequently used during AE testing. For example, in [9], the average amplitude of events is used for forming the P(R) criterion used for determining the necessity of additional testing of detected areas of AE activity.
This software is written in C#. It accepts an input of a file containing data from a certain time interval. Every line of this file contains data collected from a single sensor, in particular, registration time (up to a microsecond), sensor number, and the value of the parameter chosen for processing, e.g., amplitude. The user can set up the parameters required for their research. The software allows to select the sensor whose data will be analyzed to localize the process, indicate whether the analysis will be performed with respect to the number of signals or time, designate the calculation method for one of the four statistical indicators (arithmetic mean, median, standard deviation or b-value). The selection of the window size adjusts the accuracy grade of data evaluation and resolves the problems associated with the possible non-uniformity of data distribution in the time domain.

V. Experiments

In this study we use the data collected during two experiments designed to model the use of real-life constructions with different load types. In the first experiment (strength load) we have used steel-reinforced concrete beam samples which were being bent according to a 3-point scheme. The total volume of AE testing data we have obtained is quite significant. Thus, in some figures we have only plotted the data of sensor 2 (Fig. 1, 3, 5, 7, 10), and in others we have used the data of all four sensors (Fig. 9, 8).

In the second experiment (strength-thermal load), we have monitored a large-sized object contained in a cylindrical concrete construction. During the experiment, the control object was uniformly heated to 400°. The data were being registered by ten sensors. Fig. 2, 4, 6, 11.

VI. Processing methods

The developed analysis system provides the ability to average the processed parameter by calculating three indicators: arithmetic mean, median, and standard deviation. All of them are calculated with the use of the moving window method. This method can be explained as follows: the calculations are performed on same length sets of consecutively registered data, which are shifted by one value relatively to each other during consecutive scanning of the entire measurement interval. The data set size (moving window size) is determined by the user.

A. Simple moving average

Simple moving average (or arithmetic mean) is calculated as follows:

$$\sigma_t = \frac{1}{n} \sum_{i=1}^{n} x_i(t)$$  \hspace{1cm} (1)

where \( t \) is the time interval; \( n \) is the smoothing interval; \( x_i(t) \) is the time series.

Fig. 1 and 2 present the plots of the simple moving average. The smaller the size of the window, the faster the moving window method identifies the new trend, but with that, the final plot contains more false vibrations.

If the size is too large, then the trend will be identified slower, however, there will be fewer false vibrations as well.

Fig. 1-6 display the plots of the moving average for the first experiment (destructive testing of steel-reinforced concrete beams), where the optimal window size is 100. This size has been identified via the possibility of real-time adjustment of program parameters.

Hereinafter in this section, all plots are built with the window size set to 100 signals, for time windows this size is set to 100 milliseconds, used data slice — the entire duration of the experiment.

![Figure 1. Simple moving average. X-axis indicates the number of signals, Y-axis indicates amplitude.](image1)

![Figure 2. Simple moving average. X-axis indicates time (in milliseconds), Y-axis indicates amplitude.](image2)

B. Statistical median

Statistical median is the middle element of an ordered sample. We use the following algorithm to determine the median values: enumerate all values from 0 to \( N \) in an ascending order, then the median values are the elements indexed \( 0.5N \) and \( (0.5N+1) \) for an even \( N \), and \( 0.5(N+1) \) for an odd \( N \). Fig. 3 and 4 display the plots of the medians.

C. Standard deviation

Standard deviation is calculated as follows:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i(t) - \bar{x}(t))^2}$$  \hspace{1cm} (2)

where \( x(t) \) is the time series, \( \bar{x}(t) \) is the arithmetic mean, and \( n \) is the size of the moving window.

A larger standard deviation value indicates a larger scatter in the presented sample. A smaller value points...
VII. Plot analysis

In the beam destruction experiment, the plots of the simple moving average and the statistical median are virtually the same. This is correct for both the time scale of the entire experiment (Fig. 2 and 3) and smaller time scales (Fig. 7). Our analysis has revealed that standard deviation is the least significant parameter out of all three. Although it is of theoretical interest, statistical median and simple average have turned out to be more informative in practice. These parameters help to trace the dynamics of crack formation. According to the study [6], during static loading of metal with a crack there is no increasing trend in the time domain of AE signal amplitude, but there are individual AE signals whose amplitude exceeds the average by 45 dB. We observe a similar situation in the beam loading experiment. This change of trend can be observed on the resulting plots: in Fig. 7, which represents the phase of active formation of main cracks, the averaged amplitude values do not increase uniformly. Furthermore, the increases of amplitude of certain signals are filtered via the moving window method which allows to see the whole picture of trend change.

The selection of an optimal size of the moving window is important. Our software allows the user to make this choice empirically for the whole duration of the experiment, and then change this value proportionally to the total number of signals in a particular smaller data slice. For example, in the beam experiment, one sensor has registered 1700 events. The clearest picture was obtained with the size of the window $L = 100$ events. Thus, for the macrocrack formation period that contains 305 events, $L$ was set to $305*100/1700 \approx 18$ (Fig. 7). Sometimes, it is reasonable to reduce the window size, for example, to avoid missing the registration of relatively rare events such as macrocrack formation.

Consider Fig. 8. In this figure, arrows denote the moments of crack formation that were registered directly (visually) during the experiment. It is interesting that this plot also contains similar decline peaks at different time points. It is highly likely that those points correspond to internal cracks in the structure that could not be identified visually.
VIII. b-value analysis

It is possible to use the Gutenberg-Richter law (widely applied in seismology [1]) to study the scaling of the amplitude distribution of AE signals that appear during crack formation. In AE terms, this formula can be written as follows:

\[
\log_{10} N = a - b \times A_{\text{max}} \text{dB}
\]

(3)

where \(A_{\text{max}} \text{dB}\) is the maximum amplitude in the window (in decibels), \(a\) is an empirical constant value set to 4.8, \(b\) is a value obtained from this equation and then multiplied by 20 to be comparable to the value used in seismology[14]. The b-value is used for identifying the predominate destruction type and determining trends in construction damage development.

IX. Results and conclusions

The following results have been obtained:

1) We have developed a software package that enhances an AE control analysis system with several data analysis methods designed to increase the informativeness of testing. This change of the predominant destruction type that were presented in the following studies: [13], [15], [16], [17].

We have implemented the ability to calculate the b-value in our software package. In some cases, it turns out to be a more informative evaluation parameter for crack formation dynamics than other statistical indicators. For example, in experiments with thermal or composite (strength-thermal) loading of a large-size reinforced concrete structure, the change of b-value trends in regards to the defect formation stage are more pronounced in comparison to the experiments with strength loading of small samples. Fig. 10 and Fig. 11 show the b-value plotted by the system during analyzing the results of the strength and thermal load experiments respectively.
The software was tested on real experimental data.

2) The correctness of the employed algorithms is confirmed by the obtained results matching the previously known facts on the development of defect formation.

3) Employing this software package allowed the AE testing specialists to perform a more accurate and detailed analysis of data, which substantially increased the informativeness of testing.

In this paper, we have presented elements of analysis of data collected during both a laboratory experiment of loading and destruction of reinforced concrete beams and real-world testing of a large-size reinforced concrete construction. The conditions of conducting the first type of experiments have made it possible for us to observe certain key phases of the sample destruction process and identify them with the corresponding AE testing data. Software algorithms have performed well in these experiments: the type of obtained dependencies corresponds to the real-world processes that occurred in control objects according to the known facts about the mechanics of destruction of this type of materials. Thus, the proposed algorithms were tested successfully.

During our research, we have corrected the moving window size with respect to estimating the maximum informativeness of this parameter during the destruction stage. The resulting estimates are employed for the moving window method in the developed software for both the described experiment and other situations in which crack formation (deformation or other internal destruction) processes are obscured and occur inside of the object. In this kind of experiments, the value of such analysis increases due to the inability to visually observe material structure degradation processes and having to resort to evaluating them by indirect indicators.

The second kind of structure that was considered in this study is an object of this type. The results of additional examination of material structure via destructive methods confirmed the correctness of conclusions that were made during the AE testing with the use of our software.

The software of existing AE systems is generally limited to a set of standard plots used for a formal representation of testing results. It usually lacks advanced tools of data analysis. This is typical even for the most modern AE testing systems, such as AMSY-5 (developed in Germany) [3], which was employed during the discussed experiments.

X. Conclusion

In this work we have developed a software package intended for analyzing acoustic emission testing data. We have shown that it can be successfully employed for both laboratory experiments and large-size construction testing. We should note that this kind of analysis requires determining the optimal size of the moving window. All of the required estimates of this value for the arithmetic mean, median and standard deviation were obtained. Using these estimates, we have performed the calculations on data of impulse responses of acoustic signals. The final results indicate the efficiency of the suggested moving averaging methods in the task of analyzing acoustic emission testing data and the practicability of using the considered software.

In our further research we plan to perform the analysis of data obtained in experiments with composite loads and then generalize the results.

The development of methods for detecting signals associated with defect growth with the use of information theory methods could be a fruitful area for further work as well.

References


