Multi-scale interaction techniques for the interactive visualization of execution traces

Rémy Dautriche

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Rémy Dautriche

Thèse dirigée par **Alexandre Termier**
et codirigée par **Renaud Blanch et Miguel Santana**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **l'Ecole Doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique**

Multi-scale Interaction Techniques for the Interactive Visualization of Execution Traces

Thèse soutenue publiquement le **20 Octobre 2016**,
devant le jury composé de :

M. Bruno Raffin
Directeur de Recherche à l'INRIA Grenoble, Président
M. Emmanuel Pietriga
Directeur de Recherche à l'INRIA Saclay, Rapporteur
M. Yannick Prié
Professeur à l'Université de Nantes, Rapporteur
Mme. Karine Heydemann
Maître de Conférence à l'Université de Paris, Examinateur
M. Marc Plantevit
Maître de Conférence à l'Université de Lyon, Examinateur
M. Alexandre Termier
Professeur à l'Université de Rennes, Directeur de thèse
M. Renaud Blanch
Maître de Conférence à l'Université Grenoble Alpes, Co-Directeur de thèse
M. Miguel Santana
Directeur du centre SDT à STMicroelectronics, Co-Directeur de thèse

Abstract

Developing streaming multimedia applications on embedded systems becomes increasingly complex over time. New multimedia standards reach the market to support better resolutions and overall improved quality delivered to the end-user. Consequently, hardware platforms complexify and developing the software to fully exploit them becomes harder at each new generation. The traditional debugging method for streaming applications is the usage of execution traces. However, the amount of data generated by modern software largely increases and existing tools do not allow an efficient debugging process as they become unable to tackle large amounts of data. In this thesis, we focus on new interactive visualization techniques enriched by results of data mining algorithms for a more efficient analysis of execution traces for multimedia applications.

First, we introduce Slick Graphs, a binning and smoothing technique for time series visualization. Slick Graphs mitigate the quantization artifacts, introduced by the traditional smoothing techniques, by using the smallest possible binning intervals, i.e. pixels. We compared Slick Graphs to traditional smoothing techniques in a user study and show that the Slick Graphs are significantly faster and more accurate when working with periodic data. We then propose a novel interaction visualization framework, TraceViz, to explore the execution traces at different level of details and integrate the Slick Graphs to provide a global overview of the trace. With TraceViz, we also introduce a fast back-end to support the interactive browsing of huge traces. We perform a performance analysis to show that the TraceViz back-end outperforms the back-end used in state-of-the-art debugging tools for execution traces. Execution traces contain meaningful information that can be computed using data mining techniques.

A wide range of patterns can be computed and provide valuable information: for example existence of repeated sequences of events or periodic behaviors. However, while pattern mining approaches provide a deeper understanding of the traces, their results is hard to understand due to the large amount of patterns that have to be examined one by one. We propose a novel visual analytics method that allows to immediately visualize hidden structures such as repeated sets/sequences and periodicity, allowing to quickly gain a deep understanding of the trace. Finally, we also show how our method can be applied with different types of data than execution traces.

Résumé

Développer des applications de streaming multimedia pour systèmes embarqués devient une tâche de plus en plus complexe. De nouveaux standards multimedia apparaissent régulièrement sur le marché pour supporter de meilleures résolutions et délivrer du contenu multimedia de meilleure qualité. Une conséquence est la complexification des plateformes matérielles et du développement logiciel. La méthode traditionnelle de débogage pour les applications de streaming multimedia est l'utilisation de traces d'exécution. Cependant, la quantité de données générée par les logiciels modernes augmente et les outils existants ne passent pas à l'échelle, ne permettent plus un débogage efficace. Dans cette thèse, nous nous focalisons sur de nouvelles techniques de visualisation enrichies par des résultats d'algorithmes de fouille de données afin de permettre une analyse efficace des traces d'exécution.

Nous commençons par présenter les Slick Graphs, une technique de découpage et de lissage pour la visualisation de séries temporelles. Les Slick Graphs minimisent les artéfacts introduits par les techniques de lissage traditionnelles en utilisant le plus petit intervalle possible: les pixels. A travers une étude utilisateur, nous montrons que les Slick Graphs sont significativement plus rapides et plus précis avec des données périodiques. Nous proposons ensuite un nouveau système de visualisation interactive, TraceViz, pour explorer les traces d'exécution à différents niveaux de détails. Avec TraceViz, nous introduisons aussi un back-end permettant l'exploration interactive de trace d'exécution de taille importante. Nous fournissons une analyse de performance montrant que le back-end de TraceViz délivre des performances significativement meilleures que les back-end utilisés dans les outils de débogage disponibles aujourd'hui.

Les traces contiennent aussi de nombreuses informations importantes qui peuvent être calculées avec des algorithmes de fouille de données comme par exemple l'existence de séquences d'événements répétées au cours de la trace ou des comportements périodiques. Cependant, même si les techniques de fouille de données permettent d'avoir une meilleure compréhension des traces d'exécution, leurs résultats sont difficiles à exploiter dû au grand nombre de motifs à examiner un par un manuellement. Nous proposons une nouvelle méthode d'analyse visuelle qui permet de visualiser les structures cachées dans une traces comme les séquences répétées et la périodicité d'un ensemble d'événements, permettant de rapidement avoir une compréhension fine de la trace. Enfin, nous montrons aussi comment notre méthode peut être appliquées à différents types de données, autres que les traces d'exécution.

Acknowledgement

I received support from many people during these three years and I will try not to forget anyone!

My first thanks go to Renaud Blanch, Alexandre Termier and Miguel Santana for their advices and support they provided me during these three years of work. They provided me a great environment to work on subjects that really interest me.

I also want to thank the members of my jury: Bruno Raffin, Emmanuel Pietriga, Yannick Prié, Karine Heydemann and Marc Plantevit for reviewing my thesis and their constructive feedback.

Many thanks go to the EHCI team with who I spent a huge amount of time during this thesis. Colleagues at STMicroelectronics have also largely contributed to make these three years great. Specially I would like to thank Jérôme and Julien with who I spent most of my time at the company. I really appreciated our fruitful and technical conversations.

Friends played a big role during this thesis. To name a few, many thanks to Nico, Lucie, Benoît, Bruno, Laurent, Joanna, Gabriel, Pierre, Thomas, Pauline, Olivier for the great times, whether outside for skiing or hiking, whether at the pub ;) Special mention to Nico and Antoine for the hikes and biking rides even though I could not go as often as I wanted!

My family provided me their full support during this three years journey and largely contributed in helping me finishing this work. I am deeply thankful to my dad and Véro for encouraging me and to my siblings, Pierrick, Maëlle and Armel for all the moments we spent together.

Finally, I want to deeply thank Mi for her love, happiness, and endless support specially during the difficult moments.

Contents

Abstract i

Résumé iii

Acknowledgement v

1 Introduction 1
- 1.1 Motivation and Approach . . . 2
  - 1.1.1 Challenges in Embedded Systems and Multimedia Applications 2
  - 1.1.2 Research Approach . . . 4
- 1.2 Contributions . . . 4
- 1.3 Scientific Context . . . 6
- 1.4 Thesis Outline . . . 7

I Background 9

2 Multimedia Applications on Embedded Systems 11
- 2.1 Evolution of Video Standards . . . 12
- 2.2 Evolution of the Embedded Systems . . . 14
- 2.3 Focus on Hardware for Multimedia Decoding . . . 15
- 2.4 Decoding Multimedia Streaming Applications . . . 16
- 2.5 Debugging Multimedia Applications on Embedded Systems . . . 17
  - 2.5.1 Execution Traces . . . 18
  - 2.5.2 Tracing Systems . . . 20
- 2.6 Conclusion . . . 21

3 Related Work 23
- 3.1 Time Series Visualization . . . 23
  - 3.1.1 Time Representation . . . 25
  - 3.1.2 Multiple Time Series Strategies . . . 29
  - 3.1.3 Large Time Series Exploration . . . 33
  - 3.1.4 Exploration of Large Collections of Time Series . . . 37
  - 3.1.5 Visual Mining of Time Series . . . 40

3.2 Visualization of Execution Traces . . . 43
3.2.1 Overview of a Trace . . . 43
3.2.2 Detailed Visualization of a Trace . . . 46
3.2.3 Summary: a Gap Between Overview and Detail Visualizations 51
3.3 Pattern Visualization . . . 52

4 Challenges for Trace Debugging 57
4.1 Inaccurate Time Series Rendering . . . 58
4.2 Large Gap Between Overview and Detailed View . . . 58
4.3 Slow Back-end Performances . . . 59
4.4 Pattern Mining for the Visualization of Execution Traces . . . 60

II Contributions 61

5 Research Approach and Evaluation Methodology 63
5.1 Research Approach . . . 63
5.2 Evaluation Methodology and Validation . . . 64
5.2.1 Slick Graphs Evaluation . . . 64
5.2.2 TraceViz Evaluation . . . 64
5.2.3 Structures Visualization . . . 65

6 Slick Graphs: Slick Visualization of Time Series 67
6.1 Introduction . . . 68
6.2 Smoothing Techniques for Accurate Visualization Techniques . . . 69
6.2.1 Smooth First, Bin and Aggregate Second . . . 69
6.2.2 Bin and Aggregate First, Smooth Second . . . 70
6.3 Study Case: ThemeRiver Smoothing Algorithm . . . 70
6.3.1 Layer Building . . . 70
6.3.2 Legibility Problems . . . 71
6.3.3 Wrong period depiction . . . 73
6.3.4 Summary . . . 74
6.4 Slick Graphs . . . 74
6.4.1 Time Series as Data . . . 74
6.4.2 Slick Graphs Binning Algorithm . . . 75
6.4.3 Slick Graphs Smoothing Algorithm . . . 75
6.4.4 Encoding the Filtered-out Information . . . 76
6.4.5 Use Case: Slick Graphs as a Low-Pass Filter . . . 77
6.5 User Study: Evaluation of the SLG Smoothing Technique . . . 78
6.5.1 Hypotheses . . . 78
6.5.2 Tasks . . . 78
6.5.3 Participants . . . 80
6.5.4 Experiment data . . . 81
6.5.5 Protocol . . . 81

6.5.6 Results . . . 81
6.5.7 Discussion . . . 85
6.6 Integration with Existing Techniques . . . 85
6.6.1 Stacked Graph . . . 85
6.6.2 Interactive Horizon Graph . . . 86
6.7 Conclusion . . . 89

7 TraceViz **91**
7.1 Introduction . . . 91
7.2 Data . . . 92
7.2.1 Data Storage . . . 92
7.2.2 Statistics and Data Computation . . . 95
7.3 TraceViz Design . . . 96
7.3.1 Design Rationale . . . 96
7.3.2 TraceViz Visualization Principles . . . 97
7.4 TraceViz . . . 98
7.4.1 Layout . . . 98
7.4.2 Initial View Configuration . . . 99
7.4.3 Trace Exploration . . . 100
7.4.4 Pan and zoom . . . 101
7.4.5 Actor Selection and Aggregation . . . 101
7.4.6 Hierarchy Reordering . . . 101
7.4.7 Implementation . . . 102
7.5 Industrial Use Cases . . . 102
7.5.1 Use Case 1: Zap . . . 103
7.5.2 Use Case 2: HDMI black-outs . . . 104
7.6 Industrial Deployment . . . 106
7.6.1 STMicroelectronics Toolkit . . . 106
7.6.2 The FrameSoC platform . . . 108
7.6.3 TraceViz Architecture . . . 108
7.7 Conclusion . . . 110

8 Hidden Structures at a Glance **111**
8.1 Introduction . . . 111
8.2 Definitions and Notations . . . 112
8.2.1 Basic Definitions . . . 113
8.2.2 Structure . . . 114
8.3 Structure Computation . . . 116
8.4 Structure Visualization . . . 117
8.4.1 Goals . . . 118
8.4.2 Structures Overview . . . 118
8.4.3 Visualizing Structure Details . . . 119
8.5 Experiments . . . 121

8.5.1 Execution Traces . . . 121
8.5.2 CPython Git Repository . . . 123
8.5.3 Foundation Series . . . 125
8.6 Conclusion . . . 127

9 Study of an Integrated Debugging Workflow 129
9.1 Introduction . . . 129
9.2 Example of an Analysis Workflow . . . 130
9.3 Use case: TSRecord . . . 130
9.4 Conclusion . . . 134

10 Conclusion 135
10.1 Contributions . . . 135
10.1.1 A Smooth Visualization Technique for Time Series . . . 136
10.1.2 A Visualization Framework for Execution Traces . . . 136
10.1.3 Discovering Hidden Structures . . . 137
10.2 Future Work . . . 137

Bibliography 139

III French Summary 151

1 Introduction 153

2 Contexte Industriel 157

3 Etat de l'Art 159

4 Challenges autour du Débogage de Traces 163

5 Approche de Recherche et Méthodologie d'Evaluation 165

6 Slick Graphs: Visualisation Lisse de Séries Temporelles 167

7 TraceViz 169

8 Structures 171

9 Etude d'un Environnement Intégré de Débogage 173

10 Conclusion 175

List of Figures

1.1 This work is at the crossing intersection between Information Visualization, Data Mining and Embedded Systems. 5

2.1 Television standard resolutions (in pixels) and video formats 12
2.2 Block diagram of the STMicroelectronics Monaco MPSoC for set-top boxes 14
2.3 Simplified set-top box architecture for decoding multimedia stream . 15
2.4 GStreamer architecture overview [GStreamer, 2016] 16
2.5 Performance tool included in the Firefox development tools to analyze the execution time of the different piece of Javascript embedded in a web page. 18

3.1 Early version of a line graph and a bar chart 24
3.2 Famous charts made by the English scientist Joseph Priestley 24
3.3 Combination of a line graph and a bar chart by William Playfair in 1786 25
3.4 Paper-based visualization showing the loss of soldiers, their position and the temperatures during Napoleon's Russian Campaign. Made by Charles Minard in 1886. 25
3.5 Enhanced Interactive Spirale [Tominksi and Schumann, 2008] 26
3.6 Circular Silhouette Graph [Harris, 1999] 27
3.7 Circos visualizes multivariate data mapping radially the time [Krzywinski et al., 2009] 28
3.8 Time Curve folds a line graph to position closely similar points [Bach et al., 2015] 28
3.9 With *shared-screen* techniques, the graphs share the space and with *split-screen* techniques, the space is equally divided between the graphs. 29
3.10 ThemeRiver. It visualizes topic density variations across time. [Havre et al., 2000] 30
3.11 Different layout algorithms to stack the time series [Thudt et al., 2016] 31
3.12 Construction method of a Braided Graph [Javed et al., 2010] 32
3.13 Horizon Graph. The original line graph has been sliced into four bands below and above the baseline. The bands have been wrapped to reduce the vertical space. [Heer et al., 2009] 32

3.14 Ripple Graphs, a multi-scale time series visualization technique [Cho et al., 2014] . . . 33
3.15 SignalLens proposes a focus+context technique for time series [Kincaid, 2010] . . . 34
3.16 ChronoLenses is an interactive analytic tool based on lenses [Zhao et al., 2011b] . . . 35
3.17 Stack Zooming [Javed and Elmqvist, 2010] . . . 36
3.18 TimeNotes [Walker et al., 2016] . . . 37
3.19 Line Graph Explorer [Kincaid and Lam, 2006] . . . 38
3.20 Stroscope [Cho et al., 2014] . . . 39
3.21 TimeSearcher [Buono et al., 2005] . . . 41
3.22 Relaxed Selection Query on Time Series [Holz and Feiner, 2009] . . . 42
3.23 Ocelotl provides a trace overview using hierarchical and temporal aggregation [Pagano et al., 2013] . . . 44
3.24 ExplorViz is a treemap-based to visualize Java programs execution [Fittkau et al., 2013] . . . 45
3.25 Outline View provided in KPTrace . . . 46
3.26 With no aggregation technique, visual artifacts quickly appear when working on huge traces with Gantt Chart. Here is an example of the KPTrace view. . . . 47
3.27 Smart Traces shows several Gantt charts simultaneously, one color corresponding to a module [Osmari et al., 2014] . . . 48
3.28 Multiple views configuration to visualize the differences between two traces [Trümper et al., 2013] . . . 49
3.29 Flame Graph showing the call stacks during a program execution [Gregg, 2016a] . . . 50
3.30 ExtraViz is composed of a circular view that shows the method calls and a vertical time [Cornelissen et al., 2007a] . . . 51
3.31 Ravel. In (a) the events are ordered according to the logical time and (b) is based on the physical time. The logical time clearly make more apparent execution patterns. [Isaacs et al., 2014a] . . . 52
3.32 Visualizing frequent itemsets [Yang, 2003] . . . 53
3.33 Itemset visualizer using polylines (a) and lines (b), (c) and (d) [Carmichael and Leung, 2010] . . . 54
3.34 Circular layout and edge bundling visualization technique for frequent itemset. [Bothorel et al., 2013] . . . 55
3.35 Powerset Viewer visualizes frequent itemset [Munzner et al., 2005] . . . 55

6.1 ThemeRiver layer building. The timeseries t is split into n time windows of duration d. A statistic is computed for each time window. It gives the data points p_0, p_1 and p_2. Two consecutive data points are linked using a Bézier curve. The control points p_{na}, p_{nb} are placed horizontally on the time window boundaries. . . . 71

6.2 The graph shows a local minimum instead of the local maximum present in the input. The gray color is the histogram representing the raw data. The blue curve is what the final user will see and is the result of the smoothing method. The red dashed lines are the boundaries of the time windows. 72
6.3 Impact of the position of the time windows on the shape of the curve. The histogram represents the raw data. The blue curve is the result of the smoothing method. The red dashed lines are the boundaries of the time window. 72
6.4 Inaccurate representation of a periodic signal 73
6.5 Building of the histogram H for Slick Graph 75
6.6 Using a Gaussian as kernel, SLG can reveal low frequency patterns by increasing the smoothing factor from (a) to (d). 77
6.7 Task *Perception*. The graph in the middle is a line graph. Top and bottom graphs are either STG or SLG and their position is randomly swapped at each trial. The three graphs represent the same data. STG and SLG apply a smoothing that are equivalent. The red lines follow the mouse and are vertically aligned to help the comparison between graphs. 79
6.8 Task *Same*. Graphs are grouped in five blocks. In each block, the graph of the top is the reference, duplicated five times in total. Among the five other graphs, the participants had to find which represented the same data than the reference. The graph that was currently explored was highlighted. 80
6.9 Impact of the smoothing technique and of σ on the different dependent variables for the *Maximum* task. Error bars are 95 % CIs. 82
6.10 Mean completion time for the *Period* task. 83
6.11 Mean value error for the *Period* task. 84
6.12 Mean completion time for the *Same* task. 84
6.13 Mean correctness for the *Same* task. 85
6.14 Data smoothed with (a) STG algorithm, and (b) SLG algorithm. SLG reveal more details. 86
6.15 Narrow peak corresponds to a sudden number of tweets being emitted at the end of the talk. 87
6.16 Stream Graphs visualizing the volume of tweets emitted during SOTU 2015 88
6.17 Impact of the smoothing factor σ on IHG. The time series becomes very difficult to read when the zoom increases. Increasing σ details are filtered out and the average angle of the slope decreases, making the graph more legible. 89

7.1 Read time of 20000 events when filtering on the time window, the actor and the event type. 93

7.2 TraceViz visualization principles. 97
7.3 Building of the histogram $hist_a$ for an actor a 98
7.4 Overview of TraceViz. TraceViz interface consists of three main areas: the tree view (a), the outline view (b), the timeline view (c) and the links that connect the actors and their corresponding graphs (d). . . . 99
7.5 Initial View Configuration of TraceViz where the developer can filter the actors to show or hide and which statistics to start with. 100
7.6 Patterns appearing on the timeline for the use case Zap 103
7.7 Outline with SLG shading. The shading helps to visualize the periodicity of the behavior thanks to regularly spaced black bands. 104
7.8 TraceViz showing an execution when video blanks appeared. The system is artificially loaded with heavy some I/O using the `dd` Unix command, represented in orange. The task *jbd2-sda1-8* is scheduled directly after the *dd* task, causing delays on the treatment of the VSync IRQ callback, in the red rectangles. 105
7.9 The *jbd2/sda1-8* task is scheduled after the *dd* task (in brown). . . . 105
7.10 When the *dd* task (in brown) is unscheduled, the *jbd2-sda1-8* task (in blue) loads the CPU, causing a delay on the treatment of the callback for the VSync IRQ on the main output (in purple). 106
7.11 SoC Traces & Profiling Toolkit (STPTK). Two views are shown: the time chart (on the top) and the Outline View (on the bottom). 107
7.12 FrameSoC interface. It shows a statistics about the event producers instances as a pie chart (top left) and a tabular view (top right). On the bottom is the time chart provided by FrameSoC. 108
7.13 TraceViz architecture in STPTK and FrameSoC 109

8.1 Visualization of a structure. The root of the diagram is the itemset X of the structure. Each branch corresponds to one of its specialized sequence S_X that occurs at least once in the dataset. The thickness of the first and second segments respectively encode $supp(S_X)$ and p_X. The branch colored in red represents the structure currently highlighted by the user while exploring the data. On the top left are rendered all the items belonging to the itemset of the structure. . . . 120
8.2 Structures of an execution trace. 122
8.3 Periodic behavior of the interrupt 146 shown on the structure overview and in details. 123
8.4 Structures extracted from Git repository of the CPython project . . . 124
8.5 Structure showing two developers committing at a high rate simultaneously. 125
8.6 Visualizing the structures in the “Foundation” series from Isaac Asimov 126
8.7 Dominant structure at the beginning of the text involving the character Hari Seldon and the planet Trantor 127

9.1 Example workflow integrating TraceViz and the visual analytic tool. . 130

9.2 TSRecord trace visualized in TraceViz. 131
9.3 TSRecord trace visualized in TraceViz after having filtered-out irrelevant actors. 132
9.4 Structures computed on the TSRecord trace. 133

6.1 Slick Graphs 167

7.1 TraceViz 169

8.1 Visualisation de structures dans une trace d'exécution 171

Chapter 1

Introduction

Contents

1.1 Motivation and Approach 2

1.2 Contributions 4

1.3 Scientific Context 6

1.4 Thesis Outline 7

June 18^{th} 2016, a Saturday unusually cold for this period of the year. This is the second week of the men's soccer European championship, a hugely popular sport event followed by millions of people in Europe, here in France. It tracts a huge attention from the media: the matches are widely broadcast on television and the radio is flooded by comments on the players' performance. With all this traction, a no less mythic sport event is about to start in a much greater silence: 24 Hours of Le Mans. At the start of the grid, the most powerful prototype cars, the LMP1 category, are crossing the start line at 5:00pm, beginning to race with an average speed of 250 km/h during 24 hours. These cars are pushing the boundaries of the automotive technology, packing a total of 1000 horsepowers where about half is delivered by an electric powertrain in parallel of a more conventional combustion engine, reducing consequently the gaz consumption. 23 hours 56 minutes later, a Toyota car is leading, closely followed by a Porsche car and are about to start the final lap. You carefully follow these last moments even though the Toyota car seems to have won this race: the Porsche team is not going to take any risk and prefers to secure the second place. Suddenly, audio and video glitches occur for a couple of seconds. When the image comes back, the Toyota car is rolling slowly on the side of the track, all lights off, letting the victory escaping with the Porsche car. You missed the key moment of the race. To this point, it does not matter that no problem occurred during the viewing of the whole race before. It also does not count that you could follow the race during an almost complete day, sporadically zapping on other channels to follow the scores of the different soccer matches. All you remember are these couple of seconds of failure that spoiled the last moment of

the race, *computers never work!*

The cause of all this drama is no more than a bug in the software in charge of decoding the multimedia stream received for the television. In this thesis, we propose novel techniques to support the software developers in investigating and resolving bugs on streaming multimedia applications on embedded systems.

1.1 Motivation and Approach

We explain in this section what motivates this thesis. First, we describe the modern embedded systems for multimedia applications and the challenges the semiconductors have to tackle when developing them. Second, we present the research problems raised by the industry and our approach to investigate them.

1.1.1 Challenges in Embedded Systems and Multimedia Applications

The scenario given in the introduction is a simple example of consuming multimedia content. Nowadays, we have access to a very large collection of musics, videos and games that are available online and reachable in only a couple of seconds or minutes at most whether we are at our home or anywhere else with an Internet connection. The quality of the contents has never been better and is constantly improving. For instance, the 4k movies hit the market in early 2015 and then are becoming increasingly popular since. Simultaneously our devices pack an incredible amount of computational power while reducing their energy consumption and increasing their battery life. The smartphones in our pockets are more powerful than the computers we used couple of years ago and the borders between these devices are blurring in terms of usage. At home, our set-top boxes have turned into a powerful digital center not only able to watch high definition television and video-on-demand but also to browse the web, to play video games with appealing graphics, to do video chatting and more. All this technology also become much easier to use providing more reliability and better user interfaces.

To achieve the level of performance required by modern applications, mobile devices and set-top boxes are built upon highly integrated embedded systems. The hardware become increasingly small (in terms of physical size) to be able to reduce the energy consumption and the heat generation, thus rising the computational power output and increasing the battery life. Before, a single chip was dedicated to a single task. Now, modern systems use Multiprocessor System on Chip (MPSoC) to push further the integration of the different hardware components.

A MPSoC embeds very heterogeneous types of chip such as a Central Processing Unit (CPU), a Graphical Process Unit (GPU) used for games and 3D applications but also dedicated chips for audio and video decoding and to manage the connectivity of the system. MPSoC for mobile devices also pack different sensors such as an

accelerometer, a gyroscope and a GPS receiver.

Each new generation of MPSoC are better in terms of performance and energy consumption to enable the constructors such as Apple and Samsung to provide innovative devices delivering a better experience to the end users. Consequently, the hardware platforms become increasingly complex to design as time goes.

On the other hand, the software complexity grows significantly over time. Modern softwares have become heavily parallel to fully use the computational power of the different cores of an MPSoC. An industrial consequence of complex hardware and software products is the rising time and cost to develop and verify for such systems.

Reducing the time-to-market is mandatory for the companies to be able to tackle the aggressive time schedule needed to stay competitive in a rapidly evolving market. Multimedia applications comes with several specificities: QoS properties have to be satisfied to guarantee a smooth playback. Therefore, we distinguish two different types of bugs that can appear on a multimedia application:

1. **Functional bug.** A functional bug corresponds to the case where the software outputs an incorrect result according to a pre-defined specification.

2. **Temporal bug.** A temporal bug is typically a performance issue: the system is not able to satisfy a given time constraint. In the case of a multimedia application, 30 frames have to be decoded every second. A temporal bug happens when the decoding application is slower than the specified output frame rate.

While functional bugs can be solved using traditional debuggers, temporal bugs are more challenging for two main reasons. First, the real-time properties they have to respect makes irrelevant traditional debuggers which pause the execution, completely breaking the QoS properties of the application. Second, temporal bugs tend to appear in the last phase of the software development, during the integration. Indeed, they mostly appear due to bad a synchronization and/or communication between several components of the system, making them impossible to detect before the integration step. For this reason, temporal bugs also appear after the delivery of the development boards to the customers when they implement their own software layer. Therefore, software developers in charge of solving the temporal bugs work with a very limited amount of time and under a high level of pressure, put on one side by the customer and on the other side by the semiconductor company. In this context, developing bug free software becomes a very challenging task.

Therefore, having efficient development and debugging tools is critical for companies to deliver to the market efficient and robust software in a short amount of time.

1.1.2 Research Approach

As explained above, when debugging streaming multimedia applications, traditional debuggers do not work as explained above. A better approach consists in profiling the application at runtime and analyze the data after the execution. This process is named **tracing** [Prada-Rojas et al., 2009; Castro et al., 2011]. All the events that occurred during the execution of the applications are recorded and stored in log file called **execution trace**. For each event, several pieces of information are saved such as its timestamp and its type. Based on the information contained in the execution trace, the developers look for the source of a temporal bug. This debugging technique is not intrusive: the perturbations produced on the application are minimal and the execution is run normally. However, when working with modern systems, the amount of data generated during a single execution is huge and makes the analysis of the trace slow and fastidious.

For an efficient debugging process, software developers need debugging tools that allow them to quickly spot behavioral and temporal patterns. Doing so makes possible to filter out redundant information to focus more quickly on suspicious data. Two strategies are possible:

1. using automatic analysis techniques able to detect temporal anomalities based on data mining. Examples of such approaches include the detection of periodic behavior [Lopez Cueva et al., 2012] and QoS violations [Iegorov et al., 2015]; and

2. using visualization techniques to present to the developers meaningful insights and to support them in the exploration of huge execution traces.

In this doctoral work, we are interested in providing novel interactive visualization techniques for execution traces. Our goal is to propose new debugging tools in which the behavioral and temporal patterns appear clearly. We also focus on supporting the developers in the discovery of global and local trends in the data, making easier the filtering process. To achieve this, we want to integrate data mining results into the visualization tool to leverage their usage and being able to detect temporal and behavioral patterns and outliers.

Therefore, this thesis is located at the intersection of three domains: data mining, information visualization and embedded systems (Figure 1.1).

1.2 Contributions

In this thesis, we propose novel visualization and visual analytics techniques able to tackle a huge amount of data to support the software developers in quickly gaining insights on the execution of multimedia applications on embedded systems with execution traces to spot temporal behavioral patterns and easily filter-out irrelevant data.

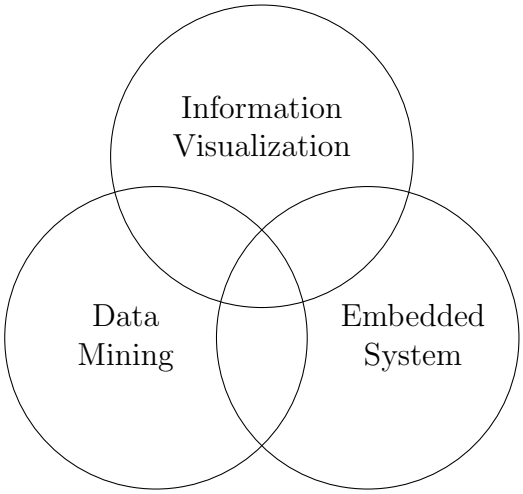


Figure 1.1: This work is at the crossing intersection between Information Visualization, Data Mining and Embedded Systems.

Our contributions consists in three points:

1. **Slick Graphs**. The software developers visualize time series to analyze execution traces, very frequently as an histogram showing the CPU load or the event density. However, time series are often shown smoothed to make them easier to read [Bar and Neta, 2006]. The related work shows that the traditional smoothing technique used by many visualizations for time-oriented data relies on binning values in small numbers of time intervals and interpolating smoothly between those values. It introduces many artifacts and makes the visualization not suitable for precise rendering. When working with execution traces or in a scientific context, a maximal visual precision is required: with a high aliasing, periodic behavior may be hidden and local extrema may be inaccurate. Our contribution, a novel visualization technique for smoothed time series called Slick Graphs, mitigate those quantization artifacts by using the smallest possible binning intervals, i.e. pixels. They nonetheless provide smooth variations by using a convolution with a kernel.

2. **TraceViz**. There exists many visualization tools to analyze execution traces but they have reached their limits with the amount of data generated by modern applications. They either provide a too generalized representation to be useful, or they show too much details leading to a fastidious data exploration. There is a huge need for an integrated debugging environment that enables the developers to quickly browse, search and filter-out the huge amount of data generated at each execution. We propose a novel interaction visualization framework to address these problems. In particular, we present a new fast back-end suitable for the interactive browsing of huge traces and a new visualization technique to explore the trace at different levels of details. This framework takes advantage of the performances of the back-end and integrates

a Slick Graph to achieve an accurate rendering, mandatory in such context to not mislead the developer in making wrong assumptions.

3. **A novel visual analytics method to visualize hidden structures in execution traces**. There is an increasing need to quickly understand the content of execution traces to shorten the time of analysis due to the competitive market of consumer electronics. A wide range of patterns can be computed and provide valuable information: for example existence of repeated sequences of events or periodic behaviors. However pattern mining techniques often produce many patterns that have to be examined one by one, which is time consuming for experts. On the other hand, visualization techniques are easier to understand, but cannot provide the in-depth understanding provided by pattern mining approaches. Our last contribution is to propose a novel visual analytics method that allows to immediately visualize hidden structures such as repeated sets/sequences and periodicity, allowing to quickly gain a deep understanding of the execution traces.

1.3 Scientific Context

This work is at the intersection of one industry and two research domains. Funded by the ANRT as an industrial CIFRE thesis, the industrial work of this thesis has been done in partnership with STMicroelectronics in the SDT group (Software Development Tools) and the research activities, in the domains of information visualization and data mining, have been respectively done at the Laboratoire d'Informatique de Grenoble (LIG) in the Engineering Human-Computer Interaction (EHCI) group and in the Scalable Information Discovery and Exploitation (SLIDE) group.

The SDT team is in charge of developing new tools for internal software developers as well as for consumers who bought STMicroelectronics platforms. Their purpose is to provide efficient solution for debugging multimedia applications on embedded systems. During most of the PhD, I worked in close collaboration with the software engineers of the SDT team. STMicroelectronics took the decision early 2016 to stop the set-top box activity, thus closing the division Digital Product Group in which the SDT team was belonging. From this point, the SDT team had to freeze all its developments and release of the development tools. The software engineers that were working on the streaming engine for set-top boxes were reallocated to different divisions, only were kept to support the customers in bug resolutions on previously sold products. These elements made impossible the integration of new tools and increased the difficulty to reach any developers that was already high due to several factors such as a heavy workload.

The research in information visualization has been carried out in the EHCI team whose research themes include human-computer interaction and visualization. The data mining work has been conducted with the SLIDE team that focus on efficient large-scale data processing with pattern mining algorithms.

1.4 Thesis Outline

The remaining of this manuscript is composed of eight chapters:

In Chapter 2, we present the evolution of the multimedia standards (Section 2.1) and of the embedded systems for streaming applications (Section 2.2). We give an overview of the hardware architecture involved for decoding multimedia streaming applications (Section 2.3) and then we describe the method to debug streaming multimedia application with real-time constraints and the specificities of debugging such applications (Section 2.4).

In Chapter 3, we present previous works done on the visualization of time series (Section 3.1). Next we focus specifically on the research approaches for visualization techniques for execution traces (Section 3.2) and patterns visualization (Section 3.3).

In Chapter 4, we summarize the main challenges discussed in related work that need to be addressed to provide more efficient debugging tools for execution traces.

In Chapter 5, we explain the approach taken during this thesis and explain the motivations behind the different choices we made to evaluate our contributions.

In Chapter 6, we introduce Slick Graphs, our novel technique for smooth and accurate visualization of time series. We explain the different strategies available for smoothing data in visualization (Section 6.2). We analyze the legibility issues of previous approaches (Section 6.3) before proposing our smoothing algorithm for accurate visualization of time series (Section 6.4). We present the user study to evaluate the benefits of our technique (Section 6.5) and show how to integrate with existing methods (Section 6.6).

In Chapter 7, we describe a new visualization framework for execution traces, TraceViz. The algorithm developed for the Slick Graphs serves here as the foundation building brick for TraceViz for its new representation of traces. We introduce a new high performance back-end that enables interactive exploration of huge traces (Section 7.2). We describe in Section 7.3 and 7.4 the tool built on top that provides an overview yet detailed enough to spot periodic and anomalous behaviors. We present two real world use cases for which TraceViz provided relevant information (Section 7.5) and how it has been integrated into the STMicroelectronics development toolkit (Section 7.6).

In Chapter 8, we present a visual analytics method to discover hidden trends and perturbations in logs. This work consists in giving an other type of overview than the one provided by TraceViz. Both the technique presented in this chapter and TraceViz can be seen as complementary visualization technique for trace analysis. After having introduced and defined the notion of structure (Section 8.2), we explain the algorithm to compute them (Section 8.3) and introduce the tool to visualize them (Section 8.4). Section 8.5 of this chapter shows experiments on various types of data to illustrate the relevance of our approach.

In Chapter 9, we show an example of integration of the works presented in the previous chapters based on a use case. After having described the integration and the potential workflow suitable for the use case in Section 9.2, we analyze the

execution trace to demonstrate how this workflow works in Section 9.3.

Part I

Background

Chapter 2

Multimedia Applications on Embedded Systems

Contents

2.1 Evolution of Video Standards 12

2.2 Evolution of the Embedded Systems 14

2.3 Focus on Hardware for Multimedia Decoding 15

2.4 Decoding Multimedia Streaming Applications 16

2.5 Debugging Multimedia Applications on Embedded Systems 17

2.6 Conclusion 21

New video formats and higher resolutions provides a better video quality to consumers. Nowadays, many devices, from set-top boxes to smartphones and tablets, are able to decode high definition medias and allow the users to widen their usage. On-the-go, mobile devices run games with high graphics quality, embed cameras that can compete with dedicated cameras and enable video chatting. On the other side, consumers request good battery life and very compact designs.

In this chapter, we show the evolution of the multimedia standards and the modern usages that appeared recently (Section 2.1) and explain how it has impacted the embedded systems architectures (Section 2.2). To ensure the delivery of good performances, multimedia applications require specific hardware (Section 2.3) and embedded software for streaming application has its own specificities (Section 1.4). Therefore, traditional tools are not relevant and developers need specific tools to test and debug these platforms (Section 2.4).

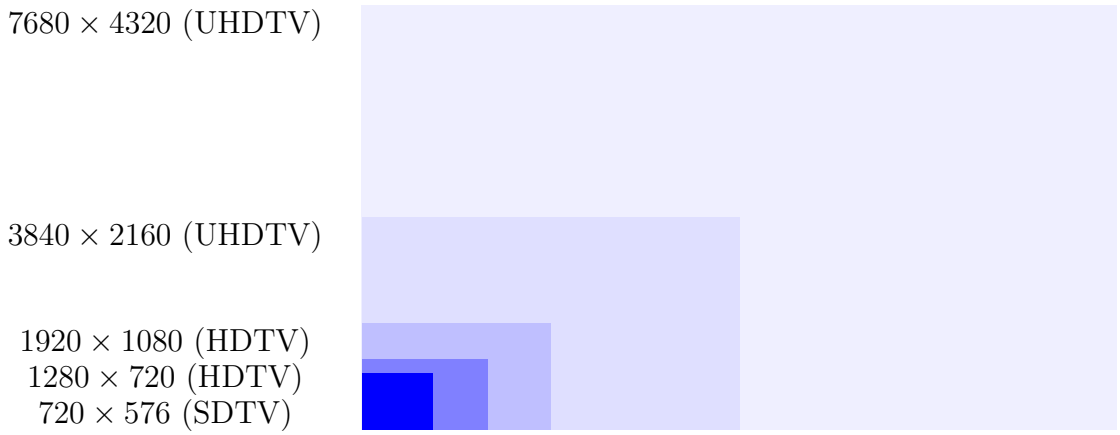


Figure 2.1: Television standard resolutions (in pixels) and video formats

2.1 Evolution of Video Standards

Smartphones, tablets, set-top boxes and connected televisions are some examples of devices broadly used on a daily basis for an ever wider range of digital activities. Consumers now expect these devices to provide a high level of performance to produce or consume multimedia contents (audio, videos, photographies) and to play games with a high quality of graphics only achievable on a personal computer couple of years ago. These applications push the hardware to its limits and need a lot of energy to run. To guarantee the longest autonomy possible, mandatory for mobile devices, embedded software has to be optimized for the underlying hardware.

Set-top boxes are a typical example of widely used devices that have followed a dramatic evolution in the last decade. Originally, a set-top box is a device that receives a digital stream for television, decodes it and sends it to a display device such as a television in most cases. During the last decade, television resolutions and formats have improved quickly. Figure 2.1 depicts the evolution of the resolution across time. At the beginning of the digital television, the set-top boxes had to decode a stream compressed using the MPEG-2 standard before sending it to an output device that will play it (i.e. a television or a monitor). The transition from analog to digital television began in the early 2000s and was complete in most developed countries in the early 2010s. Different low resolutions were supported by the first generation, ranging from 640×480 to 720×576 pixels depending on the countries, and were regrouped under the format named *standard-definition television* (SDTV). Quickly, a new video format arrived to provide a better image quality with higher resolutions: the *high-definition television* (HDTV) that supports the 1280×720 and 1920×1080 pixels resolutions, commonly named *720p* and *1080p*. The latest evolution in this domain is the introduction of the *ultra-high definition television* (UHDTV) that includes the *4k* standard (3840×2160), that reached the consumer market in early 2014, and the *8k* (7680×4320 pixels) standard for which the first highest-end televisions appeared on the market in early 2016.

To support this evolution, new standards for compression algorithms and video containers have been developed. For the HDTV video format, the MPEG-4 AVC (Advanced Video Coding), or H.264, compression algorithm was introduced [Wiegand et al., 2003]. The H.264 has been designed to work with many different types of application including online streaming and broadcasting. To achieve this, the standard supports high and low bit rates and can work with lower than HD definitions. Consequently, it is the most broadly spread video compression algorithm in the industry, from BluRay discs, HDTV broadcasts and many different web content providers such as Youtube [Youtube, 2016] and Vimeo [Vimeo, 2016]. The successor of the H.264 is the newer H.265, or High Efficiency Video Coding (HEVC), video encoding standard [Sullivan et al., 2013]. Compared to the H.264, the H.265 offers a better video compression and supports the resolutions up to the 8k for the UHDTV. It has reached the market in early 2016 but there exists few digital contents to take advantage of this new standard at the moment of writing.

We briefly described the evolution of the standards for both the physical properties of the devices (i.e. the resolution) and for the video compression standards to provide digital content with a better quality. These compression algorithms become more sophisticated at each iteration. A direct consequence is the need of more powerful hardware platforms and an increasing software complexity that are able to tackle the real-time decoding of these multimedia streams and to implement the latest standards. In parallel, the size of the television screens themselves has greatly increased to take advantage of the new resolutions. They are nowadays providing a great user experience when watching movies and for many different activities such as gaming, video conference, etc. In a short amount of time, the activity of the set-top boxes went from decoding a low resolution multimedia content to decoding simultaneously several high resolution streams and broadcasting the media to different outputs in the home. The set-top boxes also have to support gaming activities and different other usages aforementioned. These activities will become even more important in the upcoming year.

Set-top boxes are not the only devices that followed this evolution. Smartphones have followed the same path to become mobile devices that pack the same amount of computational power than the computers from a couple of years ago. More recently the part of hardware and software embedded in cars has greatly increased. Modern cars now pack new features to provide different levels of assistance such as parking assistance or automatic door unlocking when the owner arrives at proximity. Few has reached the market with even more advanced features such as fully automatic parking and meeting point (i.e. the car is able to connect to the garage door, open it, park and close it - it also comes with the capacity of analyzing the user's calendar and be warm and ready on time with no user actions) and with full autonomous driving capacity[1]. Similarly to multimedia applications, embedded systems in cars have to analyze in real-time a large amount of information, and more specifically implement computer vision algorithms to analyze the video stream recorded by

[1]Tesla Motors: `http://teslamotors.com`

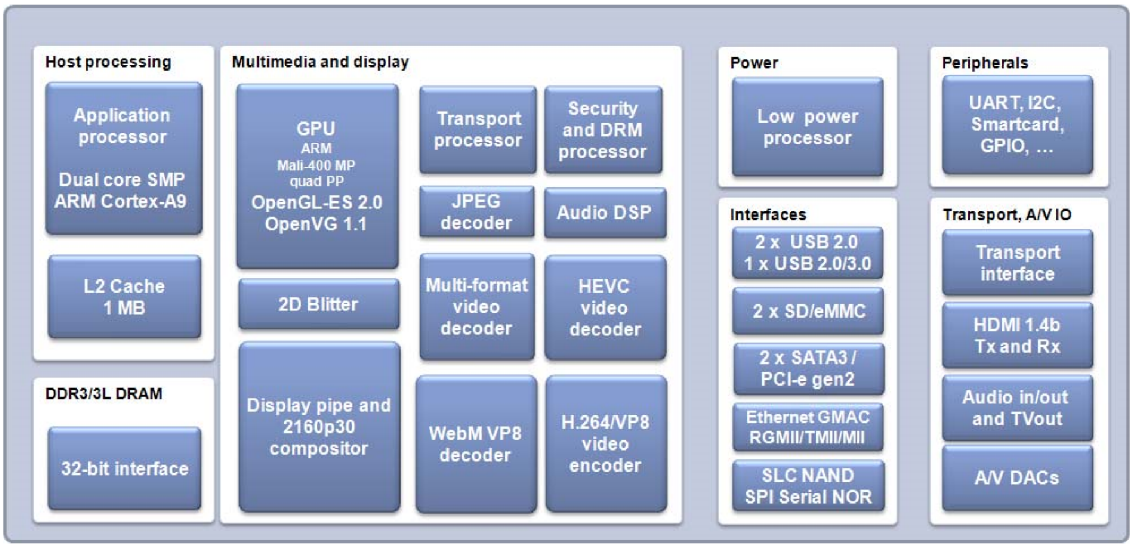


Figure 2.2: Block diagram of the STMicroelectronics Monaco MPSoC for set-top boxes. Each block represents a dedicated chip for a specialized task.
Source: `http://hackerboards.com/set-top-box-socs-move-up-to-cortex-a9-ultrahd-hevc/`

outside cameras, to be able to react quickly enough to guarantee a high-level of reliability. These examples show the increasing need of efficient tools to develop and debug real-time embedded softwares for more powerful hardware platforms.

2.2 Evolution of the Embedded Systems

The new usages and improved multimedia user experiences are supported by new generations of hardware released every couple of years making the devices more powerful with a lower energy consumption. These increased performances are possible thanks to highly integrated MultiProcessor System-on-Chip (MPSoC) that embed many specialized processing units for very specific tasks such as audio and video decoding. The main components of a MPSoC are a Central Processing Unit (CPU), memories, specialized hardware chips - or *accelerators* that can include a Graphics Processing Unit (GPU), Digital Signal Processors (DSP) etc., external connectors (Ethernet, USB, JTAG and so on) and a bus to connect these components. Wolf et al. provides a deep explanation of the MPSoC technology and the challenges specific to their design [Wolf et al., 2008].

The increasing complexity of MPSoC implies a higher cost and a longer development time, increasing the *time-to-market*. To keep proposing efficient solutions in a very competitive market, the conception methodologies has evolved. Nowadays, rather than designing a MPSoC from scratch, semiconductor companies largely rely on integrated circuit (IC) libraries. They use Intellectual Property (IP) cores on their platform and only develop some specialized chips that they integrate with the

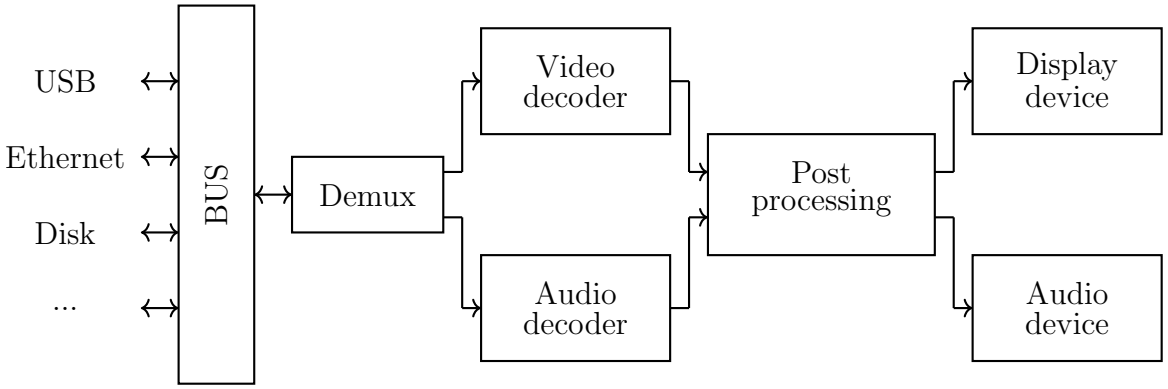


Figure 2.3: Simplified set-top box architecture for decoding multimedia stream

blocks available on the market into a single chip, a MPSoC. Several companies, such as ARM, sell these IP cores and tend to become standards as the number of systems integrating them increases.

STMicroelectronics designs and produces MPSoC for set-top boxes and develop the software layer to exploit them. The last generation, the *STiH412 'Monaco'* series [STMicroelectronics, 2016b], is composed of many cores dedicated to a large panel of different specific tasks to satisfy the modern usages (see Figure 2.2 for the block diagram). The main processor, a dual or quad-core core ARM Cortex-A9 processor [ARM, 2016b], is a general processor for set-top boxes, smartphones, tablets, etc. delivering enough computing power to handle the different tasks executed concurrently. The multimedia decoding activities are performed by several specialized processors, each of them supporting one or several specific video formats and encodings. For example, the processor under the name *Display pipe and 2160p30 compositor* is in charge to decode in real-time a *4k* stream from the television or other sources available through Video-on-Demand (VoD) services (i.e. Netflix [Netflix, 2016], Vimeo [Vimeo, 2016], etc.). The *HDTV* streams are decoded by the *Multi-format video decoder* processor. An other specificity of modern platforms that reflect the expansion of the usages is the appearance of a dedicated processor to encode videos for the purpose of video chatting (using Skype [Skype, 2016] or Google Hangouts [Google, 2016]). As explained above, gaming is an other growing activity on set-top boxes and a dedicated GPU is required. Here, an ARM Mali 400 GPU has been integrated [ARM, 2016c]. This GPU supports OpenGL ES 2.0 and provides enough power to provide a good gaming experience with appealing graphics.

2.3 Focus on Hardware for Multimedia Decoding

Decoding a stream is a complex task that involves itself several processing units (see Figure 2.3 for a simplified architecture of a basic set-top box). As aforementioned, the multimedia sources can be of different nature. After reception, the first step

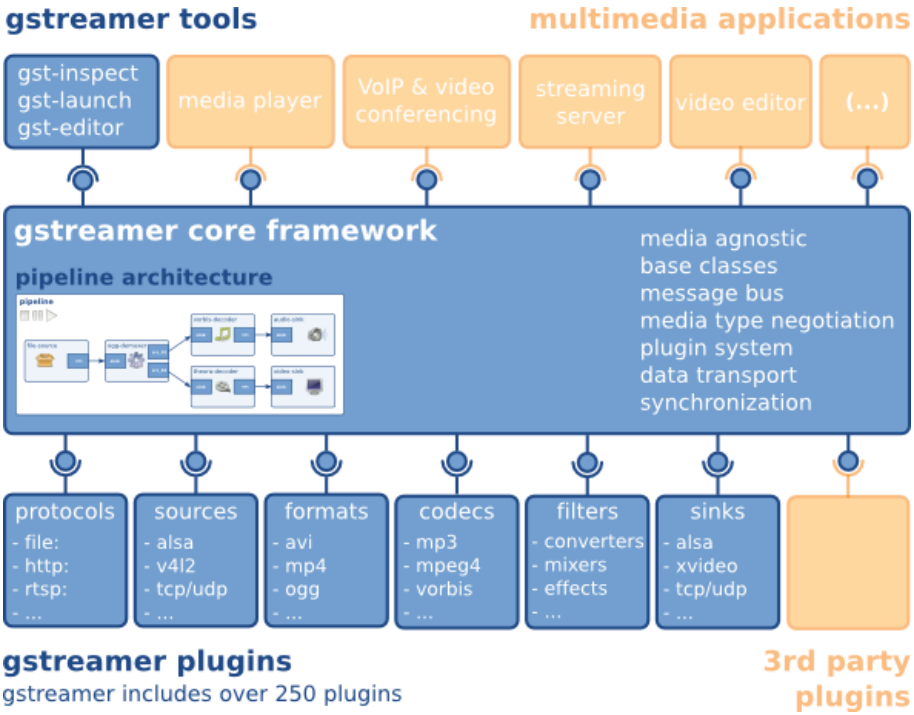


Figure 2.4: GStreamer architecture overview [GStreamer, 2016]

consists in separating the audio and video data streams according to the video format container. This step is performed by a demultiplexer and the media streams are sent to the appropriate decoders. The audio and video decoders work in parallel. The video decoder, for example the *HEVC video decoder* process unit on Figure 2.2 stores the frames into a buffer. When working with HEVC decoders, the video frames need to be reordered before being sent to the post-processing step [Sullivan et al., 2013]. Post-processing is in charge of scaling the frames and synchronizing them with the audio. It is the final step before sending the decoded streams to the output devices.

While this is a brief overwiew of a typical hardware architecture to decode a multimedia stream, it still gives an idea of the complexity of the decoding process.

2.4 Decoding Multimedia Streaming Applications

To support the newest video compression standards and to fully use the potential of the hardware platforms, the difficulty of developing the software layer constantly raises and brings new challenges. Similarly than for the hardware design, developing from scratch the multimedia applications for each platform became too expensive and time consuming. Instead, developers rely on middleware that implements the decoding pipeline, known as multimedia frameworks and adapt them for each MPSoC. Popular multimedia frameworks include Media Foundation [Microsoft, 2016]

and AV Foundation [Apple, 2016] both platform-dependent, proprietary and developed respectively by Microsoft and Apple. Famous free and cross-platform frameworks include GStreamer [GStreamer, 2016], Phonon [Phonon, 2016] and the VLC framework [VideoLAN, 2016]. All provide default implementations of the video standards and a plug-in mechanism facilitating the integration of the dedicated hardware accelerators. Figure 2.4 shows an overview of the GStreamer architecture. The *core framework* implements the decoding pipeline. It is the software implementation of the pipeline shown in Figure 2.3, used as default. The support of different video formats, encodings and so on is implemented through different plugins. Based on the framework, the end-user applications are implemented independently of any of the video parameters.

2.5 Debugging Multimedia Applications on Embedded Systems

When developing an application, debugging is used for two different tasks: (1) functional debugging and (2) temporal debugging.

First, the implementation has to respect a pre-defined behavior according a specification. When the behavior of the application does not respect the specification, the developers perform a *functional debugging* to find the source of the problem. The functional debugging workflow uses breakpoints. The execution stops at the breakpoints from which the developer can investigate the state of the application. Then, there are two different execution strategies available:

1. continue the execution normally. The analysis of the current state of the execution is correct and the execution must continue to reach the bug.

2. enter in a step-by-step mode where the execution stops at each instruction. The source of the problem is to be reached and a fine-grained analysis of the state of the different variables is required.

Second, even if the application fulfills the specifications, the execution may be slow. In this case, the debugging is used for performance tuning. It is called *temporal debugging*. A common strategy is to instrument the source code with timers to measure the execution time of critical components. This operation is called *profiling*. The execution time can be simply output or displayed using a simple graph such as a bar chart. Web browsers have democratized this technique by providing by default profiling tools to measure the loading time of each component of a web page (Figure 2.5).

For some applications, the functional debugging and the performance tuning can be done separately. In this case, the developer first aims to implement the correct behavior and then, improves the performance. However, when working with real time constraint, the execution has to satisfy a maximal response time. Its behavior is considered abnormal if the time constraint is not satisfied. In this context,

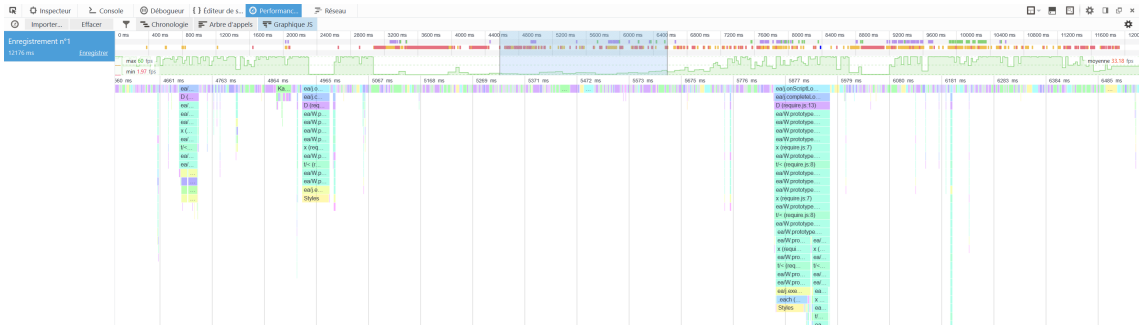


Figure 2.5: Performance tool included in the Firefox development tools to analyze the execution time of the different piece of Javascript embedded in a web page.

using breakpoints to debug the application would break the real time constraints, modifying the behavior of the application.

As explained before, a multimedia application receives one or several encoded streams and has to decode it in real time before sending it to a peripheral that will display the content. The decoding process has to respect some QoS properties so that no audio glitches or video artifacts appear [Bril et al., 2001]. More precisely, to provide a smooth playback, each of the decoding steps has to satisfy some real time constraints. For instance, to prevent video glitches, the decoder has to send 30 frames per second to the display device. Below this limit, a momentarily blank screen may appear or a frame jump may occur, perturbing the user experience. Therefore, traditional breakpoints do not allow to detect the root of a problem since the decoding would be interrupted.

To circumvent this phenomenon, a popular technique is to use execution traces to debug multimedia applications and more generally embedded softwares with real-time constraints. Using traces allows to do a post-mortem analysis of the execution, therefore not to break the QoS properties to respect at the execution time. On top of being less intrusive than interactive debuggers, tracing systems are also widely spread in the industry, leveraging the cost of this solution.

2.5.1 Execution Traces

Execution traces are text files in which the events that occurred during the program execution are saved sequentially. Therefore, traces are time series: a collection of time related events.

Definition 1. *A trace T is a list of events E so that $\forall e_i, e_j \in E, i < j$ if and only if e_i happens before e_j.*

In a trace, each event is unique and characterized by relevant information to understand what occurred during the execution. It is typically composed of the following fields:

- a *timestamp* when the event occurred.

- an *actor* that produced the event. In embedded systems, actors can be of different nature. It can be a process, a kernel module, an interrupt or a software interrupt.
- an event type. It indicates the nature of the event and includes context switches, entry/exit of a system call, a user-land function or an entry/exit of an interrupt.
- a variable number of arguments depending on the event type. For instance, in the case of a context switch, the event can have two arguments: the old and new process identifiers. When the event corresponds to the entry in a function, the arguments can corresponds to the memory address of the function and the arguments given to the function and so on.

Each event is recorded as an entry into the trace log. The developer can configure which components will be traced to mitigate the number of events but in complex application executions and target specific software modules. However, such events occur many times resulting on a huge amount of data generated in a short period of time ($\approx 10^6$ events per minute). Listing 1.1 shows an extract of a typical execution trace. The trace is read as follows:

- Each line represents an event in the trace.
- The first column is the timestamps when each event has occurred. Here the timestamps are measured in milliseconds.
- The second column is the identifier of the process (PID) that executed the instruction.
- The event type is encoded on the third column. In the example, we have a context switch (C), an entry (S) and an exit (s) in a software interrupt, an entry (E) and an exit (X) of a system call and an entry (I) and an exit (i) of an hardware interrupt.

Therefore, the first line represents a context switch from `IDLE` whose process id (PID) is 0 to the process 3 at timestamp 943920468.728392. Then, the second and third line are respectively an entry and an exit in a software interrupt located at the address `0x8059c09c` at timestamp 943920468.728405 and 943920468.728409. At 943920468.728430 ms, an other context switch happened from PID 3 to 2300 and the process 2300 performed a system call at 943920468.728517 located at the address $0x801148b4$ with one argument at $0x000001b6$ in memory. The system call ended at 943920468.728586 and returned -2 as result.

```
[...]
 943920468.728392       0 C   0  3
 943920468.728405       3 S   0x8059c09c
 943920468.728409       3 s
```

```
 943920468.728430      3 C  3 2300
 943920468.728517   2300 E  0x801148b4 0x000001b6
 943920468.728586   2300 X  0x801148b4 -2
 943920468.728648   2300 E  0x80115a20 0x0002c514
 943920468.728697   2300 X  0x80115a20 0
 943920468.728747   2300 E  0x801148b4 0x000001b6
 943920468.728804   2300 X  0x801148b4 -2
 943920468.728862   2300 E  0x80115a20 0x0002c514
 943920468.728907   2300 X  0x80115a20 0
 943920468.728925   2300 E  0x80114924 0x00000002 0x6f5e7494
 943920468.728934   2300 X  0x80114924 0
 943920468.729306   2300 C  2300 0
 943920468.733731      0 I  34
 943920468.733745      0 I  257
 943920468.733771      0 i
 943920468.733776      0 i
[...]
```

Listing 2.1: Extract of an execution trace

Traces are stored according a pre-defined format, standardize in the industry to centralize the efforts on developing debugging tools. The Best Trace Format (BTF) is an ASCII format based on CSV [Architects, 2016]. The Common Trace Format (CTF) [EfficiOS, 2016b], a mutual efforts between industrials and the Linux community and led by EfficiOS[2], aims to become the reference trace format. In parallel, an open source tool, BabelTrace, converts different trace formats into CTF and is extensible through a plug-in system [EfficiOS, 2016a]. STMicroelectronics has developed its own tracing system KPTrace [Prada-Rojas et al., 2009], that comes with its trace format.

2.5.2 Tracing Systems

During the development of applications for embedded systems, development boards provide features to support the developers during the testing and debugging tasks. They come with a specific port (a JTAG or Serial port) used to get debugging information and to control interactively the execution. This port is connected to an external machine, called the *host*. Once the recording of a trace is completed, the files are sent from the development board to the host through a dedicated debugging port.

Running on the board, there are different softwares to capture traces. The Linux kernel provides as standard its own tracing mechanism called KProbes [Krishnakumar, 2005]. Perf is an other tool implemented in the Linux kernel oriented on performance monitoring and system profiling [Carvalho de Melo, 2010]. Linux also

[2]EfficiOS: `http://efficios.com`

integrates a tracing utility named Ftrace and enables to collect events related to the kernel activity [Ftrace, 2016]. There also exists higher level toolkits that allow to collect and aggregate the information coming from different sources (KProbes, Perf, external libraries, etc.). The Linux Trace Toolkit Next Generation, LTTng, is a modern tracing software collections [LTTng, 2016]. It supports kernel-land and user-land tracing while minimizing the system overhead. It makes it a suitable solution for performance analysis of real time embedded software. It also provides tools to analyze the trace and support the CTF format.

STMicroelectronics has implemented its own stack, named KPTrace [Prada-Rojas et al., 2009] and is able to capture kernel-land events such as context switches, interrupts, system calls, etc. as well as user-land events that can be any function call at the application level. KPTrace is also a bench of tools for traces analysis, named SoC Traces & Profiling Toolkit (STPTK) [STMicroelectronics, 2016a].

2.6 Conclusion

In this chapter, we have presented the evolution of video formats and the need of increasing computational power to support the newest video standards delivering a better video quality. The widening panel of usages also imposes the platforms to diversify their capacity to run different types of application. These factors led to the democratization of MPSoC that provides a high-level of performance with a low energy consumption.

A direct consequence is a significant complexification of the development of the software. We have presented the specificities of multimedia applications and the tools used during the debugging process: the execution traces. With more complex hardware and software platforms, the amount of data to analyze for performance tuning and to discover the cause of a bug becomes too large and will keep growing in the future. Under these conditions, the existing analysis tools show their limitations and do not allow an efficient debugging in a short time. Minimizing the time-to-market of a product is fundamental for companies to stay competitive. In this context, the software developers need a new generation of visualization tools able to tackle these huge amounts of data and supporting them in the exploration of the data and in the discovery of patterns and perturbations.

In the next chapter, we review the existing research in three domains related to the visualization of execution traces. As seen above, traces are time series. Therefore, we begin by presenting research work on time series visualization techniques as a fundamental approach. Next, we focus more specifically on work about visualizing traces and describe the existing tools available in the industry. We highlight their limitation when handling a large volume of data. Lastly, after explaining how data mining helps in the analysis of traces, we review previous work on pattern visualization.

Chapter 3

Related Work

Contents

3.1 Time Series Visualization 23

3.2 Visualization of Execution Traces 43

3.3 Pattern Visualization 52

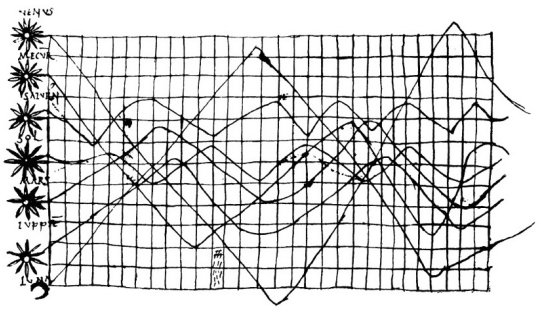
We saw that an execution trace can be seen as a time series. Indeed, we can model a trace as a collection of multiple time series, one time series by actor. Visualizing execution traces to debug multimedia applications requires to have powerful and relevant visualization tools depending on the tasks to perform. Visualizing time series and multiple time series is an active research theme. This chapter first explore previous work on visualization tools for single and multiple time series. These works will serve as background for this thesis.

Second, it is necessary to analyze visualization tools that have been designed specifically for execution traces with the perspective of understanding the behavior of the application across time. We draw a state-of-the-art of the techniques previously developed in academia and in industry for this specific domain.

Proposing a powerful visualization tool is mandatory but integrating results computed by automatic pattern detection algorithms will further support the developers in the debugging process. In the last section of this chapter, we present previous work on pattern visualization.

3.1 Time Series Visualization

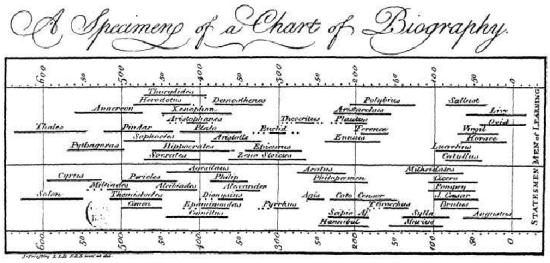
Visualizing time series has the general purpose of helping to understand the evolution of data over time like detecting trends in data, finding recurrent patterns and discovering anomalies. Playfair pioneered this domain in the 18^{th} century [Playfair, 1786] by using the line charts in his work. Often considered as the creator of the line graphs, a first prototype from the 17^{th} century depicts the planetary movement (Figure 3.1a). Earlier, in 1350, the philosopher and mathematician Nicole Oresme



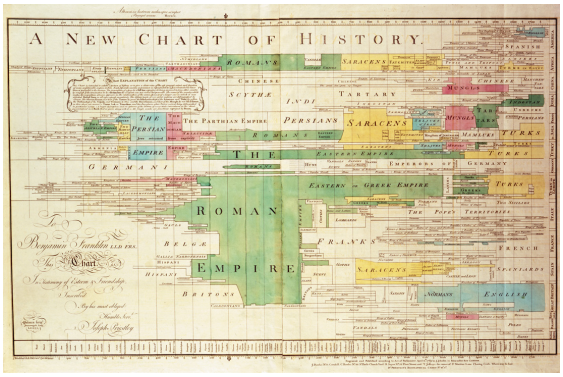
(a) Line graph from the 10^{th} century

(b) Bar chart from 13^{rd} century

Figure 3.1: Early version of a line graph and a bar chart



(a) Biography chart made in 1765



(b) History chart made in 1769

Figure 3.2: Famous charts made by the English scientist Joseph Priestley

used an early version of bar charts to plot the velocity of objects accelerating over time (Figure 3.1b).

The real development of these techniques happened during the 18^{th} century. Joseph Priestley, an English scientist, believed in the power of illustrations to transmit ideas and educate people. Among numerous publications, he designed two famous bar charts about important intellectuals (Figure 3.2a) and history (Figure 3.2b), respectively in 1765 and 1769.

In parallel, the mathematician and philosopher Johann Heinrich Lambert used a line graph to illustrate physic laws in 1767. In 1786, William Playfair used heavily bar and line charts and contributed to make them popular (Figure 3.3).

The stacked graph was invented in 1886 by Charles Minard, well-known for his visualization realized in 1869 that pictures the disaster of Napoleon's Russian campaign using a sophisticated combination of line graphs to show different data variables (Figure 3.4).

After its creation and first development, line charts have gradually gained in popularity to become the most popular technique to visualize time series [Cleveland, 1993] thanks to their simplicity that makes them easy to understand. However, line

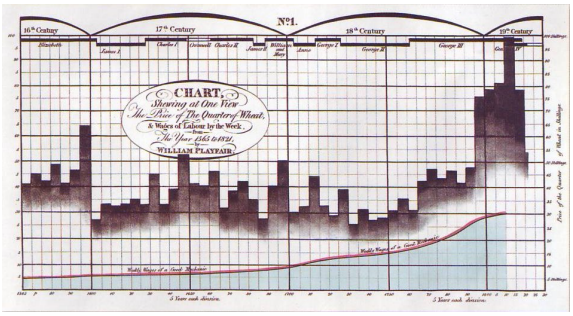


Figure 3.3: Combination of a line graph and a bar chart by William Playfair in 1786

charts have several drawbacks. Firstly, aspect ratio has a critical impact in the perception of a line graph [Cleveland, 1993]. When representing huge time series with a line graph, this implies to either having a very tiny height or forcing the user to scroll horizontally, making the navigation tedious without some form of aggregation. Secondly, drawing a huge amount of information using a line graph results in a visualization that has a high spatial frequency, for which the human eye is less sensitive. Under these conditions, the constraint given by the previous factor cannot be respected. This results in a representation very difficult to read, and can lead to a loss of perceived information.

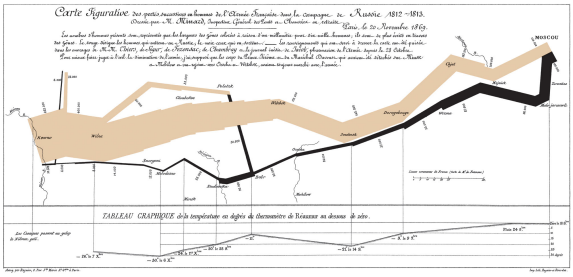


Figure 3.4: Paper-based visualization showing the loss of soldiers, their position and the temperatures during Napoleon's Russian Campaign. Made by Charles Minard in 1886.

3.1.1 Time Representation

When visualizing time-oriented data, correctly representing the time dimension of the data has a critical impact on the final rendering of the visualization but also on its efficiency for handling certain tasks. In their survey, Aigner et al. [Aigner et al., 2011] classify those visualization techniques using three criteria: data, time and visualization. They considered two visual arrangements for the time variable: cyclic and linear.

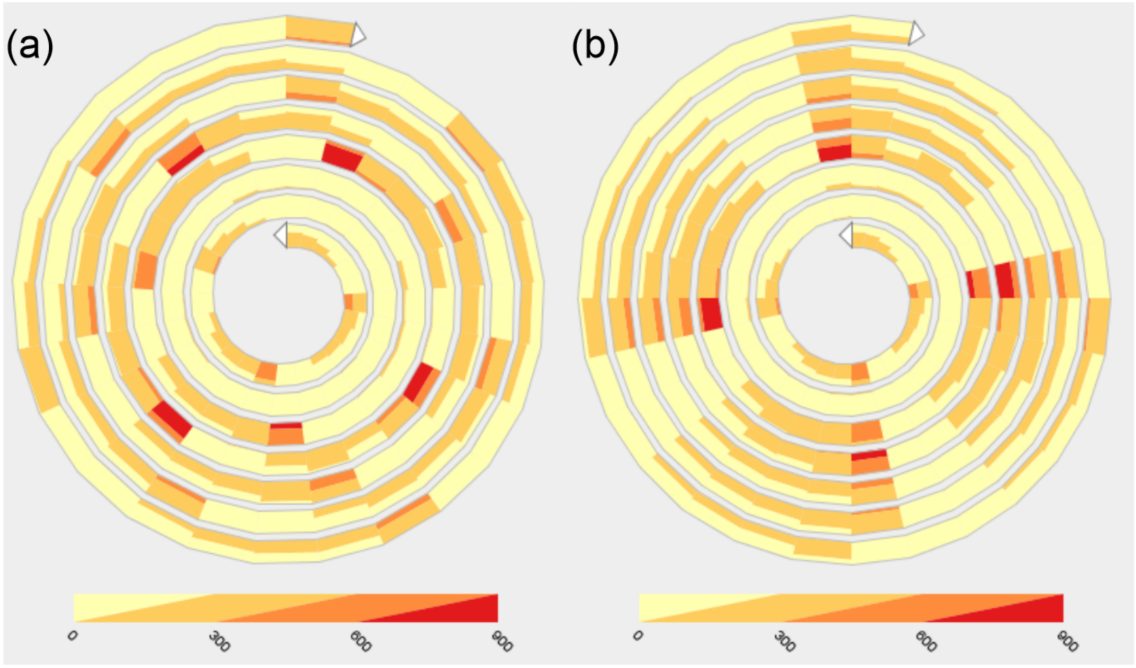


Figure 3.5: Enhanced Interactive Spirale [Tominksi and Schumann, 2008]

Cyclic Time Representation

Cyclic time representation leads to unique visualizations that are efficient for very specific purposes but can require a learning process for the user to understand them. Time can be represented either on a spiral or on a circle.

Spiral disposition Using a spiral enables the discovery of periodicity in the data. This can be done by adjusting the period of the spiral interactively or automatically using animation. Early works [Carlis and Konstan, 1998; Weber et al., 2001] propose such approach and arrange time on a spiral and support interactive data exploration for periodic patterns discovery. Spiral Display [Carlis and Konstan, 1998] arranges the time dimension around a spiral and represents the data points as dots, color spikes or charts. SpiralView [Bertini et al., 2007] applies the spiral layout for plotting security events to detect network attacks. Enhanced Interactive Spiral [Tominksi and Schumann, 2008] mixes Horizon Graphs [Saito et al., 2005] and Spiral Display to implement the overview and detail principle (Figure 3.5). Helix Icons [Tominksi et al., 2005] uses a 3D spiral (i.e a helix) to encode the temporal dimension on a 3D map. Similar to the Spiral Display, the cycle of a helix can be adjusted to discover periodic behaviors.

While disposing the time dimension on a spiral is very useful to discover patterns, it does not scale well as the number of time series to visualize increases and is not suitable when working with a large number of time series as it is the case with execution traces.

Figure 3.6: Circular Silhouette Graph [Harris, 1999]

Circular Disposition Circular disposition is an other option for cyclic time representation where the time dimension is mapped on a circle. As most of work built using a spiral, Circular Silhouette Graphs also aims to discover periodicity in the data but represents time on concentric circles for better periodicity detection [Harris, 1999] (see Figure 3.6). Circular Silhouette Graph looks promising for execution traces: it is suitable for periodic time data and is able to manage several time series by assigning each one on a circle. However, with a large amount of time series, the distortion induced by the curve of the circle makes difficult to read the values of the different graphs. On top of that, the curvature varies depending how far the circle is placed from the center.

Circos is an other proposition that maps the time on concentric circles. It has been designed to visualize genomic data and has the particularity to work with multivariate data [Krzywinski et al., 2009]. A data variable is encoded with a concentric data track, or band, using different techniques such as line graph, histograms, text, etc. (Figure 3.7). The novelty of Circos is its ability to integrate several types of visualizations into an integrated view. In the case of execution traces, the multiple time series need to be compared, thus have to be visualized with the same technique. When used in this configuration, Circos has the same limitations than Circular Silhouette Graphs, discussed above.

Summary Visualization techniques mapping the time dimension on a radial layout performs well to discover periodic patterns in the data. However, for basic tasks like reading or comparing values, the curvature of the time axis adds complexity to the visualization and slows down the understanding process. In the context of execution traces, working with multiple time series is fundamental. With a circular time domain disposition, visualizing multiple time series becomes tedious. Once again, the curvature of the axis encoding the time makes comparing different values a difficult task.

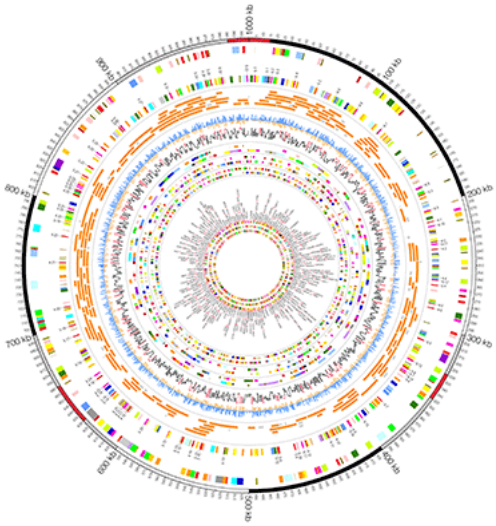


Figure 3.7: Circos visualizes multivariate data mapping radially the time [Krzywinski et al., 2009]

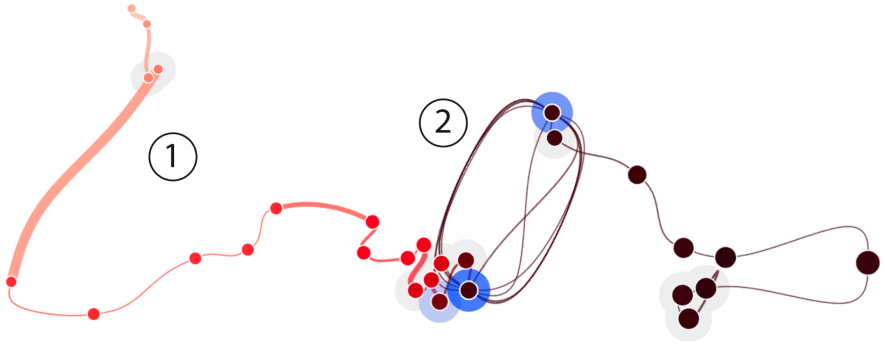


Figure 3.8: Time Curve folds a line graph to position closely similar points [Bach et al., 2015]

Linear Time Representation

Other visualization techniques for time series rely on a more classical time disposition by mapping the time dimension on the horizontal axis. In most cases, the events are disposed from left to right, the first chronological event being of the left side. It corresponds to the most popular representation of the time dimension. Most of visualization techniques with linear time disposition are variations of the line charts and aim to correct their defaults by integrating interactions and enriching their visual representations.

For example, Time Curve [Bach et al., 2015] is a technique based on line graph to visualize patterns of evolution by folding a line graph to bring closer similar points (Figure 3.8). To position the time points, a distance matrix is computed and serves

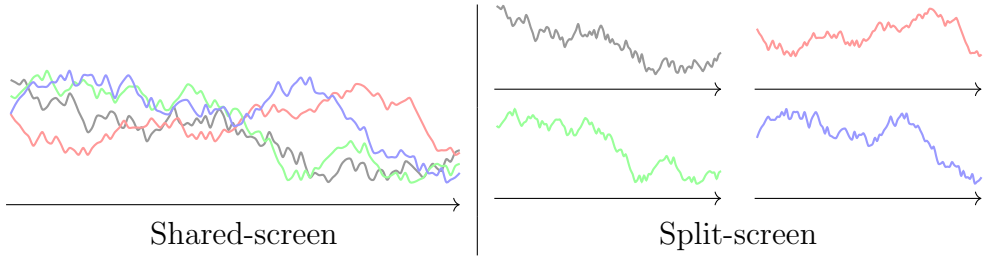


Figure 3.9: With *shared-screen* techniques, the graphs share the space and with *split-screen* techniques, the space is equally divided between the graphs.

as input to the MDS algorithm that computes the final position in 2D. The authors measure the distance between time points according to three criteria:

1. the *rank distance* that indicates how far are the points in time.
2. the *curvilinear distance* that also gives information about the temporal distance between points. However, the authors indicate it to be less precise as it depends on the positioning algorithm.
3. the *spatial distance* that encodes the similarity between two points: the closer two time points are, the more similar they are.

By locating closer similar points, Time Curve breaks the linear representation of the time dimension, which makes much more difficult reading the temporal aspect of the data. We refer to Aigner et al. [Aigner et al., 2011] for a complete review of the existing techniques for time series visualization.

On a more fundamental perspective, graphical perception of line charts greatly depends on their aspect ratio. Following this observation, Cleveland recommends an average slope of 45 degrees for the line segments (*banking to 45 degrees*), thus constraining the aspect ratio of the graph [Cleveland, 1993]. Heer et al. have proposed *multi-scale banking to 45 degree*, an automatic method to produce graphs that respect Cleveland's approach [Heer and Agrawala, 2006]. When working with multiple time series, this factor has an even greater impact due to the limited amount of space available.

3.1.2 Multiple Time Series Strategies

Many work have focused on the representation of multiple time series where the classic line charts exhibit several limitations due to limited screen space. Javed et al. [Javed et al., 2010] discuss the graphical perception of multiple time series visualizations derived from the line graph and identified two categories that differs on the screen space management: the *split-screen* and the *shared-screen* techniques. The *split-screen* techniques rely on the principle of small multiples introduced by

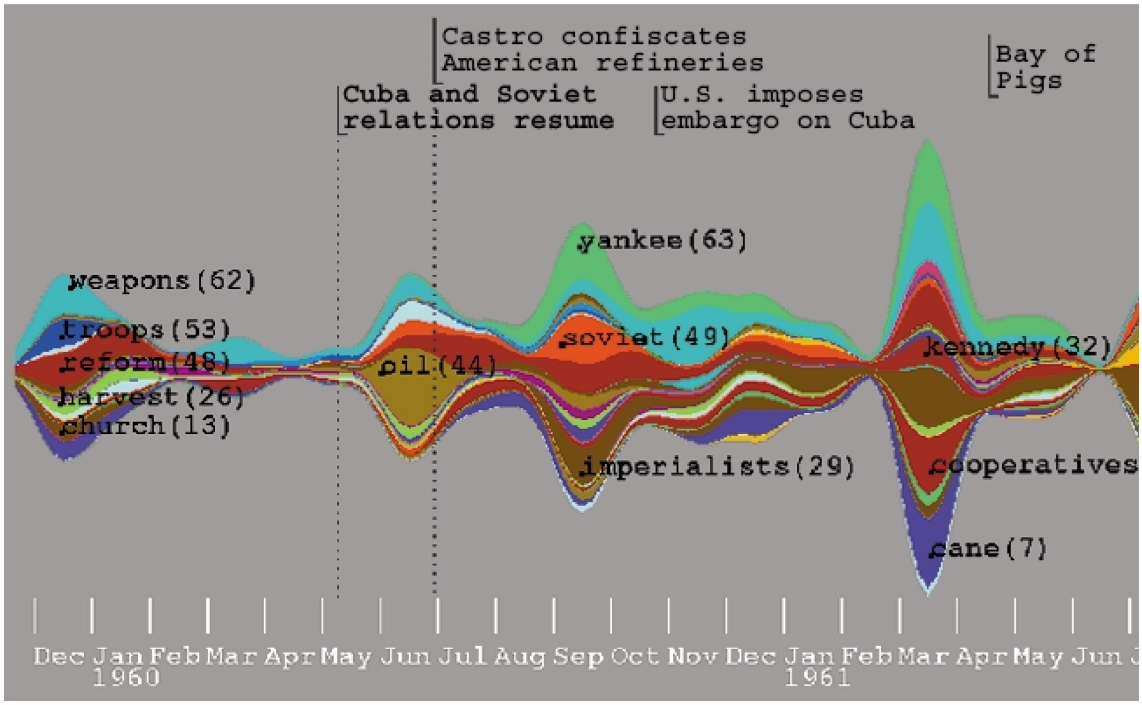
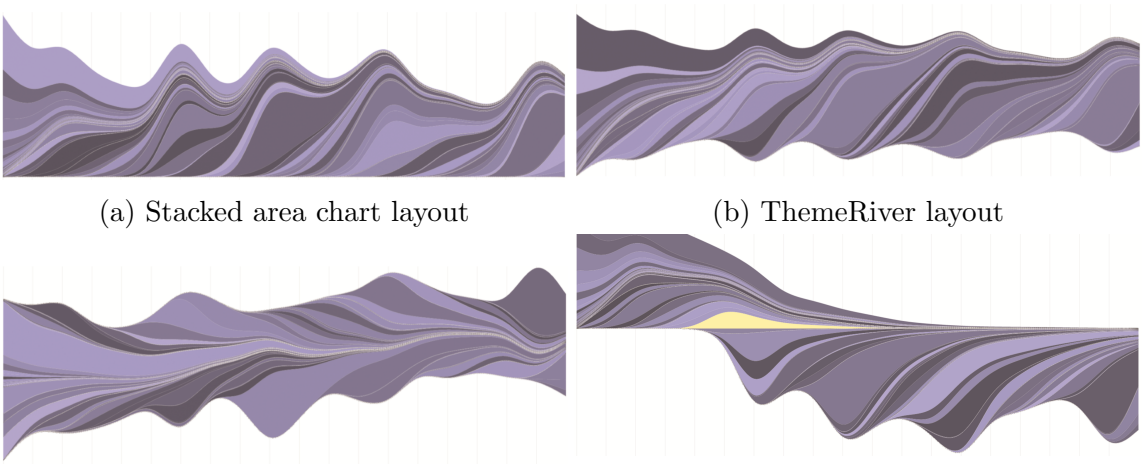


Figure 3.10: ThemeRiver. It visualizes topic density variations across time. [Havre et al., 2000]

Tufte [Tufte, 1986]. It consists in splitting the screen space S into N smaller areas of size S/N for each time series. The *shared-screen* techniques use a different approach: the time series are all represented in the same space and are differentiated using the color visual attribute. Javed et al. found that *split-screen* techniques are more suitable for reading global values while *shared-screen* techniques are better when working on local area of the graphs [Javed et al., 2010].

Shared-Screen Techniques

The *shared-screen* techniques display the time series on the same screen location and the time series are made discernible using an other attribute (e.g. the color). The multiple line charts are the most basic example consisting in rendering several line charts on the same location with different colors. Minard developed the first prototype of a stacked graph by stacking the time series that share the same unit and time domain. The layer area graph [Harris, 1999] is a more modern version where each time series is a layer in the visualization. ThemeRiver [Havre et al., 2000, 2002], based on this basic idea, propose a more sophisticated technique to visualize the different topics in a document collection (Figure 3.10). A topic is encoded in a layer whose height corresponds to the density variation of the topic across time while the overall shape provides the global variation of the document collection. A layer is also annotated to facilitate the reading. A 3D variant has been developed to encode



(a) Stacked area chart layout

(b) ThemeRiver layout

(c) StreamGraph layout

(d) Interactive ThemeRiver layout. Hovering a layer straighten its baseline

Figure 3.11: Different layout algorithms to stack the time series [Thudt et al., 2016]

two data attributes instead of one using the extra third dimension [Imrich et al., 2003]. A user evaluation demonstrates that this extension improves user reading.

Byron et al. [Byron and Wattenberg, 2008] improve the legibility and aesthetic of the Stacked Graphs by introducing several algorithms to compute their layout resulting in different rendering (Figure 3.11). Thank to their aesthetic appearance, the Stacked Graphs has gained in popularity for casual visualizations. For example, the New York Time exposed a Stream Graph on his website to show the evolution of the box office [Bloch et al., 2008] and Twitter visualized the hashtags density during an event using the origin ThemeRiver algorithm [Twitter, 2015]. Dork et al. have integrated a Stacked Graph rendered with the original ThemeRiver technique in a visualization tool and synchronized it with different views to explore the tweets about a topic [Dork et al., 2010]. Thudt et al. have evaluated the readability of the different layouts for the basic stacked charts, the ThemeRiver algorithm, the StreamGraph and an interactive ThemeRiver layout that straightens the baseline of layers in ThemeRiver [Thudt et al., 2016] (Figure 3.11d). Their results show that a minimal distortion improves the readability and the interactions helps to increase the correctness but slows down the users. These results also confirm a previous work that highlighted the legibility issues due to the stacking [Heer et al., 2009].

Instead of stacking the time series, Braided Graphs represents each time series by a silhouette graph (i.e a filled line graph) [Javed et al., 2010]. It slices the area graphs according to their intersection points and sort the depth of the bins, the highest value being the furthest to guarantee all the series remain visible. Figure 3.12 shows the construction process. The principal drawback of the Braided Graphs is their lack of scalability as the number of time series to visualize increases due to a high clutterness. User evaluation also shows that it never performs better than a multiple line graph.

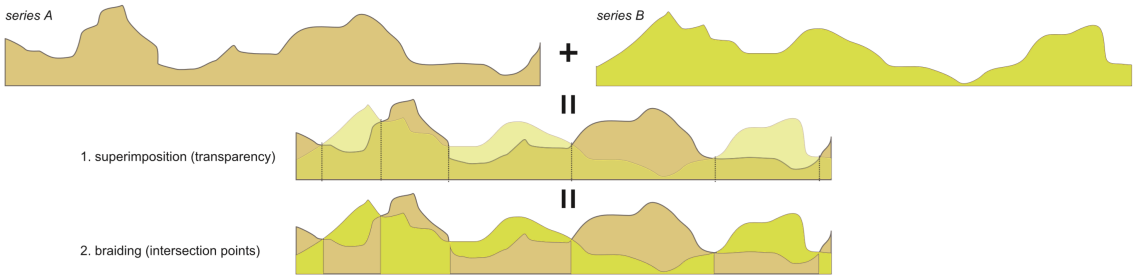
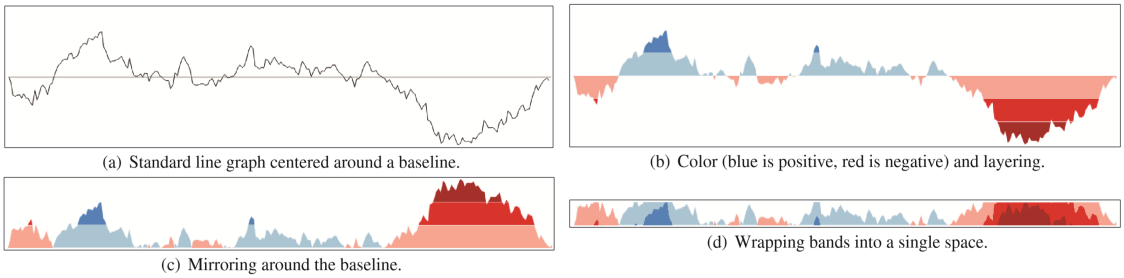


Figure 3.12: Construction method of a Braided Graph [Javed et al., 2010]



(a) Standard line graph centered around a baseline.

(b) Color (blue is positive, red is negative) and layering.

(c) Mirroring around the baseline.

(d) Wrapping bands into a single space.

Figure 3.13: Horizon Graph. The original line graph has been sliced into four bands below and above the baseline. The bands have been wrapped to reduce the vertical space. [Heer et al., 2009]

Split-Screen Techniques

Split-screen techniques result in applying the concept of small multiples developed by Tufte [Tufte, 1986] who introduced the sparklines, a small multiple of line graphs. Screen space is split into small areas, one for each time series.

The reduced line charts are a direct application of the small multiples: each time series is represented by a separated line chart that have a limited amount of screen space. The screen space available for each time series becomes very limited and many work have investigated the legibility issues of line graphs in such configuration. Horizon Graphs have been proposed to increase the user reading performance when a large number of time series are displayed on the screen [Reijner, 2008; Few, 2008; Heer et al., 2009]. It virtually increases the vertical resolution of the graph by slicing a classic line graph into bands and wrapping them (Figure 3.13) with increasing color saturation. Heer et al. [Heer et al., 2009] studied the reading performance and gave recommendations on the visual settings of the Horizon Graphs: 2 bands for a height of 6 or 12 pixels gives the best results. Javed et al. [Javed et al., 2010] compared the Horizon Graph with other techniques using the recommended parameters using up to 8 time series. Perin et al. [Perin et al., 2013] introduced Interactive Horizon Graphs by adding the pan and zoom interactions to control respectively the baseline and the number of bands and evaluated the user reading performance [Perin et al., 2013] showing their modifications decrease time completion while increase correctness. The results also show that Interactive Horizon Graph scale up to 32 time series.

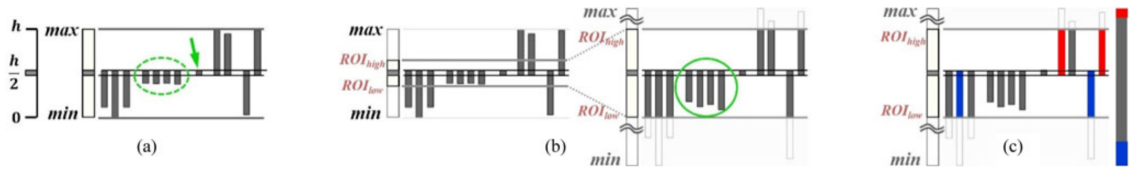


Figure 3.14: Ripple Graphs, a multi-scale time series visualization technique [Cho et al., 2014]

.

Execution traces can contain a much higher number of time series that can roughly be equal to the vertical resolution of a screen. Under these conditions, the height of a line chart would be close to one pixel. Moreover, Cho et al. [Cho et al., 2014] has shown that Horizon Graphs are not suitable for frequential analysis of time series, making them irrelevant to visualize traces of streaming applications.

Instead, they introduced Ripple Graphs [Cho et al., 2014], a multi-scale visualization where uncertainty appears. While there is no uncertainty in execution traces since at any moment, we know which process is scheduled, this aspect is not interesting for traces. Similarly to the (Interactive) Horizon Graphs, Ripple Graphs enable the user to zoom on the values by defining a Region Of Interest that eliminate the values outside of the of ROI (Figure 3.14). This feature makes the Ripple Graphs able to show graphs as low as one pixel, an interesting property for execution traces.

Visualizing execution traces is challenging: they contain a large amount of events and actors for which a time series is associated. An efficient visualization of traces should handle both dimensions. We review in the following section related work that explore large time series and that handle a large collection of time series.

3.1.3 Large Time Series Exploration

As discussed in the previous chapter, execution traces contain a large number of events and will keep growing in the future. Providing efficient visualizations to explore and analyze large traces is critical for a more efficient debugging of traces. We distinguish two approaches to visualize large time series: the multi-foci techniques and the overview+details techniques.

Multi-foci Techniques

Kincaid proposed SignalLens [Kincaid, 2010] to better study an electronic signal (Figure 3.15). It improves the navigation of an electronic signal by integrating a lens to zoom in a particular time window of the time series while keeping the remaining signal undistorted. They have implemented several lens functions (linear, quadratic, cubic, hyperbolic, spherical and Gaussian) that handles the distortion. The goal here is to address the readability problems to visualize electronic signals,

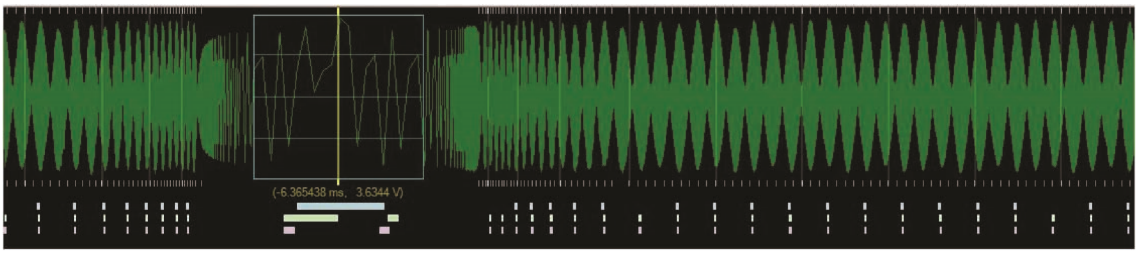


Figure 3.15: SignalLens proposes a focus+context technique for time series [Kincaid, 2010]

containing a large number of time points, on a small screen. Coupled to the line graph, a *measurement track* is provided to find more easily interesting features in a large signal. The main limitation of SignalLens is that it does not support a continuous interaction for setting the magnification factor. This is problematic when working with traces from streaming multimedia applications since periodic patterns can appear at different scales, thus at different levels of zoom.

ChronoLenses focuses on the interactive analysis of time series [Zhao et al., 2011b] (Figure 3.16). Its novelty is to provide to the users a highly interactive analysis pipeline based on lenses to explore the data, enabling the discovery of trends and outliers. To create a lens, the user typically selects an interval on the time dimension. From here, data points are transformed on-the-fly according to the type of the lens and its parameters can be adjusted afterwards. The user can also create analytic pipeline by combining several lenses together using different operators.

KronoMiner is an other flexible multi-foci visualization [Zhao et al., 2011a]. It relies on a radial layout and proposes a hierarchical organization of the ROI. At the center contains an overview of the dataset. To focus on an interval, the user can brush the ROI and a hierarchical segment is created. To edit the hierarchy, KronoMiner provides many interactions such as handles to adjust a segment.

BinX [Berry and Munzner, 2004] is a different technique that also provides an abstraction of a time series defined by the user. To achieve this, the tool proposes a single line graph visualization and allows the user to set the number of bins to use for the aggregation.

Overview+Details Techniques

The concept of overview and details has initially been introduced for image browsing [Plaisant et al., 1995] as an alternative to the scroll bar. It consists an overview of the data giving the context to the user and a view focused on a specific subset of the data.

Stack Zooming [Javed and Elmqvist, 2010] takes a different approach to provide focus+context technique for time series. Instead of relying on a single view and using distortion, a new line graph is added to the view showing a zoomed version of a selected time window, called a *strip* (Figure 3.17). The newly create graph

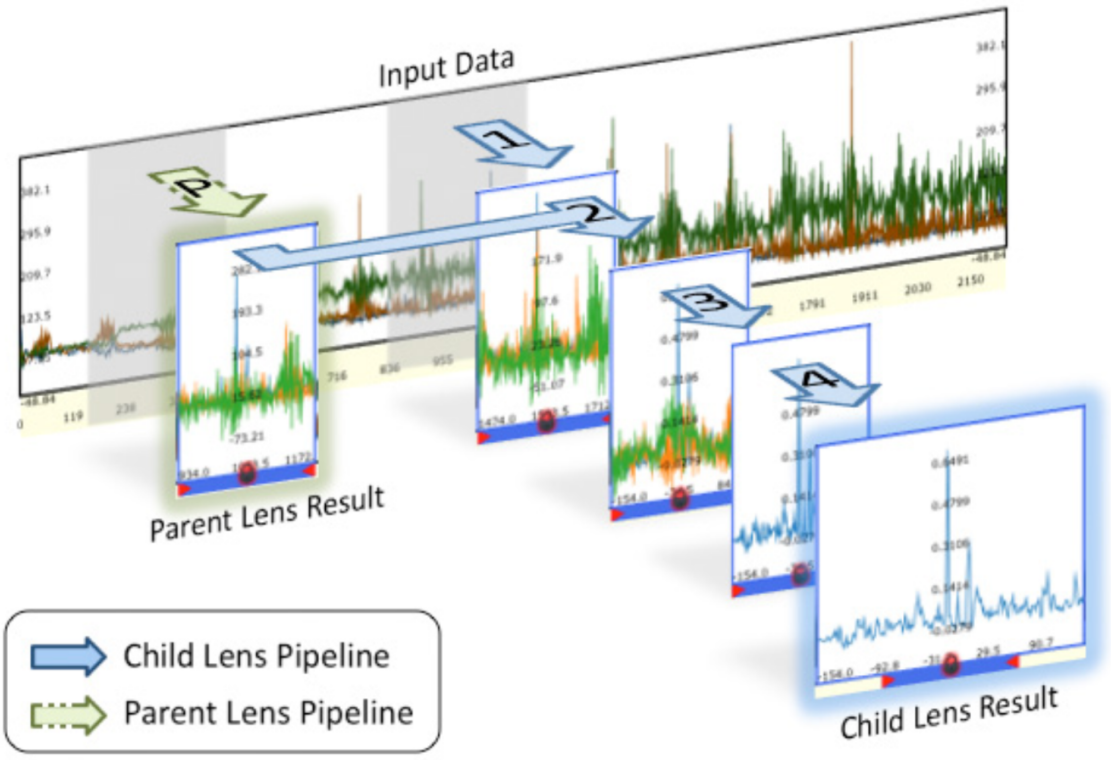


Figure 3.16: ChronoLenses is an interactive analytic tool based on lenses [Zhao et al., 2011b]

is then stacked below the original graph. From here, the user can zoom using the same mechanisms by creating an other strip on the newly added graph, and so on and so forth. Several strips can be created and the corresponding zoomed line graphs are juxtaposed on the same line. In a following work, they have conducted a user evaluation and have shown that stack zooming provides increased performances compared to the overview and details techniques [Javed and Elmqvist, 2013].

TimeNotes [Walker et al., 2016] is a more recent technique that mixes the analysis features of ChronoLenses with a hierarchical zooming similar Stack Zooming. It also provides new features such as overlay to compare easily different intervals of the time series, bookmarks and a dynamic layout (Figure 3.18).

All of these techniques rely on line charts that do not provide a clear visualization when there is a large number of time series to represent. It can be argued that a different type of visualization can be used to represent the time series. While some of the other techniques can certainly improve the scalability, it still would not be efficient for traces with a large number of actors (recall that at each actor, we can associate a time series). Indeed, the screen space is already used to show the zoomed areas and cannot be used to split the time series on several visualizations. Therefore, the only solution to visualize multiple time series with these approaches is to share

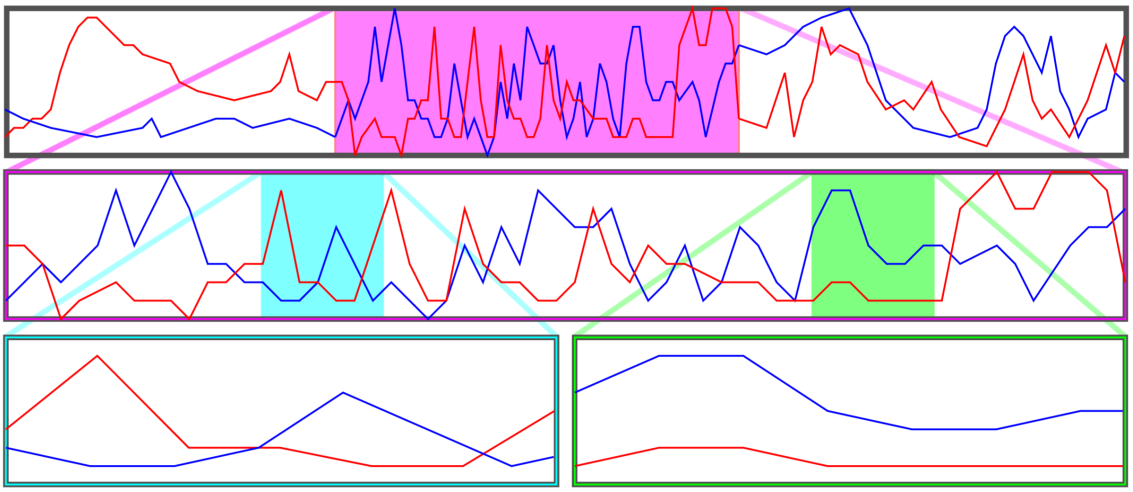


Figure 3.17: Stack Zooming [Javed and Elmqvist, 2010]

the screen space between all the time series, an inefficient setting to study local area of a graph (discussed below) as it is the case for execution traces.

Instead of relying on line charts, Hao et al. took a different approach by introducing a space filling technique based on the notion of Degree of Interest of the data [Hao et al., 2007]. It uses a matrix view where the color of the cells encode the value. The matrix has different resolutions, assigning more space for the newer time points. While the space filling technique is interesting, this multi-resolution approach is not suitable to debug execution traces: a temporal bug can appear at any moment during the execution and the same level of details is needed during the analysis. It would also hides the periodicity of the data and makes harder the detection of temporal bugs.

Summary

The main problem of these techniques when working with execution traces is their lack of scalability with respect to the number of time series to analyze. They rely on a line graph and to visualize several time series, the view will become cluttered very quickly, belonging to the *shared-screen* category. We have seen above that the *shared-screen* techniques perform well for local comparison but are not suitable for global analysis of a time series, which is mandatory in the case of traces. For instance, to understand the synchronization between different actors, we have to understand their behavior across the whole execution. When cluttered, visualizing whether an actor is scheduled or not is very tedious.

Before being able to analyze deeper the behavior of a process, the software developers have first to get a global overview of the trace, therefore of many time series. An in-depth analysis of an actor is possible only when isolated first although it is rarely a necessity to analyze with such level of details the time series of an actor. In this case, the aforementioned techniques would be usable in our context.

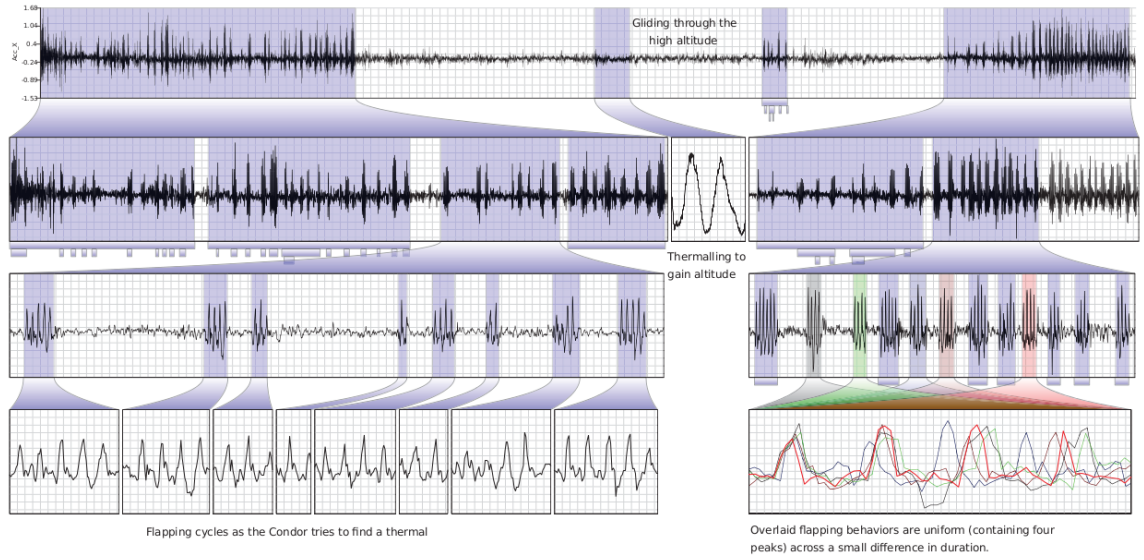


Figure 3.18: TimeNotes [Walker et al., 2016]

3.1.4 Exploration of Large Collections of Time Series

An other aspect to consider is the browsing of collection of time series. We described the existing works for the exploration and analysis of time series containing a huge amount of events. Due to their approach, they lack scalability when working with many time series. We present here the previous research done to explore many time series simultaneously.

Multi-scale visualization techniques have been largely investigated to visualize many time series. Depending on the screen-space available, the level of aggregation of the data varies, showing an abstracted or a detailed view. To spot interesting patterns, the user explores the different levels of abstraction.

Based on Stack Zooming, Javed et al. have proposed TraXplorer [Javed and Elmqvist, 2010] to explore a collection of time series. As explained before, this approach is not suitable when working with many time series. Hao et al. have introduced a tool based on a treemap [Hao et al., 2005]. The time series are visualized as bar chart and their size depends on the importance they have in the dataset. Using a treemap structure to organize the different time series makes the comparison between the different graphs tedious due the different size and the misalignment of their baseline.

Line Graph Explorer (Figure 3.19) relies on a focus+context technique to visualize a large number of time series [Kincaid and Lam, 2006]. The novelty of this work is the encoding of the data value in the color instead of using the y-dimension. By doing so, it enables the vertical compression of the graphs down to one pixel while keeping a good readability for an overview. The authors have also implemented a clustering method to order the graphs and discover similarities between the time

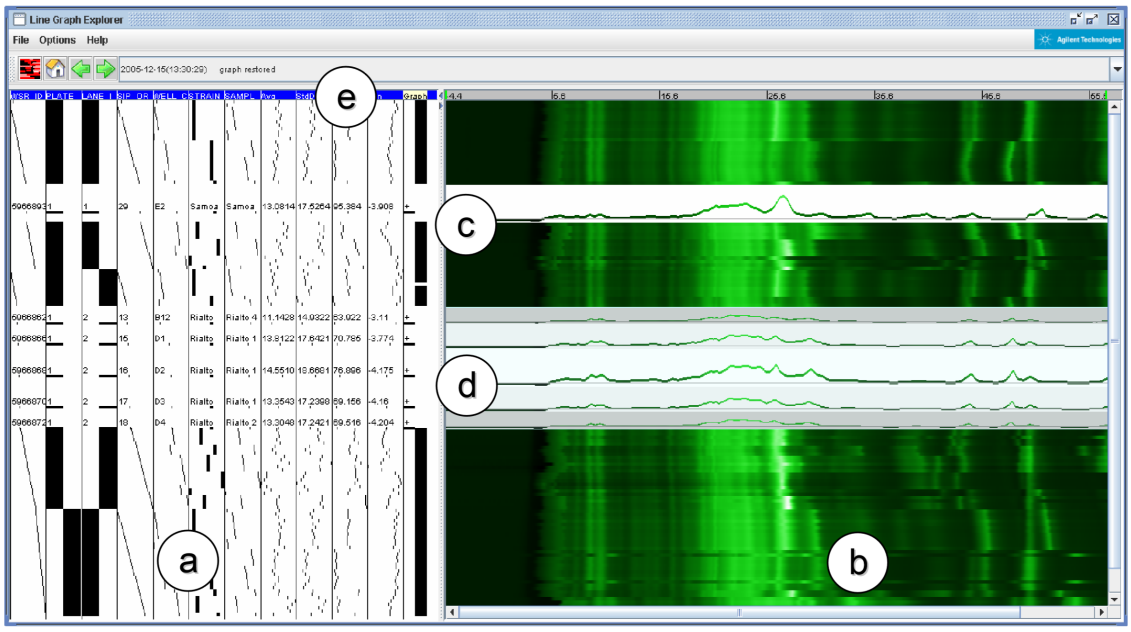


Figure 3.19: Line Graph Explorer [Kincaid and Lam, 2006]

series.

CloudLines also stacks vertically the time series and integrates a lens distortion to better analyze a single time series [Krstajic et al., 2011]. It also introduces a new compact visualization for the small multiples based on a kernel density estimation to better spot the visual clusters. The downside of this technique is a decreased capacity of handling a large number of time series compared to Line Graph Explorer since the vertical axis is used to encode the values.

LiveRAC rather uses a matrix-based layout where a graph is displayed in each cell and allows side-by-side comparisons of small multiples [McLachLan et al., 2008]. The graphs supports semantic zooming to provide meaningful representation according to the dedicated space they have in their cell.

Stroscope takes advantage of the Ripple Graphs presented earlier to visualize the time series as small multiples [Cho et al., 2014], equally dividing the vertical space between the graphs (Figure 3.20). It provides horizontal zooming interaction as well as selecting a range of values on the graphs thank to the Ripple Graphs. In a similar way than Line Graph Explorer, Stroscope provides a clustering method to classify the time series and create groups. To further facilitate the analysis of the dataset, Stroscope provides side-by-side comparison by juxtaposing either vertically or horizontally two graphs and more powerful analytic features based on statistics and clustering.

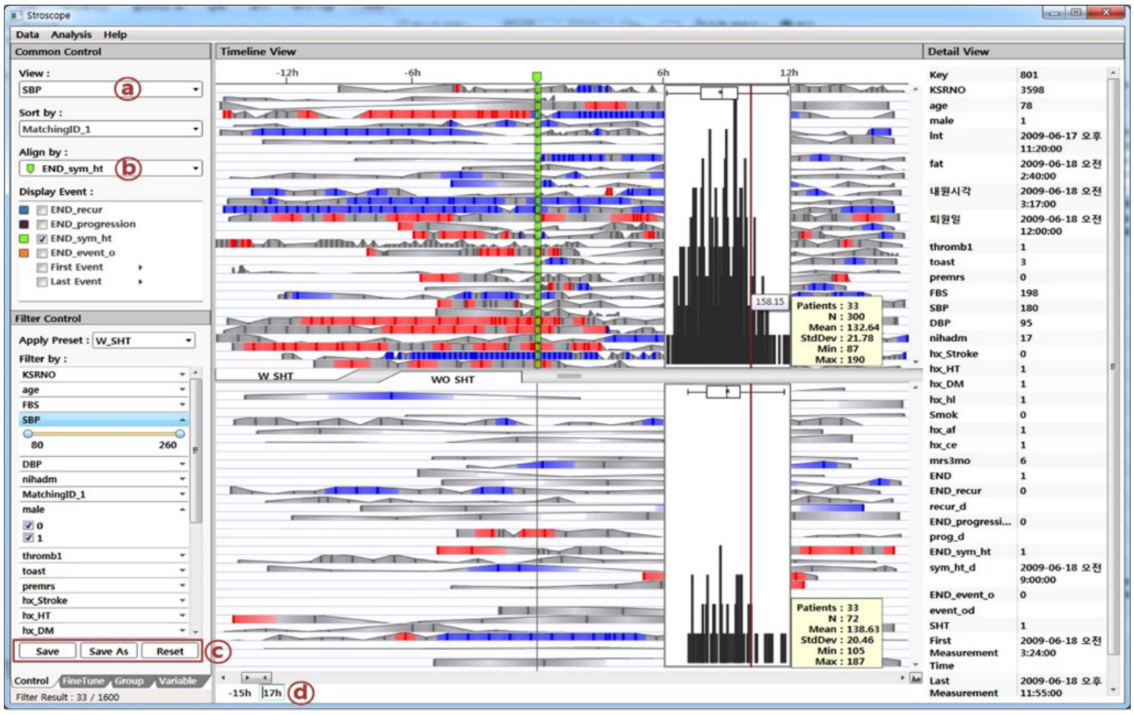


Figure 3.20: Stroscope [Cho et al., 2014]

Summary

All these works belong to the *shared-screen* techniques, suitable for global analysis. They propose scalable approaches to tackle a large amount of time series and are efficient to gain global insights on the dataset. Stroscope pushes further the analytic features by providing a filtering mechanism to select a range of values [Cho et al., 2014]. However, most of them lack interactions to explore the time dimension, required in the case of time series containing a large number of events.

The strategies to manage the vertical space has been largely studied but the horizontal space is still underexploited. Indeed, none of these works provide an efficient aggregation strategy to bin and aggregate the time series. It can result of visual artifacts due to over-plotting that negatively impact the precision of the visualization. When working with potentially periodic time series, as it is the case with execution traces, it becomes a critical problem.

Fuchs et al. have conducted a user study to evaluate the performances of the different types of glyphs for small multiples of time series [Fuchs et al., 2013]. Their results shows the line glyphs perform better to find local extrema and spot global trends in the data. This is an other aspect to consider when designing analysis tool for execution traces and indicates that line glyphs will give the best results.

3.1.5 Visual Mining of Time Series

Visual mining of time-oriented data has been vastly investigated and several surveys are available such as [Aigner et al., 2007]. One of the most common task is to find the most frequent patterns in the time series. Many techniques have been proposed to address this problem. Hao et al. introduced a visualization to frequently occurring patterns in multivariate time series [Hao et al., 2012] using *k-means* clustering. Based on the discovered patterns, the time series is condensed to offer a summarized view. To analyze execution traces, we basically need to find the repetitive patterns that correspond to the correct behavior of the application. For this, the approach proposed by Hao et al. is very efficient. However, in our case, we also need to spot the patterns that are responsible of bugs. It can be either a derivative of a regular pattern or patterns that tend to be short (in terms of number of events) and happens with a low frequency. Summarizing a trace based on the most frequent patterns is not suitable here.

TimeClassifier classifies behavioral time series semi-automatically using labeled data and user-defined template and provides interaction to browse large time series [Walker et al., 2015]. TimeClassifier has been developed to study animals' movements and is designed for expert users. When working with execution traces, the main limitation of this work is the absence of labels on the time series.

Other works took a different approach where the user has to provide an example of the pattern to search in the time series. We distinguish two approaches.

First, QuerySketch enables the user draw freely the pattern on a blank canvas to search on the time series [Wattenberg, 2001]. This approach can be difficult to use if the pattern is complex to draw or the user have no idea of what to search.

Second, techniques have been proposed where instead of drawing freely a pattern to query, the user selects it on the visualization and the tools finds its occurrences in the dataset. TimeSearcher [Buono et al., 2005] enables the user to select a rectangular area on a multi-line graph. It defines a Region Of Interest, called a TimeBox widget [Hochheiser and Shneiderman, 2004], that specifies a time interval on the horizontal axis and a value interval on the vertical axis. All the time series for which every value in the time interval belongs to the value interval are selected. The visual query can be adjusted by dragging and resizing the widget using handles. Coupled to the TimeBox widget, TimeSearcher also integrates a *SearchBox* that enables to query-by-example with the selection of a motif on a graph [Buono and Simeone, 2008]. Then, a similarity is computed using an Euclidean distance and a user-defined threshold. Finally, a *angular query* widget allows to filter the graphs based on their slope. QueryLines [Ryall et al., 2005] proposes a similar approach than TimeSearch that performing visual queries on the graph. However, instead of specifying constraints through a rectangular widget, in QueryLines, the user draws lines. The query can also be adjusted by modifying the lines but the degree of similarity cannot be user-defined as in TimeSearcher, making the queries less flexible.

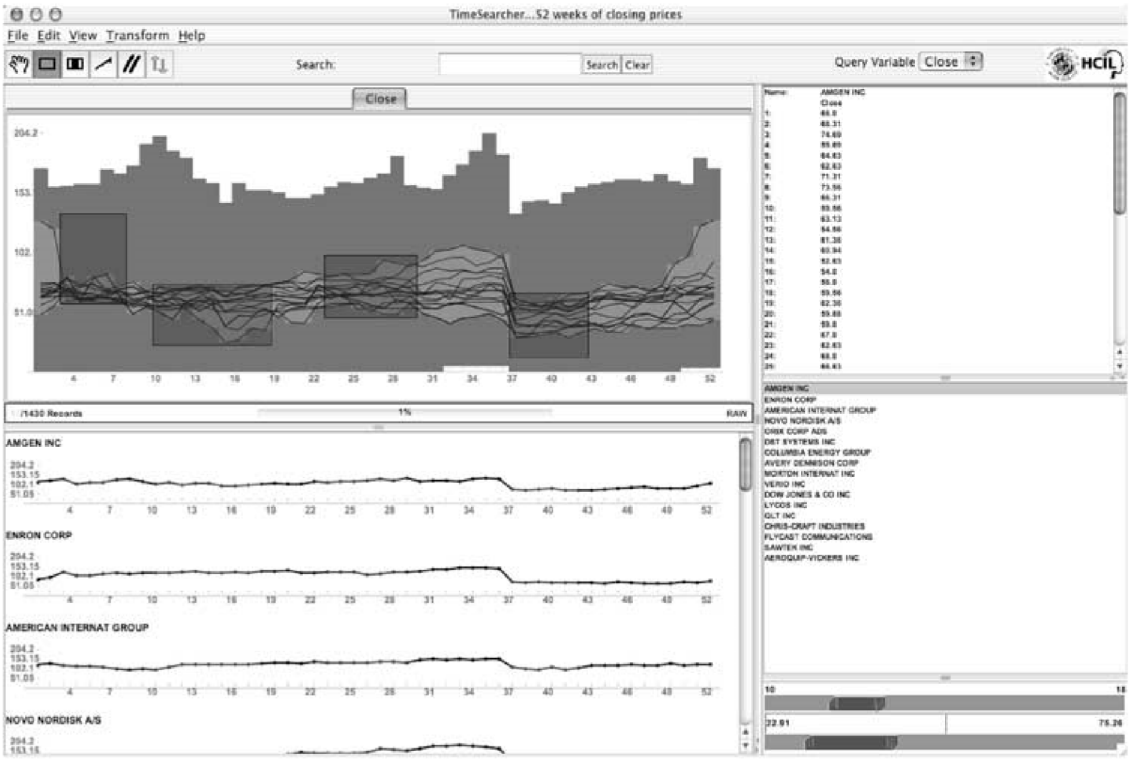


Figure 3.21: TimeSearcher [Buono et al., 2005]

Holz and Feiner have proposed a more flexible technique to perform visual queries on graphs [Holz and Feiner, 2009]. In this work, the user can create a query by sketching a pattern on the line graphs. Then, their tool finds similar occurrences of the queried pattern in the dataset. Instead of defining a threadshold used to compute the similarities as in TimeSearcher, here the tolerance depends on the distance between the sketch and the graphs: the further to the sketch, the more relaxed the query (Figure 3.22).

All of these works propose very powerful tools to find time series with similar behavior and this approach can help to find groups of actors working together (such as a couple of an interrupt and its handler). However, execution traces are large and developers may have a little to no idea regarding where to begin their analysis. Thus, specifying from scratch a region of interest may not be an easy task. This also apply to techniques where the end user specifies precisely the pattern to find such as with PatternFinder [Fails et al., 2006]. Here, the patterns are specified through a query panel and while it does not rely on the visual shape of the pattern, it still can be challenging to begin to query the dataset if the user does not know what to search. Stoffel et al. [Stoffel et al., 2013] proposed an other visual analytic tool whose goal is to detect anomalies in time series, applied to networks. The view shows vertically aligned line charts. The time series are compared against a reference model and the differences are highlighted in on the line charts. Similarly to the other tech-

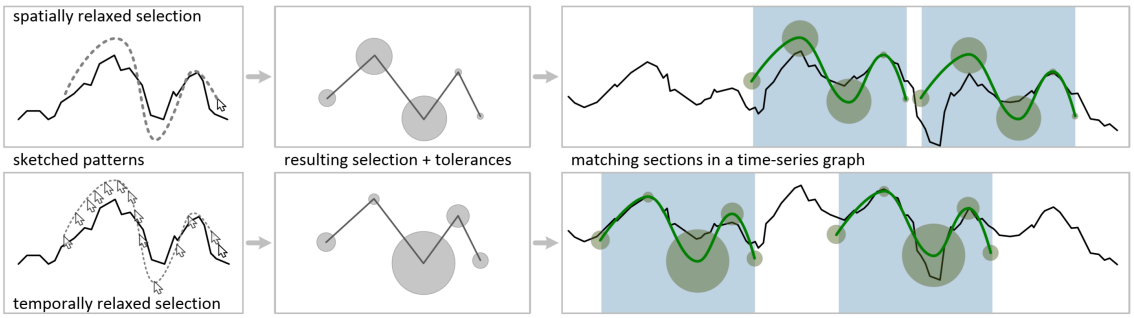


Figure 3.22: Relaxed Selection Query on Time Series [Holz and Feiner, 2009]

niques, the main difficulty for the developers remains in defining the reference model.

VizTree takes a different approach [Lin et al., 2004, 2005]. It first transforms a time series into a symbol string and then encodes it into a tree whose branches are the patterns. The frequency of a pattern is represented by the thickness of the branch corresponding to it. While with VizTree it is possible to detect the frequent patterns in a time series, it still requires an expert user to be able to exploit correctly this tool.

Summary

We saw that different approaches exist to visually mine time-oriented data. One relies on algorithm to automatically compute time series clusters [Hao et al., 2012] and find frequent patterns [Lin et al., 2004, 2005]. These techniques often require an expert user to understand how to exploit the results or to tune the algorithm before computation as it is the case with the algorithm used in VizTree.

Other approaches take as input a pattern given by the user. This pattern can be very precisely specified [Fails et al., 2006] or loosely defined through sketches drawn either directly on the graphs [Buono et al., 2005; Hochheiser and Shneiderman, 2004; Buono and Simeone, 2008; Ryall et al., 2005], either on a dedicated area [Holz and Feiner, 2009]. We explained that in the context of debugging a streaming application through execution traces, the software developers hardly have an idea of the patterns to search. Thus, based on the proposed techniques, starting the analysis with these tools can be very challenging as the query remains largely unknown.

Instead, we argue that a better approach is to take advantage of pattern mining techniques that does not need any input query and then proposing an efficient pattern visualization tool. After having presented the existing works on the visualization of execution traces, we review the pattern visualization techniques in Section 3.3 and we highlight the challenges that raises this method.

3.2 Visualization of Execution Traces

Isaacs et al. published a complete survey of performance visualization techniques [Isaacs et al., 2014b] and maintain up-to-date an online website [Isaacs et al., 2016]. In their work, they classify the goals that motivate visualizing performance data in general:

1. **Global comprehension.** When a problem occurs, several software components may be implied and a developer may not understand the whole system but a sub-part on which he is in charge. Under these conditions, tracing systems can help to understand the global behavior and how the different components communicate.

2. **Problem detection.** It aims to detect abnormal behavior. In the context of multimedia application, it can refer to bottleneck, periodicity perturbation, non-optimal scheduling and so on. It can be done with visualization techniques or automatic tools based on data mining algorithms.

3. **Diagnosis and attribution.** Isaac et al. describe this goal as the next task to perform after a bug has been found. Here, the developers go deeper in their analysis and try to find more specifically the reason.

To debug an application using traces, the most basic task is to be able to understand what happened during the execution. Thus, in this section, we present the previous works that allow to visualize the different events of an execution where the time is encoded as a dimension (often as the horizontal axis). Many other research have been undertaken on traces in general, particularly in the domain of High Performance Computing (HPC) with different tasks than debugging. Their goal may be to make apparent the software or the hardware architecture, the communications and synchronization between the different components, etc. In these works, the time dimension can be hidden or partially represented, not supporting the debugging of parallel applications. Moreover, in our context, when debugging an application, the developers are highly likely to know the architecture of the platform they are working on. Thus, we discard in the remaining of this chapter the research that aimed to visualize the architecture of the platform.

Representing the events of an execution traces quickly raise the problem of screen space: the number of events can be larger by several orders of magnitude than the number pixels available on the screen. Providing an overview of the execution helps the developers to understand the global behavior of the system and to target a specific time window. The details of the execution also brings important information for a fine-grained analysis. We first present the work focusing on showing the overview of the execution. We then describe techniques that visualize in details the traces.

3.2.1 Overview of a Trace

There exists many different tools that give a trace overview. To be meaningful, we claim that the overview of an execution trace has to provide data aggregation for

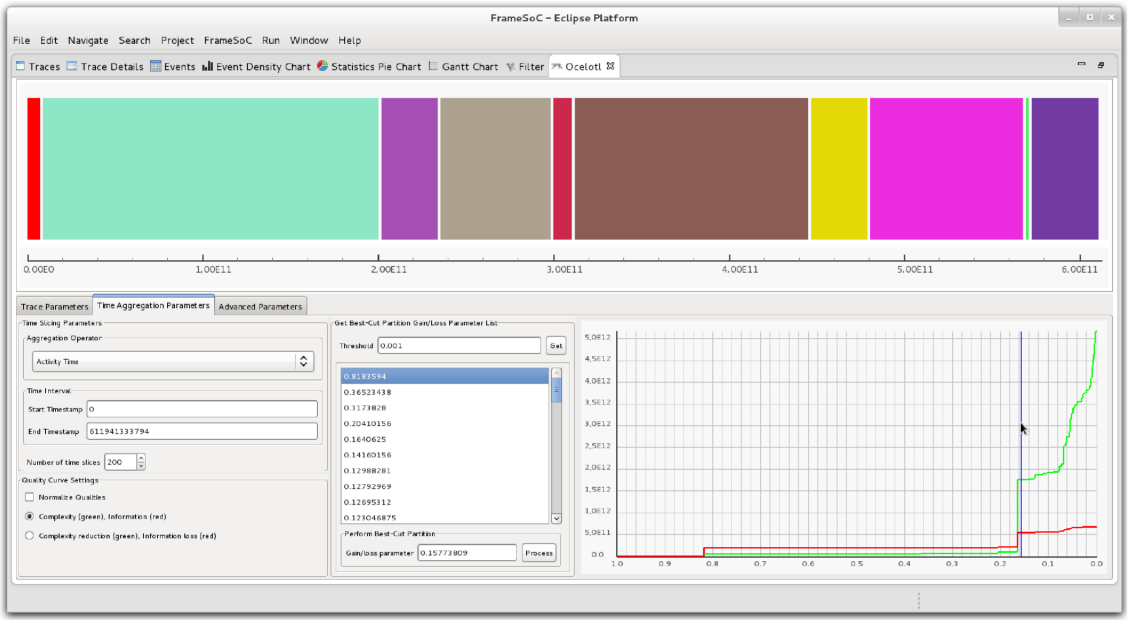


Figure 3.23: Ocelotl provides a trace overview using hierarchical and temporal aggregation [Pagano et al., 2013]

both time and event producers hierarchy (e.g. processes, interrupts, etc.), to show insights on the system load using well-known statistics and to provide user interactions to support fast data exploration and filtering. We describe below the different approaches and their drawbacks.

Ocelotl [Pagano et al., 2013; Dosimont et al., 2014] proposes a visualization that aggregates both the actors and the time dimension to obtain an overview of the execution (Figure 3.23). It comes with user interactions that allow to choose the aggregation level enabling the analyst to explore the macro-behaviors at different scales. It shows the trade-off between the level of aggregation of the loss of information enabling the user to interactively choose the aggregation level that fits best his need. However, it lacks interactions to navigate the trace and does not represent meaningful statistics for the developers.

Viva [Lamarche-Perrin et al., 2014] has a similar approach by aggregating data of both the actors and the time axis but uses a treemap to show both software and hardware hierarchies. The time dimension is visualized using animation. ExploreViz is an other treemap-based trace visualization for Java programs [Fittkau et al., 2013] (Figure 3.24). Here, the cells represents the software structure (packages, classes, etc.). The hierarchy can be interactively explored by opening and closing them. The size of the cells represents the number of active instances at a given time, chosen through a timeline located below the treemap.

While these approaches are suitable to spot load balancing issues, they are not

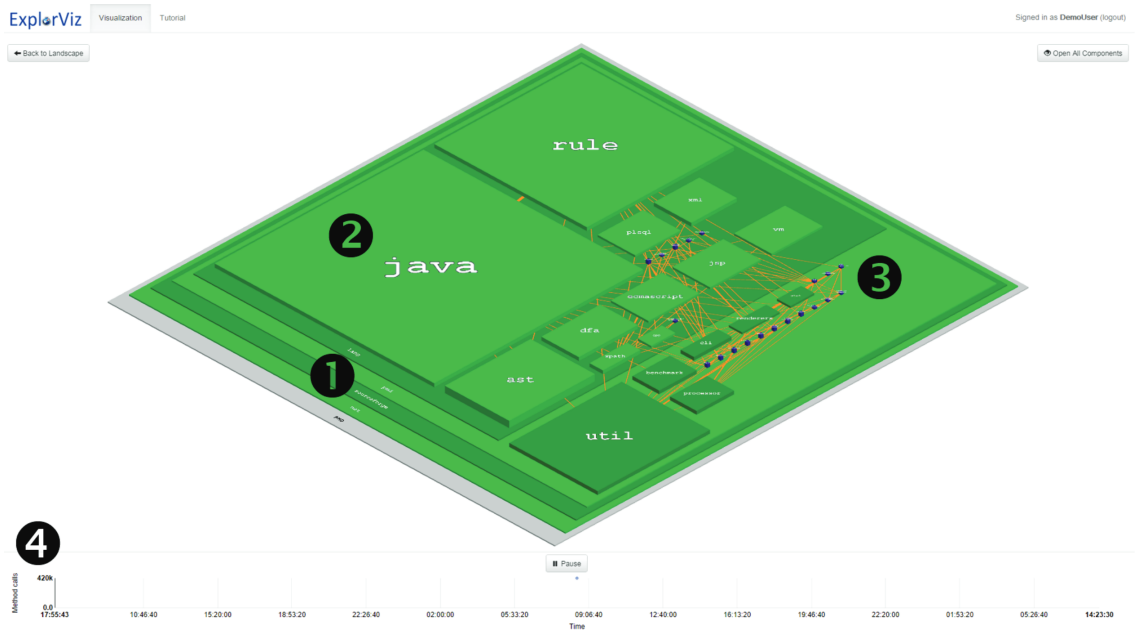


Figure 3.24: ExplorViz is a treemap-based to visualize Java programs execution [Fittkau et al., 2013]

appropriate in the context of multimedia applications where detecting synchronization is crucial: the treemaps only show a fixed time of the execution and the dynamic does not appear.

Other visualizations rely on statistics computed from the trace. KPTrace [Prada-Rojas et al., 2009], with the Outline view (Figure 3.25), and Eclipse Trace Compass [Compass, 2016] propose a bar chart where the whole trace has been aggregated using a statistic like the event density or the CPU load. This kind of view perfectly shows the overall behavior of the system across time and can be good to start the analysis with. However, the actors details are completely hidden, preventing the developer to observe the behavior of individual actor over time. This makes them irrelevant for a fine-grained analysis of the trace and reduces the relevance of the information they provide. These tools are coupled with other views to give the details of the statistic by actor. However the time space is also aggregated making the exploration of the temporal dimension impossible.

Summary

We described different visualization techniques that provide an overview of an execution trace. They rely on several strategies, whether using sophisticated aggregation algorithm, treemaps or domain-related statistics to provide high-level insights on the execution and/or the structure of the software. However, whether because of a lack of interactions or a too aggregated view, these tools do not provide enough

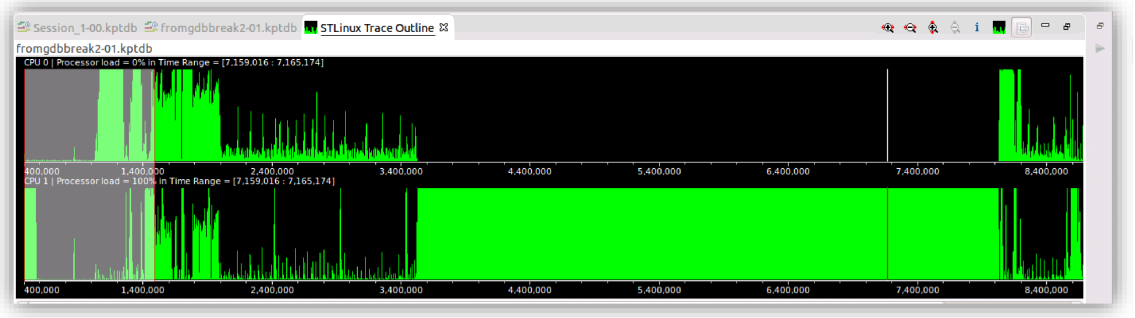


Figure 3.25: Outline View provided in KPTrace

information to efficiently start the analysis of the execution and finding the source of a temporal bug, potentially involving several actors.

The case of Ocelotl differs as Dosimont et al. have shown the capacity their approach to support the debugging of application traces. However, in some cases, it lacks interactions to apply their aggregation technique on different time intervals in the trace.

3.2.2 Detailed Visualization of a Trace

A large panel of visualization tools for traces that provide details is based on Gantt charts [Gantt, 1913]. When visualizing execution traces, the 2D time series visualization puts the time dimension on the horizontal axis, the actors on the vertical axis and represents the active time windows for each actor. It gives a detailed view of the connections between the actors.

One of the earliest work to use Gantt chart for representing traces in parallel systems is Paragraph [Heath and Etheridge, 1991] and many later work do so, from proprietary industrial solution such as KPTrace [Prada-Rojas et al., 2009] and Streamline [ARM, 2016a] to various open source projects like Eclipse Trace Compass [Compass, 2016] and FrameSoC [Dosimont et al., 2014]. However, due to the high visual clutters of Gantt charts, aliasing problems quickly arise as the amount of information to represent on the screen increases (Figure 3.26). Visual artifacts appear and without any aggregation technique, it quickly becomes impossible to visualize correctly a huge trace in its whole.

Pajé [Chassin de Kergommeaux, 2000] and Eclipse Trace Compass [Compass, 2016] have implemented simple aggregation mechanisms to address this problem. Aggregated temporal windows are encoded using different visual attributes such as the shape or the color. By doing so, the developer can be misled in the analysis since the algorithm only compute a visual aggregation instead of data aggregation, and can result in information loss. On the other side, in case of absence of aggregation, browsing large traces with Gantt charts is fastidious and behavioral patterns tend to be difficult to spot.

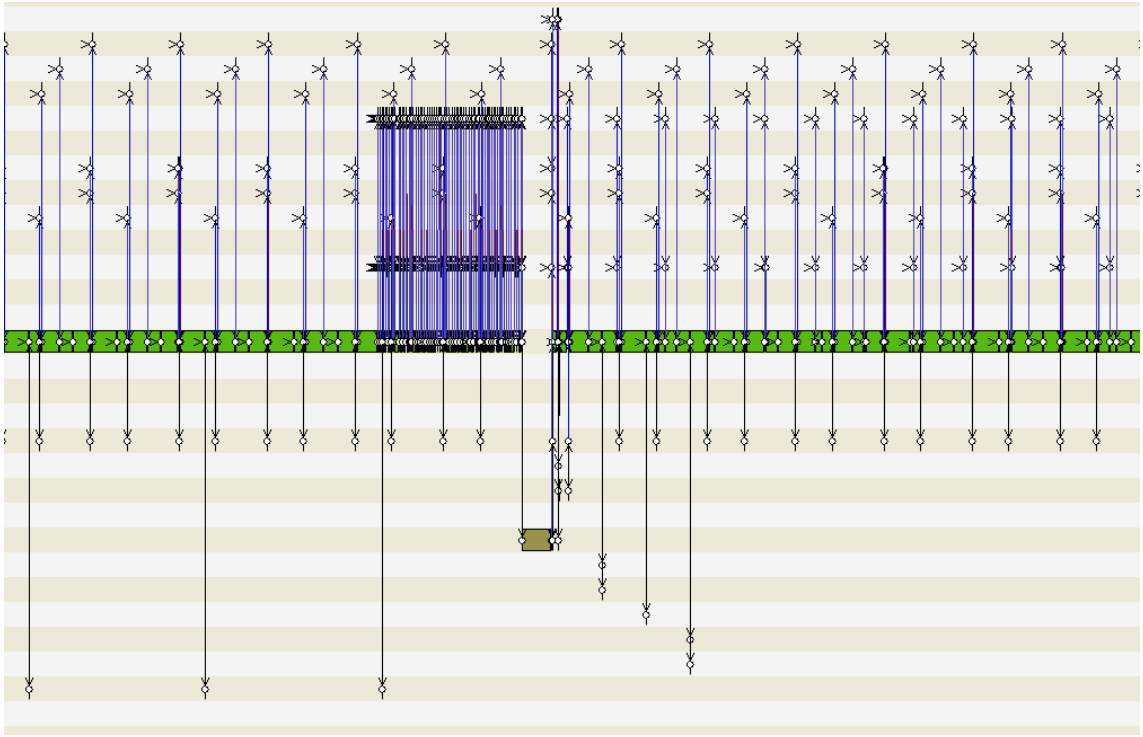


Figure 3.26: With no aggregation technique, visual artifacts quickly appear when working on huge traces with Gantt Chart. Here is an example of the KPTrace view.

Smart Traces [Osmari et al., 2014] integrates several Gantt charts with multiple views to show different hierarchical aggregations (threads, modules), minimizing the limitations of the other tools. On Figure 3.27, Smart Traces shows several Gantt charts for different modules. It enables the analysis of each module separately and minimizes the aliasing by filtering the data.

Similarly to Smart Traces, ViewFusion [Trümper et al., 2012] is a tool based on multiple views that synchronizes a timeline showing the overview of the trace, a Flame Graph [Gregg, 2016a] that represents the activity and a treemap [Schneiderman, 1992] depicting the structure of the software. Telea et al. [Trümper et al., 2013] have also proposed a visualization technique to compare two execution traces. The tool is composed of different views and shows the differences between two traces using *tubes* whose width encodes the duration of a match and the color encodes the start time difference, from red (past) to green (future) (Figure 3.28). Both ViewFusion and the latter tool provide different level of details from an overview to a detailed view but the aggregation gap between both views remains too large. Moreover, the aggregation technique implemented for the overview is not clearly described and can still mislead the users.

Flame Graphs [Gregg, 2016a,b] are an other popular visualization technique

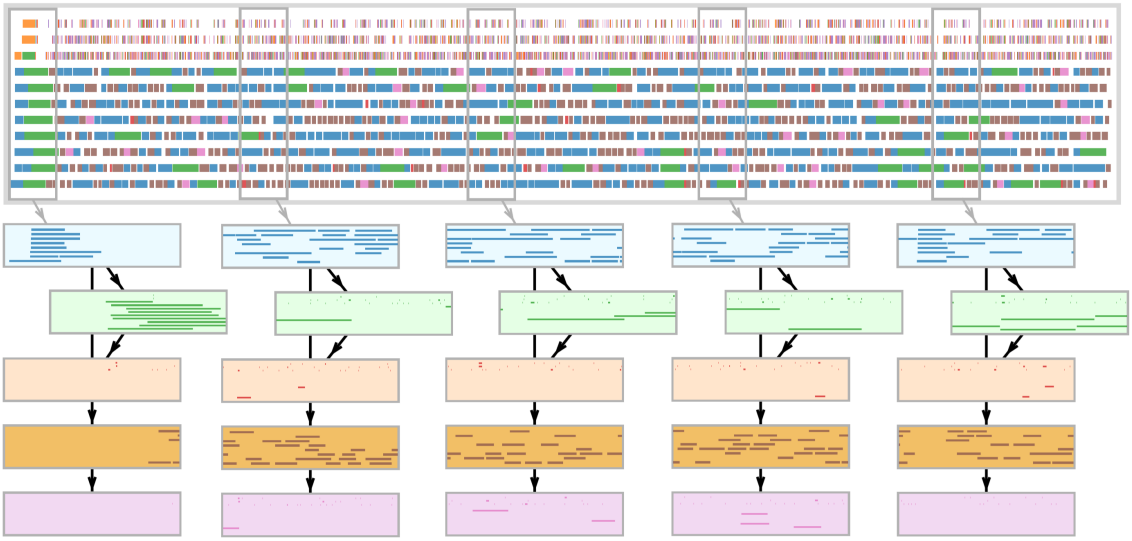


Figure 3.27: Smart Traces shows several Gantt charts simultaneously, one color corresponding to a module [Osmari et al., 2014]

based on icicle plots [Kruskal and Landwehr, 1983]. Flame Graphs depict the call stack of a program with the different call mapped vertically. Therefore, the deeper the call stack, the higher the Flame Graph (see Figure 3.29). Further techniques have been developed such as Differential Flame Graphs [Bezemer et al., 2015] inspired from the Unix `diff` command tool to better compare to traces.

A particularity of this technique is that the horizontal axis has no particular meaning. Instead, the different stacks are ordered alphabetically based on the name of the first function of the stack. This approach can be counterintuitive when debugging streaming applications and fragments the temporal patterns across the x-axis. The color used for the different boxes are also chosen randomly. The authors made this choice to easier the differentiation of the different boxes but can easily mislead the user seeing a darker box meaning a high frequency of calls for instance.

Trümper et al. have introduced a visualization tool specially designed to understand multithreaded applications [Trümper et al., 2010]. It took a overview and details approach by integrating a summarized view of the actors on which a time interval can be chosen through time span markers. The detailed view is based on an icicle plot and shows the call stack of the threads.

Cornelissen et al. introduced ExtraViz that provides a circular view showing the method calls and a timeline view [Cornelissen et al., 2007a,b]. The circular view shows the method calls that occurred during the execution. The components of the application are visualized on the circumference of the view and are linked together when a method call has occurred (see Figure 3.30). The links are grouped together to mitigate the visual clutter and are rendered using a gradient indicating the calling method. This view is synchronized with a time line where the time axis is vertical

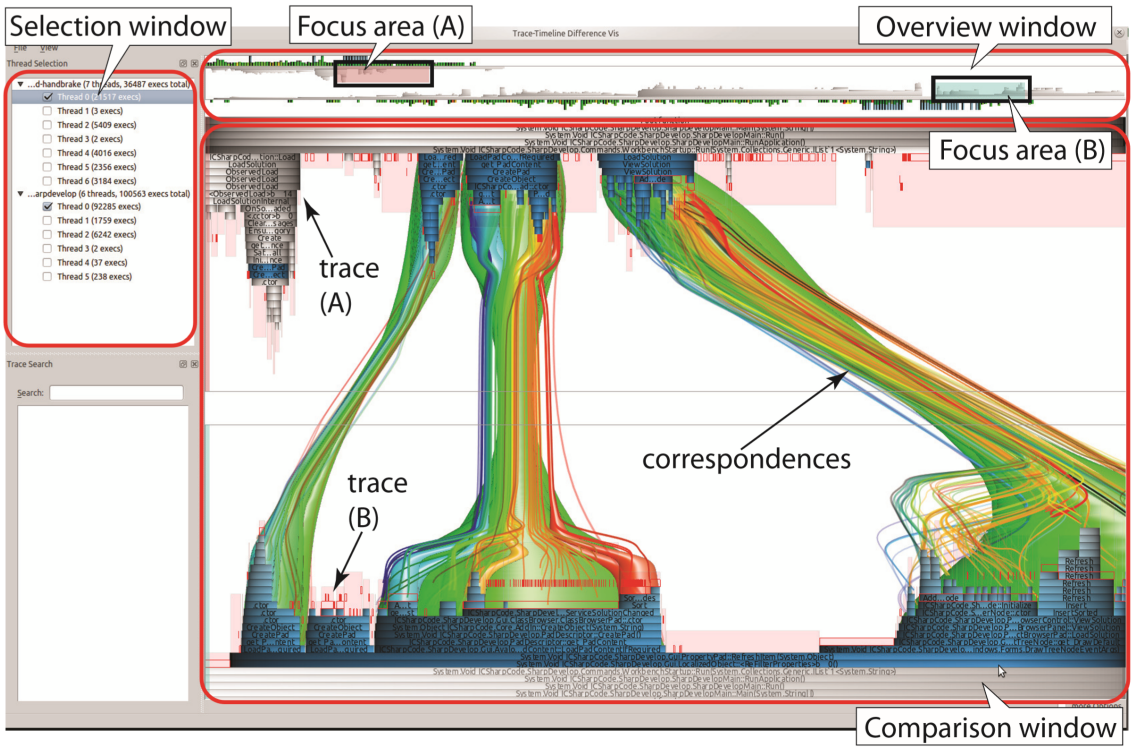


Figure 3.28: Multiple views configuration to visualize the differences between two traces [Trümper et al., 2013]

and the software components are placed horizontally. It enables the user to select on the circular view a time interval to visualize.

Ravel [Isaacs et al., 2014a] follows a different approach from all the works aforementioned by taking in consideration the *logical* time instead of the *physical* time. When working with physical time, the time point are mapped to position. It gives an accurate representation with respect to the events start and end time differences. Logical time reorganizes the events according to the Lamport clock C, a function that assigns a number to each event $event_1$ and $event_2$ so that $C(event_1) < C(event_2)$ if $event_1$ happened before $event_2$. Using the logical time loses the physical time representation but has the advantage to make more apparent the execution patterns (Figure 3.31). This approach is interesting for high performance computing debugging, however, in the context of performance analysis of embedded multimedia applications, losing the different between the start time and the end time of an event may result in making invisible synchronization troubles. Thus, this approach is not possible in this context.

ThreadScope has been specially designed to detect synchronization problems in multithreaded softwares using traces [Wheeler and Thain, 2010]. To achieve this, ThreadScope represents the execution as a 2-dimensional graph where the vertices

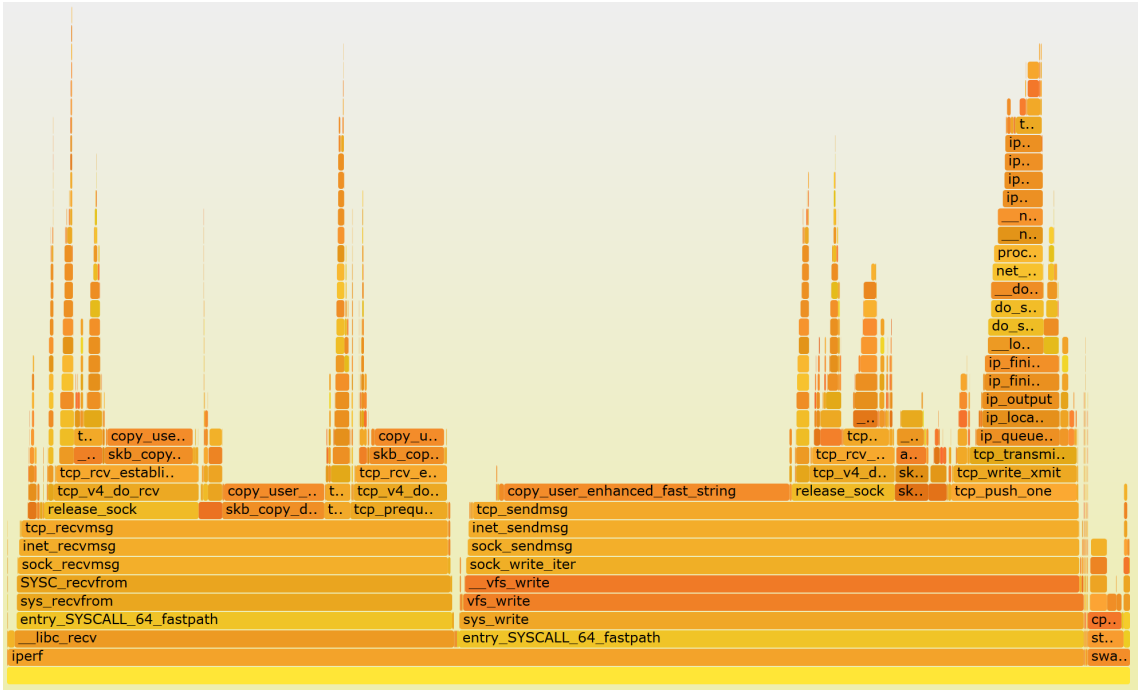


Figure 3.29: Flame Graph showing the call stacks during a program execution [Gregg, 2016a]

are either execution blocks or memory objects and the edges are special instructions executed by threads such as joins. The main limitation of ThreadScope is its lack of scalability with respect to the size of the trace. Neither aggregation mechanisms for overview and interactions are implemented.

Summary

We presented different approaches to visualize in details execution traces. Some tools such as the Gantt charts and its variations focus on providing a precise understanding of the sequence of events. Others (Flame Graph and ExploreViz) aim to explicit the function calls or the hierarchy of the software by using treemaps. Using all these tools, the software developers are able to understand in details the sequence of events on a time interval. However, scalability problems arise when working with huge traces. We showed that aliasing due to over-plotting can happen in Gantt charts for instance. ExploreViz provides a timeline to limit the time interval visualized, a solution also taken by various industrial tools such Trace Compass [Compass, 2016]. However, it makes the exploration of large traces tedious. Moreover, with streaming applications, periodic behaviors exist at different levels, from behavioral patterns composed of a couple of events to potentially large time interval involving several actors. The aforementioned techniques do not show the patterns at different scales due to the absence of aggregation technique based on data or simply because the time dimension is not explicitly encoded, hiding the synchronization between

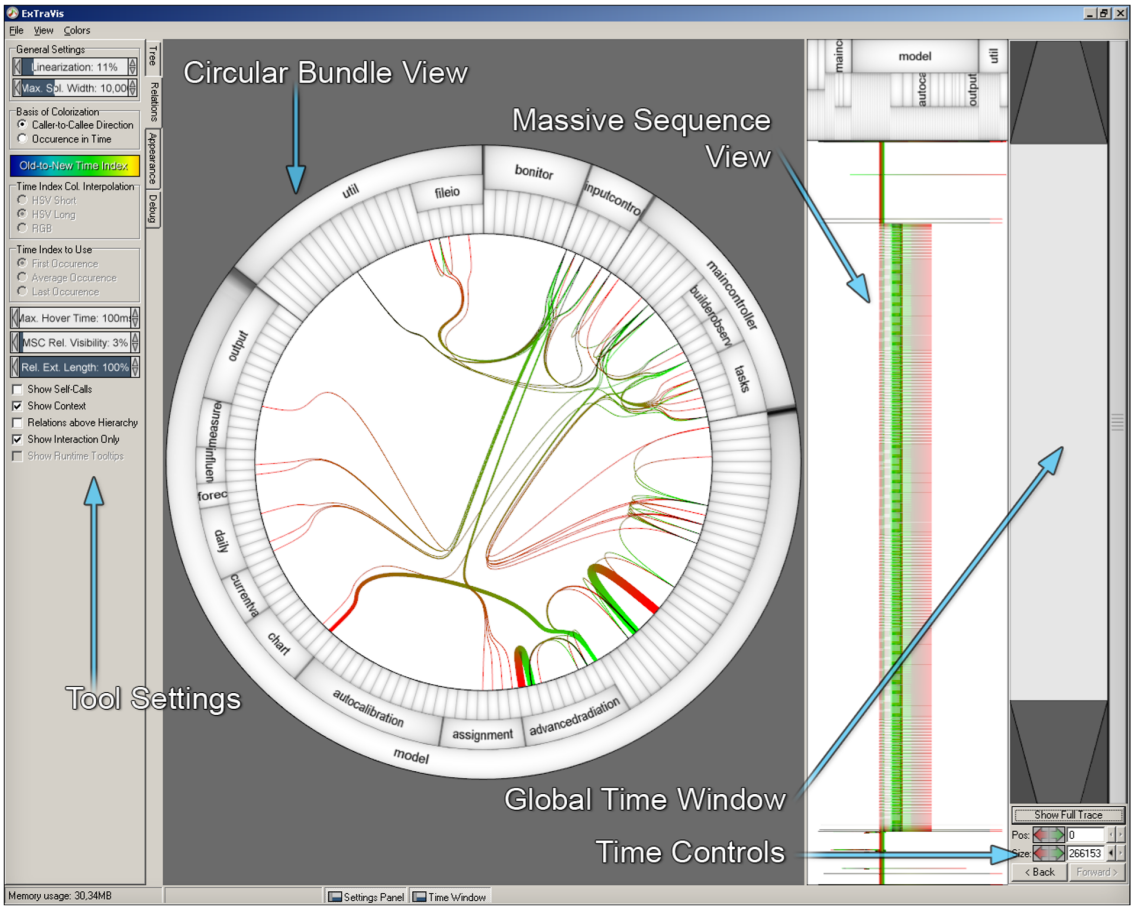


Figure 3.30: ExtraViz is composed of a circular view that shows the method calls and a vertical time [Cornelissen et al., 2007a]

the actors.

3.2.3 Summary: a Gap Between Overview and Detail Visualizations

We described the existing techniques providing a trace overview, useful to gain a global understanding of the execution but too abstracted to efficiently begin the analysis, and the approaches for a detail representation of the data very informative to study the local behaviors but not able to provide different levels of abstraction making fastidious the discovery of non-local behavioral patterns. Therefore, a gap exists between the overview and the detailed visualization techniques. While both are necessary in the analysis, developers still lack a tool that provides an overview of the execution yet with enough details to be able to quickly filter data and target a specific subset to analyze more deeply. To efficiently address the requirements of the debugging task, this tool should be able to provide a multi-scale exploration for

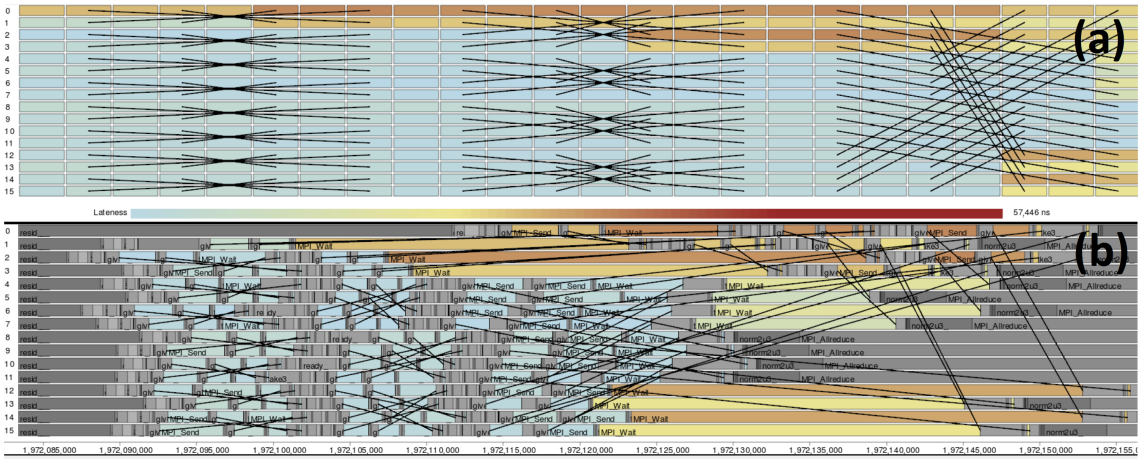


Figure 3.31: Ravel. In (a) the events are ordered according to the logical time and (b) is based on the physical time. The logical time clearly make more apparent execution patterns. [Isaacs et al., 2014a]

multiple large time series. This tool would come as a complement between a very abstracted and a very detailed visualization.

3.3 Pattern Visualization

Enriching visualization techniques with results from data mining algorithms will provide developers a more powerful toolset for debugging. There exists many different automatic techniques to find trends, outliers, abnormal behaviors, etc. However, exploiting these results is often tedious and reserved to data mining experts. In order for the developers to fully exploit them, it is necessary to leverage these results by integrating them into visualizations tools.

Multimedia applications have a periodic behavior by nature. It implies that repetitive structures will occur as well as dominant patterns. Discovering a break in the periodicity may provide a good start for a deeper analysis. Providing a powerful pattern visualization technique to the developers will further support the debugging analysis.

Pattern mining is a topic of data mining, a set of algorithms built for detection of relevant pattern inside a dataset without previous knowledge about the data. A widely used technique, and also the earliest, is the Apriori algorithm [Agrawal and Srikant, 1994] that mines frequent set of items, *itemsets*, in a dataset. Apriori was initially designed for market analysis but due to its genericity, researchers have applied this technique in many different fields. In the context of debugging streaming applications on embedded systems, pattern mining techniques have been developed to automatically detect periodic patterns [Lopez Cueva et al., 2012], periodicity perturbations [Iegorov et al., 2015], congestion points [Lagraa et al., 2012], and so on.

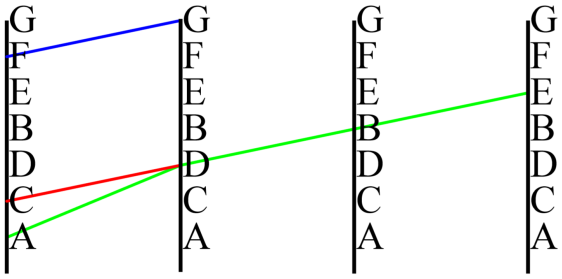


Figure 3.32: Visualizing frequent itemsets [Yang, 2003]

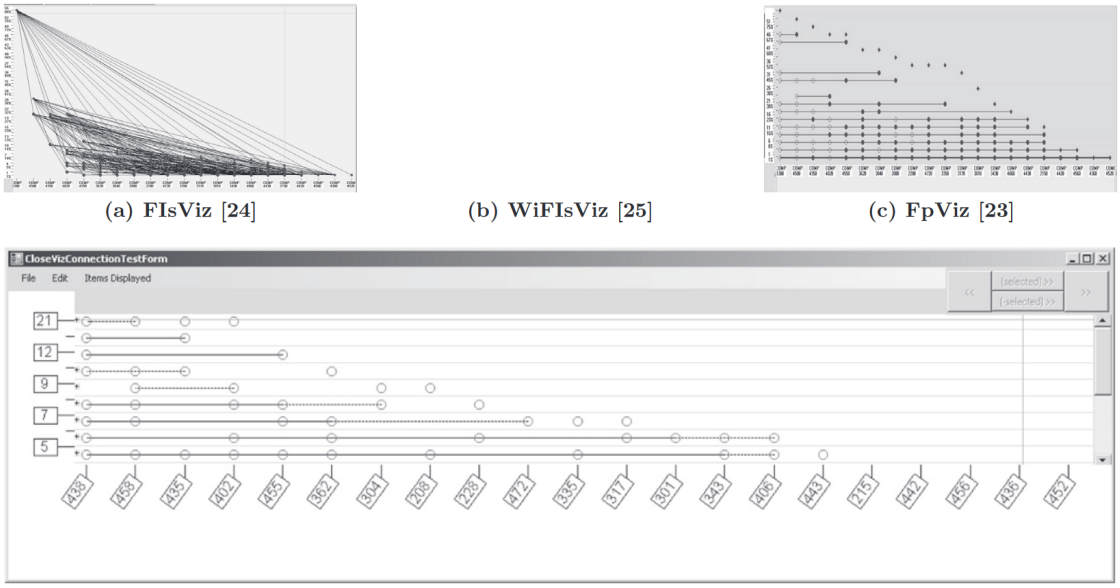
An field of research addresses the challenges of designing powerful visualizations for the mined patterns. In this section, we present the existing works on pattern visualizations and describe their limitations.

Mining frequent itemsets has been the first work in the domain of pattern mining, logically posing the first problems related to pattern visualization. The initial approach was based on using parallel coordinates [Yang, 2003, 2005] to visualize association rules and frequent itemsets (Figure 3.32). An itemset with k items (a k-itemset) is represented with curves linking k vertical axis. The thickness of the links encodes the support of the itemset. The items are placed on vertical axis and ordered by *groups*. Items belonging to the same *group* are ordered according their frequency in the dataset. The number of vertical axis depends on the longest itemset to represent. The different items are linked together by lines that connect the vertical axis, thus, a line visually represents an itemset. The pre-ordering done on the axis aims to improve the readability of the representation by minimizing intersections between the lines. The main limitation using parallel coordinates is the lack of scalability. When many patterns need to be visualized, the visualization becomes too cluttered with a large number of crossing lines making difficult the reading of a pattern.

CloseViz [Carmichael and Leung, 2010] adopts a different strategy. It visualizes only closed patterns with a single line and represents the items using a circle. It has the advantage of reducing significantly the amount of patterns to visualize. It is based on previous works FIsViz [Leung et al., 2008a], WiFIsViz [Leung et al., 2008b] and FpViz [Leung and Carmichael, 2009] (Figure 3.33).

FIsViz [Leung et al., 2008a] encodes the itemsets with polylines in a 2D rendering. The horizontal axis has k nodes for a k-itemset. The support of the items are encoded on the vertical axis. Similarly than with parallel coordinates, this technique quickly becomes tedious to read with many line crossing. WiFIsViz [Leung et al., 2008b] and FpViz [Leung and Carmichael, 2009] aim to solve this issue by grouping the patterns using common prefixes and horizontal lines instead of polylines. While the visualization benefits from these improvements, discovering relationships between the patterns and insightful information about the dataset remains a difficult task.

Other visualization techniques use a radial layout. FP-Viz [Keim et al., 2008] is a visualization tool for frequent itemsets. The items are placed on concentric



(a) FIsViz [24] (b) WiFIsViz [25] (c) FpViz [23]

(d) Our proposed CloseViz

Figure 3.33: Itemset visualizer using polylines (a) and lines (b), (c) and (d) [Carmichael and Leung, 2010]

circles and are represented by circular segments whose length encode its frequency. Therefore a k-itemset is rendered with k circular segments. The support of an itemset is encoded using a color-scale from green to red. When working with a large amount of itemsets, the information becomes tedious to read due to a high clutterness.

Bothorel et al. [Bothorel et al., 2013] proposed an other technique based on a circular layout, placing the itemset on concentric circles instead of items (Figure 3.34). The itemsets having the same cardinality are located on the same circle. The 1-itemset are disposed on the external circle and the k-itemsets on the k^{th} circle. Then, the frequent itemsets of each two neighbor circles are linked together. To improve the readability, an edge bundling algorithm is applied to simplify the graph between all the consecutive circles.

PowerSet viewer [Munzner et al., 2005] is an other frequent itemset visualization tool (Figure 3.35). The screen space is divided into horizontal bands, one band contains the itemsets of a given cardinality, the 1-itemset being on the top. An itemset is represented by a rectangle and its frequency is encoded in the color. This technique allows to have an overview of the frequent itemsets in the data but lack representation of the support, and is limited to a single type of pattern.

Note that there is a promising line of research in that field is to provide interactive interfaces for navigating the space of patterns, such as MIME from Goethals et al.[Goethals et al., 2011]. Such work are not directly related to ours as they are designed around interactions with the user to explore a potentially huge space of patterns, while we focus on a smaller space of patterns but aim at an immediate

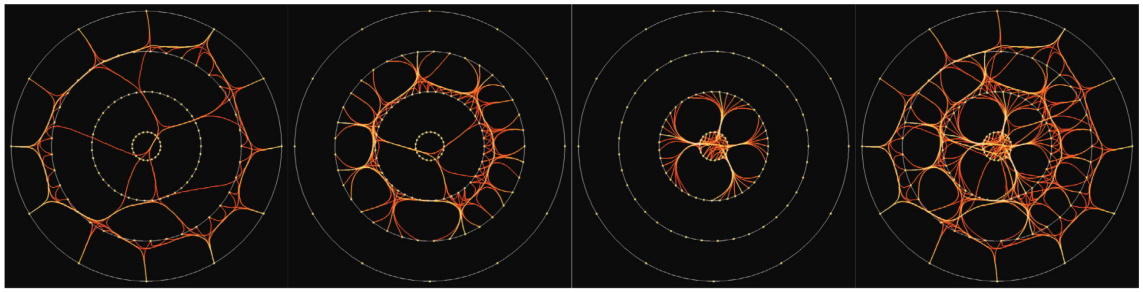


Figure 3.34: Circular layout and edge bundling visualization technique for frequent itemset. [Bothorel et al., 2013]

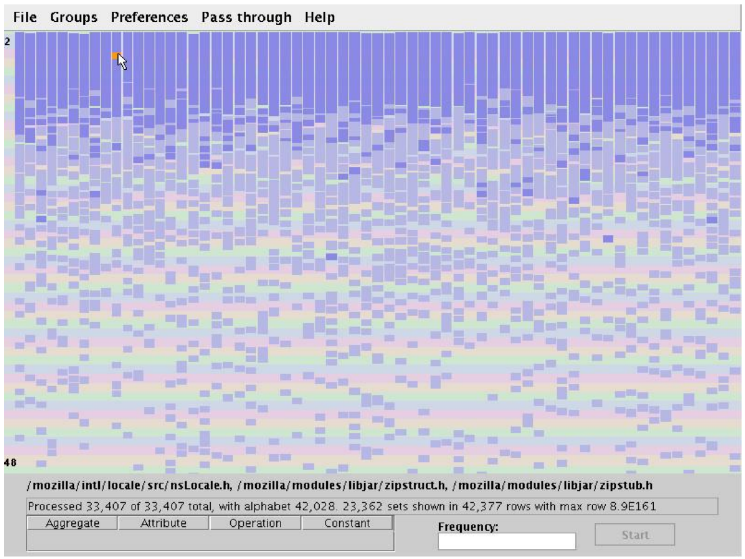


Figure 3.35: Powerset Viewer visualizes frequent itemset [Munzner et al., 2005]

understanding of the visualization.

Summary

All the previous work make the understanding of the individual items of the itemset a priority. They also rely on the complete set of frequent itemsets. In chapter 6, we propose a different approach: we consider patterns that are short (of a fixed length, only 2 or 3 items) but we put the focus on the different *structures* organizing these items: set, sequence, periodicity. The visualization technique is built around this idea: the structures are the main information shown by the visualization, avoiding combinatorial explosion while showing valuable and usually unseen information.

We present in the next chapter the challenges and research questions induced by the related works to address the general problem of providing better debugging tools for execution traces based on visualization techniques to shorten the analysis of an execution trace.

Chapter 4

Challenges for Trace Debugging

Contents

4.1 Inaccurate Time Series Rendering 58

4.2 Large Gap Between Overview and Detailed View 58

4.3 Slow Back-end Performances 59

4.4 Pattern Mining for the Visualization of Execution Traces 60

In this chapter, we recall what is the general problem to address that motivated this doctoral work and we summarize the different limitations of existing solutions to analyze execution traces.

In Chapter 2, we saw how the hardware and software platforms have evolved to deliver multimedia content with better quality, following the modern standards (1080p, 4K). We explained why the MPSoC became the solution to develop the powerful hardware needed for modern usage but introduced an increased hardware complexity. We show how the software has been impacted, resulting here again in an increased complexity coming from the new multimedia standards to support and the more sophisticated hardware.

We also described the specificities of debugging streaming multimedia applications: real-time QoS constraints to satisfy and the usage of execution traces to analyze the bugs as traditional debuggers are irrelevant in this context. A consequence when working with modern platforms is that the existing debugging tools are not able to tackle the huge amount of data in traces generated at the execution time. To stay relevant in a competitive market, semi-conductors companies, such as STMicroelectronics, have to keep the *time-to-market* as short as possible.

Under these conditions, the software developers are put under a huge pressure. Within a constant to reduced development time, they have to develop and debug increasingly complex multimedia applications using less and less relevant analysis tools as time goes. The goal of this doctoral work is to propose new tools that enable the developers to efficiently analyze execution traces to debug multimedia application in a very short time.

With this respect, we drew a state-of-the-art in time series visualization, pattern visualizations and industrial tools used nowadays. We found several limitations of the existing techniques and different challenges to solve in order to have efficient visualization tools to analyze traces. We summarize them in the following sections.

4.1 Inaccurate Time Series Rendering

On a fundamental level of visualization, the related work show a lack of techniques that provides a high visual precision, mandatory when working in a scientific or industrial context. There exists a large panel of research that investigates various aspect of temporal data visualization such as interaction techniques, different time representation (linear or cyclic), statistical analysis through visualization tools, different strategies to represent multiple time series, etc. However, we noticed the lack of work on how to draw a time series in itself, thus responding to the question *how to mitigate the visual artifacts while providing a smoothed visualization for time series?*.

Imprecise rendering can introduce visual artifact and produce inaccurate representation and mislead the user to a wrong conclusion when performing a fine grained analysis of the time series. In the context of this thesis, this would mean that the developers would be troubleshooting a wrong piece of software or being stuck in their debugging process, resulting in both cases in a waste of time and potentially to the introduction of new bugs in the large code base of the application. When a temporal bug occurred, it often concern a tiny number of events showing an abnormal sequence of system or function calls. Finding such bugs among several millions of other events is already a very time consuming and fastidious task. It is here mandatory that the visualizations showing different aspect of the behavior of the application are the most accurate possible. This represents one of the challenges to address for an efficient temporal debugging.

4.2 Large Gap Between Overview and Detailed View

For an efficient analysis of execution traces, it is mandatory to separate the different actors (the interrupts, the processes, etc.). It allows to better study the synchronization between the actors, their individual behavior, etc. Therefore, for an efficient analysis, it is important to visualize the multiple time series contained in the trace, one time series corresponding to an actor and containing all the events produced by this actor.

Many research have been done on visualizing multiple time series (Section 3.1.2) and two main strategies to represent several time series arise from these works: sharing the screen-space with all the time series (the *shared-screen* approach) or splitting it into a number of sub-regions, one per time series (the *split-screen* approach). Each

of these approaches has its pros and cons as discussed earlier and have been used to propose new solutions to different problems such as finding a local or global extrema among different time series. These strategies have been applied to trace visualization in different works presented in Section 3.2. Among all the existing tools, whether they are research prototypes or industrial tools, we noticed a consequent gap between the visualization tools that focus on giving a global overview of the trace and those aiming to provide fine grained details about the trace. The former one make hard to find temporal bugs involving a small sequence of events: they provide a very summarized view of the trace with no possibility to get more details on-demand. This kind of view is quite useful to have an abstract understanding of the behavior such as the evolution of the CPU load during the decoding but is not suitable for fine-grained analysis. The latter makes browsing large traces a fastidious and slow task: each event is accurately represented on a timeline which is the ideal representation to focus on a very specific sub-time window. However, we saw how readability problems can arise as the number of events to visualize increase. Existing tools falls in of these two categories leaving a gap in the developers toolkits and raises the question: *Which visualization techniques to develop for a tool that provides enough information about the behavior of the application to begin the investigation, yet high-level enough to be efficient at exploring the data?*

One approach found in the industrial tools is to split the trace into pages with one page containing a small portion of events of the trace. This had two advantages. The first one is that the exploration of a detailed view is easier as only a small amount of event are represented. It provided a workaround that was good enough with older generations of multimedia decoding platforms. However, with the latest generation, this approach has reached its limits and does not provide fluid trace exploration as the number of pages can be up to thousands.

4.3 Slow Back-end Performances

The second advantage of slicing the trace in pages is that it guarantees that the answers to queries made on the data are mostly contained in one page. It was an advantage in the past since the disk access and the back-end solutions were much slower than the recent ones. But the storage technologies have improved with the democratization of the Solid State Disks (SSD) that provide drastically faster reading time. Therefore, constraining the visualization to perform queries targeting most of the time a pre-defined part of the trace was a good workaround to leverage the time to access the data but is no longer relevant on modern workstations. Moreover, as the size of execution traces are dramatically increasing, the page mechanism raises an other problem as the result of a query now spread very often across several pages. Consequently, it makes some behaviors or patterns very hard to detect.

Indeed, the large amount of events generated by modern applications ($\approx 10^6$ per minute of execution) requires an efficient back-end to develop interactive tools. More precisely, with the actual computational power of modern workstations, the

back-end became the limiting factor of the visualization pipeline to interactively browse a large amount of data. The challenge can be summarize as: *How to develop a back-end solution suitable for the interactive exploration of time series containing millions of events?*

4.4 Pattern Mining for the Visualization of Execution Traces

Among research and industrial tools, very few have investigated the integration of data mining results into visualization techniques for execution traces. This can accelerate the debugging process by providing insightful information to the developers, hardly visible or computable otherwise. By nature of the logged application (a streaming multimedia application), the traces recorded during an audio and/or video decoding containing many repetitive sequences of events. We described in Chapter 2 why, by definition, decoding a multimedia content is a repetitive and periodic activity. As a quick and very simple reminder, decoding a video consists in decoding 30 frames per second. Therefore, tracing such application will produce a trace that contains repetitive sequences of events that correspond to the piece of software in charge of doing a particular task in the decoding process. Data mining can be very useful in this context. More specifically, pattern mining can reveal very useful information about the behavior of the application to the developer and makes possible to automatically detect abnormal behaviors and filter irrelevant data. For instance, we saw in the related works (see Section 3.3) that pattern mining approaches exist to mine periodic patterns in the trace [Lopez Cueva et al., 2012], detect perturbations [Iegorov et al., 2015], etc. Using such information to build a novel visualization or augmenting an existing one would result in more powerful debugging toolkits for execution traces. While able to find insightful information, these tools return the results as a long list of patterns, fastidious to analyze and that often require some data mining knowledge to understand them.

Some visualization techniques have been proposed to address this problem. However, as shown in the review of pattern visualization techniques, existing approaches often require an expert user to understand what is visualized. Moreover, data mining algorithms generate a large amount of patterns but pattern visualizations lack scalability, quickly becoming cluttered as the number of patterns to show to the end user increases. Here, the challenge to address is: *how to exploit pattern mining techniques to enrich debugging tools for execution traces?*

From the existing works, we could identify different challenges to address in order to provide new visualization tools to debug streaming multimedia applications using execution traces. Doing so would help the more general industrial problem that is to minimize the *time-to-market* of their multimedia platforms. In the second part of this thesis, we present our different contributions to address this general problem and propose solutions to the different challenges described in this chapter.

Part II

Contributions

Chapter 5

Research Approach and Evaluation Methodology

Contents

5.1 Research Approach 63
5.2 Evaluation Methodology and Validation 64

In this chapter, we introduce our research approach and the different contributions made during this doctoral work with respect to the different challenges summarized in Chapter 4. Then, we describe the strategy we followed to conduct the evaluation of our different works and give some context about STMicroelectronics to better explain our choices.

5.1 Research Approach

We have identified several limitations and problems to solve in order to provide more efficient debugging tools for execution traces. In this perspective, we address the challenges presented in Chapter 4 in the following chapters.

First, we spotted a lack of technique to accurately visualize a time series. This concerns a much broader domain than execution traces and touches a fundamental aspect of time-oriented data visualization. Therefore, we begin in Chapter 6 by focusing on this problem and propose a novel smoothing visualization technique for time series. After an analysis of the visual artifacts that can appear in existing temporal data representation, we propose a pixel-precise visualization for time series, the Slick Graphs. We will use the Slick Graphs in the two other works presented in this thesis as the basic brick for trace overview and smoothing.

The three other questions identified previously are more domain specific. We saw that there is a huge gap between the tools giving an overview of the trace and those giving many details. There also exists a limitation concerning the exploration of huge traces: actual solutions do not allow an interactive browsing despite the

now broad usage of SSD and powerful workstations. We propose a solution to these two questions in Chapter 7. We present a visualization framework called TraceViz. This proposition brings two contributions: a new fast back-end to manage execution traces and a novel visualization technique that provides an overview of the trace yet with enough details to begin to understand the relations between the different actors of the trace and to visually spot behavioral patterns. In this view, we used the pixel mapping algorithm used in the Slick Graph to guarantee an accurate visualization. We also integrated a global overview of the execution built using a Slick Graph. Coupled to the back-end, the TraceViz visualization supports interactive exploration of the trace.

Lastly, we noticed that pattern mining results are often difficult to exploit while it can enrich debugging tools with meaning insights about the execution. In a third chapter, we present a different type of overview than the one proposed in TraceViz. Here, we use pattern mining techniques to find hidden structures in the trace such as sequences of events, periodic sequences, which are the most frequent set of events that occurred, etc. The goal of this work is still to provide new information about the execution that would be hard and time consuming to find otherwise.

Through all these works, our goal is to provide new approaches for more efficient debugging tools aiming to reduce the time needed to troubleshoot streaming multimedia applications. First, by ensuring the visual representation of the trace is as accurate as possible to avoid reasoning mistakes based on artifacts. Second, by using our new pixel-based technique as the basis to build a novel type of overview. Finally, by taking advantage of data mining algorithms to provide new insights.

5.2 Evaluation Methodology and Validation

We adopted different methods of evaluation for the different works, taking into account the nature of the technique to evaluate and the industrial context in STMicroelectronics.

5.2.1 Slick Graphs Evaluation

The Slick Graphs visualization technique could be evaluated independently to the application domain as it can be applied to visualize any type of time series. Therefore, to measure the benefits of the Slick Graphs, we conducted a controlled user evaluation in laboratory, detailed in Section 6.5.

5.2.2 TraceViz Evaluation

We took a different approach to evaluate TraceViz and proceed in two steps: one for the back-end and one for the visualization. Concerning the back-end, we measured reading and writing time under different conditions and compared these measurements against SQLite, the most used database in debugging toolkit with traces.

We detail the procedure and the results in Section 7.2. The visualization part of TraceViz is obviously linked to the application domain. Here, software engineers working on video decoding are mandatory to run a user study. However, developers at STMicroelectronics are difficult to reach and spread across different offices on several sites. These conditions made impossible to conduct a formal user evaluation. Instead, we took a different approach, in both the development of TraceViz and its validation.

The development was made in collaboration with the software engineers in the tools development team at STMicroelectronics. We had an iterative development where we delivered several versions of the tool. Simultaneously, we had regular discussions where we exposed the new concepts and they provided us their feedback based on their experience and knowledge of the domain. During this phase, we also evaluated how to use TraceViz on already solved use cases.

The validation of TraceViz consists in the integration of the visualization part into the STMicrolectronics, as discussed in Section 7.6. STMicroelectronics was also interested in migrating its complete suite of tools to the TraceViz back-end. It could not be done simultaneously with the visualization part as it required significantly more work and no resources where available at this moment to do this task. Therefore, this development was initially planned for future releases of their tool but never happened due to reasons explained below. From this point, we could follow some real use cases where TraceViz successfully supported different bugs resolutions. We present two of them in Section 7.5. These elements consist in the validation of TraceViz.

5.2.3 Structures Visualization

The work presented in Chapter 8 took place last chronologically. At the moment of its development and evaluation, the situation had changed due to a modification of the STMicroelectronics strategy as explained in Section 1.3. In January 2016, STMicroelectronics has announced that the set-top box activity stops. Consequently, the teams were dissolved and engineers reallocated to different activities (more than 1000 employees impacted). While this process took several months, no further development were made. This situation made impossible to follow a similar work methodology than the one for TraceViz explained above and forced us to adapt a different evaluation strategy.

We proceeded in two steps: (1) evaluating our approach on execution traces and (2) using different types of traces to show how our method is not bounded to a single application. In the case of execution traces, we took use cases solved in the past at STMicroelectronics for which the traces and the bug explanation were available. We ran our visualization on these traces and verified that the visualized results were in adequation of the behaviors discovered by the engineers. To strengthen our evaluation, we decided to apply our technique on other type of data and checked the relevance of the visualized information. We present these evaluation in Section 8.5.

Chapter 6

Slick Graphs: Slick Visualization of Time Series

Contents

6.1 Introduction 68

6.2 Smoothing Techniques for Accurate Visualization Techniques 69

6.3 Study Case: ThemeRiver Smoothing Algorithm 70

6.4 Slick Graphs 74

6.5 User Study: Evaluation of the SLG Smoothing Technique 78

6.6 Integration with Existing Techniques 85

6.7 Conclusion 89

Representing accurately a time series is mandatory in a scientific or industrial applications. As explained in Chapter 2, execution traces of multimedia streaming applications contain many repetitive and periodic sets of events. An inaccurate visual representation of the trace can lead to visual artifacts that modify or hide the real behavior described by the events. This is not only true for execution traces but for all types of temporal data. For instance, the same reasoning applies for a time series representing the variations of temperature over several years. When visualizing the whole dataset, inaccurate visualization may position peaks at wrong positions, potentially mistakenly showing a warm month instead of a cold one.

In related works, we noticed a lack of precise technique to visualize a time series and in Chapter 4, we stated this problem as being a research question to solve in the domain of data visualization to improve the quality of debugging tools. In this chapter, we focus on addressing the following question mentioned in Section 4.1:

How to mitigate the visual artifacts while providing a smoothed visualization for time series?

6.1 Introduction

The simplest method to represent a time series is a *line graph*. While this technique is widely used, it has many limitations to visualize huge time series due to two factors, previously discussed in the related work (see Section 3.1). Briefly, the readability of the graph greatly depends on its aspect ratio and on the average slope of its segments. A solution to improve the legibility of the line graph is to smooth the data: Bar et al. found that edges are harder to read than curves [Bar and Neta, 2006]. Bézier curves are popular in information visualization and more generally in computer graphics to draw curved lines (see ThemeRiver [Havre et al., 2002], and later works [Byron and Wattenberg, 2008; Dork et al., 2010] for examples of time series visualizations using Bézier curves to smooth the data).

Many works improved the line chart using both visual techniques and new interactions (see Section 3.1). However, there has been few research on the smoothing technique for time series to improve the legibility of the line graphs while keeping a high accuracy visualization. To address these problems, we introduce *Slick Graphs*, a novel interactive visualization for time series that provides a high accuracy data smoothing technique. Data is aliased at the pixel level, reaching the limit of perception of the human eye on screens with high densities of pixels (more than 200 dots per inch). The Slick Graphs are accompanied with interactions to control the smoothing factor allowing the discovery of trends in the data at different scales and reducing the visual clutter on-demand to satisfy the Cleveland's legibility conditions of a line graph [Cleveland, 1993]. Slick Graphs are built upon line graphs, the most common visualization technique for time series [Cleveland, 1993]. Thus, they can easily be integrated with existing techniques (e.g. Interactive Horizon Graph, Stacked Graphs, etc.) to bring them an efficient solution to support huge time series and be more precise. We conducted a user study to evaluate the performance of Slick Graphs for basic tasks on time series. It shows that Slick Graphs outperform traditional smoothing technique used in visualization techniques.

We present two main contributions:

1. a binning and aggregating method that minimizes the aliasing to the pixel level; and

2. a novel high accuracy smoothing technique that can be integrated with existing visualizations for time-oriented data and we show examples to demonstrate how to do it.

We begin to describe the different possible strategies to bin, aggregate and smooth data in the goal of producing accurate visualizations of time series based on line graphs (Section 6.2). Section 6.3 provides a detailed explanation of the smoothing technique used in popular time series visualization and highlights the issues raised by the binning. We present in Section 6.4 the Slick Graphs visualization that comes with new binning and smoothing strategies based on pixels. Section 6.5

explains the controlled experiment we conducted to evaluate the efficiency of our smoothing technique compared to the previous one and finally, we provide in Section 6.6 examples of Slick Graphs integration with existing techniques and discusses the benefits.

6.2 Smoothing Techniques for Accurate Visualization Techniques

There exists a large panel of data smoothing techniques coming from various domains such as signal processing and statistics. All have different mathematical properties depending on the information of interest to extract by filtering-out the noise from the input data. For example, smoothing a signal using a Gaussian kernel convolution removes the high frequencies from the raw data as it behaves as a low-pass filter.

All these techniques can be used in visualizations to support information extraction, or in a simpler goal, to produce smooth graphs. However, the resulting rendering must be accurate by respecting the properties of the input and must guarantee the extrema are precisely located. To achieve this, the resolution (i.e. the number of pixels) of the output displaying the visualization has to be considered.

If the input has a smaller number of items than the resolution, the smoothing pass can be directly applied on the data and the result can be plotted with no transformation and be accurate. However, when the number of items contained in the data is greater than the number of pixels, some binning and aggregations techniques are mandatory. Two scenarios are possible: (1) perform the smoothing on the raw input and then bin and aggregate the smoothed values or (2) bin and aggregate the data before smoothing.

In the following paragraphs, we explore the two scenarios, give examples to better illustrate the implication for rendering precise visualization and present an in-depth analysis of an existing smoothing technique developed with the goal of producing smooth graphs.

6.2.1 Smooth First, Bin and Aggregate Second

In this case, the method consists in smoothing the input first, then bin the smoothed result and finally aggregate the data contained in each bin. Under these conditions, the result of aggregation is directly plotted using a line graph to join the aggregated points. This process does not guarantee the smoothness of the output graph as the binning and aggregating steps breaks the continuity of the resulting curve. More formally, the smoothness of a curve is measured using the geometric continuity G^n. The eye sees a curve being smooth if it is G^1 continuous (i.e when considering a point on the curve, the tangent vectors of the segments on either side of the point share the same direction). In our case, it can only happen when the aggregated values are constant. This process must not be used for achieving smooth visualizations.

6.2.2 Bin and Aggregate First, Smooth Second

In this scenario, putting the smoothing step last guarantees the smoothness of the rendering. The accuracy of the visualization now depends on the binning as it aliases the data by nature.

Let consider a time series of statistical data. Binning the time series is typically performed as follows. The time series is split into several time windows of equal duration d. They are represented by tw_n in Figure 6.1 and their boundaries are the dashed lines. The aggregation step consists in computing a statistic on the data contained in each time window (i.e average, median, min, max, etc.). These are the results used as input of the smoothing step. They correspond to p_0, p_1 and p_2 in Figure 6.1, respectively for the time windows tw_0, tw_1 and tw_2.

To achieve a precise smoothed visualization of the time series, it is necessary to minimize the aliasing introduced during the binning process. In the next section, we provide an in-depth study of the smoothing technique introduced by ThemeRiver, used in later work and based on arbitrary binning.

6.3 Study Case: ThemeRiver Smoothing Algorithm

In this section, we analyze the smoothing technique of the ThemeRiver algorithm as an example to highlight the impact of the binning process on the quality of a visualization. We describe the visual artifacts that appear due to the aliasing to better understand the role of the binning strategy and aggregation technique in the final rendering.

The ThemeRiver technique stacks different layers corresponding to several time series. As we focus on the binning technique, we do not consider the legibility issues induced by stacking different layers. A study of these issues has been described by Heer et al. [Heer et al., 2009].

6.3.1 Layer Building

The binning and aggregation steps of the ThemeRiver algorithm works as described in paragraph 3.2. The size of the time windows are typically 10 pixels wide (10 to 50 pixels).

The smoothing step consists in linked together the points p_n using a succession of Bézier curves. At each data point a curve ends and another begins. Two control points are placed to the left and to the right of each data point and are located on the boundaries of their corresponding time window. The control points, represented by p_{na} and p_{nb} in Figure 6.1, are aligned horizontally, ensuring that the global curve have a smooth shape.

In the next paragraphs we describe the legibility problems introduced by such construction method.

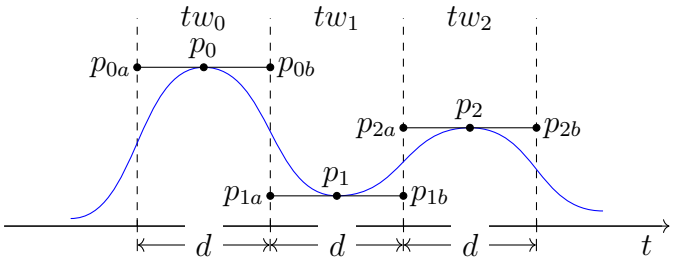


Figure 6.1: ThemeRiver layer building. The timeseries t is split into n time windows of duration d. A statistic is computed for each time window. It gives the data points p_0, p_1 and p_2. Two consecutive data points are linked using a Bézier curve. The control points p_{na}, p_{nb} are placed horizontally on the time window boundaries.

6.3.2 Legibility Problems

Arbitrary Aliasing

The direct effect of time quantization on the data is to introduce an artificial aliasing that is directly correlated to the size and thus the number of time windows. The number of points where the data is exactly represented is n with n being the number of time windows. Most of the time, n is kept relatively small making the graph very imprecise (e.g. 20 to 100 points with time windows of 10 to 50 pixels and a 1000 pixels wide graph). The information is conveyed by the other points: they only link the aggregated values in a smooth manner but are no more representative of the underlying data as a bar chart would be. This general problem has many specific consequences detailed below.

Constrained Location of the Peaks

By construction, the peaks can only appear at the control points. A well-known property of the Bézier curves is that they are contained in the convex hull defined by their control points. From the position of the control points (see Figure 6.1), it implies that it is impossible to have a peak at another point than in the middle of a time window.

Visual Inaccuracy

The graph can be visually inaccurate, hiding the local extrema or showing them shifted on the left or on the right. This is a direct consequence of the constrained location of the peaks at the center of the time windows and of the binning, as stated in the previous paragraph. Figure 6.2 shows an example where the position and width of the time windows lead to such error of representation. In this case, the smooth curve shows a local minimum instead of the local maximum that is present in the raw data.

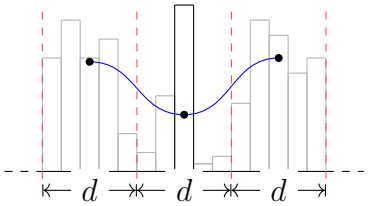


Figure 6.2: The graph shows a local minimum instead of the local maximum present in the input. The gray color is the histogram representing the raw data. The blue curve is what the final user will see and is the result of the smoothing method. The red dashed lines are the boundaries of the time windows.

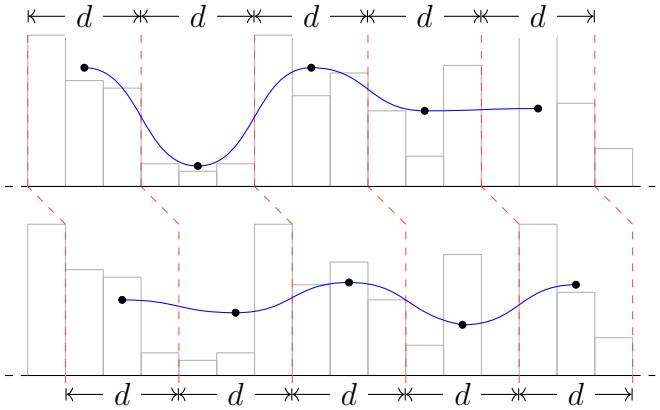
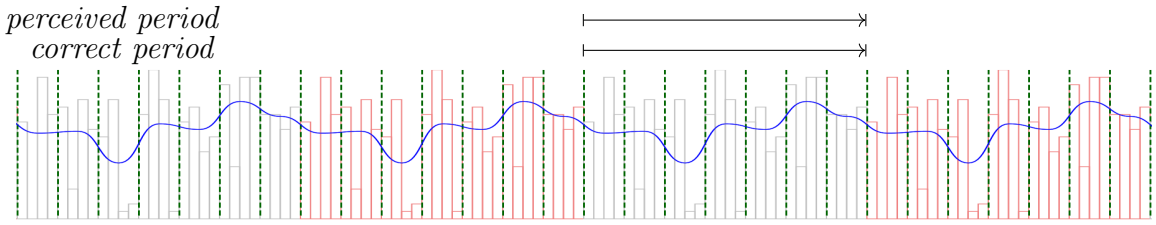


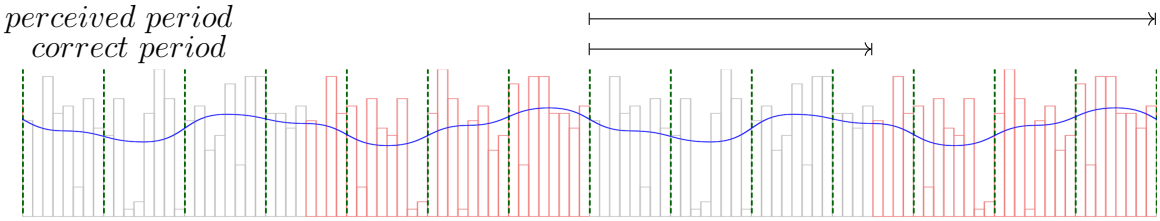
Figure 6.3: Impact of the position of the time windows on the shape of the curve. The histogram represents the raw data. The blue curve is the result of the smoothing method. The red dashed lines are the boundaries of the time window.

Furthermore, the shape of the curve greatly depends on the parameters of the time windows. Figure 6.3 shows two histograms that represent the same data. The time windows have the same duration d, and have just been shifted to the left by one sample in the bottom histogram. The boundaries of the time windows are represented by the red dashed lines. The shape of the blue curve is drastically affected by this minor change: in the top graph, the local minimum clearly appears while it remains barely visible in the bottom graph. This highlights the critical impact of the position of the time windows.

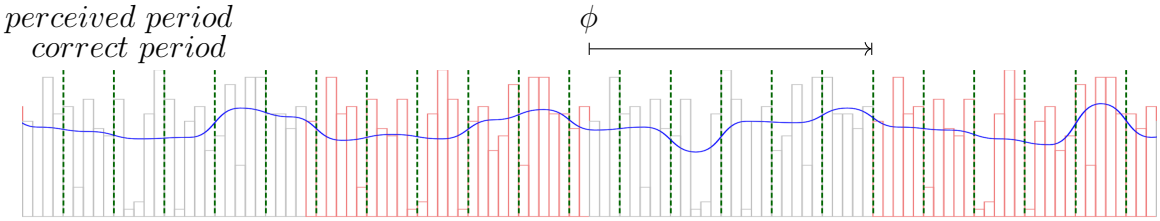
This phenomenon can be very problematic when the user can interactively manipulate the width of the time window or pan the data, e.g. as in Visual Backchannel [Dork et al., 2010]. The shape of the curve can greatly change while the user is interacting with the view, breaking the visual continuity. This makes it very difficult for the user to build a mental model of the data, a crucial process for good understanding.



(a) Layer of a Stacked Graph showing the correct period.



(b) Layer of a Stacked Graph showing the wrong period.



(c) Layer of a Stacked Graph showing no period.

Figure 6.4: Inaccurate representation of a periodic signal

6.3.3 Wrong period depiction

When working with time related data, periodic data are common. Using the ThemeRiver algorithm to represent periodic data can be misleading. In Figure 6.4, the histogram shows the raw data ; and periods (of width $p = 27$) are represented alternatively in black and red. In this case, six periods are represented. The blue curve is the smoothed result. The duration of the time windows (green dashes) varies between 4 (Figure 6.4a), 5 (Figure 6.4c), and 8 (Figure 6.4b).

Period is correctly represented.

It is possible to have the period correctly represented (see Figure 6.4a), if the width of the time windows $width_{tw}$ respects the condition $p = n \times width_{tw}$ where p is the period of the signal and n is an integer.

A wrong period is represented.

Figure 6.4b shows an example where a period is not visualized correctly. The blue curve shows a period $q = 2 \times p$ where p is the period of the raw data. More generally,

a wrong period is represented by a layer when the width of the time windows is not a multiple of the period, i.e. $p \not\equiv 0 \pmod{width_{tw}}$.

Period is hidden.

Figure 6.4c shows an example of a layer that hides a period. This case appears under the same conditions than when a wrong period is represented.

6.3.4 Summary

We have presented the legibility problems related to the ThemeRiver algorithm. These are consequences of the arbitrary aliasing that results from the low number of time windows and their large width, accentuating the aliasing. The constrained position of the local extrema, the visual inaccuracy, the instability of the shape of the curve related to the parameters of the time bins and the wrong representation of the period when working with a periodic data are all consequences to the time quantization process and of the Nyquist frequency. The smoothing technique developed for ThemeRiver is irrelevant for periodic data and data that change at a high frequency rate. However, the exposed problems have a small impact on data that vary at a low frequency rate or that follow a Poisson distribution like the number of Twitter posts about a particular topic [Dork et al., 2010], box office hits [Bloch et al., 2008], or visualizing a single person's history music listening [Byron, 2006].

In the following section, we introduce a new visualization technique that mitigates the visual artifacts introduced by the aliasing.

6.4 Slick Graphs

Related work suggests that the legibility of a line graph depends on the average slope of the line segments and on the aspect ratio of the graph. These conditions may not be respected when working with large scale and small scale variation datasets. ThemeRiver [Havre et al., 2002] proposed a smoothing algorithm suitable for large variations but irrelevant when working with small variations dataset. Slick Graphs (SLG) aims to mitigate these issues by introducing a pixel-based binning and smoothing techniques. By doing so, it reduces the aliasing up to the pixels, the smallest visual discretization achievable on a screen. In this section, we formally describe the data and explain the time series properties used by the algorithm. Then, we describe the algorithm implemented in SLG.

6.4.1 Time Series as Data

Data considered in SLG are time series consisting in collection of tuples (t_i, v_i) where t_i corresponds to the date when the event occurred and v_i to the observed value at

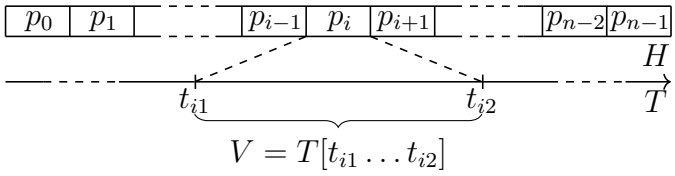


Figure 6.5: Building of the histogram H for Slick Graph

moment t_i. We have $0 \leq i < n$ with n being the number of events in the data. For a given time series, we have: $\forall i \in [0, n[,\ t_{i-1} < t_i$.

6.4.2 Slick Graphs Binning Algorithm

The SLG binning algorithm takes two parameters as entry point: a time series T and the width in number of pixels p of the space available to display the graph. The binning consists in building an histogram H of p bins, each bin corresponding to a pixel (see Figure 6.5). For each pixel p_i, we compute the two timestamps t_{i1} and t_{i2} at its boundaries and extract data contained in this time window. We have $V = T[t_{i1} \dots t_{i2}]$. Next, we compute $H(p_i)$ so that:

$$H(p_i) = f(V_i, i_1, i_2)$$

with f being an aggregation function (i.e. average, median, min, max, etc.).

The output of the binning step is the histogram H. This computation of the aggregated data outputs a result aliased at the pixel level. The visual artifacts described in the previous section due to the aliasing are still present but are mitigated to the pixel.

6.4.3 Slick Graphs Smoothing Algorithm

The smoothing step computes the smoothed values that will define the overall shape of a Slick Graph. Since a timeseries have only one dimension, SLG basically convolves the histogram H with a 1D kernel function. Thus, for each pixel p_i, we have:

$$SLG(p_i) = (H * K)(p_i) = \sum_{n=-w/2}^{w/2} H(p_i - n) \cdot K(n) \tag{6.1}$$

with w being the kernel width. Table 6.1 shows different rendering of the same input data smoothed with different kernels. The output graph produced by SLG guarantees a minimal aliasing to the pixel level.

The width w of the kernel controls the smoothing strength applied to the aggregated data: the smoothing increases as w increases. Giving the control of w to the user allows to interactively change the smoothing without changing the aggregation.

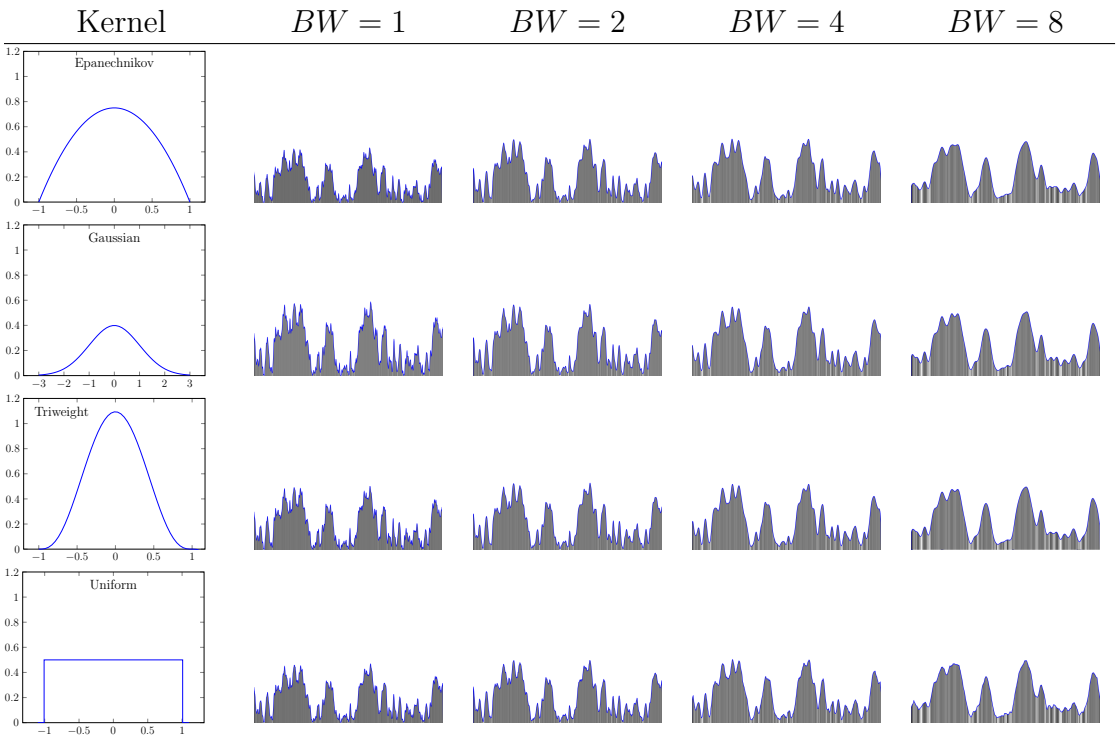


Table 6.1: Slick Graph smoothing data using different kernel functions. The kernel size is increasing from left to right and are respect equal to 1, 2, 4 and 8 pixels width. The canonical bandwidth of the kernels have been adjusted to achieve equivalent smoothing strength.

6.4.4 Encoding the Filtered-out Information

By definition, the smoothing process eliminates some information contained in the data to give a more general tendency across time. The nature of the filtered-out data depends on the kernel function used for the smoothing. To mitigate the loss of information, SLG encodes the difference between the smoothed value $SLG(i)$ and the aggregated value $H(p_i)$ in the luminance channel L_{p_i} of each pixel p_i:

$$L_i = \begin{cases} \dfrac{1}{1 + \frac{SLG(p_i)}{H(p_i)}} & \text{if } H(p_i) \neq 0 \\ 0 & \text{if } H(p_i) = 0 \end{cases}$$

It gives:

$$\begin{cases} L_i = 0 & \text{if } SLG(p_i) \gg H(p_i) \\ L_i = 0.5 & \text{if } SLG(p_i) = H(p_i) \\ L_i = 1 & \text{if } SLG(p_i) \ll H(p_i) \end{cases}$$

Thus, the local extrema appear at the exact position as bands of different shades of gray, white being for the local minimum and black for the local maximum (see

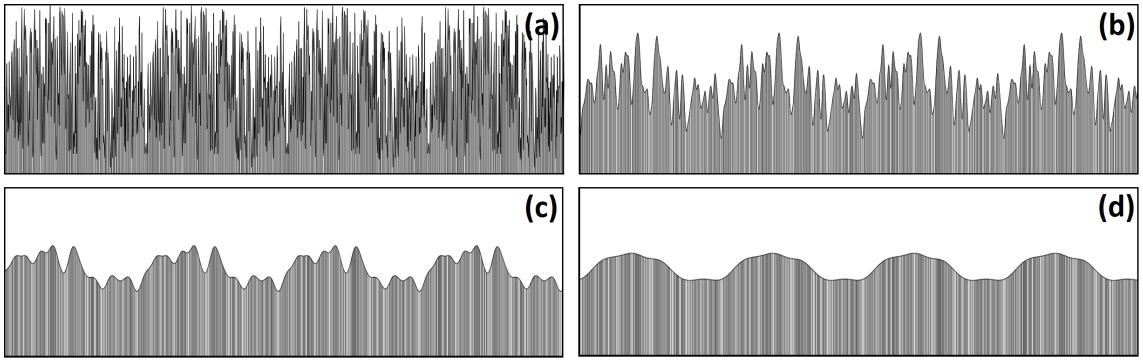


Figure 6.6: Using a Gaussian as kernel, SLG can reveal low frequency patterns by increasing the smoothing factor from (a) to (d).

Table 6.1).

6.4.5 Use Case: Slick Graphs as a Low-Pass Filter

We consider here a Gaussian kernel to smoothing with SLG. In this particular case the value of a pixel p_i is computed as:

$$SLG(p_i) = (H * G)(p_i) = \sum_{n=-\lceil 3\sigma \rceil}^{\lceil 3\sigma \rceil} H(p_i - n) \cdot G(n) \tag{6.2}$$

with:

$$G(n) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{n^2}{2\sigma^2}}.$$

Here, the Gaussian standard deviation σ is used as the smoothing parameter in SLG: when σ is small, the smoothing is low. A well-known property of the Gaussian kernel is to behave as a low-pass filter. Thus, smoothing using a Gaussian kernel reduces the high frequencies components contained in the histogram H. When the smoothing factor σ increases, the range of filtered out frequencies increases. More information is removed from the shape of the graph making apparent patterns at low frequency. The different shades of gray become lighter and darker as the difference between the smoothed value $SLG(p_i)$ and $H(p_i)$ increases.

In this use case, we consider the specific task of finding the period of a signal. With no smoothing, the raw histogram H is shown (e.g. Figure 6.6a). In this configuration, it is very difficult to detect the period visually. Increasing progressively the smoothing factor (Figure 6.6a to Figure 6.6d), the high frequency variations disappear from the shape and the period at low frequency can be easily spotted on the silhouette of the graph (Figure 6.6d). The gray shading of the graph shows the high frequency patterns.

6.5 User Study: Evaluation of the SLG Smoothing Technique

To evaluate the effectiveness of the SLG smoothing technique, we have conducted a quantitative user experiment. We wanted to investigate the impact of the SLG smoothing technique on user performance for basic tasks on time series and compare it to the smoothing technique used by ThemeRiver and Stacked Graphs, referred to STG smoothing technique in this section.

We used a Gaussian kernel for SLG during this experiment as it is the kernel giving smoother silhouette of the graphs. In the Slick Graphs version, we removed the encoding of the filtered-out information. In this study, we focus exclusively on the quality of the smoothing between the two techniques. For this, we need the user to validate his choices based on the silhouette of the graph only. Adding the encoding of the filtered-out information would perturb the experiment as the channel alpha could be used in addition of the silhouette.

6.5.1 Hypotheses

H1 *Slick Graphs smoothing technique will be more precise.* The arbitrary aliasing and the visual inaccuracy of the STG smoothing technique will increase the errors.

H2 *Slick Graphs smoothing technique will make the user faster.* The constrained location of the peaks and the inconsistent shape of the curve —that depends on the time bins parameters (see Figure 6.3)— creates visual discontinuity when exploring the data that slows down the user.

6.5.2 Tasks

Participants were asked to perform various tasks. These tasks are typical of what is relevant for time series.

Perception Task

This task is a derivative of the Perin et al.'s task *Same* [Perin et al., 2013]. Participants are asked to choose between two graphs, smoothed respectively with SLG and STG algorithm, which one represents best a *reference* time series. The reference, placed in the middle of the screen, shows the raw data using a line graph (no smoothing is applied to the graph). The two other graphs, are placed randomly above and below the reference (see Figure 6.7). The time series and the smoothing factor are chosen from a set of nine pre-defined configurations and we calculated an equivalent smoothing strength for both SLG and STG. Since we wanted to focus on the perception factor, we did not provide any interaction on the time series other

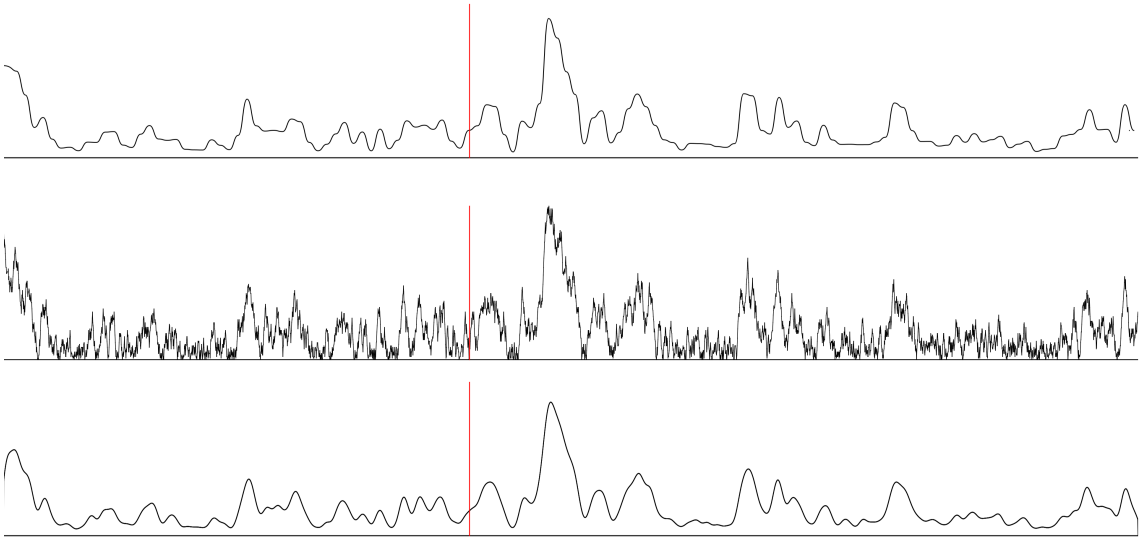


Figure 6.7: Task *Perception*. The graph in the middle is a line graph. Top and bottom graphs are either STG or SLG and their position is randomly swapped at each trial. The three graphs represent the same data. STG and SLG apply a smoothing that are equivalent. The red lines follow the mouse and are vertically aligned to help the comparison between graphs.

than a vertical cursor that follows the mouse cursor on each graph. Participants were told that the time is not important in this task.

Maximum task

In this task, conceived by Lam et al. [Lam et al., 2007], the participants had to find the maximal value on a graph, given a smoothing factor. We wanted to compare the smoothing quality of SLG and STG by measuring how far on the horizontal axis the smoothed maximum is from the real value. Participants were asked to be as precise as possible.

Period Task

We wanted to quantify how the visual accuracy impacts the user performance (in time and correctness) when evaluating the global shape of the data. Therefore, we have designed a variant of the *Slope* task introduced by Adrienko et al. [Andrienko and Andrienko, 2005] where the participant is asked to find the period of the data. We provided a continuous interaction using the mouse wheel to control the smoothing factor applied to the graph. To select the period perceived, the participant highlights the corresponding time window with a drag and validates using the keyboard. At the beginning of each trial, an arbitrary smoothing is applied to the graph.

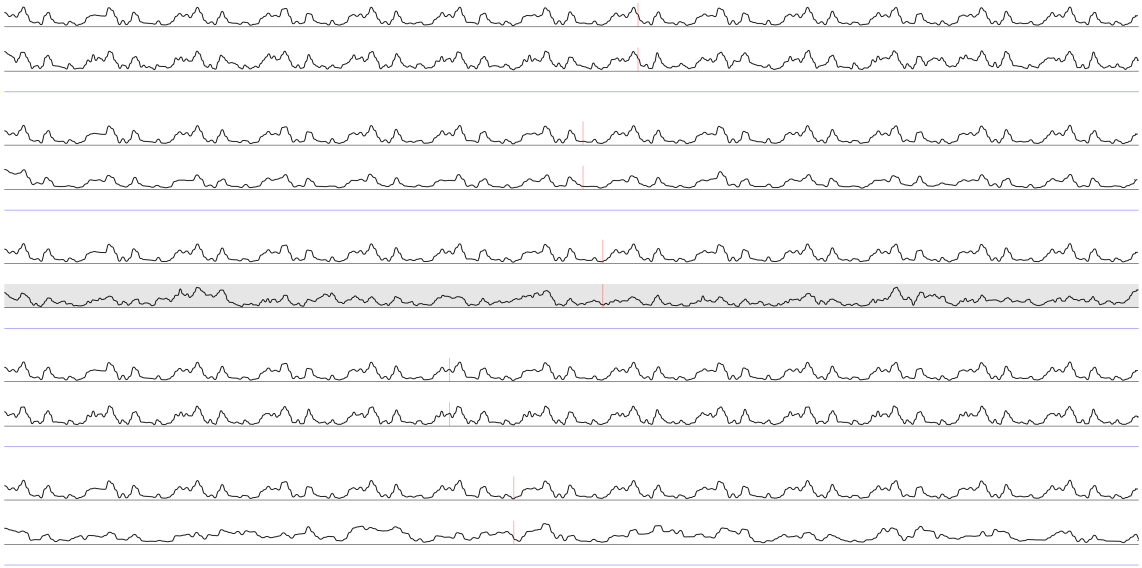


Figure 6.8: Task *Same*. Graphs are grouped in five blocks. In each block, the graph of the top is the reference, duplicated five times in total. Among the five other graphs, the participants had to find which represented the same data than the reference. The graph that was currently explored was highlighted.

Same Task

Five time series are displayed simultaneously and the participants have to select the exact same one than a reference time series. To generate the data, we added some noise to the reference. With this task, we wanted to measure time and correctness. Since we were not interested in performance of global comparison, the reference was duplicated below each time series to minimize the time and difficulty induced by doing a global comparison (see Figure 6.8). We provided the same interaction to control the smoothing factor than for the *Period* task. Participants select a time series with the mouse cursor and validate their choice with the keyboard as quickly as possible. This task was inspired by Perin et al.'s *Same* task [Perin et al., 2013] and is derivated from the *Slope* task introduced by Adrienko et al [Andrienko and Andrienko, 2005].

6.5.3 Participants

Eighteen participants were recruited, 13 males and 5 females. 9 came from our university, 9 were recruited outside. The average age of the participants was 29.5 (from 22 to 54, median 29). None of them had vision troubles to read on a screen and half had a Computer Science background. All of them were familiar with reading a line graph and one of them was familiar with other time series visualization techniques.

6.5.4 Experiment data

We used a synthetic dataset during the experiment. For each task, we generated a pool of three time series using a random walk. For the *Perception*, *Maximum* and *Period* tasks, we also chose three different smoothing factors: 6, 30 and 150 pixels wide for the time window in STG, multiplied by 0.43 to have the equivalent σ in SLG. These are the different smoothing conditions for the experiment. For the *Same* task, we added 35% of noise to the reference time series at each trial. Using a random walk algorithm allows us to have large scale and small scale variations in the dataset with properties to mimicking real world data such as financial or sensors data.

6.5.5 Protocol

We used a 24 inch LCD monitor display with a resolution of 3840×2160 pixels, a mouse to interact with the graphs and a keyboard to validate the answers using the *SPACE* key. The experiment was composed of two parts: a preliminary training phase, and evaluation phase.

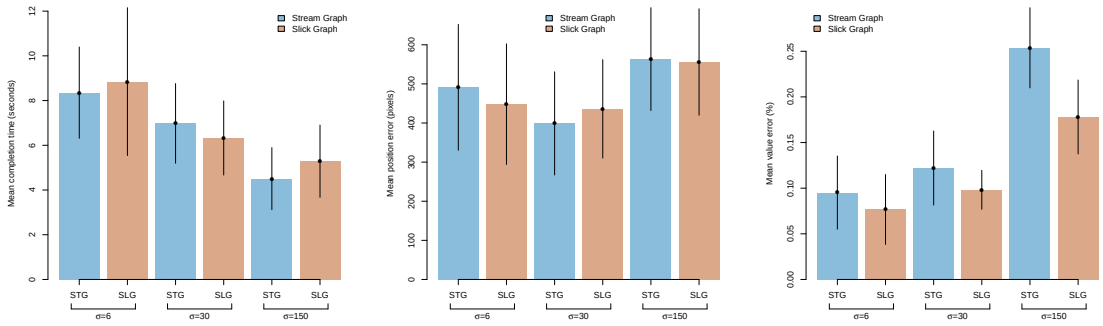
During the training phase, the participants were able to learn how to use the STG and the SLG smoothing techniques. They could control the smoothing factor using the mouse wheel. They were allowed to switch freely between the techniques and to review the time series being visualized. This phase was not limited in time and the participants were encouraged to ask questions.

The evaluation phase was divided into three blocks, one block for each task. At the beginning of each block, the instructor described the task and the participants could ask for clarifications. Then, they had a trial of training. After validating their answer, a blank screen was displayed. The next trial began when the participant pressed again the *SPACE* key. No feedback on the performance was provided during the whole experiment.

For the *Perception* task, the participants were asked to study the different graphs closely. They were told the time was not important and had 9 trials. The *Maximum* task was composed of 9 trials $\times 2$ techniques for a total of 18 trials. The participants were instructed to be as fast as possible. For the *Period* and *Same* tasks, of 9 trials $\times$ 2 techniques for a total of 18 trials each, the participants had to complete each trial within the shortest time. When all the tasks were finished, the instructor asked questions on how confident the participants felt about their answers during an informal discussion and collected feedbacks. The overall study included $9 + 18 \times 3 = 63$ trials and took an average time of forty minutes for each participant.

6.5.6 Results

We analyze the results of the experiment for each task.



(a) Completion time (b) Position error (c) Value error

Figure 6.9: Impact of the smoothing technique and of σ on the different dependent variables for the *Maximum* task. Error bars are 95 % CIs.

Perception

Two dependent variables were measured in this task: the time and the participant preference. None of them were significantly impacted by either the smoothing factor σ or the time series. Results show that the smallest the smoothing factor is, the most preferred SLG tends to be, but without statistical significance.

During the experiment, 44% of the participants reported that SLG seemed smoother than STG, thus STG was a more accurate representation of the raw data than SLG. They understood that the aliasing present in STG was details about the data: *"[SLG] looks the same but smoother"*, *"[SLG] is smoother, more progressive but it seems to erase details quicker"*, *"[SLG] is less precise in the beginning of the smoothing"*.

By the introduction of a high aliasing, STG can mislead a user who can interpret it as being real values in the data. This factor explains why the difference between SLG and STG is not significant for the preference variable.

It is hard to draw any conclusion regarding our hypotheses with this task.

Maximum

For the *Maximum* task, the dependent variables used for analysis are: the errors made by the participant, both in term of position (i.e. distance over the time axis to the time of occurrence of the actual maximum) and in term of magnitude (i.e. distance over the value axis to the actual maximum value); and the time taken by the participant to do the selection. The 3 factors manipulated are the time series, the smoothing factor, and the smoothing technique. Figure 6.9 summarizes the impact of the smoothing techniques and of σ of dependent variables.

The time is only affected by the smoothing factor ($F_{1,323} = 23.21$, $p < .0001$), and it decreases while the smoothing factor grows (see Figure 6.9a). This can be explained by the fact that the smoothing reduces the number of candidates for the

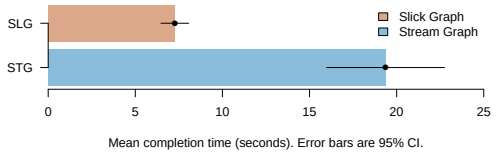


Figure 6.10: Mean completion time for the *Period* task.

maximum.

Neither the smoothing techniques nor the smoothing factor σ affect the position error (see Figure 6.9b). The position error is mainly affected by the time series. This is normal since each series has its own set of large picks that are candidate for the maximum. This dependent variable is thus not very pertinent to study the performance of SLG vs. STG.

Finally, the value error is affected both by the technique ($F_{1,323} = 11.15$, $p = .0009$) and the smoothing factor σ ($F_{1,323} = 74.03$, $p < .0001$) (see figure 6.9c). The technique also interacts with the other two factors to affect the value error, which make the interpretation of the results difficult. The value error actually grows with the smoothing factor: the linear fit gives a positive coefficient ($t_{322} = 8.08$, $p < .0001$). This is normal: the more the time series is smoothed, the less accurate the value of picks will be, since they are averaged with their surrounding, which are smaller by definition. The value error is also significantly larger when using STG than when using SLG ($F_{1,323} = 6.33$, $p < .0124$). A closer inspection shows that the difference between the two techniques varies depending on the time series. SLG always perform better than STG, but the amount of the difference, and its significance depends of the actual characteristics of the data.

This validates hypothesis H1: with SLG, users are more precise.

Period Task

For the *Period* task, the dependent variables used for analysis are: the time taken by the participant to select a period; and the ratio between the duration of this period and the actual period present in the data (i.e. selecting the real period will lead to a ratio of 1, whereas selecting a period that covers 2 actual periods will lead to a ratio of 2). Among the 3 factors manipulated (time series, smoothing factor, smoothing technique), only the technique has a statistically significant impact on the dependent variables.

The technique impacts significantly time ($F_{1,323} = 63.55$, $p < .0001$): the average time is 7.29 s for SLG and 22.22 s for STG. SLG is thus basically used 3 times faster than STG. Those averages, of course, differ significantly ($t_{322} = 7.97$, $p < .0001$). Figure 6.10 shows their 95 % confidence intervals.

The correctness of the period detection (i.e. how close the ratio is from 1) is also significantly impacted by the technique ($F_{1,323} = 24.68$, $p < .0001$). The average

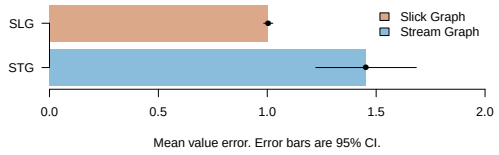


Figure 6.11: Mean value error for the *Period* task.

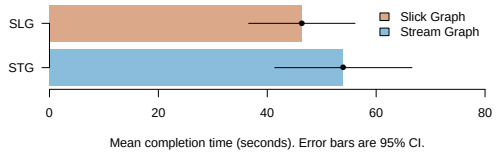


Figure 6.12: Mean completion time for the *Same* task.

ratio is 1.026 for SLG and 1.684 for STG. Their respective 95 % confidence intervals are displayed in Figure 6.11. A closer look at the distributions of the ratio shows that for SLG, most of the values (160/162) are very close to 1, while the two remaining values are close to 2; whereas for STG, 118/162 values are centered on 1 (i.e. within the $[.5, 1.5]$ range), 24/162 are centered on 2, 12/162 are centered on 3, while the remaining 8 ratios are above.

This shows that the legibility problems of STG described in Section 6.3 really affect the performance of STG vs. SLG for the *Period* task since SLG is both significantly faster and accurate. For this task, H1 and H2 are valid: users are more precise and manage to do their judgment faster with SLG.

Same

The dependent variables analyzed for this task are the completion time and the correctness.

The time is significantly impacted by the technique ($F_{1,323} = 4.22$, $p = .0408$). The average completion time is 46.35 s for SLG and 53.98 s for STG with a 95% confidence of $[41.92, 50.78]$ and $[48.14, 59.83]$ for respectively SLG and STG (see Figure 6.12).

The correctness is not significantly impacted by the technique ($F_{1,323} = 2.34$, $p < .13$). However, SLG is more correct on average: correctness is 70.37% for SLG with a 95% confidence interval of $[63.07\%, 77.67\%]$ and 62.35% for STG with a 95% confidence interval of $[55.05\%, 69.65\%]$ (see Figure 6.13).

This confirms our hypothesis H2: the users are faster with SLG when the task requires to explore the data. Visual discontinuity of the STG graph slows down the user.

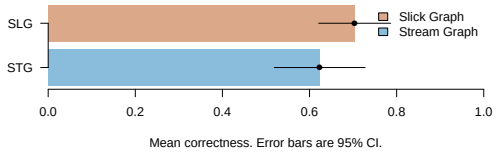


Figure 6.13: Mean correctness for the *Same* task.

6.5.7 Discussion

The results show that SLG clearly outperforms STG in terms of both accuracy and completion time when working with periodic data, making our technique particularly suited for visualizing electronic signals, execution traces, etc. Indeed, we explained how STG smoothing technique can show an incorrect period or totally hide the data periodicity. With SLG, the users also read more accurately the extrema (max task) while they tend to prefer the STG smoothed silhouette. With these insights, we strongly recommend not using STG binning and smoothing technique with data having a high variability, either small or large variation (i.e typically financial time series, sensors data, traces, logs, etc.). STG hides or moves the local extrema and introduce visual artifacts that the users interpret as being the correct representation of the data.

The user evaluation also shows that the visual discontinuity that appear with STG when changing the smoothing factor slows down the users: it breaks its cognitive model. Therefore, we also discourage the usage of STG for interactive visualization, independently of the property of the data visualized. STG is usable for casual non-interactive visualizations for time series that follow a Poisson law such as the number of tweets during an event, the box office entries, etc.

6.6 Integration with Existing Techniques

In this section, we used a Gaussian kernel for the smoothing pass. Hence, we refer to the kernel width w as σ.

6.6.1 Stacked Graph

We applied the SLG algorithm to Stacked Graphs. We chose to compute the overall shape of the graph with the original ThemeRiver algorithm (the baseline g_0 is: $g_0 = -\frac{1}{2}\sum_{i=1}^{n} f_i$ with f_i being the layers. See Byron et al. work for more details [Byron and Wattenberg, 2008]).

While correcting the drawbacks of STG, integrating SLG with Stacked Graphs has several other advantages. Using the SLG binning technique, the peaks are more precisely located: the shifting effect due to the aliasing is minimized to the pixel,

(a) (b)

Figure 6.14: Data smoothed with (a) STG algorithm, and (b) SLG algorithm. SLG reveal more details.

making extrema more accurately positioned and adjusting the smoothing factor for the interactive exploration of high and low frequencies enable the user to visualize data with more details.

As an example, we took a dataset of tweets during the State of Union Address 2015 presented by the president of the USA to the US citizen representation. The dataset was built by the Twitter visualization team and they implemented an interactive visualization with it[1]. Figure 6.16 shows different rendering of Stacked Graph using different smoothing algorithms. Figure 6.16a is rendered using the original ThemeRiver algorithm[1]. SLG algorithm was used in Figures 6.16b and 6.16d with different smoothing factors σ.

Using the SLG algorithm, the visualization can be more detailed. For instance, Figure 6.14a shows a peak on the *#jobs* layer as seen in Stacked Graph. Figure 6.14b is the same data being represented but the visualization has been computed using the SLG algorithm. The peak is composed of two small peaks that correspond to the beginning of two strong paragraphs: *"21st century businesses, including small businesses need to sell more American products overseas"* and *"21st century businesses will rely on American science, technology, research and development"*[1].

With a small smoothing factor, graph becomes very precise and local extrema can be easily spotted. On Figure 6.16b, a narrow peak appears on the right (zoomed on Figure 6.15), and corresponds to the end of the talk. This is an illustration of the accuracy the SLG algorithm brings: this peak is not visible on the original Stacked Graph visualization and this information is lost.

6.6.2 Interactive Horizon Graph

Horizon Graph (HG) is a visualization technique for multiple time series [Saito et al., 2005; Reijner, 2008; Few, 2008] that belongs to the *split-screen* techniques. HG have been designed to virtually augment the vertical resolution of the graph by dividing it in bands and wrapping them around a baseline. Perin et al. brought interactions to HG and introduced Interactive Horizon Graph (IHG) [Perin et al., 2013]. They integrated two interactions: baseline panning and value zooming. When panning the baseline, values on the graph moves up and down relatively to the position of

[1] *#SOTU2015: See the State of the Union address minute by minute on Twitter*, ¡`http://twitter.github.io/interactive/sotu2015`¿, accessed on March 31st 2015.

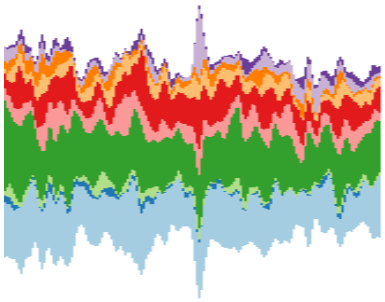


Figure 6.15: Narrow peak corresponds to a sudden number of tweets being emitted at the end of the talk.

the baseline and their color changes according to their current band. Zooming on values corresponds to increasing the number of bands, thus the virtual resolution of the graph increases.

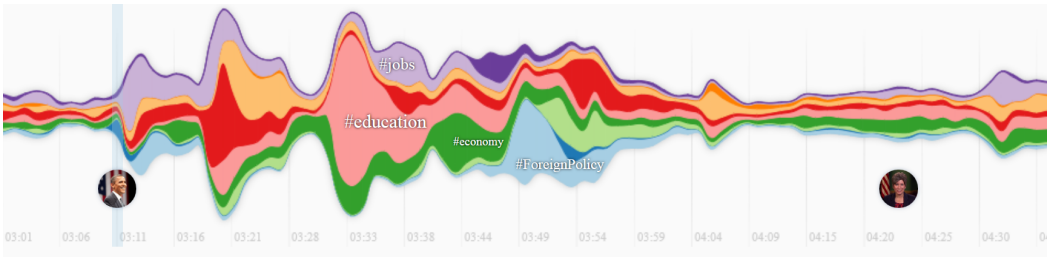
When using the zoom, the aspect ratio of the graph virtually decreases, thus increases the average slope. This effect has a huge impact on the legibility of the graph. To demonstrate this, let us consider the case when working with a time series that variates at a high frequency rate and suppose we want to explore the low frequency behavior to find the general slope of the graph. Figure 6.17a shows an example of such graph being represented with IHG at an initial configuration with an arbitrary baseline position and no zoom on the values.

Using interactions provided by IHG, it is possible to progressively increase the zoom factor to compare more easily the local extrema. Figures 6.17a, 6.17b and 6.17c show different zoom values. The graph quickly become very hard to read due to a large number of peaks appearing as many red and blue narrow bands.

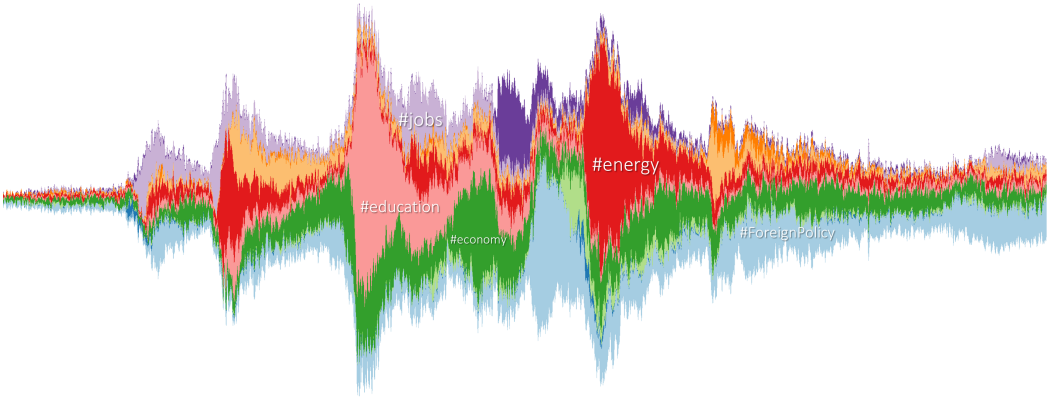
To correct this problem, we propose to use the SLG smoothing algorithm with IHG. On IHG, the left and right mouse buttons are used to control respectively the value zooming and the baseline panning. To keep the interaction centralized on the mouse, we used the mouse wheel to control the smoothing factor σ. In Figures 6.17c to 6.17e, σ is progressively increased. The overall graph becomes easier to read, the extrema can be more easily compared and the general slope of the graph is encoded in the light shades. For a more coherent visualization, we also updated the computation of the range r_i of each band b_i:

$$r_i = [(y_b + i\frac{h}{2K})\sigma, (y_b + (i+1)\frac{h}{2K})\sigma]$$
$$\text{with: } \begin{cases} i \in [-\lceil z \rceil, \lceil z \rceil[\\ h = max(|y_b - y_m|, |y_b - y_M|) \\ K = z \end{cases} .$$

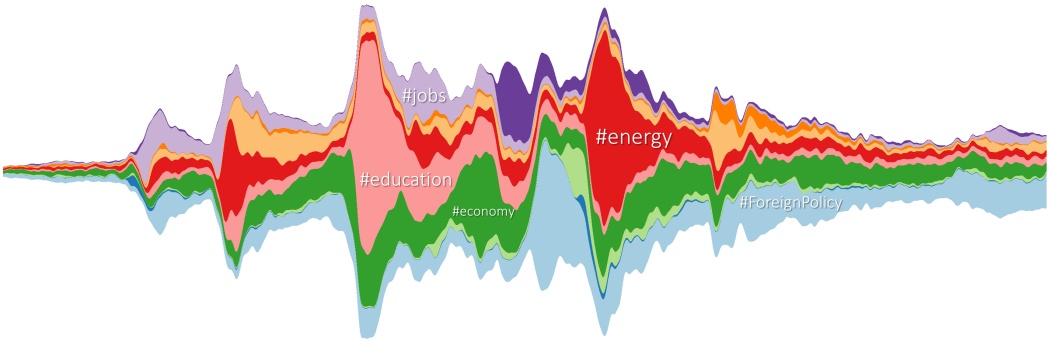
When increasing the smoothing factor, the values of the graph decreases and the highest band might not be used. Thus, we integrated σ into the computation of the range of the bands so that it adapts correctly and all the bands are reached.



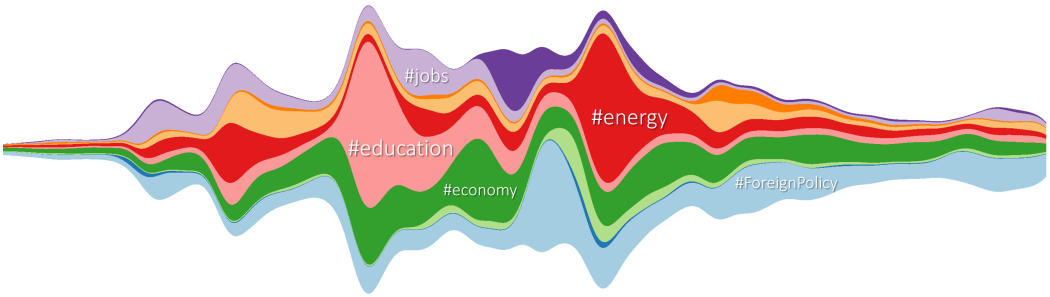
(a) Stream Graph computed with original smoothing algorithm.



(b) Stream Graph computed with SLG algorithm ($\sigma = 1$ pixel).

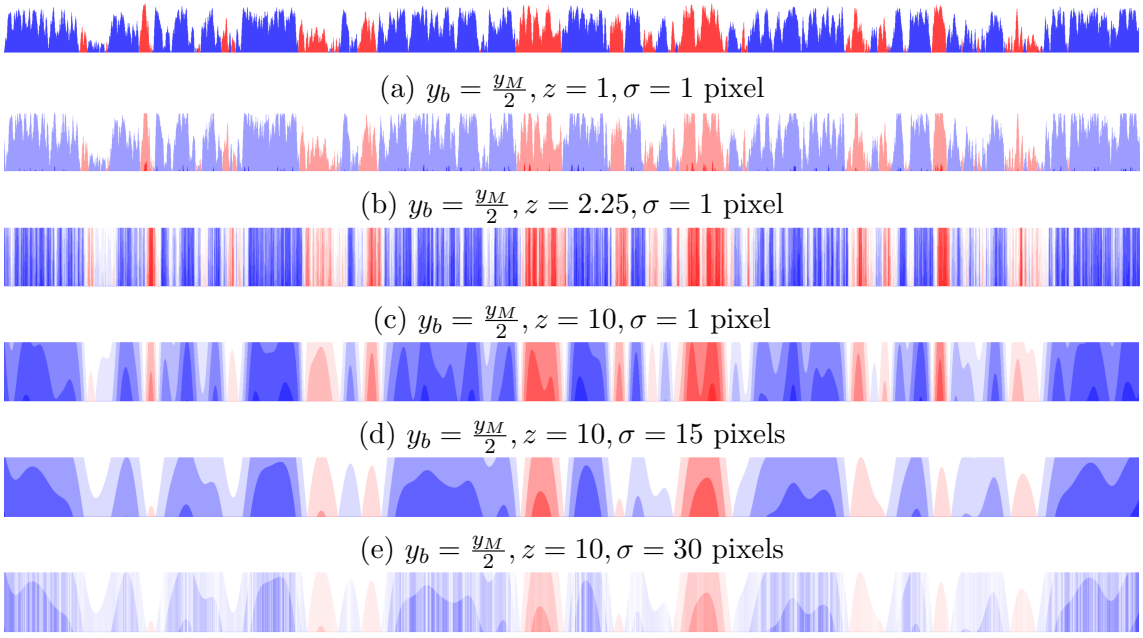


(c) Stream Graph computed with SLG algorithm ($\sigma = 6$ pixels).



(d) Stream Graph computed with SLG algorithm ($\sigma = 30$ pixels).

Figure 6.16: Stream Graphs visualizing the volume of tweets emitted during the State of the 2015 Union Address Twitter dataset using different smoothing algorithm: (a) Stream Graph[1], (b, c, d) Stream Graph computed with SLG algorithm ($\sigma = 1$, $\sigma = 6$ and $\sigma = 30$).



(a) $y_b = \frac{y_M}{2}, z = 1, \sigma = 1$ pixel

(b) $y_b = \frac{y_M}{2}, z = 2.25, \sigma = 1$ pixel

(c) $y_b = \frac{y_M}{2}, z = 10, \sigma = 1$ pixel

(d) $y_b = \frac{y_M}{2}, z = 10, \sigma = 15$ pixels

(e) $y_b = \frac{y_M}{2}, z = 10, \sigma = 30$ pixels

(f) $y_b = \frac{y_M}{2}, z = 10, \sigma = 30$ pixels, SLG bands

Figure 6.17: Impact of the smoothing factor σ on IHG. The time series becomes very difficult to read when the zoom increases. Increasing σ details are filtered out and the average angle of the slope decreases, making the graph more legible.

6.7 Conclusion

In this chapter, we proposed a solution to the research problem of representing accurately a time series (see Section 4.1).

In Section 6.3, we described the problems introduced by the different smoothing strategies, the visual artifacts that can appear due to the binning and highlighted the consequences on the visual rendering of a time series. In Section 6.4, we have introduced Slick Graphs, a novel visualization technique for time series that provides an accurate binning and smoothing technique that minimizes the aliasing and visual artifacts to the pixel. The starting point of the binning process are the pixels instead of the data. This pixel oriented process guarantees that the data are aggregated on the correct pixel, thus the peaks are visualized accurately. The histogram resulting of the binning process is then smoothing using a kernel convolution where the kernel is centered on the bins of the histogram, thus the pixels. It ensures that the final rendering provides the best precision achievable with actual screen technologies, where the aliasing effects are mitigated to the pixel. Filtered-out information is encoded in level of gray for each pixel making the local maxima appear as black bands on the graph and local minima as white bands.

We have demonstrated that the SLG smoothing technique produces a shape of

the graph that allows users to be faster and more accurate than using the broadly used STG algorithm. We have also shown examples of how to integrate SLG smoothing with existing visualization techniques for time series (Interactive Horizon Graphs and Stacked Graphs) and explain how the visual quality is improved.

Slick Graphs are a suitable visualization technique for an integration into an industrial environment that requires a high level of precision. We have already presented an example of how SLG can support the developers to detect a periodic behavior if the kernel used for the convolution is the Gaussian kernel (Section 6.4.5). Under some conditions, with an inaccurate binning strategy, the periodic patterns can be hidden. Using SLG binning and aggregation algorithm as the basis for other visualizations ensures that the aliasing is reduced to the pixel, thus the perceived periodic behaviors are correctly rendered.

In the next chapter, we present a visualization framework for execution traces that is built upon the SLG binning and aggregating algorithm and that integrates a Slick Graph in one of its view.

Chapter 7

TraceViz

Contents

7.1 Introduction 91
7.2 Data 92
7.3 TraceViz Design 96
7.4 TraceViz 98
7.5 Industrial Use Cases 102
7.6 Industrial Deployment 106
7.7 Conclusion 110

7.1 Introduction

In related work, we described in Section 3.2 that a gap exists between the tools proposing an overview of the trace and the ones providing too much details. We explained that the first ones do not allow to start the analysis of the trace correctly as the data are too aggregated. On the other side, the second ones make the navigation fastidious due to a large amount of events and the quantity of details rendered. They also make difficult the possibility to spot behavioral patterns that implies a large set of events: the user simply cannot remember sequences of events easily. Instead, the visualization has to show him such information.

We also explained that the actual limiting factor to provide an interactive browsing of large traces is the lack of an efficient data access despite the democratization of SSD.

In this chapter, we address the two following questions, stated in Section 4.2 and 4.3:

1. *Which visualization techniques to develop for a tool that provides enough information about the behavior of the application to begin the investigation, yet high-level enough to be efficient at exploring the data?*

2. *How to develop a back-end solution suitable for the interactive exploration of time series containing millions of events?*

In this chapter, we introduce a novel visualization framework called TraceViz, to tackle these two questions. TraceViz allows the developers to explore interactively huge execution traces from high-level using aggregation techniques all the way down to a single event. It makes possible to visually spot regular patterns and trends in the trace, guiding the analyst in the filtering process. Enabling the software developers to quickly filter parts of trace, whether it is temporal windows or several actors, is critical to accelerate the discovery of the reason of a bug. An execution trace can easily contains hundreds of actors and several millions of events. When a problem occurs during the decoding, most of the time, it implies a very small number of processes in specific temporal windows. Thus, efficiently filtering repetitive part of the trace and irrelevant actors can reduce drastically the number of events to investigate and shortening the time required for the trace analysis.

We based the visualization part of this work on the techniques introduced in the previous chapter with Slick Graphs. The binning and aggregating algorithm is used in the two different views developed in this work. One of them integrates a Slick Graph to provide a global overview of the trace. The SLG algorithm ensures that the visualized patterns on the time series are accurate and that the developers are not investigating what seems to be a bug but is indeed an artifact.

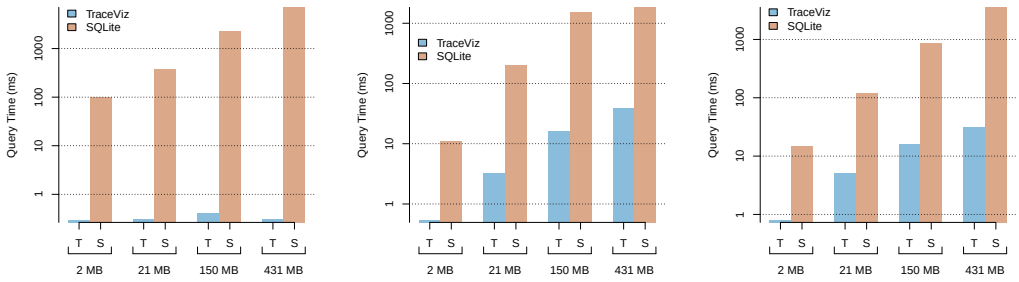
The remaining of this chapter is organized as follows. We begin with a description the underlying back-end we have implemented for supporting interactive exploration in TraceViz (Section 7.2). We continue by detailing its design principles and goals (Section 7.3). Next, we describe the graphical interface and the user interactions to efficiently explore the trace (Section 7.4). We finish by describing two industrial use cases where TraceViz allows to identify patterns and was used to find bugs (Section 7.5) and how we have integrated TraceViz into the STMicrolectronics debuggin toolkit (Section 7.6).

7.2 Data

In this section, we describe the back-end developed for TraceViz and present its performance measured with an experiment. We finish by detailing the statistics implemented in TraceViz.

7.2.1 Data Storage

Execution traces contain huge amount of data. They are composed of a large series of events. Each event has a timestamp at which it occurred, the actor which produced it, typically processes and interrupts, and a type. An event type can be an



(a) Time Window (b) Actor (c) Event Type

Figure 7.1: Read time of 20000 events when filtering on the time window, the actor and the event type.

entry/exit of a system call, an application function or an interrupt, a context switch, etc. As seen previously, developers use an incremental workflow that involves many tools where analysis of execution traces is part of it. In this context, traces need to be stored in a efficient back-end that guarantees a minimal time for data access and processing. Traditional tools use a SQLite database [Prada-Rojas et al., 2009; Pagano et al., 2013]. With recent hardware platforms and increased trace size, such database does not scale and developers experience slow data access. Deploying complex architecture with powerful severs is also prohibited since it requires streaming data over the network and remains too complex to achieve in the context of streaming multimedia decoding applications on MPSoC. To address those constraints, we have developed a back-end based on HDF5, *Hierarchical Data Format* [Folk et al., 2011]. HDF5 allows to store huge files in a hierarchical format and comes with powerful memory management for fast access to huge amount of data. Supported by the the HDF Group[1], HDF5 is widely used in scientific applications where high performance and robustness is necessary like in meteorology[2].

TraceViz stores a trace as follows:

- `/events` contains an array of all the events in the trace, chronologically sorted,
- `/actors` is an array of all the active actors,
- `/types` stores all the event types.

Running through the whole trace is done by accessing the array of events. The HDF5 driver handles the main memory and the page faults, providing a high performance data access.

We study the TraceViz back-end performance for importing, reading and querying a trace. We compare the results with an SQLite back-end largely used by analysis

[1]The HDF Group, ¡`https://www.hdfgroup.org`¿

[2]NASA scientific satellite Terra, ¡`http://terra.nasa.gov`¿

tools for execution traces. We ran the experiment on a workstation equipped with an quad-core i7 Intel CPU at 3 GHz, 16 GB of RAM and a 256 GB SSD. The design of our experiment is largely inspired by the evaluation of the FrameSoC back-end performance conducted by Pagano et al. [Pagano et al., 2013].

Trace size (MB)	2	21	150	431
TraceViz (s)	0.549	2.146	15.024	30.439
SQLite (s)	0.648	6.196	55.769	84.659

Table 7.1: Importation time for different trace size

Importation Performance Developers are used to analyze the traces with textual tools where the parsing time is hidden. A primordial tool adoption criterion is to have a minimal preliminary parsing phase. We measured the importation time for traces that range from 2 MB to 430 MB (20×10^3 to 4×10^6 events) (see Table 7.1). The importation time keeps short (30 seconds) for huge traces and linearly increases with the file size; a linear regression shows that the coefficient of determination R^2 is equal to 0.98. This result proves that the importation time in TraceViz adds few overheads compared to working with trace files in text format where there is no importation and provides much better performance than SQLite.

Trace size (MB)	2	21	150	431
TraceViz BR (ms)	0.416	0.385	0.383	0.376
TraceViz RR (ms)	0.369	0.301	0.564	0.616
SQLite (ms)	62.09	312.77	1822.95	5800.63

Table 7.2: Read time of 10000 events for different trace size. The first row reports the time to read a block of 10000 consecutive events (BR time). The second row reports the time to read 10000 events randomly chosen in the trace (RR time). The third row is the time to read a block of 10000 events in SQLite.

Reading Performance Since a long time, it is known that a system feels interactive to the users for a response time inferior to 100 milliseconds. For delay superior to 1 second, the users' cognitive model is broken and the system loses users' attention for delay superior to 10 seconds [Miller, 1968; Card et al., 1991].

TraceViz aims to provide interactive exploring and filtering techniques for traces. It largely depends on the back-end performance which has to respond in a delay inferior to 100 milliseconds independently on the query. We measured the response time of the data storage for both reading (Table 7.2) and querying operations to study if this requirement is fulfilled.

We measured the reading time in traces of different sizes under two condition. Firstly, we read blocks 10000 consecutive events in a randomly chosen part of the trace. We repeated 10 times this step and compute the average time (BR time in Table 7.2). SQLite performance has been measured under these conditions. Secondly, to minimize the impact of the cache effect of HDF5 and to simulate the result of complex queries we also measured the reading time of 10000 non-consecutive events randomly chosen on the whole trace. (RR time in Table 7.2). We notice a slight increase for bigger traces but the response time is still under the millisecond.

For both BR and RR measurements, the response time is constant at below 100 milliseconds. For SQLite, the response time grows linearly and shows it cannot provide interactive read time. The performance of the TraceViz back-end allows to browse the trace interactively.

Query Performance To better measure the back-end performance for filtering tasks, we measured the time to read 20000 events in the result of a query on a time window, an actor and an event type. The results are presented in Figure 7.1. Querying a time window is constant in time (Figure 7.1a). This comes from the format used to store the events: they are naturally sorted by their timestamp and indexed by their location in the array, allowing to use fast search algorithms. This shows that the back-end can support interactive pan and zoom. The query performance on the actor and the event type are similar. Both of the query time increase linearly with the trace size (Figure 7.1b and 7.1c), shown by a coefficient of determination respectively equal to 0.99 and 0.97 for the actors and the event types. The response time remains lower than a second under all the conditions, guaranteeing the users' cognitive model remains unbroken.

Conclusion on Back-end Performance The results of the different benchmarks shows that the back-end provides performance suitable for usage in an interactive context. It guarantees an interactive response time for the exploration of a trace and returns the result of a query in a time short enough so that it does not interfere with the users' understanding.

7.2.2 Statistics and Data Computation

Execution traces are a list of raw low-level events from which different metrics can be computed according to the goal of the developers. During our collaboration with the software developers at STMicroelectronics, we noticed that the analysis mainly involves three metrics: the *event density*, the *activity time* and the *delay between events*.

The event density describes the event distribution over time. Using this statistic, the developers can spot an abnormal number of interrupts, system calls or function calls in the application.

The activity time gives insight on the task scheduling on the CPU. For example,

the analysts can check if a task has been executed for a abnormally long period blocking other processes. This can result in the violation of QoS constraints [Iegorov et al., 2015].

The delay between events allows the checking of QoS constraints more accurately. Using their domain knowledge, the developers know which calls or interrupts are critical in the decoding process and can check their call frequency. As an example, the video decoder has to decode 25 frames per second to avoid glitches or blanks. Checking the call frequency of the function starting the decoding of a frame is a simple way to approximate the frame rate before further checking.

In TraceViz, the developers can interactively switch between these statistics. Each statistic is computed separately for every actor present in the execution trace. While it already gives meaningful low-level insights, it is sometimes relevant to perform the checking at higher level of abstraction, requiring to aggregate several actors. The developers may need to check at a component level in charge in a particular step of the decoding process. To do so, it is necessary to aggregate all the actors of this component which can include tasks and interrupts. TraceViz provides simple user interactions to create such aggregates before computing a chosen statistic on its data.

7.3 TraceViz Design

Based on the related work and our observations in STMicroelectronics, we propose TraceViz, a new interactive visualization tool for execution traces.

7.3.1 Design Rationale

- *Provide an overview of the trace.* Most of the time, developers begin to visualize the global behavior of the system during the execution. TraceViz has to provide an easily understandable overview, yet with enough details to begin the filtering process.

- *Support domain related statistics.* When debugging, developers use well-established statistics. It is primordial to integrate them into the tool to maximize the semantic of the representation.

- *Integrate well-known visualization techniques.* To mitigate the learning curve, we decided to use visualizations for time series based on line graphs, the most widely used time series representation.

- *Provide user interactions to explore and filter the data.* For an efficient browsing, TraceViz has to support interactive zooming, sorting, aggregating and filtering.

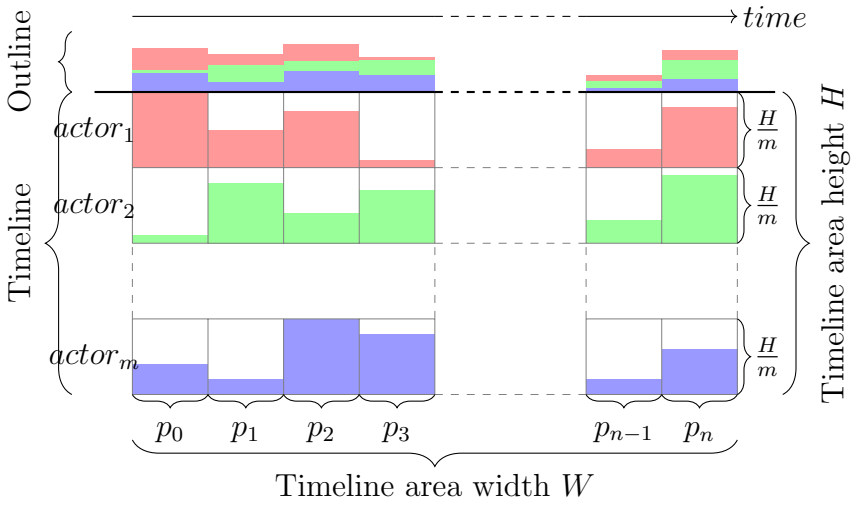


Figure 7.2: TraceViz visualization principles.

- *Visualize behavioral patterns between actors.* By nature, a streaming application repeats the same operations on a regular period. Understanding which actors are synchronized as well as visualizing the patterns will help the analyst to quickly spot trends and abnormal behaviors without using complex algorithms.

7.3.2 TraceViz Visualization Principles

Javed et al. stated that the users perform better for global tasks using *split-screen* techniques and are more efficient for local tasks with *shared-screen* visualizations [Javed et al., 2010]. TraceViz mixes both to easily visualize overall behavior and make local comparison between actors: it embeds a timeline view and an outline view that share the same time axis (Figure 7.2).

The timeline area relies on the principle of small multiples [Tufte, 1986] and belongs to the *split-screen* techniques. The goal of the timeline area is to visualize the macro-behavior of each actor such as its periodicity or a particular behavioral pattern. It also serves to represent the synchronization between different actors and to spot potential patterns at component-level of the application. For m actors in a trace T, m graphs are represented in the timeline view, one graph per actor (see Figure 7.2). The vertical resolution H (in pixel) is splitted into m horizontal areas of height $\frac{H}{m}$ pixels, where the graphs are rendered. The horizontal resolution W (in pixels) gives the number of time slices to use to segment the trace. By doing so, the data is aliased at pixel-level, the smallest aliasing achievable on current display technologies. For each actor a, we compute an histogram $hist_a$ of W bins, each bin corresponding to a pixel (see Figure 7.3). For each pixel p_i, we compute the two timestamps t_{i1} and t_{i2} at its boundaries and extract data contained in this time window. We have $V_a = T_a[t_{i1} \dots t_{i2}]$ with T_a being all the events produced by the

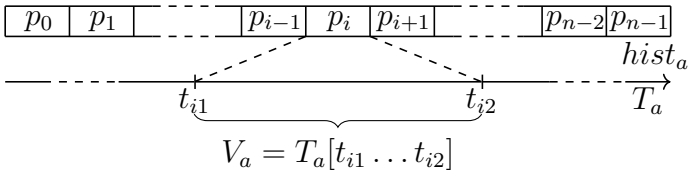


Figure 7.3: Building of the histogram $hist_a$ for an actor a

actor a in the trace T. Next, we compute $hist_a[p_i]$ so that:

$$hist_a[p_i] = f(t_{i1}, t_{i2}, V_{a_i})$$

with f being the statistics chosen by the analyst (*event density*, *activity time* or *delay between events*).

The outline area provides a more general overview of the execution to spot local peaks of activity on the system, hard to visualize on the timeline since the information is spread over m graphs. Instead of being juxtaposed, the m graphs are stacked so that the value at pixel p_i is:

$$hist_{outline}[p_i] = \sum_{a=actor_1}^{actor_m} hist_a[p_i]$$

The integration of the timeline and the outline views provides to the analyst a global overview of the execution, yet with details on the actors while showing temporal patterns. It combines the advantages of the existing high and low-level tools. Using the already established statistics as basis for the computation of the histograms minimizes the learning phase and ensures a good readability.

7.4 TraceViz

In this section, we present the user interface of TraceViz with its components and describe the different interactions implemented.

7.4.1 Layout

The TraceViz interface, shown in Figure 7.4, consists of three main areas: the tree view, the outline view and the timeline view.

The tree view shows the actors present in the trace initially ordered as a hierarchy according their nature (hardware interrupts, software interrupts and tasks). To visually make corresponding a graph and its label, links are placed between the tree view and the timeline. Their visibility is automatically updated according to the tree label to leverage the visual clutterness. Each category of the hierarchy has its own color, reported on the links.

Figure 7.4: Overview of TraceViz. TraceViz interface consists of three main areas: the tree view (a), the outline view (b), the timeline view (c) and the links that connect the actors and their corresponding graphs (d).

The outline view in the top area represents the overall activity of the system. The details of actors statistic are represented using colors, each color encoding one actor. The analyst can choose to switch to the SLG shading for a more precise frequential analysis.

The timeline view visualizes the time series corresponding to the actors. Their order is given by the hierarchy. All the graphs and the outline view have the same time axis and are aligned on the timestamp of the first event of the trace by default. At the start up, the timeline shows the whole trace: it begins on the left at the timestamp of the first event of the trace and finishes on the right at the timestamp of the last event.

The tool bar gives access to different configuration settings of the view such as defining initial filter parameters, the statistic being used to compute the data and the state of the visual functionalities such as the smoothing factor σ applied on the outline view.

7.4.2 Initial View Configuration

At the beginning of the analytic process, a window appears to set up the initial view (see Figure 7.5). The developer has to choose a statistic to compute the input data used to feed the graphs in the timeline and outline views. The statistics implemented are *event density*, *activity time* and *delay between events*, as described in section 4.2. To improve the clarity of the timeline view and to increase the speed of the filtering process, the developer can select which actors are hidden in the initial configuration

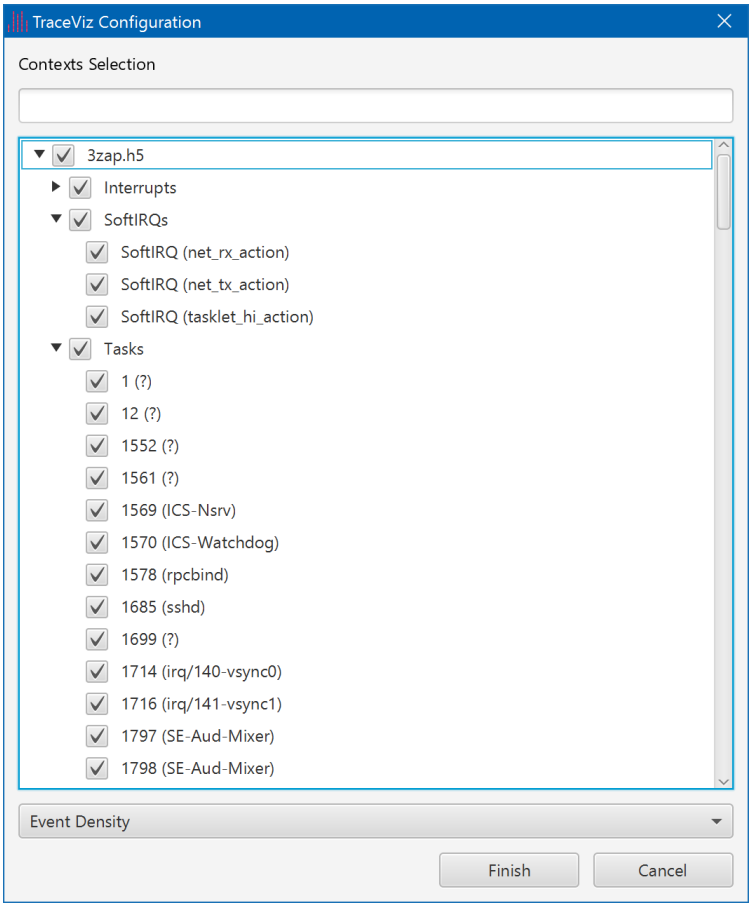


Figure 7.5: Initial View Configuration of TraceViz where the developer can filter the actors to show or hide and which statistics to start with.

and which trace points to ignore. By doing so, the view will directly display the data of interest. This filtering process is also quicker than navigating the tree hierarchy.

7.4.3 Trace Exploration

The hierarchy part of the view provides all the interactions to navigate, reorder, hide, create groups and to aggregate elements. The timeline area provides all the interactions to explore the data. Hovering the timeline area with the mouse cursor updates the hierarchy area. When hovering a graph, the hierarchy automatically scrolls to align the corresponding tree label and highlights it. When scrolling out, the actors disappear but the labels and connection of the upper levels in the hierarchy remain visible by stacking on the top and bottom. By doing so, the developer always has visible indicators that show the vertical position of the tasks and the interrupts in the timeline helping the exploration for large hierarchy sizes. When the graph of an actor is highlighted, the value under the current pixel is noted in the time cursor. Its corresponding layer in the outline and its label in the hierarchy are focused.

7.4.4 Pan and zoom

Initially, the whole trace is displayed using filters defined in the configuration settings. Depending on the trace size and duration, a pixel can encode a large time window, making apparent high-level recurrent patterns in the behavior of the different actors. After having visually detected those patterns, the developer may be interested to filter-out redundant data to focus on one of those patterns. We provide a drag interaction using the right button to select a time window of interest that will fit the view. It is also possible to continuously zoom in the trace using the mouse wheel.

A drag interaction using the left button allows to pan in the trace. While using those interactions, both the timeline and the outline views are refreshed to provide a continuous feedback and to keep a consistent visualization.

The pan and zoom interactions provide a natural way to explore behavioral patterns that can appear at high and low frequencies.

7.4.5 Actor Selection and Aggregation

When hovering the hierarchy, it behaves like a classical tree view with standard interactions: *left-click* to hide a label and its graph, *shift/mouse move* to select several consecutive labels, *control/left-click* to select non-consecutive labels, and *right-click* to access to different actions such as grouping, aggregating and hiding tree labels and their graphs.

After having selected actors, the user can create a group and name it. From this point, different actions are possible when doing a *right-click* on the group's tree label: deleting the group, hiding it or aggregating it. When a group becomes hidden, all the graphs of its children are also hidden on the timeline area. The remaining graphs spans vertically, increasing the vertical resolution. To aggregate a group, several operators are available: maximum, minimum, average and median. This operator is used on each pixel to compute the resulting graphs of the aggregation. From this point, the group behaves as any actor in the hierarchy.

7.4.6 Hierarchy Reordering

Besides execution patterns, actors' graphs can reveal similar periodic behaviors. To better compare their graphs, it is possible to place them side-by-side using a drag interaction on their corresponding tree labels. While dragging, the hierarchy stops scrolling automatically to ease the interaction and a visual feedback is displayed as a blue line to indicate where the actor would be moved when the interaction is over. The layer in the outline view are reordered according to the new position. TraceViz provides the possibility to drag an actor, a group and an arbitrary selection of actors.

7.4.7 Implementation

TraceViz is implemented in Java 8 with JavaFX 8 as the interface toolkit. As mentioned in section 4.1, HDF5 is used for the back-end and the data extraction and filtering is implemented using the Stream API of the JDK 1.8.

An operational version of TraceViz has been deployed for the developers at STMicroelectronics. This version implements fewer interactions than presented in this chapter as the back-end is an SQLite database for compatibility reasons. It has been implemented as an Eclipse plug-in and is internally shipped as a tool of the SoC Traces & Profiling Toolkit (STPTK)[3]. Part of the interface relies on the SWT toolkit while the rendering still relies on JavaFX 8.

7.5 Industrial Use Cases

In this section, we present two use cases that happened in industrial environment at STMicroelectronics.

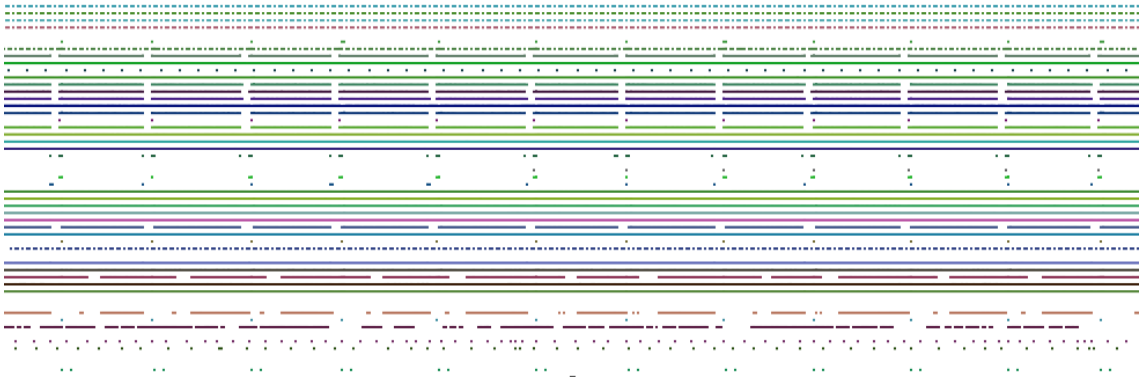
We will see in the next section that we have integrated TraceViz into the STMicroelectronics toolkit to analyze execution traces. Therefore, the software engineers have discovered TraceViz by themselves, reading the embedded documentation and have used it to uncover some bugs in traces. None of the developers have been in contact with a member of our tools development team to learn how to use TraceViz.

When STMicroelectronics sells a product for set-top box, it integrates the hardware, the software and also provides a support for the developers of the customer company. The software delivered by STMicroelectronics is composed of the operating system (STLinux), the multimedia layer in charge of the decoding and encoding (Streaming Engine) and an API to access to the Streaming Engine. After having received the board, the customers implement their own layer to provide a user-friendly experience to the end-user and integrates their own features. During this development phase, it is very common that bugs appear on the multimedia layer.

In this case, the customers contact the STMicroelectronics developers through a bug tracker where they describe precisely their problem. A collaboration begins between the two parts and eventually finishes when the problem has been solved. At the end, STMicroelectronics provides a detailed explanation of the root of the problem to the customer and provides explanations to solve it. In between, a discussion between the STMicroelectronics developers take place where they work collaboratively on the bug. When relevant, they join an execution trace to the bug tracker.

We explored this bug tracker and have extracted the use cases explained in this section from it. Hence, these use cases have been real problems. We have chosen those use cases because it demonstrates how TraceViz has been used by software

[3]SoC Traces & Profiling ToolKit (STPTK), ¡`http://www.stlinux.com/devel/traceprofile/kptrace#STPTK`¿



(a) Temporal periodic patterns. The white gaps corresponds to an interruption of the decoding process.

(b) Execution patterns. At each zap, the decoding processes are forked and the children begin to decode the new channel.

Figure 7.6: Patterns appearing on the timeline for the use case Zap

engineers without any interactions with us.

In the next sections, we describe two problems that have been reported by customers on the bug tracker. Then, we explain how the developers have used TraceViz to troubleshoot the problem based on the discussion and explanations found on the bug tracker. Hence, this evaluation is based on usage of TraceViz in real production situation with no guidance from us, and show its ability to help developers uncovering bugs.

7.5.1 Use Case 1: Zap

In this use case, we show how TraceViz makes apparent patterns in a trace. The streaming multimedia application is running under the STLinux[4] operating system on the STiH418 SoC for set-top boxes[5]. A multi-channel stream is received from the network. The application decodes one of the channel and sends it to an external display through the HDMI port. Changing the channel being decoded is called a *zap*. It basically corresponds to the scenario when a user is changing of channel when

[4]STLinux, ¡`http://stlinux.com`¿

[5]STiH418 SoC description, ¡`http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00123853.pdf`¿

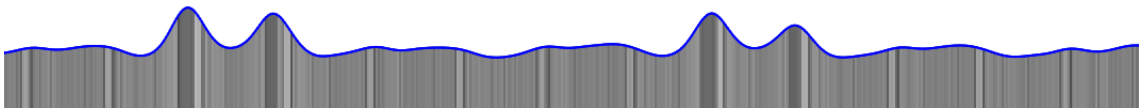


Figure 7.7: Outline with SLG shading. The shading helps to visualize the periodicity of the behavior thanks to regularly spaced black bands.

watching the television, commonly called *zapping*. When recording the trace, we performed 30 zaps consecutively, separated by a delay of 10 seconds. In Figure 7.4, TraceViz all of the trace. At a glance, patterns appear.

Firstly, temporal patterns are represented on both the outline and the timeline. On the outline, regular peaks of activity are apparent. They correspond to the moments when a zap occurred. Abnormal zap executions are quickly detected thank to a suddenly much higher event density. Using the SLG visualization, the zap appear as black vertical bands (see Figure 7.7). The abnormal zaps appear as larger black strips and local maximum on the curve. On the timeline, we can visually recognize which actors are in charge of the decoding process: the decoding is momentarily stopped when zapping and the involved processes are not scheduled during these short periods. It appears as gaps on the timeline (see top rectangle in Figure 7.4, zoomed in Figure 7.6a).

Secondly, behavioral patterns also appear on the timeline (see bottom rectangle in Figure 7.4, zoomed in Figure 7.6b). When a zap occurs, some of the decoding processes are forked. The children will decode the requested channel and the parents which decoded the previous one will stop. This pattern appear 30 times on TraceViz on the timeline.

Based on both the temporal and behavioral patterns, the developer is able to efficiently compare different actors, time windows and to filter-out redundant data to dramatically reduce the amount of data to analyze.

7.5.2 Use Case 2: HDMI black-outs

The application is running in the same environment that the previous use case. It is in charge of decoding a multimedia stream and sending it on an external display via the HDMI output. The issue is reported as sporadic audio and video blanking becoming more frequent under heavy CPU load. It has been reported that the troubleshot occurs independently of the source, whether it is the network or the local hard drive.

After this observation, the issue has been artificially reproduced by decoding a multimedia source from the local disk while loading it with some heavy I/O using the Unix `dd` command. The execution trace has been recorded under these conditions. As described in Section 2.2, the first step of the developer workflow is to open the trace in a synthetic view to check the global system behavior. In Figure 7.8, the trace has been opened in TraceViz. The visualization shows the event density over

Figure 7.8: TraceViz showing an execution when video blanks appeared. The system is artificially loaded with heavy some I/O using the `dd` Unix command, represented in orange. The task *jbd2-sda1-8* is scheduled directly after the *dd* task, causing delays on the treatment of the VSync IRQ callback, in the red rectangles.

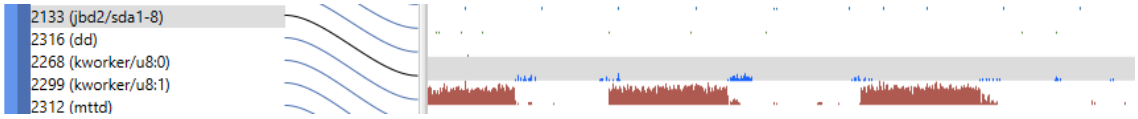


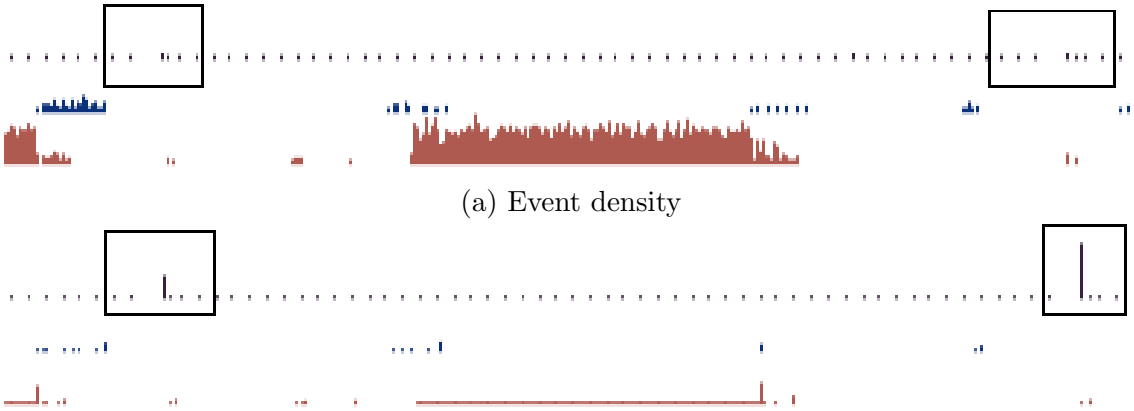
Figure 7.9: The *jbd2/sda1-8* task is scheduled after the *dd* task (in brown).

the whole execution.

The second step consists in using a more detailed view. With TraceViz opened and set to represent the event density, we instantly spot on the timeline view the *dd* task loading the system at regular periods (see Figure 7.8). The task named *jbd2/sda1-8* appears to be scheduled on the CPU directly after the *dd* task and heavily loads the system (see bottom rectangle in Figure 7.8 and Figure 7.9).

The interesting time windows are the periods when the `dd` task is not working. When zooming in one of these time windows (see red rectangles in Figure 7.8 and Figure 7.10), the task *irq/140-vsync0* has its periodic behavior disturbed.

It is the callback of the interrupt request 140 (IRQ): the vertical synchronization (VSync) IRQ on the main output. This thread is in charge of the main decoding process. TraceViz shows an abnormal scheduling delay resulting in a delayed decoding of the stream. As consequence, it introduces a delay between the frames and creates a starvation on the output to finally result as a black screen. Having found



(a) Event density

(b) Delay between events

Figure 7.10: When the *dd* task (in brown) is unscheduled, the *jbd2-sda1-8* task (in blue) loads the CPU, causing a delay on the treatment of the callback for the VSync IRQ on the main output (in purple).

the source of the blackouts, the developers could continue the debugging process by investigating the CPU scheduling, particularly on focusing the task *jdb2/sda1-8*.

Coupled with the developers' domain knowledge, we showed how TraceViz has helped the discovery of a delayed issue.

7.6 Industrial Deployment

We described TraceViz in the previous sections of this chapter. We now explain how TraceViz has been deployed internally at STMicroelectronics as part of their development tools and has been integrated into the platform FrameSoC.

These developments have been done in collaboration with the software engineers Jérôme Correnoz and Julien Dehaudt and with Cyril Fisher who was hired as an intern during six months in 2015. We adopted an agile methodology with two weeks sprints and took the research prototype describe above as the starting point.

7.6.1 STMicroelectronics Toolkit

STMicroelectronics software developers use specific tools developed internally to debug the software layer delivered with their hardware platforms for set-top boxes. This toolkit is named SoCTraces & Profiling Toolkit (STPTK)[6]. It is the direct concurrent of the ARM toolkit named DS-5 [ARM, 2016a] and the open source Eclipse project Trace Compass[7] developed by several partners including Ericsson.

[6]SoC Traces & Profiling Toolkit (STPTK): `http://www.stlinux.com/devel/traceprofile/kptrace#STPTK`

[7]Trace Compass: `http://tracecompass.org/`

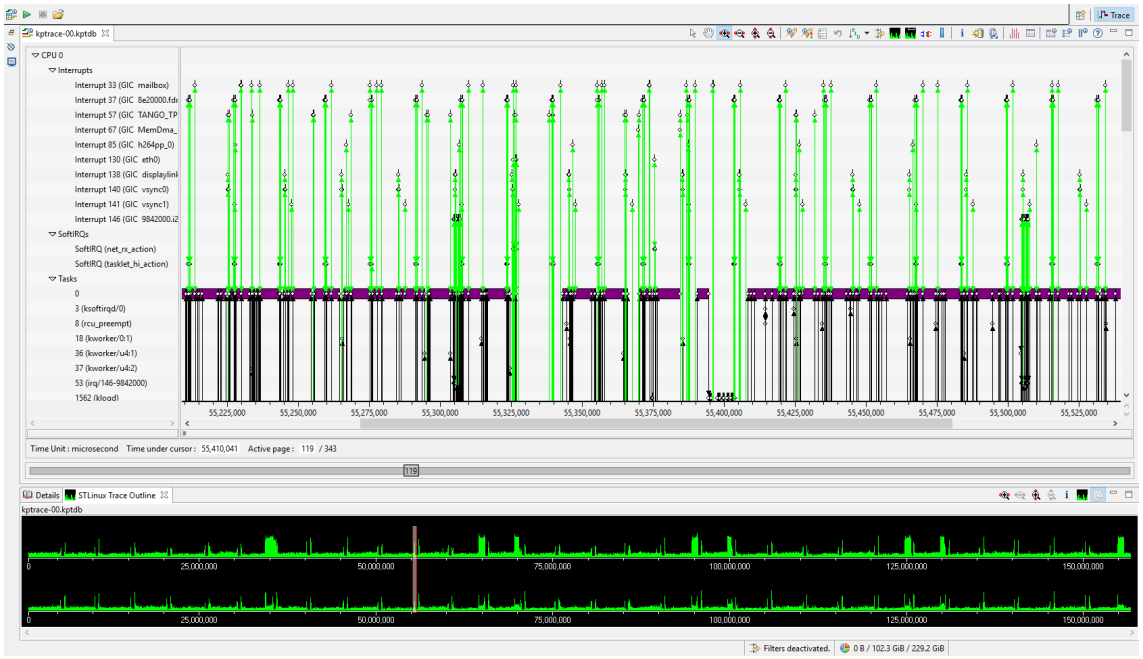


Figure 7.11: SoC Traces & Profiling Toolkit (STPTK). Two views are shown: the time chart (on the top) and the Outline View (on the bottom).

STPTK provides an integrated environment specifically for debugging embedded systems using execution traces. Therefore, it is possible to launch a trace recording with specific settings such as the function to trace and then to analyze the trace with the different integrated tools. Among them, the environment comes with different visualization tools such as a time chart and a synthetic overview of the trace named *Outline View* (Figure 7.11). STPTK is based on the Eclipse platform uses SQLite as backend. A framework has been developed in Java to perform different types of query on the trace: Trace Management Framework (TMF) and serves as a high-level SDK to work with traces. TMF was designed to work with the STMicroelectronics trace format: KPTrace. TraceViz has been integrated into STPTK and the first released was delivered on July 2015.

STPTK is released on a regular basis to both internal and external developers. While we have no precise statistics on the number of developers that have used TraceViz at STMicroelectronics, there are hundreds of software engineers working on the streaming engine and use STPTK as one of their tools. STPTK is also delivered to the customers that bought STMicrolectronics solution to build their set-top boxes. Here, there are potentially thousands of software developers that use STPTK in different companies.

These large number of potential users show that the benefits brought by TraceViz to the trace analysis were significant. Integrating TraceViz to this toolkit required a high level of quality and code robustness.

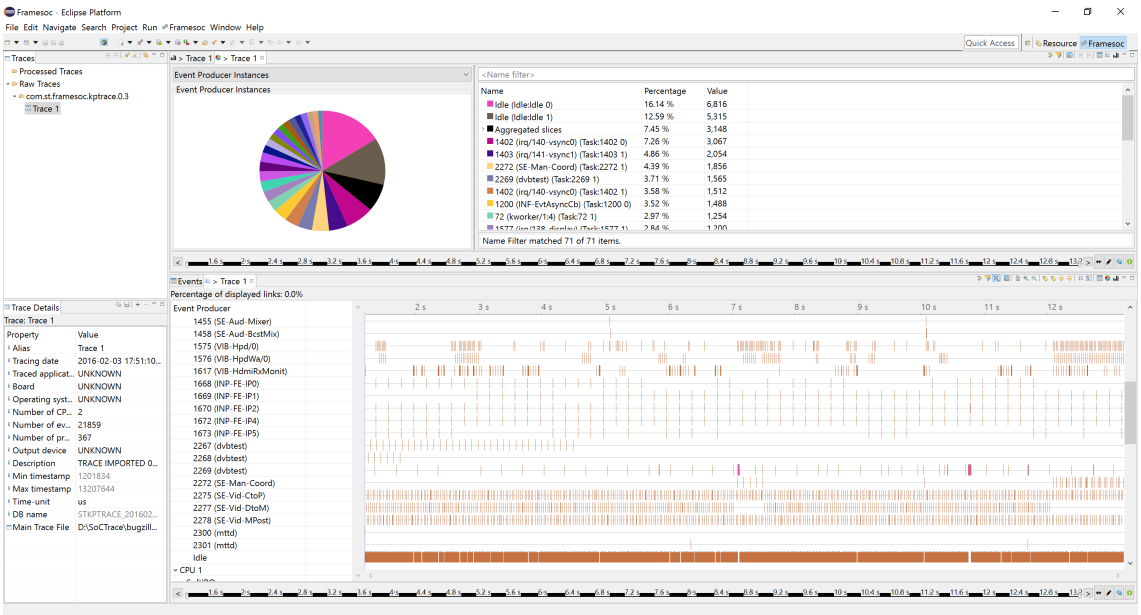


Figure 7.12: FrameSoC interface. It shows a statistics about the event producers instances as a pie chart (top left) and a tabular view (top right). On the bottom is the time chart provided by FrameSoC.

7.6.2 The FrameSoC platform

Similarly to STPTK, FrameSoC is an infrastructure to manage and analyze traces. It has been developed within the SoC-Trace project by several partners (INRIA, Probayes and STMicroelectronics). FrameSoC is an open source project[8] and has made similar technology choices than STMicroelectronics with STPTK: it is based on the Eclipse platform, developed in Java and uses SQLite as back-end. However, FrameSoC support many trace formats such as the Common Trace Format (CTF), KPTrace, Pajé, GStreamer and others. FramSoC has a modular architecture based on Eclipse plug-ins. It provides an API for the trace importers and the different analysis tools that are embedded as Eclipse plug-ins. Several tools are provided as standard such as different trace importers and some views for basic statistics and a time chart (Figure 7.12). In the context of this thesis, a KPTrace importer and TraceViz has been delivered in December 2015.

7.6.3 TraceViz Architecture

To minimize the development time and effort and to be easily adaptable for potential future STMicroelectronics requirements, TraceViz has been designed to be fully extensible. For compliance reasons, we chose to implement TraceViz on top of the Eclipse platform in Java as several Eclipse plug-ins. TraceViz has been implemented

[8]FrameSoC: `http://soctrace-inria.github.io/framesoc/`

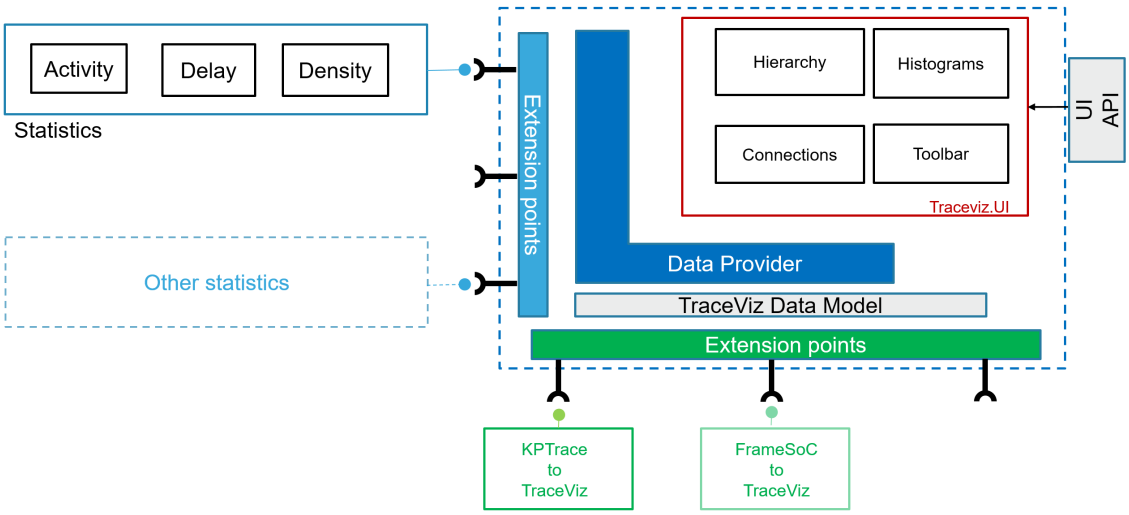


Figure 7.13: TraceViz architecture in STPTK and FrameSoC

using a Model-View-Controller (MVC) architecture and has internally a functional core and a user interface.

The user interface, referred as `Traceviz.UI` on Figure 7.13, is as described previously with few restrictions due to technical reasons: we could not port the pan and zoom interactions due to the SQLite back-end used in both projects. The performance were too poor to support quick navigation in the trace.

The functional core implements two main components: a data model and a data provider. Since TraceViz also had to execute on different infrastructures, whether it is STPTK or FrameSoC, it was mandatory not to be bound to an API or a trace format. To achieve this, TraceViz embeds its own generic trace model. It provides different API to extend it (through Eclipse extension points) to support other trace formats than KPTrace by default. This ensures a low coupling between the functional core and external contributions and provides a flexible mechanism to manage extensions since they are detected at runtime.

Following the same logic, we implemented an API to enable external developers to enrich TraceViz with more statistics than the default ones (event density, activity time and delay between events as discussed earlier in this chapter) and to provide user interface (UI) synchronization with the views of each specific environment. More precisely, we have integrated UI synchronization on TraceViz with both the time charts in STPK and FrameSoC and have exposed all the possible user actions through a public API. For instance, selecting a time window on TraceViz adjust the time frame shown in the time chart and vice versa. The STMicroelectronics implementation supports the search engine of STPTK to perform various queries on the opened trace.

The data provider acts as a controller and is in charge of the communication between the different components of TraceViz.

7.7 Conclusion

In this chapter, we addressed two research problems that were (1) filling the gap between the tools providing a global overview of the execution trace and those proposing a detailed visualization (see Section 4.2) and (2) a need of a fast back-end to interactively explore large execution traces (see Section 4.3). To achieve this, we have presented TraceViz, a novel interactive visualization framework to analyze execution traces.

In Section 7.2, we described the back-end developed for TraceViz. We needed a fast back-end able to respond in 100 milliseconds to be perceived as interactive by the software developers. To achieve this, we chose to implement our solution on top of HDF5 instead of SQLite, traditionally used in such application domain, and took advantage of the naturally sorted property of a time series that enabled us to implement binary search to find the time window to aggregate. We demonstrated its performances through an experiment described in Section 7.5 and compared them against the SQLite solution. The results show that our new back-end largely outperforms the existing solutions and is able to support the interactive exploration of huge execution traces. With our technique, we could propose an efficient solution to the actual limitations of existing tools (see Section 4.3).

On top of this new back-end, we built a novel visualization technique described in Section 7.3 and 7.4. The main goal was to develop a visualization that proposes a trade-off between a too aggregated overview and a too detailed view. Therefore, we introduced a tool that mixes two categories of visualization for time series, the *split-screen* and the *shared-screen* techniques (Section 7.3.2), that maximizes the efficieny of both local and global analysis, responding to the industrial need of such a tool (see Section 4.2).

To guarantee a high visual accuracy, we have used the SLG technique introduced in Chapter 6. The outline view is a Slick Graph. The timeline view is pixel-based and relies on the SLG binning and aggregating algorithm.

TraceViz has been integrated to the STMicroelectronics toolkit for execution traces, STPTK (Section 7.6). Since then, it has been used to solve real bugs that happened during development process of the decoding application. We have presented two of them in Section 7.5. These two elements show that TraceViz addresses correctly the industrial needs for such a tool. It has also proven that TraceViz was mature enough for a large scale deployment. This quick adoption also demonstrates that TraceViz is an efficient tool to reduce the analysis time of an execution trace by revealing patterns hidden by the existing tools.

In the next chapter, we focus on visualizing hidden structures in the trace using results from data mining algorithms. It will provide to the developers an other perspective on the behavior of the systems, hard to spot otherwise.

Chapter 8

Hidden Structures at a Glance

Contents

8.1 Introduction 111

8.2 Definitions and Notations 112

8.3 Structure Computation 116

8.4 Structure Visualization 117

8.5 Experiments 121

8.6 Conclusion 127

8.1 Introduction

In the previous chapter, we introduced TraceViz that provides a new type of overview for execution traces. TraceViz and existing visualization tools providing an overview, discussed in Section 3.2, exploit various aggregation techniques to show the raw data of the trace in an understandable way while trying to minimize visual clutter. Depending on the level of abstraction chosen, some of these structures can be identified by the user's eye as demonstrated in Chapter 7. We described how behavioral patterns of an actor or a group of actors can be spotted. This helped to filter the trace, to target specific part of the trace and to find the root cause of some bugs explained in Section 7.5. However, these methods do not explicitly show some of the structures contained in the trace.

Possible analysis of execution traces can discover a repeated structure (main "regime", disruptions of this main regime, or changes between stable regimes). Understanding what constitutes a regime is not trivial: it consists of some patterns of repetition in the events, and these patterns can, depending on the data and the use case, be of arbitrary complexity. They can be as simple as a mere repetition of a fixed set of events, or as complex as the respect of a complex sequencing of the events combined with periodicity constraints in the repetition. It would be of

tremendous help to software developers analyzing execution traces to have a way to view "at a glance" how such structures exist over the trace, with the most prominent of those structures at each period of the trace as well their evolution over the trace. Existing methods for analyzing traces fall short to these expectations.

On the other end of the spectrum are data mining methods, more precisely pattern mining methods [Lopez Cueva et al., 2012; Lagraa et al., 2014]. Pattern mining methods are designed to find repeated structures such as frequent itemsets, frequent sequences of various kinds, or periodic patterns. Their output is served as a (long) list, where results have to be examined one by one. The state-of-the-art of the pattern visualization (see Section 3.3) showed that most visualization techniques for pattern mining results focus on the problem offering a navigational interface over the set of results and we are not aware of any approach showing different patterns in context within the data, allowing an "at a glance" understanding of complex structure evolution in the data. They often require an expert in data mining to understand what is shown or do not scale correctly as the number of patterns grows and become cluttered. The work presented in this chapter introduces a new pattern visualization for execution trace and aims to address these problems (we provided a deeper description in Section 4.4) by proposing a solution to the research question:

How to exploit pattern mining technique to enrich debugging tools for execution traces?

The contribution of this chapter is to propose a novel *visual analytics* technique to understand at a glance the main structures existing in the data, as well as their evolution over time. This technique is designed for traces, and instead of letting the developers visually discover temporal patterns, it combines a data visualization approach with techniques inspired from pattern mining, but simplified for the purpose of making an understandable visualization.

Our experiments demonstrate the interest of our approach on three real use cases of varied nature: the execution trace of an embedded system, the commit log of the C implementation of the Python language GitHub repository, and the text of the "Foundation" series from Isaac Asimov.

The chapter is organized as follows: Section 8.2 provides the main definitions necessary for this work, and Section 8.3 describes our algorithm to compute the structures. Section 8.4 explains our structure visualization technique. The interest of our approach is demonstrated experimentally in Section 8.5, and Section 8.6 concludes this chapter.

8.2 Definitions and Notations

When analyzing execution traces or more generally time-oriented data, the goal is to understand the global and local trends inside the data and to find the outliers.

A large panel of knowledge discovery and data mining (KDD) techniques focus on searching frequent patterns for meaningful information with no previous knowledge on the data. They return the results under the form of frequent patterns. Such patterns can be itemsets [Cheng et al., 2008], periodic itemsets [Lopez Cueva et al., 2012] or sequential patterns [Mooney and Roddick, 2013].

Depending on the nature of the pattern, the amount of information conveyed vary. For instance, knowing the frequency of an itemset gives less information about the dataset than knowing the frequency of a sequence which itself convey less information than the frequency of a periodic sequence and so on. The more complex is the nature of a pattern, the more information is given to the analyst. Moreover, revealing how an itemset specializes into a sequence with the same items can also indicate relevant information or help filtering-out some parts of the dataset. In this section, we give basic definition in the context of mining execution traces and introduce the notion of *structure*.

8.2.1 Basic Definitions

Recall that execution traces store a sequence of events. Each event has different properties depending on the nature of the application and of the tracing system used but common characteristics remain stable. All the events have an identifier, noted as $id(e)$ for the event e. Each event also has a timestamp that corresponds to the moment when it has occurred. We note $ts(e)$ the timestamp of the event e.

Each event has a type, noted as $et(e)$. We note the set of event types as $\mathcal{T} = \{et_0, et_1, \ldots, et_n\}$ and $|\mathcal{T}|$ is the total number of event types in the data. For instance, in the case of web server logs, the event type can be the HTTP request whether it is a `GET`, `POST`, etc. When working with execution traces, the event type is the operation executed such as a context switch, an entry or exit of an interrupt or a system call. Given an event e, we note its event type as $et(e)$.

We also consider that events are generated by "event producers" that we call *actors*. An actor is an entity that produces at least one event of the dataset. We note $\mathcal{A} = \{a_0, a_1, \ldots, a_n\}$ the set of actors producing at least one event in the dataset. When working with network logs, an actor can be an IP address. In the context of debugging embedded systems using execution traces, an actor is an interrupt, a process, a kernel module, etc. We note $actor(e)$ the actor of the event e.

Our dataset $\mathcal{D}$ is a set of events contained in the execution trace chronologically ordered $\{e_0, e_1, \ldots, e_n\}$. Given an event $e \in \mathcal{D}$, its identifier $id(e)$ corresponds to its position in the dataset. We have:

$$\forall e_i, e_j \in \mathcal{D}, ts(e_i) < ts(e_j) \Leftrightarrow id(e_i) < id(e_j)$$

The set of items $\mathcal{I} = \mathcal{T} \times \mathcal{A} = \{i_0, i_1, \ldots, i_n\}$ is the set of all the event types in the data tagged by an actor. This ensures a finer-grained detailed patterns: it enables to differentiate an event type et produced by the actor a_i from an event type et produced by the actor a_j (i.e a system call performed by two different processes will

be differentiated in the set of items). An item x occurs in the dataset $\mathcal{D}$ if and only if

$$\exists e \subseteq \mathcal{D}, actor(e) \subseteq \mathcal{A}, et(e) \subseteq \mathcal{T}, et(e) \times actor(e) = x$$

An itemset, noted $X = \{x_0, x_1, \ldots, x_n\}$ where x_i is an item i.e. $x_i \in \mathcal{I}$ is an unordered set of items. The set of itemsets present in the dataset $\mathcal{D}$ is noted $\mathcal{X}$. A sequence, noted $S = \langle x_0, x_1, \ldots, x_n \rangle$, where x_i is an item i.e. $x_i \in \mathcal{I}$, is an ordered set of items.

Definition 1 (Specialization)**.** *A sequence S is a **specialization** of the itemset X if and only if $\forall x_i \in S, x_i \in X$.*

A sequence S occurs in the dataset $\mathcal{D}$ if and only if

$$\begin{aligned} \forall x_i, x_j \in S, \quad & j - i = 1, \\ & \exists e_m, e_n \in \mathcal{D}, \\ & ts(e_m) < ts(e_n), \\ & et(e_m) \times actor(e_m) = x_i, et(e_n) \times actor(e_n) = x_j \end{aligned}$$

We can deduce that if a sequence S is a specialization of an itemset $X \in \mathcal{X}$, then $S \in \mathcal{X}$. Also, an itemset X occurs in the dataset $\mathcal{D}$ if and only if there is at least one sequence S that occurs in $\mathcal{D}$ so that S is a specialization of X.

8.2.2 Structure

The most basic information computable for an itemset X is its frequency i.e. its number of occurrences in $\mathcal{D}$. The support of an itemset X, noted $supp(X)$, is the total number of occurrences of its specialized sequences: $supp(X) = \sum_i supp(S_i)$, with S_i being a specialization of X and $supp(S_i)$ the number of occurrences of the sequence S_i in $\mathcal{D}$.

A more sophisticated information about an itemset is the repartition of its specialized sequences. An itemset having k items, a k-itemset, contains $k \times k$ sequences of k items.

For instance, the sequences $\langle A, A \rangle$, $\langle A, B \rangle$, $\langle B, A \rangle$ and $\langle B, B \rangle$ are all specializations of the 2-itemset $\{A, B\}$.

We define as *dominant* the sequence S that has the highest support $supp(S)$ among the specialized sequences of the itemset X. We note the dominant sequence of an itemset X as S_X. This information is important to understand time-oriented data: when considering a couple of events e_i and e_j, it is insightful to know whether e_i occurs before e_j in most cases or not. If not, none of these sequences brings more information about the data than the itemset $\{e_i, e_j\}$.

Knowing whether a sequence is periodic or not also brings meaningful insights about the trace. Given a sequence S, we compute its period p using a Fast Fourier Transform. We define (S, p) the set of consecutive occurrences of S separated by p items in the dataset. A sequence is periodic if $|(S, p)|$ is strictly superior to a

minimum threshold ρ, set by default at $\frac{supp(S)}{2}$. The coverage of a periodic sequence is defined as $\frac{|(S,p)|}{supp(S)}$.

This provides information about whether S is very periodic or occurs mostly at irregular time intervals. Thus, for a given itemset X, we can compute the periodicity coverage of each of its specialized sequences and determine what is the maximal coverage among all the sequences of an itemsets, noted as p_X. We have:

$$p_X = max(\frac{|S_i, p|}{supp(S_i)}), \forall S_i \subseteq X$$

With the combination of the support of an itemset, the repartition of its sequences with their periodicity coverage, it becomes possible to find the sub-parts of the dataset that are mostly periodic as well as whether the dataset contains mainly itemsets (no dominant sequences) or sequences.

We formalize this intuition with the concept of *structure* for an itemset. We define a structure as follows:

Definition 2 (structure)**.** *A structure is a quintuple* $(X, supp(X), S_X, supp(S_X), p_X)$ *with* $X \in \mathcal{X}$*,* $supp(X)$ *the support of the itemset* X*,* S_X *the dominant sequence of the itemset,* $supp(S_X)$ *the support of* S_X *and* p_X *the maximal periodicity coverage among the specialized sequence of the itemset.*

In the structure, we normalize $supp(S_X)$ by $supp(X)$ and then $supp(X)$ by the total number of occurrences of all itemsets in the dataset. Note that in this chapter we focus on the properties of itemset, sequence and periodicity coverage, but other properties could easily be integrated in our tuple notation.

Example

Let compute the sequence for the 2-itemset $\{A, B\}$ having the following dataset:

$$\mathcal{D} = \{A, B, A, B, C, A, B, A, A, B, B, A, B, C, C\}$$

We have the set of items $\mathcal{I} = \{A, B, C\}$. The following 2-itemsets present in $\mathcal{D}$ are $\{A, B\}$, $\{B, C\}$ and $\{C, A\}$.

To compute the structure for $\{A, B\}$, we need to compute the support of the sequences $\langle A, B\rangle$ and $\langle B, A\rangle$. $\mathcal{D}$ contains 5 occurrences of $\langle A, B\rangle$ and 3 occurrences of $\langle B, A\rangle$, giving $supp(\langle A, B\rangle) = 5$ and $supp(\langle B, A\rangle) = 3$. We have:

$$supp(\langle A, B\rangle) > supp(\langle B, A\rangle)$$

so the dominant sequence $S_X = \langle A, B\rangle$. The support of the itemset $\{A, B\}$ is:

$$supp_{\{A,B\}} = supp(\langle A, B\rangle) + supp(\langle B, A\rangle) = 8$$

Finally, we compute p_X, being the periodicity coverage of $\langle A, B\rangle$. Let consider that the Fast Fourier Transform has computed a period $p = 3$, meaning that $\langle A, B\rangle$ begins every three events. We found that three occurrences of $\langle A, B\rangle$ are covered using this period (highlighted in blue):

$$\mathcal{D} = \{A, B, A, B, C, \boxed{A, B}, A, \boxed{A, B}, B, \boxed{A, B}, C, C\}$$

It gives $|S_X, 3| = 3$ and $p_X = \frac{|S_X,3|}{supp(S_X)} = 0.6$. After the normalization of $supp(\{A, B\})$ by the total number of itemset occurrences in the dataset (here equal to 11) and of $supp(\langle A, B\rangle)$ by the number of occurrences of the itemset $\{A, B\}$ (here equal to 8), we have the final structure $(\{A, B\}, 0.73, \langle A, B\rangle, 0.625, 0.6)$.

In this work, we propose a novel interactive technique to visualize these structure, normally hidden to the users and show how it make apparent the underlying structures in the data such as periodic behaviors and perturbations.

8.3 Structure Computation

In this section, we explain our algorithm to compute efficiently all the parameters of the structures. The goal of our tool is to show information usually hidden with the support of the structures. Therefore, it is important to keep the patterns as simple as possible and being able to quickly compute all the information necessary for the structures. Moreover, the computed results do not need to have an exact precision since they will serve as the input of a visualization. To fulfill these constraints, we designed an algorithm that computes the patterns in a naive way but in a time short enough to be used in an interactive visualization.

Algorithm 1 Build structures

Input: Dataset $\mathcal{D}$, itemset $\mathcal{I}$, minimum sequence supportρ, number of time windows W

Output: all the structures that occur in the time windows of $\mathcal{D}$

```
function BuildStructures(D, I, ρ, W)
    structs ← [ ]
    TW ← sliceDataset(D, W)
    for all w ∈ TW do                                  ▷ in parallel for each w
        freqItems ← buildFreqItems(w)
        S ← buildSequences(freqItems)
        seqOccs ← findSeqOcc(W, S)
        P ← findPeriod(seqOccs)
        structs_w ← buildStruct(seqOccs, P, ρ)
        add(structs, structs_w)
    return structs
```

The function sliceDataset splits the dataset $\mathcal{D}$ into W time windows. Slicing the dataset is an important parameter to set the precision of the results. It greatly influences the nature of the patterns discovered by the algorithms. When working with time-oriented data, the analysis becomes more local as the number of time

windows to slice the time dimension increases. The task of the analyst may be to analyze globally the dataset to study the high-level properties of the structures. In this case, the dataset will be sliced in a few number of time windows. In contrary, comparing local behaviors can support the discovery of perturbations by detecting a different sets of structures in a time window. For each time slice, all the parameters of a structure described in section 8.4 are computed for all the possible itemsets. Doing this produces local results detailed for each time slice and makes possible to detect regular behavior across of the windows as well as perturbations that happened in a time slice. The structures are computed for each time window in parallel.

The function BUILDFREQITEMS compute the number of occurrences for each item $x_i \in \mathcal{I}$. It returns a set of items $freqItems$ so that the occurrences of all the items $x \in freqItems$ covers 80% of the total number of the occurrences. By doing so, we are able to discard a large number of items that occur a few times and mitigates the computational time of the algorithm. Also, as the visualization aims to show the tendency inside the data, the sequences having a very low support are unlikely to be visible on the final rendering. Thus, discarding the least items that occur the least in the dataset prevent the sequences whose support $supp(S)$ is very low.

The function BUILDSEQUENCES generates exhaustively all the possible sequences from the items contained in $freqItems$. It returns a set $\mathcal{S}$.

FINDSEQOCC find all the occurrences for the sequences in $freqItems$. This function is the most time consuming task of the algorithm so an efficient algorithm has to be used. We have implemented the SOG algorithm [Salmela et al., 2006]. It is based on bit parallelism and q-Grams to perform exact multiple pattern matching in linear time. In our case, the alphabet Σ is the set of items $\mathcal{I}$, whose size $|\Sigma| = |\mathcal{I}| = |\mathcal{T} \times \mathcal{A}|$, can potentially be very large. The pattern to search in the dataset are the sequences for which we limit their size to be small. We have selected the SOG algorithm since it is the best performing algorithm for multiple pattern matching with a large alphabet size and a small pattern length [Kouzinopoulos and Margaritis, 2014]. We use 2-gram in our implementation: benchmark shows that using 3-gram is much slower and memory consuming than using 2-gram for up to 10^5 patterns.

The method FINDPERIOD takes as parameter all the positions of the occurrences for each sequence. For each sequence, it performs a Fast Fourier Transform (FFT) and select the period p which allows to maximize $|(S, p)|$.

The final step, implemented in the function BUILDSTRUCT is to construct the structures from the results previously computed. It returns all the structures sorted according to support of the itemsets.

8.4 Structure Visualization

Visualizing the structures as defined in the previous section can reveal meaningful information about the underlying behavior hidden in the data. According to Shneiderman [Shneiderman, 1996], a good visualization technique has to provide a

pipeline as *Overview first, zoom and filter, then details-on-demand*. When designing our technique, we followed this guideline and integrated an overview to visualize all the structures with a detailed representation of a single structure.

We begin by explaining which tasks the visualization tool has to support and what are the benefits it brings to leverage the difficulty of analyzing the structures. We continue with the description of the design of the visualization to render in a clear way a huge amount of structures, providing an overview of the dataset to understand the global trends and perturbations. The structure overview is coupled with an overview, a stacked graph of the actors built with a Slick Graph. It is identical to the overview provided with TraceViz. Finally, we explain how a structure selected by the user is rendered in a detailed visualization. Figure 8.2 shows the whole interface with the different views described below.

8.4.1 Goals

In this work, we propose a visualization technique to enable the user to quickly understand the hidden structures in the data and designed to address the following goals:

1. *Quickly understand the underlying structures contained in the dataset.* Data mining algorithms provide a very large number of patterns in most cases. These results contain meaningful insights to support the understanding of the dataset. However, it is a very complex and time consuming task to take advantage of them and this task requires an expert. Providing an intuitive visualization technique is mandatory to harness the patterns and to shorten the time needed for the analysis.

2. *Simplified parameters settings.* Data mining techniques have different parameters to tune their output. The visualization has to provide user interactions to simplify the exploration of the parameter space.

To better support the understanding of the structures, the visualization has to show the nature of the patterns whether it is an itemset, a sequential pattern and if it is periodic. For a given itemset, the information about the sequences repartition has also to be conveyed by the visual representation as well as if the sequences are periodic or not. Therefore, the visualization takes as input the support of the itemsets, the repartition of its sequences and the percentage of sequences covered by its period if any.

8.4.2 Structures Overview

The algorithm to compute the structures takes as parameter a number of time windows. Since the task cannot be pre-determined, this parameter is controlled interactively by the user at the moment of the analysis. By enabling this interactions,

our tool supports the study of the evolution of behavioral patterns from a global to a local point of view.

Many Structures Visualization

To provide a clear visualization that does not overwhelm the user by the amount of information, a structure has to be represented in a compact yet clear way. Let consider a structure $S = (X, supp(X), S_X, supp(S_X), p_X)$. The different components of the structure are mapped on visual parameters. Figure 8.2a shows the structures visualized below the overview.

A structure S is represented with a rectangle whose support $supp(X)$ is encoded in the height of the rectangle. The vertical space is completely filled with all the itemset. Its width is constrained by the size of the time slice. In each time window, the structures are ranked according to the support $supp(X)$: the greater its support, the higher its position. By doing so, more visual space is given for the most frequent itemsets which are concentrated at the top of the rendering. It ensures that the most active actors are quickly detected by the eye.

Next, the color encodes the information indicating whether an itemset has a dominant sequence or not. We define a threshold ρ to determine if the itemset has a dominant sequence. If $\rho \leq supp(S_X)$, the itemset X has a dominant sequence. In this case, the rectangle is rendered in blue otherwise, the rectangle of the itemset is filled with black. Areas of the traces where the data are more structured can be quickly spotted.

Lastly, we encode in the channel alpha (i.e. in the opacity) of the rectangle the periodicity of the sequence S_X, p_X. The more a sequence is periodic, the more transparent is the rectangle representing the structure. When working with dataset where the structures are mostly periodic, it is important to fade out the periodic data since it corresponds to the correct behavior. Table 8.1 shows different rendering of an itemset depending on the value of the parameters.

When hovering a structure, all the rectangles representing S_X become red (Figures 8.2b, 8.3) as well as the layer of the actors present in the sequence. This shows the distribution of a single sequence over the whole dataset. A visualization of the structure also appears next to the cursor to show a detailed representation of the itemset, its sequences and periodicity.

8.4.3 Visualizing Structure Details

Following the Shneiderman's guidelines, more details can shown when requested by the user [Shneiderman, 1996]. When hovering a structure, a tooltip appears representing all the values of the structures.

On the overview, to better make apparent the regular behavioral patterns, partial information about the itemsets and their specialized sequences are shown, hiding important information: how an itemset specializes into its sequences and what are precisely the items.

Dominant sequence	Periodic sequence	Rendering
✗	0%	
✗	40%	
✗	90%	
✓	0%	
✓	40%	
✓	90%	

Table 8.1: Visual representations of a structure

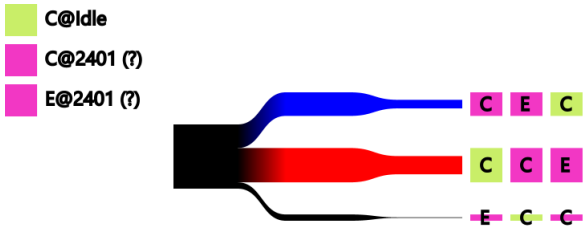


Figure 8.1: Visualization of a structure. The root of the diagram is the itemset X of the structure. Each branch corresponds to one of its specialized sequence S_X that occurs at least once in the dataset. The thickness of the first and second segments respectively encode $supp(S_X)$ and p_X. The branch colored in red represents the structure currently highlighted by the user while exploring the data. On the top left are rendered all the items belonging to the itemset of the structure.

The tooltip shows these information using a Sankey diagram (Figure 8.1). Traditionally used to represent flow of energy and resources, it encodes here the different parameters of a structure. The figure represents the structure

$$\begin{aligned}(X = \{C@2401, E@2401, C@Idle\}, \quad & supp(X) = 0.62,\\ & S_X = \langle C@Idle, C@2401, E@2401\rangle,\\ & supp(S_X) = 0.53,\\ & p_X = 0.46)\end{aligned}$$

On the top left of the tooltip, the items of X are listed next to a square filled with the color of the actor used in the overview. The root of the diagram (on the left) is the itemset X of the structure, in black to be consistent within the different views as an itemset with no dominant sequence is rendered in black. The itemset split into different branches, one branch per specialized sequence of the itemset that has at least one occurrence in the dataset. The branch are colored according to the user defined threshold ρ that set the minimum coverage of a periodic sequence and their thickness encodes the support of the sequence. The branch corresponding to the highlighted structure is colored in red. At the end of each branch, the sequence is represented using one square per item. Each square is filled with the actor's color of the item and the event type is written on the square.

On Figure 8.1, the highlighted branch is half the height of the itemset since $supp(S_X) = 0.5$.

The last segment of the branches corresponds to p_X. The wider the last segment, the higher p_X. It shows intuitively to the user how periodic the sequences are. In our example, we have $p_X = 0.46$, thus the last segment is $0.46\times$ as wide as the previous segment. Note that in the detailed visualization of a structure, the support $supp(X)$ of the itemset X is not encoded since to be meaningful, it needs to be put into context, in the time window.

8.5 Experiments

In this section, we present three use cases with different data: (1) execution traces, (2) the CPython Git repository and (3) text. For each of these, we begin by describing the input data, what are the events, the event producers, the actors and how we built the dataset $\mathcal{D}$. The experiments show that visualizing structures can efficiently reveal structural behavior and perturbations quickly using very different types of data.

8.5.1 Execution Traces

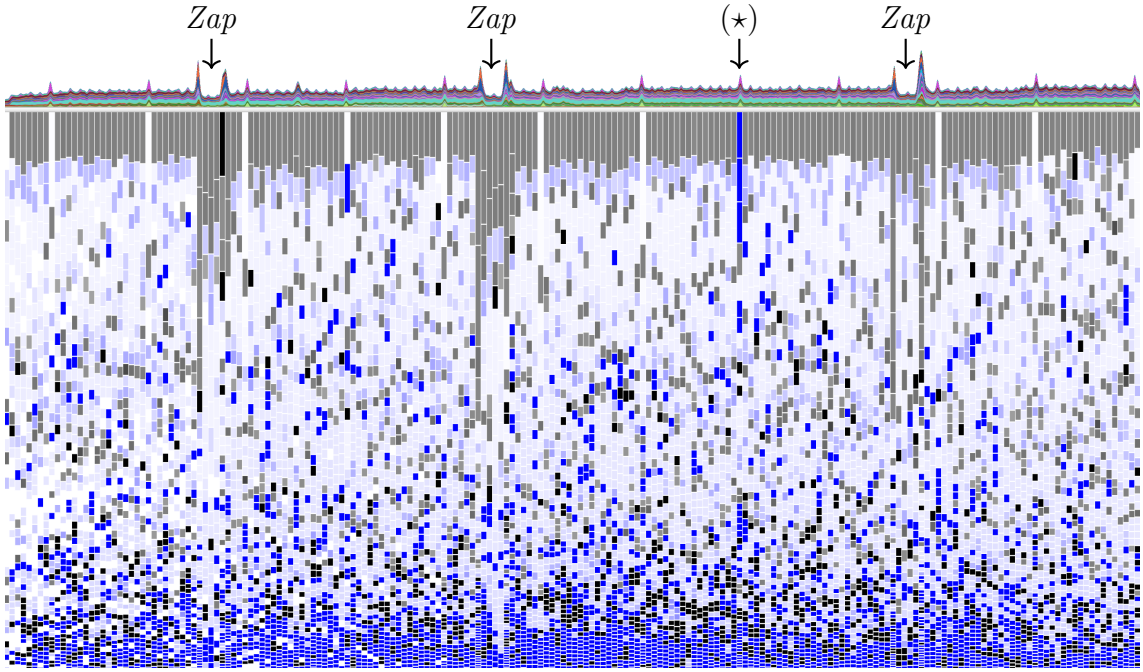
In this paragraph, the data are traces recorded during the execution of a streaming multimedia applications used to play music and video. We provided a detailed explanation of multimedia applications in Chapter 2.

Here, the dataset $\mathcal{D}$ contains all the events that occurred during the execution. The set of actors $\mathcal{A}$ are the processes and interrupts that produced at least one event during the execution, noted with their process id (PID) in the tool. The event types $\mathcal{T}$ are the instructions executed. It can be a context switch (`C`), an entry (`E`) or exit (`X`) of a system call, an entry or exit of an interrupt respectively noted as `I` and `i` in the visualization. Examples of items contained in the set of items $\mathcal{I} = \mathcal{A} \times \mathcal{T}$ are `C@1234` (a context switch on process `1234`), `E@4321` (a system call performed by the process `4321`) and `i@Interrupt 567` (exit of the interrupt `567`). In this use case, we show how understanding the structures in the data supports the developers to debug their application.

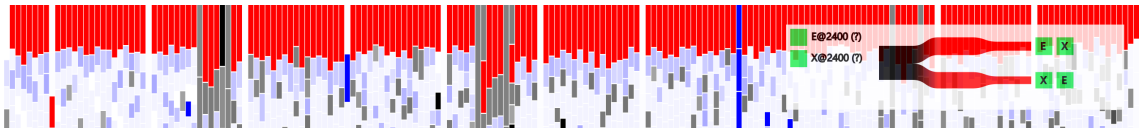
The stacked graph at the top represents the event density for each of these producers, hence a peak on the graph shows a local increase of the number of events on the system. In this use case, the execution trace has been recorded on an embedded system that decodes a multimedia stream for the television. The stream is received through the network on the Ethernet port.

During the execution, the user has changed three times the channel to decode (channel zap). These moments correspond to the three peaks of activity that appear on the stacked graph (see the overview on Figure 8.2).

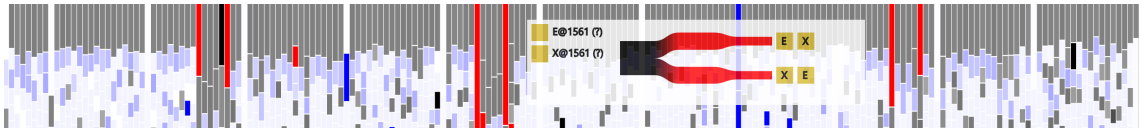
On the structure overview (Figure 8.2), three horizontal areas appear: a gray area on the top, a clear middle section and a mostly blue area at the bottom. The



(a) Global visualization interface with the overview on the top and the structure overview on the bottom.



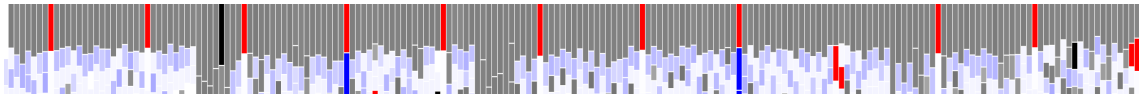
(b) Dominant structure in the trace. It involves a single process whose PID is 2400 that performs a huge number of system calls.



(c) Dominant structure during a zap channel. The process 1561 performs many system calls.

Figure 8.2: Structures of an execution trace.

top gray bar shows that the most frequent structure on the whole trace is a structure that has no dominant sequence and limited periodicity. It involves a single process (the process 2400) that performs a huge amount of system calls (Figure 8.2b). The itemset has no dominant sequence due to its small size (2 items) and a high frequency. Its behavior is disturbed when a zap occurs: there is a much higher number of frequent itemset involving different processes. The structures show that during a zap, a single process mostly works, performing many system calls (Figure 8.2c). No periodic sequence appears: this reflects the perturbation of periodic decoding behavior when switching the stream to decode.



(a) Regular pattern on the *General Interrupt Controller*

(b) Visualization of the structure corresponding to the interrupt 146. The structure is composed of only one sequence $\langle I@Interrupt146,\ i@Interrupt146\rangle$, that stands for entry and then exit of interrupt 146. The sequence is very periodic.

Figure 8.3: Periodic behavior of the interrupt 146 shown on the structure overview and in details.

A very periodic sequence occurs regularly among the most frequent structures and appear as white bands on Figure 8.2, highlighted on Figure 8.3a. It shows a behavioral pattern at a lower frequency involving an interrupt named *GIC*, namely *General Interrupt Controller* (Figure 8.3). It consists in a general hardware resource to manage the interrupts.

The visualization shows a periodic sequence: it consists in the entry and the exit of the interrupt and shows a periodicity breaking by a blue bar (noted as ($\star$) on Figure 8.2a). It shows the developer that an abnormal behavior happened in this time window.

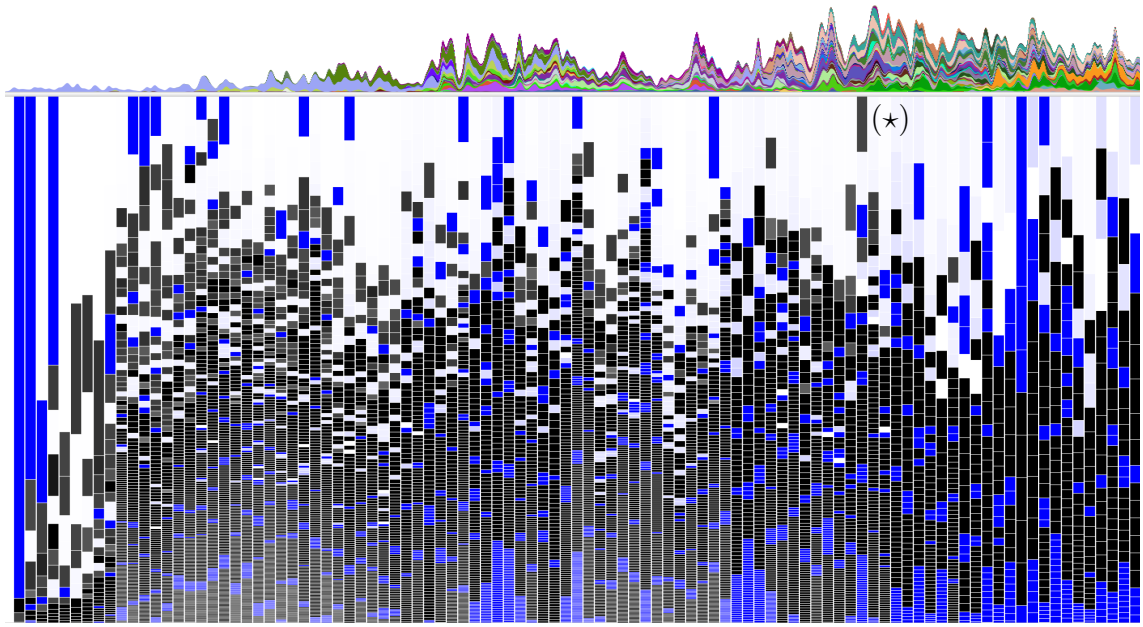
The middle section contains a large amount of periodic structures. This is induced by the nature of the application: decoding a multimedia stream is a very periodic task: frames are decoded at a constrained rate (typically 25 frames per second) to ensure a smooth video playback. On the bottom we have many sequences. This is a normal behavior since the functions are called sporadically, generating many entry/exit events in the trace.

8.5.2 CPython Git Repository

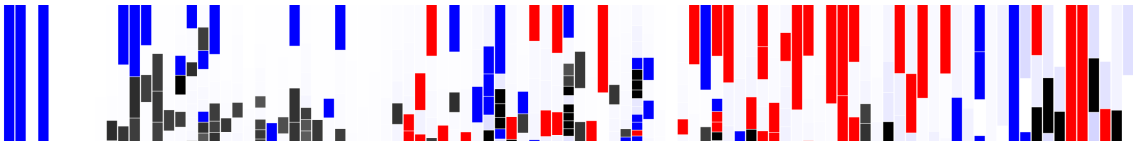
In this paragraph, we analyze a Git repository and show how the visualization of the structures gives insights on the project organization.

To build the dataset $\mathcal{D}$, we extracted the timestamp and the developer of each commit of the Git repository creating one event per commit. The set of actors $\mathcal{A}$ contains the developers that have committed in the repository and there is a unique event type `C` that stands for *commit*, thus $\mathcal{T} = \{C\}$. By doing so, we focus on the commit behavior of the developers.

Figure 8.4a shows the structures with 3 items for the Git repository of the Python



(a) CPython 3-itemset



(b) Structures in which the core developer Victor Stinner is involved

Figure 8.4: Structures extracted from Git repository of the CPython project

implementation in C, CPython[1]. We extracted the commits from the complete history of the project. A majority of the dominant structures are periodic sequences involving one developer. It can be explained by the developers' habits: it is common practice that a developer performs several commits in a row when working on a piece of software. When analyzing which developers are involved in the most frequent structures, core developers are largely represented: they work more regularly on the project than external developers. Figure 8.4b shows in red all the structures in which the core developer Victor Stinner is included. It shows that this developer is highly active, performing commits very frequently on a regular basis. The less regular developers appear as non-periodic sequences: they do not commit often enough to have a periodic behavior.

Among the most frequent structures, only one is an itemset, appearing in dark grey (noted as (⋆) on Figure 8.4a). The structure (Figure 8.5) shows that two developers have been committing concurrently leading to interleaving commits: Victor Stinner and Serhiy Storchaka, an other active core developer. Both have received

[1]Python: `http://python.org`

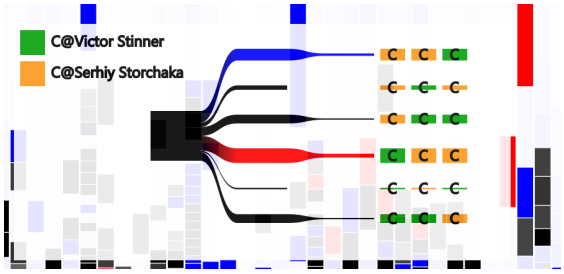


Figure 8.5: Structure showing two developers committing at a high rate simultaneously.

an award from the Python Software Foundation in July 2015[2].

Visualizing the structures related to the commits support the discovery of the main developers of a project and the regularity of the commit habits.

8.5.3 Foundation Series

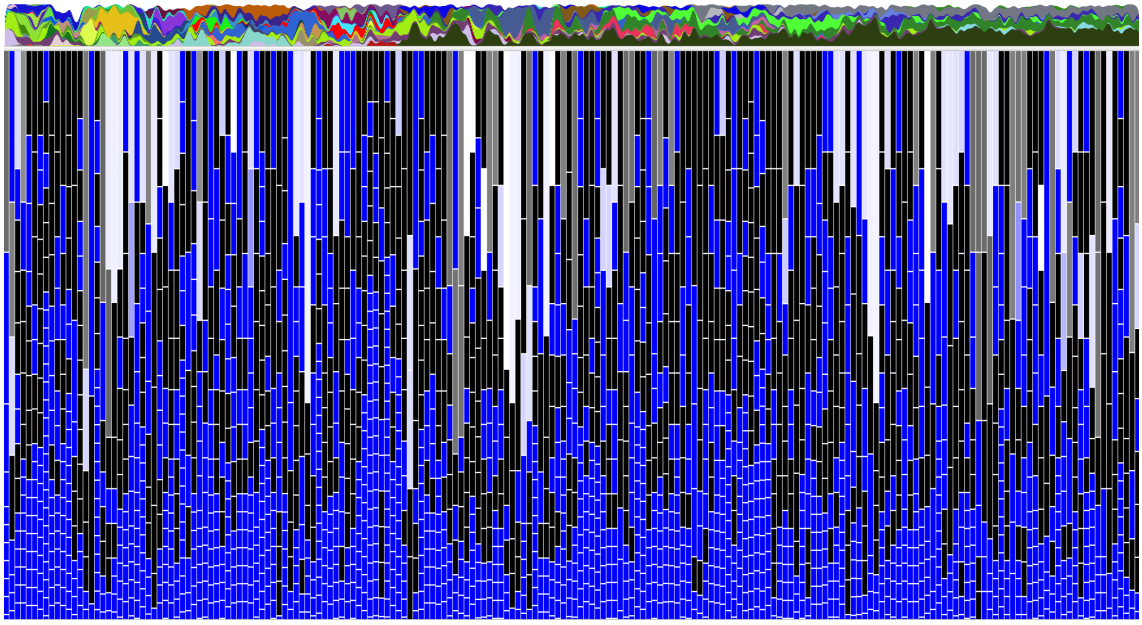
In this experiment, we describe how visualizing the structures provides relevant insights about a story. As an example, we took the "Foundation" series from Isaac Asimov.

We built the dataset $\mathcal{D}$ as follows: (1) we extracted all the characters and planets' name from the whole story, (2) we created a log file such as it contains all the occurrences of each characters and planets in the text and (3) we created an event from each occurrence in the log. It results that the set of actors $\mathcal{A}$ contains all the characters and planets of the series. We used an unique event type, named `I` for *intervention*, giving $\mathcal{T} = \{I\}$, the set of event types. We have $|\mathcal{I}| = |\{\mathcal{T} \times \mathcal{A}\}| = |\mathcal{A}|$.

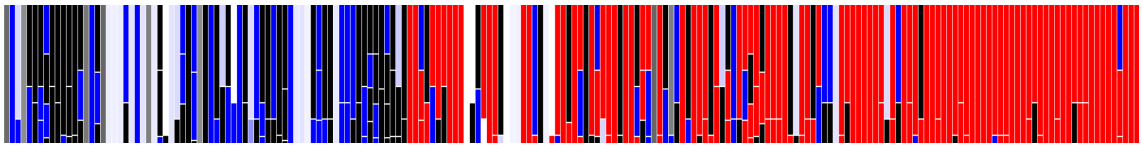
Figure 8.6 shows the visualization of the structures using 2-itemsets. We instantly notice that most of the least frequent structures have a dominant sequence. As the structures become more frequent, the structures become very periodic (whiter) and tend to have no dominant sequence.

The structure can be composed of a single character. It shows that the character is very active and his name occurred many consecutive times in the text indicating that at least a paragraph has been dedicated to the character. On Figure 8.6b, all the structures containing the item TREVIZE are highlighted. *Golan Trevize* is the protagonist of the series starting from "Foundation's Edge". The huge amount of highlighted structures that are very frequent indicates that Trevize is a very active character that interacts with many other protagonists and planets. This fits the story since he travels across the galaxy to visit many different planets in his quest to find the second Foundation. Trevize travels with the professor of ancient history Janov Pelorat. Figure 8.6c shows the structures containing the itemset $\{trevize, pelorat\}$. The visualization indicates that it is a very active structure with

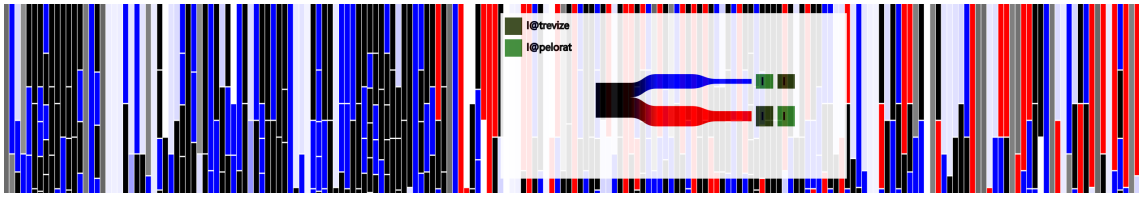
[2]PSF Service Community Award, July 2015: `https://www.python.org/community/awards/psf-awards/#july-2015`



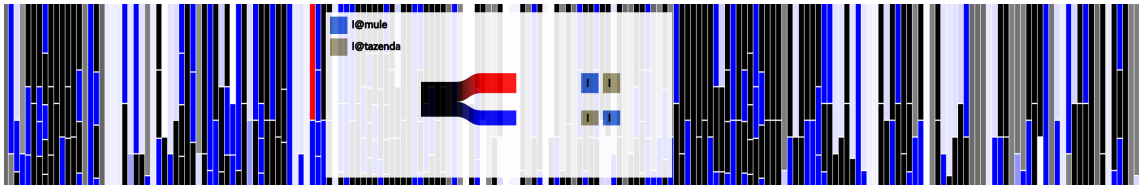
(a) Structures using 2-itemset in the Foundation Series



(b) Dominant structure in the text. The character Golan Trevize is the main protagonist starting from “Foundation's Edge”



(c) Dominant structure in the text involving the character Janov Pelorat and Golan Trevize



(d) Dominant structure in the text involving the character The Mule and the planet Tazenda

Figure 8.6: Visualizing the structures in the “Foundation” series from Isaac Asimov

no dominant sequence: the two characters interact quite evenly.

The structure can be built with a character and a planet. This case is not very frequent since the story involves many characters and planets. Figure 8.7 represents

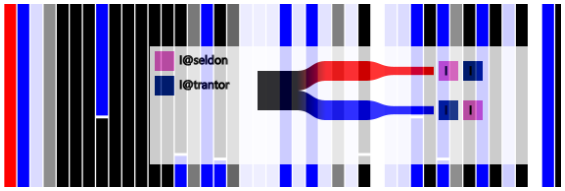


Figure 8.7: Dominant structure at the beginning of the text involving the character Hari Seldon and the planet Trantor

the most frequent structure on the first time window. It contains the items for the character Hari Seldon and the planet Trantor. It fits with the story that begins at Hari Seldon's office at the Streeling University, located on the planet Trantor. The background of the series is given to the reader: the psychohistory and the Seldon Plan.

Figure 8.6d shows a structure that is locally the most frequent involving the character The Mule and the planet Tazenda. It corresponds to the moment when The Mule believes that the Second Foundation is located on Tazenda and decide to blow up this planet.

We have shown in this experiment how visualizing the structures allows to understand the dynamic of a story and its main characters.

8.6 Conclusion

In this chapter, we addressed the question discussed in Section 4.4 that stated a lack of understandable pattern visualization techniques for execution traces. For this, we presented a novel visual analytic techniques that shows the hidden structures in traces. We simplified the patterns representation to make it scalable with the number of patterns the parameter settings to leverage the usage of this tool to non data mining expert users.

In Section 8.4, we formally defined a structure as being a combination of an itemset, its dominant sequence, its support and the support of its most periodic sequence. Based on this notion of structure, we proposed a visual analytic technique to enable the software engineers to understand "at a glance" the repetitive behaviors that can be complex patterns involving sequences of events and periodicity. Therefore, the regular behavior that implies repetitive structures are easily detected as well as the perturbations over the trace.

In Section 8.5, we have described several use cases with different types of data due to the industrial context (see Section 5.2.3) and to show the genericity of our proposition. Among them, we have demonstrated how our approach supports the analysis of an execution trace and how, with these insights, the software developers can easily focus on a temporal window of the trace, for instance where anomalous structures break the main regime. We also modeled a Git repository as a time series

using the commits and the developers as input data and books using the locations and characters of the story. In both cases, we showed how the structures have been revealed such as the most active developers and potential groups among them. In the second case, the major events could be identified as well as the structure of the story such as the focus on two characters.

With TraceViz, presented in Chapter 7, this tool provides a complementary solution to analyze the trace. TraceViz reveals more simple patterns spotted visually by the developers and provides an interactive exploration of the trace. On the other side, visualizing the hidden structures reveal more details about the events such as their type, thus allowing a finer-grained analysis.

In the next chapter, we describe how the different techniques proposed in this thesis can be used in combination to support an efficient debugging process.

Chapter 9

Study of an Integrated Debugging Workflow

Contents

9.1 Introduction 129
9.2 Example of an Analysis Workflow 130
9.3 Use case: TSRecord 130
9.4 Conclusion 134

9.1 Introduction

In the Contribution part of this thesis, we have presented three different works to propose a global solution to the problem of improving the analysis tools for execution traces to shorten the time required for the debugging of streaming multimedia applications.

We presented Slick Graphs in Chapter 6 to provide a precise visualization of smoothed time series. Next, we introduced TraceViz, a visualization framework for execution traces. We used the binning and aggregating algorithm as the basis of the TraceViz timeline view. It offers an overview of the trace and make visible some behavioral patterns related to the synchronization between the actors and their periodic execution. Finally, we introduced a different type of overview as a visual analytic tool to show the underlying structures hidden in the trace. In both proposition, we integrated a Slick Graph to visualize the global activity of the system.

In this chapter, we show how these tools can be used in a complementary way. The definition of a methodology is critical to help the software engineers to familiarize themselves with tools that change their working habits. To support our discussion, we study a bug that occurred at STMicroelectronics using a possible combination of the different techniques introduced in this thesis. We describe the

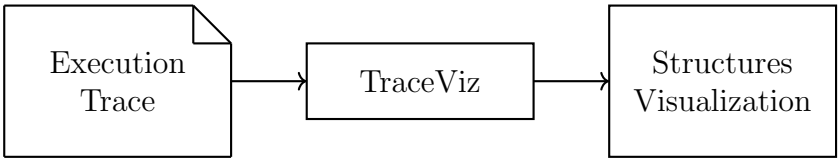


Figure 9.1: Example workflow integrating TraceViz and the visual analytic tool.

methodology adopted for this use case in Section 9.2 and we analyze the trace of the use case in Section 9.3.

9.2 Example of an Analysis Workflow

We describe a possible integration of TraceViz and the structures visualization designed for the use case presented in Section 9.3. Figure 9.1 depicts the workflow of the methodology used in this chapter.

After having recorded the trace during the decoding, the developers have in the best cases a rough idea of what went wrong but most of the time have no idea where to begin the analysis. Thus, we propose to start the analysis process by importing the trace into the TraceViz back-end and open it in TraceViz. By doing so, it gives the developers the possibility to begin to explore and filter the data through the interactions described in Section 7.4. It also shows the behavioral patterns present in the trace.

After having filtered the trace according to their knowledge and the patterns discovered, the second step of the analysis consists in using the visual analytic tool described in Chapter 8 to check whether some structures brings more information or not. We brought the following modifications to the tool:

1. We plugged the algorithm to compute and rank the structures onto the TraceViz back-end. By doing so, instead of taking as input the complete trace, the filtered-out actors are ignored and the structures are computed only on the time window of the trace visible in TraceViz.

2. The structure visualization replaces the timeline view of TraceViz when the visual analytic tool is launched. However, the hierarchy of TraceViz is still visible on the left side and hovering an actor in the hierarchy also highlights all the structures where this particular actor is involved. Figure 9.2 shows the resulting interface.

9.3 Use case: TSRecord

To illustrate this example of integration, we study a use case that occurred at STMicroelectronics. However, instead of explaining the process undertaken by the

Figure 9.2: TSRecord trace visualized in TraceViz.

software developers (as it was the case in Section 7.5), we describe in this section one possible approach for the analysis, performed by ourselves.

TSRecord is an application that receives a multimedia stream from Internet and is in charge of recording it on an external storage. It corresponds to the typical case when the user wants to record a program on television and watch it later. Similarly to for the previous use cases, TSRecord was running on the family of board, here the STiH416 SoC, and under the STLinux operating system. In this use case, the video was received through the Ethernet port of the set-top box and recorded on an USB storage.

Here, the problem is that when playing the recorded content, some video blanking and artifacts as well as audio scratches appear. This comes from some missing frames and audio data.

After importation, the complete trace is visualized in TraceViz using the *event density* statistic (see Figure 9.2). The trace contains a small number of actors due to the relative simplicity of the application. Among them, several are of low interest such as the `kptrace*` and `klogd` actors which are the KPTrace processes in charge of recording the trace (see Section 2.5.2), the SSH processes, etc. Others seem particularly interesting such as the processes `46 (usb_storage)` highlighted in grey on Figure 9.2 and the process `1798 (ts_record)`. Therefore, the first step to do is to filter-out useless processes to better focus on the processes involved in the TSRecord execution using the interaction provided by the TraceViz hierarchy 7.4.5. We hid the actors related to the trace recording and basic system processes (`sshd`, `kptrace*`, etc.) and ended up with the result shown in Figure 9.3.

After the filtering process, the interesting actors that remain are the following:

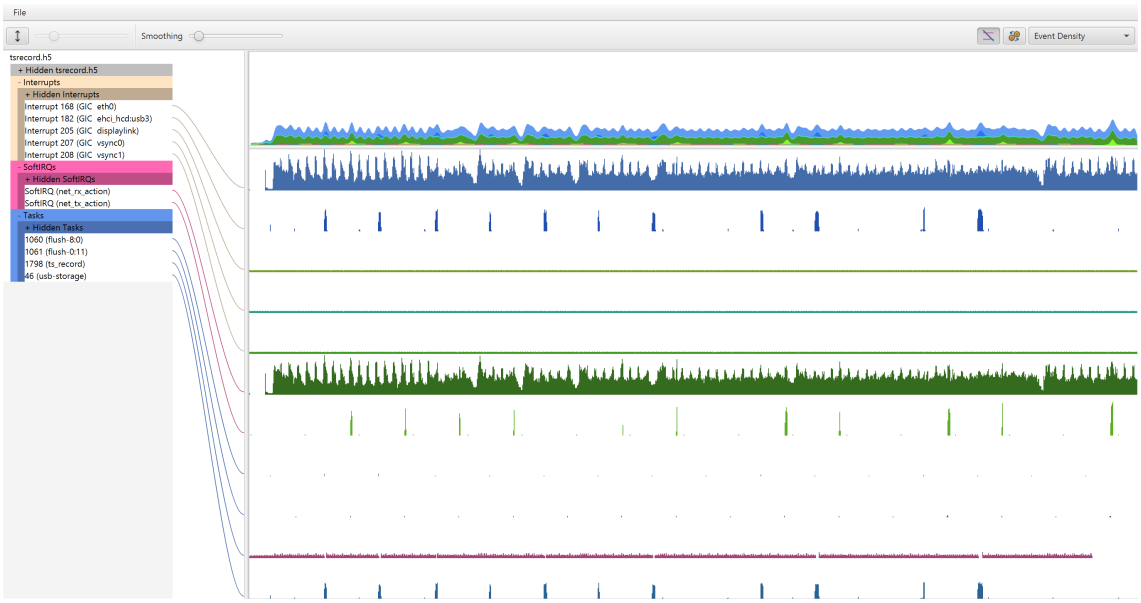


Figure 9.3: TSRecord trace visualized in TraceViz after having filtered-out irrelevant actors.

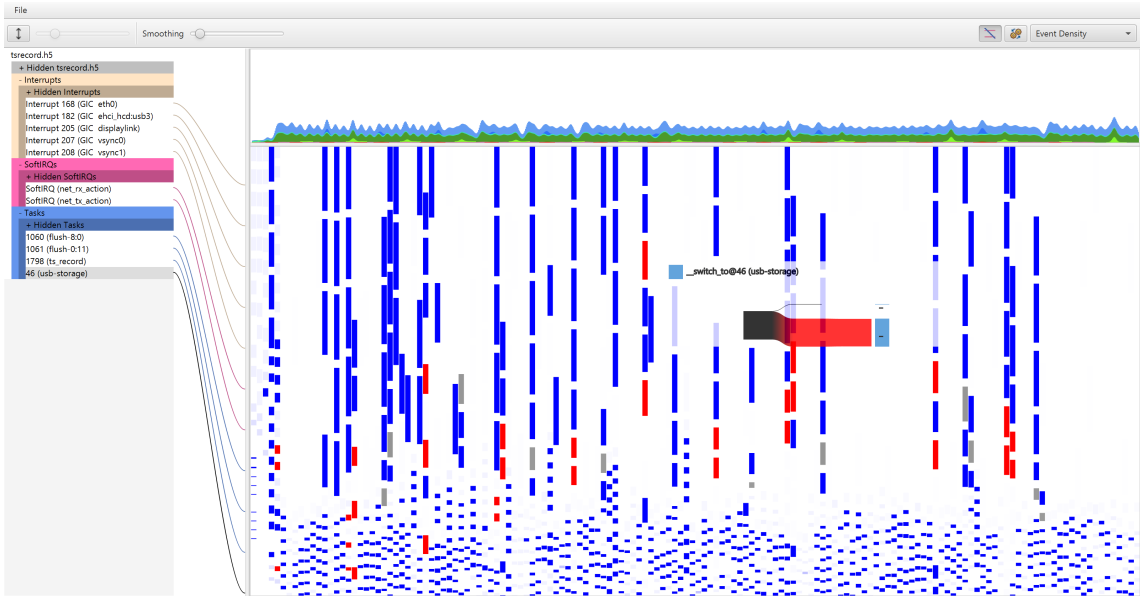
- `Interrupt 168 (GIC eth0)` and `SoftIRQ (net_*_action)`. These interrupts are related to the network. They appear to be the most active actors in Figure 9.3. Indeed, we have chosen to visualize the event density and interrupts related to the network occur when data is received, generating here a huge number of events.

- `Interrupt 182 (GIC ehci_hcd:usb3)` and `46 (usb-storage)`. As their name indicates it, both are related to the USB storage management.

- `1798 (ts_record)`] The process corresponding to the execution of the TSRecord application.

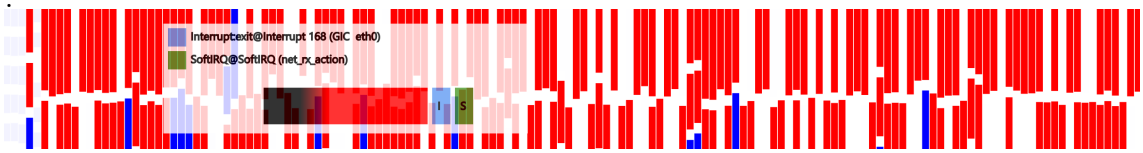
- `1060 (flush-8:0)` and `1061 (flush-0:11)` The kernel processes corresponding to the `pdflush` kernel thread in charge of writing the data on the external storage. We give more details below.

The process `46 (usb_storage)` is of particular interest. TraceViz reveals that at the beginning of the execution, it had a periodic behavior and its periodicity go disturbed at some point. Consequently, `46 usb_storage` has long periods of inactivity that can be the reason why some data is missing in the recorded video. In fact, `1798 ts_record` writes the data using the system call `write`. However, the data are not physically written on the external storage at this moment. Instead, the Linux kernel bufferizes these data into a special location of the main memory called *page cache*. A thread kernel called *pdflush* is scheduled periodically, 5 milliseconds by default, to write the data contained in the *page cache* to the external storage.



(a) Structures of TSRecord after filtering-out irrelevant processes. The activity is mostly periodic due to the nature of the application and the selected actors to compute the structures. The structures involving the process `46 (usb-storage)` have been highlighted

.



(b) Dominants structure in the trace. It involves the interrupts related to the network.

Figure 9.4: Structures computed on the TSRecord trace.

However, *pdflush* has a particular behavior: it locks the *page cache* until all the data are correctly written. This is mandatory to avoid any kind of conflict with other processes that invoke `write`.

The structures can confirm or infirm this analysis. On Figure 9.4, we can see that the vast majority of the activity is periodic with the predominance of the white color. This is normal considering the task performed by the TSRecord application and the remaining actors to compute the structures. Figure 9.4b shows that the dominant structures are related to the network activity. The outline view confirms it and this is also in adequation with the fact that many interrupts occur while receiving data over the network. By highlighting the structures involving the process `46 (usb-storage)`, that appear in red on Figure 9.4, it clearly shows that its behavior is periodic but some perturbations occurred so that it remains inactive for large periods of time. (Note that the structure diagram shows only one type of event: a context switch. This is due to the fact that no tracepoint have been placed in the code of `46 (usb-storage)` and only the moment when this process has been

scheduled are available in this trace). It confirms the analysis made using TraceViz.

Bug resolution With their own tools, the software developers at STMicroelectronics have found the information. We explain here how they solved the problem. TraceViz and the structures visualization have both shown large periods of inactivity of the process executing the system call `write` to store the data of the video. This knowledge is enough to find the root of the bug. It comes from the behavior of *pdflush* explained above that locks the *page cache* while writing physically the data to the storage. This kernel thread is called every 5 milliseconds. 5 milliseconds of video decoding generate a volume of data that can potentially take some time. We saw that *pdflush* locks the *page cache* when writing the data. During this period, the thread `46 (usb-storage)` is waiting for the lock to be release when calling *write*, blocking the complete decoding chain. In this situation, the packets keep arriving on the Ethernet port but are not dequeued from the network buffers that can saturate and drop some packets, causing some data missing in the recorded video.

The resolution of this bug was quite simple since modifying the scheduling period of *pdflush* from 5 milliseconds to a shorter was enough to solve the problem. It also confirms that the analysis made with TraceViz and the structures helped to find the correct root of the bug.

9.4 Conclusion

In this chapter, we have presented an example of integration of the different works introduced in the previous chapters. Based on this integration, we defined an example of methodology to analyze execution traces that consisted in using TraceViz to filter the data and begin the analysis before using the visual analytic tool to finish the analysis. To illustrate this possible integration, we described a use case of the analysis of an execution trace that happened at STMicroelectronics. However, instead of describing how the developers have used our tools to solve the problem, we made the analysis ourselves. We demonstrated that the methodology we have chosen is efficient for this use case and that TraceViz and the structures bring different insights of the data.

However, in some cases, using the visual analytic tool before TraceViz can be more pertinent. We have shown here a simple example of integration to highlight the complementarity of our tools but a field study with the software engineers needs to be conducted to define the best workflow and integration.

Chapter 10

Conclusion

Contents

10.1 Contributions 135

10.2 Future Work 137

Multimedia devices are omnipresent in our life. Over the last years, their evolution has been impressive and quick. We have now thin smartphones, tablets and powerful set-top boxes that deliver a huge computational power while keeping a low energy consumption to optimize their battery life. All of this would have been impossible without a very competitive market composed of huge semiconductor companies able to produce new generations of embedded systems in a very short period of time.

The time-to-market of a product has to be the shortest possible with systems always more sophisticated. The software layer is nowadays extremely complex and efficient tools are required to keep efficient the development process.

Focusing on multimedia applications, QoS properties have to be satisfied in order to guarantee a smooth audio and video decoding. However, the software is now composed of several different bricks, integrated together at the end of the development process. Temporal bugs appear at this moment and often come from synchronization or communication problems between the different software components. Fixing these bugs is done with execution traces. However, without efficient tools, analyzing traces has always been tedious. Moreover, as the size of the traces increases when debugging modern systems the existing debugging tools are reaching their limits making the debugging process almost impossible to achieve in a reasonable time without tools able to work with huge traces.

10.1 Contributions

During this study, we have proposed three main contributions in information visualization domains, visual analytics and contributed to the STMicroelectronics indus-

trial activities.

10.1.1 A Smooth Visualization Technique for Time Series

In Chapter 6, we focused on a novel interactive visualization technique based on the pixels; the Slick Graphs. The goal of this doctoral work was to propose novel debugging tools for execution traces relying on visualization techniques. Therefore, having a precise visualization for time series suitable for a usage in a scientific and industrial context was mandatory.

Slick Graphs bin the data into small time intervals that correspond to the pixels. Next, our technique smooths the values using a kernel convolution and encodes the information during the smoothing into the alpha channel of each pixel. We explained our technique in Section 6.4. We have demonstrated in Section 6.5 the efficiency of this techniques compared to the existing algorithm used in many visualization. The results show that Slick Graphs are significantly faster than Stream Graphs (+62%) and more accurate (+48%) for tasks such as finding a local extrema when working with periodic data. Finally, we explained how our smoothing algorithm can be applied to other visualization for time-oriented data to improve their accuracy (Section 6.6).

10.1.2 A Visualization Framework for Execution Traces

In Section 3.2, we studied the existing visualization tools for execution traces and have found a gap between the techniques providing a too summarized overview versus the techniques that visualize too many details making the exploration of large traces tedious. This motivated the work described in Chapter 7 where we have introduced a visualization framework for execution traces: TraceViz. TraceViz is a trade off between an overview too general to be useful and a view with too much details so that the users become overwhelmed by the amount of information represented on top of a slow trace exploration (Section 7.4). We have explained how developers can visually detect temporal and behavioral patterns in the trace (Section 7.5).

We also have built this tool on a new back-end based on HDF5 that provides performances suitable for interactive browsing large execution traces (Section 7.2). In particular, we have conducted benchmarks against the traditional back-end used in trace analysis tools, SQLite. The results show that the TraceViz back-end provides much better performance for both input and output operations.

TraceViz has been integrated into the STMicroelectronics debugging toolchain (STPTK) and has been released to the STMicroelectronics software developers (Section 7.6). Since then, a new version of TraceViz is released with each version of STPTK. Simultaneously, TraceViz has been integrated into the SoC-Trace project on the FrameSoC open source infrastructure for traces.

10.1.3 Discovering Hidden Structures

We explained our third contribution in Chapter 8. We have proposed a novel visual analytics method to visualize hidden structures in the execution traces. To achieve this, we based our approach on data mining techniques to compute the different patterns on the trace (Section 8.3) after having introduced the notion of *structure* (Section 8.2). We introduce a new visualization technique to represent them in a comprehensive way (Section 8.4). By doing so, hidden structures become apparent to reveal the underlying behavior of the application. The main regime of the execution as well as the perturbations can be visually spotted very quickly. We have illustrated the relevance of our approach with different types of data. First, we showed that analyzing execution traces with our approach enables the developers to gain meaningful insights, usually hidden (Section 8.5). Second, we adapted our algorithm to work on a broader type of data. We took two examples: the software repositories data and texts. In both cases, our method is able to clearly visualize the underlying structures of the data.

10.2 Future Work

The analysis of execution traces and more broadly of computer logs and time-oriented data is a rich area with many opportunities, not only in different research domains but also in the industrial field. Following the work done in this thesis, we believe the following perspectives are worth to be investigated:

Classification of actors As discussed in Chapter 7, for each actor in the trace, there is a corresponding time series. In TraceViz, actors are organized according to pre-defined categories given by the domain area. In addition of using this actor organization, classifying the different event producers according to a hierarchical clustering can bring meaningful insights and accelerate the debugging process. Different machine learning techniques exist to perform such classification such as the approach proposed by Khah et al [Soheily-Khah et al., 2016]. Dendrogramix is a interactive visualization technique for hierarchical clustering [Blanch et al., 2015] and investigating how it can be integrated to TraceViz provides interesting research questions in information visualization.

Anytime rendering for visualization of huge time series We have explained that one of the challenge to analyze execution traces comes from the large amount of data. We have developed a fast back-end based on HDF5 and couple it with search algorithms to achieve interactive response time. This is only a part of the solution to a larger problem that is the visualization of large datasets. At some point, any algorithm and back-end will struggle to give a result in a 10 milliseconds time frame with large-scale data. Instead of waiting to have an exact results expensive to result, a different approach consists in perform early rendering with partial results. The

first studies show that it gives promising results with the analysts being able to work on their data from incremental rendering [Fisher et al., 2012]. A PhD study is currently undertaken by Ali Jabbari, since October 2015, aiming to study this approach for large time series[1].

Studying the encoding of a data variable into several visual attributes The norm in visualization is to map each data variable onto a unique visual attribute. For instance, in Slick Graphs (Chapter 6, we encoded one data variable, the value of the bin in the histogram, into two different visual attributes, the height of the bar and the luminance channel of the pixel). It raises a more general visualization research question: how to encode one data attribute into several visual attributes to convey information hardly understandable otherwise?

Improving the structures computation We have introduced a method to compute the structures in computer logs in Section 8.3. We identified two different approaches to investigate. The first one is to integrate existing data mining algorithm such as Clospan [Yan et al., 2003] to compute the frequent itemsets and PerMiner [Lopez Cueva et al., 2012] for the periodic patterns. It would return more precise results but it may not improve information conveyed by the visual rendering that shows an overview of the structures. It would also induce a much long computation time and the impact on the tool would need to be analyzed. The second interesting improvement is to implement the core of our method, the SOG algorithm, using GPGPU techniques. The computation time would be significantly shorter [Kouzinopoulos et al., 2015]. It would benefit to the developers to analyze more quickly the trace by further integrating the structures with the visualization tools, a step closer to leverage the usage of data mining algorithm.

Investigating more complex structures As discussed in Section 8.2, we define the structure as being composed of itemsets, sequences and periodic sequences. In fact, the concept of structure can be extended to other types of patterns. Another type of data structure can include a graph for the study of structures in logs of dynamic networks. For instance, this could be used to visualize the evolution of structures in social networks such as Twitter or Facebook. The main challenge relies in the mapping of a larger number of parameters of a structure onto visual attributes while having an uncluttered rendering to keep visualizing such structure "at a glance".

[1] `http://www.theses.fr/s136307`

Bibliography

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *Proceedings of the* 20^{th} *International Conference on Very Large Data Bases, VLDB*, pages 487–499.

Aigner, W., Miksch, S., Muller, W. an Schumann, H., and Tominski, C. (2007). Visualizing time-oriented data - a systematic view. *Computers & Graphics*, 31:401–409.

Aigner, W., Miksch, S., and Schumann, H. andd Tominski, C. (2011). *Visualization of Time-Oriented Data*. Springer Verlag, London, UK.

Andrienko, N. and Andrienko, G. (2005). *Exploratory Analysis of Spatial and Temporal Data: A Systematic Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Apple (Accessed on May 9^{th}, 2016). Av foundation. `https://developer.apple.com/av-foundation/`.

Architects, T. (Accessed on May 8^{th}, 2016). Btf specification. `https://wiki.eclipse.org/images/e/e6/TA_BTF_Specification_2.1.3_Eclipse_Auto_IWG.pdf`.

ARM (Accessed on April 8^{th}, 2016a). Arm ds-5 development studio. `http://ds.arm.com/ds-5`.

ARM (Accessed on May 7^{th}, 2016b). Cortex-a9 processor. `http://www.arm.com/cortex-a9.php`.

ARM (Accessed on May 7^{th}, 2016c). Mali-400 gpu. `http://www.arm.com/products/multimedia/mali-gpu/ultra-low-power/mali-400.php`.

Bach, B., Shi, C., Heulot, N., Madhyastha, T., Grabowski, T., and Dragicevic, P. (2015). Time curves: Folding time to visualize patterns of temporal evolution in data. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):559–568.

Bar, M. and Neta, M. (2006). Humans prefer curved objects. *Psychological Science*, 17(8):645–648.

Berry, L. and Munzner, T. (2004). Binx: Dynamic exploration of time series datasets across aggregation levels. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 215–216.

Bertini, E., Hertzog, P., and Lalanne, D. (2007). Spiralview: Towards security policies assessment through visual correlation of network ressources with evolution of alarms. In *Visual Analytics Science and Technology, 2007. VAST 2007. IEEE Symposium on*, pages 139–146.

Bezemer, C. P., Pouwelse, J., and Gregg, B. (2015). Understanding software performance regressions using differential flame graphs. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 535–539.

Blanch, R., Dautriche, R., and Bisson, G. (2015). Dendrogramix: a hybrid tree-matrix visualization technique to support interactive exploration of dendrograms. In *Proceedings of the* 8^{th} *IEEE Pacific Visualization Symposium (PacificVis 2015)*, pages 31–38.

Bloch, M., Byron, L., Carter, S., and Cox, A. (2008). The ebb and flow of movies: Box office receipts 1986-2007. `http://nytimes.com`.

Bothorel, G., Serrurier, M., and Hurter, C. (2013). Visualization of frequent itemsets with nested circular layout and bundling algorithm. In *Advances in Visual Computing -* 9^{th} *International Symposium*, pages 396–405.

Bril, R. J., Hentschel, C., Steffens, E. F. M., Gabrani, M., van Loo, G., and Gelissen, J. H. A. (2001). Multimedia qos in consumer terminals. In *Signal Processing Systems, 2001 IEEE Workshop on*, pages 332–343.

Buono, P., Aris, A., Plaisant, C., Khella, A., and Shneiderman, B. (2005). Interactive pattern search in time series.

Buono, P. and Simeone, A. L. (2008). Interactive shape specification for pattern search in time series. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 480–481.

Byron, L. (2006). Listening history. `http://www.leebyron.com/what/lastfm`.

Byron, L. and Wattenberg, M. (2008). Stacked graphs: Geometry & aesthetics. *IEEE Transactions on Visualization and Computer Graphics*, 14:1245–1252.

Card, S. K., Robertson, G. G., and Mackinlay, J. D. (1991). The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 181–186. ACM.

Carlis, J. V. and Konstan, J. A. (1998). Interactive visualization of serial periodic data. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA. ACM.

Carmichael, C. L. and Leung, C. K.-S. (2010). Closeviz: Visualizing useful patterns. In *Proceedings of the ACM SIGKDD Workshop on Useful Patterns*, pages 17–26.

Carvalho de Melo, A. (2010). The new linux 'perf' tools.

Castro, M., Georgiev, K., Marangozova-Martin, V., Mehaut, J.-F., Fernandes, L. G., and Santana, M. (2011). Analysis and tracing of applications based on software transactional memory on multicore architectures. In *Parallel, Distributed and Network-Based Processing (PDP), 2011* 19^{th} *Euromicro International Conference on*, pages 199–206.

Chassin de Kergommeaux, J. (2000). Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10).

Cheng, J., Ke, Y., and Ng, W. (2008). A survey on algorithms for mining frequent itemsets over data streams. *Knowledge of Information Systems*, 16:1–27.

Cho, M., Kim, B., Bae, H. J., and Seo, J. (2014). Stroscope: Multi-scale visualization of irregularity measured time-series data. *IEEE Transactions on Visualization and Computer Graphics*, 20:808–821.

Cleveland, W. (1993). *Visualizing Data*. Hobart Press.

Compass, E. T. (Accessed on April 8^{th}, 2016). `http://tracecompass.org`.

Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J., and van Deursan, A. (2007a). Understanding execution traces using massive sequence view and hierarchical edge bundles. In 15^{th} *IEEE International Conference on Program Comprehension (ICPC '07)*, pages 49–58.

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., and van Wijk, J. J. (2007b). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81:2252–2268.

Dork, M., Gruen, D., Williamson, C., and Carpendale, S. (2010). A visual backchannel for large-scale events. *IEEE Transactions on Visualization and Computer Graphics*, 16:1129–1138.

Dosimont, D., Pagano, G., Huard, G., and Marangozova-Martin, V. (2014). Efficient analysis methodology for huge application traces. In *International Conference on High Performance Computing & Simulation*, pages 951–958.

EfficiOS (Accessed on May 12^{th}, 2016a). Babeltrace. `http://diamon.org/babeltrace`.

EfficiOS (Accessed on May 12^{th}, 2016b). Common trace format. `http://www.efficios.com/ctf`.

Fails, J. A., Karlson, A., Shahamat, L., and Shneiderman, B. (2006). A visual interface for multivariate temporal data: Finding patterns of events across multiple histories. In *2006 IEEE Symposium on Visual Analytics Science and Technology*, pages 167–174.

Few, S. (2008). Time on the horizon. `http://perceptualedge.com/articles/visual_business_intelligence/time_on_the_horizon.pdf`.

Fisher, D., Popov, I., Drucker, S. M., and Schraefel, M. C. (2012). Trust me, i'm partially right: Incremental visualization lets analysts explore large datasets faster. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2012)*, pages 1673–1682.

Fittkau, F., Waller, J., Wulf, C., and Hasselbring, W. (2013). Live trace visualization for comprehending large software landscapes: The explorviz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4.

Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM.

Ftrace (Accessed on April 8^{th}, 2016). `http://elinux.org/Ftrace`.

Fuchs, J., Fischer, F., Mansmann, F., Bertini, E., and Isenberg, P. (2013). Evaluation of alternative glyph designs for time series data in a small multiple setting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3237–3246.

Gantt, H. L. (1913). *Work, Wages, and Profits*. New York: The Engineering magazine co.

Goethals, B., Moens, S., and Vreeken, J. (2011). MIME: a framework for interactive visual pattern mining. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 757–760.

Google (Accessed on May 9^{th}, 2016). Google hangouts. `http://hangouts.google.com`.

Gregg, B. (2016a). The flame graph. *Queue*, 14(2).

Gregg, B. (2016b). The flame graph. *ACM Communications*, 59:48–57.

GStreamer (Accessed on May 9^{th}, 2016). `http://gstreamer.freedesktop.org`.

Hao, M., Dayal, U., Keim, D., and Schrek, T. (2005). Importance-driven visualization layouts for large time series data. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2005)*, pages 203–210.

Hao, M., Dayal, U., Keim, D., and Schrek, T. (2007). Multi-resolution techniques for visual exploration of large time-series data. In *Proceedings of the* 9^{th} *Join Eurographics (EuroVis)*, pages 27–34.

Hao, M., Marwah, M., Janetzko, H., Dayal, U., Keim, D., Patnaik, D., Ramakrishnan, N., and Sharma, R. (2012). Visual exploration of frequent patterns in multivariate time series. *Information Visualization*, 11:71–83.

Harris, R., L. (1999). *Information Graphics: A Comprehensive Illustrated Reference*. Oxford University Press.

Havre, S., Hetzler, E., and Nowell, L. (2000). Themeriver: Visualizing theme changes over time. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 115–124.

Havre, S., Whitney, P., and Nowell, L. (2002). Themeriver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics*, 8:9–20.

Heath, M. T. and Etheridge, J. A. (1991). Visualizing the performance of parallel programs. *Software, IEEE*, 8(5):29–39.

Heer, J. and Agrawala, M. (2006). Multi-scale banking to 45 degrees. *IEEE Transactions on Visualization and Computer Graphics*, 12:701–708.

Heer, J., Kong, N., and Agrawala, M. (2009). Sizing the horizon: The effects of chart size and layering on the graphical perception of time series visualizations. In *ACM Human Factors in Computing Systems (CHI)*.

Hochheiser, H. and Shneiderman, B. (2004). Dynamic query tools for time series data sets: Timebox widgets for interactive exploration. *Information Visualization*, 3:1–18.

Holz, C. and Feiner, S. (2009). Relaxed selection techniques for querying time-series graphs. In *Proceedings of the* 22^{nd} *Annual ACM Symposium on User Interface Software and Technology*, pages 213–222.

Iegorov, O., Leroy, V., Termier, A., Mehaut, J.-F., and Santana, M. (2015). Data mining approach to temporal debugging of embedded streaming applications. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 167–176. IEEE Computer Society.

Imrich, P., Mueller, K., Imre, D., Zelenyuk, A., and Zhao, W. (2003). Interactive poster: 3d themeriver. *Poster Compendium of IEEE Symposium on Information Visualization*.

Isaacs, K. E., Bremer, P. T., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., and Hamann, B. (2014a). Combining the communication hairball: Visualizing large-scale parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics, Proceedings of InfoVis'14*, 20(12):2349–2358.

Isaacs, K. E., Giménez, A., Bhatele, A., Schulz, M., Hamann, B., and Bremer, P. T. (2014b). State of the art of performance visualization.

Isaacs, K. E., Giménez, A., Bhatele, A., Schulz, M., Hamann, B., and Bremer, P. T. (Accessed on May 13^{rd}, 2016). Living digital library of state of the art of performance visualization. `http://idav.ucdavis.edu/~ki/STAR/`.

Javed, W. and Elmqvist, N. (2010). Stack zooming for multi-focus interaction in time series data visualization. In *IEEE Pacific Visualization Symposium (PacificVis)*, pages 33–40.

Javed, W. and Elmqvist, N. (2013). Stack zooming for multi-focus interaction in skewed-aspect visual spaces. *IEEE Transactions on Visualization and Computer Graphics*, 19:1362–1374.

Javed, W., McDonnel, B., and Elmqvist, N. (2010). Graphical perception of multiple time series. *IEEE Transactions on Visualization and Computer Graphics*, 16:927–934.

Keim, D. A., Schneidewind, J., and Sips, M. (2008). Fp-viz: Visual frequent pattern mining. In *IEEE Symposium on Information Visualization (InfoVis 05)*.

Kincaid, R. (2010). Signallens: Focus+context applied to electronic time series. *IEEE Transactions on Visualization and Computer Graphics*, 16:900–907.

Kincaid, R. and Lam, H. (2006). Line graph explorer: Scalable display of line graphs using focus+context. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 404–411.

Kouzinopoulos, C. S. and Margaritis, K. G. (2014). Multiple pattern matching: Survey and experimental results. *Neural, Parallel, and Scientific Computation*, 22:563–593.

Kouzinopoulos, C. S., Michailidis, P. D., and Margaritis, K. G. (2015). Multiple string matching on a gpu using cuda. *Scalable Computing: Practice and Experience*, 16(2):121–137.

Krishnakumar, R. (2005). Kernel korer: Kprobes - a kernel debugger. *Linux Journal*, 2005(133).

Krstajic, M., Bertini, E., and Keim, D. (2011). Cloudlines: Compact display of event episodes in multiple time-series. *IEEE Transactions on Visualization and Computer Graphics*, 17:2432–2439.

Kruskal, J. B. and Landwehr, J. M. (1983). Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37:162–168.

Krzywinski, M., Schein, J., Birol, I., Connors, J., Gascoyne, R., Horsman, D., Jones, S. J., and Marra, M. A. (2009). Circos: An information aesthetic for comparative genomics. *Genome Research*, 19:1639–1645.

Lagraa, S., Termier, A., and Pétrot, F. (2014). Scalability bottlenecks discovery in mpsoc platforms using data mining on simulation traces. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6.

Lagraa, S., Termier, A., and Pétrot, F. (2012). Automatic congestion detection in mpsoc programs using data mining on simulation traces. In *2012, 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 64–70.

Lam, H., Munzner, T., and Kincaid, R. (2007). Overview use in multiple visual information resolution interfaces. *IEEE Transactions on Visualization and Computer Graphics*, 13:1278–1285.

Lamarche-Perrin, R., Schnorr, L. M., and Vincent, J.-M. (2014). Evaluating trace aggregation for performance visualization of large distributed systems. In *Proceedings of the 2014 IEEE Internation Symposium on Performance Analysis of Systems and Software*.

Leung, C. K.-S. and Carmichael, C. L. (2009). Fpviz: A visualizer for frequent pattern mining. In *Proceedings of the ACM SIGKDD Workshop on Visual Analytics and Knowledge Discovery: Integrating Automated Analysis with Interactive Exploration*, pages 30–39.

Leung, C. K.-S., Irani, P. P., and Carmichael, C. L. (2008a). Fisviz: A frequent itemset visualizer. In *Advances in Knowledge Discovery and Data Mining*, pages 644–652.

Leung, C. K.-S., Irani, P. P., and Carmichael, C. L. (2008b). Wifisviz: Effective visualization of frequent itemset. In *2008 8^{t}h IEEE Conference on Data Mining*, pages 875–880.

Lin, J., Keogh, E., and Lonardi, S. (2005). Visualizing and discovering non-trivial patterns in large time series databases. *Information Visualization*, 4:61–82.

Lin, J., Keogh, E., Lonardi, S., Lankford, J. P., and Nystrom, D. M. (2004). Visually mining and monitoring massive time series. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 460–469.

Lopez Cueva, P., Bertaux, A., Termier, A., Méhaut, J.-F., and Santana, M. (2012). Debugging embedded multimedia application traces through periodic pattern mining. In *2012 International Conference on Embedded Software (EMSOFT)*, pages 595–602.

LTTng (Accessed on April 8^{th}, 2016). `http://lttng.org`.

McLachLan, P., Munzner, T., Koutsofios, E., and North, S. (2008). Liverac: Interactive visual exploration of system-management time-series data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1483–1492.

Microsoft (Accessed on May 9^{th}, 2016). Media foundation. `https://msdn.microsoft.com/en-us/library/ms694197.aspx`.

Miller, R. B. (1968). Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, pages 267–277. ACM.

Mooney, C. H. and Roddick, J. F. (2013). Sequential pattern mining – approaches and algorithms. *ACM Computer Survey*, 45:19:1–19:39.

Munzner, T., Kong, Q., Ng, R. T., Lee, J., Klawe, J., Radulovic, D., and Leung, C. K. (2005). Visual mining of power sets with large alphabets. Department of Computer Science, The University of British Columbia, Vancouver, BC, Canada.

Netflix (Accessed on May 9^{th}, 2016). `http://netflix.com`.

Osmari, D. K., Vo, H. T., Silva, C. T., Comba, J. L. D., and Lins, L. (2014). Visualization and analysis of parallel dataflow execution with smart traces. In 27^{th} *Conference on Graphics, Patterns and Images (SIBGRAPI)*.

Pagano, G., Dosimont, D., Huard, G., and Marangozova-Martin, V. (2013). Trace management and analysis for embedded systems. In *Proceedings of the IEEE* 7^{th} *International Symposium on Embedded Multicore SoCs*.

Perin, C., Vernier, F., and Fekete, J. D. (2013). Interactive horizon graphs: Improving the compact visualization of multiple time series. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3217–3226, New York, NY, USA. ACM.

Phonon (Accessed on May 9^{th}, 2016). `http://phonon.kde.org`.

Plaisant, C., Carr, D., and Shneiderman, B. (1995). Image-browser taxonomy and guidelines for designers. *IEEE Software*, 12(2):21–32.

Playfair, W. (1786). *The Commercial and Political Atlas*. London.

Prada-Rojas, C., Riss, F., Raynaud, X., De Paoli, S., and Santana, M. (2009). Observation tools for debugging and performance analysis of embedded linux applications. In *Conference on System Software, SoC and Silicon Debug-S4D*.

Reijner, H. (2008). The development of the horizon graph. `http://www.stonesc.com/Vis08_Workshop/DVD/Reijner_submission.pdf`.

Ryall, K., Lesh, N., Lanning, T., Leigh, D., Miyashita, H., and Makino, S. (2005). Querylines: Approximate query for visual browsing. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pages 1765–1768.

Saito, T., Miyamura, H. N., Yamamoto, M., Saito, H., Hoshiya, Y., and Kaseda, T. (2005). Two-tone pseudo coloring: Compact visualization for one-dimensional data. In *IEEE Symposium on Information Visualization*, pages 173–180. IEEE Computer Society.

Salmela, L., Tarhio, J., and Kytojoki, J. (2006). Multi-pattern matching with q-grams. *Journal of Experimental Algorithmics*, 11:1–19.

Schneiderman, B. (1992). Tree visualization with tree-maps: 2d space-filling approach. *ACM Trans. Graph.*, 11(1).

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualization. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343.

Skype (Accessed on May 9^{th}, 2016). `http://skype.com`.

Soheily-Khah, S., Douzal, A., and Gaussier, E. (2016). Generalized k-means-based clustering for temporal data under weighted and kernel time wrap. *Pattern Recognition Letters*, 75:63–69.

STMicroelectronics (Accessed on May 13^{rd}, 2016a). Soc traces & profiling toolkit (stptk). `http://www.stlinux.com/devel/traceprofile/kptrace#STPTK`.

STMicroelectronics (Accessed on May 7^{th}, 2016b). Stih4 monaco series. `http://www2.st.com/content/st_com/en/products/digital-set-top-box-ics/uhd-set-top-box-processors.html?querycriteria=productId=SC2060`.

Stoffel, F., Fischer, F., and Keim, D. A. (2013). Finding anomalies in time-series using visual correlation for interactive root cause analysis. In *Proceedings of the 10^{th} Workshop on Visualization for Cyber Security*, pages 65–72.

Sullivan, G., Ohm, J. R., Han, W., and Wiegand, T. (2013). Overview of the high efficiency video coding (hevc standard). *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649 – 1668.

Thudt, A., Walny, J., Perin, C., Rajabiyazdi, F., MacDonald, L., Vardeleon, R., Greenberg, S., and Carpendale, S. (2016). Assessing the readibility of stacked graphs. In *Proceedings of Graphics Interface 2016*.

Tominksi, C., Schulze-Wollgast, P., and Schumann, H. (2005). 3d information visualization for time dependent data on maps. In *Ninth International Conference on Information Visualization (IV'05)*, pages 175–181.

Tominksi, C. and Schumann, H. (2008). Enhanced interactive spiral display. In *Proceedings of the Annual SIGRAD Conference, Special Theme: Interactivity*, pages 53–56.

Trümper, J., Döllner, J., and Telea, A. (2013). Multiscale visual comparison of execution traces. In 21^{st} *International Conference on Program Comprehension (ICPC)*, pages 53–62.

Trümper, J., Telea, A., and Döllner, J. (2012). Viewfusion: Correlating structure and activity views for execution traces. In *Theory and Practive of Computer Graphics*.

Trümper, J., Bohnet, J., and Döllner, J. (2010). Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the* 5^{th} *International Symposium on Software Visualization*, pages 133–142.

Tufte, E. R. (1986). *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA.

Twitter (Accessed on March 31^{st}, 2015). Visualization of station of the union 2015. `http://twitter.github.io/interactive/sotu2015`.

VideoLAN (Accessed on May 9^{th}, 2016). Vlc multimedia framework. `http://videolan.org`.

Vimeo (Accessed on May 9^{th}, 2016). `http://vimeo.com`.

Walker, J., Borgo, R., and Jones, M. W. (2016). Timenotes: A study on effective chart visualization and interaction techniques for time-series data. *IEEE Transactions on Visualization and Computer Graphics*, 22:549–558.

Walker, J. S., Jones, M. W., Laramee, R. S., Bidder, O. R., William, H. J., Scott, R., Shepard, E. L., and Wilson, R. P. (2015). Timeclassifier: A visual analytic system for the classification of multi-dimensional time series data. *The Visual Computer*, 31:1067–1078.

Wattenberg, M. (2001). Sketching a graph to query a time-series database. In *CHI'01 Extended Abstracts on Human Factors in Computing Systems*, pages 381–382.

Weber, M., Alexa, M., and Müller, W. (2001). Visualizing time series on spirals. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 7–13. IEEE Computer Society.

Wheeler, K. B. and Thain, D. (2010). Visualizing massively multithreaded applications with threadscope. *Concurrency and Computation: Practice and Experience*, 22:45–67.

Wiegand, T., Sullivan, G. J., Bjontegaard, G., and Luthra, A. (2003). Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576.

Wolf, W., Jerraya, A. A., and Martin, G. (2008). Multiprocessor system-on-chip (mpsoc) technology. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1701–1713.

Yan, X., Han, J., and Afshar, R. (2003). Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 166–177.

Yang, L. (2003). Visualizing frequent itemsets, association rules, and sequential patterns in parallel coordinates. In *Computational Science and Its Applications - ICCSA 2003*, pages 21–30.

Yang, L. (2005). Pruning and visualizing generalized association rules in parallel coordinates. *IEEE Transactions on Knowledge and Data Engineering*, 17:60–70.

Youtube (Accessed on May 9^{th}, 2016). `http://youtube.com`.

Zhao, J., Chevalier, F., and Balakrishnan, R. (2011a). Kronominer: Using multi-foci navigation for the visual exploration of time-series data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1737–1746.

Zhao, J., Chevalier, F., Pietriga, E., and Balakrishnan, R. (2011b). Exploratory analysis of time-series with chronolenses. *IEEE Transations on Visualization and Computer Graphics*, 17:2422–2431.

Part III

French Summary

Chapter 1

Introduction

Les dispositifs mobiles et les set-top box sont aujourd'hui de plus en plus performants et permettent de consommer de contenus multimédia de qualité croissante avec le temps tout en couvrant un panel d'utilisation toujours plus large comme jouer à des jeux video, naviguer sur Internet, etc. Tout ceci est possible grâce la puissance délivrée par les dernières générations de systèmes embarqués, nommés Multiple-Processor-System-on-Chip (MPSoC) intégrant dorénavant différentes puces dédiées à des tâches spécifiques. Un MPSoC embarque différents types de puces très hétérogènes comme un processeur (Central Process Unit, CPU), un processeur graphique (Graphic Process Unit, GPU) et aussi des processeurs spécialisés dédiés au décodage audio et video, gérant la connectivité du système et plusieurs types de capteurs tels qu'un accéléromètre, un gyroscope, un GPS, etc.

A chaque nouvelle génération, de meilleures performances ainsi qu'une consommation électrique plus modérée permettent aux constructeurs comme Apple et Samsung de développer des dispositifs innovants avec une meilleure expérience utilisateur. Une conséquence directe de la complexification des plateformes matérielles est une augmentation du temps et du coût de développement et de vérification de ces systèmes, qu'il est indispensable de minimiser pour les constructeurs afin de rester compétitif.

Les applications de décodage multimédia ont la particularité de devoir respecter des contraintes temps réels afin de garantir une lecture fluide du contenu. Par exemple, un film doit être décodé à 30 images par seconde. On distingue deux catégories de bogues pouvant impacter ces applications :

- **Les bogues fonctionnels** qui correspondent à un mauvais résultat logiciel
- **Les bogues temporels** qui apparaissent lorsqu'un résultat n'est pas retourné suffisamment rapidement.

A la différence des bogues fonctionnels qui peuvent être résolus à l'aide des débogueurs traditionnels, les bogues temporels requièrent une approche différente pour deux raisons. Premièrement, l'utilisation des débogueurs qui suspendent l'exécution de

l'application conduisent automatiquement à un non-respect des contraintes temporelles. Deuxièment, les bogues temporels apparaissent la plupart du temps en fin du cycle de développement, au moment de la phase d'intégration. En effet, la plupart sont dûs à un problème de communication ou de synchronisation entre différents composants logiciels, les rendant impossible à détecter avant la phase d'intégration. Pour cette raison, il est très fréquent que ce genre de bogues apparaissent après la livraison au client, mettant les développeurs sous pression d'un côté par le client, de l'autre par le fournisseur. Dans ce contexte, des outils de débogage efficaces sont primordiaux.

L'utilisation des débogueurs traditionnels étant inappropriée, la technique pour déboguer des applications multimedia est d'enregistrer dans un log, appelée *trace*, les évènements apparus au cours de l'exécution et d'analyser a posteriori le comportement du système avec la trace d'exécution. Chaque évènement de la trace contient plusieurs informations telles que la date à laquelle l'évènement a eu lieu, son type et l'entité qui l'a produit. Basé sur ces informations, les développeurs peuvent alors identifier la source potentielle des bogues temporels. Deux strategies sont alors possibles pour l'analyse de la trace:

- utiliser des techniques de fouille de données
- utiliser des techniques de visualisation de données

Avec la complexification des plateformes modernes, le nombre d'évènements générés devient très important rendant l'analyse des traces lente et fastidieuse. Dans ce travail de thèse, notre but est de proposer de nouveaux outils d'analyse de traces d'exécution faisant apparaitre clairement les comportements récurrents ainsi que les motifs temporels afin de réduire le temps d'analyse. Pour atteindre cet objectif, nous allons intégrer des techniques de fouille de données et des techniques de visualisation pour être capable de détecter les tendances ainsi que les motifs aberrants. Nous proposons trois contributions à travers cette thèse :

1. **Slick Graphs.** Une nouvelle technique de visualisation pour séries temporelles qui minimise les artefacts visuels au pixel près, contrairement aux techniques existantes, comme nous le montrons dans l'etat de l'art.

2. **TraceViz.** Un framework de visualisation de traces d'exécution interactif et multi-échelles. En particulier, nous proposons un nouveau système de stockage de traces permettant l'exploration interactive de traces volumineuses ainsi qu'une technique de visualisation pour naviguer interactivement dans la trace à différentes echelles.

3. **Visualisation des Structures.** Un nouvel outil d'analyse qui permet de visualiser les structures cachées dans les traces telles que des ensembles et séquence d'évènements et leur périodicité afin de gagner rapidement une compréhension approfondie des traces d'exécution.

La suite de cette thèse est composée de huit chapitres. Dans le chapitre 2, nous expliquons l'évolution des standards multimedia ainsi que des plateformes matérielles et logicielles. Nous continuons par décrire les travaux existants sur la visualisation de séries temporelles, de traces et de motif dans le chapitre 3, résumons les questions de recherche auxquelles nous nous intéressons dans le chapitre 4 et expliquons notre démarche scientifique dans le chapitre 5. Les Slick Graphs, TraceViz et l'outil d'analyse sont respectivement présentés dans les chapitres 6, 7 et 8. Nous finissons ce manuscrit avec une étude de cas illustrant la complementarité de nos travaux dans le chapitre 9.

Chapter 2

Contexte Industriel

Nous nous intéressons dans cette thèse à l'analyse de traces d'exécution pour applications multimédia sur systèmes embarqués. Les applications multimédia, concernant l'encodage et le décodage de contenus multimédia dans le cadre de ce travail, ont certaines particularités. D'une part, ces applications sont soumises à des contraintes temps réel et sont périodiques. Par exemple, lors du décodage d'une vidéo, l'application exécute de façon périodique les opérations pour décoder une image et doit être capable de décoder 30 images par seconde pour garantir une lecture fluide. D'autre part, ces applications tirent parti des plateformes matérielles modernes composées de plusieurs puces spécialisées, nécessitant plusieurs fils d'exécution parallèles.

Par conséquent, les méthodes de débogage classiques ne fonctionnent pas dans ce cadre. Les débogueurs typiquement utilisés interrompent l'exécution et permettent aux développeurs d'étudier l'état des différents composants logiciels et de continuer l'exécution au pas à pas. Lors de la pause de l'exécution, les propriétés temps réel de l'application ne sont plus respectées, empêchant l'étude de potentiels problèmes temporels.

Une solution proposée par l'industrie est l'utilisation de traces d'exécution. Cette technique de débogage consiste à laisser l'exécution se dérouler et à enregistrer tous les évènements ayant eu lieu sur le système, au niveau matériel (interruptions), du système d'exploitation (changement de contexte, appels systèmes, etc.) et des applications (appels de fonction), dans un fichier appelé *trace d'exécution*. Une fois l'exécution terminée, les développeurs étudient son déroulement à travers l'analyse de la trace.

La problématique actuelle autour des traces est l'augmentation significative du nombre d'évènements générés par les systèmes modernes. Ceci est la conséquence d'une part de la complexification des plateformes matérielles impliquant une complexification de la couche logicielle afin de maximiser l'usage du matériel. Les systèmes modernes génèrent de l'ordre de 10^6 évènements par minute avec plusieurs centaines de processus actifs. Les outils actuels d'analyse de traces ne passent pas à l'échelle, ne permettant pas de faire une analyse efficace de traces volumineuses.

La problématique actuelle consiste à proposer de nouveaux outils permettant l'étude des traces d'exécution générées sur les systèmes modernes. Nous travaillons sur deux axes autour de cette problématique : la proposition de nouveaux outils de visualisation de traces d'exécution et l'intégration de résultats de technique de fouille de données pour la découverte automatique de motifs dans les traces.

Chapter 3

Etat de l'Art

Visualisation de Séries Temporelles

Dans ce chapitre, on s'intéresse à trois corpus de travaux : la visualisation de séries temporelles, la visualisation de traces d'exécution et enfin la visualisation de motifs.

Nous avons vu que les traces d'exécution contiennent un grand nombre d'évènements et un grand nombre d'*acteurs* (processus, interruptions, etc.). Une trace d'exécution peut se modéliser par une série temporelle, une suite d'évènement triée par ordre chronologique. De nombreux travaux se sont focalisés sur la représentation de la dimension temporelle et plusieurs approches ont été proposées.

La solution la plus classique et répandue consiste à encoder la dimension temporelle sur l'axe horizontal de la visualisation, en disposant le premier évènement chronologique le plus à gauche. La plupart de ces techniques de visualisation utilisant une représentation linéaire du temps sont des dérivés du *line graph* et ont pour but de corriger leurs défauts en y intégrant des interactions et en enrichissant leurs représentations visuelles.

Une autre solution consiste à encoder le temps sous forme cyclique, selon une disposition en spirale ou circulaire. Ces différentes représentations du temps conduisent à des visualisations uniques, spécifiques à des tâches bien précises mais peuvent nécessiter un temps d'apprentissage à l'utilisateur. La représentation cyclique du temps est performante pour la découverte de motifs périodiques dans les données. En revanche, pour les tâches plus basiques, comme la lecture et la comparaison de valeurs, la courbure de l'axe du temps ajoute de la complexité et ralentit le processus de compréhension des données. Dans cette configuration, la difficulté pour comparer les valeurs a pour conséquence de rendre la visualisation de plusieurs séries une tâche compliquée.

Visualiser de multiples séries temporelles est un domaine de recherche actif. Deux stratégies ont été identifiées pour la gestion de l'espace écran : les *split-screen* et les *shared-screen* techniques [Javed et al., 2010]. Les techniques *split-screen* reposent sur le principle de *small multiples* introduit par Tufte [Tufte, 1986]. Elles consistent à diviser l'espace écran S and N régions de taille S/N, une région par série

temporelle. Les techniques *shared-screen* ont une approche différente : les séries temporelles sont toutes représentées dans le même espace et sont différenciées en utilisant la couleur. Les techniques *split-screen* sont plus performantes pour lire des valeurs globales tandis que les techniques *shared-screen* sont meilleures pour étudier de façon locale les séries temporelles [Javed et al., 2010].

Nous avons vu dans le chapitre précédent que les traces d'exécution contiennent un grand nombre d'évènements et vont continuer à grossir dans le futur. Pour une analyse efficace de telles traces, il est nécessaire d'avoir des outils de visualisation capables de gérer de telles quantités de données. L'étude de l'existant montre qu'il existe deux méthodes pour visualiser de larges séries temporelles : les techniques multi-foci et celles fournissant une vue globale et détaillées (overview+details). La limitation principales de ces différentes approches réside néanmoins dans le passage à l'échelle par rapport au nombre de séries temporelles à visualiser. Les techniques existantes reposent sur des variantes de graphes qui deviennent rapidement sur-chargées quand plusieurs séries sont à visualiser. Dans le contexte des traces d'exécution, avant d'être capable d'étudier en profondeur le comportement d'un acteur, les développeurs doivent d'abord comprendre le comportement global du système et donc visualiser plusieurs séries temporelles. Les techniques existantes pour l'exploration d'une grande série temporelles deviennent pertinentes uniquement pour étudier le comportement d'un acteur en particulier.

Les techniques de visualisation d'une grande collection de séries temporelles ont aussi été largement investiguées. Il résulte que bien qu'elles proposent des solutions efficaces pour analyser de façon globale les séries, les techniques existantes manquent d'interactions pour explorer la dimension temporelle, nécessaire lorsque les séries comportent un grand nombre d'évènements.

Une des tâches les plus importantes lors de la visualisation de séries temporelles est l'identification de motifs fréquents. De nombreuses techniques d'analyses visuelles ont été proposées [Aigner et al., 2007]. Certaines techniques reposent sur des algorithmes qui calculent de façon automatique les motifs. Leur limitation réside dans le fait qu'elles nécessitent la plupart du temps un utilisateur expert capable de paramétrer correctement l'algorithme et de comprendre les résultats. D'autres approches prennent en entrée un motif défini par l'utilisateur et cherchent à trouver les différentes instances de ce motif dans les données. Cette approche n'est pas applicable dans le cas de l'analyse de traces d'exécution: les développeurs ne connaissent pas a priori les motifs à chercher qui pourraient correspondre à des comportements anormaux pendant l'exécution.

Visualisation de Traces d'Exécution

Les travaux existants sur la visualisation de traces d'exécution sont catégorisables en deux familles : les techniques proposant une vue globale de l'exécution et les techniques permettant l'analyse fine de l'exécution.

Les visualisations existantes qui fournissent une vue globale sur l'exécution reposent sur plusieurs stratégies en utilisant soit un algorithme d'aggrégation, des treemaps ou des statistiques spécifiques au domaine métier. Bien que les informations montrées à l'utilisateur soient pertinentes, leur manque d'interactions et une vue trop aggrégées ont pour conséquence de ne pas fournir suffisamment d'information aux développeurs pour commencer efficacement l'analyse de l'exécution et trouver la source de bogues temporels qui impliquent potentiellement plusieurs acteurs.

D'autres travaux proposent des visualisations détaillées de traces d'exécution et permettent aux développeurs de comprendre en détails les séquences d'évènements sur un intervalle de temps. Cependant, un problème de passage à l'échelle apparaît avec les traces contenant un grand nombre d'évènements. D'une part, des problèmes d'aliasing peuvent apparaître. D'autre part, dans le cas d'applications multimedia, des motifs temporels peuvent apparaître à plusieurs échelles temporelles mais les techniques existantes ne permettent pas de les visualiser dû à l'absence d'aggrégation des données ou parce que la dimension temporelle n'est pas explicitement encodés, cachant la synchronisation entre différents acteurs.

Un gap important existe entre les techniques proposant une vue aggrégée de la trace trop abstraite pour commencer efficacement l'analyse et celles proposant une vue détaillée pertinente pour une analyse locale des comportements mais fastidieuse pour la découverte de motifs à différentes échelles plus globales. En complément de ces approches, les développeurs ont besoin d'un outil intéractif multi-échelle fournissant une vue globale de l'exécution mais gardant suffisamment de détails pour filtrer de façon efficace les données avant de commencer une analyse plus fine d'une zone particulière de la trace.

Visualisation de Motifs

Les techniques de visualisation de motifs existantes cherchent à montrer l'ensemble des motifs fréquents appartenant à un domaine particulier (itemset, séquences, etc.) et à fournir des interactions permettant leur exploration. Elles nécessitent souvent un utilisateur expert. Dans le chapitre 6, nous proposons une approche différente: nous considérons les motifs courts (de taille fixe avec seulement 2 ou 3 items) mais nous nous concentrons sur les différentes structures qui organisent ces items: les ensembles, les séquences et leur périodicité. Notre technique de visualisation est construite à partir de cette idée : les structures sont l'information principale montrée par notre technique, évitant ainsi une explosion combinatoire des motifs à représenter mais aussi en visualisant des informations qui demeurent souvent cachées dans les techniques existantes.

Chapter 4

Challenges autour du Débogage de Traces

Représentation imprécises des séries temporelles

L'étude des techniques de visualisation de données temporelles a montré un manque de techniques fournissant une grande précision visuelle, obligatoire dans un contexte scientifique ou industriel. En particulier, nous avons observé un manque de travaux sur le rendu d'une série temporelle à proprement parler qui répondraient à la question suivante : *comment minimiser les artéfacts visuels tout en produisant une représentation lisse de séries temporelles ?*

Un rendu imprécis peut introduire des artéfacts visuels et induire l'utilisateur en erreur dans ses conclusions. Dans le contexte de cette thèse, une mauvaise représentation des traces pourrait mener les développeurs à modifier une mauvaise partie du code de l'application ou être bloqués dans leur processus de débogage. Trouver un bogue temporel dans une trace d'exécution comportant un nombre important d'évènements est une tâche fastidieuse. Il est ici primordial que les visualisations montrent les différents aspects du comportement de l'application de façon la plus précise possible. Ceci représente un des challenges à adresser pour un débogage temporel efficace.

Gap important entre les vues globales et les représentations détaillées

Nous avons vu dans le chapitre précédent que les visualisations existantes sont soit trop agrégées, soit trop détaillées, soulevant la question suivante : *quelle est la technique de visualisation à développer pour fournir un outil proposant suffisamment d'information sur le comportement de l'application pour commencer l'analyse tout en restant suffisamment aggrégé pour permettre une exploration efficace des données ?*

Performances insuffisantes des outils de stockage utilisés

Les outils existants reposent sur une stratégie de découpe arbitraire de la trace en un certain nombre de tranches de temps, appelées pages. Cette technique permet de réduire le temps d'accès au disque en limitant les requêtes uniquement à la page courante et a été développée à un moment où les disques étaient bien plus lents mais rend fastidieux l'exploration de traces comportant plusieurs millions d'évènements. Les disques modernes fournissent un temps d'accès aux données largement réduit avec l'utilisation de Solid State Disks (SSD). Un challenge à adresser serait de tirer parti de leurs performances pour fournir des outils interactifs : *comment développer une solution permettant l'exploration interactive de séries temporelles comportant plusieurs millions d'évènements ?*

Fouille de données pour la visualisation de traces d'exécution

Il existe des techniques de fouille de données capable de détecter automatiquement les comportements anormaux dans les traces d'exécution. Par exemple, Lopez Cueva et al. [Lopez Cueva et al., 2012] ont proposé une technique pour détecter les motifs périodiques dans la trace, et Iegorov et al. [Iegorov et al., 2015] ont développé un algorithme pour détecter les perturbations. Utiliser ces résultats pour enrichir les visualisations résulterait dans des outils de débogage bien plus performants. Ici, le challenge à adresser est le suivant : *comment exploiter les techniques de fouille de données pour enrichir les outils de débogage de traces d'exécution ?*

Chapter 5

Approche de Recherche et Méthodologie d'Evaluation

Dans l'état de l'art, nous avons remarqué un manque de techniques de visualisation pour représenter de façon précise les séries temporelles. Ce problème concerne un domaine plus large que la visualisation de traces d'exécution et touche un aspect fondamental de la visualisation de données temporelles. Par conséquent, nous commençons au Chapitre 6 par nous concentrer sur ce problème où nous proposons une nouvelle technique de visualisation pour séries temporelles, les Slick Graphs, qui minimisent les artéfacts visuels au niveau du pixel. Nous utiliserons les Slick Graphs comme brique de base pour les travaux présentés dans les chapitres suivants. Nous évaluerons les Slick Graphs à travers une évaluation utilisateur contrôllée afin de mesurer précisément les bénéfices de notre approche.

Au chapitre 7, nous introduisons une solution, TraceViz, pour réduire le gap existant dans les outils d'analyse de traces, à savoir l'uns étant trop aggrégés pour commencer l'analyse de la trace, les autres étant trop détaillés et empêchent une exploration efficace de traces volumineuses. TraceViz a été développé en collaboration avec les équipes de développement d'outils d'analyse de traces à STMicroelectronics et a été intégré et déployés aux développeurs internes ainsi qu'aux clients STMicroelectronics. Ceci constitue une validation industrielle, montrant que TraceViz répond à un réel besoin utilisateur.

Dans le chapitre 8, nous proposons une nouvelle vue globale de la trace montrant les structures d'exécution cachées. Les structures sont calculées à partir de techniques de fouille de données et apportent aux développeurs une nouvelle perspective sur l'exécution, difficilement trouvables avec les outils existants.

Chapter 6

Slick Graphs: Visualisation Lisse de Séries Temporelles

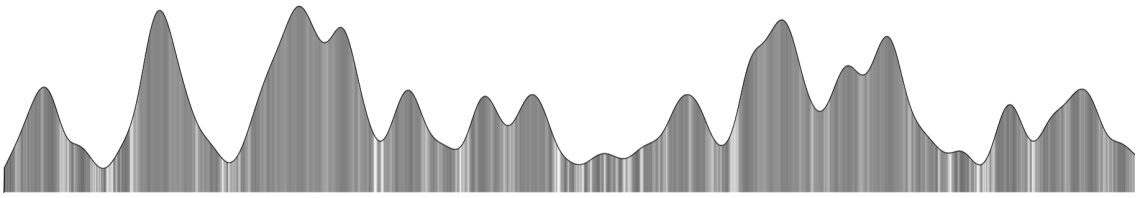


Figure 6.1: Slick Graphs

Dans ce chapitre, nous introduisons les Slick Graphs 6.1, une technique de visualisation pour séries temporelles. Les séries temporelles sont souvent montrées lissées pour permettre une lecture plus facile. Les techniques de lissage traditionnelles utilisées par de nombreuses visualisations pour des données temporelles reposent sur un découpage des valeurs en un petit nombre d'intervalles de temps et interpolent de façon lisse ces valeurs, ce qui peut introduire de nombreux artéfacts visuels. Les Slick Graphs minimisent ces artéfacts en utilisant les plus petits intervalles de temps possibles, les pixels. Ils fournissent néanmoins un rendu final lisse en applicant une convolution avec un noyau statistique sur les valeurs obtenues. Les informations filtrées, qui peuvent être perdues par le lissage, sont encodées dans la luminosité des pixels. Nous comparons les Slick Graphs aux techniques de lissage traditionnelles à travers une étude utilisateur qui contient de nombreuses tâches de comparaison. Les résultats montrent que les Slick Graphs sont plus performants que l'algorithme de lissage référent : l'algorithme introduit avec ThemeRiver. En particulier, les utilisateurs sont significativement plus rapides (+62%) et plus précis (+48%) avec les Slick Graphs en travaillant sur des données temporelles périodiques. Enfin, nous montrons comment utiliser les découpages des données en se basant sur les pixels ainsi que la méthode de lissage des Slick Graphs s'intègrent efficacement avec les techniques de visualisation existantes pour séries temporelles.

Chapter 7

TraceViz

Figure 7.1: TraceViz

Dans l'état de l'art, nous avons montré qu'un gap existe entre les outils proposant une vue globale de la trace et ceux fournissant trop de détails. Nous avons expliqué que les premiers ne permettent pas de commencer l'analyse de la trace correctement dû à une aggrégation trop importante des données. Les seconds rendent la navigation de traces volumineuses fastidieuse à cause du trop grand nombre de détails visualisés. Nous proposons dans ce chapitre TraceViz, un framework de visualisation multi-échelle qui permet aux développeurs d'explorer de façon interactive des traces avec un nombre important d'évènements en partant d'une vue très aggrégée jusqu'à avoir le détail d'un unique évènement. TraceViz est composé de deux composants principaux : un nouveau système de gestion des données et une nouvelle technique de visualisation multi-échelles. Les outils existants utilisent SQLite pour stocker les

traces. Cette solution atteint ses limites en travaillant avec les traces modernes pouvant contenir jusqu'à plusieurs millions d'évènements. Pour permettre un passage à l'échelle et fournir une exploration interactive de la trace, nous avons développé un nouveau système de stockage de traces basés sur HDF5 et prenant avantage des disques SSD modernes. Nous avons mesuré les performances de notre solution et les résultats montrent que notre approche permet un temps de réponse suffisamment court pour garantir l'interactivité des outils. Le second composant de TraceViz est un nouvel outil de visualisation prenant appui sur le système de stockage développé et basé sur les Slick Graphs afin de garantir une visualisation précise des données (Figure 7.1).

TraceViz a été développé en collaboration avec STMicroelectronics et a servi à la résolution de plusieurs bogues rencontrés en situation réelle. TraceViz a aussi été intégré dans les outils de STMicroelectronics, fournis aux développeurs internes ainsi qu'aux clients des solutions STMicroelectronics. TraceViz est donc potentiellement accessible à plusieurs milliers de développeurs.

Chapter 8

Visualisation des Structures Cachées Dans la Trace

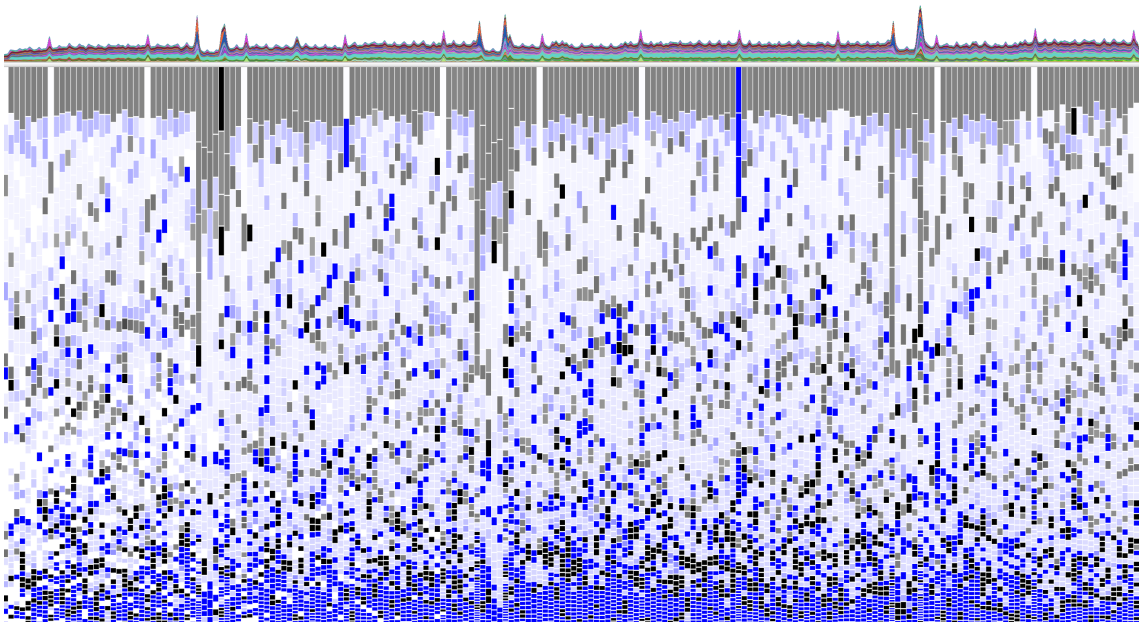


Figure 8.1: Visualisation de structures dans une trace d'exécution

Dans ce chapitre, nous proposons une nouvelle technique d'analyse visuelle pour comprendre rapidement les principales structures existantes dans les données ainsi que leur évolution au cours du temps. A la place de laisser les développeurs découvrir visuellement les motifs temporels, comme avec TraceViz par exemple, nous proposons une approche qui combine les techniques de visualisation de données avec des méthodes de fouille de données simplifiées dans le but de fournir une visualisation finale compréhensible par un utilisateur non-expert. Nous introduisons la notion de *structure*, qui regroupe différentes informations sur un itemset telles que son support, sa séquence dominante, le support de cette séquence ainsi que sa périodicité. Nous

calculons exhaustivement toutes les structures de la trace sur plusieurs tranches de temps et procédons ensuite à un classement afin de faire ressortir quelles sont les structures dominantes. Basée sur ces résultats, nous avons conçu un outil de visualisation des structures sur la trace montrant quelles sont les structures dominantes sur chacune des tranches de temps préalablement définies (Figure 8.1). Chaque structure est représentée avec un rectangle dont la couleur indique s'il s'agit d'un itemset (noir) ou d'une séquence (bleu). La périodicité est encodée dans le channel alpha du rectangle, plus la structure est périodique, plus la couleur est transparente.

Nous avons appliqué notre méthode aux traces d'exécution et les premiers résultats montrent que les structures récurrentes dans l'exécution ainsi que les perturbations apparaissent dans la visualisation. Nous avons aussi appliqué notre méthode sur différents jeux de données : du texte et des logs de Git. En utilisant du texte comme entrée, nous pouvons voir apparaître la structure du roman analysé, les différentes phases de l'histoire. L'analyse des commits Git permettent de faire ressortir les équipes de développeurs ainsi que les comportements récurrents dans les commits.

Chapter 9

Etude d'un Environnement Intégré de Débogage

Nous proposons dans ce chapitre un exemple d'intégration des différentes contributions faites dans cette thèse afin de montrer comment les différentes approches proposées sont complémentaires et peuvent être combinées. Pour illustrer notre propos, nous montrons comment un bogue peut être résolu. Nous avons proposé dans les chapitres précédents TraceViz et un outil d'analyse visuel pour identifier les structures dominantes de la trace. Nous proposons ici de commencer l'analyse de traces avec TraceViz afin de commencer par explorer et filtrer efficacement les données. En procédant ainsi, les développeurs sont capables de rapidement cibler des parties de la trace qu'ils jugent intéressantes. Une fois la trace filtrée, il est possible de faire une analyse locale plus poussée à l'aide de l'outil d'analyse visuel présenté dans le chapitre précédent. Le développeur pourra ainsi se concentrer sur les structures dominantes et identifier les perturbations dans les zones préalablement sélectionnées avec TraceViz.

Dans ce chapitre, nous montrons un exemple d'intégration de nos outils mais une étude avec les développeurs est nécessaire afin de définir la meilleure méthodologie de travail ainsi que l'intégration optimale des techniques présentées au cours de cette thèse.

Chapter 10

Conclusion

Au cours de cette thèse, nous avons proposé trois contributions dans les domaines de la visualisation d'information et contribué aux activités industrielles de STMicroelectronics. Le but de cette thèse était de proposer de nouveaux outils de débogage pour traces d'exécution basés sur des techniques de visualisation et de la fouille de données. Ainsi, avoir une visualisation précise de séries temporelles utilisable en contexte scientifique et industriel est primordial. Nous avons vu que les techniques existantes introduisaient des artéfacts visuels conduisant à un rendu imprécis des données temporelles. Par conséquent, nous avons présenté les Slick Graphs, une nouvelle technique de visualisation interactive basée sur les pixels qui permet de visualiser les séries temporelles de façon fidèle.

La deuxième contribution de cette thèse a été TraceViz, un framework de visualisation pour traces d'exécution. TraceViz est un compromis entre une vue globale trop générale pour être utile et une vue montrant un niveau de détails trop élevés pour être compris par l'utilisateur. Nous avons montré comme les développeurs peuvent détecter visuellement les motifs temporels et comportementaux dans la trace. TraceViz a été intégré à la suite d'outils STMicroelectronics.

La troisième contribution de cette thèse est une nouvelle méthode d'analyse visuelle permettant de visualiser les structures cachées dans les traces d'exécution. Pour ceci, nous avons basé notre approche sur des techniques de fouille de données pour calculer les différents motifs dans la trace. Nous avons proposé une technique de visualisation pour représenter ces structures de façon compréhensible pour des utilisateurs non-experts. De cette façon, le régime principal de l'application peut être visualisé ainsi que les différentes perturbations ayant eu lieu.

Suite à ces travaux de thèse, les perspectives suivantes constituent des pistes de travail intéressantes. Dans TraceViz, à chaque acteur de la trace correspond une série temporelle et les acteurs sont organisés par défaut selon une hiérarchie préalablement définie. En parallèle de cette organisation, les acteurs pourraient être classés avec un clustering hiérarchique ce qui permettrait d'accélérer le processus de débogage en faisant apparaître des motifs communs aux acteurs. Il existe différentes techniques

de visualisation pour explorer un clustering hiérarchique. Leur intégration avec TraceViz amènerait des questions de recherche intéressantes dans le domaine de la visualisation d'information.

Avec TraceViz, nous avons proposé l'utilisation un nouveau stockage de données permettant l'exploration interactive des traces d'exécution volumineuses actuelles. Cependant, tout système de stockage et algorithme atteind sa limite avec des données suffisamment volumineuses. Une approche différente consiste à visualiser des résultats partiels au cours du calcul. Les premières études ont montré que les analystes sont capables de raisonner avec des résultats même partiels, montrant que le rendu incrémental est une piste encourageante.

La norme en visualisation d'information est de représenter chaque variable dans les données par un unique attribut visuel mais une approche différente peut être considérée. Par exemple, dans les Slick Graphs, nous avons encodé une valeur de l'histogramme sur deux attributs : la hauteur et le canal alpha du pixel. Ceci soulève une question de recherche plus générale : comment encoder une variable des données sur plusieurs attributs visuels pour visualiser des informations difficilement compréhensibles autrement ?

Nous avons introduit une méthode pour calculer les structures dans les traces au Chapitre 8. Deux pistes de travail s'ouvrent. D'une part, il serait intéressant d'intégrer des algorithmes de fouille de données existants pour avoir des résultats plus précis. D'autre part, une autre amélioration serait d'implémenter notre méthode de calcul des structures sur GPU afin de réduire le temps de calcul et de tendre vers un outil interactif.

Nous avons utilisé dans nos structures des itemsets, des séquences et des séquences périodiques mais le concept de structure peut être étendu à d'autres types de motifs comme par exemple des graphes. Ceci permettrait de visualiser l'évolution des structures dans des réseaux sociaux comme Twitter ou Facebook. La difficulté principale serait de faire correspondre tous les paramètres des structures sur des attributs visuels tout en gardant une visualisation claire pour une compréhension rapide de ces structures.