Data Mining Approach to Temporal Debugging of Embedded Streaming Applications

Oleg Iegorov

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Oleg Iegorov

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Miguel Santana**

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

Data Mining Approach to Temporal Debugging
of Embedded Streaming Applications

Thèse soutenue publiquement le **8 avril 2016**,
devant le jury composé de :

M Raymond Namyst
Professeur à l'Université de Bordeaux, Président
M Sebastian Fischmeister
Associate Professor à l'Université de Waterloo, Rapporteur
Mme Maguelonne Teisseire
Directrice de recherche à IRSTEA, Rapporteur
M Marc Plantevit
Maître de Conférences à l'Université Claude Bernard Lyon 1, Examinateur
M Jean-François Méhaut
Professeur à l'Université Grenoble Alpes, Directeur de thèse
M Miguel Santana
Directeur du centre SDT à STMicroelectronics, Co-Directeur de thèse
M Alexandre Termier
Professeur à l'Université de Rennes 1, Co-Encadrant de thèse
M Vincent Leroy
Maître de conférence à l'Université Grenoble Alpes, Co-Encadrant de thèse

Contents

List of Figures

List of Tables

Acronyms

ASIC	Application-Specific Integrated Circuit. 12, 15
CE	Consumer Electronics. 2, 10
CPU	Central Processing Unit. 12, 92
CSDF	Cyclo-Static Dataflow. 15
DAC	Digital-to-Analog Converter. 12
DAG	Directed Acyclic Graph. 14
DSP	Digital Signal Processor. 12
ESL	Electronic System-Level. 14
FPS	Frames Per Second. 16
FSM	Finite State Machine. 14
GPU	Graphics Processing Unit. 12
HD	High Definition. 11, 15
HEVC	High Efficiency Video Coding. 12
IP	Intellectual Property. 12, 19
IP network	Internet Protocol network. 73, 74
JTAG	Joint Test Action Group. 18, 19
MCS	Minimal Contrast Sequence. 40, 44, 52, 54
MoA	Model-of-Architecture. 14
MoC	Model-of-Computation. 2, 14
MPEG	The Moving Picture Experts Group. 14
MPSoC	Multiprocessor System-on-Chip. 2, 12, 17, 62
OS	Operating System. 17, 19, 26
PC	Personal Computer. 13
QCoD	Quartile Coefficient of Dispersion. 32
QoS	Quality-of-Service. 2, 6, 11, 15, 18, 33, 36
QVC	Quartile Variation Coefficient. 32
RISC	Reduced Instruction Set Computing. 12
SATM	Streaming Application Trace Miner. 6, 26, 36, 71, 93
SDF	Synchronous Dataflow. 15
SoC	System-on-Chip. 12
WCET	Worst-Case Execution Time. 16, 17, 24

Chapter 1

Introduction

Imagine it is a Sunday night, and you are at home watching some good movie streaming from Netflix on your new home cinema system. The movie is interesting, the picture is impeccable, so is the sound. Then, suddenly, the picture freezes for a fraction of a second, and the movie continues. Nothing horrible happened, your home cinema system is still there and working properly, and you have a thought that your WiFi connection may not be that great (although you are aware that such situations never happen when you are streaming movies on your laptop). After a minute again, a slight stutter in the video occurs, and the movie goes on. After several such hiccups your movie experience is somewhat spoiled, and you are no more enjoying the whole Sunday night movie session as you were supposed to.

Without realizing it, you may have just experienced the effects of temporal bugs present in the software running on the embedded system enclosed in a shiny set-top box from your home cinema system. Temporal bugs are elusive, they do not make you immediately throw the entire home cinema away, however, they significantly degrade user experience of using multimedia devices. As a matter of fact, temporal bugs are an annoyance not only to the users of multimedia devices, but even more so to the developers of embedded software running on those devices. Indeed, there is no widely adopted tools or techniques to debug temporal issues in embedded software, which makes temporal debugging a complex and highly time-consuming part of embedded software development.

In this thesis, we intend to help software developers in resolving temporal bugs at the stage when the embedded system has not yet been delivered, and it is not too late to modify the source code of the software running on it. We propose a temporal debugging approach which automates the process of discovering the origins of temporal bugs by applying data mining algorithms on execution traces of embedded systems found in multimedia devices.

1.1 Context and Motivation

It would be hard to imagine the life of a modern person without Consumer Electronics (CE) devices. In a matter of decades, bulky and expensive portable radios and cathode ray tube TV sets have transformed into sleek and affordable smartphones and LCD panels. According to the Consumer Electronics Association, the global sales of consumer electronics reached $1.024 trillion in 2014 [1] with *multimedia* CE devices (smartphones, tablets, TVs, etc.) being the biggest players on the market. The main goals of multimedia consumer electronics consist in providing entertainment and giving access to communication media (e.g. the Internet, telephone networks, television, etc.). These goals dictate the way such devices operate on the software level: they run streaming applications which apply the same transformations on the incoming batches of data. For example, the video decoding functionality of set-top boxes, smartphones, or tablets consists in applying the same decoding algorithm to transform the encoded video stream into a set of ordered frames and display them on the screen. The constantly growing demand on the aforementioned multimedia devices fuels rapid evolution and high competitiveness of the consumer electronics market. Companies struggle to introduce new features to each new generation of their CE devices, make their products more attractive to consumers, and be the first among competitors to deliver their products to the market.

If we disassemble a modern multimedia CE device, we will most probably find an embedded system as its main electronic component. The most common choice of an embedded system platform for CE devices nowadays is a Multiprocessor System-on-Chip (MPSoC), which integrates all the electronic components, including multiple processing units, into a single chip. Among the examples of modern MPSoCs are Apple's A9X chip found in Apple iPad Pro tablets and Qualcomm's Snapdragon 810 chip found in Google Nexus 6P smartphones. Each new generation of MPSoCs aims at making CE devices smaller, more performant, consume little power, and have an affordable price tag.

From the software perspective, Dataflow Model-of-Computation (MoC) has been widely adopted as a programming paradigm for streaming applications. With Dataflow MoC, applications are modeled as a directed graph, where data flow between computational components (called actors) through communication channels. Each actor works on its own local data and exchanges data with other actors only through communication channels. This makes Dataflow MoC particularly well suited for the parallel processing nature of MPSoC platforms.

A distinguishing characteristic of embedded systems is the presence of real-time requirements imposed on them. MPSoCs found in multimedia CE devices are no exception. In fact, such systems not only require streaming applications running on them to produce a correct output (e.g. a correctly decoded video frame), but also to deliver the output on time, i.e. before the deadline, to ensure a high Quality-of-Service (QoS) of the entire

[1] http://www.cta.tech/CorporateSite/media/About-Media/2015-Global-Technology-Market-Update_CES-2015_press.pdf

system (e.g. a video frame must be decoded in 33 milliseconds to respect 30 frames per second QoS constraint). Modern multimedia CE devices are increasingly required to provide *hard* real-time performance, when not a single deadline can be missed, so that the devices are more pleasant to use and, hence, are more competitive on the market.

Debugging is an essential step in the software development process. Programmers are well trained in resolving functional bugs, i.e. bugs that make systems produce wrong output values or crash. Situations when a piece of software produces correct output values but delivers them later than expected is a result of temporal bugs, sometimes also referred to as performance bugs. In the context of real-time systems, we will call a **temporal bug** an anomalous system activity which causes the output to be delivered late, that is, after the real-time deadline. If streaming applications running on a multimedia CE device cannot produce smooth output, users will not tolerate this and rather buy a competitor's product which ensures a better user experience. This makes temporal debugging of embedded software equally important to resolving functional bugs.

Temporal bugs are usually very hard to resolve. The first reason is that the traditional debugging approach based on setting breakpoints and stepping through the application's source code inspecting the system's state, which is extremely helpful for resolving functional bugs, is often useless for temporal debugging. Indeed, the act of halting the program's execution changes temporal behavior of the entire system. Another aspect that complicates the process of temporal debugging is the necessity to consider the entire system's execution context, as temporal bugs often originate from resource sharing interactions between unrelated execution threads. Finally, temporal bugs have a tendency to appear at the last stages of embedded system development, when use cases of running various combinations of embedded software on an already manufactured chip are tested. Temporal debugging must be, therefore, performed under a great time pressure, so that the entire embedded system is not delayed from being released to the market.

Execution profiling has a long history of being an invaluable technique to optimize software performance. A profiling tool gathers various metrics during a program's execution, e.g. execution cycles, cache misses, etc., and allows programmers to detect the most significant parts of the code with respect to the chosen metrics. Unfortunately, information provided by profiling tools is often not enough to resolve temporal bugs. A profiler would show where the cycles were spent or which function calls result in a large number of cache misses, but there would be no information on how these statistics are related to the temporal bugs, and whether optimizing performance of the detected parts of the code would make temporal bugs disappear.

Execution tracing technology can be considered as an alternative to software profiling for runtime behavior analysis of embedded systems. A tracer is able to record system activity during software execution on various granularity levels: from high-level events such as function calls to low-level ones such as executions of individual instructions. A developer would then open a trace file and analyze the timestamped sequence of executed events in order to get an insight into what happened on the chip while the software was running. Recent advances in both software and hardware allowed to significantly

reduce the intrusiveness of execution tracing, i.e. the overhead imposed on the system performance by the tracer. It is, therefore, possible nowadays to record a full log of non-perturbed system activity for post-mortem analysis. At the same time, the amount of information contained in execution traces is simply overwhelming, even when the system was traced for a very short period of time.

Visual exploration of execution trace data is the most common way to perform temporal debugging of embedded software. For example, trace visualization tools based on Gantt charts (see Figure 1.1) make it easier to analyze the order of executed events during a given time interval. Despite the addition of a visual component to the task of temporal debugging, manual analysis of execution traces remains a daunting task. A software developer is confronted with too much system activity at any given moment of time to be able to grasp it and, what is even more, to find suspicious patterns of system behavior related to the temporal bugs. As a result, there is currently a great need for novel tools to automate the process of temporal debugging.

Data mining is a research field that is concerned with automatic discovery of potentially useful and comprehensible patterns from large data sets. Data mining involves four common classes of tasks: cluster analysis, classification, anomaly detection, and pattern mining. Cluster analysis seeks to find groups of closely related observations so that observations that belong to the same cluster are more similar to each other than observations that belong to other clusters. Classification assigns observations from the data set to target categories, or classes, with the goal of accurately predicting the target class for each observation. Anomaly detection is the task of identifying observations whose characteristics are significantly different from the rest of the data. Finally, pattern mining seeks to discover interesting relations between individual observations. Thanks to

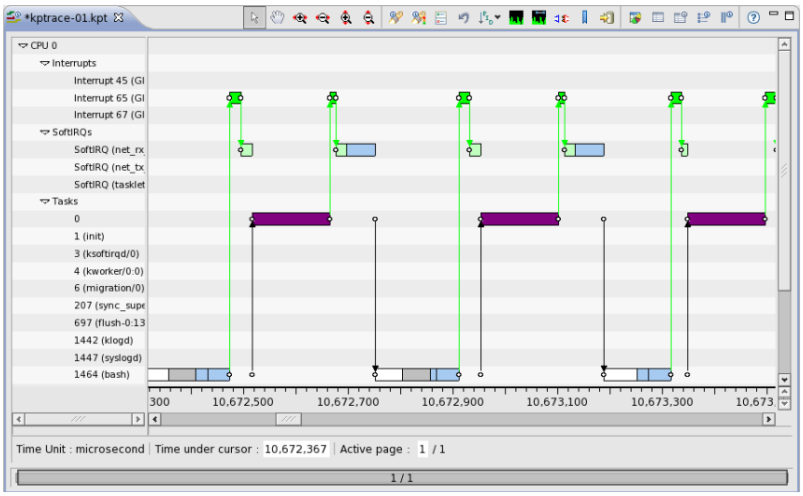


FIGURE 1.1: An example of trace visualization using a Gantt chart [2]. The Y-axis shows active execution threads. The X-axis shows the duration of individual events executed in the context of the corresponding threads.

[2] http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_perspective.html

their generality, data mining algorithms have been successfully applied in diverse areas: retail industry, biological data analysis, banking, transportation, and many others. At the same time, the universality of data mining algorithms often requires a significant effort from a domain expert to interpret the obtained results.

1.2 Aim and Scope

In this thesis, we aim to automate the process of temporal debugging of embedded streaming applications. More precisely, we are interested in proposing a data mining approach which, given an execution trace, would return a specific and concise description of system activity explaining the origin of the QoS problem.

We do not address the task of verification of real-time properties of embedded applications performed in the design stage of software development. Instead, our focus is on the final stages of software development process, when various combinations of programs are tested on an already manufactured chip imitating real use cases run on the final embedded system.

Next, we assume that it is possible to the developer to obtain an execution trace which captures unaltered system activity during a particular usage scenario of the embedded system. Moreover, the trace must be complete enough to contain the footprints of problematic system behavior. In other words, rather than containing a log of function calls from a specific software component, a trace must cover as much system activity as possible without, however, inducing the tracing program to become intrusive.

We do not pretend to replace already existing techniques used for temporal debugging purposes, e.g. profilers or trace visualization tools. Instead, our intention is to enable embedded software developers to use these tools in a more systematic way. For example, profilers can still be used in the beginning of the debugging process to decide which parts of the system must be traced. Visualization tools, on the other hand, could help to better understand the context of the detected suspicious system activity at the final step of temporal debugging.

Finally, our goal is not to develop a generic data mining algorithm that can be applied on any execution trace and whose results would require a considerable effort to be filtered and analyzed. Instead, we would like our temporal debugging approach to strike a fine balance between specificity and generality, so that it returns a precise description of the probable origin of the QoS problem but remains general enough to be applicable in a wide variety of contexts where temporal debugging needs to be performed.

1.3 Significance of the Study

Temporal debugging of embedded streaming applications remains a tedious, unsystematic, and highly time-consuming process. Software developers are faced with a lack of appropriate tools and techniques to resolve temporal bugs under the tight time constraints present at the last stages of software development process. An approach that would allow to get an insight into the origins of temporal bugs by simply "clicking on a button" is of high need. It would not only relieve software developers of the stress of looking for a needle in a haystack of raw trace data but would also allow consumer electronics companies to respect time to market, hence, to increase their chances of success in the highly competitive market of CE devices.

1.4 Overview of the Contribution

In this thesis, we propose Streaming Application Trace Miner (SATM) – a novel approach to help understand the causes of QoS violations in multimedia CE devices. In practice, SATM is a data analysis workflow that makes use of statistical methods and data mining algorithms with the aim to automate the process of temporal debugging of embedded streaming applications.

SATM takes as input (1) an execution trace captured during an embedded system's usage scenario where the target application exhibits low QoS and (2) a list of execution events which represent the actors of the target application's dataflow graph [3]. The output of SATM consists of a set of system activities related to the origin of the QoS problem and represented as patterns of execution events.

Our approach operates in two distinct stages:

I. Detection of the anomalous parts of the execution trace. The peculiarity of temporal bugs, which makes them difficult to debug, is that they tend to occur much earlier than the moment of time when the QoS of the target application starts to degrade. For example, a delayed output of a video frame (QoS problem) may be caused by a network buffer overflow (temporal bug) which takes place earlier in time when the encoded frame is being received from the network. Therefore, the goal of this step consists in finding the parts of the execution trace where the system's behavior first starts to diverge from its normal operation. SATM achieves this task in a completely automatic fashion using a one-dimensional clustering algorithm and a few statistical metrics.

II. Discovery of suspicious system activities. Once the origin of the QoS issue is tracked down to particular parts of the trace, SATM detects patterns of execution events which characterize the abnormal system activities in the most concise way. Such patterns are discovered using a minimal contrast sequence mining algorithm.

[3] strictly speaking, the presence of this list does not influence the way SATM operates, but rather the accuracy of the output results.

1.5 Scientific Context

This thesis was funded by a CIFRE ANRT partnership between STMicroelectronics and the LIG (Laboratoire d'Informatique de Grenoble) laboratory of the University of Grenoble, France. The industrial part of this work was carried out in the SDT (Software Development Tools) team of STMicroelectronics. The role of the SDT team consists in providing the company's customers with development and debugging tools tailored to the company's embedded platforms. The scientific part of this work was conducted in SLIDE (ScaLable Information Discovery and Exploitation) and CORSE (Compiler Optimizations and Runtime SystEms) teams of the LIG laboratory. SLIDE's research targets efficient large-scale data processing with data mining algorithms. One of the research axes of the CORSE team aims at proposing novel debugging techniques for multi-core embedded platforms.

1.6 Organization of the Thesis

The rest of this document is divided into six chapters, as follows:

- In Chapter 2, we present the context of multimedia embedded systems and articulate both the importance as well as the difficulty of temporal debugging. We also identify fundamental properties of these systems that allowed us to automate the process of temporal debugging, as explained in the following chapters.
- In Chapter 3, we present the first stage of our temporal debugging approach SATM. In Section 3.1, we provide theoretical underpinnings of the way suspicious parts of execution traces are detected. We then proceed, in Section 3.2, with the explanations of how this goal is achieved in practice.
- In Chapter 4, we not only discuss how the problem of temporal debugging from execution traces can be expressed as a pattern mining task (Section 4.1) and how abnormal system activity can be mined with pattern mining algorithms (Section 4.2), but we also make an important contribution to the theoretical analysis of sequential pattern mining algorithms applied on trace-like data sets (Section 4.3).
- In Chapter 5, we present three real-world use cases of streaming applications exhibiting low QoS (Section 5.1) and show how SATM has been applied to find abnormal parts in their execution traces (Section 5.2), as well as pinpoint patterns of events related to the origins of temporal bugs (Section 5.3).
- In Chapter 6, we review the existing approaches for debugging temporal issues in embedded applications, both the ones that work without execution traces (Section 6.1), as well as those requiring execution traces (Section 6.2).
- In Chapter 7, we conclude by reviewing the results of this thesis and defining the future perspectives.

Chapter 2

Background

Temporal bugs exhibit themselves as visual or sound artifacts which are familiar to all users of multimedia devices: a briefly frozen video playback or a crack in the sound are among the examples. Such bugs do not crash the system, as the user is still able to use the device, however, their presence is very likely to make the user give up on the device.

As a matter of fact, temporal bugs are an annoyance not only to the users of multimedia devices but even more so to the developers of embedded software running on those devices. Indeed, software developers are well trained to resolve *functional* bugs which manifest themselves as wrong output values and which often crash the system. There exists a set of widely used tools to perform functional debugging, such as GDB, Valgrind and all sorts of visual debuggers available in popular integrated development environments. Temporal bugs, however, are conceptually different from the functional ones as they do not result in wrong output values, nor do they crash the system.

Definition 2.1. A **temporal bug** is a non-fulfillment of the timing requirements of the system. In the context of multimedia embedded software, temporal bugs violate real-time behavior of applications by preventing them to produce output *on time*, i.e. before the scheduled deadline.

There are several aspects that make temporal bugs difficult to deal with. Firstly, in contrast to functional debugging, there are no standard tools to perform temporal debugging. Secondly, temporal bugs have a tendency to appear at the last stage of software development cycle, when various combinations of multimedia applications are run on the already manufactured chip, mimicking the real device usage, just before the whole multimedia system is delivered to the client. Therefore, a lack of standard debugging tools coupled with a great time pressure to resolve temporal bugs is a major source of stress for embedded software developers.

In order to facilitate the process of temporal debugging, we must first understand the nature of temporal bugs: why they appear, what makes them so elusive to debug,

and which technology may potentially help to resolve them. This chapter serves this purpose. We introduce multimedia embedded systems as well as the requirements that the market of CE devices is imposing on them in Section 2.1. We then discover how the enforced requirements are met on the hardware level (Section 2.2), on the software level (Section 2.3), as well as how these requirements dictate the way the software must be executed on the hardware (Section 2.4). Next, we find out why running several multimedia applications on the same embedded system may result in temporal bugs (Section 2.5), and why temporal debugging usually cannot be performed with the state-of-the-art debugging tools (Section 2.6). Finally, we learn about an extremely useful technology for temporal debugging which is software execution tracing (Section 2.7), before concluding that there currently exists a high demand on tools for automatic extraction of temporal bug-related information from execution traces (Section 2.8).

2.1 Multimedia Embedded Systems

All of us use multimedia embedded systems in our everyday life. In fact, these systems are enclosed in the most common CE devices, such as smartphones, tablets, cameras and set-top boxes, that is, if a CE device is disassembled, a multimedia embedded system can be easily identified as its main electronic component. However, an embedded system is more than electronics: besides digital and analog circuits, special purpose sensors and actuators it also comprises software that controls all the hardware elements. Both hardware and software parts of an embedded system are designed to deliver a specific set of functionalities to the user. In the case of *multimedia* embedded systems such functionalities include audio/video decoding, 3D gaming, and web browsing.

Multimedia embedded systems have numerous requirements that must be fulfilled to meet user expectations. We present a summary of these requirements below.

FIGURE 2.1: Examples of multimedia consumer electronic devices: a tablet, a set-top box and a smartphone.

- *Functionality.* New generations of multimedia embedded systems must provide new features and better performance than the previous ones in order to keep the user's interest in CE devices. Taking set-top box market as an example, a support of Full High Definition (HD) video content became a must several years ago, while Ultra HD 4K is starting to become a desirable feature, and systems that support Ultra HD 8K resolution have already appeared on the market.

  Besides new features, users want to perform more and more tasks simultaneously on their multimedia devices; for example, to watch one TV program and record another one to watch later, or to browse the Web and display a movie in an inset window in the corner of the screen (picture-in-picture feature).

- *Low energy consumption.* Even with a constant increase in built-in battery capacities, embedded systems in CE devices must consume as little energy as possible, so that users do not need to constantly worry that the battery runs off in the most inappropriate moment. In case of devices that are permanently plugged in an electrical outlet, like set-top boxes, their influence on the electricity bill should be as small as possible.

- *Small physical size.* Besides practical benefits of being able to fit into a pocket or a bag, a small size of a CE device adds valuable points to its overall attractiveness to the users. The size of multimedia embedded systems, which are the main components of CE devices, has the direct influence on the size of the final product.

- *Low cost.* In order to reach the mass market, a CE device must be based on an inexpensive multimedia embedded system. If new features come with an unaffordable price tag, customers would rather choose to wait for a cheaper alternative or decide that those features are not necessary for them at all.

- *Short time-to-market.* The market of multimedia devices is extremely competitive. If it takes a particular company a long time to introduce a new feature to its CE device, e.g. because software bugs slow down the release date of the final product, then the competitors will enjoy the biggest part of the market. It is, thus, essential for the manufacturers of integrated circuits, as well as for the embedded software developers to respect stringent temporal constraints.

- *High quality of service (QoS).* If a CE device meets all the requirements listed above but does not provide high-quality output, then customers will simply not use the device. Instead, they will prefer a rival's product which may lack some features or have a bigger size but instead provides a better usage experience. For multimedia embedded systems, the QoS consists in a smooth browsing and audio/video playback without any visual or sound artifacts, like frame glitches or audio cracks.

The following three sections show how these six requirements are met on the hardware level, on the software level, and in the way the software is executed on the hardware in multimedia embedded systems.

2.2 Hardware Perspective: System-on-Chip

A System-on-Chip (SoC) integrates all the components of an electronic system, such as a Central Processing Unit (CPU), memory modules, and numerous digital peripherals into a single chip. Tight integration of the components allows, on the one hand, to reduce the size of the electronic system, and on the other hand, significantly lower its energy consumption thanks to short wiring. The widespread use of Reduced Instruction Set Computing (RISC) CPUs also contributes to low energy consumption of SoCs. At the same time, the cost of manufacturing a single SoC is significantly lower than fabricating its components separately and assembling them inside a CE device. Designing and manufacturing such complex systems as SoCs from scratch would take prohibitively big amount of time. In order to address the time-to-market constraint, SoC manufacturers build their systems with reusable Intellectual Property (IP) cores supplied by such companies as ARM, Xilinx, Synopsys, and others. An IP core is a block of logic that describes the design of a particular system's component (a CPU, a Digital-to-Analog Converter (DAC), a memory controller, etc.) ready to be implemented on the target platform. All these benefits of SoCs resulted in their adoption as a hardware standard for embedded systems. Indeed, it is common nowadays to hear people using the terms "embedded system" and "system-on-chip" interchangeably.

The continuing customer demand for increased performance made embedded systems industry introduce chips with several programmable processors to be run in parallel, and a System-on-Chip has evolved into a Multiprocessor System-on-Chip (MPSoC). A distinctive property of such systems is that they are inherently heterogeneous, i.e. the processors found in an MPSoC are of different types [1] and have different roles depending on the embedded system's nature [84]. Moreover, an MPSoC designed for a smartphone will contain a different set of processors than the one designed for, say, a network router. As an illustrative example, consider Figure 2.2 which shows a recent STMicroelectronics' STiH412 MPSoC designed for set-top boxes. It supports Ultra HD 4K decoding at up to 2160p30 [2], triple HD decoding at 1080i60/1080p30, up to three HD concurrent transcodings making possible to target up to three tablets or smartphones, and many other features. In order to provide all this high-end functionality, the STiH412 MPSoC contains numerous processors, among them a dual-core ARM Cortex-A9 CPU [1], a quad-core ARM Mali-400 GPU [2], a dedicated processor for Ultra HD HEVC decoding, and others.

The presence of multiple processors, some of which are usually multi-core (CPUs and GPUs), in MPSoCs allows not only to increase the performance of a single application by parallelizing its execution, but also to run concurrently several data- and calculation-heavy applications in order to meet the functionality requirement of CE devices. A typical MPSoC is a striking example of how semiconductor companies succeed in squeezing

[1] Usually, a combination of several CPUs, Digital Signal Processors(DSPs), Graphics Processing Units(GPUs) and various types of Application-Specific Integrated Circuits(ASICs).

[2] The first number (2160 or 1080) denotes the vertical resolution, i.e. the number of pixels along the vertical axis; the letter p stands for progressive scan, the letter i stands for interlaced video; the second number (30 or 60) signifies the frame rate.

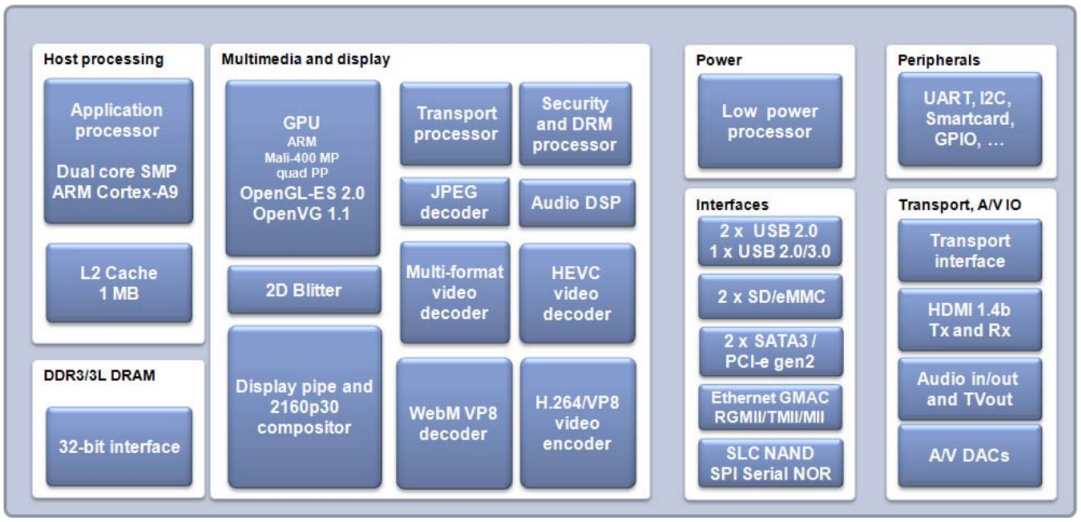


FIGURE 2.2: Block diagram of STMicroelectronics' STiH412 MPSoC for set-top boxes.

as much processing power as possible into a single chip. Figure 2.3 provides an illustration of how the size of a conventional Personal Computer (PC) motherboard compares to the size of iPad3's MPSoC, both having roughly the same number of components. Besides introducing several processors, designers of MPSoCs use all existing hardware-based optimizations to speed-up computations and data access: speculative execution, branch prediction, shared caches, etc.

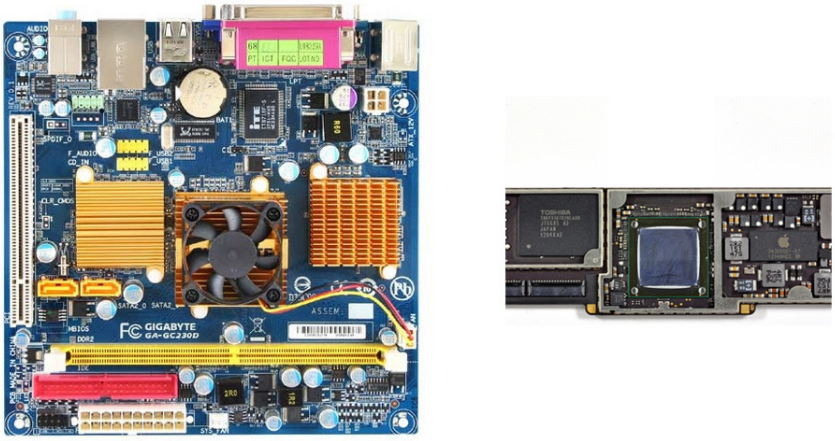


FIGURE 2.3: Comparison of physical sizes of a conventional PC motherboard (left) and an MPSoC found in iPad 3 (right). This image is roughly to scale [3].

In conclusion, the adoption of SoC and later MPSoC platforms as the hardware solution for multimedia embedded systems not only helped to meet the cost, size, and power consumption requirements of CE devices, but also allowed to run more powerful applications in a concurrent way in order to meet constantly growing functionality demands of customers.

[3] http://www.extremetech.com/computing/126235-soc-vs-cpu-the-battle-for-the-future-of-computing

2.3 Software Perspective: Dataflow Computational Model

New generations of multimedia embedded systems usually introduce changes to the platform's hardware: new components are added, old components are replaced with more performant ones, and so on. Embedded software, thus, needs to be either updated or completely rewritten in order to harness new computational resources of the underlying hardware. At the same time, market competition requires that the software is ready promptly after the new chip comes out of the factory. In the end, it is the combination of hardware and software that constitutes an embedded system.

To address the time-to-market requirement, embedded systems manufacturers have adopted the Electronic System-Level (ESL) design which encompasses concurrent design of hardware and software parts of an embedded system. Having the formal descriptions of the target application as well as of the target chip, ESL tools are able to synthesize an application specification written in a high-level language (such as C/C++) to an implementation which will be ready to run on the target hardware as soon as it is released [33].

Software-wise, ESL synthesis tools rely on *computational models* (MoC, for Model-of-Computation) for the description of the desired functional and non-functional requirements of the target application; hardware-wise, it is a Model-of-Architecture (MoA) which is required for the description of the platform design. A MoC defines semantics of the target application: which components it can contain, how the components can be interconnected, and how they interact. In other words, a MoC describes how to specify and execute algorithms. Every programming language has at least one underlying MoC. The most popular MoC for sequential programming paradigm is a Finite State Machine (FSM) MoC where a program has a global state which is modified sequentially by the modules. However, applications that typically run on SoCs do not conform to such computational model, as they apply same transformations on a stream of arriving batches of data. A global state, thus, does not exist because each piece of data in the stream has its own, independent state. These applications are called *streaming* applications[4]. Indeed, if we take video decoding as an example, the same decoding algorithm (e.g. MPEG-4) is applied on each encoded frame from the incoming data stream in order to produce a video stream ready to be displayed on the screen of a multimedia device.

A computational model that is particularly well adapted to design streaming applications is the *Dataflow MoC* [45]. Dataflow represents a program as a set of computational modules (called actors) which exchange batches of data (called tokens) through First-In First-Out communication channels. From a graphical point of view, a dataflow program can be represented as a Directed Acyclic Graph (DAG) where actors are vertices and communication channels are edges. Each actor may itself be a DAG or a procedure that applies transformations on the input data (Figure 2.4). An actor can start its execution as soon as enough data tokens are available on the incoming communication channel(s).

[4] In the rest of this thesis we will use the terms "streaming application" and "multimedia application" interchangeably

FIGURE 2.4: An example of a dataflow graph

A prominent property of the Dataflow MoC is that the actors operate only on local data (cf. the global state of the FSM MoC). This property makes dataflow applications inherently parallel, as each actor can execute independently and concurrently with other actors as long as its input is available. Given the parallel nature of MPSoCs, the Dataflow MoC is a de facto standard for embedded multimedia software design.

There exists a great variety of dataflow computational models, among them are the Synchronous Dataflow (SDF), the Cyclo-Static Dataflow (CSDF), and many others [64]. Figure 2.5 presents a simple CSDF graph containing three actors a_1, a_2, a_3 and two communication channels e_1, e_2. Each time an actor is invoked, it produces a specific number of tokens to its outbound communication channel and consumes a specific number of tokens from the inbound communication channel, as specified in the square brackets. Consider actor a_2. The first time a_2 is invoked, it consumes 1 token and produces 1 token. The next time it is invoked, it consumes 3 tokens and produces 1 token, the next time it consumes 1 token and produces 3 tokens, then 1 token is consumed and 1 token is produced again, and so on.

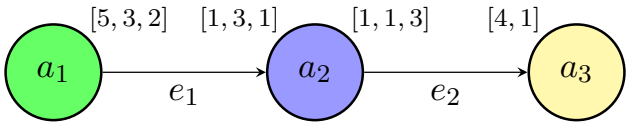


FIGURE 2.5: An example of a Cyclo-Static Dataflow graph

2.4 Scheduling Perspective: Hard Real-Time Constraints

Multimedia applications conceived to run in modern consumer electronic devices process high volumes of arriving data streams with sophisticated algorithms in order to provide a high-quality output, that is, a high QoS. For example, customers want their set-top boxes to decode an HD video stream coming from an Internet TV provider (functionality requirement) and to reproduce it smoothly and perfectly synchronized with audio and subtitles (QoS requirement).

One approach to ensure the output quality of an embedded multimedia application would be to use an ASIC, so that the application's algorithm is implemented completely in hardware, hence, the QoS is guaranteed on the hardware level. However, the diversity of applications a modern CE device must support, as well as the desire of application developers to use their software on a wide set of multimedia platforms require the use of software-based solutions. This implies that all the complexity of multimedia algorithms as well as application scheduling must be addressed on the software level. As shown in the previous section, Dataflow MoC provides an efficient way of designing embedded

multimedia applications. Scheduling, in its turn, must guarantee the *real-time* functioning of the applications. In fact, multimedia embedded systems is an example of real-time systems: their correct behavior depends not only on the operations they perform being logically correct, but also on the time at which they are performed. A real-time system must guarantee its response within specific time constraints, referred to as *deadlines*. For example, a video codec running on a CE device must not only correctly decode each encoded frame but should also output 30 of them per second, that is, respect a deadline of 30 Frames Per Second (FPS). If a set-top box correctly decodes video frames but is not able to do this fast enough, it cannot be called a multimedia device.

Historically, multimedia embedded systems were considered as *soft real-time* systems: occasional deadline misses which resulted in frame drops or audio/video desynchronizations were not critical to the functioning of a system and were either ignored or played a role of a trigger to downgrade the quality of the media (e.g. downscale video resolution or reduce the frame rate). In modern CE devices, however, the QoS reduction is not a choice, as the customers will not tolerate a low-resolution video playback on their TVs or presence of frame glitches if the device is assumed to output high-quality video without visual artifacts. As a result, new multimedia standards as well as high market competition motivate *hard real-time* scheduling of multimedia applications, so that application's output is *never* delivered later than a specific deadline [19].

The majority of hard real-time scheduling algorithms deal with applications modeled as a set of independent periodic or sporadic tasks [23]. Such models, however, do not comply with dataflow applications where the time an actor can start its execution depends on the availability of data tokens produced by other actors to its inbound communication channel(s). Therefore, the actors in Dataflow MoCs do not necessarily conform to periodic or sporadic task models. At the same time, it was analytically proven that embedded streaming applications modeled with Dataflow MoCs can be scheduled as a set of *strictly periodic tasks* [13] [14], meaning that each actor is scheduled as a task which is invoked at strict moments of time defined by its period. This means that a variety of hard real-time scheduling algorithms can be applied on an embedded streaming application given its dataflow model and Worst-Case Execution Times(WCETs) of each of its actors (computed with static analysis tools or profiling on the target platform [82]). As a result, if each task of a multimedia embedded application is invoked with a period calculated by a hard real-time scheduling algorithm, then the whole application is guaranteed to never miss the deadline.

2.5 Temporal Bugs in Embedded Streaming Applications

Hard real-time scheduling of streaming applications provides QoS guarantees for multimedia embedded systems. In return, it requires that the two following conditions are met:

1. Careful WCET analysis is performed as a preliminary step. It consists in calculating the upper bound on each actor's execution time given the available resources on the target hardware.

2. Scheduling is performed offline, i.e. before the target application actually executes. This way, the hard real-time scheduling algorithm can guarantee the availability of processing resources when a critical situation arises.

Unfortunately, each of these conditions is very difficult to meet on multimedia embedded systems.

Firstly, it is complicated to accurately predict an application's execution time on MPSoCs because these chips are conceived to be efficient rather than predictable. As was stated in Section 2.2, a whole assortment of optimizations are used in MPSoCs in order to improve the average-case performance of embedded software: from instruction prefetching to multi-level caches. This makes an accurate estimation of the WCET troublesome on MPSoC platforms.

Secondly, embedded multimedia systems are dynamic systems which are supposed to run multimedia applications upon a user request. In other words, it is the user who decides which combinations of applications (called *use cases*) to run, and at which moments of time. In these conditions, the offline scheduling of an application would need to take into account every possible temporal combination of any imaginable use case the application can be a part of, which is clearly unmanageable given the variety of software supported by modern CE devices.

Having these two restrictions of embedded multimedia systems, it comes as no surprise that the deadlines assigned by a hard real-time scheduling algorithm to an application's actors can be violated during the execution of various use cases on an MPSoC. Such violations are called temporal bugs to emphasize the fact that temporal constraints are not respected. As stated in Section 2.4, the temporal aspect is equally important for a real-time system as the functional one: its output is not only expected to be correct, but also to be delivered on time. Therefore, both temporal and functional debugging of real-time systems are equally important.

2.6 Complexities of Temporal Debugging

Temporal bugs are notorious for being hard to debug. They do not cause the system to crash, but make the quality of the output unacceptable. Temporal bugs often originate from unexpected interactions between the target application and other applications or Operating System (OS) processes running concurrently on an MPSoC. The difficulty of temporal debugging primarily stems from the failure of interactive debugging [5] – the most popular debugging technique in software development – to handle this type

[5] also known as start/stop debugging or "stop 'n' stare" debugging

of bugs. An interactive debugger, for example GDB [72], allows to stop program execution at user-specified breakpoints and inspect the state of the system. Interactive debugging of embedded systems is possible thanks to dedicated debugging ports, e.g. JTAG or Serial Wire, available on the development/evaluation boards used by the programmers to develop embedded software. Through such ports, a debugger running on the programmer's host machine can start, stop the embedded system, set breakpoints on instructions, and step through an embedded application's code [67]. Unfortunately, quite often this technology is of little use to resolve temporal bugs, as we explain with the two following observations.

- Interactive debugging relies on bug reproducibility. At the same time, it is highly unlikely that the same temporal bug occurs at the same moment of time when a use case is executed again. The reason is the non-deterministic nature of parallel applications [73], but more importantly, unpredictable scenarios of resource sharing among the applications from a particular use case. Moreover, stopping the system in order to inspect its state changes the real-time behavior of the system [74]. Therefore, a temporal bug may be simply non-reproducible if an interactive debugger is used.
- In case the bug is reproducible with an interactive debugger, there is no information on where to set a breakpoint and how to inspect system state. System activity responsible for a temporal bug may occur at any moment of time before the drop in QoS, and the developer has no means to know which particular variable must be inspected in order to understand the nature of the observed temporal bug.

Given these deficiencies of the interactive debugging, there exists an alternative debugging technique which consists in analyzing system states after the program has finished its execution, i.e. perform the debugging in a post-mortem fashion. Such technique is possible thanks to the execution tracing technology which we discuss next.

2.7 Execution Tracing Technology

Execution tracing consists in capturing system events during a program's execution and storing them into a text file (called a *trace*) on the developer's host machine. Once the program has finished its execution, the developer analyzes the trace in order to understand system behavior building up to the observed QoS problem.

In its simplest form, execution tracing is purely software based. It consists in *instrumenting* various software components by adding `print` statements in their source code in order to log the system's progress through the software code. More modern solutions implement libraries and kernel modules to automate code instrumentation through tracepoints [61], kprobes [43], kernel markers and so on. These are hooking mechanisms that provide static instrumentation which can be enabled at runtime. Some operating

systems, e.g. Linux or STLinux [5], offer these tracing solutions to software developers through their native tracing tools, e.g. LTTng [25] [4] for Linux or KPTrace [68] [3] for STLinux. But also, development frameworks such as GStreamer [76] come with pre-instrumented code allowing programmers to choose the necessary level of tracing. With these solutions, code instrumentation becomes transparent to programmers relieving them of rewriting the source code of software components. Unfortunately, purely software based tracing can add a considerable performance overhead to embedded software due to the latency introduced by the instrumentation code.

Starting from the early 2000s, some of the biggest semiconductor IP suppliers, such as ARM, ARC, MIPS and others, started to address the problem of the intrusiveness of software based tracing by introducing specific IP blocks which enabled purely hardware based tracing of SoC's components [79] [39] (e.g. CoreSight architecture from ARM [6] or SmartRT unit from ARC [7]). Once included in the final design of a SoC, such circuits capture a sequence of executed instructions and data accesses, compress this information and store it in on-chip memory modules before transferring the trace to the host machine either through the JTAG interface, or a dedicated trace port. The fact that tracing is performed in hardware makes it completely non-intrusive. At the same time, the amount of information contained in such traces is very large, as system activity is captured with instruction-level granularity. Indeed, with purely hardware based tracing software developers have no means to specify the particular events they wants to trace, i.e. to collect execution traces with event-level granularity.

To address the drawbacks of both aforementioned tracing approaches, there is a recent trend among the IP core suppliers to enable *hardware assisted software based tracing* (e.g. System Trace Macrocell from ARM [8]). With such solutions, software developers are allowed to specify events to be traced by reusing existing operating system's built-in tracing frameworks as well as performing application-level code instrumentation [53]. Instrumented events are then communicated to the dedicated hardware components through an OS driver, so that only instructions related to the instrumented code are captured, timestamped, stored in on-chip buffers, and finally transferred back to the host machine. This approach significantly reduces runtime overhead introduced by code instrumentation and makes execution tracing an invaluable technique to record an embedded system's activity without altering its temporal behavior.

Once a system trace is transferred to the host machine, it is the software developer's responsibility to analyze it. Essentially, a trace is a timestamped list of events executed on a particular SoC component. Consider Table 2.1 which shows a short excerpt from a KPTrace-generated trace obtained from a single core of ARM's Cortex-A9 CPU found in STiH416 MPSoC while it was running a video decoding application. Each line of the trace contains an event executed by the processor at a given timestamp. For example, line 6 denotes that a function call (type K) CPureSwQueueBuffer::ReleaseDisplayNode was

[6] http://www.arm.com/products/system-ip/debug-trace/

[7] https://www.synopsys.com/dw/ipdir.php?ds=arc-real-time-trace

[8] https://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/coresight-system-trace-macrocell.php

made by the process 1598 at timestamp 943920294.747273. Line 8 shows that a context switch (type *C*) has occurred, so that process 2569 started its execution at timestamp 943920294.747415.

Line	Timestamp	Process ID	Type	Event info
1	943920294.747078	0	C	0 1825
2	943920294.747086	1825	K	"stm_display_output_handle_interrupts"
3	943920294.747165	1825	K	"stm_display_output_get_connection_status"
4	943920294.747198	1825	C	1825 0
5	943920294.747261	0	C	0 1598
6	943920294.747273	1598	K	"CPureSwQueueBuffer::ReleaseDisplayNode"
7	943920294.747294	1598	K	"CHqvdpLitePlaneV2::PresentDisplayNode"
8	943920294.747415	1598	C	1598 2569
9	943920294.747625	2569	I	33
10	943920294.747635	2569	I	260

TABLE 2.1: An example of an execution trace

The increasing complexity of modern MPSoCs as well as of the use cases run on these chips make manual trace analysis a daunting task [83]. Software developers are confronted with too much system activity at any given moment of time in order to be able to grasp it and, what is even more, to find suspicious system behaviors related to the QoS problems. The size of a trace file largely depends on the number of traced events, the duration of tracing and the number of traced hardware components. In our experience, however, we have dealt with approximately 1GB of raw trace data generated per 2 minutes of execution on a dual-core ARM CPU. Therefore, the need for techniques to automate execution trace analysis in order to perform temporal debugging of multimedia embedded systems is higher as ever.

2.8 Conclusion

In this chapter, we have discussed what is hidden inside a modern multimedia embedded system, both hardware- and software-wise. This knowledge is essential to understand the nature of temporal bugs in multimedia systems, hence, to be able to come up with a technique to help developers in resolving this type of bugs.

All the components of multimedia embedded systems are there to answer the stringent requirements dictated by the consumer electronics market. On hardware level, SoCs, which have evolved to MPSoCs, deliver an unprecedented performance and rich functionality to modern multimedia devices while constantly shrinking their size as well as preserving their low price and energy consumption. On software level, the short time-to-market requirement is met by designing multimedia software using dataflow computational models, so that the tasks of manufacturing an MPSoC and developing applications to be run on it can be performed in parallel. Last but not least, the components of a multimedia application must be executed strictly periodically, so that the output of the whole system is never delivered late, and the quality of multimedia meets the high demands of the users of CE devices.

As we studied the innards of a modern multimedia embedded system, it became much clearer where the temporal bugs which degrade user experience are coming from. A large variety of possible combinations of multimedia applications as well as the presence of operating system processes that all run concurrently result in situations where a particular application fails to deliver its output on time. Hence, a QoS problem appears.

Finally, we have seen that standard debugging tools fall short to resolve temporal bugs, and execution traces can be a valuable source of information to use in temporal debugging. However, it remains up to software developers to navigate the masses of data contained in execution traces, and there is currently a great need to help developers in extracting pertinent to temporal bugs knowledge from execution traces.

Chapter 3

Detecting Anomalous Zones in Execution Traces

Execution tracing is a powerful technology to capture system activity on literally any granularity level: from function calls to memory accesses. It enables embedded software developers to record the events executed on an MPSoC in order to perform post-mortem analysis of system behavior. Such kind of analysis is especially useful when developers need to improve the QoS of a particular application. However, the process of identification of system activity that reduces the application's QoS, that is, the process of temporal debugging, is an extremely complex task. One of the main difficulties in temporal debugging is the absence of a starting point, i.e. a pair of particular events in program execution between which the analysis must be performed. Making an analogy with interactive debugging, it is like if the developer did not know where to set a breakpoint, and then had no clue on which variables to look at in order to spot anomalous values.

In this chapter, we show that, given an execution trace of a multimedia embedded system and a dataflow graph of the application with low QoS, it is possible to automatically track down the parts of the trace that contain abnormal system activity explaining the origin of the temporal bug. In Section 3.1, we provide an analytical explanation of how this can be done. In Section 3.2, we explain how the detection process is implemented in our temporal debugging approach SATM using parameter-less clustering (Section 3.2.1) as well as some statistical metrics (Section 3.2.2).

3.1 Propagation of Execution Delay in Dataflow Graphs

In this section, we argue that a streaming application with low QoS (i.e. a streaming application whose output was delayed at least once during system execution) that was modeled with the Dataflow MoC and scheduled under hard real-time constraints must

contain at least one actor that does not respect its period at runtime. We introduce the notion of *execution delay propagation* that justifies this argument.

Hard real-time scheduling of a dataflow graph G consists in finding out how to execute graph actors $a_i, i = 1..N$ as a set of strictly periodic tasks τ_i given actors' worst-case execution times on the target platform ($WCET_i$). Each τ_i is defined by a tuple $\tau_i = (S_i, D_i, P_i)$, where $S_i \geq 0$ is the start time of τ_i, $D_i \geq WCET_i$ is its deadline and $P_i \geq D_i$ is its period, so that τ_i is invoked at $t = S_i + mP_i, m = 0, .., \infty$ and executes for no longer than D_i [1].

A periodic schedule for G is called *valid* if it can be repeated infinitely, i.e. the invocation rate q_i of an actor-producer a_i is aligned with the invocation rate q_{i+1} of the corresponding actor-consumer a_{i+1}, so that the communication channel between them has bounded buffer capacity. This means that the schedule must derive task periods P_i, such that

$$q_1 P_1 = q_2 P_2 = \cdots = q_{N-1} P_{N-1} = q_N P_N = \alpha \tag{3.1}$$

where each q_i can be directly found from production and/or consumption properties of actors in the dataflow graph G [46] [17]. The product $q_i P_i$ designates the duration of an actor a_i's *iteration*, and, as can be seen from Equation 3.1, has the same value α for all the actors in G. Let's denote by a_{i+1} the actor that consumes the tokens produced by a_i. A valid periodic schedule guarantees that if

$$S_{i+1} = S_i + q_i P_i \tag{3.2}$$

then a_{i+1} will always find the required number of input tokens each time it is invoked [13].

Having the dataflow graph presented in Figure 2.5 and given the following actors' worst-case execution times: $wcet_1 = 5$, $wcet_2 = 3$, $wcet_3 = 2$, a valid periodic schedule would derive start times $S_1 = 0$, $S_2 = 24$, $S_3 = 48$, invocation rates $q_1 = 3$, $q_2 = 6$, $q_3 = 4$, periods $P_1 = 8$, $P_2 = 4$, $P_3 = 6$, and, hence, iteration duration $\alpha = 24$ (see Figure 3.1). Deadlines D_i, $wcet_i \leq D_i \leq P_i$, are calculated based on the amount of available resources on the target hardware platform [14].

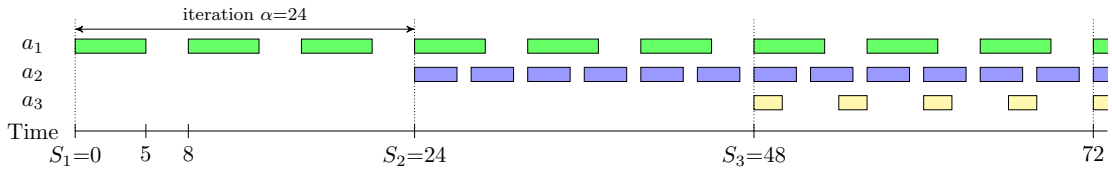


FIGURE 3.1: Valid periodic schedule for the CSDF graph from Figure 2.5

Proposition. A dataflow actor which has not respected its period after at least one invocation at runtime, i.e. the execution time was superior to its period, causes the execution delay to propagate through the dataflow graph resulting in the delayed output of the very last actor, that is, the delayed output of the whole application.

[1] In the rest of this section, the modeling term *actor* and its scheduling counterpart term *task* will be used interchangeably

Proof. Let's denote t_i^k the time of the k^{th} invocation of an actor a_i. Periodic scheduling of the dataflow graph G implies that

$$t_i^{k+1} = t_i^k + P_i. \tag{3.3}$$

If at e^{th} invocation, a_i's execution time $exec\_time_i^e$ turned out to be $exec\_time_i^e > P_i$, then $t_i^{e+1} = t_i^e + exec\_time_i^e > t_i^e + P_i$. It follows that $\forall c \geq e$: $t_i^{c+1} > t_i^c + P_i$, i.e. the subsequent invocation times of a_i are shifted by $delay_i^e = exec\_time_i^e - P_i > 0$. As the result, the subsequent iteration of a_i is delayed as well.

It can be seen from Equations 3.1 and 3.2 that the start time of a_{i+1}'s p^{th} iteration is equal to the start time of a_i's $(p+1)^{th}$ iteration. Thus, if a_i's $(p+1)^{th}$ iteration is delayed, then a_{i+1}'s p^{th} is delayed as well. The same applies to all the a_j, $i \leq j \leq N$, which means that the p^{th} iteration of a_N is shifted by $delay_i^e$, hence, application's output is delayed by $delay_l^e$. □

Figure 3.2 helps to understand the notion of execution delay propagation using the schedule from Figure 3.1. As a_1's e^{th} invocation took longer than P_1 to execute, $delay_1^e = exec\_time_1^e - P_1$ was introduced to the pipeline, which propagated through a_2 and a_3 actors resulting in the delayed output of the application.

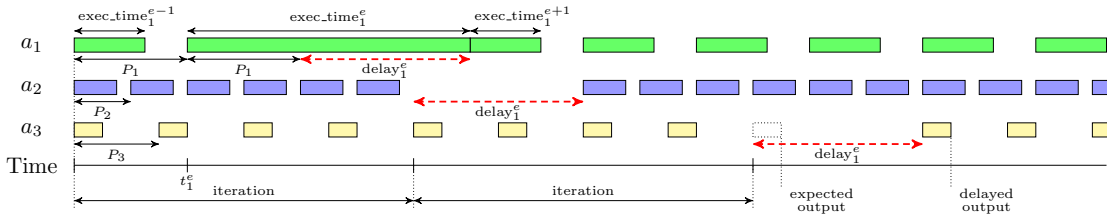


FIGURE 3.2: Illustration of the propagation of execution delay in a dataflow graph

The proposition shows that in order to explain the reason the application output was delayed, it is essential to find the most upstream actor in the application's dataflow graph which had not respected its period at least once.

3.2 Mining Actors' Periods from Execution Traces

In order to identify delayed invocations of an actor, one needs first to determine the actor's period. Although it is possible that the person who performs the debugging is aware of the periods of all the dataflow actors (e.g. because the scheduling is hard-coded into the program, or the information from the scheduler is easily accessible), we argue that automatic extraction of the actors' periods from the execution trace provides multiple benefits. Firstly, it relieves the developer from the need to manually navigate through the trace looking for the parts where the period of a given actor is not respected. Secondly, it makes SATM suitable to the developers who are unaware of the periods computed by the scheduling algorithm, which is often the case in industry where the

performance debugging teams are separated from the software coding teams. Therefore, the first stage in SATM consists in mining actors' periods from the execution trace. The most upstream actor, that is, the one situated the closest to the dataflow pipeline's head, that exhibits significant delays in its invocation rate is signaled for further analysis addressed in Chapter 4.

We firstly provide the necessary formalism. Let $\mathcal{E} = \{e_1, e_2, .., e_m\}$ be a set of all system and application *events* that the developer has decided to trace. An event can be a particular interrupt, like *Interrupt168*, or a function call executed in the context of a specific thread, e.g. *sys_read[process23]*, or else a context switch from one thread to another, like *__switch_to[process23, process351]*. Each line of a trace file contains a timestamped *occurrence* of an event e that started its execution at timestamp ts: $occ = (ts, e)$; $ts \in \mathbb{N}, e \in \mathcal{E}$. A *trace* D, thus, can be represented as a sequence of events' occurrences $D = \langle occ_1, occ_2, \ldots, occ_l \rangle = \langle (ts_1, e_1), (ts_2, e_2), ..., (ts_l, e_l) \rangle$ where $\forall i \in [1, l]$ $ts_i \in \mathbb{N}, e_i \in \mathcal{E}$, and the events' occurrences are ordered by the timestamps. For example, $\langle$(1, $Interrupt168$),(4, sys_read [$process23$]), (6,$\_\_switch\_to[process23, process351]$)$\rangle$ is a simple trace of length $l = 3$.

The mapping between a dataflow actor and the corresponding event in the execution trace is done in the following way. If a given actor is represented by a single function, then each appearance of this function in the trace is considered as an actor's occurrence. In case the actor was programmed as a software module, the appearance of the first function executed in the context of this module represents the actor's occurrence. In the following, we explain how SATM detects the period of a given event.

A naïve approach to find the period of a given event, or conclude that the event shows aperiodic behavior during the application's execution, would consist in extracting all the occurrences of the event from the execution trace and comparing the time intervals between the successive event's occurrences. However, the preemptive behavior of multitasking OSs running on MPSoCs does not allow to apply this approach to discover events' periods. As an OS can preempt the event's execution in favor of other processes and system tasks, the trace may contain several event's occurrences that semantically belong to a single scheduled event's invocation. Consider Figure 3.3 which presents a visualization of an excerpt from an execution trace captured in a time window from 15 to 22 seconds from the beginning of trace recording. In this excerpt, we can observe executions of an event e_i showed as filled rectangles. Suppose that we know beforehand that the event's period calculated by the scheduling algorithm is equal to 2.5 seconds, and it was always met during the execution. The trace contains 11 occurrences of e_i in this time window. So, if we apply the aforementioned naïve approach by extracting time intervals between consecutive event's occurrences, which are {0.3, 0.5, 0.3, 1.4, 0.6, 0.4, 1.5, 0.5, 0.3, 0.3}, the lack of a constant value among them would signal non-periodic invocation rate of the event e_i, which is wrong. In fact, such simple comparison fails to infer that short intervals are caused by the preemption, while the significantly longer ones correspond to the invocation rate of the event. Therefore, we need a preprocessing step to automatically group event's occurrences that are temporally close to each

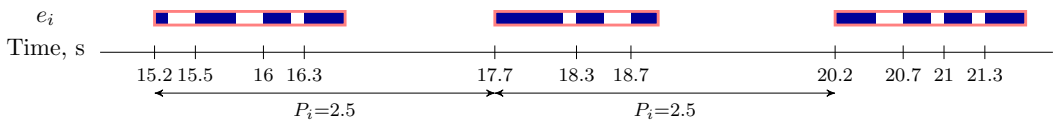


FIGURE 3.3: An excerpt from an execution trace showing the occurrences of an event e_i as filled rectangles and its invocations scheduled with a period $P_i = 2.5$ seconds as hollow rectangles.

other. With respect to the Figure 3.3, such preprocessing would consist in grouping filled rectangles into the hollow ones.

The next two sections show how SATM performs automatic grouping of event's occurrences by applying a parameter-less clustering, and how it detects the event's period once the event's occurrences have been grouped.

3.2.1 Clustering Event's Occurrences

Grouping similar objects is the goal of *cluster analysis* or *clustering*, a branch of data mining. Cluster analysis groups objects based only on the information found in the data that describes the objects and their relationships. Such data is represented as a set of attributes, and the size of this set is called the *dimensionality* of data. The goal of clustering is that the objects within a group are similar to each other and different from objects in other groups [75]. In our case, the objects are the occurrences of a particular event in the execution trace. Each occurrence has a single attribute: its timestamp. Therefore, the relationship between the occurrences is expressed as the difference between their timestamps. This way, the less time has passed between two occurrences, the more similar they are. Our goal is to cluster temporally close occurrences into the same group which will denote a single event's occurrence for the future analysis. The requirement that we impose on clustering is its fully automatic behavior, i.e. no user-defined parameters are allowed. Interestingly, the vast body of clustering algorithms deals with high-dimensional data and various manually set thresholds, leaving automatic one-dimensional clustering without proper attention. In [22] Cooper et al. propose an algorithm for one-dimensional clustering of digital photo collections. In their case, objects were individual photos characterized by a single attribute: the time they were taken. The goal was, therefore, to group photos that were taken close in time into photo collections without user intervention. With only a slight enhancement, we were able to apply their time-based similarity analysis to accurately and automatically cluster an event's occurrences, as we explain next.

Building a similarity matrix

The first step in clustering event's occurrences consists in building their similarity matrix S. For this, we firstly extract all the N occurrences of the target event e from the execution trace and sort them in the ascending order of their timestamps: $\langle(ts_1, e),$

$(ts_2, e), \ldots, (ts_N, e)\rangle$, $ts_1 < ts_2 < \cdots < ts_N$. The similarity matrix S will then have the dimension $N \times N$ and contain the similarity measures between each pair of occurrences computed as $S(i,j) = exp(-|t_i - t_j|)$. The bigger values, thus, will indicate more similar occurrences. It is easy to see that S is a symmetric matrix. Figure 3.4 shows a similarity matrix for the event's occurrences from the Figure 3.3 as well as its representation as a heat map, where the darker the cell is – the more similar corresponding occurrences are.

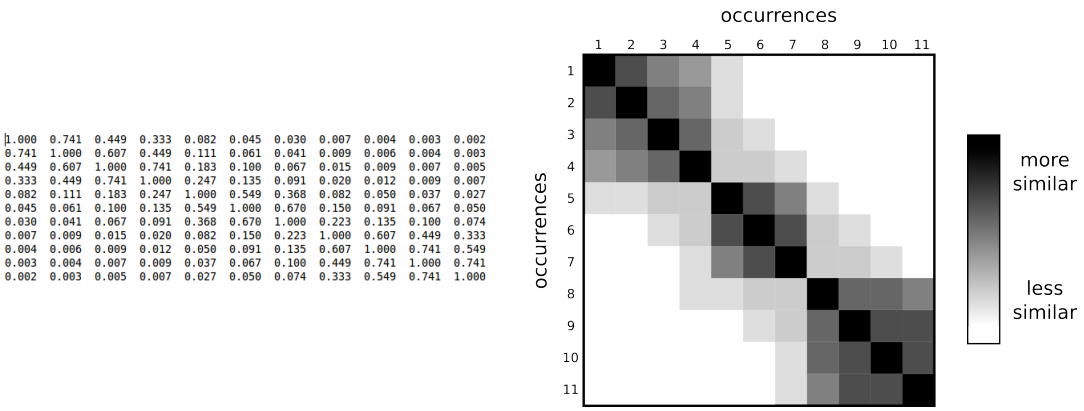


FIGURE 3.4: Similarity matrix for the event's occurrences from Figure 3.3

Discovering clusters' borders using novelty scores

Once the similarity matrix is constructed, SATM proceeds to detect the boundaries between the groups of similar event's occurrences by computing the novelty score for each event's occurrence. The novelty score quantifies the similarity of the groups of occurrences seen both before and after the given occurrence [30]. This way, if a given occurrence has a big novelty score, it signifies that the next occurrence belongs to a different cluster. It is worth mentioning that a cluster can, of course, consist of a single event's occurrence. Consider the heat map for the similarity matrix from Figure 3.4 where one can easily distinguish three dark-colored clusters of similar occurrences. Indeed, the boundaries between the clusters of event's occurrences are the centers of checkerboard patterns along the main diagonal. We would, thus, expect the novelty scores of the occurrences occ_4, occ_7 and occ_{11} to have bigger values than the novelty scores of other occurrences.

Foote [30] proposed a method to compute novelty scores by correlating a Gaussian-tapered checkerboard kernel along the main diagonal of a similarity matrix. *Correlation* here denotes the sum of the element-by-element product of the given kernel and the extracted submatrix of the same size (see Equation 3.4). In order to understand how such correlation allows to detect clusters' boundaries, let's consider the following example of a gaussian-tapered kernel of size 4×4, as well as its decomposition into two terms:

$$\begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

The first term measures self-similarity of the upper-left and lower-right 2×2 regions of a submatrix it is correlated to. The result of such correlation will have a small value if both regions are *self-similar*, i.e. the temporal differences between the occurrences in each region are small. The second term measures the cross-similarity between the same two regions. The result of its correlation with a given diagonal submatrix of the similarity matrix will have a small value if the regions are substantially *similar to each other*. The difference of the two values will have a big value if the upper-left and lower-right 2×2 regions are self-similar but different from each other.

Kernels can be smoothed to avoid edge effects using windows that taper towards zero at the edges. Figure 3.5 presents the kernel we used in SATM.

Finally, the correlation is achieved using the following formula:

$$\nu_i = \sum_{l,m=-p}^{p} S[i+l, i+m]g[l, m], \tag{3.4}$$

where g is the kernel, p is its size, and i is the index of the current event's occurrence for which the novelty score is calculated.

It is important to note that all the elements of a similarity matrix situated further than p positions away from the main diagonal are never used in the computation of the novelty score. Therefore, to reduce storage and computation requirements of the similarity matrix building step, SATM computes only $S[i+q][i+r]$, $\forall q, r$, $0 \leq q \leq p$, $0 \leq r \leq p$ portion of the similarity matrix S for every event's occurrence occ_i, $0 < i \leq N$.

Applying Formula 3.4 on every element situated on the main diagonal of the similarity matrix S will give us the novelty score for each event's occurrence. The next goal is to identify the occurrences with significantly high novelty score; these occurrences will denote clusters' boundaries. Cooper et al. [22] use a manually chosen threshold to do that. We, on the other hand, wanted the clustering step to be completely automatic.

Detecting significantly high novelty scores using Otsu's method

Back in 1979, Nobuyuki Otsu proposed a method to automatically transform gray-level images to binary ones [60], which is since widely used in computer vision and image processing. Otsu's method allows to detect if a given gray-level pixel is closer to being white or black. Interestingly, this method can be applied to any problem requiring to automatically find a threshold between two groups of objects, so that the variance between the objects in the same group is minimal.

$$\begin{bmatrix} 0.1 & 0.5 & -0.5 & -0.1 \\ 0.5 & 1 & -1 & -0.5 \\ -0.5 & -1 & 1 & 0.5 \\ -0.1 & -0.5 & 0.5 & 0.1 \end{bmatrix}$$

FIGURE 3.5: Gaussian-tapered kernel used in SATM

Consider our problem of detecting event's occurrences with significantly high novelty scores, which we will call the *boundary occurrences*. Each boundary occurrence will be the last element of a cluster it forms with the preceding occurrences (or the only element of a cluster if the preceding occurrence has a significantly high novelty score as well). Using a gaussian-tapered kernel, we have computed the novelty score for each event's occurrence. Otsu's method now allows to iterate through all *possible* thresholds between these novelty scores, and choose such threshold that would split the novelty scores into two groups: one with boundary occurrences, and another one with the remaining non-boundary ones, such that the variance among the elements in each group is minimal.

Consider the novelty scores computed with Equation 3.4 for the event's occurrences from Figure 3.3 using the kernel from Figure 3.5: 0.78, 1.50, 0.92, 2.66, 1.49, 1.09, 2.73, 1.35, 1.00, 0.81, 2.73. If we take a threshold equal to 1.4 the sum of variances of the two groups of novelty scores is 2.229. On the other hand, a threshold equal to 1.7 gives the value 0.598 which is smaller, hence, this threshold is better.

In practice, SATM starts from the threshold equal to the minimum value of the novelty score and iterate with a step of 0.1 [2] until the maximal value of the novelty score is reached. The threshold with the minimal value is chosen, and all the occurrences with the novelty score higher than the threshold are treated as boundary occurrences.

Clustering on different temporal granularities

Different events execute in the embedded system at different temporal granularities. For example, a network interrupt will have thousands of occurrences during a few minutes of trace capture, while a process that dumps data from memory to the external hard drive executes once per several minutes. It is, thus, important that the clustering of the event's occurrences is performed on various temporal granularities, so that a set of clusters found on the most appropriate granularity level can be chosen. The most appropriate would be a level where the sum of similarity measures between each pair of elements from the same cluster has the highest value, while the sum of similarity measures between each pair of elements from the adjacent clusters has the lowest value. Cooper et al. [22] exploit scale-space theory in order to perform such multi-scale clustering.

Scale-space theory is a basic tool to analyze structures at different scales; it is used extensively in signal processing and computer vision communities. This theory was developed on the premise that the human eye perceives same objects differently on different scales. For example, as you read this text, you consider every letter as a separate entity, as a single cluster. At the same time, it will not be problematic for you to count the number of words in a given line, as you would force your retina to slightly blur the letters in a single word so that the spaces between the words are easily detectable. The letters of the same word will be, thus, clustered into the same group to facilitate the detection of spaces. The same applies on a coarser level, if you are asked to count the number of paragraphs on a single page. Detecting objects on different scales

[2] such step value is sufficient thanks to the smoothing applied in constructing the similarity matrix S, as well as using multiple temporal granularities discussed in the next subsection.

is completely natural to us. Computers, on the other hand, cannot do this without user-specified details on how to reproduce any given image on different scales. The scale-space theory provides a solution to this problem. In short, its main idea is to use gaussian space smoothing on various scales to blur images and remove detail and noise.

Returning to our problem of detecting the clusters of event's occurrences at different temporal granularities, we apply Cooper et al's approach for "blurring" temporal differences between individual elements. It consists in constructing a similarity matrix S for each chosen temporal granularity K using the following formula:

$$S_K(i,j) = exp(-\frac{|t_i - t_j|}{K}), \tag{3.5}$$

For execution traces, a good choice of granularities is: nanosecond, microsecond, millisecond, and second. Tracing tools do not normally capture temporal differences inferior to a nanosecond, and it is highly unlikely that an event is executed so rarely that its clusters cannot be correctly detected on the minute scale. The value of K would depend on the temporal resolution of the trace tool. If the timestamps in the execution trace are given in nanosecond resolution, then $K = \{1,\ 10^3,\ 10^6,\ 10^9\}$; if it has millisecond resolution, then $K = \{1/10^6,\ 1/10^3,\ 1,\ 10^3\}$, and so on. Once SATM constructs a similarity matrix S for each K, it detects clusters of event's occurrences at every scale K using the method described previously in this section. Having the clusters' boundaries on each scale $\mathcal{B}_K = \{b_1, \ldots, b_{n_K}\}$, the last action is to determine which temporal resolution results in the most pronounced clusters. It is done using a confidence score computed for each scale K:

$$\begin{aligned} C(\mathcal{B}_K) = & \sum_{l=1}^{|\mathcal{B}_K|-1} \sum_{i,j=b_l}^{b_{l+1}} \frac{S_K(i,j)}{(b_{l+1} - b_l)^2} \\ & - \sum_{l=1}^{|\mathcal{B}_K|-2} \sum_{i=b_l}^{b_{l+1}} \sum_{j=b_{l+1}}^{b_{l+2}} \frac{S_K(i,j)}{(b_{l+1} - b_l)(b_{l+2} - b_{l+1})} \end{aligned} \tag{3.6}$$

The first term quantifies the average within-class similarity between the event's occurrences within each cluster. The second term quantifies the average between-class similarity between the event's occurrences in adjacent clusters. By negating the second term, the confidence measure combines each cluster's average self-similarity and the dissimilarity with the adjacent cluster(s). The temporal granularity K on which the clusters' boundaries have the highest confidence score is chosen as the final set of clusters.

Complete algorithm for temporal clustering of event's occurrences

Finally, we provide the description of the algorithm to cluster event's occurrences based on the work of Cooper et al. [22] which is used in our temporal debugging approach SATM.

Function `ClusterActorOccurrences`(T, e, K)*: Cluster the occurrences of e found in the trace T using the best temporal resolution from K.*

Data: T: execution trace, e: target event, K a list of temporal resolutions to consider.
Result: $\mathcal{B}$: a set of clusters' boundaries.
foreach *occurrence of e in T:* $occ_i = (ts_i, e)$ **do**
  insert ts_i in the sorted in the ascending order array Occ
end
foreach $k \in K$ **do**
  S_k = similarity matrix for Occ (see Formula 3.5)
  ν_k = set of similarity scores for Occ (using Formula 3.4 with the kernel from Figure 3.5)
  $\mathcal{B}_k$ - clusters' boundaries computed with `OtsuMethod`(ν_k) [60]
end
$\mathcal{B}$ = argmin($C(\mathcal{B}_k)$) (see Formula 3.6)

Algorithm 1: Algorithm for clustering event's occurrences.

3.2.2 Detecting Violations of Event's Period

Once the event's occurrences found in the execution trace are clustered, SATM proceeds to discover the event's period. It collects the values of time intervals between the pairs of consecutive event's clusters and measures how different from each other these intervals are. In statistical terms, having a distribution of time intervals, SATM determines its central tendency and measures its dispersion. On the one hand, this allows to find the event's period (equal to the central tendency if the dispersion is small), or conclude that the event is not periodic (if the dispersion is large). On the other hand, the parts of the trace where the event does not respect its period (outliers of the distribution) can be easily detected.

As a measure of the distribution's central tendency we use the *median* which is equal to the middle element of the distribution. An appealing property of the median is its high resistance to outliers: for a distribution where half of the elements have the same value V, and another half - arbitrarily big values, the median returns V as the value of central tendency. As the measure of dispersion we use the Quartile Coefficient of Dispersion (QCoD), also known as Quartile Variation Coefficient (QVC) [89]. It is a dimensionless measure, i.e. it has no units, and thus we can use the same dispersion threshold for distributions with different central tendency values. Quartile coefficient of dispersion is defined as

$$QCoD = \frac{Q_3 - Q_1}{Q_3 + Q_1}, \tag{3.7}$$

where Q_1 and Q_3 are the first and the third quartiles accordingly.

In case the value of $QCoD$ for the distribution of time intervals between the event's clusters is small enough to consider the event as a periodic one [3], we proceed to detect the gaps in the event's periodicity, i.e. time intervals that are much bigger than the event's period. The precise "much bigger" value can be obtained from the inter-quartile rule for outliers [77]: distribution elements that fall above $Q_3 + 3 * IQR$ can be considered as outliers, where $IQR = Q3 - Q1$ is the inter-quartile range.

Consider the following timestamps of the clustered occurrences of an actor a represented in the execution with the event e: $t_1 = 45$, $t_2 = 75$, $t_3 = 104$, $t_4 = 134$, $t_5 = 164$, $t_6 = 352$, $t_7 = 382$, $t_8 = 413$, $t_9 = 443$, $t_{10} = 538$, $t_{11} = 568$. A corresponding distribution of time intervals between the adjacent occurrences is

$$\sigma = \{30, 29, 30, 30, 188, 30, 31, 30, 95, 30\},$$

and the median equals to 30. Next, we decide if this value can be considered as the event's period, i.e. whether most of the time intervals tend to be equal to the median, using QCoD dispersion measure (Equation 3.7). $Q_1 = 30$, $Q_3 = 31$, and $QCoD = 1/61 \times 100\% \approx 1.6\%$. A small value of $QCoD$ implies that the distribution σ is well centered on its median, even with the presence of two big outliers. Hence, we infer that the event e is periodic with the period equal to 30. We then apply the inter-quartile rule, which detects 95 and 188 to be outliers in σ, as these values fall above $Q_3 + 3 * IQR = 34$.

Assume that a is the most upstream actor from the application's dataflow graph which has outliers in its distribution of occurrences in the execution trace. It is essential to discover the cause of the presence of these outliers, which, as proved in Section 3.1, result in the reduced QoS of the multimedia embedded system.

Having the timestamps of a's (clustered) occurrences and the list of its outliers, an execution trace D can be split into two sets: D_{neg}, where the target actor is invoked periodically (outlier-negative), and D_{pos}, where its period is broken (outlier-positive). Using the example above, $D_{neg} = \{ [0, t_1]; [t_1,t_2]; [t_2,t_3]; [t_3,t_4]; [t_4,t_5]; [t_6,t_7]; [t_7,t_8]; [t_8,t_9]; [t_{10},t_{11}]; [t_{11},t_{last}] \}$, $D_{pos} = \{[t_5,t_6]; [t_9,t_{10}]\}$ where $[t_i, t_j]$ denotes the part of the trace between the timestamps t_i and t_j and is called a *subtrace*, while t_{last} is the timestamp of the last event in the trace D.

In the next chapter, we deal with the problem of detecting system activities that characterize the D_{neg} parts of the execution trace, but not the D_{pos} ones. We argue that such system activities represent strong pointers to the temporal bug present in the system from which the execution trace was obtained, and are an extremely useful information for the developers performing temporal debugging.

[3] based on our experience in characterizing events found in execution traces, $QoCD < 10\%$ is a good indicator of the event's periodicity

3.3 Conclusion

In this chapter, we presented a method to extract anomalous zones from an execution trace that need to be analyzed in order to perform temporal debugging of multimedia embedded systems.

As we have argued in Section 3.1, the QoS of a multimedia embedded system is degraded when the last actor in the target application's dataflow graph a_N does not respect its deadline, hence, delivers delayed output. This can be caused either by the delayed input from the next-to-last a_{N-1} actor, or, in case the input was available in time, by some anomalous system activity during the a_N actor's last execution. We then showed how the most upstream actor that has not respected its deadline is detected in out temporal debugging approach SATM using a combination of parameter-less clustering (Section 3.2.1) and a few statistical measures (Section 3.2.2).

The cluster analysis is an essential part of SATM due to the fact that an actor's period cannot be found directly from the trace because of the preemptive behavior of the OSs running on MPSoCs. To group the actor's occurrences in the execution trace on the temporal axis, we have enriched the algorithm proposed by Cooper et al. [22] with Otsu's method [60], and have got a clustering algorithm that does not need any manually set parameters. In practice, SATM clusters actor's occurrences with high accuracy. The reason of this is twofold. On the one hand, the actor's period is normally quite bigger than its worst-case execution time. On the other hand, valid use cases usually do not overutilize the resources of the underlying embedded platform. Hence, no heavy preemption takes place, and the actor's occurrences related to a single invocation are compactly grouped on the temporal axis.

Once the actor's occurrences are clustered, the quartile coefficient of dispersion is used to determine the actor's period, in case the actor exhibits periodic behavior. If the given actor has missed its deadline at least once, the corresponding part(s) of the execution trace are extracted and are used to detect the system activity correlated to the deadline miss. The goal of the next chapter is to show how such system activity can be discovered using data mining algorithms.

Chapter 4

Mining Abnormal System Activity from Execution Traces

Temporal debugging is a notoriously difficult task, primarily because a software developer is not allowed to pause a program's execution in order to inspect the state of the system. Fortunately, the developers can get a log of the entire system's activity during a problematic system execution thanks to the tracing technology. On the other hand, a huge amount of raw data in the execution trace makes searching for abnormal system activity related to the temporal bug incredibly hard. In Chapter 3, we showed how to detect the parts of the trace where the buggy system activity is present, using the domain knowledge on both software and hardware aspects of embedded systems found in consumer electronic devices. At the next step, ideally, the developer herself notices some unusual system behavior after examining the suspicious parts of the trace. However, in order to do this, the developer must possess a deep understanding of the whole system. Firstly, she needs to know exactly what constitutes normal system activity at execution time; that is, have a knowledge about each component of the underlying system (OS processes, drivers, all the running applications, etc.). Secondly, the developer must be able to differentiate a harmlessly diverged system activity from the one that may result in delayed output of the target application. On top of that, the multi-core components of modern MPSoCs make manual analysis of the totality of system behavior even more difficult as several execution threads must be analyzed simultaneously. For these reasons, developers need a tool that can detect unusual system behavior in the suspicious parts of an execution trace in an automatic way.

In this chapter, we investigate how the task of discovering unusual system behavior correlated to the observed temporal bug can be accomplished using sequential pattern mining algorithms. In Section 4.1, we establish a relationship between the problems of temporal debugging and contrast pattern mining. Next, in Section 4.2, we consider different ways to mine contrast patterns, focusing on their extraction from the set of frequent patterns in D_{pos} dataset. At the same time, we evaluate the performance of

running the relevant state-of-the-art pattern mining algorithms on execution trace data. Finally, in Section 4.3, we take a closer look at the execution trace data from the data mining perspective in order to understand why mining the complete set of contrast patterns from real-world traces turns out to be such a hard task.

4.1 Detection of Abnormal System Activity as a Pattern Mining Task

In Chapter 3, we showed how SATM detects anomalous parts of an execution trace relying on the domain knowledge of the embedded systems from consumer electronic devices. As a result of its first stage, SATM splits the original trace into two sets of subtraces: D_{pos} containing the anomalous parts of the trace, and D_{neg} containing the normal parts of the trace. The goal of the second stage of SATM is to automatically find what makes the dataset D_{pos} different from the dataset D_{neg}. In other words, we would like to understand which system activity is present in anomalous parts of the trace, but not in the normal ones.

We proceed from the principle, that if the system behaves in a certain way only in the subtraces from D_{pos}, then this behavior is correlated to the reason the system exhibits low QoS. Note that one cannot state definitely if such behavior is itself an origin of temporal bugs, or a consequence of some system activity that either occurred while all the components of the application had correct temporal behavior, or comes from the untraced parts of the system. However, due to the tendency to appear only in the anomalous parts of the trace, such suspicious system activity provides an extremely valuable information to software developers, so that they can know which particular part of the system is involved in the temporal bug. Consider, for example, a situation where some process is constantly writing data to a shared data structure in the subtraces from D_{pos} but not in the subtraces from D_{neg}. Once this behavior is detected as an abnormal one, it is necessary to verify what is the reason that the particular process performs those writes. The problem can then be assigned to a particular group of developers, making temporal debugging a methodical process rather than a trial-and-error one.

A relevant way of characterizing abnormal system behavior consists in discovering (or, **mining**) unusual combinations of raw system events appearing in the execution trace. A particular combination of system events can be considered unusual if it appears mostly/only in the anomalous parts of the trace (i.e. subtraces from D_{pos} dataset), and not during the normal system operation (i.e. the D_{neg} dataset). Each subtrace can be viewed as an ordered collection of events executed on the target system. The task of discovering groups of features that appear together in a given set of collections of features is addressed by the subfield of data mining called *pattern mining*.

Originally, pattern mining was introduced as a task of uncovering interesting patterns of the products purchased together by customers in a supermarket [8]. That is why, the terms used in pattern mining field come from the market basket analysis. An **item** ($\boldsymbol{e}$) is

a particular entity represented symbolically, like a specific product available for purchase in a supermarket. A complete set of available items (e.g. products in the supermarket) is called an **alphabet** ($\boldsymbol{I} : e_i \in I, i \in [1, |I|]$, where $|I|$ is the size of the alphabet). A **transaction** ($\boldsymbol{T} : (e_1, e_2, \ldots, e_{|T|}), e_i \in I, i \in [1, |T|]$) denotes a collection of items having some semantic relation to each other (e.g. items bought by a single customer at a particular visit to the supermarket, or all items bought in the supermarket during a particular time span). A transaction is considered **sparse** when the ratio of distinct items occurring in it to the size of the alphabet is small. Conversely, a transaction is **dense** when this ratio is big. A **transactional database** ($\boldsymbol{DB} = \{T_1, T_2, \ldots, T_{|DB|}\}$) is a set of all recorded transactions. A **pattern** ($\boldsymbol{p} : (e_1, e_2, \ldots, e_{|p|}), e_i \in I, i \in [1, |p|]$) is a group of items that is tested against some interestingness measure in the given transactional database DB. A pattern containing l items (i.e. $|p| = l$) is called an ***l*-pattern**. In case the interestingness measure is expressed as the frequency with which a pattern occurs in the transactional database DB, the **support** ($\boldsymbol{sup_{DB}(p)} = |\{T\} \mid occ(p, T)^1|$) denotes the number of transactions containing a specific pattern. If in order to be interesting, a pattern must be **frequent** in the transaction database DB, then the mining process needs **minimum support** threshold (min_sup), so that only such patterns p that $sup_{DB}(p) \geq min\_sup$ are mined. It is common to express the support in percentages, denoting the relative number of transactions containing the target pattern. For example, $min\_sup = 100\%$ means that we are interested only in the patterns occurring in all the transactions from a given dataset.

Mining the patterns that satisfy a given interestingness measure is a non-trivial task. If mining is performed in a naïve way, one needs to consider the complete **search space**, that is, all possible patterns, and then verify the value of interestingness measure for each member of the search space across all the transactions. Such approach is, obviously, too computationally expensive even for transactional databases with relatively small number of short transactions and modest size of the alphabet. Fortunately, data mining field enjoyed an important number of algorithms which improved computational complexity of pattern mining task while allowing to mine different kinds of patterns with various interestingness measures [6]. High diversity of the application areas of pattern mining algorithms is explained by the ability of these algorithms to mine interesting patterns from any symbolic dataset. In the rest of this section, we express the task of detecting anomalous system behavior from two sets of execution traces as a pattern mining task.

Transforming an execution trace represented as a set of subtraces into a transactional database is a straightforward task. The only necessary preprocessing consists in dropping events' timestamps from each subtrace. This way, D_{pos} and D_{neg} datasets can be viewed as transactional databases where each transaction contains an ordered collection of system events.

The task of discovering abnormal system activity in the D_{pos} dataset then becomes a task of mining patterns of system events that appear in transactions from D_{pos}, but not in transactions from D_{neg}. Looking for patterns that represent the differences between

[1] the precise definition of $occ(p, T)$ function depends on the type of the pattern p

the given datasets is the goal of **contrast pattern mining** [26]. Contrast patterns proved to be useful in various applications, such as capturing discriminative gene group interactions, building accurate classifiers, constructing clusters without distance functions, and others.

The choice of the type of contrast patterns is crucial. On the one hand, patterns must be informative enough to express system behavior correlated to the observed temporal bug. On the other hand, they should not be too complex to mine from real-world execution trace data. Consider the following list of pattern types which can be used to represent a contrast between D_{pos} and D_{neg} datasets, ordered by increasing expressibility, but also mining complexity:

- An *itemset* A is, at its name suggests, a set of items: $A = \{e_1, e_2, \ldots, e_N\}$, $e_i \in I$, $i \in [1, N]$. Therefore, there is no order relation between the items, and a single item can appear only once in an itemset.
- A *sequence* S is a totally ordered collection of items where each item can appear multiple times: $S = \langle e_1, e_2, \ldots, e_N \rangle$, $e_i \in I$.
- An *episode* E is a partially ordered collection of items where each item can appear multiple times. For example, an episode $\langle e_1, \{e_2, e_3\} \rangle$ means that both e_2 and e_3 must occur after e_1, but no order is imposed between e_2 and e_3.
- A *temporally annotated episode* T is an episode enriched with the information on when a particular item started and finished its execution. That is, each $e_i \in T$ is expressed as a tuple (e, t_s, t_f), where e denotes e_i's corresponding system event, while t_s and t_f denote the timestamps at which e_i started and finished its execution.

Given a single transaction, the number of all possible patterns in it increases with the expressibility of the patterns' type. This means that there will exist much more sequential patterns than itemsets, much more episodes than sequential patterns, and much more temporally annotated episodes than general episodes. The least computationally expensive task, therefore, is contrast itemset mining. However, the inherent order of events in execution traces cannot be captured with itemset patterns. It is, therefore, not possible to express interleavings of events' executions, such as deadlocks, with itemsets. Sequential patterns, on the other hand, allow to capture various execution scenarios where the order as well as the number of occurrences of a particular event is important. If contrast sequence mining was a comparatively easy task, it would be even better to mine contrast episodes. Episodes are well adapted to parallel nature of embedded hardware, as they allow to express both the order between sequentially executed events, as well as concurrency of system events run on different cores. Finally, the temporally annotated episodes would allow to mine contrasts characterized by the duration of events' execution, or the latency between events' executions. Unfortunately, as we will see in the rest of this chapter, mining contrast sequences from execution traces is already an extremely complex task. In this thesis, therefore, we focus on mining contrast sequential patterns, or simply, contrast sequences.

Next, we present some definitions necessary to formally describe the pattern mining problem one needs to solve in order to detect anomalous behavior in execution traces. Figure 4.1 shows a running example of the D_{pos} and D_{neg} datasets which we will use throughout this chapter to familiarize the reader with pattern mining terms.

D_{pos}

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
T_1 :	⟨ b,	c,	a,	d,	b,	a,	d,	b ⟩
T_2 :	⟨ c,	a,	b,	a,	d,	e,	b,	d ⟩

D_{neg}

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
T_1 :	⟨ b,	c,	d,	b,	a,	d,	a,	d,	b ⟩
T_2 :	⟨ c,	e,	b,	d,	a,	d,	b,	a,	b ⟩

FIGURE 4.1: A running example of D_{pos} and D_{neg} datasets.

Definition 4.1. Given a dictionary of items I, a **sequence** $\boldsymbol{s}$ is an ordered list of items, denoted as $s = \langle e_1, e_2, \ldots, e_m \rangle$ where $e_i \in I$ for $1 \leq e_i \leq m$.

Definition 4.2. A sequence $s' = \langle a_1, a_2, \ldots, a_n \rangle$ is a **subsequence** of another sequence $s = \langle b_1, b_2, \ldots, b_m \rangle$, denoted as $s' \sqsubset s$, if there exist integers $1 \leq i_1 < i_2 < \cdots < i_n \leq m$ such that $a_1 = b_{i_1}$, $a_2 = b_{i_2}$, $\ldots$, $a_n = b_{i_n}$. Similarly, s is a **supersequence** of s', denoted as $s \sqsupset s'$. If $s \sqsupset s'$, then s is said to **contain** s'.

Definition 4.3. Given a transactional database DB and a sequence s, the **support** of s in DB, denoted as $sup_{DB}(s)$, is the number of transactions $T \in DB$ that contain s: $sup_{DB}(s) = |\{T\}|$ such that $T \sqsupset s$.

Definition 4.4. Given a transactional database DB and a threshold $min\_sup \in [1, |DB|]$, sequence s is called **frequent** in DB w.r.t. min_sup if $sup_{DB}(s) \geq min\_sup$. Similarly, s is called **infrequent** in DB w.r.t. a threshold max_sup if $sup_{DB}(s) \leq max\_sup$.

Example 4.1. Consider the D_{pos} dataset from Figure 4.1. $I = \{a, b, c, d, e\}$; transaction $T_1 = \langle$b,c,a,d,b,a,d,b$\rangle$ is a 8-sequence; $\langle$a,b$\rangle \sqsubset T_1$; $T_1 \sqsupset \langle$b,c$\rangle$; $sup_{D_{pos}}(\langle$a,b$\rangle) = 2$; $sup_{D_{pos}}(\langle$b,c$\rangle) = 1$. Given $min\_sup = 2$, $\langle$a,b$\rangle$ is frequent in D_{pos}, while $\langle$b,c$\rangle$ is infrequent in D_{pos}.

In order to complete our specification of the contrast patterns to be mined from execution traces, we need to consider another aspect - their *conciseness*. There is no doubt that for a developer, the easiest patterns to analyze are the shortest ones, i.e. those with the smallest number of items. If the contrast between anomalous and normal parts of a trace can be expressed with a 3-pattern, then adding more items to this pattern will make the task of analyzing the specific faulty system behavior harder. Therefore, we are interested in mining the set of *minimal contrast sequences*, i.e. such contrast sequences that do not contain shorter contrast sequences in them (see Definition 4.5 further on).

Finally, we need to define an interestingness measure to distinguish contrast patterns from the non-contrast ones. Contrast patterns are often defined as patterns whose supports differ significantly among datasets under contrast (D_{pos} and D_{neg}, in our case). There exist three common interestingness measures to express "supports differ significantly" for a given candidate pattern p [26]:

- growth rate α [27]: $\frac{sup_{D_{pos}}(p)}{sup_{D_{neg}}(p)} > \alpha$;
- support difference β [15]: $sup_{D_{pos}}(p) - sup_{D_{neg}}(p) > \beta$;
- two thresholds (min_sup and max_sup) [41]: $sup_{D_{pos}}(p) > min\_sup$, and $sup_{D_{neg}}(p) < max\_sup$.

Two thresholds allow to express the contrasting property of a candidate pattern in the most precise way, which is important in order to understand the performance of contrast pattern mining algorithms presented further in this chapter. Therefore, the final definition of a minimal contrast pattern can be written as the following:

Definition 4.5. Given D_{pos} and D_{neg} transactional databases, as well as $min\_sup \in [1, |D_{pos}|]$ and $max\_sup \in [1, |D_{neg}|]$ thresholds, a **minimal contrast sequence** is such a sequence s that $sup_{D_{pos}}(s) \geq min\_sup$, $sup_{D_{neg}}(s) \leq max\_sup$, and $\nexists s' \in MCS$ such that $s' \sqsubset s$, where $\boldsymbol{MCS}$ denotes the set of all minimal contrast sequences.

Example 4.2. Consider D_{pos} and D_{neg} datasets from Figure 4.1. Let $min\_sup = 2$ and $max\_sup = 0$. Then, $\langle a, b, d\rangle \in MCS$, as $sup_{D_{pos}}(\langle a, b, d\rangle) = 2$, $sup_{D_{neg}}(\langle a, b, d\rangle) = 0$, while $\langle a, b, a, d\rangle$ is a contrast sequence, but it is not minimal, as $\langle a, b, d\rangle \sqsubset \langle a, b, a, d\rangle$.

In this section, we showed how the task of discovering the differences between two sets of subtraces can be addressed by mining minimal contrast sequences from them. As explained, after the original execution trace is split into two sets of subtraces, the only operations necessary to prepare datasets for mining consist in dropping the timestamps of the tracepoints, and encoding the tracepoints themselves as integers, so that the same tracepoints are encoded as same integers. In the next section, we examine the methods to mine the complete set of MCS from D_{pos} and D_{neg} datasets.

4.2 Minimal Contrast Sequence Mining

This section plays a dual role. From the one hand, we introduce the state-of-the-art sequential pattern mining algorithms that can be applied to mine minimal contrast sequences. From the other hand, we evaluate the performance of these algorithms on execution trace data. It is important to note that we have implemented parallel versions of all the algorithms presented in this chapter in Java programming language and evaluated them on a server machine equipped with two Intel Xeon E5-2650 v2 processors (32 cores in total) and 128GB of RAM. For evaluation purposes, we have developed a synthetic trace generation tool. It allows to create datasets of specified complexity which preserve the essential properties of real-world execution traces coming from multimedia embedded systems.

Synthetic trace generation tool

As explained in Chapter 2, the runtime behavior of an embedded streaming application consists in periodic invocations of its dataflow actors. An actor's execution can be observed in a trace as a sequence of tracepoints (or, *events*) run by the actor's source code. If we encode every tracepoint as an integer and drop the timestamps, the original trace can be represented as a sequence of integers containing actors' executions as its subsequences. It is important to remember that during real-world use cases, the computational resources of an MPSoC are shared among various applications' and also system's threads running simultaneously; moreover, not all system tasks are invoked periodically. Therefore, any actor's execution can be preempted in favor of some other application's or system's activity at any moment of time. These facts can be observed in an execution trace when a given sequence of events appears repetitively in the trace (which represents a particular actor's execution), but is interleaved with some unrelated (or *noisy*) events. As an example, consider Figure 4.2 which shows two small excerpts from an execution trace of the GStreamer application (see Section 5.1 for more details on the application and the way its trace was obtained). Two executions of the *demux* actor are highlighted in bold. As we can see, both executions are preempted with some unrelated system activity (the second one is preempted more heavily).

As we have seen in Chapter 3, D_{pos} and D_{neg} datasets are obtained by splitting the trace into subtraces (or, *transactions*) using the event corresponding to the most upstream faulty dataflow actor. In theory, each actor of the application's dataflow graph can be scheduled with a different period. In practice, however, it is common for adjacent actors to be scheduled with the same period. Therefore, if we try to model a real-world execution trace of an embedded streaming application, each transaction in D_{pos} and D_{neg} can be represented as the same sequence of tracepoints mixed with random events.

Our synthetic trace generation tool incorporates the knowledge on real-world execution traces of embedded streaming applications described above. The tool allows to generate D_{pos} and D_{neg} datasets of various complexities by configuring the following set of parameters:

- *alphabetSize* – the number of distinct events appearing in the trace;
- *commonSequenceLength* – the number of events in the sequence common to all the transactions;
- *numTransactionsDpos* - the total number of transactions in the D_{pos} dataset;
- *numTransactionsDneg* - the total number of transactions in the D_{neg} dataset;
- *noiseRate* - a real number from the interval $[0, 1]$ representing the probability that a given event in the execution trace will be followed by some random noisy event;
- *mcsLength* - the length of the contrast sequence to be inserted into the D_{pos} dataset and to be removed from the D_{neg} dataset.

```
Timestamp;DebugCategory;DebugLevel;source_file;line;function
```

```
9468623319;LOG;qtdemux;qtdemux.c;3777;gst_qtdemux_combine_flows
9468633974;LOG;qtdemux;qtdemux.c;3807;gst_qtdemux_combine_flows
9468638945;DEBUG;GST_MEMORY;gstmemory.c;88;_gst_memory_free
9468645018;LOG;qtdemux;qtdemux.c;6233;qtdemux_parse_samples
9468651723;TRACE;GST_REFCOUNTING;gstobject.c;273;gst_object_unref
9468654964;DEBUG;qtdemux;qtdemux.c;6244;qtdemux_parse_samples
9468664013;LOG;qtdemux;qtdemux.c;6270;qtdemux_parse_samples
9468674850;LOG;qtdemux;qtdemux.c;6356;qtdemux_parse_samples
9468687881;DEBUG;qtdemux;qtdemux.c;6459;qtdemux_parse_samples
9468718535;LOG;qtdemux;qtdemux.c;4325;gst_qtdemux_loop
9468723949;LOG;GST_SCHEDULING;gstpad.c;3758;gst_pad_chain_data_unchecked
9468728766;LOG;qtdemux;qtdemux.c;3604;gst_qtdemux_prepare_current_sample
9468739686;LOG;qtdemux;qtdemux.c;6233;qtdemux_parse_samples
9468751020;LOG;qtdemux;qtdemux.c;6622;qtdemux_parse_samples
9468756514;DEBUG;basetransform;gstbasetransform.c;1563;default_prepare_output_buffer
9468762275;DEBUG;qtdemux;qtdemux.c;4214;gst_qtdemux_loop_state_movie
9468772093;LOG;qtdemux;qtdemux.c;4235;gst_qtdemux_loop_state_movie
```

```
9532237498;LOG;qtdemux;qtdemux.c;3777;gst_qtdemux_combine_flows
9532256379;LOG;adapter;gstadapter.c;341;gst_adapter_push
9532277147;LOG;qtdemux;qtdemux.c;3807;gst_qtdemux_combine_flows
9532294784;LOG;adapter;gstadapter.c;256;update_timestamps
9532316032;LOG;qtdemux;qtdemux.c;6233;qtdemux_parse_samples
9532339010;DEBUG;qtdemux;qtdemux.c;6244;qtdemux_parse_samples
9532344612;LOG;adapter;gstadapter.c;262;update_timestamps
9532353386;LOG;qtdemux;qtdemux.c;6270;qtdemux_parse_samples
9532387517;DEBUG;audiodecoder;gstaudiodecoder.c;1332;gst_audio_decoder_push_buffers
9532417173;LOG;qtdemux;qtdemux.c;6356;qtdemux_parse_samples
9532443726;DEBUG;audiodecoder;gstaudiodecoder.c;1349;gst_audio_decoder_push_buffers
9532460651;DEBUG;qtdemux;qtdemux.c;6459;qtdemux_parse_samples
9532473863;LOG;audiodecoder;gstaudiodecoder.c;1371;gst_audio_decoder_push_buffers
9532501299;LOG;adapter;gstadapter.c;835;gst_adapter_take_buffer
9532519113;TRACE;GST_REFCOUNTING;gstobject.c;273;gst_object_unref
9532543000;TRACE;GST_REFCOUNTING;gstobject.c;247;gst_object_ref
9532547375;LOG;adapter;gstadapter.c;850;gst_adapter_take_buffer
9532562718;LOG;qtdemux;qtdemux.c;4325;gst_qtdemux_loop
9532578395;TRACE;GST_REFCOUNTING;gstminiobject.c;360;gst_mini_object_ref
9532596995;LOG;qtdemux;qtdemux.c;3604;gst_qtdemux_prepare_current_sample
9532611281;LOG;adapter;gstadapter.c;561;gst_adapter_flush_unchecked
9532624635;LOG;qtdemux;qtdemux.c;6233;qtdemux_parse_samples
9532641737;LOG;adapter;gstadapter.c;581;gst_adapter_flush_unchecked
9532657344;LOG;qtdemux;qtdemux.c;6622;qtdemux_parse_samples
9532671155;TRACE;GST_REFCOUNTING;gstminiobject.c;440;gst_mini_object_unref
9532689358;DEBUG;qtdemux;qtdemux.c;4214;gst_qtdemux_loop_state_movie
9532703116;LOG;adapter;gstadapter.c;590;gst_adapter_flush_unchecked
9532719228;LOG;qtdemux;qtdemux.c;4235;gst_qtdemux_loop_state_movie
```

FIGURE 4.2: Two excerpts from an execution trace showing two executions of the *demux* dataflow actor (corresponding events are in bold) preempted with unrelated events (in normal font).

At first, the tool randomly generates a sequence containing *commonSequenceLength* events represented as integers from the interval $[1, alphabetSize]$. This sequence is replicated *numTransactionsDpos* and *numTransactionsDneg* times to create the basis of D_{pos}

and D_{neg} transactional databases respectively. The tool then randomly generates a contrast sequence of length *mcsLength* and inserts this sequence into all the transactions from the D_{pos} dataset. The position of insertion is chosen randomly, but remains the same for all the transactions in D_{pos}. The noisy events are injected into D_{pos} and D_{neg} datasets in the following way. The first event of a transaction is considered. A noisy event is inserted after the first event with probability equal to *noiseRate*. The second event of the transaction is then considered. It can be either the second event from the original transaction, or the noisy event injected after the first event. Again, a noisy event is injected after the second event with probability *noiseRate*, and so on. This way, a given pair of adjacent events from the sequence shared by all the transactions can be separated by $S = \sum_{i=1}^{\infty} noiseRate^i$ noisy events after this step. Observe that S is a sum of an infinite geometric series where $noiseRate < 0$, thus, $S = \frac{1}{1-noiseRate}$. The average transaction length after injecting the noise is equal to $commonSequenceLength \times S$, or

$$Average\ transaction\ length = \frac{commonSequenceLength}{1 - noiseRate}. \tag{4.1}$$

Finally, our synthetic trace generation tool removes the occurrences of the contrast sequence from the transactions in the D_{neg} dataset. Figure 4.3 provides an example of D_{pos} and D_{neg} datasets generated by the tool given the following parameters: *alphabetSize*=25, *commonSequenceLength*=20, *numTransactions*=4, *noiseRate*=0.3, *mcsLength*=3.

contrast sequence: **8 1 12**
common sequence: **7 10 4 6 15 19 14 8 14 19 8 8 10 11 6 18 18 17 20 9**

D_{pos} dataset:

transaction #1: **7 10 4 6** 1 1 1 16 **15** 6 15 14 **19 14 8** **8 1** 4 8 **12** 6 **14** 20 **19 8** 1 **8 10 11** 8 15 **6 18 18** 7 **17 20 9**
transaction #2: **7 10 4 6 15** 4 1 **19 14** 11 9 **8** 3 6 **8 1** 19 13 16 **12** 16 **14** 20 6 **19** 5 **8 8** 7 **10 11** 11 10 **6 18** 14 11 6 **18 17** 13 **20** 9
transaction #3: **7 10 4 6 15 19 14 8** **8** 2 **1 12 14 19** 16 **8 8 10 11 6** 9 **18 18** 12 **17 20** 10 **9**
transaction #4: **7 10** 12 **4 6 15 19 14 8** 8 **8** 20 17 **1** 17 9 11 1 19 10 **12** 12 **14** 7 1 **19** 3 **8 8 10** 17 4 20 **11 6** 8 **18 18 17 20 9**

D_{neg} dataset:

transaction #1: **7** 13 **10 4 6 15 19 14** 14 **8 14** 11 **19** 18 1 20 20 **8** 1 6 15 3 4 **8 10 11** 20 **6 18** 6 **18 17** 19 **20** 8 5 **9**
transaction #2: **7** 18 7 **10 4 6** 19 **15 19 14** 4 15 10 15 **8 14 19 8 8** 8 9 20 **10 11** 9 15 10 **6 18** 10 **18 17 20 9**
transaction #3: **7 10 4 6** 11 **15 19** 19 **14 8** 9 14 **14 19 8** 11 **8 10 11 6** 12 **18 18** 8 9 13 18 7 **17** 18 **20** 5 2 11 13 **9**
transaction #4: **7 10 4 6 15** 7 **19 14 8** 11 5 **14** 2 11 **19** 4 **8 8** 7 **10 11 6 18** 15 **18** 20 **17 20 9**

FIGURE 4.3: Example output of the trace generation tool

In the rest of this section, we consider three ways of mining minimal contrast sequences from execution traces. The first two are able to return the complete set of *MCS*, either

by mining the target patterns directly from the datasets, or extracting them from other types of patterns. The third one allows to reduce the complexity of the task by mining an incomplete set of minimal contrast sequences, by allowing the user to control either the maximum number of events allowed to occur in the transactions between the elements of a contrast sequence, or the maximum length of minimal contrast sequences.

4.2.1 Direct Mining of Minimal Contrast Sequences

A direct approach to mine the complete *MCS* set consists in considering *all* sequential patterns from D_{pos} and D_{neg} datasets, verifying their supports separately in D_{pos} and D_{neg}, and returning only such sequences s that $sup_{D_{pos}}(s) \geq min\_sup$ and $sup_{D_{neg}}(s) \leq max\_sup$.

Unfortunately, this approach is not computationally feasible even for unrealistically small execution traces. The problem comes, firstly, from the combinatorial number of candidate sequential patterns in a single transaction [2]; and secondly, from the fact that the main technique in pattern mining allowing to significantly prune the search space of candidate patterns known as the Apriori principle [8] (see Definition 4.6) can not be used in contrast pattern mining [27]. Moreover, techniques for efficient direct mining of contrast itemsets can not be applied for mining contrast sequences [20]. Up until today, there has not been proposed any algorithm for an efficient direct mining of contrast sequential patterns (cf. contrast string mining from Section 4.2.3).

4.2.2 Indirect Mining of Minimal Contrast Sequences

By definition, the set of contrast sequences lies in the intersection of the sets of frequent sequences in D_{pos} and infrequent sequences in D_{neg} (see Section 4.1). The set of minimal contrast sequences is a proper subset of the set of contrast sequences (see Figure 4.4). Therefore, the set of minimal contrast sequences can be mined in the following way:

1. Mine all frequent sequences in D_{pos}.
2. Filter out sequences frequent in D_{neg} in order to get the set of contrast sequences.
3. Extract minimal contrast sequences from the set of contrast sequences.

Frequent sequence mining is an active research area that has produced numerous algorithms and approaches to efficiently navigate the search space of sequential patterns. As can be seen from Figure 4.4, it is also possible, in theory, to extract the *MCS* set from the set of infrequent in D_{neg} sequences. However, the task of mining infrequent sequential patterns has not been addressed in the literature due to potentially huge number of sequences not present in a given transactional database. Therefore, in this section, we

[2] There are 2^n subsequences in a sequence of length n

focus on the extraction of minimal contrast sequences from the set of frequent sequential patterns in D_{pos}.

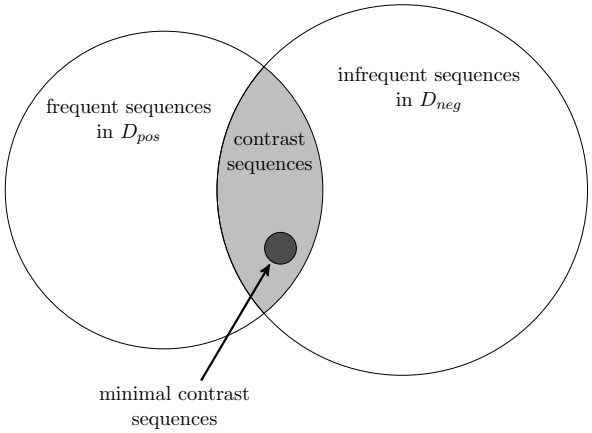


FIGURE 4.4: A Venn diagram showing the relations between the sets of sequential patterns.

Frequent sequence mining

Frequent sequential patterns proved to be extremely useful nuggets of information for the analysis of various types of data: weblogs, biological sequences, purchase and sales histories, network traffic, etc. Such patterns, due to their utility, have been studied extensively in the data mining community during the last two decades. Two main strategies to mine frequent sequential patterns from a transactional database are presented below.

- *Candidate generate-and-test* approach (GSP [7] and SPADE [88] algorithms): generate candidate [3] sequential patterns of size $l + 1$ from the frequent sequential patterns of size l and test their support to prune the infrequent ones.
- *Pattern-growth* approach (the PrefixSpan [63] algorithm): having a frequent sequential l-pattern s, all the frequent sequences having s as *prefix* (see Definition 4.7) are explored before the other frequent sequential l-patterns are considered.

Independently of the way the search space is explored, all sequential pattern mining algorithms use the Apriori property in order to avoid generating a combinatorial number of candidate patterns [35]:

Definition 4.6. Apriori property. Every nonempty subsequence of a frequent sequential pattern is a frequent sequential pattern.

This way, if a candidate pattern does not pass the frequency test, all of its supersequences will likewise fail the frequency test and, thus, can be pruned from the search space.

[3] a *candidate* pattern is a pattern from the search space considered by the mining algorithm

Example 4.3. Consider D_{pos} dataset from Figure 4.1. Let $min\_sup = 2$. 1-sequence $\langle e \rangle$ is infrequent in D_{pos} as $sup_{D_{pos}}(\langle e \rangle) = 1$. Therefore, none of its supersequences, e.g. $\langle a, b, e \rangle \sqsupset \langle e \rangle$ or $\langle e, c \rangle \sqsupset \langle e \rangle$ can be frequent in D_{pos}.

In practice, pattern-growth approach proved to be more space- and computationally efficient than candidate generate-and-test one [63]. Therefore, in this thesis, we consider the state-of-the-art pattern-growth algorithm called PrefixSpan [63] to mine the complete set of frequent sequential patterns. We firstly introduce a set of definitions necessary to understand the workings of the PrefixSpan algorithm.

Definition 4.7. Given a sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$, a sequence $s' = \langle e'_1, e'_2, \ldots, e'_m \rangle$, $(m \leq n)$ is called a **prefix** of s, denoted as $s' \sqsubset_p s$, if $e'_i = e_i$ for $i \leq m$.

Definition 4.8. Given a sequence s and 1-sequence e_1, $e_1 \sqsubset s$, the subsequence from the beginning of s to the first appearance of item e_1 in s is called the **first instance** of 1-sequence e_1 in s. Recursively, we can define the first instance of a $(i+1)$-sequence $\langle e_1, e_2, \ldots, e_i, e_{i+1} \rangle$ from the first instance of the i-sequence $\langle e_1, e_2, \ldots, e_i \rangle$ (where $i \geq 1$) as the subsequence from the beginning of s to the first appearance of item e_{i+1} which also occurs after the first instance of the i-sequence $\langle e_1, e_2, \ldots, e_i \rangle$.

Note that the first instance of s' in s is always a prefix of s.

Definition 4.9. Given sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$. Let $p = \langle e_1, e_2, \ldots, e_m \rangle$ $(m < n)$ be a prefix of s: $p \sqsubset_p s$. Sequence $q = \langle e_{m+1}, e_{m+2}, \ldots, e_n \rangle$ is called the **suffix** of s with regard to prefix p, denoted as $q \sqsubset_s s$.

Definition 4.10. Let s be a sequential pattern in the transactional database DB. The **s-projected database**, denoted $DB|_s$, is a collection of suffixes of transactions in DB with regard to the first instances of s.

Example 4.4. Consider D_{pos} dataset from Figure 4.1. ⟨c,a,b,a,d⟩ $\sqsubset_p T_2$. The first instance of ⟨a,b⟩ in T_1 is ⟨b,c,a,d,b⟩. ⟨e,b,d⟩ $\sqsubset_s T_2$. Finally, $D_{pos}|_{\langle a,b,d \rangle} = \{\langle b \rangle, \langle e,b,d \rangle\}$.

The problem of mining frequent sequential patterns can be decomposed into a set of subproblems:

1. Let $\{s_1, s_2, \ldots, s_n\}$ be the complete set of frequent 1-sequences in a transactional database DB. The complete set of frequent sequences in DB can be divided into n disjoint subsets. The ith subset $(1 \leq i \leq n)$ is the set of frequent sequences with prefix s_i.

2. Let α be a frequent l-sequence and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all frequent $(l+1)$-sequences with prefix α. The complete set of frequent sequences with prefix α, except for α itself, can be divided into m disjoint subsets. The jth subset $(1 \leq j \leq m)$ is the set of frequent sequences with prefix β_j.

Based on this observation, PrefixSpan considers sequential pattern mining as a divide-and-conquer problem (Algorithm 2). Given a frequent 1-sequence s_1, the algorithm proceeds to discover all the frequent sequences having s_1 as a prefix. Once all such frequent sequences are mined, the next frequent 1-sequence s_2 is considered, and so on. Similarly, upon generating a frequent l-sequence α, PrefixSpan continues to mine all frequent sequences having α as prefix before considering other frequent l-sequences sharing the same prefix of length $(l-1)$ with α.

Algorithm `PrefixSpan`(min_sup,SDB)
  input : min_sup: minimum support threshold, SDB: transactional database.
  output: The complete set of frequent sequential patterns
  `__PrefixSpan`($\langle\rangle$, SDB)
Subroutine `__PrefixSpan`(α,$S|_\alpha$)
  input : α: sequential pattern, $S|_\alpha$: the α-projected database.
  $\mathcal{B} \leftarrow$ frequent w.r.t min_sup length-1 sequences in $S|_\alpha$
  foreach $b \in \mathcal{B}$ **do**
    $\alpha' \leftarrow$ append b after the last element of α
    output α'
    $S|_{\alpha'} \leftarrow$ projected database for α'
    `__PrefixSpan`(α',$S|_{\alpha'}$)
  end

Algorithm 2: PrefixSpan algorithm

Assume that there exists a lexicographical ordering $\leq$ among the items in the dictionary. Conceptually, PrefixSpan algorithm considers the search space of sequential patterns as a **lexicographic tree** which can be constructed in the following way. The root node of the tree is labeled with $\emptyset$. Recursively, we can extend a node at level l in the tree (denoting an l-sequence) by adding one item after its last element. This way, we get a child node at the next level $l+1$. The children of a node are generated and arranged according to the chosen lexicographical ordering. PrefixSpan traverses the lexicographical tree in a **depth-first** manner and prunes the entire subtrees rooted at infrequent sequential patterns (cf. Apriori property from Definition 4.6), constructing this way a lexicographic frequent sequence tree. Figure 4.5 shows a lexicographic frequent tree for the D_{pos} dataset from Figure 4.1 given $min\_sup = 2$.

Once the complete set of frequent sequences in the D_{pos} transactional database is mined, we need to filter out such sequences which are also frequent in the D_{neg} transactional database, in order to obtain the set of contrast sequences. This can be done by taking each sequence s, such that $sup_{D_{pos}}(s) \geq min\_sup$, and marking it as a contrast one only if $sup_{D_{neg}}(s) \leq min\_sup$. In Figure 4.5, contrast sequences are circled. Notice that all the descendants of a contrast sequence in the lexicographic sequence tree are its supersequences, hence, are not minimal. It is possible to avoid generation of such non-minimal contrast sequences if we enrich the PrefixSpan algorithm with the contrast pruning [4].

[4] contrast pruning was firstly introduced by Ji et al. [41] and adopted in the ConSGapMiner algorithm which we will present in Section 4.2.3

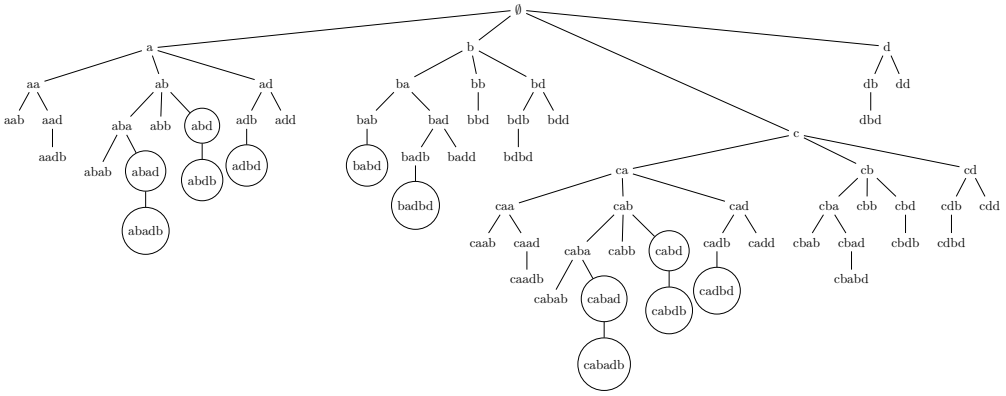


FIGURE 4.5: A frequent sequence tree for the D_{pos} dataset from Figure 4.1 given $min\_sup = 2$. Contrast sequences (with respect to the D_{neg} dataset) are circled.

Definition 4.11. Contrast pruning. For each sequence $s = \langle a_1, a_2, \ldots, a_n \rangle$ such that $sup_{D_{pos}}(s) \geq min\_sup$, verify its support in D_{neg}. If $sup_{D_{neg}}(s) \leq min\_sup$, then output s as a contrast sequence, and stop growing it (i.e. prune all its descendants in the lexicographic sequence tree).

This way, the whole subtrees rooted at contrast sequences are pruned reducing the number of candidate sequences. Algorithm 3 shows how contrast pruning can be easily integrated into the PrefixSpan algorithm, making it essentially a contrast sequence mining algorithm.

Algorithm `ContrastPrefixSpan`(min_sup,max_sup,D_{pos},D_{neg})
  input : min_sup: min support threshold for the D_{pos} dataset, `// global constant`
    max_sup: max support threshold for the D_{neg} dataset, `// global constant`
    D_{pos}: database of anomalous subtraces,
    D_{neg}: database of normal subtraces `// global constant`
  output: The complete set of contrast sequential patterns
  `__ContrastPrefixSpan(`$\langle\rangle$, D_{pos}`)`
Subroutine `__ContrastPrefixSpan(`α,$D_{pos}|_\alpha$`)`
  input : α: sequential pattern, $D_{pos}|_\alpha$: the α-projected D_{pos} database.
  $\mathcal{B} \leftarrow$ frequent w.r.t min_sup length-1 sequences in $D_{pos}|_\alpha$
  foreach $b \in \mathcal{B}$ **do**
    $\alpha' \leftarrow$ append b after the last element of α
    if α' *is infrequent w.r.t.* max_sup *in* D_{neg} **then**
      output α'
      return
    end
    else
      $D_{pos}|_{\alpha'} \leftarrow$ projected D_{pos} database for α'
      `__ContrastPrefixSpan(`α',$D_{pos}|_{\alpha'}$`)`
    end
  end

Algorithm 3: ContrastPrefixSpan algorithm

The set of contrast sequences returned by the ContrastPrefixSpan algorithm, however, is still not minimal. Consider Figure 4.5. Due to the depth-first tree traversal, there is no way to know that, for example, ⟨a,b,a,d⟩ will be a supersequence of a contrast sequence ⟨a,b,d⟩, as ⟨a,b,a⟩-rooted subtree is traversed before the ⟨a,b,d⟩-rooted one due to the chosen lexicographic order of items. Extraction of minimal contrast sequences from the non-minimal ones can be done by sorting all contrast sequences by length and checking if each of them contains any contrast sequences of smaller size.

We next show the experimental results of running the PrefixSpan algorithm on synthetic trace data (see Figure 4.6). We test both the original version of PrefixSpan as well as the one enriched with contrast pruning, which we refer to as ContrastPrefixSpan. In the first case, the reported computational complexity applies to the mining of all frequent sequential patterns in the D_{pos} dataset. In the second case, it applies to the mining of the set of (non-minimal) contrast sequences between D_{pos} and D_{neg} datasets. The experiments were conducted in the following way. We fixed the values of all the parameters of the synthetic trace generator except *commonSequenceLength* parameter, whose values are shown on the bottom X-axis (the upper X-axis shows the corresponding average transaction lengths computed using Equation 4.1). For each value of *commonSequenceLength*, we generated 9 traces with our synthetic trace generation tool, ran PrefixSpan and ContrastPrefixSpan on them, and reported the number of candidate sequential patterns as well as the running time with box-and-whisker plots where the ends of the whiskers show minimum and maximal values among 9 traces, boxes represent the interquartile range and the lines connect median values. This approach of generating 9 synthetic traces for a given set of parameters' values and reporting the results with a box-and-whisker plot was applied to all the experiments presented further in this chapter. Moreover, all the algorithms tested in this chapter were run with $min\_sup = 100\%$ for the D_{pos} dataset and $max\_sup = 0\%$ for the D_{neg} dataset. Notice that such combination of values represents the most restrictive case of sequential pattern mining among all other pairs of values assigned to min_sup and max_sup and, hence, result in the fewest number of candidate patterns.

Analyzing the results presented in Figure 4.6, the first observation we can make is that the running time of PrefixSpan is well correlated to the number of candidate patterns tested by the algorithm. Indeed, the running time of the algorithms that mine sequential patterns from a lexicographic tree grows proportionally to the number of candidate patterns. The second observation is that the PrefixSpan algorithm fails to efficiently mine the complete *MCS* set even from small trace datasets. Even though contrast pruning allows to eliminate a significant number of candidate patterns, the computational complexity grows exponentially with the average transaction length. This implies that the total size of search space needed to be explored by the algorithm is still too large to allow an efficient mining of contrast sequences from industrial trace data.

Closed frequent sequence mining

Mining and analysis of the complete set of frequent patterns are both daunting tasks. There are two aspects to this problem: a big number of mined frequent patterns makes

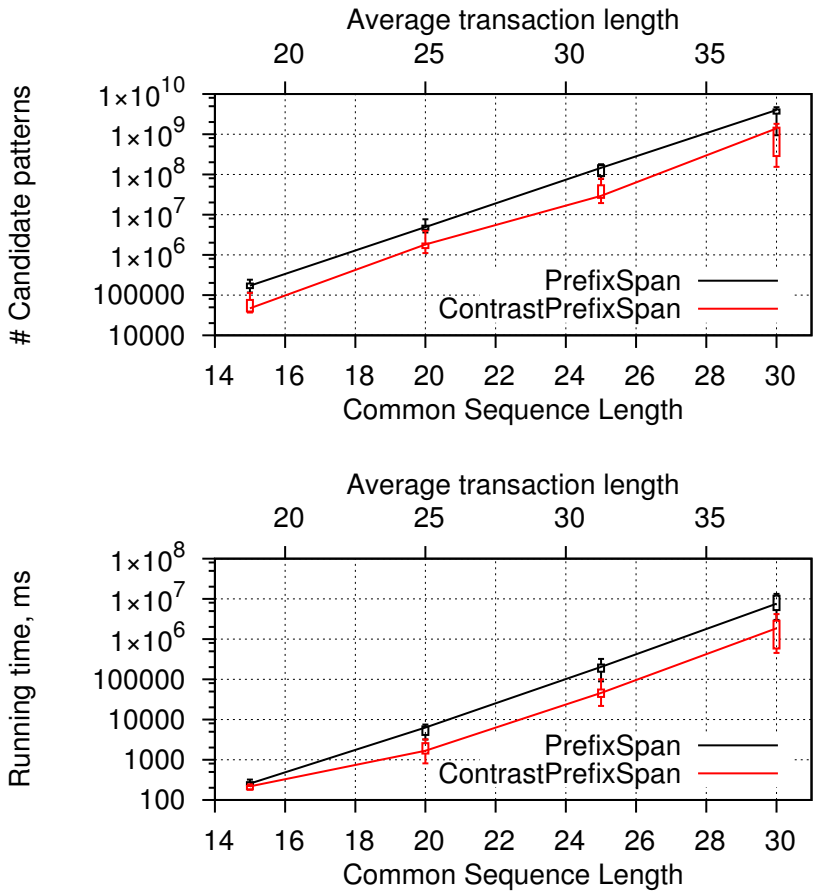


FIGURE 4.6: Computational complexity of the PrefixSpan algorithm on synthetic trace data generated with the following parameters: $alphabetSize = 20$, $numTransactionsDpos = 10$, $numTransactionsDneg = 30$, $noiseRate = 0.2$, $mcsLength = 3$.

their analysis hard, while a high time complexity of the mining task does not allow to discover frequent patterns from big datasets. Both these aspects are addressed by a class of algorithms which reduce the complexity of frequent pattern mining by discovering a condensed and lossless representation of frequent patterns called closed frequent patterns [62].

Definition 4.12. Given a transactional database DB, a sequence s is **closed** if there exists no supersequence s' (i.e. $s \sqsubset s'$) having the same support in DB (i.e. $sup_{DB}(s) = sup_{DB}(s')$).

Example 4.5. Consider the D_{pos} dataset from Figure 4.1. ⟨b,a,b⟩ is not a closed sequence, as ⟨b,a,d,b,d⟩ $\sqsupset$ ⟨b,a,b⟩ and $sup_{D_{pos}}(\langle b,a,d,b,d\rangle) = sup_{D_{pos}}(\langle b,a,b\rangle) = 2$. At the same time, ⟨b,a,d,b,d⟩ is a closed sequence, because $\nexists s \sqsupset \langle b,a,d,b,d\rangle$, such that $sup_{D_{pos}}(s) = sup_{D_{pos}}(\langle b,a,d,b,d\rangle) = 2$.

There exist two state-of-the-art closed frequent sequence mining algorithms: CloSpan proposed by Yan et al. [85] and BIDE proposed by Wang et al. [80]. We chose the BIDE algorithm for the task of mining closed frequent sequences from the D_{pos} dataset, as it has smaller space and runtime complexity than the CloSpan algorithm [80].

The BIDE algorithm can be viewed as an extension of the PrefixSpan algorithm presented earlier in this chapter. It uses the same depth-first traversal of the lexicographical tree in order to navigate the search space of sequential patterns. The name of the algorithm is derived from the adopted BI-DirEctional closure checking scheme which allows to determine if a candidate sequential pattern is closed without consulting the set of already mined closed sequential patterns. This allows to reduce the space complexity of the frequent sequence mining task, as the algorithm does not require to keep the already mined closed sequences in main memory. At the same time, the BackScan pruning scheme adopted in the BIDE algorithm complements the Apriori-based pruning of the search space and permits to further reduce the number of candidate patterns, therefore, to decrease the computational complexity of the mining task. We next introduce the BackScan pruning method, as knowing its operation is necessary to understand the performance results of running BIDE on execution trace data discussed in Section 4.3 [5]. BackScan is based on the concepts of the first instance of a sequence with respect to its supersequence (see Definition 4.8), i-th last-in-first appearance of an element in a given sequence, and i-th semi-maximum period of a sequence s with respect to its supersequence.

Definition 4.13. For a sequence t containing an n-sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$, the ***i*-th last-in-first appearance** with respect to s in t is denoted as LF_i and defined recursively as:

1. if $i = n$, it is the last appearance of e_i in the first instance of s in t;
2. if $1 \leq i < n$, it is the last appearance of e_i in the first instance of s in t while LF_i must appear before LF_{i+1}.

Definition 4.14. For a sequence t containing an n-sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$, the ***i*-th semi-maximum period** of s in t is defined as:

1. if $1 < i \leq n$, it is the piece of t between the end of the first instance of $\langle e_1, e_2, \ldots, e_{i-1} \rangle$ in t and the LF_i of s in t.
2. if $i = 1$, it is the piece of t locating before the 1st last-in-first appearance with respect to s in t.

Example 4.6. Consider transaction T_1 in the D_{pos} dataset from Figure 4.1. T_1's first element $e_1 = b$, second element $e_2 = c$, etc. Given $s = \langle$a,b,d$\rangle$, $s \sqsubset T_1$, LF_3 with respect to s in T_1 is e_7, LF_2 is e_5, LF_1 is e_3 (and not e_6). 1^{st} semi-maximum period of s in T_1 is $\langle$b,c$\rangle$, 2^{nd} is $\langle$d$\rangle$, and 3^{rd} is $\langle$a$\rangle$. If we consider transaction T_2, $s \sqsubset T_2$, then 1^{st} semi-maximum period of s in T_2 is $\langle$c$\rangle$, 2^{nd} is $\langle\rangle$, and 3^{rd} is $\langle$a$\rangle$.

The BackScan pruning method can be informally described as follows. Given a sequence $s = \langle e_1, e_2, \ldots, e_n \rangle$, we firstly identify all the first instances of s in the transactions t

[5] The interested reader can consult the original work by Wang et al [80] to get a detailed explanation of the bi-directional closure scheme which we will not present in this thesis.

which contain s. Then, within each first instance of s, all n semi-maximum periods are computed. We check if there exist an integer i ($1 \leq i \leq n$) and an item e' which appears in each of the i-th semi-maximum periods of s in transactions from DB. If such an item exists, then s along with all its supersequences q having s as prefix ($s \sqsubset_p q$) can be safely pruned. This idea stems from the fact that all such q along with s can not be closed, as for $s' = \langle e_1, e_2, \ldots, e_{i-1}, e', e_i, \ldots e_n \rangle$, the following is true: $sup_{DB}(s') = sup_{DB}(s)$ and $s \sqsubset s'$.

Example 4.7. Consider the D_{pos} dataset from Figure 4.1 and its frequent sequence tree from Figure 4.5. 1^{st} semi-maximum period of 1-sequence ⟨a⟩ is ⟨b,c⟩ in T_1 and ⟨c⟩ in T_2. Therefore, the whole subtree rooted in ⟨a⟩ is pruned with BackScan, as any frequent sequence with prefix ⟨a⟩ will be discovered from the node ⟨c,a⟩ in the frequent sequence tree. Indeed, as can be seen in Figure 4.5, ⟨a⟩'s subtree is embedded as a child of ⟨c⟩'s subtree rooted in ⟨c,a⟩.

To sum up, BIDE explores the lexicographic tree of sequential patterns in the depth-first manner and performs both the bi-directional closure check as well as the BackScan pruning check on each candidate sequential pattern in order to determine if the candidate pattern is closed, and in case it is not – if its whole subtree can be pruned.

Extraction of minimal contrast sequences from the set of closed frequent in D_{pos} sequences ($CFS_{D_{pos}}$) is less intuitive than from the set of all frequent sequences presented in the previous section. We analyzed BackScan and contrast prunings and came to the conclusion that a closed contrast sequence cannot be pruned on the premise that all its descendants in the lexicographic sequence tree are non-minimal contrast sequences. In order to see why contrast pruning cannot be applied if closed sequential patterns are mined, consider the following example. $D_{pos} = \{\langle$d,a,b,c$\rangle, \langle$a,b,c,d$\rangle\}$, $D_{neg} = \{\langle$b,a,d,c$\rangle, \langle$d,c,b,a$\rangle\}$, and $min\_sup = 1$, $max\_sup = 0$. The set of MCS contains two sequences ⟨a,b⟩ and ⟨c,d⟩. The set of $CFS_{D_{pos}}$ is {⟨a,b,c⟩, ⟨d⟩, ⟨a,b,c,d⟩, ⟨d,a,b,c⟩}. Upon generating the candidate sequence ⟨a,b,c⟩, BIDE will detect that it is closed with $sup_{D_{pos}}(\langle$a,b,c$\rangle) = 2$ and will grow it further to find that ⟨a,b,c,d⟩ is also closed but with $sup_{D_{pos}}(\langle$a,b,c,d$\rangle) = 1$. If we add the contrast pruning, then ⟨a,b,c⟩'s subtree will be pruned as ⟨a,b,c⟩ is contrasting and, hence, all supersequences of it will not be minimal contrast sequences. Therefore, ⟨a,b,c,d⟩ will not be generated, and ⟨c,d⟩ $\in MCS$ is not contained in any output sequence, hence, is lost. Therefore, the MCS set has to be extracted from the $CFS_{D_{pos}}$ set as a post-processing step. It is a non-trivial task, as one needs to test subsequences of each closed frequent sequence. In case $CFS_{D_{pos}}$ contains a significant number of sequences, this task is very computationally costly because of the combinatorial number of subsequences in a single sequence. Another option would be to output the set of closed contrast sequences instead of the set of minimal contrast sequences. This will require simply checking the support of every $s \in CFS_{D_{pos}}$ to keep only the contrast ones. The downside of this option is, of course, the increased amount of manual analysis required from the developers to perform on each output sequence.

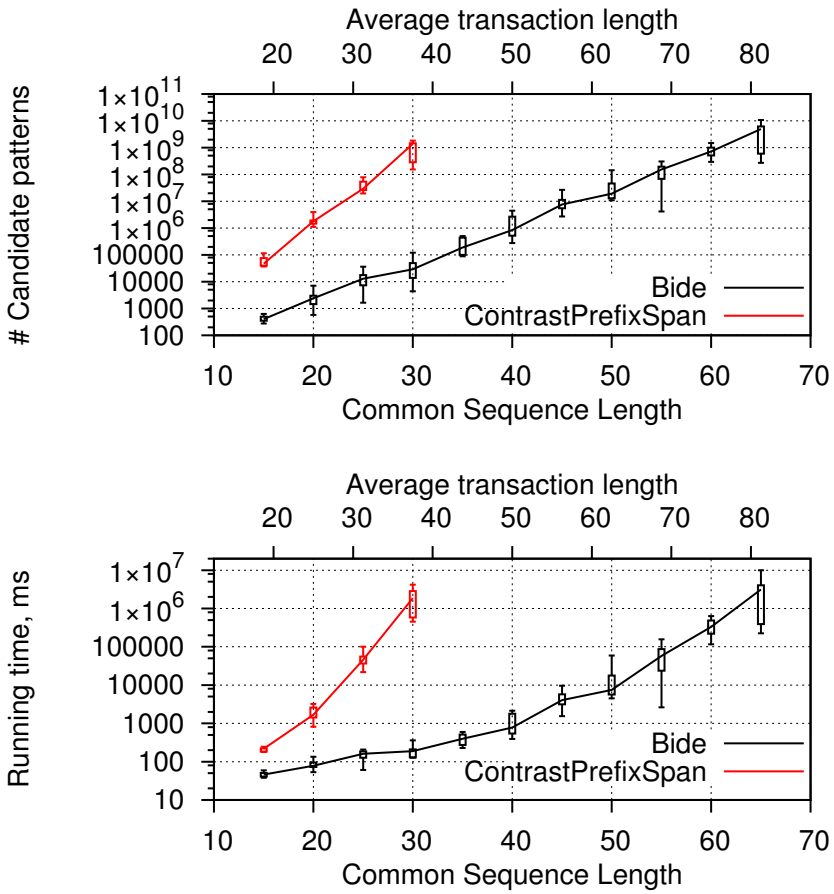


FIGURE 4.7: Comparison of the computational complexities of the Bide algorithm and of the ContrastPrefixSpan algorithm run on the same synthetic traces described in Figure 4.6

Figure 4.7 shows computational complexity of running the Bide algorithm on D_{pos} datasets which we used for the experiments on the PrefixSpan algorithm (see Figure 4.6). We also plotted the performance results for the ContrastPrefixSpan algorithm in order to facilitate the comparison of Bide's and PrefixSpan's performances on synthetic trace data. As we can see, Bide prunes much more candidate patterns than ContrastPrefixSpan, and is able to mine traces twice as big as those mined by ContrastPrefixSpan in less time. Nevertheless, it is important to note that patterns mined by the Bide algorithm are closed frequent sequences in the D_{pos} dataset, hence, minimal contrast sequences must be extracted from them in a post-processing step, which can be quite computationally expensive. The main conclusion we can make from Figure 4.7, however, is that even with an aggressive pruning performed by the BackScan technique, Bide will not be able to efficiently mine the full set of closed frequent sequential patterns from industrial trace data even with $min\_sup = 100\%$. Therefore, the investment one puts into developing an efficient approach to extract the set of *MCS* from closed frequent sequences will not be justified for the final goal of an efficient *MCS* set mining from trace data.

In addition to closed frequent patterns, there exists another condensed representation of frequent patterns called **maximal frequent patterns**. A pattern p is maximal, if there does not exist any frequent pattern q, such that q contains p. Despite the existence of efficient algorithms to mine the set of maximal frequent itemsets (e.g. state-of-the-art MaxMiner algorithm [16]), the problem of mining maximal frequent sequences received very little attention in the literature due to its high complexity. Luo and Chung [52] proposed to mine maximal sequences by sampling, therefore, making the output set incomplete. Fournier-Viger et al. [31] proposed to use a slightly changed BackScan pruning from the BIDE algorithm in order to mine maximal frequent sequences. The change consists in pruning a candidate sequence s if there exists an item that appears in at least min_sup number of i-th semi-maximum periods of s rather than in all i-th semi-maximum periods. Notice that when $min\_sup = |D_{pos}|$, as used in the experiments above, the set of maximal frequent sequences is equal to the set of closed frequent sequences, hence, such change in the BackScan pruning technique does not result in any performance improvement compared to the BIDE algorithm.

Finally, **sequential generator patterns** (or, sequence generators) can be considered as "reverse"-closed sequences. A sequence s is a generator if there does not exist any subsequence of s that has the same support as s [32] [86]. Minimal contrast sequences, thus, can be directly identified during sequence generators mining without a post-processing stage. Unfortunately, the pruning technique adopted in sequential generator pattern mining is too restrictive for efficient mining of such patterns, as it allows to prune a candidate sequence only if there exists a subsequence with exactly the same projected database.

4.2.3 Mining Minimal Contrast Sequences with Constraints

As we have seen in Section 4.2.2, the state-of-the-art sequence mining algorithms do not succeed in tackling the complexity of mining minimal contrast sequences from execution trace data. In this section, we consider adding constraints to the final *MCS* set in order to reduce the complexity of the task.

Firstly, we use the *maximum gap* constraint which defines the maximal number of items that may appear between the elements of a sequential pattern in a transaction (see Definition 4.15). We start with the most restrictive situation when the maximum gap is equal to 0, i.e. when the elements of a sequential pattern must appear contiguously within the transactions. In such case, sequential patterns are called *strings*, and we find out how contrast strings can be mined. Then, we leave the choice of the maximum gap constraint to the user. Secondly, we use the *maximum length* constraint, which determines the upper limit on the number of elements in a sequential pattern, and discover the influence of this constraint on the complexity of mining minimal contrast sequences from execution traces.

Definition 4.15. The **maximum gap constraint** is specified by a positive integer max_gap. Given two sequences $s' = \langle a_1, a_2, \ldots, a_n \rangle$ and $s = \langle b_1, b_2, \ldots, b_m \rangle$, $n < m$, if

there exist integers $1 \leq i_1 < i_2 < \ldots i_n \leq m$, such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_n = b_{i_n}$ and $i_{k+1} - i_k \leq max\_gap$, $\forall k \in [1, n-1]$, then s' is a subsequence of s with respect to the maximum gap constraint max_gap.

Example 4.8. Consider the transaction $T_1 = \langle$b,c,a,d,b,a,d,b$\rangle$ from the D_{pos} dataset in Figure 4.1. Having $max\_gap = 2$, $\langle$c,b,a$\rangle \sqsubset T_1$ and $\langle$b,b,a$\rangle \not\sqsubset T_1$ with respect to the gap constraint.

Mining minimal contrast strings

The problem of mining *contrast strings* from a set of sequential databases was introduced by Chan et al. [20]. Their approach is similar to the indirect mining of minimal contrast sequences introduced in Section 4.2.2, but uses a suffix tree instead of the prefix tree to navigate the search space. The suffix tree structure is known to allow fast extraction of strings from sequential databases at the cost of the large amount of space required to store it. Indeed, the algorithm proposed by Chan et al. needs to construct a suffix tree which represents all the existing strings in the D_{pos} sequential database. The frequent strings are then discovered from the constructed tree, and only the contrast ones from D_{neg} are kept in the final extraction step. Later, [29] and [81] improved both computational and space complexity of Chan et al's algorithm.

In order to evaluate the computational complexity of minimal contrast string mining on synthetic trace data we decided not to implement the algorithms cited in the previous paragraph, but rather use the ConSGapMiner algorithm which allows to mine minimal contrast sequences with user-defined maximum gap constraint and which we present later in this section. By making maximum gap constraint equal to 0, ConSGapMiner algorithm can be applied to mine minimal contrast strings.

Figure 4.8 shows that mining minimal contrast strings has linear computational complexity with respect to the size of transactions, hence, is significantly faster than mining the complete set of contrast sequences. However, it is important to remember that the set of returned patterns consists of strings, i.e. sequences whose elements appear contiguously in the dataset. Consider the example trace dataset from Figure 4.3. Regardless of the chosen values for min_sup and max_sup thresholds, minimal contrast string algorithm will never return the injected contrast sequence. Therefore, despite its appealing computational complexity, minimal contrast string mining cannot be applied to detect anomalous system activity from execution traces.

Mining minimal contrast sequences with user-defined maximum gap constraint

The notion of minimal contrast sequences was introduced by Ji et al. in [41] [6]. The authors proposed an algorithm called ConSGapMiner that allows to mine minimal contrast sequences with respect to the maximum gap constraint, which we will refer to as gap-constrained sequences, or simply sequences, if the gap constraint is implied from the context.

[6] the authors used the term *distinguishing* instead of *contrasting*

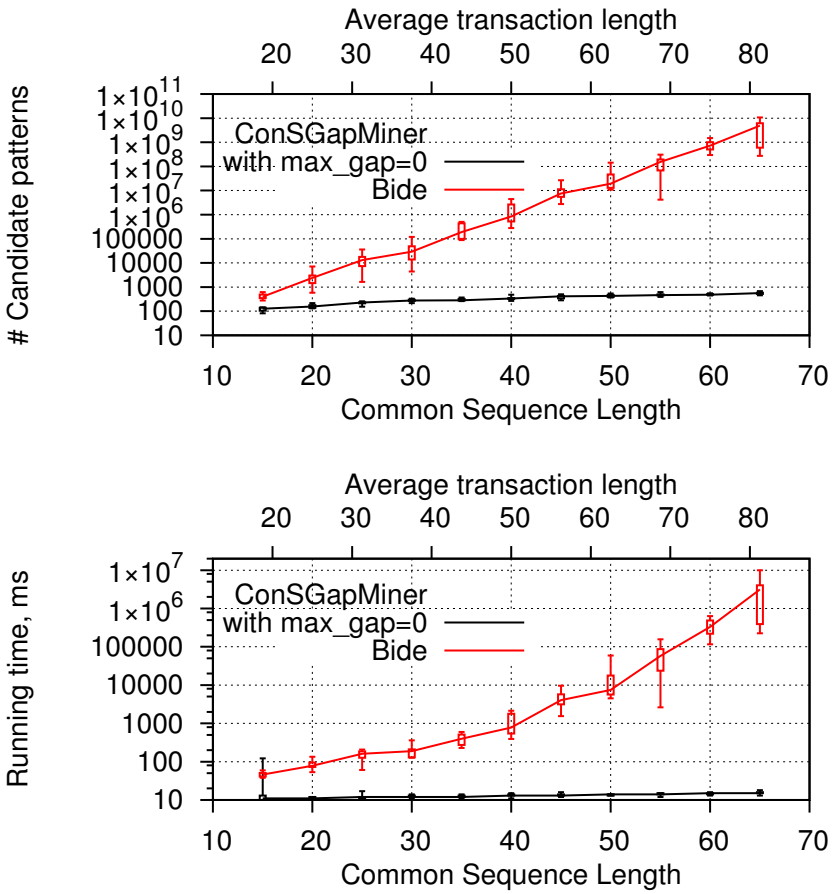


FIGURE 4.8: Comparison of the computational complexities of the Bide algorithm and of the ConSGapMiner algorithm with *max_gap* fixed at 0 run on the same synthetic traces described in Figure 4.6

Conceptually, ConSGapMiner adopts the same technique of indirect mining of minimal contrast sequences that we presented in Section 4.2.2, exploring the space of frequent in D_{pos} sequences via depth-first traversal of the lexicographic sequence tree, and keeping in memory only those sequences which are infrequent in D_{neg}. ConSGapMiner uses both the Apriori pruning and the contrast pruning introduced in Section 4.2.2 (see Definition 3) to avoid growing the patterns that will not be prefixes of any $s \in MCS$. The authors pay special attention to the problem of support calculation of gap-constrained sequences using bitset and boolean operations. We omit the details of these operations as they do not aim at pruning the search space, hence, reducing the number of candidate sequences. Finally, due to the depth-first tree traversal of the lexicographic sequence tree, the post-processing step is used to extract minimal contrast sequences from the set of mined contrast sequences. Note that with $max\_gap = \infty$, ConSGapMiner performs exactly the same search space prunings as ContrastPrefixSpan (see Algorithm 3).

Figure 4.9 shows that the ConSGapMiner algorithm outperforms or matches the computational complexity of the BIDE algorithm on synthetic execution traces only for the smallest values of max_gap parameter. Given such poor performance results, the

ConSGapMiner algorithm can be applied to mine minimal contrast sequences from execution traces only with very small values of *max_gap* parameter. In this case, we run into the same problem as with minimal contrast string mining: the target contrast sequence has a high chance of not being returned in the result set. Indeed, considering the example trace dataset from Figure 4.3, the value of *max_gap* must be set to at least 6 in order for the algorithm to mine the target contrast sequence with *min_sup* of 100%.

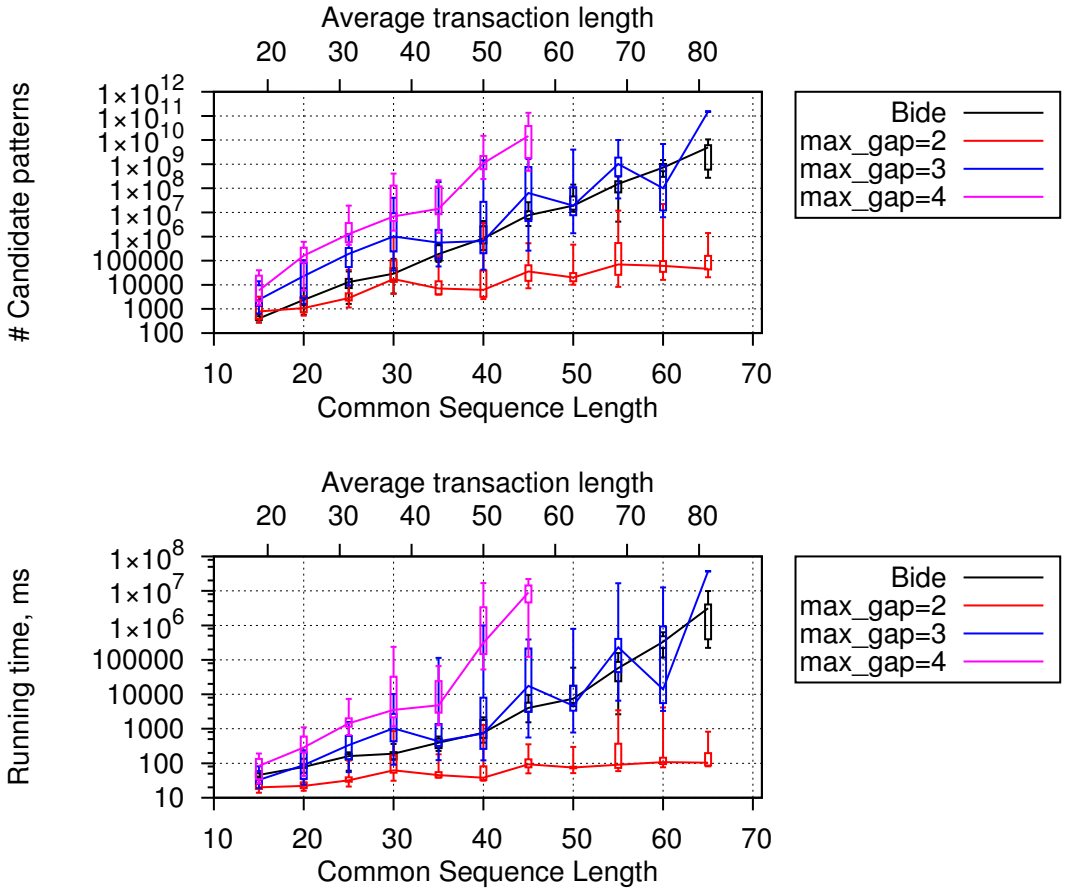


FIGURE 4.9: Comparison of the computational complexities of the Bide algorithm and of the ConSGapMiner algorithm (tested with various *max_gap* values) run on the same synthetic traces described in Figure 4.6[7].

In order to improve the performance of mining gap-constrained contrast sequences, an interesting possibility would be to mine gap-constrained closed contrast sequences. Unfortunately, as we found out, the BackScan pruning is not compatible with the maximum gap constraint. In order to understand why, consider the following example. $D_{pos} = \{\langle a,d,b,c\rangle, \langle a,d,b,c\rangle\}$, $D_{neg} = \{\langle a,e,d,b,c\rangle, \langle a,d,b\rangle\}$, $min\_sup = 2$, $max\_sup = 0$, and $max\_gap = 1$. The gap-constrained *MCS* set contains a single sequence $\langle a,b,c\rangle$. If we use BackScan pruning while exploring the lexicographic tree of frequent in D_{pos} sequences, $\langle a,b,c\rangle$ will never be output. Observe, that upon generating the candidate sequence $\langle a,b\rangle$, BackScan check will detect that item d always appears in $\langle a,b\rangle$'s 2^{nd}

[7] we show only the minimum number of candidate patterns and running time for $max\_gap = 3$ on the traces with $commonSequenceLength = 65$, as the running time on the other 8 synthetic traces was prohibitively large (more than 24 hours).

semi-maximum period. Therefore, ⟨a,b⟩ with its subtree will be pruned, as all the frequent sequences having ⟨a,b⟩ as prefix will be contained in the frequent sequences having ⟨a,d,b⟩ as prefix. At the same time, sequence ⟨a,d,b,c⟩ is not contrasting. Therefore, if BackScan pruning is used, the output set of contrast sequences is empty. At the same time, Li and Wang [47] reported to mine closed frequent sequences with maximum gap constraint. However, their definition of closeness is different from the state-of-the-art one (see Definition 4.12). In fact, the authors consider a sequence as a closed one only if it is not a *contiguous* subsequence of a sequence that has the same support. Therefore, their version of BackSpace pruning strategy allows to prune a candidate sequence only if there exists an item constantly appearing before its first element (i.e. in the 1^{st} semi-maximum period), and not between a given pair of consecutive elements (i.e. i-th semi-maximum period, see Definition 4.14). Such pruning is obviously too restrictive, and does not allow to substantially reduce the number of candidate sequences.

Mining minimal contrast sequences with maximum length constraint

Candidate sequence pruning based on maximum length constraint can be integrated to the already familiar depth-first exploration of the frequent sequence tree. Such pruning basically allows to cut the frequent sequence tree at *max_length* level by pruning all candidate sequences once the number of their elements exceeds *max_length*. In practice, we enriched the ContrastPrefixSpan algorithm with the *max_length* pruning, and once the algorithm returned the set of contrast sequences, we applied the post-processing step described in Section 4.2.2 to filter out all non-minimal contrast sequences. We refer to this mining algorithm as SATM in the rest of this chapter. The performance results of mining the *max_length*-constrained *MCS* set with SATM are presented in Figure 4.10.

Note that SATM runs slower than Bide for the same number of candidate patterns (e.g. see the results for a trace with the common sequence length set to 35 in Figure 4.10). This observation can be explained by the database projection technique used in both PrefixSpan and Bide algorithms. This technique allows to speed-up the discovery of locally frequent items used to extend a particular candidate pattern as the candidate patterns grow in length. Therefore, having the same number of candidate patterns, Bide needs to scan a smaller part of the dataset than SATM, as Bide's candidate patterns will have bigger length (no *max_length* pruning in Bide).

Experimental results show that by introducing the *max_length* constraint, we make the process of mining the constrained *MCS* set from execution traces have linear computational complexity, i.e. the mining algorithm's running time is expected to have a linear growth with the increasing average transaction length in execution trace data. At the same time, none of the contrast sequences having *max_length* or less elements is lost. Given the fact that we are interested only in *minimal* contrast sequences, we can expect that the anomalous system activity is represented in the execution trace by less than N events, where N is a value of *max_length* allowing the mining algorithm to terminate in a reasonable time. Therefore, we adopt the *max_length*-constrained minimal contrast sequence mining algorithm SATM as the last step of our temporal debugging approach.

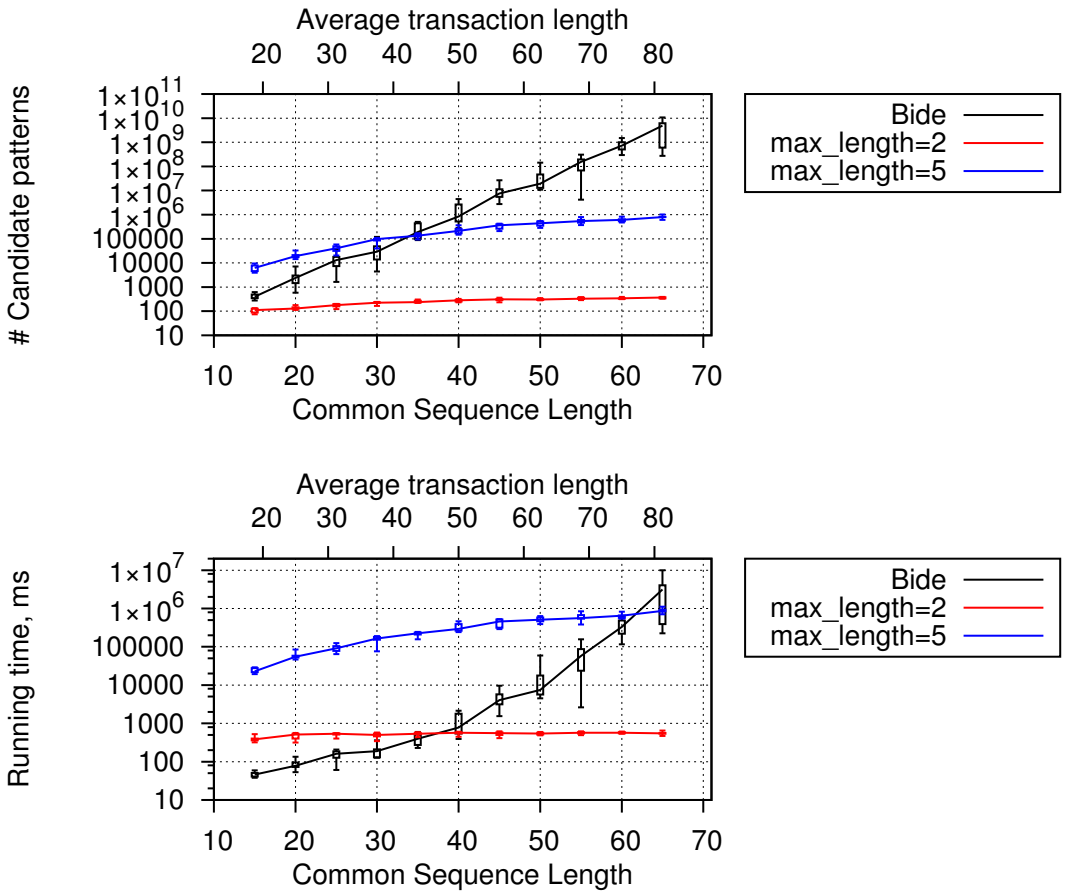


FIGURE 4.10: Comparison of the computational complexity of the Bide algorithm and of the SATM algorithm (tested with various *max_length* values) run on the same synthetic traces described in Figure 4.6

4.3 Considerations on Contrast Pattern Mining from Execution Traces

State-of-the-art sequential pattern mining algorithms fail at the task of an efficient discovery of the complete set of minimal contrast sequences from execution traces of multimedia embedded systems. Of course, implementing distributed versions of these algorithms and running them on a computer cluster would increase their performance. Before doing this, however, it is a wise solution to firstly verify the maximal size of execution trace data that is mineable by parallelized versions of the algorithms on a server machine. Such step would allow to assess the complexity of the task and decide if a distributed version would be able to handle real data. This was our motivation to implement the parallel versions of the state-of-the-art sequential pattern mining algorithms and use a synthetic trace generator to test their performance. As we showed in Section 4.2, the obtained results were disappointing. Indeed, as Figure 4.7 shows, the Bide algorithm run with $min\_sup = 100\%$ required more than an hour to mine all closed frequent sequences from a trace with the average transaction length of 80 events. Moreover, the algorithm's running time grew exponentially with respect to the average

transaction length. The ConSGapMiner algorithm, in its turn, needed too much time to mine a constrained *MCS* set even with small *max_gap* values, as shown in Figure 4.9. These results were very surprising to us, as sequential pattern mining algorithms are known to be successfully applied in various research and industrial fields in order to discover valuable patterns from real data of much bigger size than synthetic traces used in Section 4.2. In this section, therefore, we aim at understanding what precisely makes it so hard to mine minimal contrast sequences from trace data.

We firstly take a closer look at the data used in the literature to evaluate sequential pattern mining algorithms. For historic reasons, sequential pattern mining algorithms were tested on market basket-like datasets. Such datasets are characterized by a big number of short, sparse transactions. Moreover, the goal of such evaluations is to show to which extent a particular algorithm allows to decrease *min_sup* threshold while still allowing to efficiently mine the set of frequent sequential patterns. Trace datasets obtained by the method described in Chapter 3, however, are conceptually different from market basket-like datasets. Indeed, D_{pos} and D_{neg} trace datasets are characterized by very long, dense transactions, while the total number of transactions in both datasets is relatively small. Moreover, with trace data, we are interested only in frequent patterns from the D_{pos} dataset mined with big values of *min_sup*, because anomalous behavior is supposed to characterize the majority of the faulty parts of an execution trace. Fortunately, state-of-the art sequential mining algorithms are also often evaluated on biological sequential databases having long and dense transactions, just like our trace data. However, the reported performances on biological sequence data looked drastically different from the ones we get with synthetic trace data. We, therefore, proceeded to compare the performances of sequential pattern mining algorithms on biological sequences and synthetic traces sharing the same parameters.

For the experiments, we took the first protein family pair, namely *DUF1694* and *DUF1695* [8], used to evaluate ConSGapMiner algorithm [41]. Table 4.1 shows the characteristics of these protein families. We then generated nine sets of synthetic D_{pos} and D_{neg} datasets having the same characteristics. We then ran both Bide and ConSGapMiner algorithms on protein data as well as on synthetic trace data and compared the results [9]. The Bide algorithm was run with $min\_sup = 100\%$ to mine all frequent closed sequences from D_{pos} datasets. Likewise, the ConSGapMiner algorithm was run with $min\_sup = 100\%$ and $max\_sup = 0\%$ in order to mine contrast sequences between D_{pos} and D_{neg} datasets for various values of *max_gap*. Table 4.2 and Figure 4.11 show the performance results of Bide and ConSGapMiner algorithms correspondingly.

Protein family name	alphabet size	# transactions	av. transaction length
DUF1694(D_{pos})	20	16	123
DUF1695(D_{neg})	20	5	186

TABLE 4.1: Characteristics of protein datasets

[8] available at http://pfam.xfam.org/

[9] for synthetic trace data, we picked the median values of the number of generated candidate patterns and the running time among 9 generated pairs of D_{pos} and D_{neg} datasets.

As can be observed in Table 4.2, the difference in computational complexity of the Bide algorithm run on biological sequence data and trace data is flagrant. The same can be said about the ConSGapMiner algorithm (see Figure 4.11). It is, therefore, not only the size but some structural peculiarity of trace data that makes it hard to be mined with state-of-the-art sequential pattern mining algorithms. In the rest of this section, we take a closer look at trace data and analyze the reasons why Apriori, BackScan, and contrast prunings are not effective enough to allow mining the complete set of minimal contrast sequences from industrial trace data.

Origin of the D_{pos} dataset	# candidate patterns ($\times 10^6$)	Running time (s)
Protein	5.3	4.3
Trace	28,887.3	12,414.8

TABLE 4.2: Performance results of the Bide algorithm on protein and trace datasets

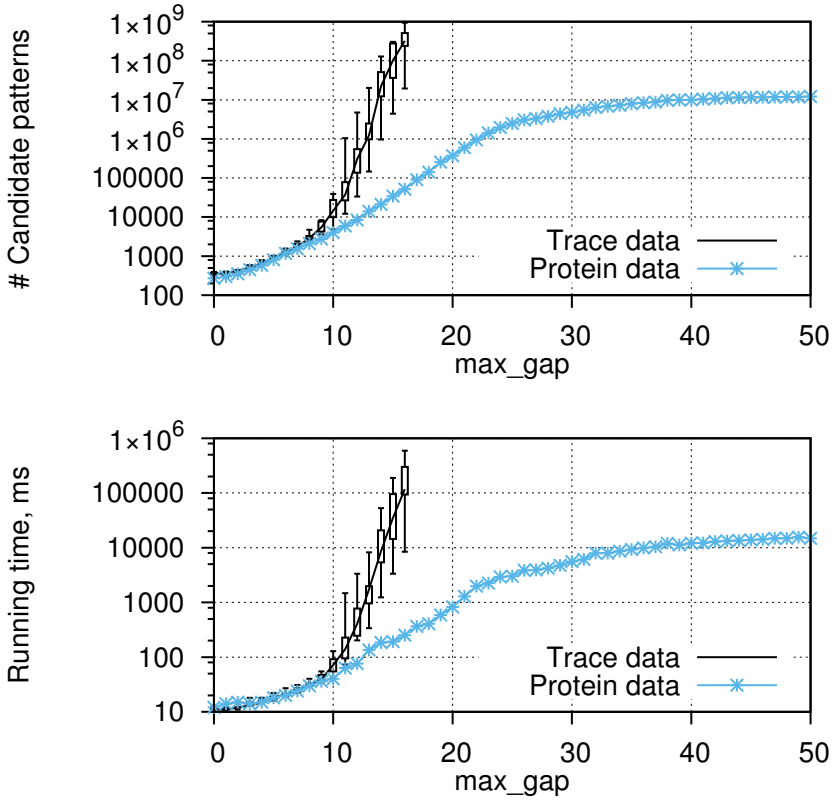


FIGURE 4.11: Performance results for the ConSGapMiner algorithm run on protein and trace datasets

4.3.1 Apriori Pruning on Trace Datasets

Recall that transactions obtained with splitting the original execution trace by the target actor's invocations share a sequence of events representing the executions of actors having the same period as the target actor. Assume that every transaction in the D_{pos} dataset consists only of this common sequence, without any noisy events between its elements. If the common sequence has N elements, then the sequential pattern search space will

contain 2^N frequent sequences, even with *min_sup* equal to 100%. Therefore, the Apriori pruning will not prevent sequential pattern mining algorithms to eliminate any of those 2^N candidates which is obviously too big of a number even for relatively small values of N. Moreover, adding "noisy" events to transactions will make the number of candidate patterns to grow even further. To provide an idea of how costly is generation of 2^N patterns, Figure 4.12 shows the running time of PrefixSpan algorithm which, as explained in Section 4.2.2, relies only on the Apriori pruning and, hence, needs to explore all 2^N candidate patterns.

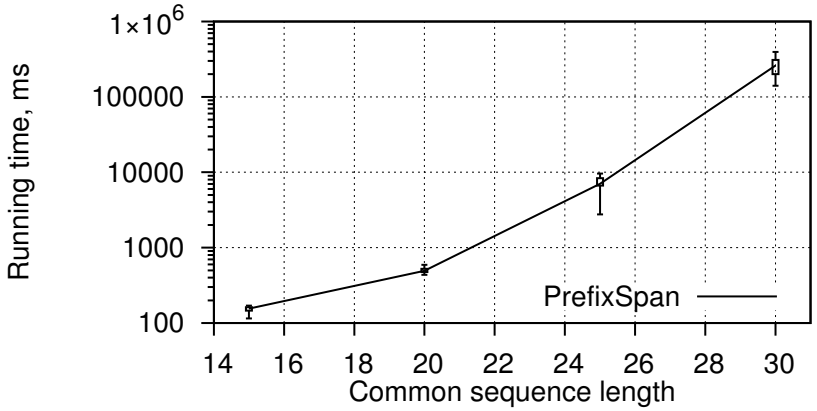


FIGURE 4.12: Performance results for PrefixSpan run on a the D_{pos} dataset containing identical transactions of various lengths; alphabetSize=20, numTransactionsDpos=10, noiseRate = 0.0

4.3.2 BackScan Pruning on Trace Datasets

It is easy to understand why Apriori pruning does not allow to improve computational complexity of mining frequent sequential patterns from the datasets having long and dense transactions. However, if the Apriori pruning is coupled with the BackScan pruning, as done in the Bide algorithm, then the set of frequent closed sequential patterns can be efficiently mined from such datasets, as reported in [80]. In an idealistic (and, unfortunately for embedded software programmers, unrealistic) case when every single instruction is executed on an MPSoC platform with the same period, all the transactions will be represented as the same sequence of events of length N. In this case, Bide will test only N candidate sequences, as all the 1-patterns, except the one appearing first in the transactions, will be pruned with BackScan. In reality, however, the common sequence shared among all the transactions in a trace dataset is mixed with a very big number of events which occur seemingly random in the dataset. Therefore, we need to consider not only the length of the common subsequence but also the amount of such noisy events in order to understand the reason BackScan fails to efficiently prune the search space in trace datasets.

Figure 4.13 shows how the computational complexity of the Bide algorithm changes depending on the amount of noisy events present in transactions from the D_{pos} dataset. The exponential growth of both the number of candidate patterns and the running time

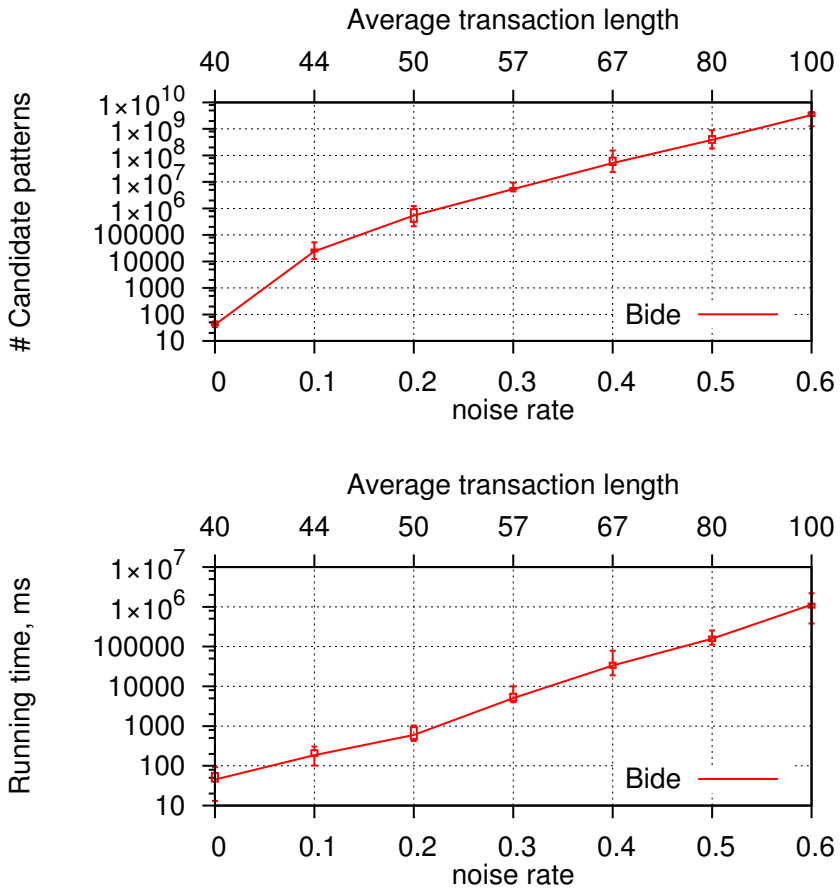


FIGURE 4.13: Computational complexity of the Bide algorithm on synthetic trace datasets generated with various noiseRate values; commonSequenceLength=40, alphabetSize=20, numTransactionsDpos=10.

suggests that the cause of Bide's bad performance on trace data lies in random noisy events. However, when we generated a dataset having the same parameters, but where each transaction consisted of a completely random sequence of events, Bide showed a very good performance (see Figure 4.14). Indeed, randomly generated transactions no longer share long common sequences, and the Apriori pruning, being a part of the Bide algorithm, allows the explored prefix tree to be relatively shallow, while BackScan further enhances the performance of the algorithm on such data.

This is how we arrived at a conclusion, that it is the combination of a rather long common subsequence between all the transactions in the D_{pos} dataset and the presence of noisy events that makes the Bide algorithm fail to efficiently mine the set of closed frequent sequences from trace data. Indeed, in the protein dataset DUF1694 presented earlier in Section 4.3 the longest common subsequence among protein sequences has only 14 elements [10], which allowed Bide to show a good performance on this dataset. To confirm our findings, we ran Bide on synthetic traces with various average transaction lengths, where the increase in transaction length resulted both from augmenting the common

[10]We used a heuristic algorithm for finding the longest common subsequence available at http://search.cpan.org/~vmoiseev/Algorithm-MLCS-1.02/lib/Algorithm/MLCS.pm

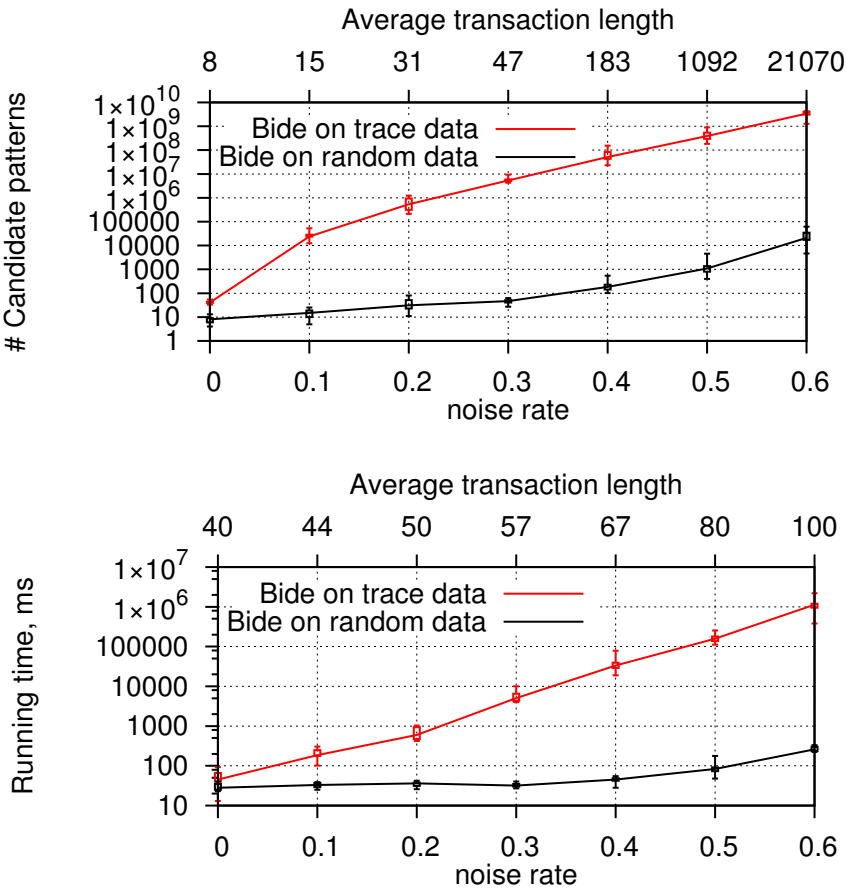


FIGURE 4.14: Comparison of the computational complexity of the Bide algorithm run on a trace dataset and a randomly generated dataset.

sequence length, and adding more noisy events. Indeed, as can be seen in Figure 4.15, Bide shows an exponential growth in computational complexity with respect to the average sequence length. Finally, increasing the number of transactions in the D_{pos} dataset does not result in the reduced number of candidate patterns (see Figure 4.16). All in all, this was a very surprising finding, as pattern mining literature does not report an exponential growth of the computational complexity of closed frequent sequence mining algorithms with respect to the dataset size.

4.3.3 Contrast Pruning on Trace Datasets

Contrast pruning introduced by Ji et al. [41] (see Definition 3) is effective in case there exists a big number of relatively short contrast sequences between D_{pos} and D_{neg} datasets. Indeed, their presence would allow to prune a lot of branches of the prefix tree which start close to the root, hence, significantly reduce the part of the search space needed to be explored by the mining algorithm. At the same time, D_{pos} and D_{neg} datasets obtained from execution traces do not contain a big number of short contrast sequences. The first reason is that transactions in D_{pos} are highly similar to those in D_{neg}, as they come from the same use case execution session. The second reason is that, normally, the

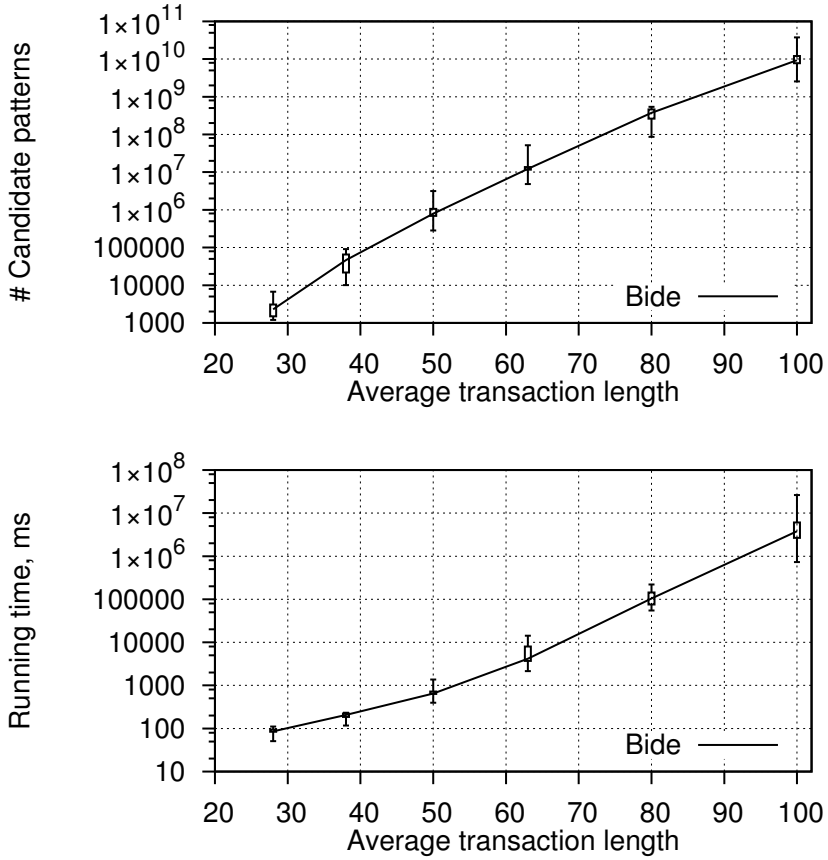


FIGURE 4.15: Performance of the Bide algorithm on trace datasets generated with *alphabetSize*=20, *numTransactDpos*=10, and average transaction lengths defined by the following pairs of values of *commonSeqLength* and *noiseRate* parameters: (20;0.3), (25;0.35), (30;0.4), (35;0.45), (40;0.5), (45;0.6).

number of transactions in D_{neg} is significantly bigger than in D_{pos}. Indeed, transactions from the D_{pos} dataset represent anomalous parts of the execution trace, which are much less numerous than the normal parts. Therefore, short sequences that are frequent in D_{pos} and represent random combinations of noisy events will have a high probability to occur in some transactions from D_{neg}, too. These two properties of frequent sequences in execution trace data are, from the one hand, an important hindrance to efficient mining of minimal contrast sequence, but from the other hand, is a good news for a developer, as the output is expected to be very compact and contain only sequences strongly correlated to the observed temporal bug.

4.3.4 Can Bioinformatics Help Mining Contrast Sequences from Execution Traces?

As we found out earlier in this chapter, the structural similarity of transactions in D_{pos} and D_{neg} datasets combined with the external characteristics of trace data, such as the size of the alphabet, the number and the length of transactions, make the state-of-the-art sequential pattern mining algorithm exhibit unacceptable performance on execution

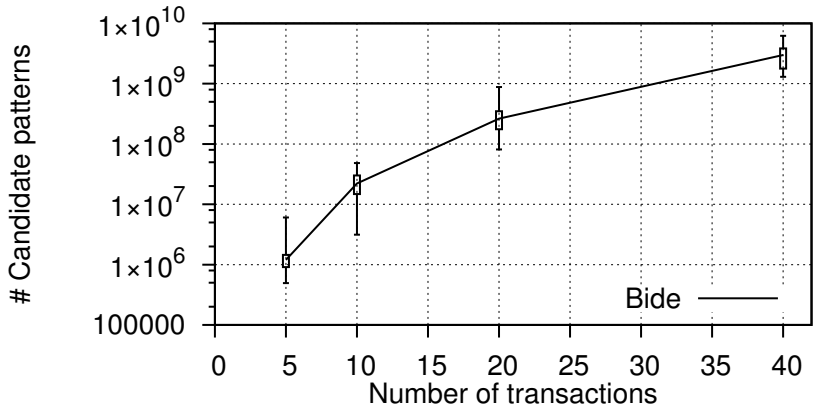


FIGURE 4.16: Performance of the Bide algorithm on trace datasets with various number of transactions. *alphabetSize*=20, *commonSeqLength*=35, *noiseRate*=0.45, average transaction length = 64.

trace data. A reader familiar with the problem of biological sequence analysis, however, may point that biological sequences (or, bio-sequences) representing protein, DNA, or RNA macromolecules have similar properties to trace data. Indeed, protein macromolecules, for example, can be represented as long sequences of 20 different types of amino acid residues. Moreover, proteins belonging to the same family are structurally similar to each other. Therefore, a group of proteins belonging to the same family is essentially a dataset of long, dense, and similar to each other sequences. In this section, we take a brief look at bio-sequences as well as at the task of finding similarities and differences between groups of bio-sequences, in order to understand if the algorithms from bioinformatics can be applied to mine minimal contrast sequences from trace data.

The main interest of comparing bio-sequences for biologists is to understand if a group of proteins (or else DNAs, RNAs) originate from a common ancestor. If a group of proteins share a common subsequence (called a *conserved* subsequence) then there is a high chance that such subsequence has a particular functional role, making proteins similar. A plethora of techniques was proposed in bioinformatics field in order to identify such subsequences, most notably global and local sequence alignment algorithms [55]. While sequence alignment allows to visually compare a pair or a group of bio-sequences, this task becomes prohibitively expensive if input sequences grow in size and in quantity. At the same time, pattern mining algorithms provide a more efficient alternative for grouping bio-sequences into families. If a common pattern is discovered in a set of biologically related sequences, it is possible that the presence of this particular pattern is important for the biological function of the corresponding macromolecule [18]. Contrast pattern mining, in its turn, finds its utility in bioinformatics when a pair of protein families must be compared in order to understand which functionalities, expressed as conserved sequences, make those families different [48]. Contrast patterns then can be used either to represent discriminating functionalities between two protein families or to classify an unlabeled protein to the most appropriate family.

It can be seen that the characteristics of bio-sequences as well as the goal of contrast conserved subsequence mining are highly similar to our problem of abnormal system activity detection in given parts of the execution trace. But can pattern mining algorithms developed for biological data be used in our context of execution trace data? The answer, unfortunately, is no. In fact, pattern mining algorithms used in bioinformatics field incorporate some properties specific to biological sequences in order to deal with the computational complexity of extracting conserved subsequences from a group of bio-sequences. A short list of such properties is presented below.

1. Pattern similarity. The definition of similarity between patterns mined from biological sequences is different from the strict symbolic and order similarity we require from minimal contrast sequences mined from trace data. Two subsequences of different bio-sequences way be considered equal even if an item at position i in one subsequence is not the same item found at position i in another subsequence. The reason of this is that, in case of proteins, an amino acid can mutate (i.e. become another amino acid), or swap positions while still preserving the functional role of the given region within a protein. That is why, pattern mining algorithms found in bioinformatics literature rely on amino acid substitution matrices to assess how much a particular mutation or swapping of amino acids change the functional role of a protein's region [18].

2. Constraints on mined patterns. It is important to restrict the considered area of protein sequences in order to correctly detect possible amino acid substitutions in a given pattern. Therefore, biological sequence mining algorithms impose various constraints on patterns, such as regular expressions, maximum gap, size of sliding window, etc [70] [58]. These constraints allow to considerably reduce patterns' search space, without introducing a risk of missing functional regions in a given set of bio-sequences. For example, if the elements of a pattern appear scattered over the input sequences, then this pattern does not represent a functional region.

3. Heuristic algorithms. Biologists are often interested in the most significant functional regions of a given set of macromolecules. Therefore, there exists a variety of scoring functions to determine which patterns are the most significant among all existing patterns. That is why heuristic pattern mining algorithm are widely used in bioinformatics field in order to efficiently mine only the most significant patterns with respect to a given scoring function.

Incorporation of domain knowledge into bio-sequence pattern mining algorithms make them particularly efficient on real biological data. On the other hand, none of the properties of pattern mining from bio-sequences listed above make sense for execution trace data. Firstly, the same event can have different properties depending on the particular context it was executed in. Hence, there is no analogy for bio-sequence substitutional matrices that can be constructed for trace data. Secondly, the elements of a contrast pattern can appear in any part of a transaction, because the relative positions of a pair

of events within a trace are not a reliable indicator of their semantic relationship. Finally, the full set of minimal contrast patterns must be output, as their significance can be assessed only by the developer, and not using some scoring function. In conclusion, bio-sequence pattern mining algorithms can not be directly applied to mine minimal contrast sequences from execution traces, as trace data do not share same properties with bio-sequence data.

4.4 Conclusion

In this chapter, we discussed the problem of contrasting two sets of traces: D_{pos} containing anomalous parts of the original system's execution trace and D_{neg} capturing its normal behavior, both obtained with the method presented in Chapter 3.

In Section 4.1, we expressed the problem of detecting anomalous system behavior in the D_{pos} dataset as a pattern mining task. More specifically, we argued that minimal contrast sequences mined between D_{pos} and D_{neg} datasets is a good representation of anomalous system behavior, as they allow to reveal fine-grained system activity characteristic of traces in the D_{pos} dataset but not of those in the D_{neg} dataset.

In Section 4.2, we explained how minimal contrast sequences can be mined using state-of-the-art sequential pattern mining algorithms. In order to compare the performances of these algorithms, we developed a synthetic trace generation tool. In Section 4.2.1, we argued that a direct extraction of the *MCS* set from the set of all possible sequences of execution events is a computationally infeasible task even for the smallest, non-realistic traces. We then proceeded to show how the *MCS* set can be extracted from the set of frequent in the D_{pos} dataset sequential patterns (Section 4.2.2). To perform this task, we introduced the state-of-the-art frequent sequence mining algorithms, and then tested their performance on synthetic trace datasets. We then concluded that such algorithms do not allow to efficiently extract the full *MCS* set from execution trace data. Therefore, in Section 4.2.3, we considered adding constraints to the final *MCS* set in order to reduce computational complexity of pattern mining task. First, we analyzed the mining of minimal contrast strings, i.e. sequences whose elements must appear contiguously in the execution trace. It showed a good performance on trace data but allowed to mine only a very limited set of patterns from the execution traces. Second, we left the choice of the maximum number of events allowed to appear between the elements of a sequential pattern (the *max_gap* constraint) to the user. We then evaluated ConSGapMiner – a state-of-the-art algorithm to mine minimal contrast sequences with the *max_gap* constraint. This algorithm showed poor performance on synthetic traces even for the smallest values of the *max_gap* parameter, which made it unsuitable for mining suspicious system activities from execution traces. Finally, we considered the task of mining the *MCS* set with the *max_length* constraint, which limits the maximum number of elements in a sequential pattern. As it showed a good performance for relatively big values of *max_length* and was not supposed to lose a big fraction of possible

suspicious system activities, we decided to adopt such way of mining minimal contrast sequences as the final stage of SATM.

Curious about the gap between the reported in the literature performance of sequential pattern mining algorithms and our own experience on trace data, we analyzed the computational complexity of these algorithms in Section 4.3. As we found out, the combination of a long common subsequence shared by the traces, as well as the presence of a big number of noisy elements in each trace makes pruning techniques adopted in sequential pattern mining algorithms fail to cope with search space explosion. We also concluded that a rich set of algorithms developed for mining patterns from biological sequences is, unfortunately,not applicable to mine MCS from execution trace data.

Chapter 5

Use Cases

In Chapters 3 and 4, we presented our novel approach for temporal debugging of embedded streaming applications which we called SATM. Its main goal is to automate the process of resolving temporal bugs, so that given a minimal input from the developer (an execution trace of the system and a dataflow graph of the target application), SATM returns a precise sequence of executed events related to the origin of the QoS problem. Such automatic approach is of great value for software developers, as temporal bugs have a tendency to manifest themselves at the final stages of software development, when a stringent delivery deadline of the entire embedded system imposes a great pressure on the developers to come up with a fix as quickly as possible. The currently most popular way to perform temporal debugging consists in manual analysis of executed events using trace visualization tools. This process usually requires a profound expertise and a lot of time in order to unearth a suspicious system activity.

As we showed in Chapter 3, SATM can pinpoint problematic zones in an execution trace, as long as the target application complies with the properties of embedded streaming software (dataflow model of programming, periodic scheduling). At the same time, detection of anomalous system activity in the extracted parts of the execution trace is an extremely complicated task. Indeed, in Chapter 4 we found out that the state-of-the-art pattern mining algorithms are not able to efficiently extract a complete set of execution event sequences representing the most succinct contrasts between different parts of a trace.

In this chapter, we show that despite the theoretical complexity of the temporal debugging problem, SATM still fulfills its goal of discovering abnormal system activity from real-world execution traces. We present three use cases of streaming applications exhibiting temporal bugs and show how SATM was successfully applied on their execution traces.

5.1 Description of Use Cases

In this section, we present three use cases which we considered in order to test the proposed temporal debugging approach. The first one was created by ourselves using GStreamer – an open-source multimedia framework widely adopted in embedded systems. The other two are industrial use cases coming from STMicroelectronics. We next introduce each of these use cases.

GStreamer use case

GStreamer is a popular dataflow-based framework for creating streaming multimedia applications [76]. As GStreamer is an open-source software, there exists a variety of freely available components that can be easily plugged together to form complex dataflow applications. Examples of popular applications based on GStreamer include Totem (the default movie player for GNOME desktop), Pitivi video editor and others [1]. It is, of course, also possible to combine proprietary components using the GStreamer framework which made it a widely adopted standard to program embedded streaming applications, which put a particular accent on QoS properties.

Our goal of creating a multimedia application using the GStreamer framework and then manually injecting a temporal bug in it was to get an execution trace of a real-world streaming application containing an anomalous sequence of events known in advance. This way, we were able to evaluate the ability of SATM to pinpoint the problematic system activity.

We assembled a simple audio/video-decoding application using the existing components coming with the GStreamer development toolkit. Figure 5.1 shows the dataflow graph of the created application with the name of components as available in `gst-plugins-good` library [2]. This application essentially takes a media file, demultiplexes its audio and video elementary streams, transfers them to the respective decoders for actual decoding (*faad* and *avdec_h264* actors on Figure 5.1), and finally sends the decoded frames to the default audio and video rendering devices.

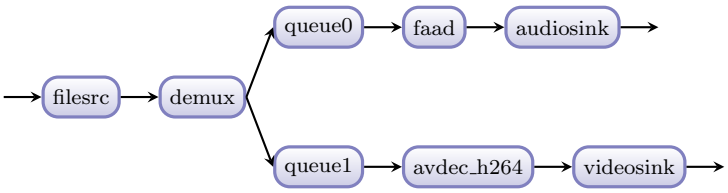


FIGURE 5.1: Dataflow graph of the GStreamer application

In order to introduce temporal bugs into the created media application, we decided to plug in our custom *intruder* component just before the video decoder actor, as shown in Figure 5.2. The goal of this component is to introduce a delay to the dataflow graph at random moments of time, so that the application produces a stuttering video because of

[1] http://gstreamer.freedesktop.org/apps/
[2] http://gstreamer.freedesktop.org/modules/gst-plugins-good.html

late decoded frames. This is done in the following way. *Intruder* implements 3 random bit generators named A, B and C. Each of them calls a specific function depending on the value of the generated bit (e.g. $A1$, $B0$). These functions are instrumented, hence, their executions are captured in the trace. If a sequence "010" is generated (i.e. a sequence of $A0$, $B1$ and $C0$ function calls is observed in the trace), then *intruder* sleeps for a significant amount of time, so that the delay propagates through the downstream actors resulting in late video frame display, thus, reducing the application's QoS.

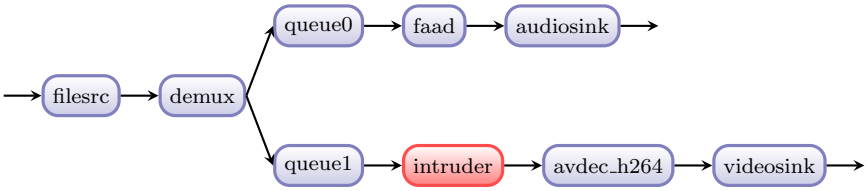


FIGURE 5.2: Dataflow graph of the GStreamer application with the intruder actor

The Gstreamer framework provides an extensive tracing functionality. It is possible to choose between 8 levels of execution tracing, where the highest one logs all memory dump messages and the lowest one writes to the trace only fatal errors. We decided to trace our GStreamer-based application on level 7, which results in tracing all function calls and log messages, but without including the information about the current state of the working memory. This way, we ran our media application for 1 minute on a Linux machine and obtained an execution trace containing 1,201,855 timestamped events with the alphabet size (i.e. number of distinct events) of 1363.

TSRecord use case

This use case consists of a single application called TSRecord designed for set-top boxes equipped with STiH416 MPSoC, and whose goal is to receive a video stream from an Internet Protocol network (IP network) and record it onto secondary storage in the same format it was received. This application represents an important functionality of set-top boxes to record a TV program for later viewing. Given the simplicity of TSRecord, its dataflow graph contains a single actor called *write*, as shown in Figure 5.3. An observed QoS problem concerned high-definition videos which contained numerous missing frames once recorded onto an external USB hard drive. The trace provided by STMicroelectronics software developers has the alphabet of 111 elements and a total of 725,689 recorded events during 4 minutes of execution tracing.

FIGURE 5.3: Dataflow graph of the TSRecord application

DVBTest use case

DVBTest is a streaming application widely used by STMicroelectronics software developers to test the decoding and encoding of video streams received from various transmission medias (satellite, IP network, cable, etc.). Figure 5.4 shows its dataflow graph which consists of 7 actors. The QoS problem in this use case concerned video streams coming from an IP network, as they contained missing frames once decoded with DVBTest. This problem appeared right after a particular software upgrade. The obtained trace contains 534,779 events recorded during 2 minutes of DVBTest execution, and has the alphabet size of 135 distinct events.

FIGURE 5.4: Dataflow graph of the DVBTest application

5.2 Detection of Anomalous Zones

In this section, we apply SATM on the three use cases presented in Section 5.1 in order to detect the anomalous parts in their execution traces. The goal of this section is twofold. First, we show the importance of clustering actor's inter-occurrence intervals for the task of discovering the actor's period. Second, we report the first actors in dataflow graphs having a significant amount of outlier intervals, which are subsequently used to split the execution trace into D_{pos} and D_{neg} datasets.

5.2.1 GStreamer Use Case

Recall that we introduced the GStreamer use case as a "semi-synthetic" one: having injected temporal bugs ourselves, we know beforehand that the first actor having significant violations of its period is the *avdec_h264* actor. Indeed, as the *intruder* actor sleeps for a significant amount of time every once in a while, the delay propagates through the downstream actors, namely *avdec_h264* and then *videosink*, and causes the output of the entire application to be delivered late. At the same time, all other actors, *intruder* included, must always respect their periods.

SATM succeeded in finding the periods of all the GStreamer actors, as shown in Figure 5.5. Without clustering, however, actors do not appear to be invoked periodically, as we show further.

Consider Figure 5.6 which presents two distributions of inter-occurrence intervals of the *demux* actor: the one of unprocessed intervals extracted from the adjacent

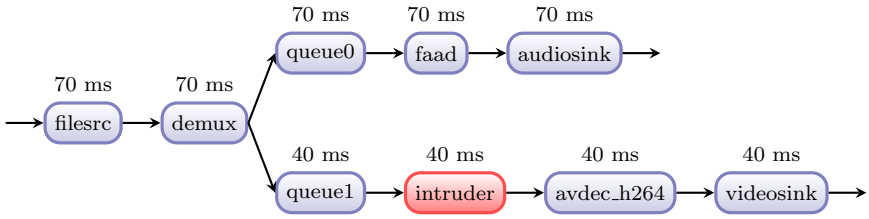


FIGURE 5.5: Dataflow graph of the GStreamer application with the actors' periods detected by SATM

occurrences of the *demux* actor in the execution trace, and the other one of clustered intervals. Without clustering, the intervals tend to be equal to 2 ms with the value of the *QCoD* metric equal to 43%. Such a high rate of dispersion signifies that the values of temporal intervals between the *demux*'s invocations are highly scattered, i.e. *demux* does not appear to be invoked periodically. At the same time, we know beforehand that this actor, in fact, executes periodically. Therefore, the conclusion one makes analyzing the *demux*'s invocations as they appear in the trace is erroneous. However, if the intervals are clustered prior to period detection, then the *demux* actor tends to be invoked every 70 ms with *QCoD* equal to 1.1%. A small dispersion rate tells that this actor appears in the execution trace as a highly periodic event once its occurrences are clustered.

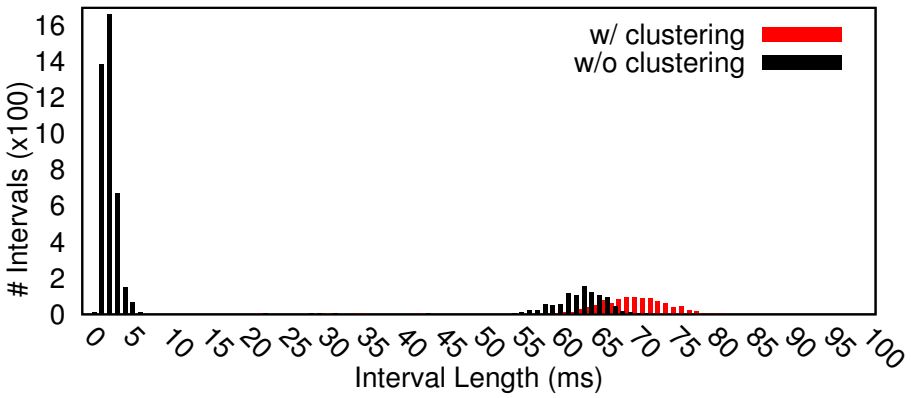


FIGURE 5.6: Inter-occurrence interval distribution of the *demux* actor

Next, consider Figure 5.7 which shows the distributions of inter-occurrence intervals of the *avdec_h264* actor. Similar to the *demux* actor, in case the clustering is not applied, the *QCoD* metric has a high value of 35.7%, signaling that the intervals between the *avdec_h264*'s invocations are highly scattered around the median of 40 ms. At the same time, if clustering is applied before the dispersion is measured, *QCoD* becomes equal to 3.1%, showing that the intervals between the *avdec_h264*'s invocations are very close to its central tendency equal to 40 ms. This way, although the medians of both distributions are equal to each other, we cannot conclude that *avdec_h264* is a periodic actor if its intervals were not previously clustered.

After applying the inter-quartile rule for outliers SATM detected 241 intervals between the *avdec_h264*'s invocations which were significantly bigger than the detected period of 40 ms (see the intervals falling into the range [80 ms, 130 ms] in Figure 5.7). Being the most upstream actor having numerous outlier intervals, *avdec_h264* was used to split the execution trace into the D_{pos} and D_{neg} datasets. As the result, D_{pos} had 241 subtraces with the average number of events per subtrace equal to 1574 (the maximal number of events in a subtrace was 2802), while D_{neg} had 1187 subtraces with the average number of events per subtrace equal to 628 (the maximal number of events in a subtrace was 1888).

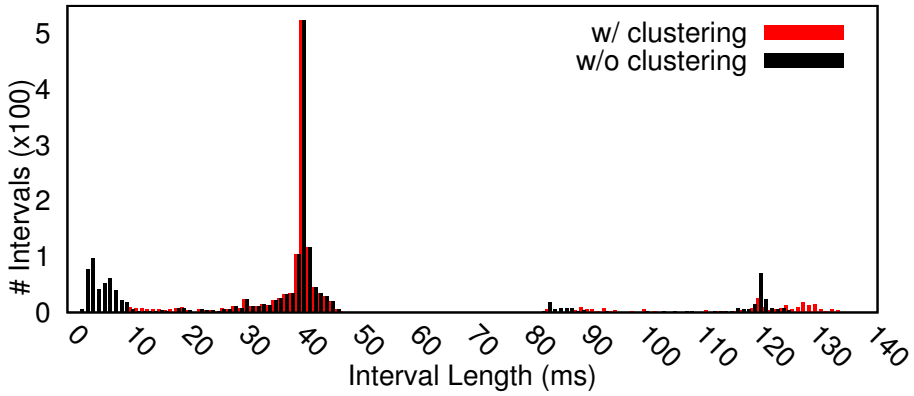
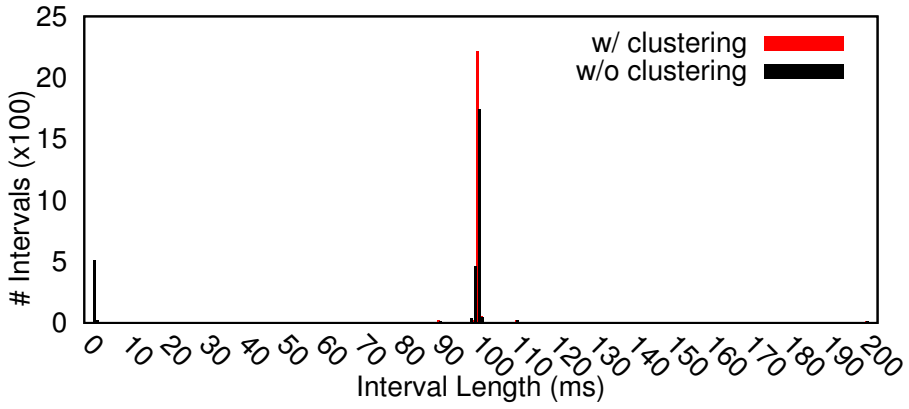


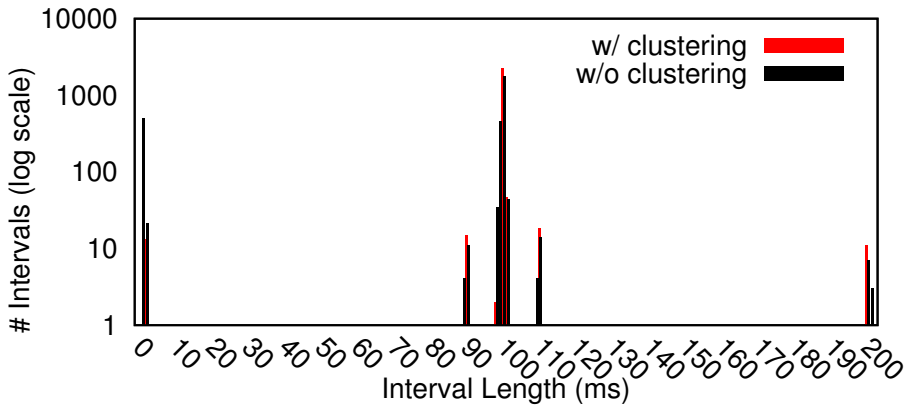
FIGURE 5.7: Inter-occurrence interval distribution of the *avdec_h264* actor

5.2.2 TSRecord Use Case

Recall that the TSRecord application presented in Section 5.1 has only one actor called *write*. Figure 5.8 shows the distributions of both clustered and non-clustered intervals between *write*'s invocations in the execution trace. The value of $QCoD$ is equal to 0.6% when the clustering is not applied, and is equal to 0.2% when SATM clusters the inter-occurrence intervals of the actor, while the median is equal to 100 ms in both cases. As we can observe, the *write* actor appears to be highly periodic even if no clustering is applied (see Figure 5.8(a)). Indeed, TSRecord is a "lightweight" use case with very little application or system activity occurring on the underlying hardware. Therefore, the *write* actor does not compete with other application or system threads for computational resources, and a limited preemption is observed in the execution trace. SATM then applied the quartile rule for outliers and detected 26 intervals whose values greatly exceeded the found period of 100 ms (clearly visible in Figure 5.8(b)). Finally, SATM split the original execution trace with respect to the outlier intervals and returned a D_{pos} dataset having 26 subtraces with 1025 events on average (the maximum subtrace length is 2216), as well as a D_{neg} dataset having 2287 subtraces with 293 events on average (the maximum subtrace length is 1064).



(A)



(B)

FIGURE 5.8: Inter-occurrence interval distribution for *sys_write* actor: (a) with Y-axis showing the number of intervals multiplied by 100; (b) with Y-axis in log scale.

5.2.3 DVBTest Use Case

Figure 5.9 presents the actors' periods discovered by SATM from the execution trace of the DVBTest use case. Note that the *Demuxer* actor was not instrumented by the developers and, hence, was not present in the execution trace.

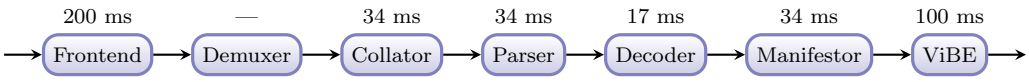


FIGURE 5.9: Dataflow graph of the DVBTest application with the actors' periods detected by SATM

Consider Figure 5.10 which shows the distribution of the inter-occurrence intervals of the *ViBE* (Video Backend) actor whose role is to send video frames to the display of the multimedia device. As we can see in Figure 5.10(a), SATM did not simply merge smaller intervals with the bigger ones in case they occur in each other's vicinity, but instead grouped intervals of different lengths into substantially longer ones having a uniform length of 100 ms. The observed clustering results seem counterintuitive but can be easily understood knowing the operation of a

video decoder. One of the important functionalities of a video decoding application is the adjustment of the number of decoded video frames sent to the display in a given interval of time. The problem comes from the difference between the number of frames decoded per second, known as the *frame rate*, and the number of frames rendered on the screen per second, known as the *refresh rate* of the display. Vertical synchronization (vsync) technique is used in video decoding applications to deal with this problem [3]. Depending on the video decoding frame rate and the display's refresh rate, vsync chooses which frames to drop or, conversely, retain from sending to the display for an additional cycle. Returning to the results of clustering the *ViBE* actor's inter-occurrence intervals, we can observe in Figure 5.10(a) (for the case when clustering was not applied) that most of the time this actor is invoked every 17 ms, meaning that a decoded frame is sent to the display every 17 ms. However, one fifth of the frames are retained from being sent to the display for additional 17 ms. This way, the ViBE actor appears in the trace with the following intervals: 17 ms, 17 ms, 17 ms, 17 ms, 34 ms, 17 ms, 17 ms, 17 ms, 17 ms, 34 ms, 17 ms, and so on. The clustering algorithm adopted in SATM was able to detect this pattern of the ViBE actor's invocations and grouped them into $17 + 17 + 17 + 17 + 34 \approx 100$ ms intervals. Indeed, the low value of $QCoD$ equal to 0.8% shows that this actor can be observed in the execution trace every 100ms.

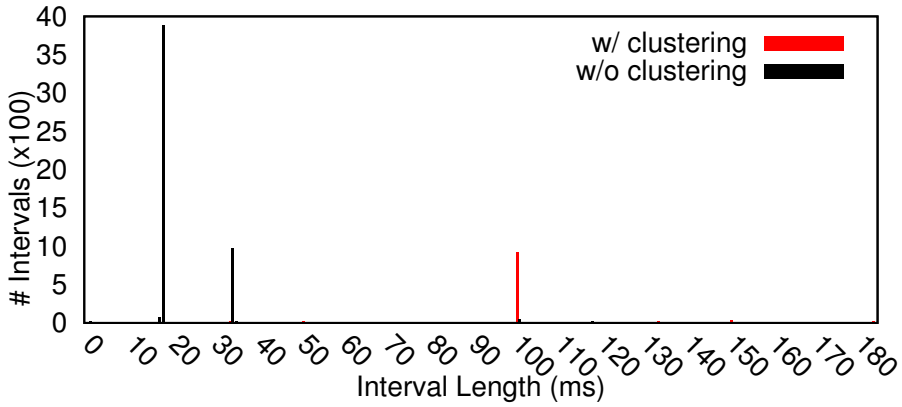
ViBE was also the first actor in the DVBTest dataflow graph which had a number of intervals significantly bigger than the detected period of 100 ms (the outlier intervals are clearly visible in Figure 5.10(b)). Therefore, SATM used the clustered intervals between this actor's occurrences in order to split the original execution trace into the D_{pos} and D_{neg} datasets. As a result, the D_{pos} dataset consists of 66 subtraces having 729 events on average (the maximum number of events in a subtrace is 1630); and the D_{neg} dataset consists of 972 subtraces with an average of 500 events per subtrace (4474 being the maximum number of events in a subtrace).

5.3 Mining Suspicious System Activity

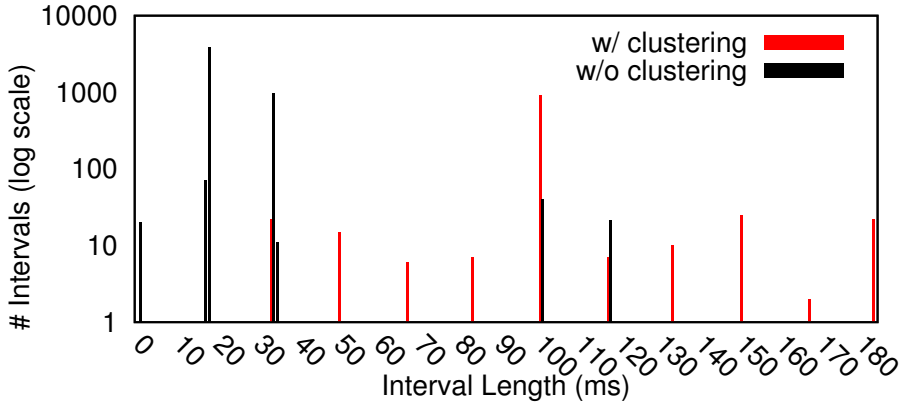
Having detected the anomalous parts of an execution trace, SATM proceeds in mining minimal contrast sequences (i.e. the *MCS* set), which will describe what makes those parts of the trace anomalous. As we argued in Chapter 4, the most appropriate way to mine the *MCS* set is to use a "hybrid" algorithm which combines the PrefixSpan algorithm with the contrast pruning from the ConSGapMiner algorithm and introduces the *max_length* constraint. In the rest of this section, we will refer to this mining algorithm as SATM.

In this section, we present both quantitative and qualitative analysis of applying SATM to mine the *MCS* set from the GStreamer, TSRecord and DVBTest use

[3] http://hardforum.com/showthread.php?t=928593a



(A)



(B)

FIGURE 5.10: Inter-occurrence interval distribution for the *ViBE* actor: (a) with Y-axis showing the number of intervals multiplied by 100; (b) with Y-axis in log scale.

cases. In Sections 5.3.1, 5.3.2 and 5.3.3, we report the number of minimal contrast sequences returned by SATM for the three use cases and then provide an analysis of the mined sequences in order to see if they succeed in indicating a system activity correlated to the observed QoS issue. Finally, in Section 5.3.4, we discuss the influence of the *max_length*, *min_sup* and *max_sup* thresholds on both the quantity and the quality of the *MCS* set and then suggest a strategy to pick the values for these three thresholds so that to SATM returns the most concise and precise *MCS* set in minimal execution time.

5.3.1 GStreamer Use Case

Figure 5.11 presents the number of mined minimal contrast sequences with respect to the values of the *min_sup* threshold (for the D_{pos} dataset) and the *max_sup* threshold (for the D_{neg} dataset).

As we can see, even with the most extreme values of *min_sup* and *max_sup* (100% and 0% correspondingly) SATM mines 10 minimal contrast sequences. These

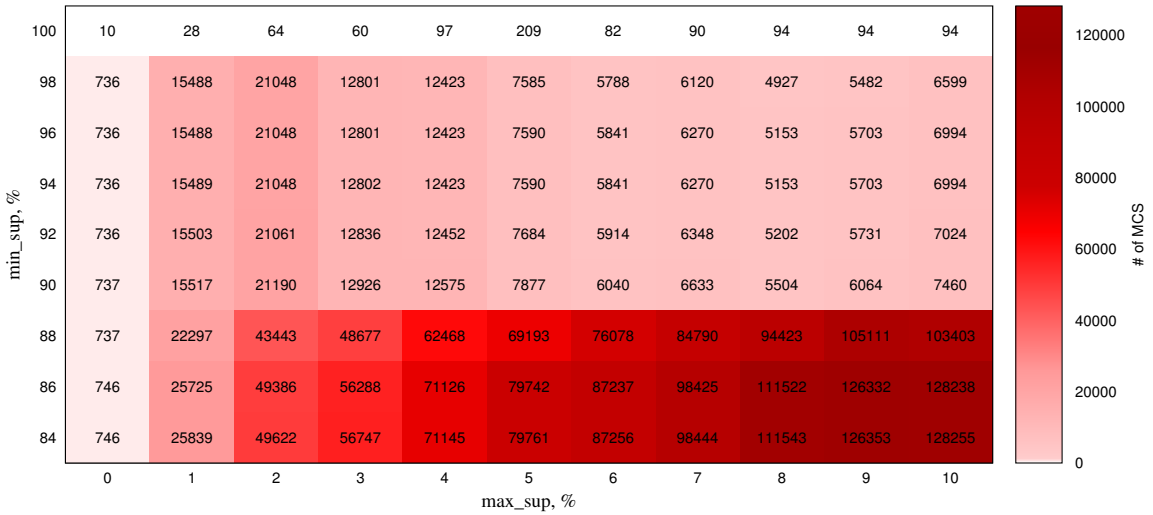


FIGURE 5.11: The number of minimal contrast sequences mined for the GStreamer use case (with $max\_length = 3$).

sequences are presented below (the lines in red correspond to the functions called by the *intruder* actor):

1. ⟨ TRACE;intruder;gstintruder.c;268;gst_intruder_chain;C0,
   LOG;pulse;pulsesink.c;660;gst_pulsering_stream_request_cb;autoaudiosink0 ⟩

2. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;268;gst_intruder_chain;C0,
   LOG;;pulse;pulsesink.c;1621;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

3. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;268;gst_intruder_chain;C0,
   LOG;;pulse;pulsesink.c;1568;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

4. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;268;gst_intruder_chain;C0,
   LOG;;pulse;pulsesink.c;1559;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

5. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;268;gst_intruder_chain;C0,
   LOG;;pulse;pulsesink.c;1550;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

6. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;263;gst_intruder_chain;B1,
   LOG;;pulse;pulsesink.c;1568;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

7. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;263;gst_intruder_chain;B1,
   LOG;;pulse;pulsesink.c;1559;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

8. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;263;gst_intruder_chain;B1,
   LOG;;pulse;pulsesink.c;1550;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

9. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
   TRACE;intruder;gstintruder.c;263;gst_intruder_chain;B1,
   LOG;;pulse;pulsesink.c;660;gst_pulseringbuffer_commit;autoaudiosink0 ⟩

10. ⟨ TRACE;intruder;gstintruder.c;252;gst_intruder_chain;A0,
    TRACE;intruder;gstintruder.c;263;gst_intruder_chain;B1,
    TRACE;;intruder;gstintruder.c;268;gst_intruder_chain;C0 ⟩

Notice that the 10^{th} minimal contrast sequence in the above list is exactly the sequence of function calls performed by the *intruder* actor that causes the delayed output of the application.

As the *min_sup* value goes down, the number of mined minimal contrast sequences increases, meaning that the developer needs to do more work in order to analyze the returned sequences of events. The target contrast sequence mentioned above [4], however, is always present in the mined *MCS* set. The same holds true when we increase the value of the *max_sup* parameter. This means that, with SATM, discovering the origin of the temporal bugs introduced into the GStreamer application is always possible regardless of the chosen values of the *min_sup* and *max_sup* parameters. At the same time, a drastic growth of the number of minimal contrast sequences observed when *min_sup* goes down can be explained with the following observation. All the events called by the *intruder* actor are never executed by other GStreamer actors. Given the fact that *intruder* is always invoked just before the *avdec_h264* actor (see the dataflow graph in Figure 5.2), which was used to split the original trace into the D_{pos} and D_{neg} datasets, the normal subtraces (i.e. from the D_{neg} dataset) contain *intruder*-specific events close to the end, right before the next occurrence of the *avdec_h264* actor which starts a new subtrace. This is not the case for subtraces found in the D_{pos} dataset, as an *intruder*'s invocation is not followed by the appearance of the *avdec_h264* actor but other, upstream actors continuing to execute normally. Therefore, SATM returns a big number of sequences of the form ⟨*intruder*-specific event; other events⟩ which never occur in D_{neg} due to the way the original trace is split into subtraces.

To sum up, the temporal debugging of the GStreamer use case with SATM consists in analyzing only 10 short sequences of raw system events, in case the thresholds' values are chosen correctly. In Section 5.3.4 we will suggest a strategy to quickly find such values.

[4] we will call a *target* sequence the one that characterizes the QoS problem in the best way among all the returned sequences

5.3.2 TSRecord Use Case

Consider Figure 5.12 which shows the number of minimal contrast sequences mined by SATM from the TSRecord's execution trace. As we can observe, there exist combinations of *min_sup* and *max_sup* values for which no contrast sequences are returned (see the white zone in Figure 5.12). This happens due to the wrongly-detected outlier intervals in the period detection step of SATM: some intervals are signaled as deviating too much from the period, while there is no detected anomalous activity in the corresponding parts of the trace, hence, white horizontal zones; and some intervals are signaled as normal ones, while the discovered abnormal sequence of events is present in the corresponding parts of the trace, hence, white vertical zones. The most restrictive pair of the *min_sup* and the *max_sup* thresholds for which SATM succeeds in mining contrast sequences is 94% and 1% correspondingly. In this case, we get 66 sequences of length 2 or 3. Each of these sequences contains at least one occurrence of `__switch_to45(usb-storage)` (to which we refer further as event X) or `Interrupt182(GICehci_hcd:usb3)` (event Y), and some sequences include both of them. The same is observed when *max_sup* is increased to 2% or 3%. However, when *max_sup* is set to a value in the interval $[4\%, 17\%]$ (and *min_sup* is kept at 94%), only 2 minimal contrast sequences are mined, both of them being length-1 sequences: $\langle X \rangle$ and $\langle Y \rangle$. The same picture is observed for all other combinations of *min_sup* and *max_sup* values: no matter how big is the returned *MCS* set, either it includes two length-1 sequences $\langle X \rangle$ and $\langle Y \rangle$, or every minimal contrast sequence contains an occurrence of $\langle X \rangle$ or $\langle Y \rangle$.

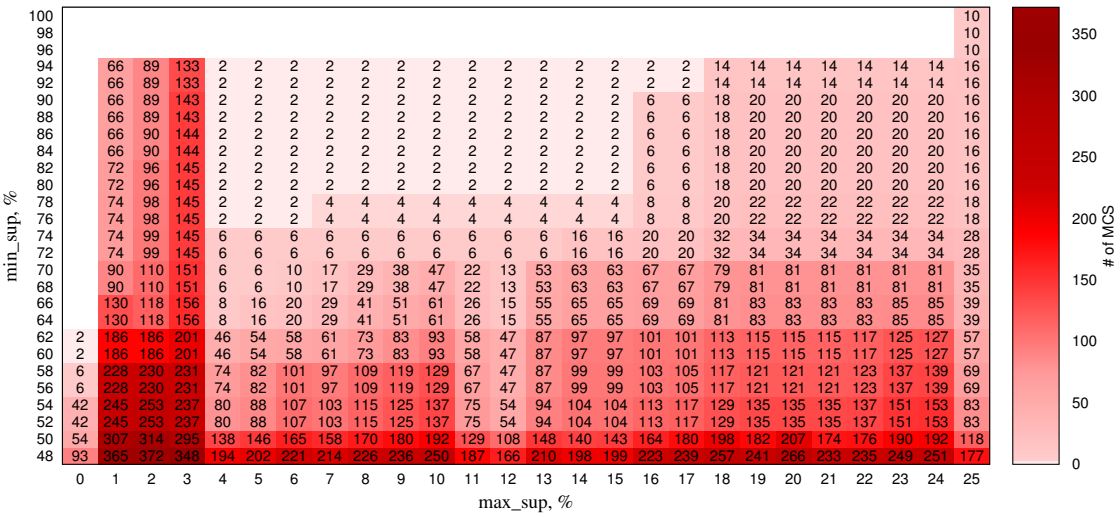


FIGURE 5.12: The number of minimal contrast sequences mined for the TSRecord use case (with $max\_length = 3$).

As we can infer from their names, both events X and Y are related to the USB port activity. Therefore, according to SATM, the cause of the observed QoS issue was somehow correlated to the transfer of data to the external USB hard drive. The fact that there is no USB port activity during long intervals of time (subtraces in

the D_{neg} dataset), was enough for the software developers at STMicroelectronics to understand the real cause of the degraded QoS. It was related to somewhat notorious behavior of the Linux page cache: the data is not written to the secondary storage immediately after calling the *write* system call, but is put into the part of main memory called page cache; it is then the responsibility of the *pdflush* kernel thread, invoked by default by the operating system every 5 seconds, to initiate the writing of the data stored in the page cache to the secondary storage. The peculiarity of *pdflush* operation is the complete blocking of those pages from the page cache that contain data to be sent to the secondary storage. Therefore, if TSRecord tries to write the newly received data from network buffers to one of such pages, it becomes blocked until *pdflush* finishes its operation. The amount of memory required to store 5 seconds of a high-definition video can easily exceed the size of network buffers. Therefore, some network packets are lost due to the network buffers overflow, hence, some video frames are dropped in the recorded stream. A possible fix of this problem is to reduce the *pdflush* invocation interval [5]. Note that the *pdflush* event itself was not contained in minimal contrast sequences mined by SATM. This is due to the fact that this event often occurs in the non-anomalous parts of the trace, before *write* becomes blocked. However, *pdflush* triggers numerous calls to X = `__switch_to45(usb-storage)` and Y = `Interrupt182(GICehci_hcd:usb3)`, and this knowledge helped the developers to isolate *pdflush* as the real cause of low QoS of the TSRecord application.

5.3.3 DVBTest Use Case

Similar to the previous two use cases, we use a heat map depicted in Figure 5.13 to show the number of minimal contrast sequences mined by SATM from the DVBTest's execution trace. As with the TSrecord use case, the most restrictive pair of thresholds which results in a non-empty MCS set is not $min\_sup = 100\%$, $max\_sup = 0\%$ but $min\_sup = 84\%$, $max\_sup = 1\%$. In the latter case, SATM mined 57 contrast sequences, each of them being a length-3 sequence that contains a length-2 subsequence composed of two identical events `CPureSwQueueBufferInterface::SelectPictureForNextVSync:Keep_same_node_on_display`, to which we refer further on as event X. When max_sup is increased to be equal to 2%, only one minimal contrast sequence is returned: $\langle X, X \rangle$. If max_sup is increased further, $\langle X, X \rangle$ is always present in the returned set, but other sequences having a single occurrence of X start being output.

A contrast sequence $\langle X, X \rangle$ indicates that the temporal bugs have something to do with the retainment of the frame in the queue buffer. As was explained in Section 5.2.3, an already decoded video frame can be retained in a queue buffer for an additional display refresh cycle if this is required for vertical synchronization.

[5] www.westnet.com/~gsmith/content/linux-pdflush.htm

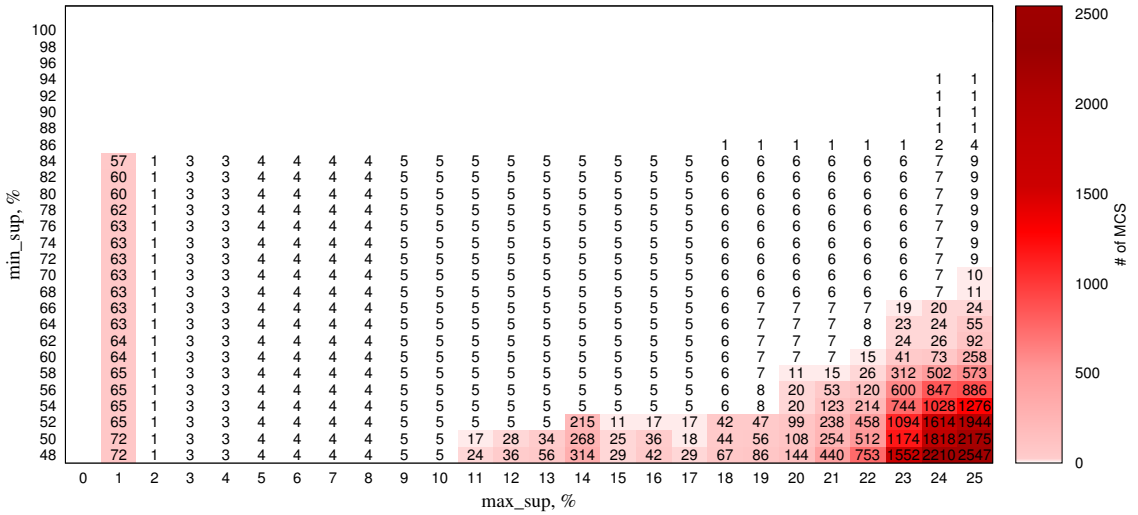


FIGURE 5.13: The number of minimal contrast sequences mined for the DVBTest use case (with $max\_length = 3$).

The double appearance of X, however, signifies that the frame is retained for two additional refresh cycles. In fact, if we look at subtraces in the D_{pos} dataset, we can notice that event X often appears more than two times in the same subtrace. Such prolonged retentions of a single frame in the queue buffer leads to a situation where the new frames cannot be placed into the buffer, hence, no frames are sent to the display during several refresh cycles, and a video glitch is observed. At the same time, during normal DVBTest execution (subtraces in the D_{neg} dataset) a single frame is either not placed in the queue buffer at all, or is retained for only 1 additional cycle (i.e. X does not appear in the subtrace or appears only once). That is why, a length-2 sequence $\langle X, X \rangle$ was signaled as a minimal contrast one. Once informed about this contrast sequence, the developers in STMicroelectronics quickly realized that there was a bug in the algorithm that computes the number of cycles a given frame needs to stay in the queue buffer.

5.3.4 Discussion

As we have seen in the previous three sections, if the *MCS* set returned by SATM for a given configuration of *min_sup* and *max_sup* threshold values is not empty, then the target minimal contrast sequence can be found in it. We consider that using SATM with $min\_sup < 50\%$ and $max\_sup > 25\%$ is irrelevant, hence, have not tested such pairs of parameter values. Indeed, as explained in Section 4.1, a pattern can be called a contrast one if it appears frequently in the D_{pos} dataset and, at the same time, rarely in the D_{neg} dataset. That said, we are still left with the problem of choosing the values for *max_length*, *min_sup* and *max_sup* parameters, so that both the effort to uncover the target minimal contrast sequence from the list of all the returned sequences as well as the running time of the mining

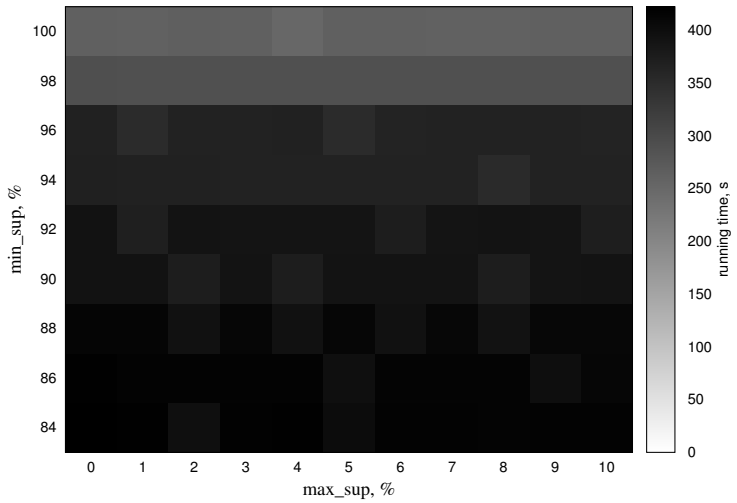
algorithm are minimized. This section's goal is to provide a justified answer to this problem.

Consider Figure 5.14 which shows the running times of SATM with respect to the values of *min_sup* and *max_sup* parameters (*max_length* was still fixed at 3) for all three use cases. Taking also into account Figures 5.11, 5.12 and 5.13, we can notice that both the size of the returned *MCS* set and the algorithm's running time go up as the *min_sup*'s value decreases. This observation can be explained by the way the mining algorithm obtains the *MCS* set from all possible sequences. Recall that the adopted strategy consists in extracting minimal contrast sequences from the set of frequent sequences in the D_{pos} dataset (see Section 4.2.2). Therefore, the lower the *min_sup* value, the bigger the number of candidate sequential patterns, hence, the running time spent verifying their supports in the D_{neg} dataset, and also the bigger the number of minimal contrast sequences.

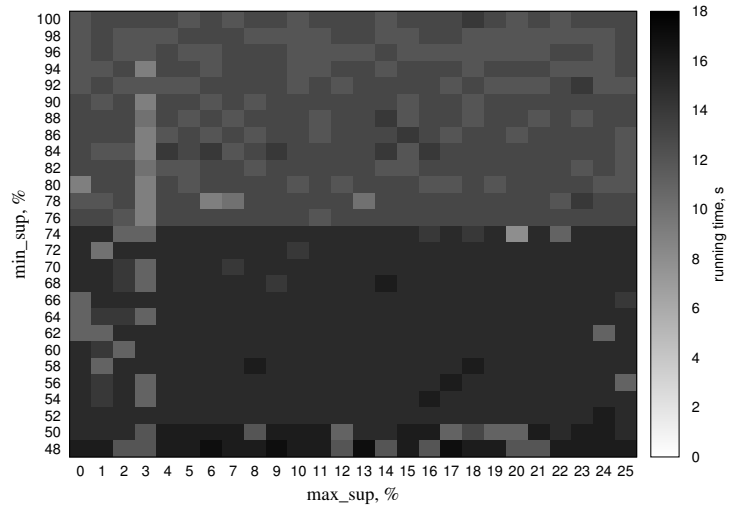
The influence of the *max_sup* threshold's value on the size of the returned *MCS* set as well as on the algorithm's running time is less obvious than of the *min_sup* threshold. It is easy to see that a bigger value of *max_sup* results in more contrast sequences being mined. However, one cannot predict the number of minimal contrast sequences with respect to the value of *max_sup*. In order to understand why, consider the following example datasets: $D_{pos} = \{\langle x, a, b\rangle, \langle x, a, b\rangle, \langle x, a, b\rangle\}$, $D_{neg} = \{\langle a, b\rangle, \langle x, a, b\rangle, \langle a, b, x\rangle, \langle a, b\rangle\}$. In this case, if both *max_length* and *min_sup* are assigned to value 3 while $max\_sup = 0$, then the returned *MCS* set is empty. If the *max_sup*'s value is increased to 1, then two minimal contrast sequences are returned: $\langle x, a\rangle$ and $\langle x, b\rangle$. However, if the *max_sup*'s value is increased again to be equal to 2, there is only one minimal contrast sequence in the given datasets: $\langle x\rangle$.

As *max_sup* is increased, which makes the mining algorithm return more contrast sequences, one expects the running time of the mining algorithm to decrease, as contrast pruning removes more patterns from the search space. However, as we can see in Figure 5.14, there is no noticeable decrease in running time for bigger values of *max_sup*. This can be explained with the small value of the *max_length* parameter. Indeed, as we fixed the *max_length* threshold to 3, the depth of the search space tree is limited to 3 levels, making contrast pruning much less effective than if the *max_length* threshold was not used. At the same time, contrast pruning helps to reduce the running time spent in the extraction of minimal contrast sequences from all mined contrast sequences, performed as a post-processing step.

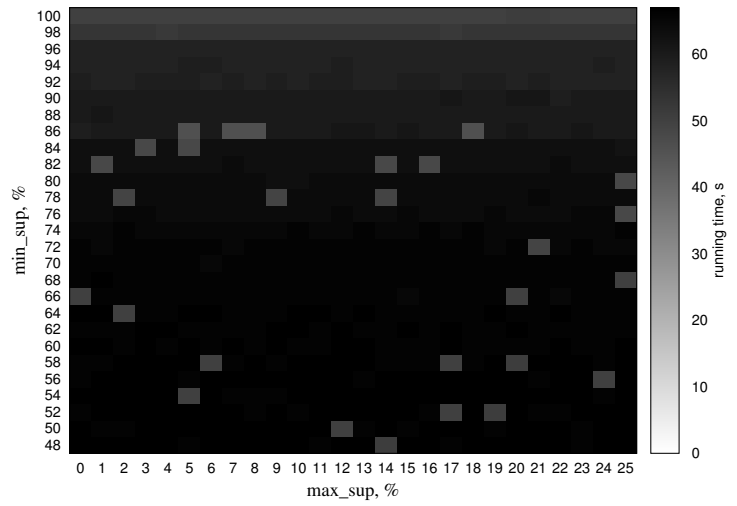
Finally, the influence of the *max_length* threshold on the mining algorithm's running time was already presented in Section 4.2.3: the algorithm's computational complexity grows exponentially with the increase of the *max_length* threshold's value.



(A) GStreamer use case



(B) TSrecord use case



(C) DVBTest use case

FIGURE 5.14: Performance results for the SATM mining algorithm run on the three use cases

Having discussed the influence of *min_sup*, *max_sup* and *max_length* parameters on the computational complexity of the mining algorithm, as well as on the effort to analyze the returned *MCS* set, we now present a strategy to choose the values for these three thresholds. The value for the *max_length* parameter must be chosen first. We propose to gradually increase the value of *max_length* until the performance of the mining algorithm run on the target D_{pos} and D_{neg} datasets becomes prohibitively big; *min_sup* and *max_sup* parameters in their turn remain fixed at 100% and 0% correspondingly. The maximal acceptable value of *max_length* will depend on the characteristics of the datasets as well as on the available computational resources. Once such value of *max_length* is found, one needs to find the most constrained values of *min_sup* and *max_sup* thresholds for which the returned *MCS* set is not empty. In other words, the top left corner of the non-white rectangle of the heat map must be experimentally found (e.g. 94% and 1% in Figure 5.12, 84% and 1% in Figure 5.13, or 100% and 0% in Figure 5.11). Note that it does not make much sense to test all the *min_sup*'s values from 100% down to 1% with *max_sup* being fixed at 0%, before running the mining algorithm with bigger values of *max_sup*. Similarly, it is unjustified to run SATM with all possible *max_sup*'s values while *min_sup* is fixed at 100%, without trying first to lower down the *min_sup*'s value. Once a combination of *min_sup* and *max_sup* values resulting in a non-empty *MCS* set is found, the returned minimal contrast sequences must be analyzed by a software developer. If the number of returned sequences is too big, a good solution could be to try to increase the value of *max_sup* parameter, hoping that noisy contrast sequences disappear, leaving a smaller number of more precise sequences.

5.4 Conclusion

In this chapter, we evaluated our proposed temporal debugging approach SATM on three use cases. The first use case consisted of a GStreamer-based media decoding application whose execution was deliberately perturbed by ourselves. The second and the third use cases came from STMicroelectronics and contained industrial embedded streaming applications exhibiting low QoS.

As we showed in Section 5.2, SATM successfully detected anomalous zones in the provided execution traces of each use case, in a fully automatic manner. The clustering step in its turn proved to be extremely important for the period detection of the applications' actors. In Section 5.3.1, we presented the results of mining suspicious system activity in the detected anomalous zones using SATM, and also explained how the reported minimal contrast sequences allowed to understand the origin of the temporal bugs. Finally, we proposed a strategy to choose the values for three parameters used by SATM to mine the set of minimal contrast sequences, so that the relevant system activity is detected as quickly as possible.

All in all, SATM proved to fulfill its task of automatic detection of system activity related to the origins of the QoS problems in real-world embedded streaming applications.

Chapter 6

Related Work

Temporal and functional debugging of embedded software are two conceptually different problems which need to be addressed with different tools and techniques. Software developers are well trained in functional debugging, as the appropriate tools have been available for them for decades. Indeed, learning to set breakpoints in the application's source code and inspect the system state using GDB-like debuggers has always been an important part of any programming course. Unfortunately, as we explained in Section 2.6, stopping system execution is not an option when the temporal behavior of the entire system must be analyzed.

In this chapter, we examine how temporal debugging of embedded streaming applications is performed in the absence of generally accepted solutions. In Section 6.1, we review the techniques to perform temporal debugging in case system execution traces are not available to software developers. As we explained in Section 2.7, until relatively recently execution tracing was an expensive and, more importantly, a highly intrusive process, which could easily perturb temporal behavior of the entire system. In Section 6.2, we take a look at more recent temporal debugging techniques which have emerged after the advances in both embedded software and hardware made the process of execution tracing easy and non-intrusive.

6.1 Temporal Debugging Without Execution Traces

The literature dedicated to temporal debugging of streaming applications without using execution traces is extremely scarce. The proposed debugging approaches make use of interactive debuggers but try to alleviate their intrusiveness with rather exotic techniques. Some authors suggested to perform temporal debugging of real-time applications using execution time prediction [56]. This way, an application is monitored with a conventional interactive debugger using virtual

time predicted by a cache simulator, so that the debugger is not aware of the real wall-clock time altered by its presence. Other authors [9] [10] proposed using a complete system simulation which provides a timing model for streaming applications, so that the system's temporal behavior is not altered by the attached interactive debugger.

Given the lack of software solutions specifically tailored to temporal debugging, programmers relied on performance profiling tools in order to get some insight into the causes of violations of temporal constraints in their embedded applications. A profiler gathers various performance metrics (e.g. time and memory complexity, frequency of specific function calls, etc.) during a program's execution and provides a summary of *hotspots*, i.e. the most prominent sections of code with respect to the chosen metric(s). The goal of profiling tools is to assist a programmer in deciding which parts of the application's code require optimizations, so that the performance of the whole system can be improved.

There exist several types of profiling [1]:

- Software based profiling requires injecting instrumentation code into the program's source code before or during compilation. At runtime, the profiler counts the number of times each instrumented block of code is executed and reports the gathered statistics once the program terminates [34]. The problem with software based profiling is that the injected instrumentation code changes the program, therefore, alters the temporal behavior of the entire embedded system [65].
- Hardware based profiling [11] uses on-chip performance counters in order to monitor specific events occurring during program's execution: CPU cycles, cache misses, memory reads/writes, particular instruction executions, and many others. The number of events that can be tracked simultaneously, however, is limited by the number of performance counters available on the target hardware. For example, the ARM Cortex-A9 CPU [1] found in STMicroelectronics' STiH412 MPSoC (see Figure 2.2) has only 6 event counters [2]. The role of the profiling tool is then to collect the values from the performance counters for the particular application components and provide a summary of the hotspot areas of code. In contrast to software based profiling, hardware based profiling introduces only a marginal performance overhead. Indeed, the only thing a profiler needs to do at runtime is to read from time to time the values stored in the hardware performance counters [21], instead of interrupting the program's execution every time a particular event occurs.

[1] We do not discuss simulation based profiling in this thesis, as it is much slower and less precise than software/hardware based profiling and has a very limited usage for debugging software on complex SoCs that have been already put into production

[2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4237.html

There are two ways hardware based profiling can be performed: either in a counting, or in a sampling mode [54]. With the counting mode, the hardware counter is increased each time a given event occurs. The developer, thus, only gets the total number of a particular event's executions as the result of profiling. From the other hand, the sampling mode [12] allows to get information on the precise code segment that caused each event. This is possible due to the support of hardware interrupts on a counter overflow. When the number in a counter reaches a predefined value, hardware interrupt is generated, and information about the part of code that caused the last event is collected.

Although profilers are actively used nowadays for performance optimization, they are far from being sufficient to resolve temporal bugs. Indeed, profiling tools report aggregate values for performance metrics without providing any details on the cause of the observed values, and more importantly on their relation to temporal bugs. This results in a trial-and-error temporal debugging, where software developers optimize the performance of the reported hotspots hoping that such optimizations will eliminate temporal anomalies.

To sum up, temporal debugging was an extremely tedious and approximate process before tracing technology became mainstream in embedded systems community and favored the emergence of more advanced temporal debugging solutions.

6.2 Temporal Debugging With Execution Traces

Thanks to an active participation of both the suppliers of IP cores and the kernel space software developers, it is possible nowadays to trace execution of the whole embedded system in a convenient and non-intrusive way (see Section 2.7 for more details). Therefore, embedded software developers now have a way to inspect system behavior once the execution has finished, i.e. in a post-mortem fashion. This is particularly good news with respect to the temporal debugging problem, as the whole history of system execution can be navigated back and forth until suspicious activity causing temporal bugs is detected.

Anyone who has seen a real execution trace of an embedded system knew immediately that trace analysis is not a simple task. A very long timestamped sequence of events coming from different parts of a system, which is essentially an execution trace, is not the most analysis-friendly type of data. Fortunately, there exists a plethora of trace visualization tools which can make the process of trace analysis much more productive and pleasant.

Trace visualization

A common approach to perform trace analysis consists in searching for a particular part of the trace with **grep**-like tools and then manually inspecting system activity in that part using trace summarization and visualization tools [69]. Many different techniques have been proposed to visualize execution traces [37][71][28], but the most popular one consists in displaying the timeline of system execution with a Gantt chart [24][36][59]. Consider Figure 6.1 which shows a 1 millisecond slice of an execution trace displayed with STLinux Trace Viewer – a trace visualization tool based on Gantt chart representation developed in STMicroelectronics. The visualization includes a list of threads, as can be seen on the left side of the figure, and a timeline view of the events executed on the processor in the context of these threads. Black arrows denote context switches from one thread to another, while the green arrows show the interrupts executed in the context of the active thread. The length of the rectangles in the timeline view corresponds to the duration of a particular thread invocation, and the colors denote execution of different functions (the white color means that the executed function was not instrumented).

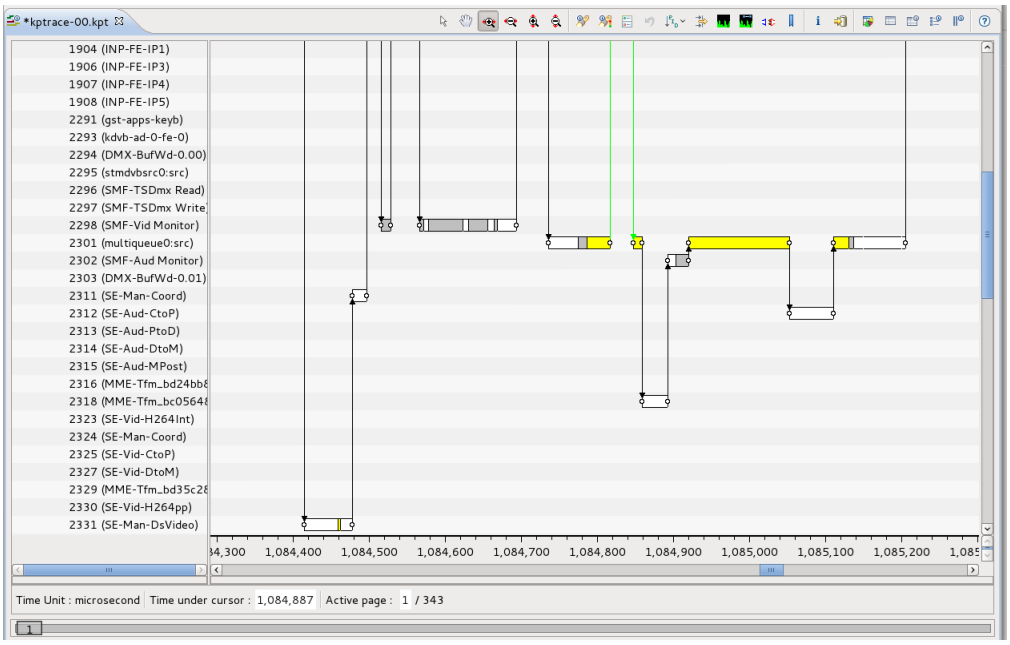


FIGURE 6.1: A screenshot of STLinux Trace Viewer showing the visualization of a 1 millisecond slice of an execution trace.

Trace visualization tools can be very helpful for a fine-grained analysis of a very specific part of a trace. In case temporal debugging must be performed, however, the developer normally does not know which segment of a trace contains faulty system behavior. Therefore, the totality of the execution trace must be analyzed. Consider Figure 6.2 which shows the visualization of a 1 second slice of the same trace presented in Figure 6.1. Note that this trace slice represents only a $1/343^{th}$ of the original trace file. Moreover, it shows the activity of a single CPU core.

Even if the faulty system behavior was present in this part of the trace, it would be still very difficult to spot it manually.

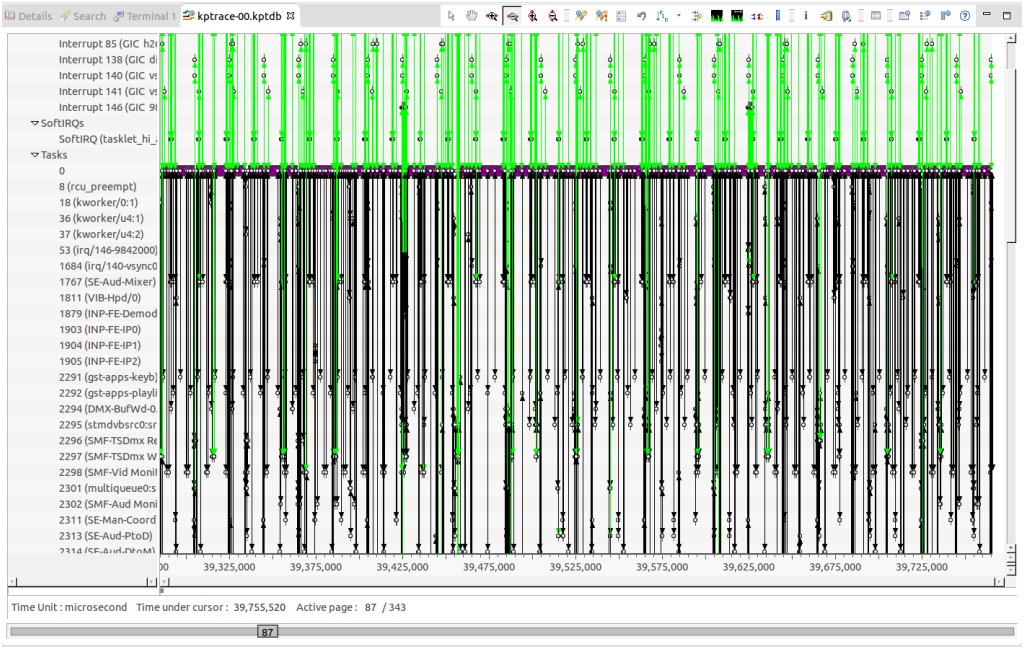


FIGURE 6.2: A screenshot of STLinux Trace Viewer showing the visualization of a 1 second slice of an execution trace.

Manual visual analysis of execution traces does not scale and is usually very hard when performed with the goal of system-level temporal debugging. Therefore, the current trend is to automate the process of trace based temporal debugging to relieve software developers from the burden of poking around large amounts of information [83].

Data mining in execution traces

The general goal of data mining field is automatic knowledge discovery from big amounts of data (see Section 4.1 for more details). As data mining algorithms are usually oblivious of the provenance of the provided data and, hence, have been successfully applied in various domains, it is a logical idea to use such algorithms on execution traces in order to get some insights into the origins of temporal bugs. In this section, we compare the most relevant data mining approaches for temporal debugging of embedded streaming applications to our Streaming Application Trace Miner (SATM) .

Lo et al. [49] address the same data mining problem as we do: discovering contrast patterns from program execution traces for debugging purposes. Different from our approach, the authors propose to mine contrast closed iterative patterns.

An iterative pattern [50] is essentially a sequential pattern. At the same time, the support of an iterative pattern p in a sequential database DB includes not

only the number of transactions T, such that $T \sqsupset p$, but also the number of non-overlapping instances (or, iterations) of p in each transaction T. For example, given $DB = \{T_1 : \langle A, B, A, C, B\rangle, T_2 : \langle C, A, B\rangle\}$, the support of an iterative pattern $p = \langle A, B\rangle$ in DB is $sup_{DB}(p) = 3$ and not 2, as p has two iterations in T_1. An algorithm to mine frequent closed iterative patterns was proposed in [50].

In the Experimental section of their work, Lo et al. [49] report that they were not able to mine the full set of contrast closed iterative patterns from program execution traces due to the high computational complexity of the task. Instead, the authors mine such called contrast closed *unique* iterative patterns, that is, contrast closed iterative patterns where each distinct element can occur only once. Such patterns are obviously too restrictive and would not allow to express contrast sequences containing several identical elements, such as the contrast sequence reported for the DVBTest use case in Section 5.3.3.

The work of Yu et al. [87] presents an approach to debug applications' performance problems which is conceptually similar to SATM. The authors introduce the notion of *cost propagation* in the systems consisting of interacting components, akin to our delay propagation, in order to discover a component which is responsible for the performance bug. Their approach mines a number of execution traces capturing both "good" and "bad" application performances to find event patterns that are inherent to the "bad" traces and that originate from a set of predefined suspicious components. Similarly, Hsu et al. [38] mine contrast patterns from two sets of execution traces using a suspiciousness metric computed for individual traced events.

In contrast to [87] [38], SATM does not require any assumptions on the location of the performance bug or on the type of events related to the bug. Instead, we address a specific context, which is embedded streaming software, by including its specificities (periodic execution, real-time deadlines) in the core of SATM.

The work of Lagraa et al. [44] applied data mining algorithms on execution traces of streaming applications run on simulated MPSoC based platforms in order to resolve memory contention bugs. Their approach detects individual instructions that exhibit abnormally high latency and mines patterns of execution events that frequently occur in the vicinity of these instructions. At the same time, if the system's low QoS does not originate from memory contention, memory access latency becomes useless to locate an anomalous part of the execution trace.

Cueva et al. [51] propose to resolve temporal bugs in embedded streaming applications using periodic patterns mined from execution traces. This was the first work that took into account periodic nature of streaming applications in order to mine more meaningful patterns from execution traces.

A periodic pattern is an itemset (i.e. a set of events) that appears periodically in a system execution trace and whose elements occur temporally close to each other.

In order to define an exact threshold value for "temporally close", the authors propose to split the execution trace into subtraces using either a time interval or a particular event, so that events occurring in the same subtrace are considered temporally close to each other. The period of a pattern is then defined as the number of subtraces that separate two consecutive pattern's occurrences. The authors proposed the PerMiner algorithm to mine all frequent periodic patterns in the execution trace. Once a complete set of frequent periodic patterns is mined, the developer has to manually analyze it and select the patterns having gaps in their period.

The work of Cueva et al. [51] made a valuable contribution to data mining field by introducing an efficient way of mining periodic patterns. There are, however, several limitations to use periodic patterns for temporal debugging purposes. Firstly, the periods of all the periodic patterns must align with the way the execution trace was split into subtraces. For example, if a time interval of 10ms was used to split an execution trace, and a pattern has a period of 7ms or 23ms, then such pattern will not be mined by PerMiner. Secondly, the set of mined periodic patterns must be analyzed in its entirety in order to spot unusual patterns or the ones having gaps in their period. Such manual verification can be difficult if a big number of periodic patterns was mined.

Kengne et al. [42] propose to discover anomalies in streaming applications by applying distance metrics on normal and buggy execution traces. The idea of their approach consists in enabling software developers to automatically detect a high-level problem present in a given execution trace, under condition that there exists a distance metric specific to the observed anomaly. At the same time, no information is given to the developer on the cause of the observed anomaly. Moreover, both the list of metrics characterizing possible anomalies as well as the trace corresponding to a normal application's execution are required as the input from the developer.

Chapter 7

Conclusion

The process of temporal debugging of multimedia embedded systems often feels like looking for a needle in a haystack. Searching for a tiny, shiny object in a big pile of dried grass is a daunting task; but at least you know what you are looking for. From the other hand, given a sequence of tens of thousands of events present in a typical execution trace of an embedded system, software developers usually have no idea which particular set of events in that trace would provide them an insight into the origins of the system's low QoS. Software developers, however, should not be blamed for the lack of understanding of the fine-grained behavior of the whole embedded system captured in an execution trace. The problem is rather with the current lack of appropriate tools and techniques to perform execution trace analysis for temporal debugging purposes.

Next, consider consumer electronics devices that enclose multimedia embedded systems in their sleek bodies: mobile phones, tablets, set-top boxes, and many others. The market of such devices is extremely competitive and dynamic: dozens of companies (among them are such giants as Apple, Samsung, LG, etc.) struggle to propose the best product among the competitors, and launch it on the market as soon as possible. At the same time, even the most stylish, powerful device with small energy consumption and an affordable price tag will not stand up to competition if its software contains temporal bugs. Indeed, customers simply do not buy multimedia devices exhibiting low QoS (which is often an indicator of the presence of temporal bugs), as it greatly reduces user experience, that is, makes the device not pleasant to use. Moreover, temporal bugs tend to appear at the last stages of the software development process, when the device has already left the manufacturing facility, but some use cases of running various applications cause the device to exhibit low QoS.

As a result, if one couples the urgency of temporal debugging to the lack of appropriate tools to perform this task, he/she can easily empathize with embedded software developers in their exasperation of dealing with this type of bugs.

We believe that data mining research and embedded software debugging has a potential to be a new fruitful meeting point between computer science and software engineering. With the recent advances in the execution tracing technology, it became possible to unintrusively capture the whole embedded system's activity for later analysis. At the same time, data mining subfield of computer science has showed its universality for automatic discovery of useful information from big amounts of data in diverse areas: from the traffic control to web-based recommendation systems.

7.1 Contributions

In this thesis, we have proposed to automate the search of suspicious system activity in execution traces of multimedia embedded systems exhibiting low QoS using data mining algorithms. The following list summarizes the contributions of this work:

I. A data mining approach to temporal debugging of embedded streaming applications.
The aim of this doctoral work was to propose a debugging tool that, given a precise input, would return a specific and concise description of system activity related to the origin of the system's low QoS. We achieved this goal with Streaming Application Trace Miner (SATM) – our data mining approach to temporal debugging of embedded streaming applications. SATM takes an execution trace of the embedded system as well as the list of dataflow actors of the application showing low QoS and returns a set of execution event sequences which represent suspicious system activities related to the origin of temporal bugs. The operation of SATM is divided into two distinct stages.

During the first stage (Section 3.2), SATM narrows down the search area in the execution trace. To achieve this in a completely automatic fashion, SATM relies on the notion of execution delay propagation (our third contribution). Implementation-wise, SATM uses the one-dimensional clustering algorithm proposed in [22] which we enriched with Otsu algorithm [60] to eliminate any parameter setting required from the developer. In order to find the outlier clusters, we applied a simple outlier detection technique used in boxplot diagrams [77].

During the second stage (Section 4.2), SATM mines minimal contrast sequences between the anomalous parts of the execution trace detected in the first stage

and the rest of the trace. To make this stage computationally feasible, we introduced the *max_length* parameter to the PrefixSpan algorithm [63] which we further enriched with the contrast pruning from [41].

In Chapter 5, we validated SATM on three real-world use cases. The first use case involved a media application programmed with the GStreamer framework where we manually injected temporal bugs. The other two were industrial use cases coming from STMicroelectronics. In all three cases, SATM was able to pinpoint a concise sequence of events which directly pointed at the cause of the temporal bugs.

II. Analysis of the computational complexity of state-of-the-art pattern mining algorithms applied on execution trace data.
While investigating numerous pattern mining algorithms in order to find the one which would allow to mine minimal contrast sequences from execution traces (second stage of SATM), we were encountering the same problem, again and again: the algorithms would not show the promised in the literature performance, even on the smallest, synthetic execution traces. After months of fruitless attempts, we started to look closer at our data and realized that the problem comes from the intrinsic characteristics of trace data, rather than its sheer size. More precisely, high computational complexity of pattern mining algorithms run on trace data comes from two facts: (1) embedded streaming applications exhibit highly repetitive behavior, hence, their traces share a long common sequence of events, and (2) there are a lot of random events between the elements of the aforementioned sequence due to the shared computational resources of MPSoC platforms. We identified this as an interesting new research problem in the pattern mining field which merits being investigated further.

III. The notion of execution delay propagation in dataflow graphs.
We studied the common properties of streaming applications run on embedded systems in order to find an automatic way of detecting abnormal parts of execution traces where the system first starts to deviate from its normal behavior. After an extensive analysis of both the dataflow programming model used to design streaming applications, as well as the real-time environment of multimedia embedded systems, we have come up with a notion of execution delay propagation. As we explained in Section 3.1, according to this notion, once a particular actor from the application's dataflow graph misses its real-time deadline, the difference between the expected and the observed times of sending its output to the subsequent actor propagates through the downstream actors resulting in delayed output of the whole application. This understanding allowed us to make the first step towards tracking down the elusive temporal bugs in embedded streaming applications.

Our first and third contributions have been published in [40].

7.2 Limitations

Although SATM is based on a solid theoretical foundation and showed its utility for temporal debugging of real streaming applications, we can identify several limitations of our approach:

1. SATM is tailored to solve a particular type of problems dictated by the industrial nature of this work.
The goal of this doctoral work was to propose a temporal debugging approach for the software developed in STMicroelectronics. Therefore, SATM incorporates some properties specific to embedded streaming applications (dataflow programming model, strictly periodic execution, delay propagation), hence, in situations where temporal bugs must be found in the software that does not have these properties, SATM cannot be applied.

2. Usage of the max_length parameter in minimal contrast sequence mining step limits the applicability of SATM.
Indeed, in cases where the most concise representation of abnormal system activity consists of more than *max_length* events, SATM will not be able to discover such system activity. As explained in Chapter 4, our research showed that the task of mining the complete set of minimal contrast sequences from execution trace data is prohibitively expensive using state-of-the-art pattern mining algorithms. By introducing the *max_length* parameter we tried to find a balance between the time required to perform the mining step and the expressiveness of the returned patterns.

7.3 Perspectives

From our experience of performing research in the industrial context we have observed yet another example of how both computer science and software engineering can benefit from each other. Software debugging can become a more methodical process if more research is done on the application of data mining algorithms on system traces. At the same time, data mining subfield of computer science can be enriched with conceptually new algorithms and techniques aiming at knowledge extraction from the not yet thoroughly investigated data type which is system traces. In this thesis, we have done a step forward in both of these directions. We envision to continue our work in the following directions:

1. Automate the choice of values for min_sup, max_sup and max_length parameters during minimal contrast sequence mining.
Even though we advertised SATM in this thesis as an automatic approach to temporal debugging, it is still not *completely* automatic. Minimal contrast sequence mining algorithm used in the second stage of SATM requires the setting of values

for *min_sup*, *max_sup* and *max_length* parameters. In Section 5.3.4, we have proposed a strategy to manually find the best values for these parameters. It would be beneficial to automate this search, so that the developer only analyzes the returned contrast sequences and chooses to continue the mining process, or to stop it if the cause of the system's low QoS has been identified.

2. Apply a more advanced outlier detection method for more accurate identification of anomalous parts of the execution trace.
As we have seen in Section 5.3, in practice, the first stage of SATM is prone to returning some false positive and false negative anomalous parts of the execution trace. The problem comes from the simplicity of SATM's outlier detection mechanism. Indeed, Tukey's intuition [77] to use 3 inter-quartile ranges for outlier identification works well for quick analysis of normally distributed data using box plot diagrams. At the same time, it may fail to correctly separate regular observations from outliers when the data is skewed or not normally distributed. A more advanced outlier detection method [75] could alleviate this problem. Moreover, reducing the number of false positives and negatives will make the process of choosing the values for *min_sup* and *max_sup* parameters in the contrast mining algorithm more intuitive.

3. Investigate the further applications of data mining algorithms on trace data for software debugging purposes.
During this thesis, we have briefly collaborated with the AMfoRS team [1] from the TIMA laboratory and the Convecs team [2] from the LIG laboratory. Both collaborations targeted an application of pattern mining algorithms on simulation traces produced by model checking software. While working with the AMfoRS team, we co-supervised the internship of a first year Master's student and helped him to apply a frequent closed pattern mining algorithm [78] to characterize simulation traces of program executions which violated system's temporal properties [3]. We succeeded to detect the system's activities which characterized the causes of temporal properties violations, but the size of simulation traces was rather small (a single trace had ∼15-25 events). The second, more recent collaboration is also aimed at debugging simulation traces produced by model checking software. However, a considerable size of collected simulation traces caused the running of state-of-the-art pattern mining algorithms on them too costly.

These two collaborations have strengthened our belief that data mining algorithms is an invaluable technique for software debugging process. It is, therefore, very important to continue the investigation of possible areas of application of data mining algorithms in software engineering. Indeed, a PhD thesis of Gianluca Barbon started in October 2015 [4] aims to continue our collaboration with the

[1] http://tima.imag.fr/tima/en/amfors/amforsoverview.html
[2] http://convecs.inria.fr/
[3] For more information on the way such traces were generated, see [66].
[4] http://www.theses.fr/en/s141898

Convecs team in searching for the ways data mining algorithms could help in debugging counterexamples generated by model checking software.

4. Continue the search of novel pattern mining algorithms for execution trace data. As we concluded in Chapter 4, the state-of-the-art pattern mining algorithms fail to efficiently mine execution traces due to structural peculiarities of this type of data. It is, therefore, important to continue the research on novel algorithms which would make pattern mining on execution trace data feasible. With this goal in mind, we have started a collaboration with one of the authors of [57] to verify if their sequence mining algorithm based on constraint programming can make the task of discovering minimal contrast sequences more efficient.

Bibliography

[1] ARM Cortex-A9 CPU. http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[2] ARM Mali-400 GPU. http://www.arm.com/products/multimedia/mali-gpu/ultra-low-power/mali-400.php.

[3] KPTrace for STLinux. http://www.stlinux.com/devel/traceprofile/kptrace.

[4] LTTng tracing framework for linux. https://lttng.org/.

[5] STLinux distribution and development environment. http://www.stlinux.com.

[6] Charu Aggarwal and Jiawei Han, editors. *Frequent Pattern Mining*. Springer, 2014.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 3–14. IEEE, 1995.

[8] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[9] Lars Albertsson. Temporal debugging and profiling of multimedia applications. In *Electronic Imaging 2002*, pages 196–207. International Society for Optics and Photonics, 2001.

[10] Lars Albertsson and Peter S Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 191–198. IEEE, 2000.

[11] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[12] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, 1997.

[13] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 195–204. ACM, 2011.

[14] Mohamed Bamakhrama and Todor Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 83–92. ACM, 2012.

[15] Stephen D Bay and Michael J Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 302–306. ACM, 1999.

[16] Roberto J Bayardo Jr. Efficiently mining long patterns from databases. In *ACM Sigmod Record*, volume 27, pages 85–93. ACM, 1998.

[17] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[18] Alvis Brazma, Inge Jonassen, Ingvar Eidhammer, and David Gilbert. Approaches to the automatic discovery of patterns in biosequences. *Journal of computational biology*, 5(2):279–305, 1998.

[19] Reinder J Bril, Christian Hentschel, Elisabeth FM Steffens, Maria Gabrani, G van Loo, and Jean HA Gelissen. Multimedia qos in consumer terminals. In *Signal Processing Systems, 2001 IEEE Workshop on*, pages 332–343. IEEE, 2001.

[20] Sarah Chan, Ben Kao, Chi Lap Yip, and Michael Tang. Mining emerging substrings. In *Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings. Eighth International Conference on*, pages 119–126. IEEE, 2003.

[21] Thomas M Conte, Burzin A Patel, Kishore N Menezes, and J Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. In *Int'l Journal of Parallel Programming*. Citeseer, 1996.

[22] Matthew Cooper, Jonathan Foote, Andreas Girgensohn, and Lynn Wilcox. Temporal event clustering for digital photo collections. *ACM Transactions*

on Multimedia Computing, Communications, and Applications (TOMCCAP), 1(3):269–288, 2005.

[23] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[24] J Chassin de Kergommeaux, Benhur Stein, and Pierre-Eric Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, 2000.

[25] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[26] Guozhu Dong and James Bailey. *Contrast Data Mining: Concepts, Algorithms, and Applications*. Chapman & Hall/CRC, 1st edition, 2012.

[27] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 43–52. ACM, 1999.

[28] Damien Dosimont, Generoso Pagano, Guillaume Huard, Vania Marangozova-Martin Huard, and Jean-Marc Vincent. Efficient analysis methodology for huge application traces. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 951–958. IEEE, 2014.

[29] Johannes Fischer, Volker Heun, and Stefan Kramer. Optimal string mining under frequency constraints. In *Knowledge Discovery in Databases: PKDD 2006*, pages 139–150. Springer, 2006.

[30] Jonathan Foote. Automatic audio segmentation using a measure of audio novelty. In *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, volume 1, pages 452–455. IEEE, 2000.

[31] Philippe Fournier-Viger, Cheng-Wei Wu, and Vincent S Tseng. Mining maximal sequential patterns without candidate maintenance. In *Advanced Data Mining and Applications*, pages 169–180. Springer, 2013.

[32] Chuancong Gao, Jianyong Wang, Yukai He, and Lizhu Zhou. Efficient mining of frequent sequence generators. In *Proceedings of the 17th international conference on World Wide Web*, pages 1051–1052. ACM, 2008.

[33] Andreas Gerstlauer, Christian Haubelt, Andy D Pimentel, Todor P Stefanov, Daniel D Gajski, and Jürgen Teich. Electronic system-level synthesis methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1517–1530, 2009.

[34] Susan L Graham, Peter B Kessler, and Marshall K McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.

[35] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques (Second Edition)*. Elsevier, 2006.

[36] Michael T Heath, Jennifer Etheridge, et al. Visualizing the performance of parallel programs. *Software, IEEE*, 8(5):29–39, 1991.

[37] Danny Holten, Bas Cornelissen, and Jarke J Van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 47–54. IEEE, 2007.

[38] Hwa-You Hsu, James A Jones, and Alessandro Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 439–442. IEEE Computer Society, 2008.

[39] Xiao Hu and Shuming Chen. Applications of on-chip trace on debugging embedded processor. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, pages 140–145. IEEE, 2007.

[40] Oleg Iegorov, Vincent Leroy, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. Data mining approach to temporal debugging of embedded streaming applications. In *Proceedings of the 12th International Conference on Embedded Software*, pages 167–176. IEEE Press, 2015.

[41] Xiaonan Ji, James Bailey, and Guozhu Dong. Mining minimal distinguishing subsequence patterns with gap constraints. *Knowledge and Information Systems*, 11(3):259–286, 2007.

[42] Christiane Kamdem Kengne, Noah Ibrahim, Marie-Christine Rousset, and Maurice Tchuente. Distance-based trace diagnosis for multimedia applications: Help me ted! In *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pages 306–309. IEEE, 2013.

[43] R Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux Journal*, 2005(133):11, 2005.

[44] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 755–760. European Design and Automation Association, 2013.

[45] Edward Lee, Thomas M Parks, et al. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[46] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[47] Chun Li and Jianyong Wang. Efficiently mining closed subsequences with gap constraints. In *SDM*, pages 313–322. SIAM, 2008.

[48] Xiaoqing Liu, Jun Wu, Feiyang Gu, Jie Wang, and Zengyou He. Discriminative pattern mining and its applications in bioinformatics. *Briefings in bioinformatics*, page bbu042, 2014.

[49] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 557–566. ACM, 2009.

[50] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469. ACM, 2007.

[51] Patricia López Cueva, Aurélie Bertaux, Alexandre Termier, Jean François Méhaut, and Miguel Santana. Debugging embedded multimedia application traces through periodic pattern mining. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 13–22. ACM, 2012.

[52] Congnan Luo and Soon Myoung Chung. Efficient mining of maximal sequential patterns using multiple samples. In *SDM*, pages 415–426. SIAM, 2005.

[53] Roberto Mijat. *Better Trace For Better Software With CoreSight STM (White paper)*. ARM, 2010. https://www.arm.com/files/pdf/Better_Trace_for_Better_Software_-_CoreSight_STM_with_LTTng_-_19th_October_2010.pdf.

[54] Shirley V Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Computational Science—ICCS 2002*, pages 904–912. Springer, 2002.

[55] David W Mount and David W Mount. *Bioinformatics: sequence and genome analysis*, volume 2. Cold spring harbor laboratory press New York:, 2001.

[56] Frank Mueller and David B Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[57] Benjamin Negrevergne and Tias Guns. Constraint-based sequence mining using constraint programming. In *Integration of AI and OR Techniques in Constraint Programming*, pages 288–305. Springer, 2015.

[58] Andrew F Neuwald, Jun S Liu, David J Lipman, and Charles E Lawrence. Extracting protein alignment models from the sequence database. *Nucleic Acids Research*, 25(9):1665–1677, 1997.

[59] Daniel K Osmari, Huy T Vo, Claudio T Silva, Joao LD Comba, and Lauro Lins. Visualization and analysis of parallel dataflow execution with smart traces. In *Graphics, Patterns and Images (SIBGRAPI), 2014 27th SIBGRAPI Conference on*, pages 165–172. IEEE, 2014.

[60] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11(285-296):23–27, 1975.

[61] Myeong-Chul Park, Young-Joo Kim, In-Geol Chun, Seok-Wun Ha, and Yong-Kee Jun. A gdb-based real-time tracing tool for remote debugging of soc programs. In *Advances in Hybrid Information Technology*, pages 490–499. Springer, 2007.

[62] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99*, pages 398–416. Springer, 1999.

[63] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *icccn*, page 0215. IEEE, 2001.

[64] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-François Nezan. Dataflow model of computation. In *Physical Layer Multi-Core Prototyping*, pages 53–75. Springer, 2013.

[65] Ramesh V Peri, Sanjay Jinturkar, and Lincoln Fajardo. A novel technique for profiling programs in embedded systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.

[66] Laurence Pierre, Luca Ferro, Zeineb Bel Hadj Amor, Philippe Bourgon, and Jérôme Quévremont. Integrating psl properties into systemc transactional modeling—application to the verification of a modem soc. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 220–228. IEEE, 2012.

[67] Kevin Pouget, Patricia Lopez Cueva, Matheus Santana, and Jean-François Méhaut. Interactive debugging of dynamic dataflow embedded applications. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 345–354. IEEE, 2013.

[68] Carlos Prada-Rojas, Frederic Riss, Xavier Raynaud, Serge De Paoli, and Miguel Santana. Observation tools for debugging and performance analysis of embedded linux applications. In *Conference on System Software, SoC and Silicon Debug-S4D*, 2009.

[69] Carlos Prada-Rojas, Miguel Santana, Serge De-Paoli, Xavier Raynaud, et al. Summarizing embedded execution traces through a compact view. In *Conference on System Software, SoC and Silicon Debug S4D*, 2010.

[70] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[71] James Roberts. Tracevis: an execution trace visualization tool. In *In Proc. MoBS 2005*. Citeseer, 2005.

[72] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

[73] Darlene Stewart, W Morven Gentleman, et al. Non-stop monitoring and debugging on shared-memory multiprocessors. In *Software Engineering for Parallel and Distributed Systems, 1997. Proceedings., Second International Workshop on*, pages 263–269. IEEE, 1997.

[74] Daniel Sundmark. *Deterministic replay debugging of embedded real-time systems using standard components*. Mälardalen University, 2004.

[75] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[76] Wim Taymans, Steve Baker, Andy Wingo, Ronald S Bultje, and Stefan Kost. Gstreamer application development manual (1.5. 0.1). 2008.

[77] John W Tukey. Exploratory data analysis. 1977.

[78] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *Discovery science*, pages 16–31. Springer, 2004.

[79] Bart Vermeulen, Neal Stollon, Rolf Kühnis, Gary Swoboda, and Jeff Rearick. Overview of debug standardization activities. *Design & Test of Computers, IEEE*, 25(3):258–267, 2008.

[80] Jianyong Wang, Jiawei Han, and Chun Li. Frequent closed sequence mining without candidate maintenance. *Knowledge and Data Engineering, IEEE Transactions on*, 19(8):1042–1056, 2007.

[81] David Weese and Marcel H Schulz. Efficient string mining under constraints via the deferred frequency index. In *Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, pages 374–388. Springer, 2008.

[82] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[83] Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, and Brian JN Wylie. Large event traces in parallel performance analysis. *Architecture of Computing Systems, ARCS 2006 (19a. Frankfurt-Main, Alemanya)*, 2006.

[84] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, 2008.

[85] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM*, pages 166–177, 2003.

[86] Shengwei Yi, Tianheng Zhao, Yuanyuan Zhang, Shilong Ma, and Zhanbin Che. An effective algorithm for mining sequential generators. *Procedia Engineering*, 15:3653–3657, 2011.

[87] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[88] Mohammed J Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

[89] Daniel Zwillinger and Stephen Kokoska. *Standard Probability and Statistical Tables and Formula*. Chapman & Hall, Boca Raton, 2000.

Abstract

Debugging streaming applications run on multimedia embedded systems found in modern consumer electronics (e.g. in set-top boxes, smartphones, etc) is one of the most challenging areas of embedded software development. With each generation of hardware, more powerful and complex Systems-on-Chip (SoC) are released, and developers constantly strive to adapt their applications to these new platforms. Embedded software must not only return correct results but also deliver these results on time in order to respect the Quality-of-Service (QoS) properties of the entire system. The non-respect of QoS properties lead to the appearance of temporal bugs which manifest themselves in multimedia embedded systems as, for example, glitches in the video or cracks in the sound. Temporal debugging proves to be tricky as temporal bugs are not related to the functional correctness of the code, thus making traditional GDB-like debuggers essentially useless. Violations of QoS properties can stem from complex interactions between a particular application and the system or other applications; the complete execution context must be, therefore, taken into account in order to perform temporal debugging. Recent advances in tracing technology allow software developers to capture a trace of the system's execution and to analyze it afterwards to understand which particular system activity is responsible for the violations of QoS properties. However, such traces have a large volume, and understanding them requires data analysis skills that are currently out of the scope of the developers' education.

In this thesis, we propose SATM (Streaming Application Trace Miner) - a novel temporal debugging approach for embedded streaming applications. SATM is based on the premise that such applications are designed under the dataflow model of computation, i.e. as a directed graph where data flows between computational units called actors. In such setting, actors must be scheduled in a periodic way in order to meet QoS properties expressed as real-time constraints, e.g. displaying 30 video frames per second. We show that an actor which does not eventually respect its period at runtime causes the violation of the application's real-time constraints. In practice, SATM is a data analysis workflow combining statistical measures and data mining algorithms. It provides an automatic solution to the problem of temporal debugging of streaming applications. Given an execution trace of a streaming application exhibiting low QoS as well as a list of its actors, SATM firstly determines exact actors' invocations found in the trace. It then discovers the actors' periods, as well as parts of the trace in which the periods are not respected. Those parts are further analyzed to extract patterns of system activity that differentiate them from other parts of the trace. Such patterns can give strong hints on the origin of the problem and are returned to the developer. More specifically, we represent those patterns as minimal contrast sequences and investigate various solutions to mine such sequences from execution trace data.

Finally, we demonstrate SATM's ability to detect both an artificial perturbation injected in an open source multimedia framework, as well as temporal bugs from two industrial use cases coming from STMicroelectronics. We also provide an extensive analysis of sequential pattern mining algorithms applied on execution trace data and explain why state-of-the-art algorithms fail to efficiently mine sequential patterns from real-world traces.

Résumé

Débogage des applications de streaming qui s'exécutent sur les systèmes embarqués multimédia (trouvés dans les boîtes décodeurs, smartphones, et autres appareils électroniques grand public) est l'un des domaines les plus exigeants dans le développement du logiciel embarqué. Les nouvelles générations du materiél embarqué introduisent des nouveaux systèmes sur une puce, qui fait que les développeurs du logiciel doivent adapter leurs logiciels aux nouvelles platformes. Le logiciel embarqué doit non seulement fournir des résultats corrects mais aussi le faire à temps réél afin de respecter les propriétés de qualité de service (Quality-of-Service, QoS) du système. Lorsque les propriétés QoS ne sont pas respectées, les bugs temporels font son apparition. Ces bugs se manifestent comme, par exèmple, des glitches dans le flux vidéo ou des craquements dans le flux audio. Le débogage temporel est en général difficile à effectuer car les bugs temporels n'ont pas souvent de rapport avec l'exactitude fonctionnelle du code des applications, ce qui fait les outils de débogage traditionels, comme GDB, peu utiles. Le non-respect des propriétés QoS peut originer des interactions entre les applications ou entre les applications et les processus systèmes. Par conséquent, le contexte d'exécution entier doit être pris en compte pour le débogage temporel. Les avancements récents en collecte des traces d'exécution permettent les développeurs de recueillir des traces et de les analyser après la fin d'exécution pour comprendre quelle activité système est responsable des bugs temporels. Cependant, les traces d'exécution ont une taille conséquente, ce qui demande aux devéloppeurs des connaissainces en analyse de données qu'ils normalement ne possèdent pas.

Dans cette thèse, nous proposons SATM - une approche novatrice pour le débogage temporel des applications de streaming. SATM repose sur la prémisse que les applications sont conçues avec le modèle dataflow, i.e. peuvent être représentées comme un graphe orienté où les données coulent entre des unités de calcul (fontions, modules, etc.) appelées "acteurs". Les acteurs doivent être ordonnés de manière périodique afin de respecter les propriétés QoS représentées par les contraintes de temps-réél, e.g. une affichage des 30 trames vidéo par seconde. Nous montrons qu'un acteur qui ne respecte pas de façon répétée sa période pendant l'exécution de l'application cause la violation des contraintes temps-réél de l'application. Pratiquement, SATM est un workflow d'analyse de données venant des traces d'exécution qui combine des mesures statistiques avec des algorithmes de fouille de données. SATM fournit une méthode automatique du débogage temporel des applications de streaming. Notre approche prend en entrée une trace d'exécution d'une application ayant une basse QoS ainsi qu'une liste de ses acteurs, et tout d'abord détecte des invocations des acteurs dans la trace. SATM ensuite découvre les périodes des acteurs ainsi que les séctions de la trace où la période n'a pas été respectée. Enfin, ces séctions sont analysées afin d'extraire des motifs de l'activité système qui différencient ces sections des autres séctions de la trace. Tels motifs peuvent donner des indices sur l'origine du problème temporel dans le système et sont rendus au devéloppeur. Plus précisément, nous représentons ces motifs comme des séquences contrastes minimales et nous étudions des différentes solutions pour fouiller ce type de motifs à partir des traces d'exécution.

Finalement, nous montrons la capacité de SATM de détecter une perturbation temporelle injectée artificiellement dans un framework multimédia GStreamer, ainsi que des bugs temporels dans les deux cas d'utilisation des applications de streaming industrielles provenant de la société STMicroelectronics. Aussi, nous fournissons une analyse extensive des algorithmes de fouille de motifs séquentiels appliqués sur les données venant des traces d'exécution, et nous expliquons quelle est la raison que les algorithmes de pointe n'arrivent pas à fouiller les motifs séquentiels à partir des traces d'exécution de façon efficace.