UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Patricia López Cueva

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

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

Debugging Embedded Multimedia Application Execution Traces through Periodic Pattern Mining

Thèse soutenue publiquement le **8 juillet 2013**,
devant le jury composé de :

Mr, Frédéric Pétrot
Professeur à Grenoble INP, Président
Mr, Hiroki Arimura
Professeur à l'Université Hokkaido, Rapporteur
Mr, Gilles Sassatelli
Directeur de recherche CNRS, Rapporteur
Mr, Takashi Washio
Professeur à l'Université d'Osaka, Examinateur
Mr, Jean-François Boulicaut
Professeur à l'Université de Lyon, Examinateur
Mr, Jean-François Méhaut
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mr, Alexandre Termier
Maître de conférences à l'Université Joseph Fourier, Co-Directeur de thèse
Mr, Miguel Santana
Directeur du centre IDTEC à STMicroelectronics, Co-Directeur de thèse

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, multimedia embedded systems populate the market of consumer electronics with products such as set-top boxes, tablets, smartphones and MP4 players. This highly competitive industry pressures manufacturers to design *better* products (better in terms of performance and power consumption), always providing new features, and before the competitors (time-to-market). Moreover, semiconductor manufacturers need to provide costumers not only with the hardware, but also with software that simplifies the development of applications. Therefore, a significant part of this pressure is passed on to software developers, who need to develop, debug and optimize their software in as little time as possible while dealing with platforms which never cease to increase in complexity.

1.1 Embedded Systems

As has been the case for personal computers, the computational power provided by consumer electronics has not ceased to increase, motivated by ever increasing user demand. The more computational power the system has, the more resource-intensive applications the device is able to run. This effect fuels the user's imagination about what would be possible with even more computational power, thus creating more demand.

Therefore, multimedia embedded systems are required to keep increasing their computational power while reducing their power consumption as much as possible and keeping (and if possible reducing) their size and/or weight. Currently, this effect is more observable on smartphones and tablets, where the user concerns over operating time are one of the main keys when deciding which product to purchase, but it is also present in the set-top box market since costumers do not want a big and energy-greedy device at home that substantially increases the electricity bill.

Increase in hardware complexity. In order to comply with these requirements, these systems make use of highly integrated System on Chip (SoC) solutions, which present lower power consumption, faster circuit operation, and a smaller physical size than standard microprocessors. Moreover, SoCs reduce the number of physical chips needed for a system which in turn reduces their production cost, and therefore, their price.

SoCs generally present a heterogeneous architecture composed of one or more processors, that control all components, and accelerators, that provide the computational capability needed for multimedia applications such as audio/video decoding, image processing and Three-Dimensional (3D) graphics. The different components of the system access a shared main memory and external devices through several communication buses connected to the different communication ports: serial, USB, Ethernet, HDMI, etc.

Initially, SoC architectures contained one or maybe two instruction-set processors, but soon the need for higher performance made designers introduce multiple instruction-set processors on the same chip, thus creating Multiprocessor System on Chip (MPSoC). MPSoCs present multiple processors for control, data processing, media processing, etc. However, the parallelism introduced by the use of several processors makes these systems more complex to implement.

In order to keep up with the demanded computational power, semiconductor manufacturers long used the solution of increasing the clock frequency, which substantially increases system power consumption. When the level of power consumption became unacceptable, semiconductor manufacturers adopted multi-core processors which provide high performance by taking advantage of parallel processing. However, architectures using multi-core processors present yet another level of complexity due to the parallel processing and the hierarchical memory normally found on them.

Summarizing, all the solutions adopted by semiconductor manufacturers to answer the demand of computational power while keeping the power consumption as low as possible have substantially increased the complexity of the hardware by introducing more and more parallelism on the system, thus complicating the development of software fully adapted to these architectures.

Increase in software complexity. This increase in hardware complexity entails an increase in software complexity. In order to optimize the system performance without overloading any part of it, the software needs to manage resource sharing as well as to balance the load on all cores/processors. To achieve this, many aspects need to be taken into account: memory access patterns (in order to place the threads working on the same data on cores sharing cache memory), communication patterns (so that the messages do not need to travel far), thread placement (in order to achieve a balanced load of the work), and so on.

The research on optimization of applications running on multi-core architectures is yet to find an optimal solution fitting all systems and applications, which is out of the scope of this thesis. Nevertheless, we can observe that this is not an easy task for developers who need to deal with the high level of parallelism and resource sharing while taking into account the complex architecture their software is going to be executed on.

Software Development and Validation. This increasing complexity in both the software and the underlying hardware, and ever tighter time-to-market pressures are some of the key challenges faced when designing multimedia embedded systems. Thus, developing efficient and robust applications for these systems is a challenging issue. With the increase in the complexity of embedded systems, the software development and validation processes have increased in cost and duration.

As part of the validation process, developers try to discover and eliminate as many bugs as possible. However, some bugs might remain hidden until the integration process, e.g. differences between the simulator and the real platform might reveal previously hidden bugs, which generally are very costly because they are very difficult to diagnose and solve. Therefore, it is critical for companies developing embedded software to have comprehensive programming frameworks with advanced features for debugging and optimizing their software.

1.2 Debugging Embedded Systems

Between the different debugging and performance analysis techniques, the most commonly used techniques are functional and performance debugging:

Functional Debugging consists in checking whether the software does what its design says it should do. The main tools used in software functional debugging are interactive debuggers. Interactive debuggers allow developers to interact with the application by pausing the execution using breakpoints. Then, they can inspect the state of the system and execute the application step-by-step to better inspect it. This technique is useful to identify easy-to-find bugs that leave some kind of evidence, i.e. the application halts or fails.

Performance Debugging consists in checking whether the software does what it should do on the execution time defined by the design. The main tools used in software performance debugging are profilers. Profilers offer an overview of the application execution giving summary information about memory usage, execution time, etc. This information is useful in order to have a global idea of the performance of the application, and to identify hot-spots and bottlenecks on the application execution.

Interactive debugging and profiling do not provide enough information to debug or optimize certain aspects of multimedia applications, especially the ones related with the complex interaction between the components. When using interactive debugging to find these type of bugs, it is difficult to know when to pause the execution or whether by pausing the execution the bug will be reproducible. Moreover, the behavior of these bugs is difficult to repeat since the execution of multi-threaded applications is non-deterministic. On the other hand, profilers ignore the order in which the events take place on the system and, if this is what is affecting the performance of the system, they do not provide enough information to be able to diagnose the problem.

Therefore, a more comprehensive tool, that offers the possibility of carrying out simultaneously functional and performance debugging, has lately gained a place in the debuggers toolset: tracing.

A multipurpose solution: Tracing. Tracing consists in recording the execution of the system and the application for a postmortem analysis [CGMM+11] [KWK10] [PRRR+09] [KMW12]. The intrusiveness of this technique is normally controlled by the developer by selecting which events to trace and the intrusion introduced by the tracing process of an event,

which depends on the tracing technique. Tracing techniques were quite costly at the beginning, e.g. trap based tracepoints and output on the console, but significant improvements have lead to new tracing techniques such as kernel markers, function based tracepoints, fast tracepoints, etc., and new event recording techniques such as the use of a circular buffer. Moreover, dedicated hardware has lately been introduced on embedded systems to minimize the impact generated by tracing an event.

Generally, tracing techniques record the events but also their timestamp information, which allows for a detailed analysis of the execution. Moreover, it facilitates the analysis and diagnosis of bugs generated by the complex interaction between components or processes of the system. As explained above, these types of bugs are the most difficult to diagnose and solve. Moreover, with the introduction of multi-core processors, these bugs are becoming more and more common, which drives developers to use execution trace analysis to debug their applications.

Furthermore, most of the information provided by profilers can be easily calculated using the execution trace, while keeping chronological information that allows a detailed analysis of the hot-spots identified by the profiling information.

Since execution traces provide a more complete analysis of the application execution, developers of embedded applications are turning more and more towards the use of execution traces for debugging and optimizing their applications. Therefore, in this thesis we are going to focus on the analysis of execution traces as an application debugging and optimizing technique.

Currently, once an execution trace is retrieved, it is manually analyzed by the developer in order to debug the application. By manually, we mean for example the use of search tools such as `grep` in order to find a specific part of the trace, or the use of visualization tools that offer a structured view of the trace for an easier analysis.

As explained above, the software and hardware of embedded systems have faced an increase in complexity due to the parallelization of the computation, power and memory optimizations, and so on. Consequently, the size of execution traces of applications running on these systems has also increased, and we can only expect an even bigger increase with the introduction of many-core processors in embedded systems.

Therefore, the manual analysis of execution traces is becoming an unmanageable task. In order to reduce the debugging time, automatic analysis techniques for execution traces are needed. These techniques should help the developer to understand what is happening in the application, i.e. to identify correct and anomalous behaviors. As mentioned above, statistical or profiling information do not give enough detail to decipher the behavior of the application, thus a more sophisticated technique is needed.

1.3 Automatic analysis of big volumes of data: Data Mining

Data mining techniques have long been used to extract useful information from large data sets through automatic data analysis [Kan02]. These techniques mix traditional data analysis methods with sophisticated algorithms for processing large data volumes in order to discover useful information. The four main data mining tasks are clustering, classification, pattern mining and anomaly detection.

Clustering groups objects into classes, without knowing the set of classes in advance. Clustering could help in the analysis of the trace by grouping together similar areas of the trace. However, the description provided is to coarse-grained for the detailed analysis needed by developers.

Classification techniques receive a set of classes of objects and assign new objects to these classes. For example, classification could help finding anomalous events, although anomalies need to be known in advance, which is difficult in application analysis where the errors can come from many different sources. Moreover, classification techniques are semi-automatic since they require a set of classes as inputs but what developers need is an automatic tool.

Anomaly detection techniques help finding events that do not fit into an established normal behavior. But in order to do so they need to be able to tell whether an event or a pattern is an anomaly which, as explained in the previous paragraph is not as easy as it might sound.

Pattern mining finds regularities in the trace that are called *patterns*. Since developers need to understand what is happening in the application in order to debug it, we consider that pattern mining techniques can help in the analysis of execution traces by automatically discovering hidden relationships between the events. Moreover, the description provided by these techniques is fine-grained, which is the level of detail developers need to analyze the behavior of the application.

Different types of patterns give different kinds of information. For example, frequent itemsets can give us the information about the co-occurrence of different events on the trace, e.g. an interrupt and its corresponding software interrupt, or a middleware function called by the application and all low level functions called by it. Sequences add information about the order in which these events happened, e.g. a `mutex_lock` is normally followed by a `mutex_unlock`, and a frame cannot be displayed until all decoding operations have been carried out over it.

Therefore, to adapt the analysis to multimedia applications, we need to define what kind of behavior it is necessary to discover in order to understand whether there is a problem to debug or any space for optimization. Important characteristics of multimedia applications are the **periodic** processing of frames and the assurance of a certain Quality of Service (QoS). If all operations over a frame are carried out in the time interval given by the sampling rate, then the user of the embedded system will not see any blanks on the image or hear any cracks/silences in the sound. On the contrary, if any of these operations takes longer than expected, the QoS is affected.

Therefore, we propose to use pattern mining techniques on execution traces to discover **periodic** behaviors in order to debug and optimize multimedia applications. Since main operations on multimedia applications are **periodic**, most sub-operations probably present a **periodic** behavior as well. By discovering these behaviors developers can better understand what is going on inside the application. Moreover, by discovering a **periodic** behavior it is easy to discover when it is not respected which might indicate the location of a bug in the application.

As an example, while playing back an audio file at 48 kHz, an audio frame should be read from memory, decoded and sent to the speakers every 20 milliseconds. Let's imagine that the developer tests this functionality of the multimedia application and that he/she

hears a gap or crack in the sound. In order to debug this, the developer can analyze the trace of the execution, shown in Figure 1.1. **Periodic pattern mining** techniques would give the developer the information about which events are executed together on a **periodic** basis. Therefore, the events related with reading a frame from the memory (blue), decoding it (red) and sending it (green) to the speakers would be discovered by the pattern mining techniques and their periodicity would be of 20 milliseconds. By analyzing the pattern and its occurrences, the developer can discover where the periodicity was not respected (no events in time interval 60-80 ms) and therefore where the crack in the sound took place on the execution.

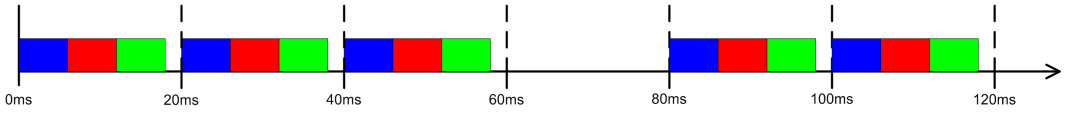


Figure 1.1: Visualization of an audio file playback

Periodic pattern mining techniques have not been extensively studied yet, even less regarding the analysis of execution traces. Therefore, in this thesis, we are interested in discovering **periodic** patterns in execution traces of multimedia applications in order to debug and optimize these applications. In the rest of this chapter, we present the contributions of this thesis as well as the context in which this thesis was carried out. Finally, we present the structure of this document.

1.4 Contributions

In this thesis, we propose a new debugging technique based on frequent periodic pattern mining of execution traces. For this, we exploit a definition of periodic pattern proposed by Ma et al. [MH01] which is well suited to the variety of frequent periodic patterns that can be found on multimedia application execution traces.

Our contributions are the following:

- A condensed representation of the set of frequent periodic patterns, called Core Periodic Concept (CPC). As is the case for other exhaustive frequent pattern mining problems, periodic pattern mining suffers from combinatorial explosion that leads to a huge number of results with high redundancy among them. A classical solution is to use condensed representation such as closed [PBTL99] or non-derivable [CG02] patterns. However, in our settings, closure techniques cannot be exploited, as periodic patterns are ternary relations where Galois connexion can not be defined. We thus proposed a novel condensed representation for frequent periodic patterns based on a triadic approach, that we call Core Periodic Concepts (CPC). CPC reduce the number of patterns outputted by two to three orders of magnitude, without loss of information.
- A study of the **"connectivity" properties** related to CPC, where connectivity corresponds to the spacing of the transactions supporting the CPC. In periodic patterns there is a strong relationship between the periods and the transactions which is not taken into account in the generic triadic approach. These properties allow to prune from the search space exploration, candidates that will not lead to CPC.

- A **depth-first algorithm** that mines the CPC set called *PerMiner* without enumerating the whole set of frequent periodic patterns, of proven polynomial space and polynomial-delay time complexity. This algorithm makes use of the defined connectivity properties to avoid generating all frequent periodic patterns by pruning out the ones that are not going to generate a CPC.

- A first step towards a **methodology** to use periodic pattern mining to debug multimedia application execution traces. Application developers are not familiar with pattern mining algorithms so we propose a methodology to help them understand what are the different steps they need to follow in order to use our approach.

- A **periodic pattern visualization tool** that helps in the analysis of the results facilitating the discovery of anomalies in the periodicity of the patterns. The results of the *PerMiner* algorithm are presented in a text file with a CPC on each line of the file. Considering that periodic patterns contain their list of occurrences as part of their information, this visualization tool provides a graphical visualization of the list of occurrences which facilitates the identification of unexpected behaviors.

1.5 Scientific Context

As we have seen in this introduction, this thesis touches two main research fields which are embedded systems and data mining. Regarding the part of the thesis related with embedded systems, i.e. the debugging of embedded multimedia applications using execution traces, this thesis was carried out in collaboration with the expertise center Integrated Development Tools Expertise Center (IDTEC) from STMicroelectronics S.A and scientifically suppervised by the research team NANOSIM from the University of Grenoble.

IDTEC team works on the design, development and support of debugging and observation tools for the development of multimedia embedded applications. The current objective is to design and develop a set of generic tools adapted to the new generation of MPSoCs platforms, i.e. with a high parallelism due to the increase in the number of cores.

The main research fields of NANOSIM are High Performance Computing (HPC) and embedded systems. Concretely, regarding the field of embedded systems, NANOSIM team is interested in the next generation of debugging tools which include mechanisms for observing, debugging and tracing embedded system. As these systems become more complex, the data generated by the observation of the system is going to acquire a considerable volume, so NANOSIM is also interested in the management of these big volumes of data.

Regarding data mining, this thesis was supervised by the research team HADAS from the University of Grenoble. HADAS research field is data mining and concretely HADAS team is interested in data mining techniques to extract patterns of interest from large amounts of data.

1.6 Organization of the Thesis

The rest of this document is organized in three main parts:

- Background: in Chapter 2 we aim at giving a comprehensive background view of the multimedia application debugging difficulties and limitations that we try to overcome with the work presented in this thesis. Also, we present basic concepts related to pattern mining that will be useful to understand the contributions of this thesis.
- Pattern Mining Contribution: in this part we introduce all definitions and theory needed to understand the proposed CPC definition (Chapter 3), the proposed algorithm to mine the set of CPCs (Chapter 4) and scalability experiments (Chapter 5).
- Debugging Contribution: in this part we first introduce a first step towards a methodology for the debugging of multimedia application execution traces through periodic pattern mining (Chapter 6), a new periodic pattern visualization tool that facilitates the analysis of the results (Chapter 7) and several use cases that assess the utility of our approach (Chapter 8).
- Concluding Part: this part presents a study of previous work on periodic pattern mining, ternary relations, pattern visualization and debugging of embedded systems (Chapter 9), and finally presents the conclusions and future work. (Chapter 10).

2 Background

Consumer electronics users are continuously demanding to be able to run more and more applications on their devices, up to the point of almost replacing personal computers, as is the case of tablets and smart-phones. Most of these applications are multimedia applications: video and audio playback, video and audio recording, videoconferencing, speech recognition, etc., which require a substantial amount of computational power.

Moreover, users demand products to be as small and light-weight as possible (more portable / less cumbersome) and to consume as little power as possible (longer battery life / less impact on the electricity bill). In order to answer these demands, consumer electronics' architecture should provide a high level of integration and parallelism. Indeed, MPSoC architectures, fulfilling these requirements by definition, are the best-fit solution for consumer electronics.

Therefore, in Section 2.1, using as an example SoCs present on set-top boxes, we study the evolution of MPSoC architectures. In Section 2.2, we introduce the architecture of multimedia applications. In Section 2.3, we present current and future tracing mechanism in embedded systems. In Section 2.4, we study the current trace analysis techniques. In Section 2.5, we explain some basic concepts of data mining which will be helpful to understand the rest of this document. Finally, in Section 2.6, we conclude this chapter.

2.1 Embedded Systems in Multimedia

Embedded systems are purpose-built systems that generally have limited resources (memory) and hard requirements (power consumption, size, weight, etc.). These systems are gradually becoming part of our everyday life, being found in a vast variety of products ranging from transport systems over consumer electronics up to home appliances.

The high integration level of these systems, necessary to provide the required computational power while keeping a low power consumption and a manageable size and weight, is provided by MPSoC architectures [Wol04]. Indeed, MPSoCs are responsible for this invasion of embedded systems for the reasons stated above, but also due to their low production cost, as they can be produced on large numbers by highly automated processes.

As we will show in this section, the complexity of these architectures never cease to increase in order to be able to comply with the functional requirements (multitasking, performance, etc.) but also the non-functional requirements such as the manufacturing cost and the power consumption between others. Therefore, platform and software framework design process is an intricate task.

When faced to functional and non-functional requirements of a system, the first step of the design is the architectural design, i.e. a block diagram showing the main operations and data flows of the system. Then, this diagram is refined into two diagrams: one for hardware and one for software [Wol01]. One of the most difficult tasks is to decide which operations are carried out by the hardware and which by the software.

Classical design approaches appoint separate teams for software and hardware design. Following these approaches, the platform is designed and validated against a given set of requirements and specifications, which tends to produce over-designed platforms. Moreover, the software design cannot start until the hardware design finishes. This means that any problem discovered during software development related with the platform cannot be solved, forcing the engineers to think of a workaround. Thus, the whole design process takes far too long with these approaches [JW05].

Time-to-market pressures joined to the cost of designing a new platform have motivated semiconductor companies to adopt design methodologies where the collaboration between the software and the hardware design teams is stronger.

A good example is the co-design of hardware and software together, which allows optimizations that would have remain hidden otherwise, such as the introduction of application-specific operations or reconfigurable hardware. For example, the software could be tested without waiting for the prototype platform to be ready, being able to give quick feedback about the platform. Also, it means that software development starts well before than in classical design approaches, and therefore, the whole design process becomes much quicker. The drawback of co-design methods is that the set of possible design choices can be huge. Therefore, it might take a long time to choose the best fit for a given design problem.

Designing a SoC from scratch is a long and expensive task for semiconductor companies, which in conjunction with the need for decreasing time-to-market, has pushed them towards design methodologies focused on maximizing platform reusability, such as platform-based design methodology [MC03]. According to Chang et al. in [MC03], *"platform-based design is an organized method to reduce the time required and risk involved in designing and verifying a complex SoC, by heavy reuse of combinations of hardware and software Intellectual Property (IP)"*. The use of IP offers the advantage of reducing cost and design and development time of SoCs [CCH+99]. The objective is to design a platform optimized for a particular application domain but that is flexible enough to be tailored to a specific product.

One of the reasons for the expansion of MPSoC architectures is the existence of standards regarding the applications of these architectures [WJM08]. Complying with standards opens a wide market that can justify the cost of platform design. Standards generally provide reference implementations that can be reused by platform designers as a start point for their design. This standard-based design methodology consists in transforming the standard reference implementation, normally single threaded, into a parallel implementation and deciding which parts of the computation are to be carried out by hardware. Then, the different configurations can be validated by simulators until a final configuration is chosen. Moreover, in

these highly competitive markets, semiconductor companies need to specialize their products by providing better characteristics than the competition, such as a lower power consumption or a good balance between hardware and software operations.

The main industry that has driven the evolution of MPSoC architectures is the consumer electronics industry. Indeed, MPSoC are currently found on products such as mobile phones, set-top boxes, tablets, and so on. In this section, we are going to analyze the evolution of these MPSoCs, using as an example MPSoCs used on set-top boxes produced by STMicroelectronics, in order to show the increase in complexity that these architectures have experienced.

SoCs

Initially, set-top boxes features consisted on decoding streaming television content from a source signal into a display device such as a television screen. Moreover, satellite and digital TV were provided by private companies who used proprietary coding schemes. In the late 90's, the establishment of the MPEG-2 standard [mpe] opened this market to the competition, by offering a generic coding of video and audio.

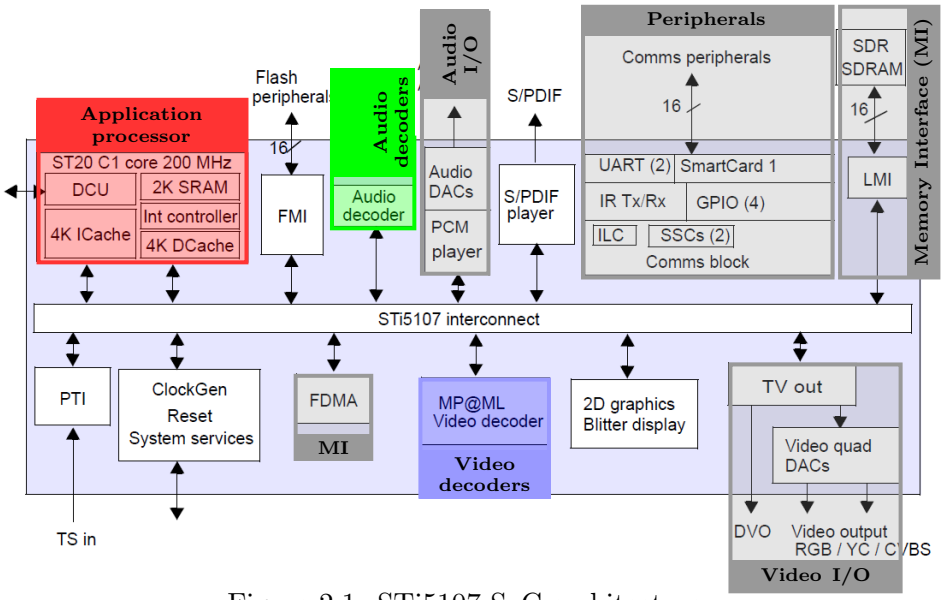


Figure 2.1: STi5107 SoC architecture

The first set-top boxes supporting MPEG-2 standard contained SoC in order to provide the necessary computational power. A good example is the SoC *STi5107* whose architecture can be seen in Figure 2.1. This SoC is used in cable, satellite or digital terrestrial set-top boxes, and it contains an *ST20* core as the application processor, the necessary connectivity, a video decoder and an audio decoder.

First MPSoCs

With the increase in the number of available channels and thus program offer, the situation where a user wanted to watch two programs being broadcast at the same time became common. The solution offered was to watch one program and record the other one to be watched later on. To be able to offer this service, semiconductor companies had to increase the parallel computation of their set-top boxes which motivated the introduction of MPSoC architectures on their products.

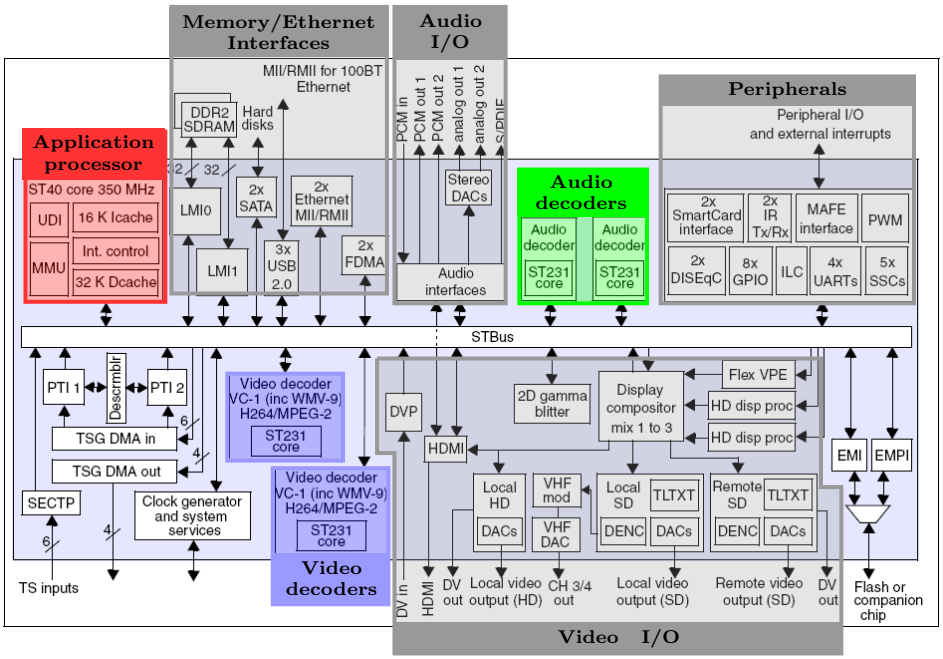


Figure 2.2: STi7200 SoC architecture

Generally, these MPSoCs contain an application processor, in charge of interfacing with the user and controlling the operation of the rest of the system, and several dedicated processors such as Digital Signal Processors (DSPs). Examples of these architectures are the PNX4008 [Sem] produced by NXP Semiconductors as part of the Nexperia family of processors, and the *STi7200* SoC produced by STMicroelectronics [STMb].

The architecture of *STi7200* SoC is shown in Figure 2.2. This SoC is used on digital terrestrial, satellite, cable and IP high-definition set-top boxes. Apart from all the necessary connectivity, this system-on-chip contains an *ST40* core [STMa], as the application processor, four *ST231* cores [FH00], in charge of the audio and video decoding, and several deeply embedded cores such as *STxP70*. Thanks to its multiple decoders, this SoC is able to decode two high-definition (HD) programs simultaneously and handle up to six external transport streams from different sources.

Multi-core MPSoCs

Smart-phones and tablets have decent-sized displays which makes them suitable to run certain multimedia applications, that were quite limited on previous mobile devices, such as video conferencing or video playback. Once at home, smart-phones and tablets have suitable displays that can show a television program, and therefore, users demand being able to watch television on these mobile devices.

Regarding other set-top box services, service providers have introduced games as a service, allowing the user to rent or buy games which are stored and played on the set-top box. Since nowadays HD and 3D televisions are more and more common, service providers are able to propose HD and 3D games to their users. However, these games require a big amount of computational power and a high interaction level with the user.

Therefore, in order to provide these services, set-top boxes had to increase their parallel computational capabilities and their computational power. The historical solution of increasing the clock frequency to gain more computational power soon found its wall when the heat produced was too expensive to cool down, and even threatened with melting the system. In order to overcome this barrier, multi-core architectures have been introduced in MPSoC architectures in order to continue increasing the computational power while limiting as much as possible the heat and the power consumption.

Currently, mainly dual-core processors have been used on multimedia embedded systems, e.g. Snowball [Sno] is a good example of a dual-core architecture. This year, the first quad-core smart-phones have appeared on the market such as the 1.4GHz Exynos 4 Quad chip contained on the Samsung Galaxy S3, or the NVIDIA Tegra 3 SoC with a 1.5GHz quad-core ARM processor contained on the HTC One X.

Regarding set-top boxes, Orly STiH416 MPSoC, designed by STMicroelectronics, provides high performance with a low power consumption. A block diagram of this MPSoC, shown in Figure 2.3, shows an ARM® Cortex^{TM}-A9 MPCore^{TM} dual core [ARMa] and a ST40 as the application processors, a GPU ARM® Mali-400 MP quad core [ARMb] for HD user interfacing and HD games, and many other embedded processors (STxP70, SLIM core, etc.).

Orly MPSoC provides the necessary capabilities and computational power needed for nowadays set-top boxes. Concretely, this MPSoC is able to decode HD and 3D streams to be watched on the new generation of televisions, play 3D games, and decode four HD streams simultaneously, as well as providing the necessary security for a safe content broadcasting across different mobile devices.

As shown by this example, multi-core architectures have been introduced on the application processor but also on the accelerators in order to answer the demand for more parallel processing. Since it is predictable that the demand for more computational power will not stop, semiconductors companies are considering the transition to many-core architectures.

Many-core MPSoCs

Consumer electronics are already able to carry out many heavy computational applications and to parallelize most of their functionality. But as we have seen on this section, users will

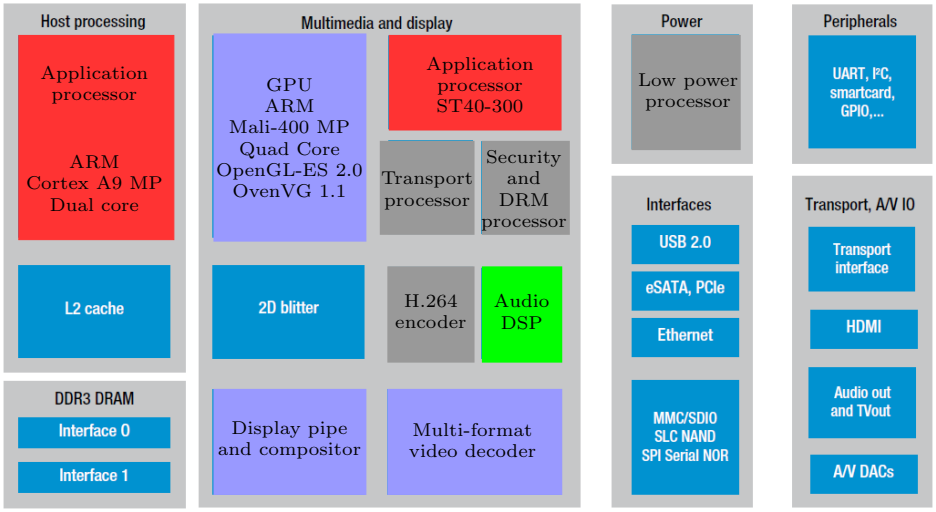


Figure 2.3: Orly Block Diagram

certainly demand new applications for their devices such as speech recognition, object recognition, augmented reality and so on. These new applications are going to require a significant increase in computational power of nowadays systems, while the energy consumption will be required to be reduced even more. Moreover, new standards such as HEVC require more computational power in order to provide a higher definition sound and image.

As we have seen on this section, semiconductor companies answer this demand for computational power by increasing the number of processing units (processors but also cores) of their SoCs. In this sense, semiconductor companies are currently working on the development of many-core processors which will provide the necessary parallelization to execute heavy computational applications over a big number of cores, and lower the energy consumption necessary to execute them.

Some examples are Tile64 [BEA+08] from Tilera Comporation, that contains 64 cores connected by a mesh network on chip where each core has its own L1 and L2 caches, the first member of the *MPPA MANYCORE* processor family from KALRAY, that contains 256 cores per chip organized in 16 clusters of 16 cores interconnected by a Network on Chip (NoC) [KAL], and *STHorm*, designed by STMicroelectronics and CEA, based on Platform 2012 architecture [MBF12].

Platform 2012 architecture is shown in Figure 2.4. This many-core processor is composed of a variable number of *clusters* that can be replicated to provide scalability, all connected by an efficient NoC infrastructure. Each cluster can contain up to 16 processors with independent instruction streams. A system bridge connects this many-core processor to an application processor ARM® Cortex^{TM}-A9.

This manycore platform was designed to be flexible as well as scalable. What initially is a homogeneous architecture, can be quickly transformed into a heterogeneous architecture by the installation of hardware accelerators, which implement functions that would be too slow in their software version. Moreover, this allows specializing the platform for a given product without having to redesign the whole platform.

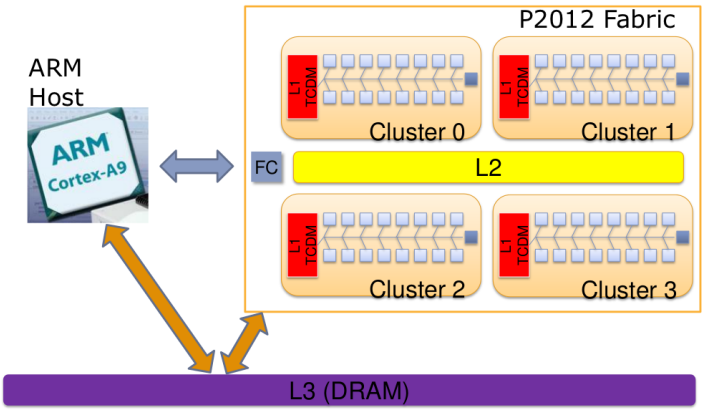


Figure 2.4: Platform 2012

In this evolution, we can clearly see the importance of the reusability and flexibility of new platforms. These characteristics increase the time-in-market of the platform, thus increasing the factor profit/cost of designing a new platform.

Synthesis

As we have shown in this section, by using as an example the evolution on the set-top box architectures, MPSoC architectures' complexity is gradually increasing. In the market of consumer electronics, this increase in complexity is motivated by the user demand for multimedia applications requiring higher and higher computational power, and the increase on the number of parallel tasks, e.g. number and complexity of data streams to decode and display.

An important aspect of this complexity is the new parallelism being offered by MPSoC architectures. Shared memory MPSoC architectures present heterogeneous architectures with a coarse grain parallelism between the different processors, while many-core architectures present a much more fine-grained parallelism. Both types of parallelism need to be handled differently by the software.

Developing software that make the most of this new parallelism is not an easy task. Software should do it for the underlying architecture but without having to completely modify it when changing platform. Following the same philosophy than in platform design, the reuse of software reduces the time required to design and develop the software for a new platform.

Therefore, in the next section we analyze the software architecture of multimedia applications to see how software tries to face the increasing hardware complexity of multimedia embedded systems.

2.2 Multimedia Applications

As said above, developing multimedia software that runs on MPSoC architectures is a complex and expensive task. In order to simplify this process, it is key to be able to reuse software

from one platform to another.

Multimedia applications handle different media sources such as text, images, sound and/or video, depending on the type of multimedia applications: media players, video conferencing, video recorders, audio or video editors, and so on. Simply said, multimedia applications carry out a series of transformations to a stream of data. These transformations are not specific of a particular multimedia application, which facilitates the reutilization of the software.

The software infrastructure found on multimedia embedded systems, shown in Figure 2.5, is split into three layers: multimedia applications, the middleware or multimedia framework, and the operating system. Multimedia applications are generally platform independent since they sit on top of multimedia frameworks that isolate the application from the platform by providing the necessary services. Multimedia frameworks are in communication with the platform-dependent components of the operating system, thus even if they are platform independent, the platform is going to dictate which services the multimedia framework is going to be able to provide to multimedia applications. Finally, the operating system is platform dependent since it has to communicate directly with the platform devices through drivers.

Multimedia frameworks, such as GStreamer (shown in Figure 2.6) or VLC, offer a wide variety of processing elements or modules that can be combined into a pipeline, whose structure and size depends on the type of multimedia application. Moreover, this pipeline architecture is well suited to exploit the parallelism and heterogeneity of MPSoC architectures.

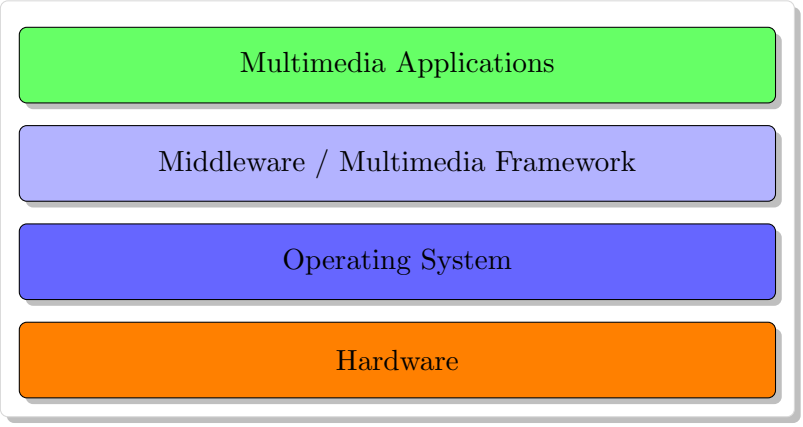


Figure 2.5: Multimedia Infrastructure.

An example of a pipeline for a simple media player is shown in Figure 2.7. The advantage of using such a framework to implement multimedia applications is that by using *plug-ins* (piece of software that can be added to a bigger application and used transparently and without any modification) the developer can easily add, for example, support to new data formats or sources.

Nowadays, most multimedia frameworks and applications are developed following the imperative programming paradigm. But as we have seen, the increase in parallelism coupled with hard non-functional requirements are going to render this programming model inadequate for efficient MPSoC software development [wPOH09]. Thus, new programming models, such as data-flow models [KS05] [BMR10], component-based model [vOvdLKM00]

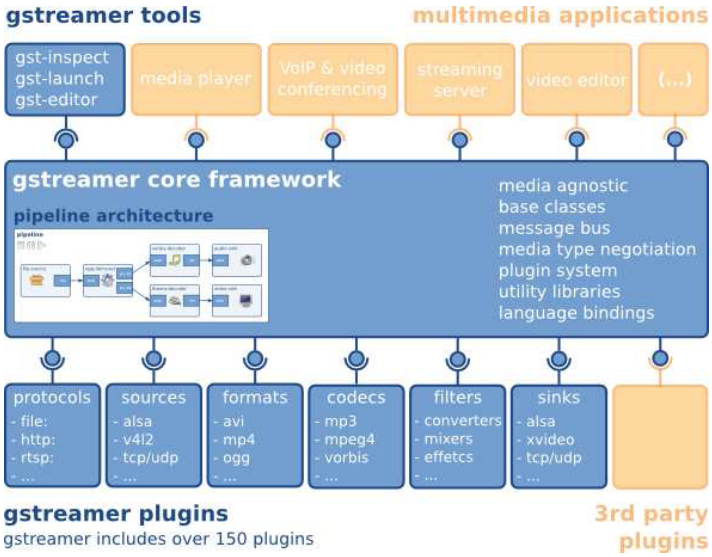


Figure 2.6: Gstreamer Framework

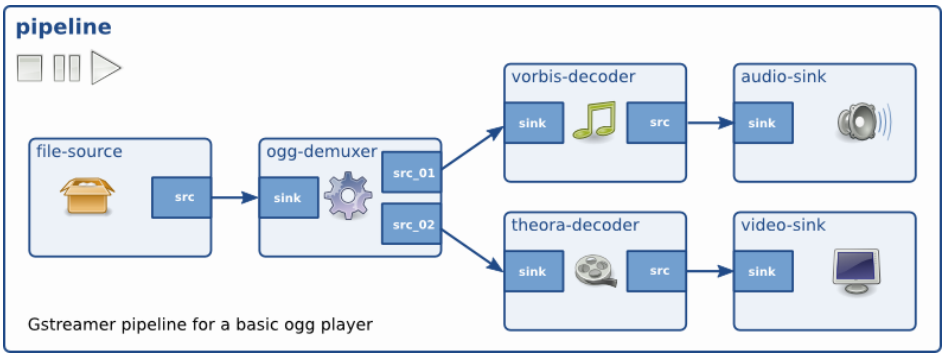


Figure 2.7: GStreamer pipeline for a simple media player

[BDRL05], and so on, are being considered as a viable option to imperative programming. These programming models are better suited to take advantage of the new offered parallelism, specially the fine-grained parallelism offered by many-core architectures.

Nevertheless, during the design and development of a product (platform + software), the complexity of the hardware and the software makes the debugging and validation phases considerably long and expensive. Regarding software, in order to debug and validate software running on an embedded system, developers need to have access to the components of the embedded system. Thus, in the next section, we are going to study how the platforms and software components, presented so far, are accessed during the debug and validation phases of the development process.

Concretely, the next section is focused on how the platform and the software support tracing mechanisms. We believe that tracing should be the debugging and validation technique used by default when working on embedded systems since it is less intrusive than interactive debuggers and cheaper than hardware solutions such as In Circuit Emulator (ICE).

2.3 Embedded Software Debugging

During the development of software for embedded systems, development and evaluation boards are used by developers to test their applications. Development boards usually include a dedicated debugging port, e.g JTAG or Serial-Wire, that allows the developer to control the execution from an external machine, called host. Through this port, an interactive debugger can start and stop the processor, single step through code, and set breakpoints on instructions and watchpoints on data accesses.

Certain aspects of the execution, specially those related to interactions between components, are hard to debug using interactive debugging or profilers. In this context, execution tracing presents itself as a more comprehensive technique that allows developers to carry out functional and non functional debugging simultaneously. Indeed, functional debugging can be covered at early stages of the design and development process through simulation, while non functional debugging can only be covered on the late phases once the final platform is available.

Different techniques exist to observe the execution of a software running on an embedded system, which vary from purely software-based to hardware-supported tracing techniques. In this section, we are going to study the different solutions proposed by semiconductor companies in order to trace embedded systems.

Software-based Tracing Techniques

The lowest level of software-based tracing consists in instrumenting the code with print statements in order to obtain a log of the execution. More modern solutions provide libraries and/or kernel modules with different instrumentation techniques, i.e. tracepoints, kprobes, kernel markers, and so on. Some operating systems offer these tracing capabilities, such as *Linux* or *STLinux*. Indeed, certain tracing tools, such as LTT [ltt] for Linux or *KPTrace* for STLinux, make use of these tracing capabilities offered by the operating system in order to provide a system level view of the software execution. But also, framework solutions such as GStreamer, with the instrumentation included in the code, offer a way of tracing the pipeline by turning on the debugging output [gst].

As an example, on an *ST40* core such as the one found in STi7200 SoC shown in Figure 2.2, applications run on STLinux. This operating system provides a tracing tool based on *KProbes* [Kri05] called *KPTrace* [kpt], which registers system and application events: interrupts, context switches, function calls, system calls, etc.

Generally, the amount of memory in embedded systems is limited, thus traced data cannot be kept in local memory but has to be transfered out of the board. Therefore, the tracing software has to send the trace out of the board to a host machine by using one of the board connexions already available, such as the serial or the Ethernet interface.

The drawback of purely software-based tracing solutions is that the amount of information that can be traced is limited. Any instrumentation of the code modifies the execution of the code, which in soft real-time systems such as multimedia systems might change the timing of the software. Therefore, these solutions should be use carefully in order to limit their intrusiveness.

Hardware-supported Tracing Techniques

Dedicated tracing hardware has been part of embedded platforms since the apparition of performance counters for profiling purposes. Later on, more complicated hardware was introduced in order to obtain traces regarding the instruction pipeline, bus transactions and memory accesses. This kind of information is useful in many situations but is too detailed when debugging complex software.

In order to give an insight into the software execution, dedicated tracing hardware has lately been introduced by semiconductor manufacturers to support software-based tracing techniques. The main areas where hardware can help are the storage of the trace and the retrieval of the trace by the host machine. The fact of conveying a trace out of the chip is expensive, thus solutions that deal with this operation without blocking the processor would allow the generation of more comprehensive traces. Also, whether the trace is stored in memory or immediately sent out of the board, dedicated hardware would reduce the intrusiveness of the equivalent purely software-based solutions.

One of the objectives of supporting software-based tracing techniques with dedicated hardware is to be able to carry out a system-wide analysis of the execution. The problem is that nowadays many embedded systems contain IPs from different vendors, thus the way of accessing each component might differ widely. This is specially visible on the debug interfaces, and thus the need to reduce time-to-market has driven several standardization activities. Between them, in 2004 a group of semiconductor companies, between them ARM, Lauterbach and STMicroelectronics, formed an alliance group called MIPI. One of the objectives of this group is to standardize the debugging and tracing interfaces of multimedia embedded systems [VKR+08].

In general terms, the proposed solution consists on having dedicated hardware modules, where the components of the architecture can write their traces, not only cores and accelerators but hardware components such as a bus profiler, that collect tracing information and send it through a dedicated trace port. There might exist extra hardware modules in charge of unifying all sources into a unique execution trace before sending it out of the chip.

Some examples of these solutions are TRACE32 [tra] by Lauterbach and CoreSight [cor] by ARM. STMicroelectronics is also working on a system-wide trace analysis infrastructure based on the use of a System Trace Module (STM) [PTBS09]. Moreover, in order to unify traces coming from different software layers and components of the system, STMicroelectronics has defined a Multi-Target Trace API [mtt] that provides a unified way of logging traces.

Nevertheless, once trace data has been retrieved from the platform, developers are in charge of analyzing it in order to diagnose functional or non functional problems of the software. In the next section, we study current techniques of trace analysis and conclude that in the future these techniques are not going to be able to treat next generation traces.

2.4 Execution Trace Analysis on Embedded Systems

Execution trace analysis consists on tracing what happens during the execution of a software and carrying out a post-mortem analysis of the execution trace. Considering that execution traces are generally a stored file or database of events, it is necessary to analyze at least a

part of the trace in order to diagnose a problem or decide whether there was no error.

Generally, visualization and analysis tools help in the analysis of execution traces by offering a graphical visualization of the trace and several analysis functionalities, such as a profiler that helps analyzing the CPU time of each process, the memory used, etc. Different techniques have been proposed to visualize execution traces [HCvW07] [Meh02] [NAW+96], but the most used one is a Gantt chart showing the timeline of the execution.

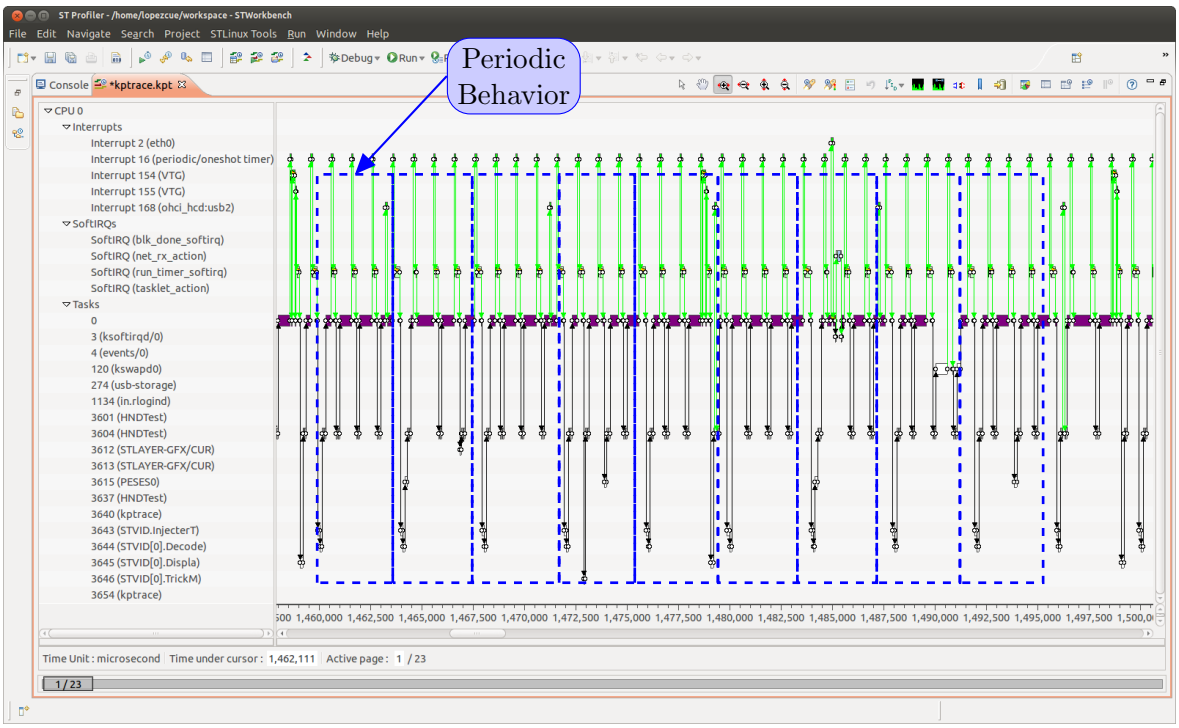


Figure 2.8: Trace viewer showing an example of a multimedia application execution trace.

Figure 2.8 shows a trace visualization and analysis tool implemented by STMicroelectronics, called *STLinux Trace Viewer* [STL]. The trace showed in the figure was generated by *KPTrace* during the execution of a media player on STLinux. The visualization of the trace consists in a timeline view of the events of the trace. The list of threads that were active during the execution are shown on the left hand side of the figure. The vertical position of each event in the timeline indicates the thread that generated the event while the horizontal position indicates its timestamp.

Then, in the top part of the figure the green arrows represent an interrupt while in the bottom part of the figure the black arrows symbolize context switches between different tasks. Moreover, the horizontal rectangles represent the execution of the corresponding process during a length of time given by the width of the rectangle. As it can be observed, certain periodic behaviors, mentioned in Section 2.2, are easily identifiable.

Considering the increase in the number of processors/cores on MPSoCs, the fact that each processor will trace its execution and that future dedicated hardware will allow a significant level of detail of the execution, it can be observed that the execution traces are going to

significantly increase in volume. It is already quite difficult to analyze current execution traces just by using visualization tools such as STLinux Trace Viewer, because of the huge amount of information already available. Therefore, in the near future, it is going to be very difficult to analyze execution traces manually.

Pattern mining techniques have already been used in multiple domains to extract useful information from big amounts of data. We consider that pattern mining techniques applied to the analysis of execution traces would reduce substantially the time needed to analyze them. Concretely, since this thesis is focused on the analysis of execution traces of multimedia applications and that multimedia applications present a somehow periodic behavior, in this thesis we propose to use periodic pattern mining techniques to analyze execution traces of multimedia applications in order to quickly diagnose execution errors.

In Chapter 3 we will present in detail the formal framework supporting the mining of periodic patterns on execution traces. But first, in the next section, we are going to introduce some basic concepts of pattern mining that will be helpful in order to understand the following chapters.

2.5 Pattern Mining

As part of existing data mining techniques, pattern mining techniques explore data blindly, i.e. find regularities on the data without any previous knowledge of the nature of the regularities. In the context of this thesis, pattern mining techniques offer a fine-grained description of the analyzed data, which is the level of detail that developers need to extract from execution traces in order to understand what happened during the execution.

Pattern mining techniques were first used on market basket analysis to identify sets of products that were often bought together by customers [AS94]. The algorithm proposed by Agrawal et al., called *Apriori*, was able to identify sets of items (products) *frequently* bought together over a set of customer transactions. In this context, a transaction was the set of items bought by a specific customer at a specific moment in time. This technique of pattern mining is known as *frequent itemset mining*.

At first, frequent itemset mining was applied to the market basket analysis, but the approach proposed by Agrawal *et al.* was generic, which allowed it to be applied to other problems. In our context, we have applied pattern mining techniques to the analysis of execution traces which requires a transformation of the input data (execution trace) into the input data format of pattern mining algorithms (transactional database). This can be done by splitting the trace into "windows" and then considering the events of the same window part of the same transaction of the database. A criteria used to split an execution trace might be a time interval as shown Table 2.2 which is the result of splitting the execution trace presented on Table 2.1 using a time interval of 10 ms.

Let's imagine that we are only interested in events that occurred together in at least two occasions, the following sets:

```
{mutex_lock}
{interrupt}
{context_switch}
```

Table 2.1: Example of an execution trace

```
317658 interrupt
317660 context_switch
317664 mutex_unlock
317669 mutex_lock
317673 context_switch
317675 print
317680 context_switch
317684 print
317692 mutex_lock
317693 context_switch
317695 mutex_unlock
317697 print
317698 mutex_lock
317702 interrupt
317705 context_switch
317706 mutex_unlock
```

Table 2.2: Sequence of sets of events

Time interval	Set of events
317658 ms - 317668 ms	interrupt, context_switch, mutex_unlock
317668 ms - 317678 ms	mutex_lock, context_switch, print
317678 ms - 317688 ms	context_switch, print
317688 ms - 317698 ms	mutex_lock, context_switch, mutex_unlock, print
317698 ms - 317708 ms	mutex_lock, interrupt, context_switch, mutex_unlock

```
{mutex_unlock}
{print}
{interrupt, mutex_unlock}
{interrupt, context_switch}
{mutex_lock, print}
{context_switch, print}
{mutex_lock, mutex_unlock}
{context_switch, mutex_unlock}
{mutex_lock, context_switch}
{interrupt, context_switch, mutex_unlock}
{mutex_lock, context_switch, print}
{mutex_lock, context_switch, mutex_unlock}
```

appear in at least two transactions, i.e. have a **support** of two transactions. This constraint is an input of frequent itemset mining algorithms called **minimum support threshold** (*min_sup*), as it is important to limit the number of results mined by the algorithms to the ones the user is interested in.

In order to generalize the example given in Table 2.2, let's consider $\{a = mutex\_lock, b = interrupt, c = context\_switch, d = mutex\_unlock, e = print\}$, and transform the sequence of sets of events into a **transactional database** (shown in Table 2.3). Such database is the input of frequent itemset mining algorithms. This way, any data that can be transformed to a transactional database can be analyzed by frequent itemset mining.

Table 2.3: Example of a transactional database

TID	Items
1	b c d
2	a c e
3	c e
4	a c d e
5	a b c d

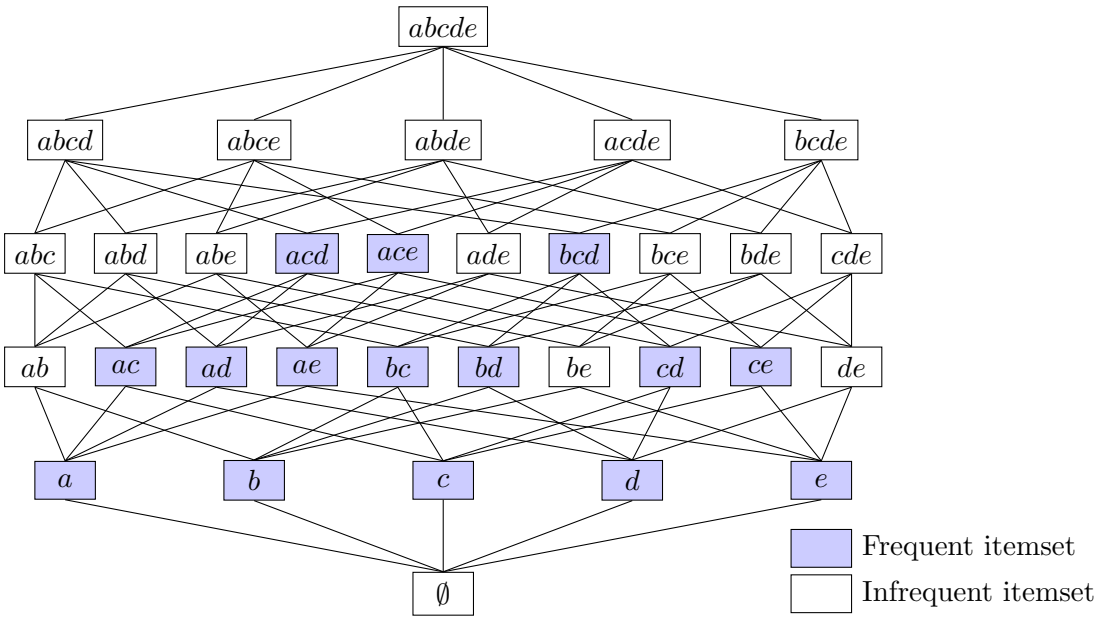


Figure 2.9: Spanning tree for the frequent itemset problem.

Enumerating all combinations of possible items of all possible lengths, called *candidate itemsets*, and then counting the number of appearances, called *support*, can take a long time, especially for big datasets. An example can be seen on Figure 2.9, which shows the spanning tree of the transactional database given on Table 2.2, where from 32 candidate itemsets only 15 are frequent. Therefore, each algorithm makes use of certain techniques and data properties on their enumeration strategy in order to reduce the computing time by avoiding the enumeration of candidate itemsets that cannot be frequent.

For example, *Apriori* is a level-wise search algorithm, meaning that the enumeration is carried out by generating all candidate itemsets of the same size on each step, and checking if each one of them is a frequent itemset, i.e. if it appears in at least min_sup transactions.

In order to avoid enumerating infrequent candidates, Agrawal *et al.* proposed the **anti-monotony property** that states that a superset of an infrequent itemset can not be frequent. For example, any superset of $\{de\}$ which is infrequent (frequency = 1) will be also infrequent: $\{ade\}$, $\{bde\}$ and $\{cde\}$ frequencies are respectively 1, 0 and 1, and are therefore infrequent.

Following the example presented on Table 2.3 with a minimum support threshold of two transactions, the set of frequent itemsets that an algorithm such as *Apriori* would mine is shown in Table 2.4.

As we can see in Table 2.4, the set of all frequent itemsets is highly redundant, e.g. knowing that the itemset $\{ac\}$ is frequent implies that the itemsets $\{a\}$ and $\{c\}$ are also frequent. This is due to the fact that frequent pattern mining algorithms are exhaustive which means that the number of results generated by these algorithms is normally large and can become difficult to analyze.

In order to reduce the result set, Pasquier et al. [PBTL99] proposed a condensed representation of the set of frequent itemsets, called **closed frequent itemset**. Put simply, a closed frequent itemset is a frequent itemset to which it is not possible to add any other item without changing its support. One of the advantages of mining closed frequent itemsets is that the result set is smaller but no information is lost, i.e. it is easy to know the support of any subset of a closed frequent itemset.

Table 2.4: Set of frequent itemsets

Frequent Itemsets	Support
$\{a\}$	3
$\{b\}$	2
$\{c\}$	5
$\{d\}$	3
$\{e\}$	3
$\{ac\}$	3
$\{ad\}$	2
$\{ae\}$	2
$\{bc\}$	2
$\{bd\}$	2
$\{cd\}$	3
$\{ce\}$	3
$\{acd\}$	2
$\{ace\}$	2
$\{bcd\}$	2

Table 2.5: Set of closed frequent itemsets

Closed Frequent Itemsets	Support
$\{c\}$	5
$\{ac\}$	3
$\{cd\}$	3
$\{ce\}$	3
$\{acd\}$	2
$\{ace\}$	2
$\{bcd\}$	2

The set of closed frequent itemsets of the example given in Table 2.3 is shown in Table 2.5. As explained before, $\{ce\}$ is a closed frequent itemset since it is not possible to add an item without changing its support: $\{ace\}$'s support is 2, $\{bce\}$'s support is 0 and $\{cde\}$'s support is 1, while $\{ce\}$'s support is 3.

By comparing the set of frequent itemsets (Table 2.4) to the set of closed frequent itemsets (Table 2.5) we can observe that a considerable reduction of the result set is achieved by mining the set of closed frequent itemsets. Moreover, the support of frequent itemsets that are not part of the set of closed frequent itemsets can be deduced, e.g. the support of $\{bc\}$ is the same as the support of its superset $\{bcd\}$, otherwise $\{bc\}$ would be a closed frequent itemset.

Pasquier *et al.* also proposed an algorithm to mine closed frequent itemset based on *Apriori* algorithm. Their algorithm generates frequent patterns and then obtains the closed frequent patterns by applying a closure operator. They reduce the computational cost of the algorithm by generating frequent key patterns instead of frequent patterns, but this set can still be exponential in the set of closed patterns. Moreover, in order to avoid generating duplicates, the algorithm needs to keep in memory the set of closed frequent patterns, which

can be very expensive and weakens the scalability of the algorithm.

Other algorithms such as CLOSET [PHM00], CHARM [ZjH02] and LCM [UAUA04] try to solve this problem by using a depth-first enumeration strategy, in order to make their algorithms more efficient. The depth-first enumeration strategy explores a branch until no more closed frequent itemsets can be generated on that branch, in which case the algorithm goes back to the next unexplored branch.

An important aspect of the depth-first enumeration strategy is the generation of duplicates. As it can be observed on Figure 2.9, several branches of the spanning tree arrive to the same itemset, e.g. $\{ab\}$ can be reached by adding the item b to the itemset $\{a\}$ or by adding the item a to the itemset $\{b\}$. Therefore, a depth-first enumeration strategy needs to ensure that every pattern is generated only once. A solution would be to remember all closed frequent itemsets that have been generated, but with big datasets containing a big number of closed frequent itemsets this solution becomes inefficient.

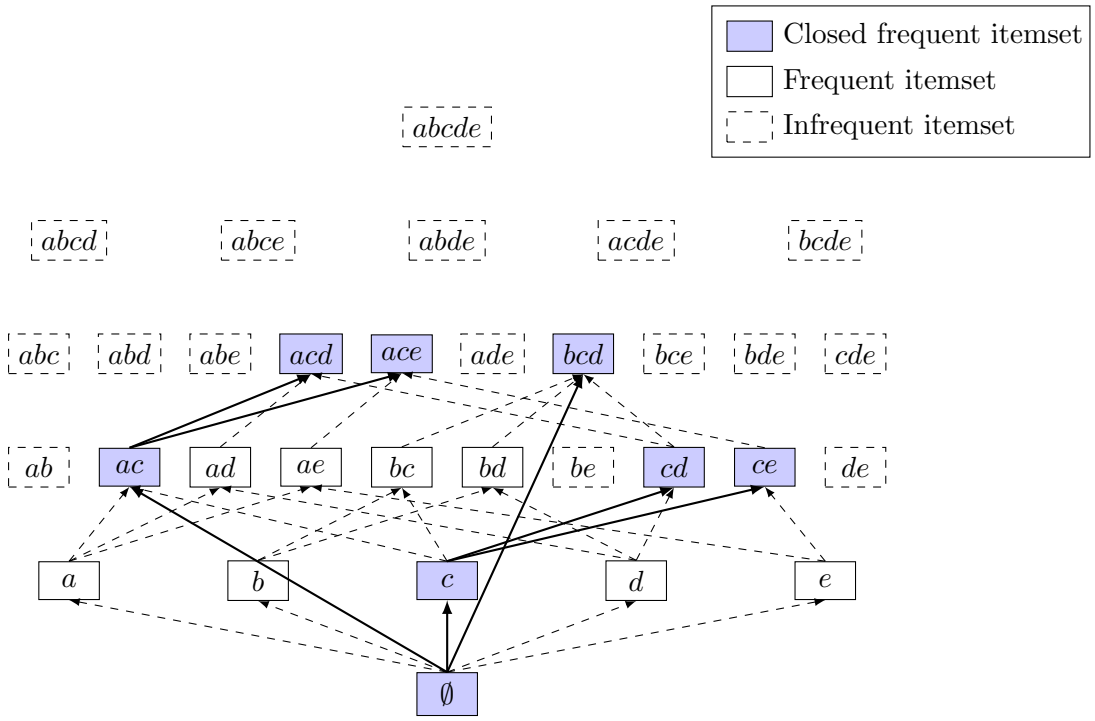


Figure 2.10: Closed frequent itemset enumeration tree.

Uno *et al.* [UAUA04] propose a test that says whether it is the first time a closed frequent itemset has been generated, which is called the **first parent test**. The *first parent* of a closed frequent itemset Q is the closed frequent itemset $P \subset Q$ that is its parent in the enumeration tree. For instance, in Figure 2.10 $\{ac\}$ is the first parent of $\{acd\}$ and $\{ace\}$. Indeed, in Figure 2.10, all solid lines represent first parent relationships.

Also, to avoid enumerating frequent itemsets that are not closed frequent itemsets, LCM algorithm uses a closure operator to enumerate only closed frequent itemsets. The closure operator associates a frequent itemset P with its closed frequent itemset Q such that $P \subseteq Q$ and $D[P] = D[Q]$, i.e. both P and Q are present in the same transactions of the dataset.

2.6 Conclusions

In this chapter, we have reviewed the evolution of the MPSoCs used on consumer electronics motivated by the user demand for more computational power, in order to carry out more multimedia applications, generally very computationally heavy. But, by increasing the parallelism of these architectures, with the introduction of multi-core and many-core processors, the development of applications has become a difficult task.

We have also seen how the software architecture of multimedia applications takes advantage of this new parallelism by splitting tasks on different modules connected in the form of a pipeline. When carrying out several operations at the same time, such as decoding two data streams simultaneously in order to have one on the television and another being recorded on the set-top box hard drive, several pipelines are working in parallel. Moreover, the decoding of a frame can be also parallelized by splitting the frame into small pieces that can be treated in parallel. A bug or a performance problem on these situations is not easy to diagnose and solve.

Therefore, we have seen how tracing gives the right level of detail that allows developers to analyze the execution of multimedia applications in detail. The drawback of this technique is the size of the traces which can become impossible to analyze manually, which will drive the need for using automatic analysis tools such as pattern mining.

Finally, we have introduced certain basic concepts of pattern mining that will be used and extended on this thesis.

The rest of this thesis is divided into three parts. First, we present our contributions on periodic pattern mining. Second, we present our contributions on debugging of multimedia applications on embedded systems by applying our pattern mining approach to the analysis of execution traces. And third, we present the related works, the conclusions and perspectives.

Part I

Pattern Mining Contribution

Pattern mining algorithms are based on a formal framework that mathematically defines the search space as well as the properties of the searched pattern. Indeed, the contributions of this thesis on data mining, presented in this part, are a formal framework for periodic pattern mining as well as an efficient algorithm to mine a condensed representation of the set of frequent periodic patterns from transactional databases. In the next part of the thesis, these contributions are applied to the analysis of execution traces of multimedia applications since it is the context of this thesis. Nevertheless, these contributions could be applied to other contexts such as production computer networks [MH01], web access log analysis [YWY03], study of protein sequences [HC04] and so on.

In Chapter 3, we present a formal framework that defines the frequent periodic patterns to be mined as well as a condensed representation of the set of frequent periodic patterns and some properties of this condensed representation. Then, in Chapter 4, we present an efficient algorithm that makes use of the properties defined in Chapter 3 to mine the condensed representation of the set of frequent periodic patterns without enumerating the whole set of frequent periodic patterns. Moreover, this algorithm presents a polynomial delay time complexity and a polynomial space complexity, which make the algorithm scalable in terms of mining time and memory use.

3 Core Periodic Concepts

The objective of this thesis is to propose new trace analysis techniques for debugging multimedia applications. The nature of multimedia applications, whose execution turns around the treatment of frames, made us choose periodic pattern mining to analyze the execution traces. In this chapter, we present a formal framework that allows us to address the problem of frequent periodic pattern mining.

Concretely, in Section 3.1, we present a definition of frequent periodic pattern, very similar to the one Ma et al. presented in [MH01]. Based on itemset mining, a periodic pattern consist in an itemset, i.e. a set of items with no ordering imposed, that is found periodically (every n transactions). The presented definition allows unrestricted-sized gaps in the periodicity, i.e. parts of the dataset where the itemset is not found where expected. Also, since the periodicity of the patterns is not known in advance, all possible periods are taken into account. This "freedom" allows us to discover different kind of behaviors of different parts of the software, and what is more important, where these periodic behaviors are disturbed, which present themselves as gaps in the periodicity. But this also means that the definition is not very restrictive, thus the corresponding mining process can generate a huge amount of patterns.

As we will see in this chapter, the set of frequent periodic patterns is highly redundant, in terms of itemsets but also in terms of periods. Thus, in Section 3.2, we introduce the triadic approach taken in order to solve the part of the redundancy problem related to the itemsets. Then, in Section 3.3, we propose a condensed representation of the set of frequent periodic patterns called Core Periodic Concepts (CPC) that solves the part of the redundancy problem related to the periods. Finally, in Section 3.4, we present certain connectivity properties of Core Periodic Concepts that allow us to implement an efficient algorithm for mining Core Periodic Concepts (CPC).

3.1 Frequent Periodic Pattern

Let $\mathcal{I}$ be the set of all items, i.e. $\mathcal{I} = \{i_1, i_2, ..., i_r\}$, a **dataset** $\mathcal{D}$ is an ordered set of transactions $\{t_1, t_2, ..., t_n\}$ where each transaction is a subset of $\mathcal{I}$, i.e. $t_k \subseteq \mathcal{I}$ for $1 \leq k \leq n$, and where the order is defined by $t_i < t_j$ if and only if $i < j$.

We use the following notations regarding the dataset $\mathcal{D}$:

- $|\mathcal{D}|$ denotes the number of transactions in $\mathcal{D}$.
- $||\mathcal{D}|| = \sum_{i=1}^{|\mathcal{D}|} |t_i|$ denotes the size of $\mathcal{D}$.

Table 3.1: Example dataset

t_k	Itemset	t_k	Itemset	t_k	Itemset
t_1	a, b	t_5	a, b	t_9	k, l
t_2	c, d	t_6	g	t_{10}	a, b
t_3	a, b	t_7	h, i		
t_4	e, f	t_8	a, b, j		

An itemset X is denoted by $\{x_1, x_2, ..., x_j\}$ where x_t is an item for $1 \leq t \leq j$, i.e. $x_t \in \mathcal{I}$. Considering $X \subseteq I$, we say that an itemset X *occurs* in the transaction t_k if and only if $X \subseteq t_k$.

Given a transaction t_k, its transaction identifier, denoted as $tid(t_k)$, is its position in the dataset, i.e. k. We also define the distance d between two transactions t_i and t_j as the difference between their transaction identifiers, i.e. $d = j - i$ with $i \leq j$.

When an itemset occurs over a set of transactions and the distance between any two consecutive transactions is constant, this set of transactions forms a **cycle**.

Definition 3.1 (Cycle)**.**

Given an itemset X and a period p, a cycle of X, denoted ***cycle***(X, p, o, l)*, is a maximal set of l transactions in $\mathcal{D}$ containing X, starting at transaction t_o and separated by equal distance p:*

$$cycle(X, p, o, l) = \{t_k \in \mathcal{D} \mid 0 \leq o < |\mathcal{D}|, X \subseteq t_k, k = o + p * i, 0 \leq i < l, X \nsubseteq t_{o-p}, X \nsubseteq t_{o+p*l}\} \tag{3.1}$$

Example: *In the dataset in Table 3.1, the itemset $X = \{a, b\}$ is found at a period $p = 2$ on transactions from t_1 ($o = 1$) to t_5, therefore it forms a cycle of length $l = 3$, denoted $cycle(\{a, b\}, \mathbf{2}, 1, 3) = \{t_1, t_3, t_5\}$. Note: Periods are presented in bold for clarity.*

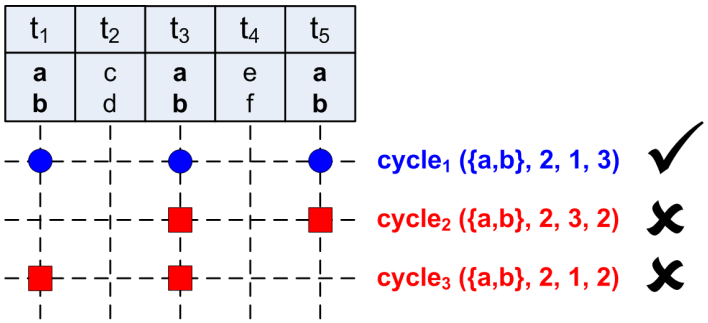


Figure 3.1: Examples of cycles and non-cycles

Figure 3.1 offers a graphical representation of a set of transactions where $cycle_1(\{a, b\}, \mathbf{2}, 1, 3)$ is *maximal*. $cycle_2(\{a, b\}, \mathbf{2}, 3, 2)$ and $cycle_3(\{a, b\}, \mathbf{2}, 1, 2)$ are not maximal since they

can be extended with transactions t_1 and t_5 respectively, and therefore they are not valid cycles in our context.

A set of consecutive cycles (the end of a cycle happens before the beginning of the following cycle) over the same itemset and the same period forms a **periodic pattern**. For simplicity of notation, each cycle of the set of cycles of a periodic pattern is represented by its origin o and its length l, i.e. by the tuple (o, l), since all cycles in the set share the same itemset and period.

Definition 3.2 (Periodic Pattern)**.**

*An itemset X together with a set of cycles C and a period p form a **periodic pattern**, denoted $P(X, p, s, C)$, if the set of cycles $C = \{(o_1, l_1), ..., (o_k, l_k)\}$, with $1 \leq k \leq m$ and m being the maximum number of cycles of period p in the dataset $\mathcal{D}$, is a set of cycles of X such that:*

1. *All cycles have the same period p.*
2. *All cycles are consecutive:*
   $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $1 \leq i < j \leq k$, we have $o_i < o_j$.
3. *Cycles do not overlap:*
   $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $i < j$, we have $o_i + (p * (l_i - 1)) < o_j$.

*With s being the **support** of the periodic pattern, i.e. the sum of all cycle lengths in $C = \{(o_1, l_1), ..., (o_k, l_k)\}$, calculated with the formula*

$$s = \sum_{i=1}^{k} l_i \tag{3.2}$$

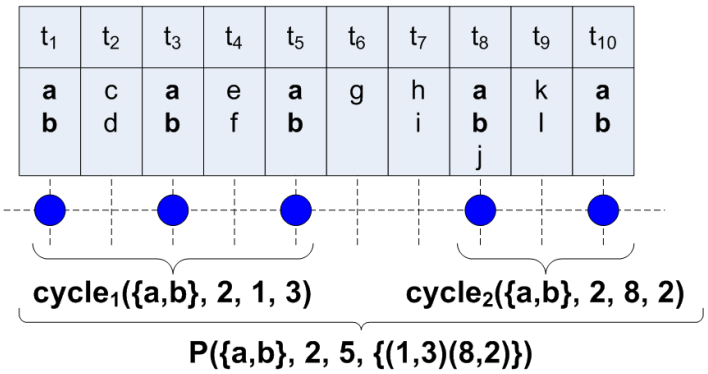


Figure 3.2: Periodic pattern formation

Example: *In Figure 3.2 we can observe that the cycles $cycle_1(\{a, b\}, \mathbf{2}, 1, 3)$ and $cycle_2(\{a, b\}, \mathbf{2}, 8, 2)$ form a periodic pattern $P = (\{a, b\}, \mathbf{2}, 5, \{(1, 3)(8, 2)\})$ with period 2 and a support of 5 transactions.*

It is important to note that we do not impose any restrictions on the length of the gap allowed between two cycles of a periodic pattern. By making this choice we are able to have irregular gaps in the periodicity.

We now introduce the notion of **frequent periodic patterns**.

Definition 3.3 (Frequent Periodic Pattern)**.**

*Given a minimum support threshold min_sup, a periodic pattern P is **frequent** if its support is greater than min_sup, i.e. $P(X, p, s, C)$ is frequent if and only if $s \geq min\_sup$.*

Example: *Given the dataset shown in Table 3.1 and a minimum support of two transactions ($min\_sup = 2$), the set of frequent periodic patterns is presented in Table 3.2. Note: Only periods not greater that the length of the dataset divided by min_sup are considered.*

Table 3.2: Set of frequent periodic patterns

Frequent Periodic Patterns	
$P_1(\{a\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_9(\{a,b\}, \mathbf{4}, 2, \{(1,2)\})$
$P_2(\{b\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_{10}(\{a\}, \mathbf{5}, 2, \{(3,2)\})$
$P_3(\{a,b\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_{11}(\{b\}, \mathbf{5}, 2, \{(3,2)\})$
$P_4(\{a\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{12}(\{a,b\}, \mathbf{5}, 2, \{(3,2)\})$
$P_5(\{b\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{13}(\{a\}, \mathbf{5}, 2, \{(5,2)\})$
$P_6(\{a,b\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{14}(\{b\}, \mathbf{5}, 2, \{(5,2)\})$
$P_7(\{a\}, \mathbf{4}, 2, \{(1,2)\})$	$P_{15}(\{a,b\}, \mathbf{5}, 2, \{(5,2)\})$
$P_8(\{b\}, \mathbf{4}, 2, \{(1,2)\})$	

As we can see in Table 3.2, the set of frequent periodic patterns is highly redundant. On one hand, all combinations of large itemsets are consider as patterns. For example, P_1, P_2 and P_3 have the same occurrences, and P_1 and P_2's itemsets are subsets of P_3's itemset.

On the other hand, combinations of small periods by addition or multiplication generate redundant periods. For example, all occurrences of P_9 are included in P_3's occurrences and P_9's period **4** is multiple of P_3's period **2**. Since they have the same itemset, we can consider that P_3 is more representative that P_9.

In real datasets tens of thousands of frequent periodic patterns might be found, which are impractical to analyze manually. In order to produce a condensed representation of the set of frequent periodic patterns we adopt a triadic approach by introducing the periods into the dataset.

3.2 Triadic approach to Periodic Pattern Mining

The dataset introduced in Section 3.1, corresponds to a relation between two attributes, *items* $\times$ *transactions*, i.e. $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{D}$ and each $r \in \mathcal{R}$ can be represented by a couple $r = \{(i,t) | i \in \mathcal{I}, t \in \mathcal{D}\}$, denoting that the item i occurs on transaction t.

In order to mine periodic patterns, the period should be included in the dataset, but in order to do so, the binary relation $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{D}$ has to be transformed into a ternary relation $\mathcal{Y} \subseteq \mathcal{I} \times \mathcal{P} \times \mathcal{D}$, with $\mathcal{P}$ the set of all possible periods that we limit to the range $[1..|\mathcal{D}|/min\_sup]$.

Ternary relations have been studied by Formal Concept Analysis through a *triadic approach* [LW95]. The concepts of periodic triadic context and periodic concepts are presented here including some modifications needed in order to adapt them to periodic pattern mining.

Definition 3.4 (Periodic Triadic Context)**.**

*A **periodic triadic context** is defined as a quadruple $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ where $\mathcal{I}$ is the set of items, $\mathcal{P}$ is the set of periods, $\mathcal{D}$ is the set of transactions, and $\mathcal{Y}$ is a ternary relation between $\mathcal{I}$, $\mathcal{P}$ and $\mathcal{D}$, i.e. $\mathcal{Y} \subseteq \mathcal{I} \times \mathcal{P} \times \mathcal{D}$.*

An element of the relation $y \subseteq \mathcal{Y}$ is denoted by the triple $y = \{(i, p, t) | i \in \mathcal{I}, p \in \mathcal{P}, t \in \mathcal{D}\}$ and is read: the transaction t forms part of a cycle of period p of the item i.

Table 3.3: Representation of the ternary relation $\mathcal{Y}$

$\mathcal{I}/\mathcal{P}-\mathcal{T}$	2										3									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
a	×		×		×			×		×					×			×		
b	×		×		×			×		×					×			×		
...																				

$\mathcal{I}/\mathcal{P}-\mathcal{T}$	4										5									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
a	×				×								×		×			×		×
b	×				×								×		×			×		×
...																				

Example: *Given the dataset shown in Table 3.1, the corresponding periodic triadic context is shown in Table 3.3. Each cross in the table represents an element of the ternary relation $\mathcal{Y}$. For example, $P_1(\{a\},\mathbf{2},5,\{(1,3)(8,2)\})$ in Table 3.2 is transformed into the triples $(a,2,t_1),(a,2,t_3),(a,2,t_5),(a,2,t_8),(a,2,t_{10})$ shown by the corresponding crosses in Table 3.3. For simplicity, items not forming any cycle of any possible period are not shown on the table.*

Definition 3.5 (Frequent Triple)**.**

*Given a minimum support threshold min_sup, a triple (I,P,T), with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$, $T \subseteq \mathcal{D}$ and $I \times P \times T \subseteq \mathcal{Y}$, is **frequent** if and only if $I \neq \emptyset$, $P \neq \emptyset$ and $|T| \geq min\_sup$.*

Example: *In Table 3.3, given a minimum support threshold of two transactions ($min\_sup = 2$), we can observe several frequent triples such as $(\{a\},\{2,4\},\{t_1,t_5\})$ or $(\{a,b\},\{2\},\{t_1,t_3,t_5\})$, since the number of transactions forming those triples is greater or equal to 2.*

The set of frequent triples is highly redundant since it includes all possible combinations between items, periods and transactions included in the ternary relation $\mathcal{Y}$. A *lossless* reduced representation of the set of triples was introduced by Wille [Wil95] and named *triadic concepts*. Saying the representation is lossless means that it is possible to reconstruct the set of triples from the set of triadic concepts without any extra information.

Here we adapt this definition to periodic pattern mining, called **periodic concept**, by saying that the set of periodic concept is a lossless condensed representation of the set of frequent triples, i.e. triples such that the size of their transaction list is greater or equal than a given minimum support threshold.

Definition 3.6 (Periodic Concept)**.**

*A **periodic concept** of a periodic triadic context $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ is a **frequent** triple (I, P, T) with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$ and $T \subseteq \mathcal{D}$, such that none of its three components can be enlarged without violating the condition $I \times P \times T \subseteq \mathcal{Y}$.*

Example: *In Table 3.4, we can observe the set of periodic concepts extracted from the set of frequent triples obtained from the dataset shown in Table 3.3. The triples forming this set are periodic concepts since it is not possible to extend any of the attributes of the triple without violating the relation $\mathcal{Y}$.*

Table 3.4: Set of periodic concepts

Periodic Concepts
$T_1(\{a,b\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$
$T_2(\{a,b\}, \{2,4\}, \{t_1, t_5\})$
$T_3(\{a,b\}, \{2,5\}, \{t_3, t_5, t_8, t_{10}\})$
$T_4(\{a,b\}, \{2,3,5\}, \{t_5, t_8\})$
$T_5(\{a,b\}, \{2,3,4,5\}, \{t_5\})$

It can be observed that the set of periodic concepts is a lossless representation of the set of frequent periodic patterns, even if they are presented using a different notation. Moreover, it is important to note that the set of periodic concepts presented on Table 3.4 is considerably smaller than the set of frequent periodic patterns presented in Table 3.2.

The set of periodic concepts can be translated into a set of frequent periodic patterns by calculating the cycles included in the set of transactions of each periodic concept. For instance, $T_2(\{a,b\}, \{2,4\}, \{t_1, t_5\})$ contains only one cycle of period 4 with transactions t_1 and t_5 which would give us the periodic pattern $P_9(\{a,b\}, \mathbf{4}, 2, \{(1,2)\})$ from Table 3.2. Moreover, it is possible to deduce as well P_7 and P_8 since their itemsets are subsets of $\{a,b\}$.

Nevertheless, the set of periodic concepts still contains redundant information in terms of redundant periods. For example, considering the periodic concepts T_1 and T_2 in Table 3.4, we can see that T_2 is "included" in T_1, i.e. they have the same itemset and the transactions belonging to T_2 are a subset of the transactions belonging to T_1. Also, period **4** of T_2 can be "deduced" from the set of transactions of T_1, as seen above. As a result, T_2 can be removed without losing information. The same logic can be applied to T_3, T_4 and T_5, reducing the set to pattern T_1.

In order to obtain a condensed representation of the set of periodic concepts, we propose removing periodic concepts with redundant periods. For this, we present here the definition of *core periodic concept* which allows us to extract the set of periodic concepts that does not contain redundant periods.

3.3 Core Periodic Concepts

In this section, we present the core periodic concepts and prove that the set of core periodic concepts is a **condensed representation** of the set of frequent periodic patterns. A condensed representation of a set of frequent patterns, is a subset of the set of frequent pat-

terns, such that it is possible to efficiently derive the whole set of frequent patterns from it [MT96]. In the context of pattern mining, efficiently means without accessing the dataset $\mathcal{D}$. The **Core Periodic Concepts** is a subset of the periodic concepts, filtering out periodic concepts that have redundant periods.

Definition 3.7 (Core Periodic Concept)**.**

*A periodic concept (I, P, T) is a **Core Periodic Concept** if there does not exist any other periodic concept (I', P', T') such that $I = I'$, $P' \subset P$ and $T' \supset T$.*

Example: *In Table 3.5, we can observe the set of CPCs extracted from the set of periodic concepts shown in Table 3.4. For instance, $T_2(\{a, b\}, \{2, 4\}, \{t_1, t_5\})$ is not a CPC since there exists $T_1(\{a, b\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$ with the same itemset $\{a, b\}$, a smaller set of periods $\{2\} \subset \{2, 4\}$ and a bigger set of transactions $\{t_1, t_3, t_5, t_8, t_{10}\} \supset \{t_1, t_5\}$.*

Table 3.5: Set of core periodic concepts

Core Periodic Concepts
$M_1(\{a, b\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$

It is important to note that the set of core periodic concepts shown in Table 3.5 is considerably smaller than the set of periodic concepts shown in Table 3.4, and therefore smaller than the set of frequent periodic patterns shown in Table 3.2, and that it does not contain redundant periods.

The set of core periodic concepts can be translated into a set of periodic concepts by calculating the bigger periods formed by the transactions of each core periodic concept. For instance, from M_1 in Table 3.5 we can obtain periods 2 ($\{t_1$,t_3,t_5,t_8,$t_{10}\}$), 3 ($\{t_5$,$t_8\}$), 4 ($\{t_1$,$t_5\}$) and 5 ($\{t_3$,t_5,t_8,$t_{10}\}$). Then it is possible to calculate the maximal subsets involving several periods on the same set of transactions. For example,t_1 and t_5 are found in periods 2 and 4 generating the periodic concept $T_2(\{a, b\}, \{2, 4\}, \{t_1, t_5\})$ from Table 3.4. The same process can be used to deduce T_3, T_4 and T_5.

In order to fit the periodic pattern notation introduced in Definition 3.2, the set of core periodic concepts should be post-processed. For each core periodic concept, a set of frequent periodic patterns is generated where each periodic pattern contains a period from the set of the periods, and the set of corresponding cycles, calculated from the set of transactions. In the example, from Table 3.5 which contains only one minimal periodic generator $M_1(\{a, b\}$, $\{2\}$, $\{t_1$,t_3,t_5,t_8,$t_{10}\})$, we obtain the frequent periodic pattern $P(\{a, b\}, \mathbf{2}, 5, \{(1, 3)(8, 2)\})$, which corresponds to P_3 from Table 3.2.

The set of core periodic concepts is a lossless representation of the set of frequent periodic patterns since:

- The set of periodic concepts can be deduced from the set of core periodic concepts, as explained above, and,
- The set of frequent periodic patterns can be deduced from the set of periodic concepts, as explained in Section 3.2.

3.4 Connectivity property of Core Periodic Concepts

The first contribution of this thesis, i.e a condensed representation of the set of frequent periodic patterns named Core Periodic Patterns, has been presented in [LCBT$^+$12]. In this paper, we named "Core Periodic Patterns" as "Minimal Periodic Generators", but the term "Minimal Periodic Generator" might be confusing as our representation properties differ from so-called generator/free/key itemsets [CG02] [Bou05]. Therefore, we correct our naming in this thesis. In this paper, we also presented a basic algorithm to mine core periodic concepts. This algorithm worked in three phases: first, it generated all frequent triples from a dataset; second, using Data-Peeler [CBRB09] it generated the set of periodic concepts; and last, it extracted the set of core periodic concepts from the set of periodic concepts.

The advantage of this algorithm is that is easy to understand since it follows the logic we have followed while presenting the definitions in this chapter. Nevertheless, this algorithm presents poor scaling capabilities since it generates the whole set of triples, and then the whole set of periodic concepts to mine the set of core periodic concepts. Moreover, these sets are loaded into memory from step to step which, due to the combinatorial nature of items and periods, can use a big amount of memory.

A more efficient algorithm would be a level-wise search algorithm that enumerates items and periods to mine the set of core periodic concepts. However, this algorithm would need to generate the set of periodic concepts in order to individually check whether the current periodic concept is a core periodic concept. Thus, in order to be able to say whether a periodic concept $C(X, P, T)$ is a core periodic concepts, the candidate set of core periodic concepts $CCPP$ needs to be kept in memory. This way, if there exist a candidate CPC $B(X', P', T') \in CCPP$ such that $X = X'$, $P' \subset P$ and $T' \supset T$, then C is not a CPC. Otherwise, C can be added to the set $CCPP$. Likewise, if there exists a candidate CPC $B(X', P', T') \in CCPP$ such that $X = X'$, $P \subset P'$ and $T \supset T'$, then B is not a CPC and it can be replaced by C in $CCPP$.

Therefore, at least the candidate set of core periodic concepts needs to be kept in memory during the algorithm. The number of core periodic concepts, being closely related with the set of frequent itemsets, can be exponential on the number of possible items, which makes this solution not scalable in terms of memory usage [AU09].

Therefore, in the rest of this section, we contribute an in depth analysis of Core Periodic Patterns, and show that to determine if a periodic concept is a CPC, it is sufficient to perform a simple test on its transaction list, without needing access to of any other CPC. Moreover, we define a necessary condition on the CPC's transaction list that will help guide an enumeration algorithm to mine CPCs efficiently.

In periodic pattern mining, there is a strong relationship between the periods and the transactions. Transactions belonging to the same cycle will be spaced by a multiple of the period. Here, we define how to obtain the corresponding list of transactions from a given itemset and a given period, definition that will be used in the definition of the connectivity properties.

Definition 3.8 (Transaction List)**.**

The list of transactions corresponding of a given itemset X and a given period p, denoted $tidlist(X, p)$, returns the list of all t transactions such that $(X, p, t) \in \mathcal{Y}$.

This definition characterizes a very important property of the Core Periodic Concepts that is going to allow us to know if a periodic concept is a CPC exclusively looking at its set of periods and the set of transactions associated to each of the periods.

Theorem 3.9. *A periodic concept (X, P, T) is a CPC if and only if $\forall p \in P$ it is true that $tidlist(X, p) = T$.*

Proof:

1. *A periodic concept (X, P, T) is a CPC if $\forall p \in P$ $tidlist(X, p) = T$. Proving the theorem by contradiction.*
   Having into account that (X, P, T) is a CPC, let's consider that there exists $p' \in P$ such that $tidlist(X, p') \supset T$. It is not $tidlist(X, p') \neq T$ because $tidlist(X, p') \subset T$ is not possible by definition of periodic concept (see definition 3.6). In that case, we can be sure that there exists a periodic concept (X, P', T') with $T' = tidlist(X, p')$ and $p' \in P' \subset P$, such that $\forall p'' \in P'$ $tidlist(X, p'') = T'$. By construction we have that $T' \supset T$ and $P' \subset P$, so (X, P', T') is a CPC and (X, P, T) is not (see definition 3.7), which is a contradiction.

2. *Given a periodic concept (X, P, T) such that $\forall p \in P$ $tidlist(X, p) = T$, then (X, P, T) is a CPC. Proving the theorem by contradiction.*
   If (X, P, T) is not a CPC is because there exist another periodic concept (X, P', T'), which is a CPC, such that $P' \subset P$ and $T' \supset T$. As proved above, if (X, P', T') is a CPC then $\forall p' \in P'$ we have that $tidlist(X, p') = T'$, and therefore $tidlist(X, p') \supset T$. On the other hand, $p' \in P'$ and $P' \subset P$, which means that $p' \in P$ and, following the hypothesis, we have that $tidlist(X, p') = T$, which is a contradiction of the previous statement.

Theorem 3.9 means that to check if a periodic concept is a CPC, it is not necessary to compare it with any other periodic concept or CPC. This allows for an online checking of CPC status of a periodic concept found during search space exploration, removing the need for a post-processing step. However, search space exploration would still need to enumerate all periodic concepts.

To solve this, we introduce a notion of *connectivity* in the transactions of a transaction list, that will allow to prune earlier in the search space exploration periodic concepts that will not lead to CPC. The following proposition will allow us to generate a reduced set of triples directly related to the final set of core periodic concepts. From this reduced set of triples, we are going to generate a reduced set of periodic concepts. Then each one of the periodic concepts will be checked, using Theorem 3.9, to know whether it is a core periodic concept or not.

Definition 3.10 (Full Connectivity)**.**

A triple (X, P, T) *is fully connected if* $\forall p \in P$ *and* $\forall t \in T$ *there exist another* $t' \in T$ *such that the distance between* t *and* t' *is equal to* p*.*

Proposition 3.11. *A Core Periodic Concept is fully connected.*

Proof: (Proof by contradiction) Let's suppose that a CPC (X, P, T) is not fully connected. Then, there exist a period $p \in P$ and a transaction $t \in T$ such that $(X, p, t) \in \mathcal{Y}$, but there does not exist any $t' \in T$ such that the distance between t and t' is equal to p. In that case, we have that $t' \in tidlist(X, p)$ but $t' \notin T$, therefore $tidlist(X, p) \supset T$ which contradicts Theorem 3.9.

In the next section, we are going to present an efficient algorithm that directly mines the set of Core Periodic Concepts without generating the whole set of periodic concepts by making use of the properties presented above.

4

Core Periodic Concept Mining Algorithm

In Chapter 3, we have presented a condensed representation of the set of frequent periodic patterns called Core Periodic Concepts (CPC). This definition is based on a ternary relation between the set of items, the set of periods and the set of transactions. Generally, ternary relations need to enumerate independently each attribute of the relation, as is done in DataPeeler algorithm [CBRB09]. Nevertheless, in our context the set of periods and the set of transactions of a core periodic concept are strongly dependent. It is thus possible to propose a more efficient enumeration strategy based on the itemset part of the triples that exploits the connexion between periods and transactions. Such strategy can benefit from recent advances in closed itemset enumeration.

In this chapter, we present an efficient algorithm, called PerMiner, to mine the set of Core Periodic Concepts from a transactional database in Section 4.1. Then, in Section 4.2, we present the analysis of the complexity of PerMiner algorithm. We introduce a parallel version of PerMiner algorithm, presented in Section 4.3, which can exploit the parallelism of multi-core processors in order to substantially reduce the computational time. Finally, in Section 4.4, we prove the soundness and completeness of PerMiner algorithm, and we prove that PerMiner does not generate duplicates.

4.1 PerMiner Algorithm

PerMiner algorithm is a depth-first search algorithm, inspired by the works of Arimura and Uno on pattern enumeration [UAUA04, AU09] which define the properties a closed pattern enumeration algorithm should have in order to present polynomial delay time and polynomial space complexity.

Therefore, in this section, we are going to analyze the techniques, proposed by Arimura and Uno, that allow a tree-shaped enumeration of the lattice of closed itemsets. Then, we are going to explain why these techniques are not immediately applicable to core periodic concepts. However, thanks to the strong relationship between the periods and transactions of a CPC, we will show that it is possible to reuse these techniques on the enumeration of the itemset part of a CPC and extend them with the computation of frequent periods and their transaction sets.

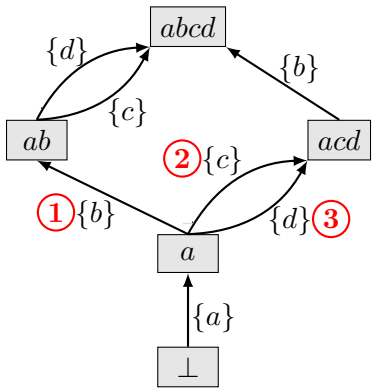
In order to explain PerMiner algorithm we are going to show how the closed itemset enumeration of LCM algorithm, proposed by Uno et al. [UAUA04], works. LCM algorithm is an itemset mining algorithm that makes use of the enumeration techniques proposed by Arimura and Uno. As said above, PerMiner algorithm is going to enumerate only the itemset part of CPCs, thus its structure is going to be based on LCM's structure. Therefore, we are going to explain LCM and then briefly explain what extra techniques are necessary to generate the period and transaction sets of the CPCs. Finally, we will explain PerMiner algorithm in detail and illustrate this explanation with an example.

Definition 4.1 (Augmentation)**.** *Given an element e of the input dataset, a pattern Q is an augmentation of a pattern P if $Q = P \cup \{e\}$.*

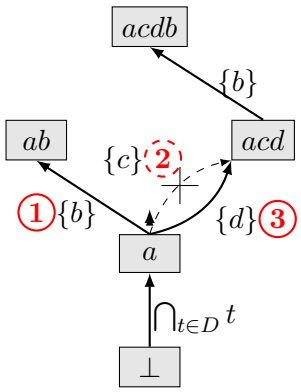
The pattern enumeration technique used by Arimura and Uno consists on a depth-first enumeration starting from the empty set. Algorithms using this technique make use of a recursive function. This recursive function tries to expand the pattern given as input using all possible augmentations. For each augmentation, a closed pattern is generated, then the recursive function is invoked using the generated pattern as input. As several different patterns can lead to the same closed pattern, done naively this enumeration technique creates a enumeration of the lattice of closed patterns in the shape of a directed acyclic graph (DAG), as can be seen in Figure 4.1 (b).

TID	Itemset
t_1	a b f
t_2	a b c d
t_3	a c d
t_4	a b c d
t_5	a b g

(a) Dataset



(b) DAG-shaped enumeration



(c) Tree-shaped enumeration

Figure 4.1: **First Parent Test:** From DAG-shaped enumeration to tree-shaped enumeration.

We can observe that the closed itemset $\{abcd\}$ can be generated by augmenting itemset $\{ab\}$ with both c and d, or by augmenting itemset $\{acd\}$ with b. Therefore, in order to avoid generating duplicates, the list of closed itemsets found so far needs to be kept in memory during the algorithm execution. The problem is that the number of closed itemsets may be exponential on the number of items, which makes the algorithm not scalable in terms of memory usage.

To avoid this, Arimura et al. [UAUA04] [AU09] introduced the concept of *first parent of a closed pattern*. The first parent of a closed pattern defines a unique pair (pattern, augmentation) that can generate this closed pattern. Concretely, Uno et at. [UAUA04] establish an arbitrary order on the enumeration that allows to easily test whether the current pattern with the given augmentation is the first parent of the generated closed pattern, and

that performs a tree-shaped enumeration of the lattice of closed patterns. A tree-shaped enumeration means that a closed pattern is generated by a unique augmentation of another closed pattern, as shown in Figure 4.1 (c).

Indeed, in LCM, the first parent test can be computed in polynomial time, accessing only the current pattern and the closed pattern, thus not needing to access any other patterns, which makes this solution of polynomial space complexity.

Algorithm 1: LCM

```
   Data: dataset D, minimum support threshold ε
   Result: Output all closed itemsets in D
1  begin
2      ⊥_clo ← ⋂_{t∈D} t
3      output ⊥_clo
4      D_{⊥_clo} = {t \ ⊥_clo | t ∈ D}
5      foreach e ∈ I s.t. e ∉ ⊥_clo do
6          expand(⊥_clo, e, D_{⊥_clo}, ε)
1  Function expand(P, e, D, ε)
2
       Data: Closed frequent itemset P, item e, reduced dataset D_P, minimum support
             threshold ε
       Result: Output all closed itemsets of the form P ∗ e
3      begin
4          if support_{D_P}({e}) ≥ ε then                    /* Frequency test */
5              P_ext := ⋂_{t∈D_P[{e}]} t                  /* Closure computation */
6              if maxItem(P_ext) = e then                   /* First parent test */
7                  Q = P ∪ P_ext
8                  output Q
9                  D_Q = {t \ Q | t ∈ D_P[{e}]}             /* Dataset Reduction */
10                 foreach i ∈ I s.t. i < e and i ∉ Q do   /* Itemset enumeration */
11                     expand(Q, i, D_Q, ε);
```

Since PerMiner is based on the enumeration technique used by Uno et al. in [UAUA04], the structure of PerMiner algorithm is very similar to the structure of LCM algorithm. Therefore, to better understand PerMiner algorithm, below we are going to explain LCM algorithm. Then, we are going to explain which techniques have been adapted or introduced by PerMiner algorithm in order to mine CPCs by enumerating only the itemset part of the CPC.

The pseudo-code of LCM presented in Algorithm 1 differs from the original version presented in [UAUA04]. For pedagogical purposes, the authors presented in [UAUA04] a simplified version of the algorithm with an enumeration technique less efficient than the one used on LCM implementation. Here, we are going to explain the version found in the implementation of LCM.

LCM algorithm starts computing the closure of $\perp$ in line 2, just in case there is an itemset

that occurs in all transactions in the dataset. Taking as an example the dataset presented in Figure 4.1 (a), the itemset $\{a\}$ is present in all transactions in the dataset, so $\perp_{clo} = \{a\}$.

Then, to start the enumeration, the recursive function *expand*, in line 2 of Algorithm 1, is called with the closure of $\perp$ and all possible items of the set of items $\mathcal{I}$ that do not belong to $\perp_{clo}$. Therefore, function *expand* receives a closed frequent itemset $P \subseteq \mathcal{I}$, an item e to augment it, the reduced dataset D_P of P and the minimum support threshold ε.

For example, considering $\perp_{clo} = \{a\}$, *expand* would be invoked with all items in $\mathcal{I}$ except a, i.e. $\{b, c, d\}$. The dataset passed to *expand* is reduced by removing the items in $\perp_{clo}$ from the dataset, i.e. $D_{\perp_{clo}} = \{\{b, f\}, \{b, c, d\}, \{c, d\}, \{b, c, d\}, \{b, g\}\}$. In fact, the step of reducing the dataset is necessary for a correct enumeration of the itemsets, as we will see later on.

Function *expand* first checks whether the augmentation e given as a parameter is frequent, i.e. checks if the item e occurs in at least ε transactions. If that is the case, the closure of the augmentation e is calculated by intersecting the transactions containing it.

For example, invoking *expand* with $P = \{a\}$, $e = b$ and $D_{\perp_{clo}}$, one can observe that the itemset $\{b\}$ is present in 4 transactions, which considering a minimum support threshold of 2 transactions means that the itemset $\{b\}$ is frequent. The intersection of the transactions containing b gives the itemset $\{b\}$ (1 in Figure 4.1(b)(c)). On the other hand, using the item c as an augmentation, the itemset $\{c\}$ is also frequent since it is present in 3 transactions and its closure is $\{c, d\}$ 2. Finally, using the item d as an augmentation, the itemset $\{d\}$ is also frequent since it is present in 3 transactions and its closure is $\{c, d\}$ 3.

Then, *expand* checks if $P \cup \{e\}$ is the first parent of Q, which consists in checking whether the maximal item of the closure of the augmentation pattern P_{ext} is equal to the item used as augmentation.

Following the example, the maximal item of the itemset $\{b\}$ 1 is b which is equal to its augmentation item b, therefore the itemset $\{b\}$ passes the first parent test. In the case of the itemset $\{c, d\}$ 2, its maximal item is d which is not equal to its augmentation item c, therefore it does not pass the first parent test. Finally, in the case of itemset $\{c, d\}$ 3, it passes the first parent test since its maximal item d is equal to its augmentation item d.

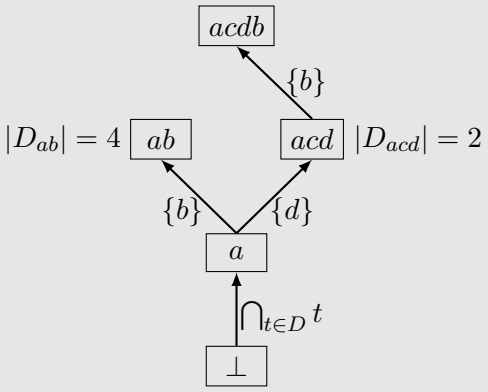
If the current itemset passes the first parent test, it is outputted by the algorithm. Then, its reduced dataset is calculated and used in the recursive calls. The reduced dataset consists in the transactions containing the closed itemset Q, except that items belonging to Q are removed from the reduced dataset.

Following the example, only the itemsets $\{b\}$ 1 and $\{d\}$ 3 passed the first parent test. Therefore, in the case of the itemset $\{b\}$ 1, the closed itemset Q, formed by union of the input closed itemset and the augmentation itemset, i.e. $Q = \{a\} \cup \{b\} = \{a, b\}$, is outputted. Then, its reduced dataset is calculated, i.e. $D_{a,b} = \{\{f\}, \{c, d\}, \{c, d\}, \{g\}\}$. In the case of the itemset $\{d\}$ 3, the closed itemset Q is equal to $\{a, c, d\}$. Then, Q is outputted and its reduced dataset is calculated, i.e. $D_{a,c,d} = \{\{b\}, \{b\}\}$.

LCM Itemset Enumeration Order

LCM imposes a particular order on the itemsets for enumeration. It is also supposed that there exists an arbitrary order on items which is set to decreasing frequency order. In our example dataset, the items already follow this order: a (freq:5) < b (freq:4) < c (freq:3) ≤ d (freq:3) < f (freq:1) ≤ g (freq:1). In the general case, a first step to reorder items is necessary in LCM, and also in our approach.

The order chosen for itemset enumeration is descending frequency order. Such order, together with the first parent test presented, allows to have very shallow enumeration branches for the most frequent itemsets. As the frequency and closure computation are the most costly for these itemsets, it is an efficient optimization.



For example, from the dataset in Figure 4.1(a), the figure on the right hand side of this text shows that the dataset of the itemset $\{a, b\}$ is bigger than the dataset of the itemset $\{a, c, d\}$. This is because the itemset $\{a, b\}$ has been obtained from augmentation b of the itemset $\{a\}$ which, according to the frequency order, is more frequent than the augmentation d, since b $(freq : 4) <$ d $(freq : 3)$. Therefore, it is less expensive, computationally speaking, to enumerate the itemset $\{a, b, c, d\}$ from an augmentation of itemset $\{a, c, d\}$.

As said above, the items belonging to the closed itemset Q are removed from the reduced dataset. Let's take as an example the augmentation of the closed itemset $\{a, c, d\}$ with the item b. If items a, c and d are not removed from the dataset, the dataset would be $D'_Q = \{\{a, b, c, d\}, \{a, b, c, d\}\}$. Then, in line 5 of *expand* function, P_{ext} would be equal to $\{a, b, c, d\}$. Since the maximal item in $\{a, b, c, d\}$ is d, which is different from the augmentation b, the first parent test would fail to generate the closed itemset $\{a, b, c, d\}$. To avoid this, the current closed itemset would have to be removed from the calculated P_{ext} or a more complex first parent test would need to be put in place. Therefore, to avoid losing efficiency, it is necessary to reduce the dataset before the recursive call.

For the recursive calls, only items smaller than the item used as augmentation are used to try to expand the current closed itemset. This is an important part of the enumeration technique that makes possible the already explained simple first parent test.

Let's imagine that all possible items can be used as augmentations. For example, it would be possible to augment the closed itemset $\{a, b\}$ with the item d which would generate the closed itemset $\{a, b, c, d\}$. But as we will see later on, the closed itemset $\{a, c, d\}$ augmented by the item b also generates $\{a, c, d, b\}$. In both cases, the first parent test would fail to block the generation of duplicates. Therefore, the use of items smaller than the current augmentation for the recursive call allows for a simple and efficient first parent test.

Following the example, the closed itemset $\{a, b\}$ (1) comes from augmentation b, thus it can only be further augmented with item a ($a < b$). Therefore, there does not exist any possible augmentation of this closed itemset, thus no recursive call is made. On the other hand, the augmentation from which the closed itemset $\{a, c, d\}$ (3) comes from is d, which leaves item b as possible augmentation. Therefore, *expand* is recursively invoked with $Q = \{a, c, d\}$, the item b, and the calculated reduced dataset $D_{a,c,d}$.

Let's finish the example, the augmentation b is frequent and generates by closure the itemset $\{b\}$. The maximal item of $\{b\}$ is b, thus equal to the augmentation b which means that the itemset $\{b\}$ passes the first parent test. Therefore, the closed itemset $Q = \{a, c, d, b\}$ is outputted and, since there are no more items in the dataset, LCM algorithm finishes.

Therefore, LCM algorithm for the dataset shown in Figure 4.1 (a) and a minimum support threshold of 2 transactions, obtains the set of closed itemsets $\{\{a\}, \{a, b\}, \{a, c, d\}, \{a, c, d, b\}\}$.

Table 4.1: Simple dataset

t_1	t_2	t_3	t_4	t_5
a	.	a	.	a
b	.	b	.	b
c	.	.	.	c

Due to their triadic nature, CPCs cannot benefit immediately from existing enumeration techniques. This comes from the behavior of periods since the natural specialization operation between CPCs can modify the period. For example, consider a CPC $C_1 = (\{a, b\}, \{2\}, \{t_1, t_3, t_5\})$ from the dataset shown in Table 4.1. Intuitively, its specialization is the CPC $C_2 = (\{a, b, c\}, \{4\}, \{t_1, t_5\})$. As can be seen, the period changed in C_2 to become a multiple of the period of C_1. Therefore, it is not possible to reuse the set of frequent periods from one CPC to its specialization.

Indeed, as the period of a CPC is in fact a set, periods of a more specific CPC can be multiples or additions of the initial period set. Therefore, existing first parent tests, based on strict set inclusion, cannot be exploited directly by CPCs.

It could be possible to build a first parent test for CPCs that relies on elaborated computations on the period. However, unlike the standard triadic case where no terms of a triple can be deduced from each other just given the triple, here the period term of a periodic concept has a strong dependency with the transactions term of that triple. Our solution is thus to exploit this fact in order to drop the period completely from the first parent test, and to structure the enumeration around the enumeration of the itemset part of the CPC.

So, what are the changes from LCM's enumeration to PerMiner's enumeration? The main structure of both algorithms is very similar, i.e. first, checking if the candidate pattern is frequent; second, computing its closure; third, first parent test; and finally, outputting the pattern and making the recursive calls.

Since PerMiner mines core periodic concepts, instead of simply checking the frequency of the candidate itemset, PerMiner generates all frequent periods of the candidate itemset. Then, following the definition of CPC (see Definition 3.7), the set of frequent periods is reduced to the set of periods with maximal sets of transactions, since they are the only ones that can generate CPCs. The periods of the set of frequent periods that share the same set of transactions are grouped together. With these two steps, PerMiner algorithm maximizes the set of transactions, and then the set of periods of candidate CPCs.

The closure computation is similar in both algorithms, with the difference that PerMiner does not use the whole set of transactions containing the candidate itemset. Instead, for each

candidate CPC, the itemset is computed by intersecting the contents of all transactions in the CPC's set of transactions. This part allows PerMiner algorithm to maximize the itemset of a set of candidate CPC, thus obtaining a set of actual CPCs.

Then, the first parent test is used to prune out patterns that would generate duplicates in the enumeration. The first part of the first parent test in PerMiner algorithm is identical to the first parent test in LCM. Then, due to the possible generation of several CPCs on each step of the algorithm, an extra filtering and an exclusion list are used to avoid generating duplicates.

Finally, when outputting the discovered patterns, LCM outputs a unique closed frequent itemset while PerMiner can output several CPCs per augmentation. Since several CPCs can contain the same itemset, an extra test is carried out to make the recursive call only once per itemset. The enumeration strategy, explained before by which only items smaller than the current augmentation are used as augmentation of the current pattern, is used in both algorithms. Therefore, as can be observed, all elements of the itemset enumeration used in LCM are present in PerMiner, with extra filters due to the nature of CPCs that assures the correct enumeration of CPCs without generating duplicates.

Algorithm 2: Core Periodic Concepts Miner

```
1 procedure PerMiner (D, min_sup);
  Data: dataset D, minimum support threshold min_sup
  Result: Output all Core Periodic Concepts that occur in D
2 begin
3     if |D| ≥ min_sup then
4         ⊥_clo ← ⋂_{t∈D} t
5         output (⊥_clo, {1..|D|/2}, D)
6         D_{⊥_clo} = {t \ ⊥_clo | t ∈ D}
7         foreach e ∈ I with e ∉ ⊥_clo do
8             perIter(⊥_clo, D_{⊥_clo}, e, ∅, min_sup)
```

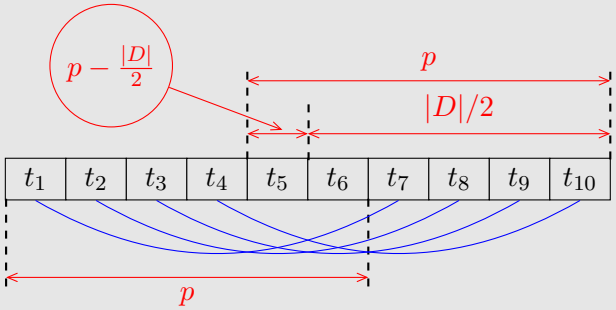
Concretely, PerMiner algorithm, described in procedure PerMiner of Algorithm 2, starts by generating the CPC present in all transactions in the dataset (line 4), i.e. $(\perp_{clo}, \mathcal{P}, \mathcal{D})$.

Once the CPC $(\perp_{clo}, \mathcal{P}, \mathcal{D})$ has been generated by intersecting all transactions in the dataset, it is outputted in line 5, and expanded by invoking the recursive function *perIter* in line 8. All items that do not belong to $\perp_{clo}$ are used as augmentations, see line 7. For each augmentation, the recursive function *perIter* is invoked with $\perp_{clo}$, the input dataset minus the itemset $\perp_{clo}$, and the augmentation.

perIter, described in Algorithm 3, is in charge of the enumeration. *perIter* takes as input the itemset X of a previously found CPC $C = (X, P, T)$, a reduced dataset D_X, an augmentation e, an exclusion list el and a minimum support threshold *min_sup*.

$\perp_{clo}$'s set of periods

The set of periods of the CPC with itemset $\perp_{clo}$ is the whole set of possible periods. This is because, following the definition of cycle 3.1, a cycle contains at least two transactions. In order for a frequent period to appear in the whole set of transactions, the maximum period would be $|D|/2$. Otherwise, with a bigger period there would be a gap in the middle of the dataset.



An example is shown in the figure above, where with a period of 6 there is a gap of 2 transactions in the middle of the dataset, i.e. t_5 and t_6, that are not connected to any other transaction. Concretely, given a period p and a dataset D, the number of transactions not connected to any other transaction by a period p is equal to

$$\left(p - \frac{|D|}{2}\right) \cdot 2 \tag{4.1}$$

Therefore, the set of periods containing all transactions in the dataset are the periods for which the Equation 4.1 is smaller or equal to zero:

$$\begin{aligned} \left(p - \frac{|D|}{2}\right) \cdot 2 &\leq 0 \\ 2p - |D| &\leq 0 \\ p &\leq \frac{|D|}{2} \end{aligned} \tag{4.2}$$

perIter computes the sets of frequent periods with the corresponding set of transactions of the current itemset in lines 4-6. Then, for each set of frequent periods, the maximal itemset present in the corresponding set of transactions is calculated by intersecting the transactions in lines 7-10. Note that the set of transactions corresponding to a set of frequent periods is always a subset of the set of transactions containing the current itemset.

The difference between the closure computation in LCM and PerMiner can be seen in the example shown in Figure 4.2. Here, the intersection of the whole set of transactions done by LCM generates the itemset $\{a, b\}$, while PerMiner computes the intersection of the transactions belonging to the frequent period $\{2\}$, i.e. t_1, t_3 and t_5, generating the itemset $\{a, b, c\}$.

Algorithm 3: Iterative CPC Generator

```
1  procedure perIter(X, D_X, e, el, min_sup);
   Data: Itemset of a discovered CPC X, reduced dataset D_X, item e, exclusion list el,
         minimum support threshold min_sup.
   Result: Output all Core Periodic Concepts whose itemset is prefixed by X and whose
           transactions are in D_X, with minimal support min_sup.
2  begin
3      A := {e}
4      B := getPeriods(tidlist(A), min_sup)                    /* Period computation */
5      B' := B \ {b | ∄b' ∈ B such that b.occs ⊂ b'.occs}
6      G := group(B')
7      S ← ∅                                                   /* Closure computation */
8      foreach g ∈ G do
9          A' := ⋂_{t∈g.occs} t
10         S := S ∪ (A', g.periods, g.occs)
11     S := filter(S);                                         /* First parent test */
12     new_el ← el
13     enum ← ∅                                                /* Itemset enumeration */
14     foreach (A', P, T) ∈ S do
15         if max_elem(A') = e then
16             Q = X ∪ A'
17             if el_test(Q, el) then
18                 output (Q, P, T)
19                 if Q ∉ enum then
20                     D_Q = reduce(D_X, Q, e, min_sup)        /* Dataset Reduction */
21                     foreach i ∈ I with i < e and i ∉ Q do
22                         perIter(Q, D_Q, i, new_el, min_sup)
23                     enum := enum ∪ Q
24                 new_el := new_el ∪ Q
```

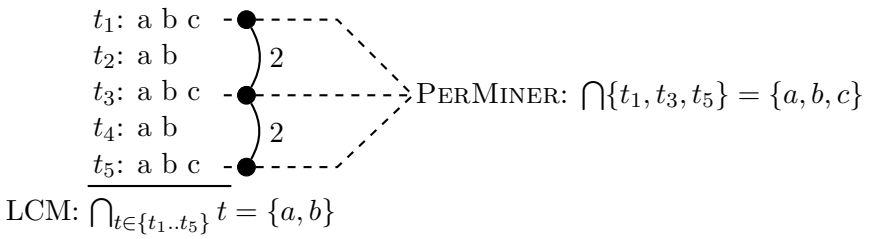


Figure 4.2: Difference between LCM's closure and PerMiner's closure

After this step, *perIter* has a set of CPCs. Finally, before outputting the CPCs and making the recursive call, the first parent test is carried out in order to avoid generating duplicates in lines 11, 15 and 17. In *perIter*, the first parent test consists in three phases, the first phase is a filter that removes any CPC whose itemset is a superset of the itemset of any other CPC on the set. Then, the second phase is the first parent test used in LCM algorithm. And finally, the third phase consists on checking whether the CPC's itemset contains any of the itemsets in the exclusion list. This last step is needed due to the fact that when several CPCs are outputted, this induces several calls to *perIter* with itemsets which are likely to reach the same CPCs later in the enumeration, in ways not handled by classical itemset first parent test.

If a pattern passes the first parent test, it is outputted in line 18 and the recursive call to *perIter* in line 22 is made using its itemset as parameter, and with all possible augmentations. In PerMiner, like in LCM, the set of possible augmentations of an itemset of a CPC is bounded by the item used as augmentation on the current *perIter* execution, see line 21 of Algorithm 3. There might be several CPCs to output with the same itemset, in which case all CPCs are outputted but the recursive call is made only once to avoid generating duplicates.

In the rest of this section, we are going to explain through an example each step of *perIter* recursive call. The dataset is presented in Table 4.2 and the schema of enumeration is shown in Figure 4.3.

Table 4.2: Example dataset

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14-19}	t_{20}	t_{21}	t_{22-30}	t_{31}	t_{32-41}	t_{42}
a	.	a	.	.	a	.	a	.	.	a	.	a	.	a	b	.	b	.	c
b	.	b	.	.	c	.	c	.	.	b	.	b	.	c	c	.	c	.	d
c	.	c	.	.	d	.	d	.	.	c	.	c	.	d	.	.	d	.	e
d	.	.	.	.	.	.	.	.	.	.	.	d	.	e	.	.	e	.	.

Figure 4.3 is read from bottom to top following the execution of *perIter* with the element $\{c\}$ as augmentation, i.e. $perIter(\perp, D, c, \emptyset, min\_sup)$. On the right hand side of the figure, the lines of Algorithm 3 responsible for each step of the algorithm are stated. The crosses on the paths represent where the corresponding tuple or pattern is removed from the computation, and on the right side of the cross the reason of the removal is stated. Dots represent that the test on that line passed, or that the operation was successfully carried out, in which case the output is shown on the right hand side of the dot. Each branch represents a possible CPC that the algorithm will prune, join or grow depending on the execution. The branches have been labeled with numbers, e.g. (n), to facilitate the comprehension of the example during the explanation.

The minimum support threshold is 3 transactions. The first part of PerMiner algorithm generates $(\perp, \mathcal{P}, \mathcal{D})$ since there is no item present in all transactions of the database.

In line 3 of *perIter*, the candidate itemset A is formed by the augmented item received as input by perIter, i.e. $\{c\}$.

Period Computation. *getPeriods*, described in Algorithm 4, only returns periods that verify Property 3.11: $\forall (p, t) \in B$ the triple $(A, \{p\}, t = tidlist(A, p))$ is fully connected. This avoids generating triples that cannot become CPCs. For example, a standard triadic concept

Algorithm 4: Period Generator

```
1  function getPeriods(T, min_sup)
   Data: Transaction list T, minimum support threshold min_sup
   Result: A list of tuples (period, transaction list of the period)
2  B ← ∅
3  foreach period ∈ [1..|D|/min_sup] do
4      b.occs ← ∅
5      b.periods := period
6      i := 0
7      while i < (|T| − 1) do
8          if T[i].checked == false then
9              j := i + 1
10             while j < |T| AND (T[j] − T[i]) <= period do
11                 if (T[j] − T[i]) == period then
12                     b.occs := b.occs ∪ i; T[i].checked := true
13                     b.occs := b.occs ∪ j; T[j].checked := true
14                     k := j + 1
15                     while k < |T| AND (T[k] − T[j]) <= period do
16                         if (T[k] − T[j]) == period then
17                             b.occs := b.occs ∪ k; T[k].checked := true
18                             j := k
19                         k++
20                 j++
21         i++
22     if |b.occs| >= min_sup then
23         B := B ∪ b
24 return B
25 end function
```

miner would generate $(\{a, c\}, \{2, 5, 10\}, \{t_1, t_3, t_{11}, t_{13}\})$ as a triadic concept, but this triadic concept would not become a CPC since the periods in it are not fully connected, i.e. t_1 is not connected to any other transaction of the set with a period of 5 transactions, as happens with all other transactions of the set, see Figure 4.4 for details.

getPeriods takes as input a set of transactions T and a minimum support threshold *min_sup*. For each possible period, *getPeriods* finds all cycles of the given period on the set of transactions T (see Definition 3.1), and forms a tuple between the period and the transactions forming part of the cycles. The tuple is returned only if the set of transactions contains more than *min_sup* transactions.

Following the example, *getPeriods* is invoked with the set of transactions $\{t_1, t_3, t_6, t_8, t_{11}, t_{13}, t_{20}, t_{21}, t_{31}, t_{42}\}$, where the itemset $\{c\}$ is found, and a minimum support threshold of 3 transactions. The set of possible periods is $\{1..14\}$, i.e. $|D|/min\_sup$. For clarity, all cycles of frequent periods are shown in Figure 4.4. Over period 2, there are three cycles

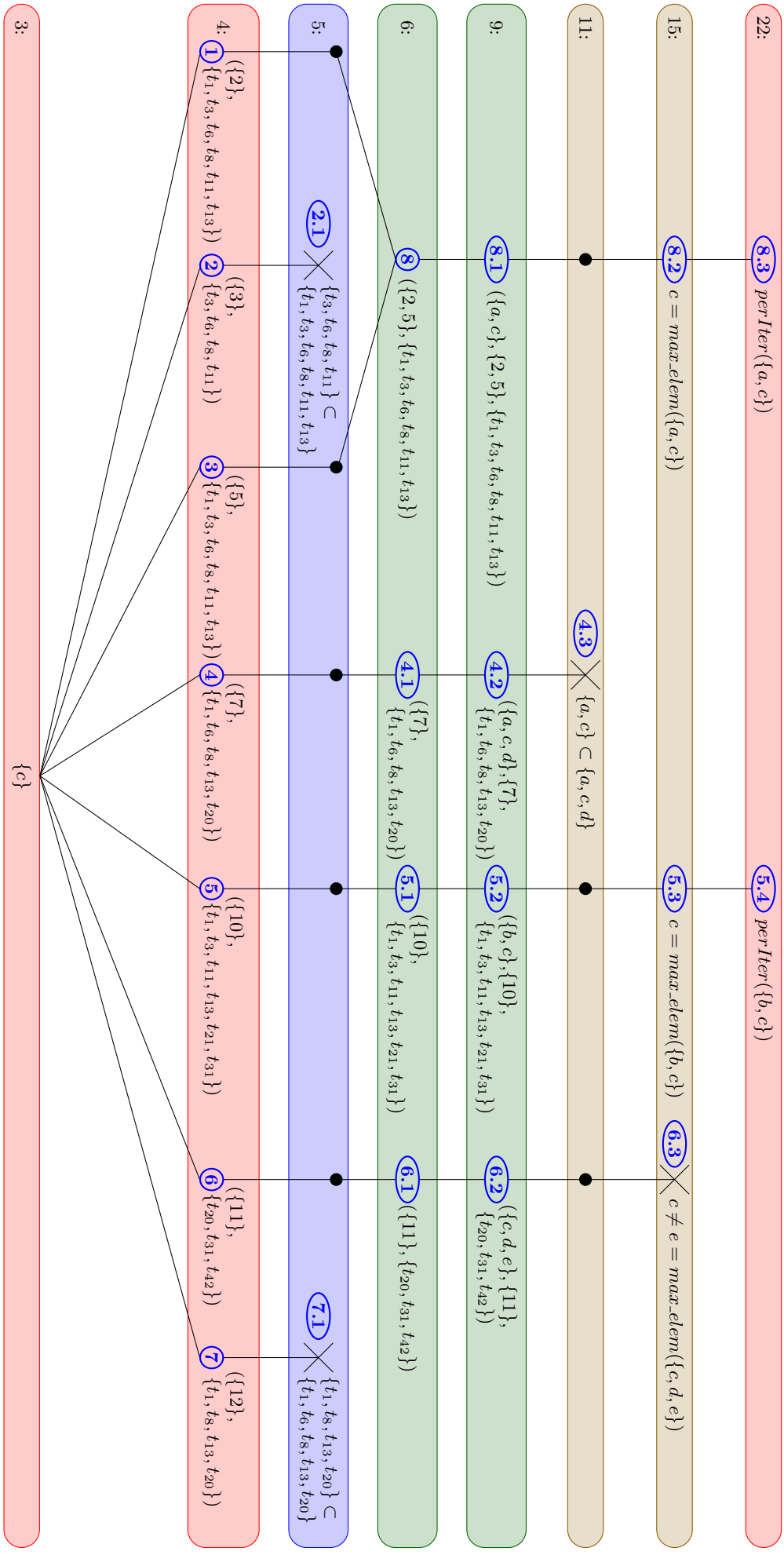


Figure 4.3: Graphical visualization of *perIter* execution

starting in transactions t_1, t_6 and t_{11} respectively, all of length 2, which gives the set of cycles $\{(1,2),(6,2),(11,2)\}$, following notation specified in Definition 3.1, where each cycle is represented by a tuple $(origin, length)$. Then, the set of transactions contained in those cycles is $\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\}$. Since the length of the set of transactions (6) is greater than the minimum support threshold (3), the tuple $(\{2\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ① is added to the output of the function.

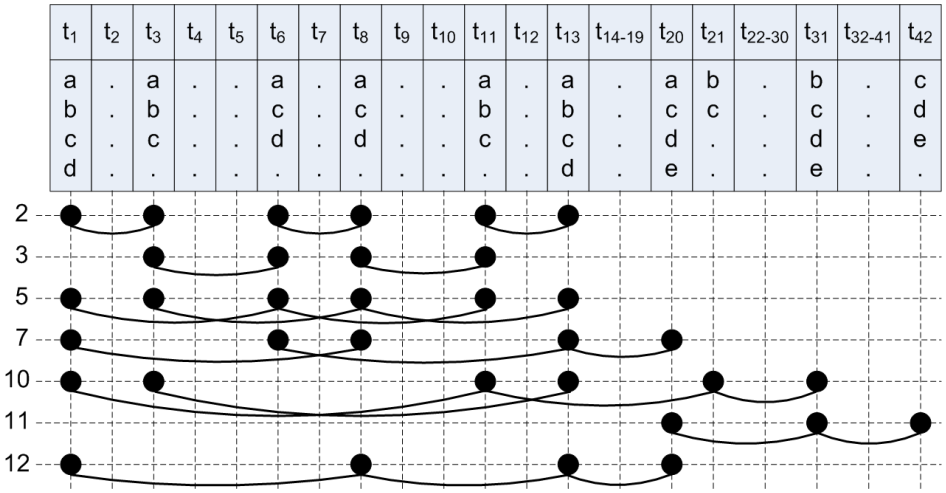


Figure 4.4: Dataset periods visualization

The same procedure is carried out with the rest of the periods, giving the set $B = \{(\{2\},\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ①, $(\{3\},\{t_3, t_6, t_8, t_{11}\})$ ②, $(\{5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ③, $(\{7\}, \{t_1, t_6, t_8, t_{13}, t_{20}\})$ ④, $(\{10\},\{t_1, t_3, t_{11}, t_{13}, t_{21}, t_{31}\})$ ⑤, $(\{11\}, \{t_{20}, t_{31}, t_{42}\})\}$ ⑥, $(\{12\}, \{t_1, t_8, t_{13}, t_{20}\})$ ⑦.

In line 5 of Algorithm 3, the set of periods is reduced by exploiting Theorem 3.9. That is, any tuple whose transaction list is contained in any other tuple's transaction list, is removed from the set.

Following the example, given the set B calculated above, the set $B' = \{(\{2\},\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\}), (\{5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\}), (\{7\},\{t_1, t_6, t_8, t_{13}, t_{20}\}), (\{10\},\{t_1, t_3, t_{11}, t_{13}, t_{21}, t_{31}\}), (\{11\}, \{t_{20}, t_{31}, t_{42}\})\}$ is generated. Tuple $(\{3\},\{t_3, t_6, t_8, t_{11}\})$ **2.1** is removed from the set since its transaction list is contained in the transaction list of the tuple $(\{2\},\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ①. Likewise, tuple $(\{12\}, \{t_1, t_8, t_{13}, t_{20}\})$ **7.1** is removed from the set since its transaction list is contained in the transaction list of the tuple $(\{7\}, \{t_1, t_6, t_8, t_{13}, t_{20}\})$ ⑤. Theorem 3.9 tells us that for all periods of a CPC, their corresponding transaction list is the transaction list of the CPC. Tuple $(\{3\},\{t_3, t_6, t_8, t_{11}\})$ would generate the periodic concept $(\{a, c\}, \{2, 3, 5\},\{t_3, t_6, t_8, t_{11}\})$ which is not a CPC, since $tidlist(\{a, c\}, 2) = \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\} \neq \{t_3, t_6, t_8, t_{11}\}$.

In line 6 of Algorithm 3, the set of periods is further reduced by function *group*, described in Algorithm 5, which groups tuples with the same set of transactions. Following the example, given the set B' calculated above, tuples $(\{2\},\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ① and $(\{5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ③ have the same set of transactions and produce the tuple $(\{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ ⑧. Therefore $G = \{(\{2, 5\},\{t_1, t_3, t_6, t_8, t_{11}, t_{13}\}), (\{10\},\{t_1, t_3,$

$t_{11}, t_{13}, t_{21}, t_{31}\}$), ($\{7\}$,$\{t_1, t_6$,$t_8$,$t_{13}, t_{20}\}$), ($\{11\}, \{t_{20}, t_{31}, t_{42}\}$)}.

Closure Computation. At this point, we have the precise period and transaction sets of a set of CPCs. In order to obtain the corresponding set of CPCs, the itemsets have to be maximally grown. In lines 7-10 of Algorithm 3, for each tuple in G, the transactions are intersected to get the maximal itemset (line 9 of Algorithm 3) contained in them. Then, the CPC formed by the maximal itemset, the set of periods and the set of transactions is included on the set S. Therefore, at this point of the algorithm we obtain a set of CPCs.

Following the example, given the set of tuples G calculated above, lines 7-10 of Algorithm 3 generate the set $S = \{(\{a, c\}, \{2, 5\}$,$\{t_1, t_3, t_6, t_8, t_{11}$,$t_{13}\})$ **8.1**, $(\{a, c, d\}, \{7\}$,$\{t_1, t_6, t_8$,t_{13}, $t_{20}\})$ **4.2**, $(\{b, c\}, \{10\}, \{t_1, t_3, t_{11}, t_{13}$,$t_{21}, t_{31}\})$ **5.2**, $(\{c, d, e\}, \{11\}, \{t_{20}, t_{31}, t_{42}\})\}$ **6.2**.

Algorithm 5: Group tuples with same transaction list

```
1 function group(B)
  Data: List of tuples (period, transaction list of the period) B
  Result: A list of tuples grouped by transaction list
2 foreach b, b' ∈ B do
3     if b.occs == b'.occs then
4         b.periods := b.periods ∪ b'.periods
5         B := B \ b'
6 return B
7 end function
```

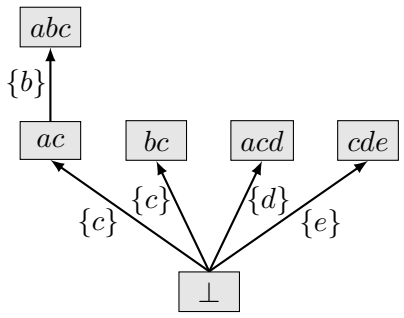


Figure 4.5: Enumeration tree of the example used to illustrate PERMINER algorithm. Note that, augmentation c of itemset $\perp$ generates two CPCs $(\{a, c\}, \{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ and $(\{b, c\}, \{10\}, \{t_1, t_3, t_{11}, t_{13}$,$t_{21}, t_{31}\})$.

First Parent Test. The first parent test is performed in lines 11, 15 and 17. If a CPC passes the three phases of the first parent test, it means that the current augmentation of the input itemset X, i.e. $X \cup \{e\}$, is its first parent. First, in line 11, any CPC of S whose itemset is the superset of the itemset of another CPC of S is suppressed by function $filter$, described in Algorithm 7. Note that the suppressed itemset will be generated by an augmentation of the smaller itemset later on the enumeration tree or by another branch of the enumeration tree.

Following the example, from the set S calculated above, the itemset $\{a, c, d\}$ from CPC $(\{a, c, d\}, \{7\}, \{t_1, t_6, t_8, t_{13}, t_{20}\})$ (4.2) contains the itemset $\{a, c\}$ from CPC $(\{a, c\}, \{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ (8.1) and is thus suppressed from the set (4.3). The CPCs with itemset $\{a, c, d\}$ will be generated by the branch exploring the augmentation $\{d\}$ of the empty set, as can be seen in Figure 4.5. The itemset $\{a, c, d\}$ cannot be generated by an augmentation of the itemset $\{a, c\}$ since the augmentations are limited by the maximum item in $\{a, c\}$ which is $\{c\}$. Itemsets $\{a, c\}$, $\{b, c\}$ and $\{c, d, e\}$, from CPCs $(\{a, c\}, \{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ (8.1), $(\{b, c\}, \{10\}, \{t_1, t_3, t_{11}, t_{13}, t_{21}, t_{31}\})$ (5.2) and $(\{c, d, e\}, \{11\}, \{t_{20}, t_{31}, t_{42}\})\}$ (6.2) respectively, do not contain the itemset of any other CPC, thus they are kept in the set S.

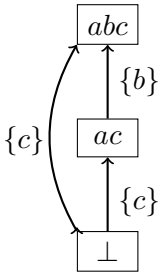


Figure 4.6: Importance of first phase of first parent test

But, the itemset $\{a, c, d\}$ would have been considered not valid by the second part of the parent test anyway (line 15 of Algorithm 3), so why is the first part of the first parent test useful? In order to better show the interest of this test, consider the case of augmenting the empty set with the element $\{c\}$, as shown in Figure 4.6. It generates two CPCs with itemsets $\{a, b, c\}$ and $\{a, c\}$. Without the test on line 11, both itemsets $\{a, b, c\}$ and $\{a, c\}$ would pass the tests in lines 15 and 17 of Algorithm 3, be outputted and further explored. Then, by exploring $\{a, c\}$ with the augmentation $\{b\}$, a CPC containing $\{a, b, c\}$ would be generated and outputted, therefore generating a duplicate.

The second part of the first parent test, in line 15, carries out the same first parent test used in LCM algorithm, which avoid generating duplicates.

Following the example, for each CPC in the set S, line 15 of Algorithm 3 checks whether the maximum item on the CPC's itemset is equal to the item used in the augmentation of the itemset X, in this case the item c. First, (8.2) $max\_elem(\{a, c\})$ is c which means that it passes the test. Then, $max\_elem(\{b, c\})$ is c which means that it passes the test. Finally, $max\_elem(\{c, d, e\})$ is e which is different from c and therefore fails the test, so the CPC is abandoned since it will be generated by another branch of the enumeration tree (6.3) (see Figure 4.5).

Finally, the third part of the first parent test, in line 17 of Algorithm 3 checks whether the current itemset Q contains any of the elements of the exclusion list el, done by function *el_test* shown in Algorithm 6. A problem of our enumeration strategy is that a single *perIter* invocation can produce several CPCs having similar itemsets. Then, for each of these itemsets a recursive call is made, all of which make an LCM-like enumeration, with a risk of generating duplicates. We thus further refine the order of CPC generation using an "exclusion list", slightly modified from Boley et al. [BHPW07]. This exclusion list prevents those "parallel" enumerations from generating the same itemsets, by excluding itemsets produced by one branch from the enumeration of further branches.

Algorithm 6: Execution List Test

```
1 function el_test(Q, el)
  Data: Itemset Q, Exclusion list el
  Result: True if none of the elements in el is included in Q. False otherwise.
2 foreach X ∈ el do
3     if X ⊂ Q then
4         return False
5 return True
6 end function
```

Following the example, since the exclusion list el received as parameter of *perIter* is empty, this test does not have any effect on the current list of CPCs, which means that $\perp \cup \{c\}$ is the first parent of CPCs $(\{a, c\}, \{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$ (8.1) and $(\{b, c\}, \{10\}, \{t_1, t_3, t_{11}, t_{13}, t_{21}, t_{31}\})$ (5.2). Therefore, these CPCs will be outputted and *perIter* will be recursively called with all possible augmentations of these two itemsets. The only possible augmentation of $\{a, c\}$ is b, and the only possible augmentation of $\{b, c\}$ is a. However, these two augmentations give the same itemset $\{a, b, c\}$ which means that both invocations of *perIter* would generate the same CPCs. Therefore, it is necessary to remember the itemsets that have already been used in *perIter* invocations in order to avoid generating duplicates.

Initially, the exclusion list *new_el* used as parameter in *perIter*'s recursive invocations is equal to the exclusion list el received as input, line 12 of Algorithm 3. Then, after each invocation of *perIter*, the itemset used in that invocation is added to the exclusion list *new_el* (line 24 of Algorithm 3).

In our example, let's suppose that the invocation to *perIter* with the itemset $\{a, c\}$ is made first. In that case, the exclusion list used would be a copy of the exclusion list received as input $el = \emptyset$ (line 12 of Algorithm 3). After *perIter*'s invocation, the exclusion list *new_el* is updated with the itemset $\{a, c\}$ in line 24 of Algorithm 3, i.e. $new\_el = \{\{a, c\}\}$.

Then, in *perIter*'s invocation with itemset $\{b, c\}$, the exclusion list $new\_el = \{\{a, c\}\}$ is passed as parameter. This invocation would generate a CPC $(\{a, b, c\}, \{2, 10\}, \{t_1, t_3, t_{11}, t_{13}\})$ with itemset $\{a, b, c\}$. Then, in line 17 of Algorithm 3 this CPC would be removed from the algorithm since the element $\{a, c\}$ of the exclusion list is contained in the itemset of the CPC $\{a, b, c\}$.

Algorithm 7: Filter CPCs

```
1 function filter(S)
  Data: List of CPCs S
  Result: A filtered list of CPCs
2 foreach (A, P, C), (A', P', C') ∈ S do
3     if A ⊂ A' then
4         S := S \ (A', P', C')
5 return S
6 end function
```

Itemset Enumeration. If these tests pass, the CPC is outputted in line 18. An extra test is carried out in lines 13, 19 and 23, that makes sure that if in S there are two CPC with the same itemset part A', the itemset is expanded only once by the algorithm.

Then, a reduced dataset is built by function *reduce* in line 20, shown in Algorithm 8. The reduced dataset will contain only the transactions that support the itemset of the CPC (line 20 of Algorithm 8). Also, any infrequent item as well as the elements of the itemset A' are removed from the transactions in the reduced dataset (lines 3 to 7 of Algorithm 8).

Algorithm 8: Database Reduction

```
1 function reduce(D_X^reduced, A', e, min_sup)
  Data: Database D_X^reduced, Itemset A', element e, minimum support threshold min_sup
  Result: Reduced Database of A': D_A'^reduced
2 D_A'^reduced = D_X^reduced[e]
3 foreach i ∈ I do                    /* All items of I with support smaller than */
  |                                      /* min_sup are removed from the database */
4 |  if support(i) < min_sup then
5 |  |  Suppress i from all transactions in D_A'^reduced
6 foreach i ∈ A' do            /* All items of A' are removed from the database */
7 |  Suppress i from all transactions in D_A'^reduced
8 return D_A'^reduced
9 end function
```

For each distinct itemset in the set of CPCs, *perIter* is recursively invoked. As in LCM, the possible augmentations of the itemset are bounded by the augmentation item received as parameter in the current execution of *perIter*, see line 21. Therefore, *perIter* in line 22 is invoked as many times as possible augmentations there are, with the reduced database as parameter.

To finish the example used over the explanation of PerMiner algorithm, the CPC $(\{a,c\},\{2,5\},\{t_1,t_3,t_6,t_8,t_{11},t_{13}\})$ **8.1** is outputted by the algorithm and then its reduced dataset is calculated. Then, the itemset contained on the CPC, i.e. $\{a,c\}$ is used on the recursive call to *perIter* to continue the enumeration **8.3**. The possible augmentations of the itemset $\{a,c\}$ are bounded by its augmentation item c which only leaves the option of using item b as augmentation. Augmentation b of $\{a,c\}$ generates the CPC $(\{a,b,c\},\{2,10\},\{t_1,t_3,t_{11},t_{13}\})$. Then, the exclusion list *new_el* gets incremented with the itemset $\{a,c\}$.

Moreover, the CPC $(\{b,c\},\{10\},\{t_1,t_3,t_{11},t_{13},t_{21},t_{31}\})$ **5.2** is outputted by the algorithm and its reduced dataset is calculated. Then, the itemset contained on the CPC, i.e. $\{b,c\}$ is used on the recursive call to *perIter* to continue the enumeration **5.4**. The possible augmentations of the itemset $\{b,c\}$ are bounded by its augmentation item c which only leaves the option of using a as augmentation. As has been said above, augmentation a of $\{b,c\}$ would generate the CPC $(\{a,b,c\},\{2,10\},\{t_1,t_3,t_{11},t_{13}\})$ which has already been generated by the previous invocation to *perIter* with itemset $\{a,c\}$, but since $new\_el = \{\{a,c\}\}$ and $\{a,c\} \subset \{a,b,c\}$, the third part of the first parent test (line 17 of Algorithm 3) would stop the generation of this CPC.

To finish the current example, augmentation d of $\perp$ generates CPC $(\{a, c, d\}, \{7\}, \{t_1, t_6, t_8, t_{13}, t_{20}\})$, and augmentation e of $\perp$ generates CPC $(\{c, d, e\}, \{11\}, \{t_{20}, t_{31}, t_{42}\})$, as shown in Figure 4.5. The result set of CPCs mined by PerMiner algorithm over the dataset 4.2 with a minimum support threshold of 3 transactions is shown in Table 4.3.

Table 4.3: Set of core periodic concepts

Core Periodic Concepts
$(\{\perp, \{1..14\}, \{t_1..t_{42}\}\})$
$(\{a, c\}, \{2, 5\}, \{t_1, t_3, t_6, t_8, t_{11}, t_{13}\})$
$(\{b, c\}, \{10\}, \{t_1, t_3, t_{11}, t_{13}, t_{21}, t_{31}\})$
$(\{a, b, c\}, \{2, 10\}, \{t_1, t_3, t_{11}, t_{13}\})$
$(\{a, c, d\}, \{7\}, \{t_1, t_6, t_8, t_{13}, t_{20}\})$
$(\{c, d, e\}, \{11\}, \{t_{20}, t_{31}, t_{42}\})$

4.2 Complexity analysis

Considering $n = |\mathcal{D}|$, $\mathcal{P} = n/2$ and $\mathcal{I}$ the set of all items, defined in Section 3.1. An algorithm is of **polynomial space complexity** if the maximum size of its working space is bounded by a polynomial in the total input size $||\mathcal{D}||$.

Theorem 4.2 (Polynomial Space Complexity)**.**

PerMiner *enumerates the family of core periodic concepts with polynomial space.*

Proof. Since PerMiner is a depth-first algorithm, the maximum memory used by it is the maximum memory used by an enumeration path, with a number of nodes in the path bounded by $|\mathcal{I}|$, as can be seen on Figure 4.7. Then, each invocation to *perIter* contains two space complexity factors: the reduced dataset D_X and the exclusion list *el*.

A new reduced dataset of size $O(||\mathcal{D}||) = O(n \cdot |\mathcal{I}|)$ is generated in each invocation of perIter. Therefore, in the whole enumeration path the space complexity of reduced datasets is $O(n \cdot |\mathcal{I}|^2)$.

The exclusion list *el* is a list of itemsets, where each itemset has a size of $O(|\mathcal{I}|)$. There is one exclusion list per node which contains the exclusion list of the previous invocation, and a maximum of $|\mathcal{P}|$ new itemsets are included in the exclusion list in each node of the path, which makes a series:

$$
\begin{array}{ll}
 & O(|\mathcal{P}||\mathcal{I}|) \quad \text{\# First step} \\
 & O(|\mathcal{P}||\mathcal{I}| + |\mathcal{P}||\mathcal{I}|) \quad \text{\# Second step} \\
 & O(|\mathcal{P}||\mathcal{I}| + |\mathcal{P}||\mathcal{I}| + |\mathcal{P}||\mathcal{I}|) \quad \text{\# Third step} \\
 & \ldots \\
 & \ldots \\
\text{Total:} & O(|\mathcal{P}||\mathcal{I}| \cdot \frac{|\mathcal{I}| \cdot (|\mathcal{I}|+1)}{2}) = O(\frac{n|\mathcal{I}|^2 \cdot (|\mathcal{I}|+1)}{4}) \approx O(n|\mathcal{I}|^3) \quad \text{\# Worst case}
\end{array}
$$

So, in total, the space complexity of PerMiner algorithm is $O(n|\mathcal{I}|^2 + n|\mathcal{I}|^3) \approx O(n|\mathcal{I}|^3)$. □

Likewise, an algorithm is of **polynomial delay complexity** if the maximum computational time between two consecutive outputs is bounded by a polynomial in the total input size, i.e. $||\mathcal{D}||$ (defined in Section 3.1).

Theorem 4.3 (Polynomial Delay Complexity)**.**

PerMiner *enumerates the family of core periodic concepts with polynomial delay.*

Proof. The time complexity of an invocation to *perIter* is dominated by the call to *getPeriods* in line 4, the intersection computation of lines 7-10 and the third phase of the first parent tests in line 17 of Algorithm 3. *getPeriods* operates only on the list of transactions of A and has a complexity of $O(n^2)$ since the list of transactions is traversed once for each possible period. The maximum period possible is $|\mathcal{D}|$, if we consider that the minimum support threshold value is 1 transaction and that the set of periods is bounded by $|\mathcal{D}|/min\_sup = |\mathcal{D}|/1 = |\mathcal{D}|$.

The intersection computation of lines 7-10 is carried out once for every element on the set G, which is a subset of the result of *getPeriods*. The maximum number of results returned by *getPeriods* is bounded by the number of possible periods, i.e $|\mathcal{D}|$. Each intersection of lines 7-10 has a time complexity of $O(n^2 \cdot |\mathcal{I}|)$ and therefore, the set of intersections has a time complexity of $O(n^3 \cdot |\mathcal{I}|)$.

The third phase of the first parent test in line 17 consists on checking whether the current itemset Q contains any of the elements of the exclusion list el. As shown above, the exclusion list is in the worst case of size $O(n|\mathcal{I}|^3)$. Therefore, the third phase of the first parent test has a time complexity of $O(n|\mathcal{I}|^4)$.

Therefore, the overall complexity of *perIter* is thus $O(n^2 + n^3 \cdot |\mathcal{I}| + n|\mathcal{I}|^4)) = O(n \cdot (n + n^2 \cdot |\mathcal{I}| + |\mathcal{I}|^4)) \approx O(n^3 \cdot |\mathcal{I}|^4)$ which is polynomial.

Since PerMiner follows a tree-shaped enumeration strategy, shown in Figure 4.7, the time delay between the output of two solutions is polynomial. Let C_1 be a solution, and C_2 be the next solution output. Due to the structure of the enumeration, the first parent of C_2 is necessarily on the path from the root to C_1. Considering the worst case shown on Figure 4.8, the number of nodes on this path is in $O(|\mathcal{I}|)$, and for each node of this path at most $O(|\mathcal{I}|)$ "failing augmentations" are explored (from line 21 of *perIter*). Thus the main computation of *perIter* without recursive call (lines 3-18), is done at most $O(|\mathcal{I}|^2)$ times, hence the polynomial delay. □

4.3 Parallelization

In this section, we propose a simple parallelization of PerMiner algorithm, based on the parallelization of LCM algorithm by Negrevergne et al. [NTMU10], in order to exploit multicore processors. In [NTMU10], the authors defined a work sharing execution model, called *Melinda*.

Melinda consists in a shared memory space, called *TupleSpace*, accessible by all threads. Threads can either deposit or retrieve data units, called *Tuples*, via the primitives $put(Tuple)$ and $get(Tuple)$, and Melinda is in charge of handling the synchronization in the access to

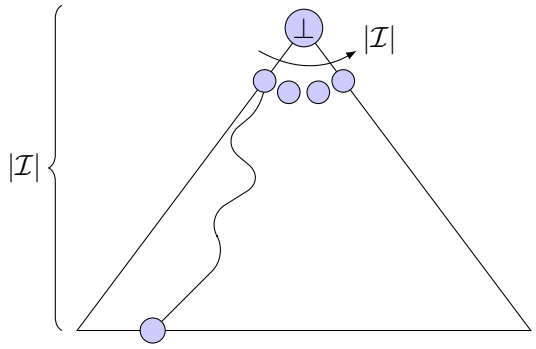


Figure 4.7: Enumeration Tree.

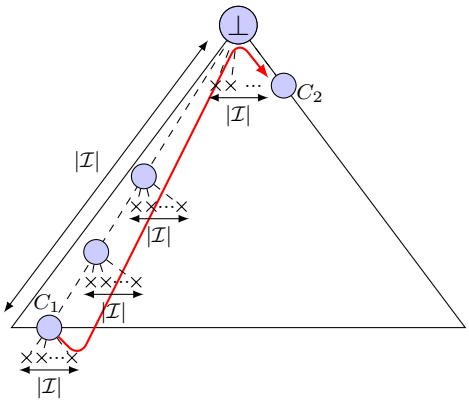


Figure 4.8: Time between two outputs (worst case).

the shared memory space. This way, a thread can generate work for other threads by depositing tuples in the TupleSpace. Then, when a thread is idle, it retrieves a tuple from the TupleSpace. By sharing the work between all cores, Melinda assures a good load balance.

Melinda is based on the Linda approach presented by Gelernter et al. in [Gel89]. While Linda accepts tuples with a heterogeneous structure, and allows retrieving tuples by querying on tuples values, Melinda works with fixed-structure tuples and does not allow querying on tuples values. These restrictions make Melinda a very low overhead framework than can handle millions of tuples, with a high throughput on tuple insertion/suppression.

PPerMiner is the parallel implementation of PerMiner using Melinda and is shown in Algorithm 9. Likewise, *pperIter* is the parallel implementation of *perIter* and is shown in Algorithm 10. Similarly to PLCM [NTMU10], we define a work unit as the processing of a pattern augmentation by *pperIter* function without the recursive calls. In order to render PerMiner algorithm parallel, the recursive calls (line 8 of PerMiner and line 22 of *perIter*) are replaced by the generation of a new tuple containing the current closed pattern, the reduced dataset and an augmentation, and its deposition in the TupleSpace (lines 14-15 of PPerMiner and 27-28 of *pperIter*).

Moreover, PPerMiner spawns as many threads as the parameter *num_threads* indicates, and each thread executes *ThreadFunction* function in Algorithm 9. This is, each thread retrieves a tuple at a time from the TupleSpace and invokes *pperIter* with its data. Once there is no more tuples to process and no thread is working, Melinda sends a termination signal to PPerMiner to end the program.

Depth-based cutoff

The parallelization strategy that we exploit for PPerMiner is a classical strategy for tree-recursive algorithms. One of the issues of this strategy is that due to the combinatorial nature of items, a substantial amount of tuples might be generated, which would increase the overhead introduced by Melinda.

Specially in the case of using low minimal support values, the amount of patterns gen-

Algorithm 9: Parallel Core Periodic Concept Miner

```
1  procedure PPerMiner (D, min_sup, num_threads, max_depth);
   Data: dataset D, minimum support threshold min_sup, number of threads
         num_threads, depth threshold max_depth
   Result: Output all Core Periodic Concepts that occur in D
2  begin
3      if |D| ≥ min_sup then
4          foreach i ∈ [1..num_threads] do
5              spawn ThreadFunction()
6          ⊥_clo ← ⋂_{t∈D} t
7          output (⊥_clo, {1..|D|/2}, D)
8          D_{⊥_clo} = {t \ ⊥_clo | t ∈ D}
9          foreach e ∈ I with e ∉ ⊥_clo do
10             tuple.pattern = ⊥_clo
11             tuple.dataset = D_{⊥_clo}
12             tuple.extension = e
13             tuple.el = ∅
14             tuple.depth = 0
15             put(tuple)
16         wait for all threads to finish

1  Function ThreadFunction()
2
3      begin
4          while get(tuple) do
5              perIter(tuple.pattern, tuple.dataset, tuple.extension, tuple.el, min_sup,
6                      tuple.depth,max_depth)
```

erated might be very large with a *deep* exploration of the enumeration tree. This deep exploration creates long patterns that are supported by few transactions. Due to the low number of transactions of these patterns, their computing time can be very fast. When the computing time is close to the overhead introduced by Melinda, it is no worth it any more to use the parallel version of the algorithm since it is more expensive than treating these patterns sequentially.

Therefore, the solution is to add a cutoff point based on the depth of the current pattern in the enumeration tree. Starting by depth 0 in PPerMiner, each successful augmentation increments the depth of the current pattern. Then, given a depth threshold *max_depth*, in line 22 of Algorithm 10, if the current depth value d is greater than *max_depth* a recursive call to *perIter* is made instead of generating a tuple for the TupleSpace. This is, the work unit is treated sequentially by the current thread. This way, the overhead introduced by Melinda is only paid once for the whole enumeration subtree.

Algorithm 10: Iterative CPC Generator

```
1  procedure perIter(X, D_X, e, el, min_sup, d, max_depth);
   Data: Itemset of a discovered CPC X, reduced dataset D_X, item e, exclusion list el,
         minimum support threshold min_sup, depth d, depth threshold max_depth.
   Result: Output all Core Periodic Concepts whose itemset is prefixed by X and whose
           transactions are in D_X, with minimal support min_sup.
2  begin
3      A := {e}
4      B := getPeriods(tidlist(A), min_sup)
5      B' := B \ {b | ∄b' ∈ B such that b.occs ⊂ b'.occs}
6      G := group(B')
7      S ← ∅
8      foreach g ∈ G do
9          A' := ⋂_{t∈g.occs} t
10         S := S ∪ (A', g.periods, g.occs)
11     S := filter(S);
12     new_el ← ∅
13     enum ← ∅
14     foreach (A', P, T) ∈ S do
15         if max_elem(A') = e then
16             Q = X ∪ A'
17             if el_test(Q, el) then
18                 output (Q, P, T)
19                 if Q ∉ enum then
20                     D_Q = reduce(D_X, Q, e, min_sup, min_red)
21                     foreach i ∈ I with i < e and i ∉ Q do
22                         if d ≤ max_depth then
23                             tuple.pattern = Q
24                             tuple.dataset = D_Q
25                             tuple.extension = i
26                             tuple.el = new_el
27                             tuple.depth = d + 1
28                             put(tuple)
29                         else
30                             perIter(Q, D_Q, i, el, min_sup)
31                     enum := enum ∪ Q
32                 new_el := new_el ∪ Q
```

Locality issues

An important aspect to take into account regarding Melinda's performance is cache locality. When a pattern Q is computed by PPerMiner, the pattern and its reduced dataset D_Q are stored in cache. These elements are going to be necessary for the computation of the generated tuples, i.e. the immediate children of Q in the enumeration tree. But, since the tuples are stored in the TupleSpace, any idle thread can retrieve one of the tuples that can benefit for data locality. If the thread is not executing in the same core, this benefit is lost.

Melinda tries to make use of this data locality when possible. The TupleSpace is decomposed in several queues, one per core. When a thread requests a tuple from the TupleSpace, Melinda retrieves in priority a tuple from the queue of the core where the thread is running. Only if the corresponding queue is empty, Melinda checks other queues for tuples. This method guarantees a best effort locality.

4.4 PerMiner's Soundness and Completeness

In this section, we are going to prove the soundness and completeness of PerMiner. And then, we are going to prove that PerMiner does not generate duplicate CPCs.

Theorem 4.4 (Soundness)**.**

All patterns returned by PerMiner *are Core Periodic Concepts.*

Proof by contradiction. Proving that all patterns returned by PerMiner are CPCs come to prove that the set S of patterns generated in lines 7-10 from Algorithm 3 is a set of CPCs since a subset of the set S is outputted in each execution of *perIter*, line 18 of Algorithm 3. For each A in *perIter*, the periods and closure are computed giving as a result the set S of CPCs. Let's consider that an element $M(I, P, T)$ of S is not a CPC. Following the definition of CPC (see Definition 3.7), if M is not a CPC it is because there exist another CPC $N(I', P', T')$ such that $I = I'$, $P' \subset P$ and $T \subset T'$.

Since M is generated on the branch of A, $A \subseteq I$, and therefore $A \subseteq I'$ since $I = I'$. The existence of the CPC $N(I', P', T')$ means that A is present in all transactions of T'. The function *getPeriods* in line 4 of *perIter* receives the set of transactions containing A and generates a set of tuples (period, transaction list).

Following Theorem 3.9, N being a CPC means that *getPeriods* generates tuples for each period in P', i.e. $\forall p' \in P' \; \exists (p', tidlist(A, p') = T') \in B$. Also, M belongs to S which means that the tuples $\forall p \in P \; (p, T)$ pass the filters in lines 5 and 6 of *perIter*. This means that all those tuples belong to B.

We have that $T \subset T'$, which means that line 6 of *perIter* will remove all tuples (p, T) with $p \in P$ from B since there exists at least one other tuple (p', T') with $p' \in P'$ such that $T \subset T'$. This process removes all tuples of M which means that M can not belong to S which is a contradiction of the hypothesis, therefore proving that all patterns contained in S (line 7 of *perIter*) are CPCs. □

In order to prove the completeness of PerMiner algorithm, we first prove the following lemma:

Lemma 4.5. *When given a CPC $C(X, P, T)$ and an augmentation $e \in \mathcal{I}$, perIter mines all CPCs who have (C, e) as first parent.*

Proof by contradiction. Let's suppose that, given a CPC $C(X, P_X, T_X)$ and an augmentation $e \in \mathcal{I}$, there exists a CPC $M(I, P, T)$ whose first parent is (C, e) that is not outputted by *perIter*. There are three possibilities:

1. CPCs with itemsets bigger than I are outputted, i.e. $\forall N(I', P', T')$ outputted $I \subset I'$. For this to happen, at least one tuple (P', T') in G (line 6) generates I' by intersection in line 9, i.e. $\bigcap_{t \in T'} t = I'$, and all tuples generating the itemset I by intersection do not belong to G, i.e. $\forall (P'', T'')$ such that $\bigcap_{t \in T''} t = I$, $(P'', T'') \notin G$. But, we know that at least the tuple (P, T) produces M by closure, i.e. $\bigcap_{t \in T} t = I$. Now, for this tuple not to belong to G means that either i) it has not been generated by *getPeriods* (line 4) or ii) it has been filtered out in line 5.

   i) *getPeriods* receives as input the transaction list where $X \cup \{e\}$ occurs. By hypothesis, $X \cup \{e\}$ is the first parent of I, therefore $X \cup \{e\} \subseteq I$. This means that $X \cup \{e\}$ occurs in all transactions in T, and maybe more, and $X \cup \{e\}$ is frequent in all periods in P. As M is not found, it means that *getPeriods* generates tuples (p, T''') $\forall p \in P$ with $T \subseteq T'''$. $T \subset T'''$ would generate by intersection a smaller itemset than I, i.e. $\bigcap_{t \in T'''} t = I''' \subset I$. This would mean that $X \cup \{e\}$ is the first parent of I''', which means that $X \cup \{e\}$ cannot be the first parent of I. This contradicts the hypothesis that says that $X \cup \{e\}$ is the first parent of I. Therefore, the only option possible is $T = T'''$ which means that *getPeriods* generates the tuple (P, T) if M exists.

   ii) Once the tuple (P, T) has been generated by *getPeriods*, it might be filtered out in line 5 of *perIter*. This would mean that there exists at least a tuple $(P', T') \in B$ such that $T \subset T'$. $T \subset T'$ implies that $P \supset P'$ since $\forall p' \in P'$ $(p', T') \in \mathcal{Y}$ and $T \subset T'$ means that $\forall p' \in P'$ $(p', T) \in \mathcal{Y}$, therefore the set of periods P of the tuple (P, T) will contain at least the set of periods P'. I occurs in all transactions in T and since $T \subset T'$, the intersection of the transactions in T' give an itemset smaller or equal than I, i.e. $\bigcap_{t \in T'} t = I'$ and $I' \subseteq I$. If $I' \subset I$, then $\perp \cup \{e\}$ is first parent of I' which makes impossible for $X \cup \{e\}$ to be the first parent of I. If $I' = I$, then there exists a periodic concept (I', P', T') such that $I = I'$, $T \subset T'$ and $P \supset P'$. The existence of this periodic concept makes impossible for M to be a CPC (see Definition 3.7). Therefore, (P', T') cannot exist and the tuple (P, T) cannot be filtered out in line 5 of *perIter*.

   The tuple (P, T) cannot be filtered out by the function *group* either since this function only groups periods that have the same transaction list and it does not remove any periods or transaction lists.

   Therefore, if M is a CPC and $X \cup \{e\}$ is its first parent, (I, P, T) is enumerated and belongs to the set S in line 10 of *perIter*. With M in the set S, no CPC with bigger itemset than I would pass the filter of line 11 of *perIter*, i.e. $\forall N(I', P', T')$ with $I \subset I'$, N is not outputted.

2. CPCs with itemsets smaller than I are outputted, i.e. $\forall N(I', P', T')$ outputted $I \supset I'$. If $N(I', P', T')$ with $I \supset I'$ exists, then $X \cup \{e\}$ is the first parent of I' and not the first parent of I. This contradicts the hypothesis.

3. No CPC is outputted by $perIter(X, D_X, e, min\_sup)$. We have already proven that if M is a CPC and $X \cup \{e\}$ is its first parent, M will belong to the set S in line 10 of *perIter*. Then, for M not to be outputted it has to be either filtered out by function *filter* in line 11 of *perIter*, or by the second or third part of the first parent test in lines 15 and 17 of *perIter*.

   M is filtered out by *filter* only if there exists $N(I', P', T')$ in S such that $I' \subset I$, which would mean that $X \cup \{e\}$ is the first parent of I' and not the first parent of I. This contradicts the hypothesis. Likewise, if M does not pass the second or third part of the first parent test in lines 15 and 17, it would mean that $X \cup \{e\}$ is not the first parent of I which contradicts the hypothesis.

This proves that if M is a CPC and $X \cup \{e\}$ is its first parent, then M is outputted by *perIter*. □

Theorem 4.6 (Completeness)**.**

PerMiner *returns the complete set of Core Periodic Concepts.*

Proof. Proving that PerMiner returns the complete set of Core Periodic Patterns comes to prove that PerMiner in Algorithm 2 generates the CPC that happens in all transactions of the database and that all the possible augmentations are explored, and then, proving that given a CPC (X, P, T), *perIter* in Algorithm 3 finds all CPCs who have $((X, P, T), e)$ with $e \in \mathcal{I}$ as first parent.

PerMiner in Algorithm 2 outputs the CPC that happens in all transactions of the database in line 5 that has been calculated in line 4 by intersecting all transactions in the dataset. The only way this CPC would not be outputted would be if the size of the dataset was smaller than the minimum support threshold in which case the dataset contains no CPCs.

Then, PerMiner explores all possible augmentations of this CPC since all items in $\mathcal{I}$ that do not belong to the CPC's itemset are used as augmentations. Indeed using as augmentation an item already present in the CPC's itemset would generate the same CPC which would be a duplicate.

We have already proven in Lemma 4.5 that, given a CPC (X, P, T) and an augmentation $e \in \mathcal{I}$, *perIter* mines all CPCs whose first parent is $((X, P, T), e)$. Therefore, the first invocation of *perIter* is made with all possible augmentations of the CPC calculated in PerMiner, which mines all CPCs whose first parent is $((\perp_{clo}, \mathcal{P}, \mathcal{D}), e)$ with $e \in I \setminus \perp_{clo}$. It is thus guaranteed that all CPCs having $\perp_{clo}$ as first parent are found with all possible augmentations. The only possibility to "miss" a CPC would be to miss an augmentation, but they are exhaustively explored by line 7 of Algorithm 2. Then, *perIter* is applied recursively to these CPCs until no new CPC can be generated.

Therefore, we have proved that PerMiner returns the complete set of Core Periodic Concepts. □

Theorem 4.7 (No Duplicates)**.**

PerMiner *does not generate duplicate CPCs.*

Proof. Proving that PerMiner does not generate duplicate CPCs comes to prove that *perIter* does not generate duplicates. This is because the CPC generated by PerMiner algorithm in line 5 of Algorithm 2 is only outputted once and then augmented using all possible augmentations. Therefore, PerMiner does not generate the CPC contained in all transactions of the dataset more than once.

[Proof by contradiction] Let's suppose that *perIter* generates a duplicate of the CPC $M(X, P, T)$. This means that the first parent test in *perIter* algorithm (lines 11,15 and 17 of Algorithm 3) has not blocked M from being outputted in line 18 of Algorithm 3 while it had already been outputted by another branch of the enumeration tree.

The enumeration technique and the first parent test (line 15 of Algorithm 3) assure that each itemset is generated only by the biggest augmentation of its first parent, exactly in the same way as LCM. Therefore, as long as there is only one CPC generated by *perIter* invocation, similarly to classical itemset enumeration algorithms, no duplicates are generated.

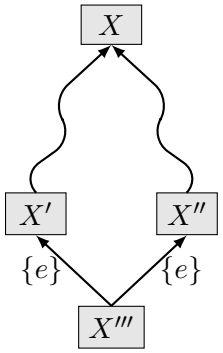


Figure 4.9: Enumeration generating duplicates.

Thus, M can have been generated only in a branch where in at least one occasion, more than one CPC were generated by the same invocation of *perIter*. In this case, at least two CPCs with itemsets X' and X'', with $X' \subset X$ and $X'' \subset X$, were generated by the same invocation of *perIter* over the same augmentation $\{e\}$ of an itemset X''', as shown in Figure 4.9. We know that $X' \nsubseteq X''$ and $X'' \nsubseteq X'$ since if not the function *filter* would have been filtered out one of them, as shown in Algorithm 7. Then, these two CPCs where further expanded until both generated $M(X, P, T)$.

Let's suppose that *perIter* is invoked first using itemset X'. After a certain number of augmentations, M is generated and outputted. Then, the itemset X' is added to the exclusion list *new_el* before invoking *perIter* with the following itemset, which would be X''.

After that, perIter is invoked using the itemset X'' with the exclusion list *new_el* as parameter. After a certain number of augmentations, the union in line 16 of Algorithm 3 would give $Q = X$. Then, the first parent test in line 17 of Algorithm 3 would not pass since $X' \in el$ and $X' \subset X$, as can be seen in Algorithm 6.

Therefore, M could not have been generated more than one by *perIter* which proves that PerMiner does not generate duplicate CPCs. □

5 Scalability Experiments

PerMiner algorithm was implemented in C++ and run on a multi-processor computing server with four Intel Xeon X7560 processors (8 cores each) at 2.27 GHz with 64 GB RAM and 32 cores. In this chapter, we evaluate the efficiency of the PerMiner algorithm over synthetic and real world datasets.

Concretely, in Section 5.1, we present a comparative analysis based on synthetically generated data. Then, in Section 5.2, we carry out experiments over a real multimedia application execution trace in order to evaluate the efficiency of PerMiner algorithm over real data. In Section 5.3, a brief experimental evaluation of PerMiner parallel scalability is carried out over a real multimedia execution trace. Finally, in Section 5.4, we conclude this chapter.

5.1 Comparative Analysis on Synthetic Data

In this section, we compare the efficiency of PerMiner algorithm with the algorithm presented in [LCBT+12], called *3-STEP* from now on. As will be shown in Chapter 9, no previous study has considered mining a condensed representation of the set of periodic patterns considering both the redundancy of the periods and the items. Therefore, to the best of our knowledge, there does not exist any other available algorithm with which we could compare PerMiner. This is the reason why PerMiner algorithm is compared to a previous version of the algorithm presented in [LCBT+12].

3-STEP consists of a first step that generates all frequent triples from a dataset, a second step that mines the whole set of triadic concepts, using the DataPeeler algorithm [CBRB09], and a third step that extracts the set of CPCs. Algorithms for all steps were implemented in C++. Our objective is to compare the scalability of PerMiner and *3-STEP* when varying main characteristics of the data.

The datasets used in this analysis have been artificially generated using a modified version of IBM Quest Synthetic Data Generator [IBM]. We have implemented this modified version by adding a periodic pattern generation algorithm in order to include periodic patterns with irregular gaps in the dataset. The size of the periodic patterns is determined based on a Poisson distribution with mean μ_1. The period of each pattern follows a Poisson distribution

with mean μ_2. Items forming the pattern are randomly chosen from the list of available items.

To generate gaps in the periodicity, we use a parameter α, which is uniformly distributed between 0 and 1. With probability α, the next transaction will belong to the pattern. Otherwise, a gap is introduced, whose length is based on a Poisson distribution whose mean is the value of the period of the pattern.

We begin by varying the total number of items from 100 to 1000 items. The minimum support threshold is set to 10% over a dataset of 1000 transactions and an average length of transaction of 10 items. 20 periodic patterns are generated with 6 items of length ($\mu_1 = 6$) and a period of 4 ($\mu_2 = 4$). The parameter α is set to 0.9. Figure 5.1 contains two plots: the wall clock time of the executions against the total number of items on the left, and the number of CPCs generated against the total number of items on the right. Note that for 100 and 200 items experiment *3-STEP* could not finish in less than 10^5 seconds and therefore they are not shown in the figure.

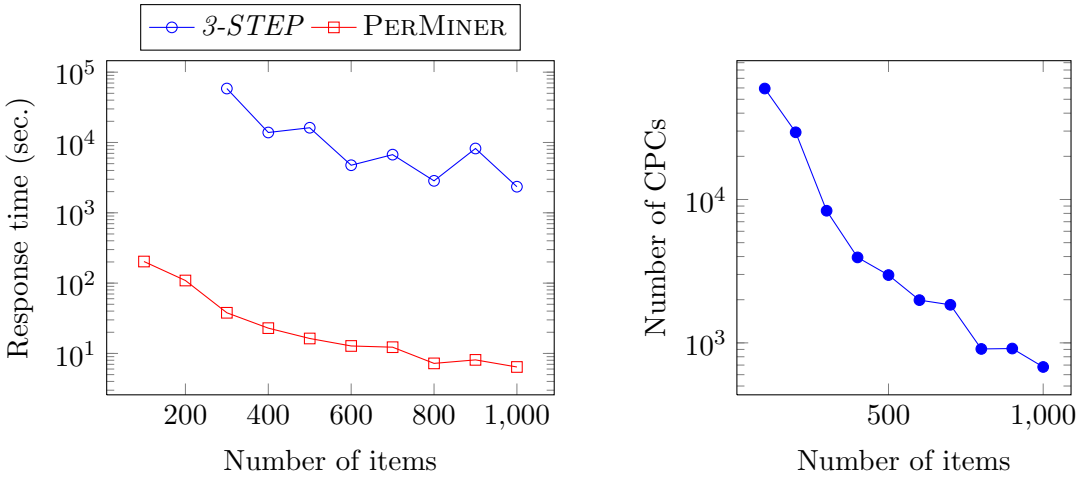


Figure 5.1: Number of items comparative plot

As we can observe in the left-side Figure 5.1, in these tests PerMiner is faster than *3-STEP* by two to three orders of magnitude. It is important to notice that the response time decreases when the number of items increases. This can be explained by the fact that the transactions are relatively short, only 10 items per transaction, and therefore, by increasing the number of possible items, the number of periodic patterns decreases (as shown in the right-side Figure 5.1).

We continue by varying the total number of transactions, from 1000 to 10000 transactions. We reuse the test parameters of the previous test while fixing the set of items to 200 distinct items. Figure 5.2 contains two plots: the wall clock time of the executions against the total number of transactions on the left, and the number of CPCs generated against the total number of transactions on the right. Note that for any of the experiments *3-STEP* could not finish in less than 10^5 seconds and therefore they are not shown in the figure.

The fact that *3-STEP* does not manage to execute in less than 10^5 seconds is due to the high number of CPCs generated in this experiments, starting at 29500 CPCs in the case of 1000 transactions. The more CPCs there is, the more patterns *3-STEP* has to clean from the

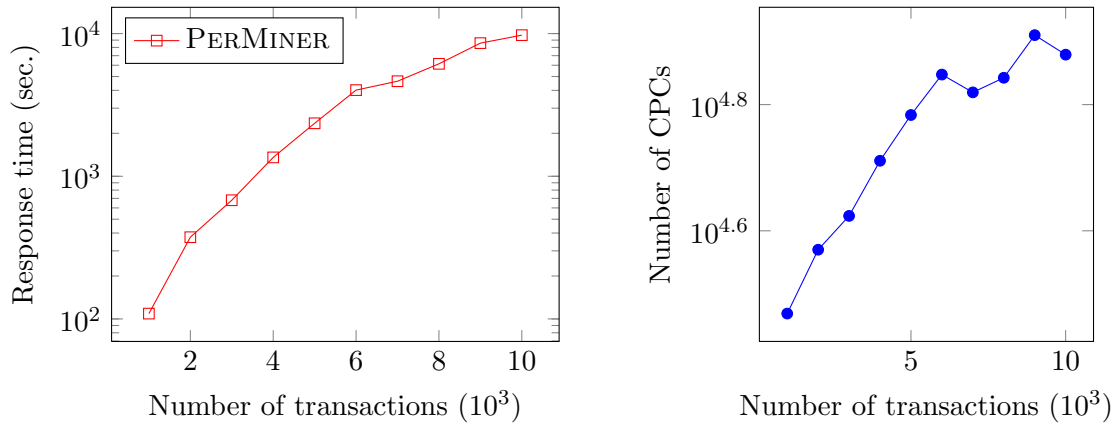


Figure 5.2: Number of transactions comparative plot

output of DATA-PEELER since DATA-PEELER generates a high number of redundant patterns, patterns that are pruned much earlier by PERMINER algorithm. This curve confirms that PERMINER's time complexity is polynomial in the number of transactions.

Finally we vary the minimum support threshold, from 10% to 90%. We reuse the test parameters of the previous test while fixing the set of items to 200 distinct items and the number of transactions to 10000. For this test the period of the periodic patterns is set to 2 ($\mu_2 = 2$) and the parameter α is set to 1. These modifications are necessary in order to test high values of support since they offer a big degree of periodicity with no disruptions. Figure 5.3 plots the wall clock time of the executions against the minimum support threshold on the left, and the number of CPCs generated against the minimum support threshold on the right.

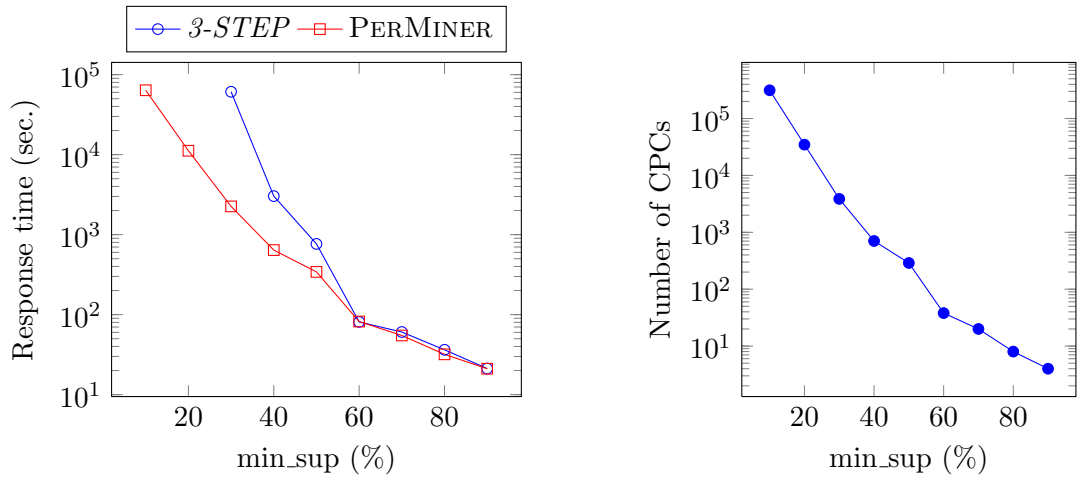


Figure 5.3: Minimum support threshold comparative plot (synthetic data)

In this experiment, shown in Figure 5.3, the number of CPCs generated for supports bigger than 50% is so low (less than 40 CPCs) that both algorithms present very close execution times. Here it is important to note that high values of support would generate CPCs with

very low period values. Indeed, high values of support would give patterns that are present in most of the transactions of the database, which is not the purpose of this approach but more the purpose of frequent itemset mining algorithms. This is why *3-STEP* is not able to output results in less than 10^5 seconds for support values less than 30%.

Then, the smaller the support the biggest the difference between the running time of the two algorithms is: for a minimum support value of 30% PerMiner is faster than *3-STEP* by two orders of magnitude. As explained before, the running time of *3-STEP* algorithm is affected drastically by the number of CPCs. Over small supports, a huge number of CPCs are generated (due to the big degree of periodicity induced during data generation) which makes the running time of *3-STEP* algorithm increase drastically, while the running time of PerMiner algorithm is polynomial with respect to the number of CPCs.

5.2 Comparative Analysis on Real Data

In these experiments we have used an execution trace of a video and audio decoding test application, called *HNDTest*, running on a *STMicroelectronics* development board. This test application is used to test the multimedia middleware in G3 platforms. The execution trace consists on a log of the application and system execution containing a line for each event. Each line contains the timestamp and the description of the event. Since the input for our algorithm is a transactional database, we need to pre-process the execution trace by splitting it into time intervals. All events executed during the same time interval will belong to the same transaction on the database.

The execution trace used on these experiments contained 528360 events, with 72 distinct items, and it was split into time intervals of 10ms. We obtained 15000 transactions with an average of 35 items per transaction. Figure 5.4 plots the CPU time of the executions against different minimum support thresholds on the left, and the number of CPCs generated against the minimum support threshold on the right. Note that for a support of 10% *3-STEP* could not finish in less than 10^5 seconds and therefore they are not shown in the figure.

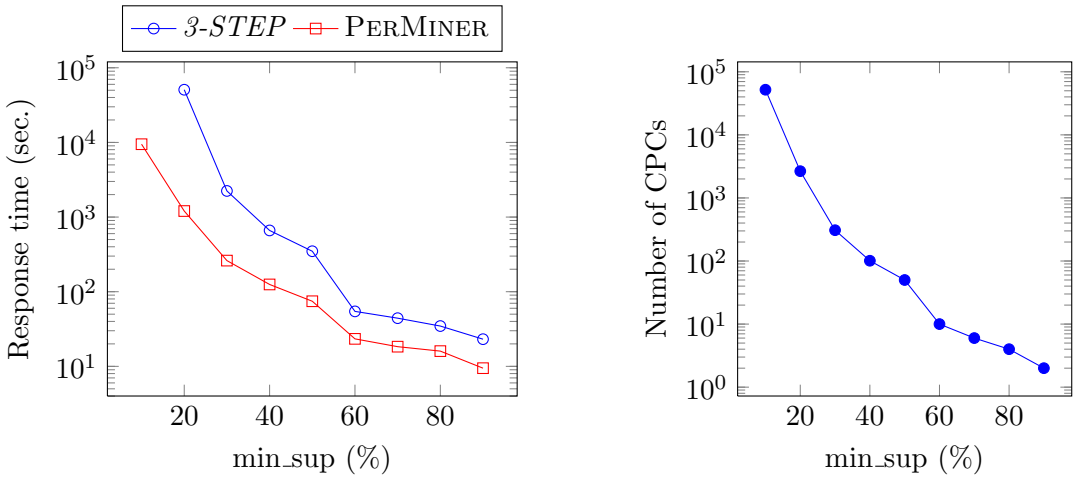


Figure 5.4: Minimum support threshold comparative plot (real data)

We can observe in Figure 5.4 that PerMiner is one to two orders of magnitude faster than *3-STEP* for support values where *3-STEP* was able to output results in less than 10^5 seconds. This experiment shows that PerMiner is well suited for analyzing real multimedia execution traces, and that is the only algorithm that can compute the set of CPCs with low support values in a reasonable amount of time. Such low support values are the only ones that offer the possibility of performing a fine-grained analysis of the application execution.

5.3 Experimental evaluation of PerMiner's parallelism

In Chapter 4, we have introduced a parallel version of PerMiner algorithm that is able to exploit nowadays multicore platforms in order to reduce PerMiner algorithm's execution time. In order to measure the efficiency of the parallel version we are going to measure its speedup.

The speedup is obtained by dividing the time of the sequential version by the time of the parallel version using n threads.

$$speedup_n = \frac{sequential\ execution\ time}{execution\ time\ with\ n\ threads} \tag{5.1}$$

Then, a parallel version of an algorithm has a good speedup if its speedup is linear to the number of threads, i.e. $speedup_n = n$.

Therefore, here we present an experiment that shows the speedup of the parallel version of the algorithm from 4 to 32 cores. For this experiment we reused the dataset used in the previous experiment and fixed the minimum support value to 10%. Figure 5.5 plots the CPU time of the executions against the number of threads on the left, and the speedup against the number of threads on the right.

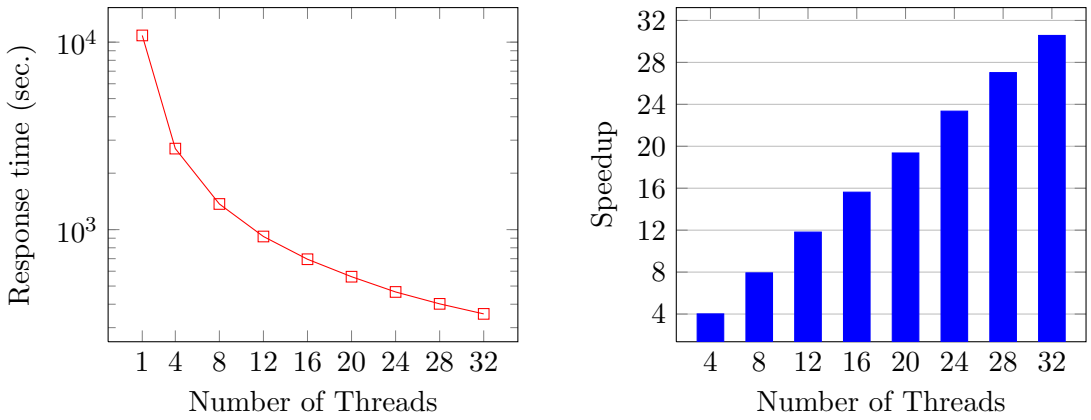


Figure 5.5: Parallelism evaluation

In Figure 5.5 we can observe that the parallel version of PerMiner algorithm obtains a speedup almost linear to the number of threads: PerMiner has excellent parallel scaling capabilities.

5.4 Conclusions

In this chapter, we have presented a comparative analysis on both synthetically generated and real data, of PerMiner algorithm against *3-STEP* algorithm, presented in [LCBT+12].

Through experiments we have seen that thanks to its output polynomial complexity PerMiner is consistently two to three orders of magnitude faster than the state of the art. The results have shown that PerMiner is the only algorithm that can efficiently mine CPCs from complex datasets with low support values.

Moreover, we have shown that PerMiner presents excellent parallel scalability on multicore architectures, allowing to further reduce mining time or handle even more complex datasets.

In Chapter 8 we present a couple of use cases where an analysis of the results of executing PerMiner over real multimedia application traces is carried out. This shows that PerMiner algorithm is not only efficient but also useful in the analysis of multimedia application traces.

Part II

Embedded Systems Contribution

Patter Mining techniques are widely used to automatically analyze big amounts of data, and extract useful information from them. These techniques are well known and understood by data mining experts but not so much by the end user of these techniques. At the end of the day these techniques are created to be used over real data and the only ones capable of analyzing the output of pattern mining techniques are the domain experts. Nevertheless, pattern mining techniques need certain input from the user that is going to help deciding what patterns are interesting. Therefore, the end user needs to be able to provide this input in order to obtain useful output.

Therefore, in Chapter 6, we present a first step towards a methodology to allow developers to use our approach to debug multimedia application execution traces. This methodology covers the complete process, from the preprocessing of the execution trace in order for it to be exploitable by PerMiner algorithm, to result analysis. This involves taking certain decisions that are not obvious for developers, thus we are going to give some guidelines to help developers in taking the good decisions that will allow them to progress in the debugging of their software.

Then, in Chapter 7, we present *CPCViewer*, a visualization and analysis tool of the result set of core periodic concepts outputted by PerMiner algorithm. Finally, in Chapter, 8, we present several use cases that show how the methodology presented in Chapter 6 is used over real multimedia application execution traces.

6

Towards a Methodology for Debugging Multimedia Embedded Applications through Periodic Pattern Mining

Generally, software developers are not familiar with data mining techniques, such as the proposed periodic pattern mining technique, and therefore might be reticent to use these new techniques. Indeed, a methodology is needed in order to give developers the necessary guidelines that are going to allow them to use our approach to analyze their multimedia application execution traces.

As can be seen in Figure 6.1, the workflow of this methodology consists in three main steps: (1) the preprocessing of the execution traces into transactional databases, (2) the mining process, and (3) the postprocessing of the results.

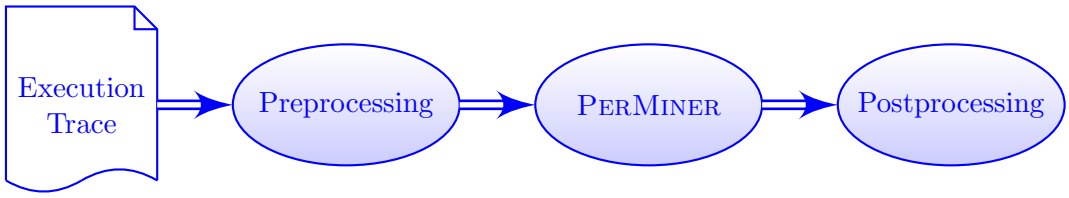


Figure 6.1: Methodology Workflow

Thus, in this chapter, we present a first step towards a methodology that explains how developers can use our solution to analyze multimedia application execution traces, from the raw execution trace to the analysis of the mining results. Concretely, in Section 6.1, we show how to preprocess the execution traces in order to obtain a transactional database exploitable by PerMiner algorithm (see Chapter 4). Then, in Section 6.2, we give guidelines to better choose the value of minimum support threshold to be used in PerMiner algorithm. Next, in Section 6.3, we discuss how the analysis of PerMiner results can be carried out. Concretely, we study how visualization can help to understand PerMiner results by using the visualization tool presented in detail in Chapter 7. Then, we present a competitors finder tool that finds pairs of patterns that are in competition, which might help to discover conflicts between different system components. Finally, in Section 6.4, we conclude this chapter.

6.1 Preprocessing of execution traces into transactional databases

An execution trace is a sequence of events registered during the execution of an application. However, most pattern mining algorithms search for patterns common to a sequence or a multiset of sets of elements [AS94], i.e. a transactional database. In our context, a transactional database is a sequence of sets of elements since the order and distance between the transactions plays an important role in our approach. Therefore, in order to render execution traces exploitable by pattern mining algorithms such as PerMiner, the execution trace has to be split into a sequence of sets of events. Here, we propose two possible methods to split the trace:

Time Interval First proposed in [LCSZ04], it consists in grouping in the same transaction all events of the trace registered in the same time window. Indeed, this method is related to the platform's performance since the architectural characteristics of the platform, such as the processor clock frequency, define the frequencies of the events present in the trace.

Function Name It consists in grouping in the same transaction all events of the trace that occurred between two occurrences of a given function. This method is related to the software being analyzed, since the software defines which functions are important for the analysis.

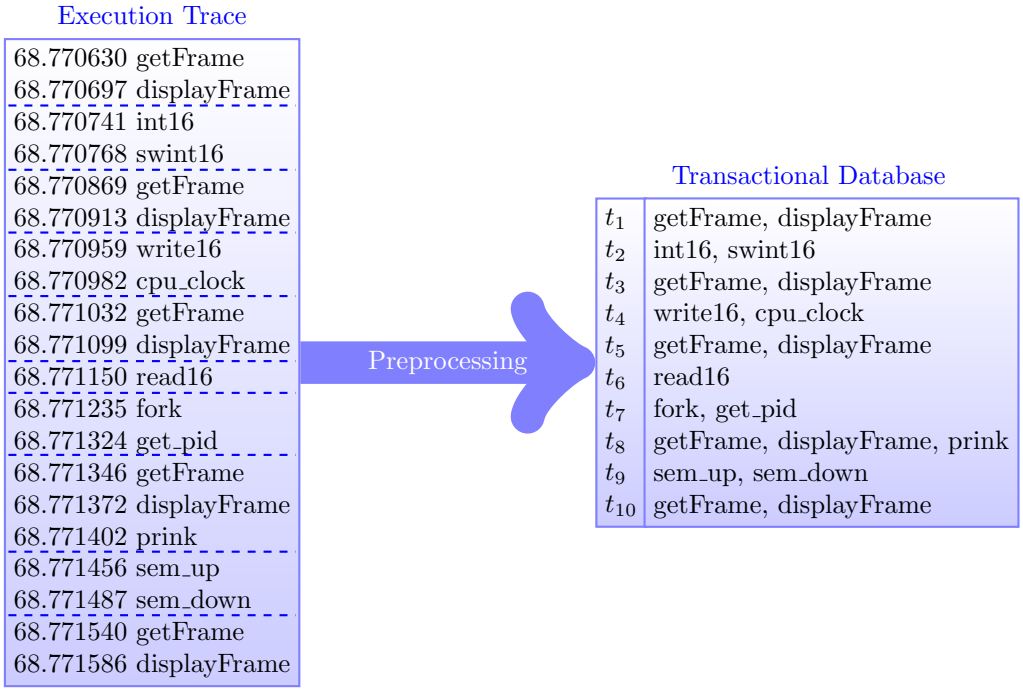


Figure 6.2: Preprocessing of an execution trace by splitting it into time intervals of 0.1 ms. Timestamp: seconds.microseconds

As an example, in Figure 6.2, we can observe the result of splitting a trace using a time interval of 0.1 ms.

Event Order. Our approach is focused on discovering sets of events that occur together periodically in the execution, but the order in which those events are executed is not taken into account, i.e. the sets $\{write16, cpu\_clock\}$ and $\{cpu\_clock, write16\}$ are considered the same set. Due to this fact, the timestamp of the events is not included in the transactions, so the information about when exactly an event was executed is not considered in the analysis. Moreover, considering that the executing environment is multi-threaded, the order of the events might change according to decisions taken by the scheduler. Therefore, even if the order is not taken into account, our approach is able to discover interesting relationships between events in the execution trace.

For a more exhaustive analysis where the order of event execution is considered important, there exist pattern mining techniques that can discover ordered sets of events [PHMa+01] [ZXHW10], called sequences, that occur frequently in the trace. Nevertheless, these techniques are more complex and computationally expensive. The results obtained by these techniques would be useful to analyze aspects of the execution different from those analyzed in our context. Therefore, these techniques are out of the scope of this thesis.

Event information. Generally, the minimum amount of information specified by event is the event name/type and its timestamp, as can be seen in Figure 6.2. Then, for each event in the trace, more parameters can be included in the trace such as the PID of the process that generated the event, the arguments of a function if the event is a function invocation, and so on. In data mining, each element of a transactional database is considered as a unique identifier of an event, e.g. *getFrame* in Figure 6.2. Therefore, if two events with the same name/type need to be differentiated, some extra information needs to be added to the corresponding elements in the database.

For instance, consider the case where it is important to know which process generated each event in the trace. In that case, the Process Identifier (PID) of the process could be attached to the event name. As an example, consider the execution trace shown in Figure 6.3 where the PIDs of the processes executing each event are part of the event information. The event *"68.770630 178 getFrame"* could be represented as *178_getFrame* in the transactional database. Then, the invocations of the function *getFrame* by process 178 would be considered as different elements in the database from the invocations made by process 135.

This way developers can apply a first filter to the execution trace in order to choose the level of detail of each event in the transactional database. It is important to note here that theoretically, for the same trace, the more distinguishable elements there are in the database, the slower PerMiner algorithm is going to run and the less amount of patterns will be generated. This is because PerMiner algorithm is exponential respect to the number of items. Therefore, when the events are too specialized, developers might find that none or very few patterns are generated by PerMiner algorithm since the data becomes too sparse.

For instance, if a minimum support threshold of 3 transactions is specified as input of PerMiner algorithm, from the transactional database in Figure 6.2 the core periodic concept (CPC) $(\{getFrame, displayFrame\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$ would be outputted by the algo-

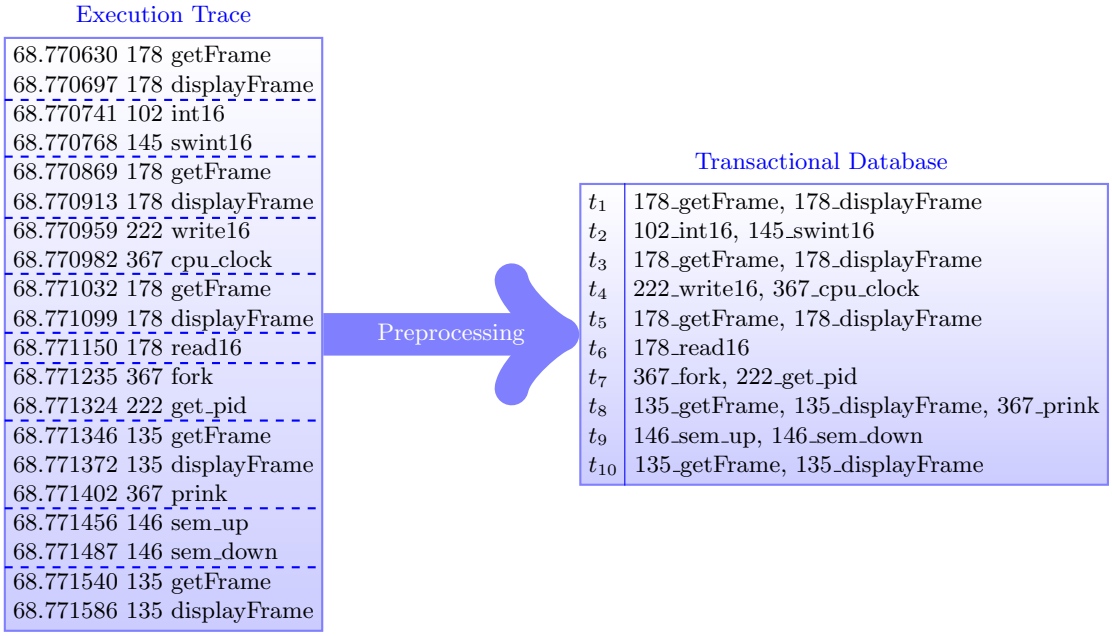


Figure 6.3: Preprocessing of an execution trace with a time interval of 1ms and attaching the PID of the process to the events. Event: Timestamp PID EventType.

rithm, since the itemset $\{getFrame, displayFrame\}$ occurs at a period of two transactions between transactions t_1, t_3 and t_5 in the first cycle, and t_8 and t_{10} in the second cycle (see Definition 3.1 in Section 3.1 for details). On the other hand, from the transactional database in Figure 6.3 the CPC $(\{178\_getFrame, 178\_displayFrame\}, \{2\}, \{t_1, t_3, t_5\})$ would be outputted by the algorithm, but the CPC $(\{135\_getFrame, 135\_displayFrame\}, \{2\}, \{t_8, t_{10}\})$ would not be outputted since it is present in only two transactions.

In conclusion, developers should decide which information is relevant for each event in the trace, always trying not to overspecialize the events in order to obtain meaningful results.

Splitting the trace. Choosing the way to split an execution trace in order to obtain a transactional database is a delicate task that is going to affect drastically the set of results obtained at the end of the mining process. Indeed, a certain degree of domain specific knowledge is necessary in order to correctly choose the splitting parameters. Regarding multimedia application execution traces, developers could use their knowledge about frame decoding, including the timings and functions involved in the process. On the other hand, system designers could use their knowledge about the platform itself in terms of timings of the different components.

For instance, a good start point of the analysis of a trace would be to split the trace using the name of the function that starts the decoding of a frame. This value might be considered coarse-grained but it allows the developer to analyze the execution on a wide perspective.

This would group the events related with the decoding of an individual frame on the same transaction. Then, the mining process would discover recurrent sets of events that might perturb the periodicity of the decoding of frames.

Another option would be to split the trace using a completely different approach, such as a short time interval, e.g. a tenth of the decoding rate. In the result dataset, the mining process would find periodicities that were hidden by the big decoding loop. Developers might find perturbations in these periods or even some periodicities that were not expected and would need a more detailed analysis.

Nevertheless, developers might have to change the granularity used to split the trace during the analysis of an execution trace, which might start by using a coarse-grained granularity in order to discover big perturbations on the application execution, and might change to more fine-grained granularity in order to discover the cause of these perturbations.

Additionally, developers might come up with new methods to split the trace thanks to their domain specific knowledge. An example might be an hybrid method considering two function names as the start and the end of the transaction, but allowing a certain degree of tolerance in the timings of these events, as shown in Figure 6.4.

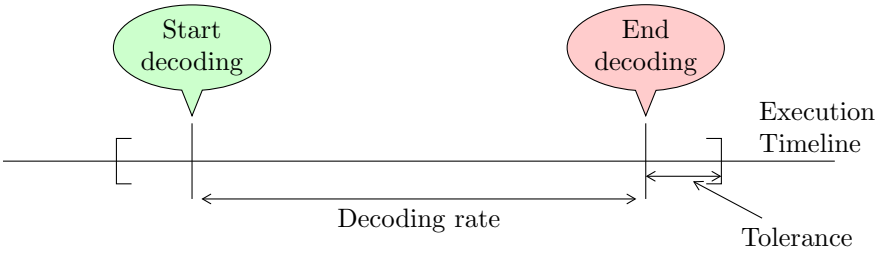


Figure 6.4: Function name with tolerance.

6.2 Mining Core Periodic Concepts

Once an execution trace has been transformed into a transactional database by the method introduced above, it can be exploited by PerMiner algorithm. As explained in Section 2.5, most frequent pattern mining algorithms, including PerMiner, require as input a minimum support threshold that specifies the minimum number of transactions in which a pattern must occur to be considered frequent. This threshold is a way of telling the algorithm which patterns we are interested in.

In PerMiner algorithm, the minimum support threshold indicates the minimum number of transactions a core periodic concept (CPC) should have on its transaction list in order to be outputted by the algorithm. Therefore, the minimum support threshold is independent from the value of the periods since its value is not modified according to the period of a CPC.

When mining CPCs for the first time from a transactional database, the minimum support threshold should be set fairly high in order to limit the set of results to patterns found in a big part of the execution. This allows developers to do a quick first analysis of the results, since the higher the minimum support threshold is, the quicker PerMiner algorithm execution is. The minimum support threshold being high, the set of results might not contain any pattern with any perturbation but it gives developers a first idea of the execution. Indeed, the set of results should contain only expected patterns, groups of events that were expected

to be found together with a given period, that will help developers to check whether the main functionality seems to work correctly.

If no perturbation was found in this early stage of the analysis, developers should gradually decrease the value of the minimum support threshold to gradually consider the new patterns mined by PerMiner. Indeed, while developers are carrying out the analysis of a set of results, a more detailed set of results might be mined by PerMiner algorithm.

6.3 Postprocessing of PerMiner's output

A manual analysis of PerMiner results in their text format would be nearly as long as manually analyzing the execution trace itself. Therefore, during the postprocessing of PerMiner results analysis tools should be use in order to facilitate the analysis of these results. In this section, we propose two analysis tools: a CPC visualization tool and a competitors finder tool.

Visualization and Analysis of Periodic Patterns

As most frequent pattern mining algorithms, PerMiner outputs the discovered patterns in text format that can be stored in a text file. Each line of the file contains a CPC, i.e. a list of items, followed by a list of periods and finally a list of transactions, all lists delimited by parenthesis and the different elements separated by commas. For example, the CPC $\{\{a, b\}, \{2, 4\}, \{t_1, t_3, t_5, t_7, t_9, t_{11}, t_{13}\}\}$ would be presented as follows:

```
(a,b) (2,4) (1,3,5,7,9,11,13)
```

As can be seen, it is not easy to analyze the periodicity of a pattern over a set of numbers (transaction identifiers). Moreover, the number of patterns outputted by PerMiner can be large. Therefore, a visualization tool can be useful in order to analyze the set of CPCs outputted by PerMiner algorithm. Indeed, one of the contributions of this thesis consists in a CPC visualization and analysis tool, called *CPCViewer*, presented in detail in Chapter 7. *CPCViewer*, shown in Figure 6.5, graphically visualizes the periodicity of each pattern facilitating its analysis.

As has been said, the output of PerMiner algorithm can be stored in a text file. This text file is the input of *CPCViewer*. CPCViewer reads the file and shows: on the top part, the list of itemsets forming part of the set of CPCs, and on the bottom part, the periodicity of a selected CPC.

At the beginning of the analysis process of a set of CPCs, developers should study the list of itemsets presented in the top part of the visualization tool in order to identify set of events that were expected to be found together. By selecting each of these sets of events, their periodicity can be analyzed on the bottom part of the visualization tool. Developers should analyze whether the expected periodicity for each set of events was respected. If a gap in the periodicity is discovered, then developers should analyze in detail, possibly by using an execution trace visualization tool, the part of the execution trace corresponding to that gap in order to discover what caused it.

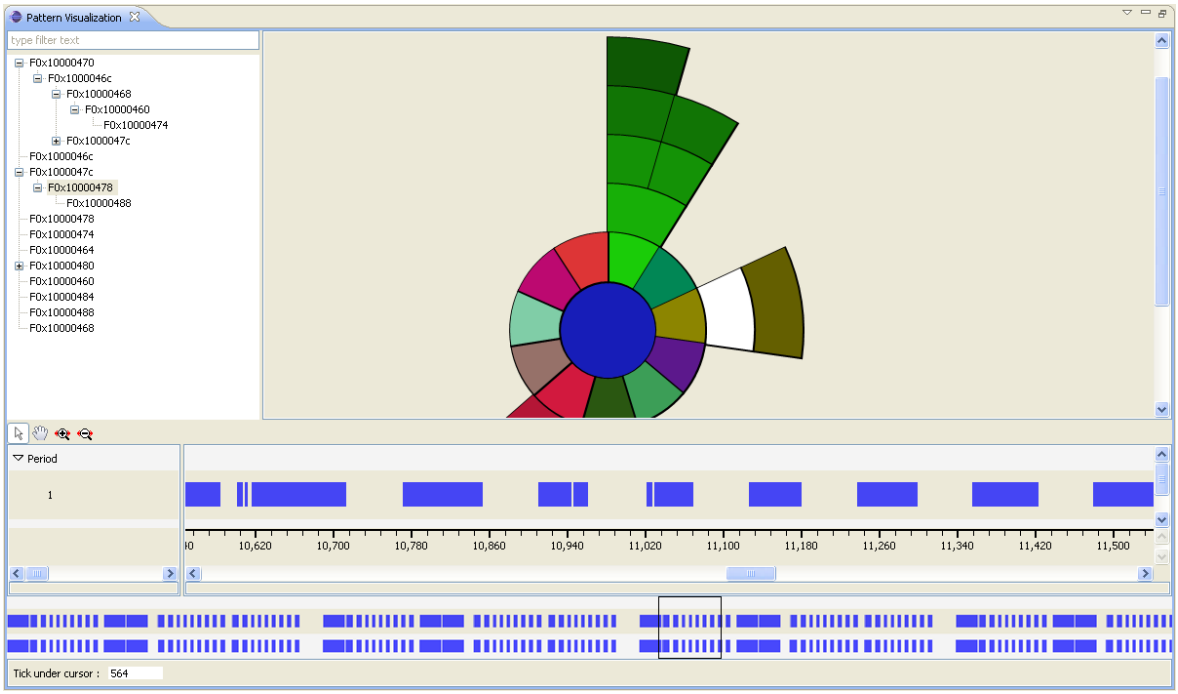


Figure 6.5: Periodic pattern visualization tool

Once the known or expected sets of events have been analyzed, developers should then study the rest of itemsets in the itemset list in order to identify sets of events that were not supposed to be found together. This might help to identify certain anomalies on the application execution. Then, the analysis of the periodicity of each set of events points developers to parts of the execution where those sets of events were found together. With this information, developers should analyze in detail the corresponding parts of the execution trace to decide whether it is an anomaly or a normal behavior.

Last, developers should analyze the rest of the itemsets in order to discover hidden periodicities or relationships between different parts of the system. Moreover, by following this process, developers gain a deeper knowledge of the system.

The set of CPCs might also be analyzed by other automated analysis tools to discover relationships between the CPCs that are difficult to identify on the proposed visualization tool. As an example, next, we present a **competitors finder** tool, which is an automated analysis tool that discovers pairs of CPCs that are in *competition*.

Competitors Finder

Here, we introduce a new analysis tool called *Competitors Finder* that helps to identify pairs of *competitor* patterns from the set of core periodic patterns outputted by PerMiner. Then, we give some guidelines about how this tool should be used during the analysis of a set of CPCs. This tool allows developers to easily identify possible conflicts between different parts of the system, i.e. between the application and the operating system, between different modules or drivers of the operating system, and so on, saving a significant amount of time to

developers. In this context, we consider that the value of *competition* between two patterns is the inverse of the overlap between the two patterns. As an example, Figure 6.6 shows a pair of competitor patterns, i.e. one pattern occurs during the gaps of anther pattern.

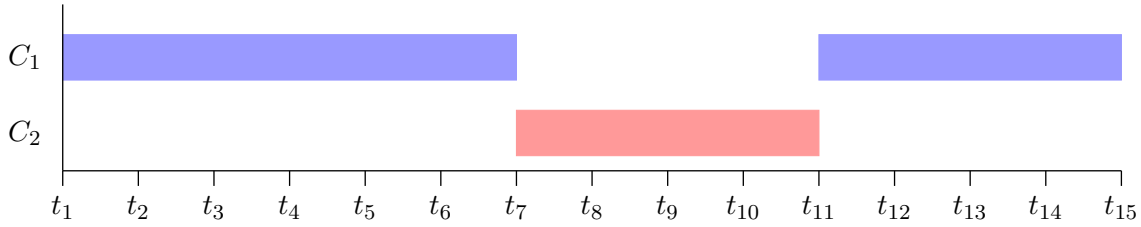


Figure 6.6: Example of a pair of competitors

Algorithm 11: Competitors Finder

```
1 procedure CompetitorsFinder(CPC, num_trans, min_ratio);
  Data: Set of CPCs CPC, Database length num_trans, minimum ratio of competition
        min_ratio
  Result: Set of pairs of competitors MaxComp
2 forall the C, C' ∈ CPC with C ≠ C' do
3     if Not yet checked pair (C', C) then
4         comp_ratio := competition_ratio(C, C', num_trans);
5         if comp_ratio ≥ min_ratio then
6             output(C, C', comp_ratio);
7 end procedure;
```

Competitors Finder Tool. The procedure *CompetitorsFinder* in Algorithm 11 receives a set of core periodic concepts and a minimum ratio of competition *min_ratio*. For each pattern, this procedure compares it with all other patterns and calculates the competition ratio by invoking *competition_ratio* function in Algorithm 12. If the competition ratio is bigger than *min_ratio*, then the pair of CPCs is outputted as well as their competition ratio.

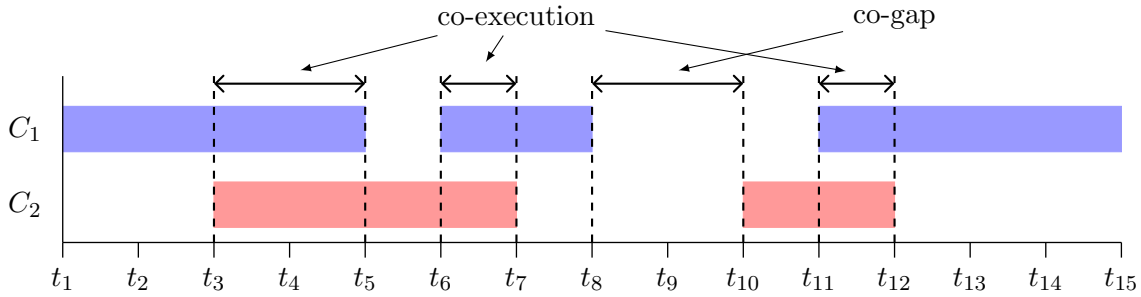


Figure 6.7: Competitors Finder Example

The function *competition_ratio* in Algorithm 12 calculates the competition ratio between two core periodic concepts. For this, the set of cycles of both core periodic patterns are

Algorithm 12: Competition Ratio Calculator

```
1  function competition_ratio(P, P', num_trans);
   Data: Two core periodic patterns P and P', Database length num_trans
   Result: The ratio of competition between P and P'
2  C = getCycles(P);
3  C' = getCycles(P');
4  coexec = 0;
5  cogap = min(C[0].start, C'[0].start);          /* Gap before first cycles (a) */
6  i = 1;
7  j = 1;
8  while i ≤ C.length AND j ≤ C'.length do
9      if C'[j].end < C[i].start then                                   /* (b) */
10         cogap = cogap + min(C[i].start, C'[j + 1].start) − C'[j].end;
11         j + +;
12     else if C'[j].start ≤ C[i].start AND C'[j].end ≤ C[i].end then   /* (c) */
13         coexec = coexec + C'[j].end − C[i].start;
14         j + +;
15     else if C'[j].start ≥ C[i].start AND C'[j].end ≥ C[i].end then   /* (d) */
16         coexec = coexec + C[i].end − C'[j].start;
17         i + +;
18     else if C'[j].start ≥ C[i].end then                              /* (e) */
19         cogap = cogap + C'[j].start − C[i].end;
20         i + +;
   /* Gap after all cycles (f)                                                */
21 cogap = cogap + num_trans − max(C[C.length].end, C[C.length].end);
   /* Total competition ratio = cogap + coexecution                           */
22 competition = cogap + coexec;
23 return (1 − (num_trans − competition)/num_trans) ∗ 100;
```

generated, by invoking the function *getCycles*, in order to be able to calculate the overlap between both set of cycles. Then, both sets of cycles are scanned simultaneously to get the total value of co-gap and co-execution. Co-gap occurs when both patterns share a gap during a certain number of transactions, e.g. in Figure 6.7 there is a co-gap between transactions t_8 and t_{10}. Co-execution occurs when both patterns occur simultaneously during a certain number of transactions, e.g. in Figure 6.7 there is a co-execution between transactions t_3 and t_5.

The scan is divided in three phases: first, the gap before the first cycles is considered; (a) second, all the co-gaps and co-executions between each pair of cycles (b)(c)(d)(e); and third, the gap from the last cycles to the end of the execution is considered (f). All possible cases between two cycles are graphically shown in Figure 6.8 for clarity.

The function *getCycles* receives a core periodic pattern and generates its list of cycles, whose definition can be found in Definition 3.1 in Section 3.1. Cycles are associated to a unique period but a core periodic pattern contains a list of periods. In reality, the minimum period value is the one we are interested in since the rest of the periods are related to it either

Algorithm 13: Generator of set of cycles

```
1  function getCycles(P);
   Data: A core periodic pattern P(itemset, periods, transactions)
   Result: The set of cycles from the core periodic pattern
2  cycles ⇐ ∅;
3  period = periods[0];                                   /* periods is ordered */
4  checked[1..transactions.length] = FALSE;
5  foreach i ∈ [1..transactions.length] do
6      cycle ⇐ (start = 0, end = 0, length = 0);
7      j = i + 1;
8      while ((j ≤ transactions.length) AND
       (transactions[j] − transactions[i] ≤ period)) do
9          if ((checked[i] == FALSE) AND (checked[j] == FALSE) AND
           (transactions[j] − transactions[i] == period)) then
10             cycle.start = i;
11             cycle.length = 1;
12             checked[i] = TRUE;
13             checked[j] = TRUE;
14             k = j + 1;
15             while ((k ≤ transactions.length) AND
               (transactions[k] − transactions[j] ≤ period)) do
16                 if ((checked[k] == FALSE) AND
                   (transactions[k] − transactions[j] == period)) then
17                     cycle.end = k;
18                     cycle.length + +;
19                     checked[k] = TRUE;
20                 k + +;
21         j + +;
22     if cycle.length ≥ 2 then
23         cycles := cycles ∪ cycle;
24         cycle ⇐ ∅;
25 return cycles;
26 end function;
```

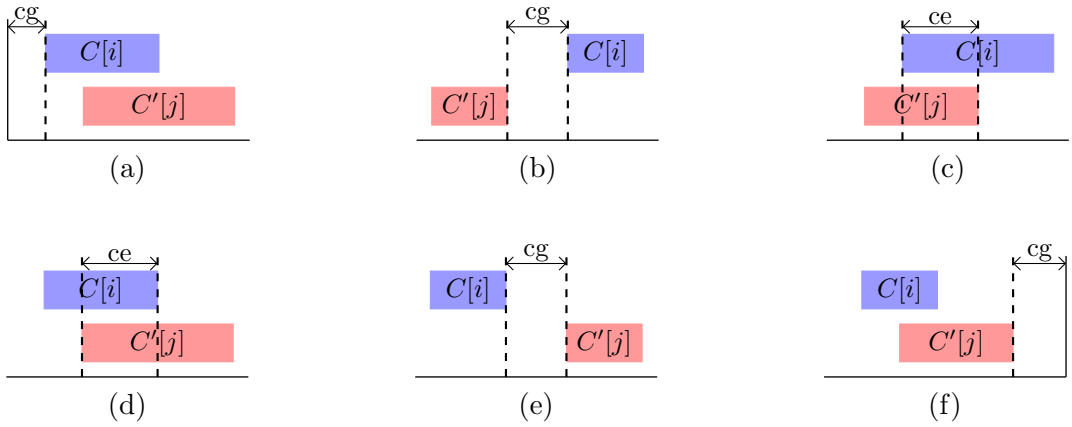


Figure 6.8: Graphical representation of all cases between two cycles in *competition_ratio*. **Note:** cg = co-gap, ce = co-execution.

by multiplication or by addition. In order to generate the list of cycles, the transaction list is scanned and the transactions distanced by the period value are considered on the same cycle. For each cycle, only the start and the end are stored.

Let's illustrate the explanation with an example. Consider two patterns C_1 = ($\{a, b\}$, $\{2\}$,$\{t_1, t_3, t_5, t_6, t_8, t_{11}, t_{13}, t_{15}\}$) and C_2 = ($\{c, d\}$, $\{2\}$, $\{t_3, t_5, t_7, t_{10}, t_{12}\}$), shown in Figure 6.7. Over 15 transactions, the tool calculates the co-execution and the co-gap of both patterns. In this case, co-execution has a value of 4 while co-gap has a value of 2. Therefore, the competition ratio is equal to 15 - (4 + 2) = 6, which is 40 % of the total execution. If *min_ratio* was set to 50 % these two patterns would not be considered as being in competition.

Usage. Initially, the minimum ratio of competition should be set fairly high in order to limit the set of results to pairs of patterns that are in competition for the most part of the execution of the application. This will allow developers to quickly identify mayor conflicts between different parts of the system.

Unfortunately, this tool is not able to tell the difference between a conflict and a standard resource sharing. This analysis tool compares CPCs but it does not search for any extra meaning or information on the itemset part of the CPCs. Without context information on the itemset, it is impossible to say whether two CPCs, identified by the tool as being in competition, represent a conflict between two parts of the application or a standard resource sharing. Therefore, developers analyzing the output of this tool are the only ones that have enough information to carry out this differentiation.

If all the pair of competitor patterns found were due to standard resource sharing, developers should gradually lower the value of the minimum ratio of competition to discover pair of patterns that are in competition only in part of the execution. This will allow developers to identify localized conflicts between different components of the system.

6.4 Conclusions

In this chapter, we have defined the first guidelines towards a methodology that explains how our approach should be used in order to analyze multimedia application execution traces. This methodology covers the preprocessing of execution traces, the mining process and the postprocessing of the mining results.

In the preprocessing of execution traces, certain aspects need to be taken into account in order to obtain meaningful results in the mining process. Thus, we first gave indications about how much information should be included for each event, and how to choose the method to split the execution trace. Moreover, we proposed two generic splitting methods, based on the use of time intervals and software functions. Although, domain experts could propose other methods more adapted to multimedia application execution traces.

During the mining process, PerMiner algorithm is applied to transactional databases in order to obtain a set of core periodic concepts. The only input of PerMiner algorithm, apart from the transactional database, is the minimum support threshold which indicates the minimum number of transactions in which a CPC should occur in order to be outputted. Therefore, we gave some guidelines about how to choose the value of the minimum support threshold during the analysis of an execution trace.

Finally, we showed that the most efficient way of analyzing the output of PerMiner algorithm is by using analysis tools. Different analyses can be carried out over a set of core periodic concepts in order to obtain interesting information from them. Thus, we studied how a CPC visualization tool, presented in the next chapter, can be used to visually analyze the output of PerMiner algorithm. Moreover, we proposed a competitors finder tool which might help in finding conflicts between different parts of the analyzed software. We also studied how this tool should be used to analyze a set of core periodic concepts.

In summary, we showed that in each of the steps, domain specific knowledge is required in order to make the right decisions towards a successful analysis of the execution traces. For a more detailed methodology, a thorough study needs to be carried out in collaboration with multimedia application developers and systems engineers in order to be able to propose the needed approach for each step of the methodology.

7

CPCViewer: a CPC Visualization Tool

Generally, pattern mining algorithms output the mined patterns in text format. Indeed, PerMiner algorithm outputs the list of mined CPCs on a text file, with a line per CPC. Each CPC is composed by three parts: a set of itemsets, a set of periods and a set of transactions. The elements of each set are separated by commas, each set is delimited by parentheses, and the sets are separated by white spaces. As an example, below we can observe the result of executing PerMiner with the dataset presented in Chapter 4 and shown in Table 4.2.

```
(a,c) (2,5) (1,3,6,8,11,13)
(b,c) (10) (1,3,11,13,21,31)
(a,b,c) (2,10) (1,3,11,13)
(a,c,d) (7) (1,6,8,13,20)
(c,d,e) (11) (20,31,42)
```

In this case, the output only contains five CPCs, but when using a bigger dataset, the set of outputted patterns can be very large. We can already observe in this small example that the periodicity is not easy to analyze. Exclusively with the list of transactions, the user has to expend time building up the cycles and calculating where there is a gap between cycles, which implies that the periodicity has not been respected. Also, the longer the list of core periodic concepts is, the more difficult it is to analyze the relationships between the different itemsets (set of events) contained in them.

"*A picture is worth a thousand words*". Therefore, in order to help developers to easily analyze the list of outputted patterns, we present here a visualization tool, called *CPCViewer*, that graphically visualizes a set of core periodic concepts. CPCViewer receives as input a text file containing a set of CPCs, mined by PerMiner algorithm, and visualizes them as shown in Figure 7.1.

Two important aspects that are going to help developers to identify interesting patterns are the analysis of the relationships between the itemsets, and the analysis of the periodicity of the patterns. Therefore, the proposed visualization tool is divided in two parts: a top part where the list of itemsets forming part of the set of core periodic concepts can be explored; and

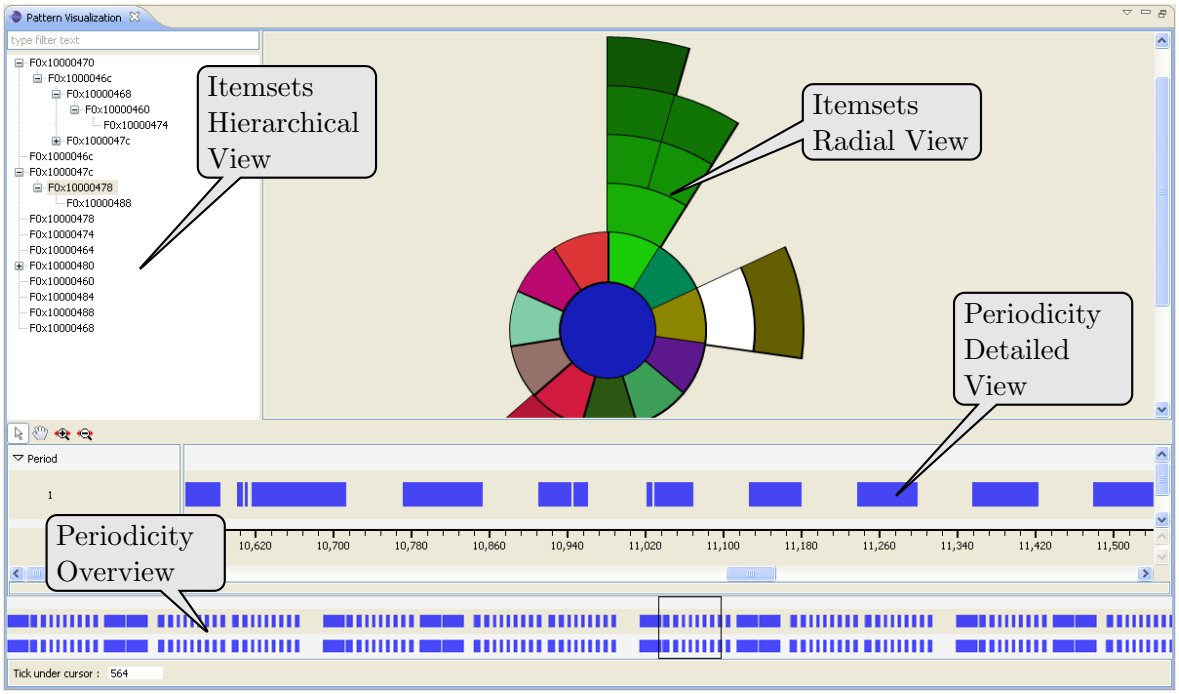


Figure 7.1: Periodic pattern visualization tool

a bottom part that, given an itemset selected on the top part of the visualization, graphically represents the core periodic concepts related with it.

In this chapter, in Section 7.1 we are going to explain in detail how the set of itemsets is presented and can be analyzed by the user. Then, in Section 7.2 we are going to explain in detail how the periodicity of the CPCs is visualized and can be explored by the user.

7.1 Itemset Visualization

The main part of information of a CPC is the itemset. The user can usually decide just by looking at the itemset if the corresponding CPC is interesting or not, decision that depends on the context of the dataset.

In the context of execution trace analysis, the itemsets are sets of events that occur during the execution of a piece of software. By analyzing the sets of events, developers can identify sets of events that were expected to occur together. Then, a quick check of the periodicity can help developers to identify moments in the execution where the expected periodicity was not respected, needing a more detailed analysis. Also, by analyzing the sets of events, developers can identify sets of events that were not supposed to occur simultaneously. In this case, the periodicity shows developers the areas of the execution where a detailed analysis should be carried out in order to understand why those events happen together in the execution.

Therefore, when the user loads a file containing a list of CPCs outputted by PerMiner algorithm, this list is loaded into *CPCViewer* visualization tool. First, the list of itemsets is

extracted from the input list of CPCs. This list of itemsets is presented by the tool in two forms: a hierarchical view on the top left part of the visualization tool, and a radial view of the hierarchy on the top right part of the visualization tool.

Itemsets Hierarchical View

The set of itemsets is organized as a hierarchy where the higher levels of the hierarchy contain items with a high frequency, this is the most frequent items. The idea of using a hierarchy comes from the fact that the enumeration used in PerMiner, being centered on itemsets, forms an enumeration tree, such as in Figure 4.5. Rebuilding this enumeration tree would be too computationally expensive. Thus, a relaxed version of the enumeration tree is build to show the relationships between the different itemsets. The hierarchy is build following the principle: if a pattern X is contained in another pattern Y, then X is the parent of Y in the hierarchy. Thus, a sub-node is a specialization of its node.

Let's take as an example the set of itemsets $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$, $\{a, c, d\}$ and $\{c, d, e\}$, shown at the beginning of this chapter. The corresponding hierarchy of these sets would be:

⊟ a, c
  ├ b
  └ d
⊟ b, c
  └ a
c, d, e

were $\{a, b, c\}$ is a specialization of both itemsets $\{a, c\}$ and $\{b, c\}$, and $\{a, b, d\}$ is a specialization of $\{a, c\}$.

Itemsets Radial View

The same itemsets hierarchy is represented by a radial view on the top right side of the visualization tool. This visualization was first used by Keim et al. in [KSS05]. Here, the radial view gives a high level view of the itemsets hierarchy, giving a quick view of the level of relationship between the different itemsets.

Initially, the root of the radial view is empty and all segments of first level represent the first level nodes of the hierarchy. The size of each segment is proportional to the number of elements on that level, and each segment is colored on a different color to be able to tell the difference between them. All elements in the subtree of a node are colored with the same color as the node, in a darker tone on deeper levels. When the user selects a segment, its color changes to white to be able to tell which segment is selected at any moment of the analysis.

Since the itemsets of core periodic concepts can be very large in terms of number of items, the radial view is scalable by showing only a maximum number of levels. In order to visualize the hidden levels of the hierarchy, the user can double click on a segment. This will make the selected segment the root of the radial view and the hidden levels visible. At any point

the user can double click on the root of the radial view to go back to the initial radial view configuration, i.e. the empty root.

When an element of the hierarchy or a segment of the radial view is selected by the user by clicking on it, the core periodic concepts related with the selected itemset are graphically visualized on the bottom part of *CPCViewer*.

7.2 Periodicity Visualization

The part of *CPCViewer* in charge of visualizing the periodicity of the set of CPCs related to a selected itemset is divided in two parts: a bottom part containing an overview of the occurrence lists of the set of core periodic concepts, and a more detailed view of part of the set of occurrences. Indeed, this part of the visualization follows the principle of "Overview & zoom", proposed by Ben Schneiderman in [Shn86], by which visual data mining tools should provide users first a general vision of the data "Overview", followed by a closer look in order to gain in detail "zoom".

Periodicity Overview

In the context of execution trace analysis, the number of transactions of a dataset depends on the parameter chosen to split the trace and the length of the trace, see Section 6.1 for details. As has been said before, execution traces are voluminous nowadays, but there is a great chance that they will be even bigger with the increase on the number of cores on MPSoC architectures and the tracing of each element of the platforms. In consequence, the number of transactions of a dataset obtained from these execution traces will increase in the same degree. In order to cope with a large number of transactions, an overview of the whole set of transactions with the occurrences of the core periodic concepts highlighted is offered to the user. This way the user can quickly observe where the periodicity has not been respected, and select the corresponding part of the visualization to be analyzed in the detailed part of the visualization.

The horizontal visualization space is divided into vertical lines, each vertical line corresponding to a transaction in the database. Then, for each transaction, if it is inside a cycle of the corresponding CPC, the vertical line representing the transaction is colored in blue, otherwise the vertical line remains uncolored. Indeed, *CPCViewer* shows an abstraction of each CPC by coloring complete cycles and not only the transactions that form part of the cycle. This way, the intra-cycle "gaps", i.e. placed between two transactions of a cycle, are not shown, making the true "gaps" between the cycles stand out.

For example, the visualization of the periodicity of CPC $(\{a, c\}, \{2, 5\}, \{1, 3, 6, 8, 11, 13\})$ in Table 4.3 is shown in Figure 7.2. This CPC has two periods, each period is represented individually. For each period, the set of cycles of that period is generated. Then, each cycle is visualized by coloring blue all transactions between the first and the last transaction in the cycle. For example, for period 2 there are three cycles $(1, 2)$, $(6, 2)$, $(11, 2)$. Let's take the cycle $(6, 2)$ that starts in transaction 6 and has a length of 2 transactions. This means that the space between transaction 6 and transaction 8 $(offset + period \cdot (support - 1))$ is colored in blue. When two cycles of the same period overlap, an extra period line is added

in order to show the overlapping cycles individually. Such is the case of period 5 with two overlapping cycles $(1, 3)$ and $(3, 3)$. Cycle $(1, 3)$ covers transactions t_1 to t_{11}, while cycle $(3, 3)$ covers transactions t_3 to t_{13}. Both cycles overlap in transactions t_3 to t_{11}, thus two period lines are used to represent period 5 where each of the cycles is perfectly visible. Otherwise, using only one line, the existence of these two cycles would not have been easy to see.

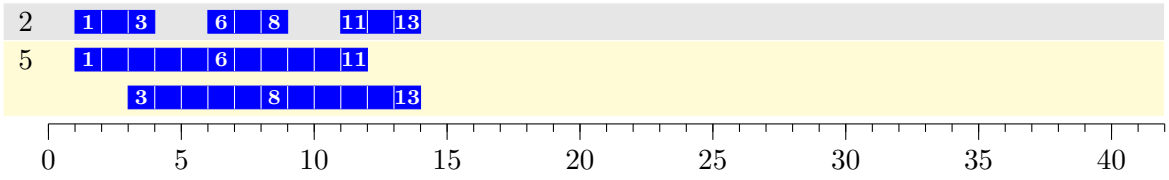


Figure 7.2: Example of periodicity visualization

Periodicity Detailed View

For a detailed analysis of the periodicity, the overview is not detailed enough due to the necessary zoom applied to fit the whole set of transactions on the screen. Therefore, a more detailed view of part of the overview is shown to the user. This detailed view is split in two zones: the zone on the left shows the periods of each of the CPCs being visualized, and the right part shows the detailed view of the occurrences of the CPCs with a time-line indicating the current transaction numbers begin visualized.

The user can navigate this detailed view backwards and forwards on the list of transactions as well as being able to zoom in and out of the detailed area. This way the user can select the level of detail needed for the analysis. At any moment, the part of the overview being visualized on the detailed view is indicated by a rectangle on the overview part.

7.3 Conclusions

In this chapter, we have presented *CPCViewer*, a core periodic concept visualization tool that facilitates the analysis of a set of core periodic concepts outputted by PerMiner algorithm. This visualization tool offers a separated analysis of the set of itemsets, contained in the set of core periodic concepts, from the periodicity of each core periodic concept.

We believe that the itemset is the first attribute of a core periodic concept the user would be interested in, and that the need of carrying out an analysis of the periodicity of the corresponding CPCs is going to be decided just by analyzing the itemset. Therefore, two different views of the same information are offered to the user, consisting in a hierarchy of the set of itemsets shown by making use of a tree and a radial view.

On one hand, the visualization using a tree offers a quick view of the main and most frequent itemsets. From there, the user can navigate the hierarchy into the specializations (subtrees) of each of the main itemsets. On the other hand, the radial view offers a global view of the hierarchy up to a certain number of levels to avoid overwhelming the user. The

user can use these view to have a global idea of the whole hierarchy which helps in the navigation of the tree hierarchy.

Regarding the periodicity of each CPC, the visualization tool offers an abstraction that allows an easy identification of gaps in the periodicity. Moreover, the use of the principle "Overview & Zoom" allows a quick identification of interesting zones by use of the overview part of the visualization, which can then be analyzed in detail in the zoomed part of the visualization.

In the next chapter, we are going to show some use cases where our approach has been applied to the analysis of the execution traces of two multimedia applications.

8

Use Cases: Analysis of multimedia application execution traces

The use cases presented in this chapter aim to show that our approach can help in debugging multimedia application traces. The execution traces used in the experiments were obtained by making use of *KPTrace* [kpt] *STLinux* tracing tool, since applications we focus on run on *STLinux* operating system. *STLinux* is a modified version of Linux kernel, designed for STMicroelectronics products. Indeed, *KPTrace*, a module in *STLinux* is in charge of tracing certain system events of the execution such as context switches and interrupts, but also application events such as function calls. The execution traces are accessible thanks to a NFS connection with the development boards used in the experiments.

Therefore, in Section 8.1, the execution trace of a video and audio decoding test application called *HNDTest* is analyzed by mining the set of core periodic concepts and then using the competitors finder tool presented in Section 6.3. In Section 8.2, the execution trace of an audio decoding pipeline programmed on top of *GStreamer* multimedia framework, briefly presented in Section 2.2, is analyzed. Moreover, *CPCViewer* visualization tool presented in Chapter 7 is used in order to analyze the periodicity of the set of core periodic concepts mined by PerMiner algorithm. Finally, in Section 8.3, we conclude this chapter.

8.1 HNDTest Application

HNDTest application is a multimedia test application based on STAPI [sta09], which is a middleware designed by STMicroelectronics to exploit their set-top boxes. Through the user interface of the application the user can carry out control operations such as the initialization of the platform, multimedia operations such as playback, and so on. Moreover, the application delivers status information through the serial port.

We retrieved a trace from an execution of a video and audio playback of *HNDTest* application run on an *ST40* processor on a development platform containing an *STi7200* SoC, presented in Section 2.1. The execution trace occupied 7.2 MB of memory.

The first analyses by splitting the trace into coarse-grained windows such as the decoding of a frame, did not show any behavior worth analyzing in detail. Therefore, we decided to preprocess the trace by splitting it into finer-grained transactions by using a time interval of 1 ms. This value allows to discover relationships between small sets of items that would have

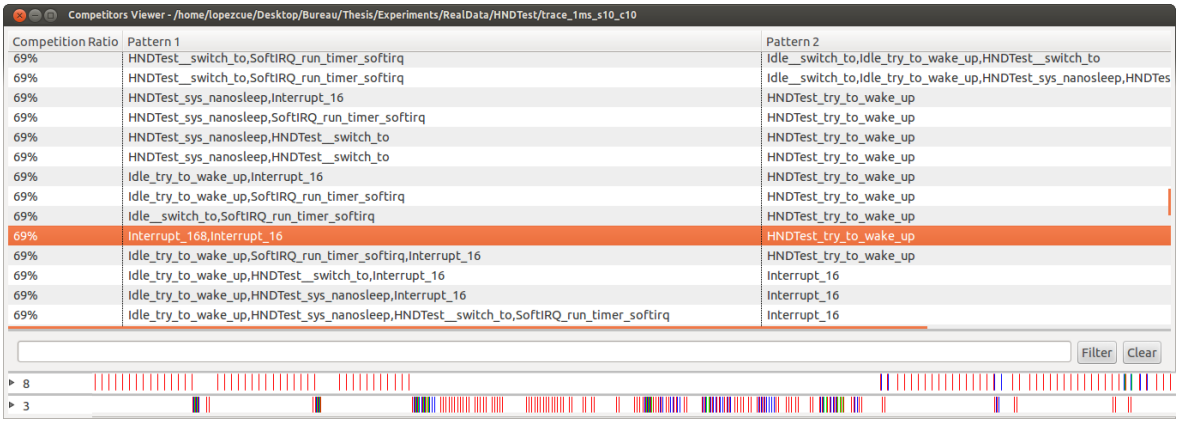


Figure 8.1: Conflict between application and operating system

been difficult to find using a coarser-grained preprocessing technique. After preprocessing, the dataset contained 12843 transactions with an average transaction length of 8 items.

During the mining process with PerMiner algorithm we used a minimum support threshold of 10%. We chose such a low value due to value chosen for the preprocessing of the trace. With such a low value, it was to expect that bigger values of periods would become interesting, and therefore a low level of minimum support threshold would allow the discovery of such periods.

PerMiner algorithm produced 758 core periodic concepts in about 195 seconds. This trace would produce 51,446 triadic concepts and 18,459 frequent periodic patterns. Therefore, we can observe that mining core periodic concepts considerably reduces the number of patterns to analyze.

The bigger number of triadic concepts with respect to the number of frequent periodic patterns can be explained by the fact that the number of periods in frequent periodic patterns is limited to one per pattern while in the triadic concepts is not limited. Consequently, all possible combinations of all lengths of the set of possible periods are generated as triadic concepts. All these extra patterns are then removed by the core periodic concept miner. Also, the transactions of the dataset used in this example have an average of 8 items per transaction which produces patterns with relatively short itemsets that consequently do not produce many combinations in terms of frequent periodic patterns.

Then, as part of the analysis of the set of core periodic concepts mined, we used the tool *Competitors Finder* (see Section 6.3) to identify possible conflicts between different entities of the system. We highlight here a pair of competitor patterns found by the tool involving on one hand, *Interrupt 16*, which is the clock of the processor, and *Interrupt 168*, which is a USB port interrupt, and on the other hand, a system call (`try_to_wake_up`) involving thread *HNDTest*.

Indeed, in Figure 8.1 we can observe that these two patterns are in competition: the top pattern (interrupts) stops executing when the bottom pattern (*HNDTest*) increases its activity. In this context, the processor periodically polls the device connected to the Universal Serial Bus (USB) port to check whether it has data to transfer. Interrupt 168 acts here as a reminder to the processor to check the USB port. The actual process of checking the USB

port would be done by the corresponding software interrupt to avoid masking the interrupts for too long during execution. When *HNDTest* increases its activity, it masks the interrupts and therefore prevents the processor from transferring any incoming data from the USB port, which causes a delay in transmission.

This might have further repercussions if the USB reception buffer becomes full while waiting to transfer the data, causing the data to be overwritten. Moreover, this might affect the correct transfer of a file from the device connected to the USB to the board, or even cause errors in a video decoding operation when the source is located in the device connected to the USB.

8.2 GStreamer Application

GStreamer, presented in Section 2.1, is a pipeline-based multimedia framework that has been adopted by many different corporations such as Nokia, STMicroelectronics and so on. In this experiment, a simple GStreamer pipeline was used to playback an audio file, and the execution was registered into a trace.

The platform used during this experiment was a development platform containing an Orly SoC, presented in Section 2.1, and the application was run on the ARM processor. The output of the application revealed that the mixer was detecting an underrun, i.e. data was being fed at a lower speed than needed to keep up with the timeouts, and the timeout was expiring. Therefore, we decided to split the trace using a value of 32 ms, which is rate at which the mixer maps audio samples, in order to see whether the expected rate was preserved or not. The preprocessing phase produced a transactional database with 1250 transactions with an average transaction length of 35 items.

Then, PerMiner algorithm was run, with a support threshold of 10%. It mined 787 core periodic concepts in about 28 seconds. This trace would have produced 21,588 triadic concepts and 3,086,321 frequent periodic patterns.

In this example, the number of frequent periodic patterns is much bigger than the number of triadic concepts. This is because the transactions in the dataset of this example are very long, 35 items per transaction in average, and therefore generate patterns with a long itemset. For this reason, the number of frequent periodic patterns per itemset and per period is large since there is a frequent periodic pattern for each combination (of any length) of the items in the itemset.

In the core periodic concepts mined, we expected to find some patterns with a period of 1 (32ms) over all the data. This would mean that the functions in charge of the decoding of an audio frame happen every 32 ms, i.e. at the correct periodicity. Surprisingly, the patterns found exhibited *gaps* in the periodicity. An example of the periodicity of the patterns found is shown in the bottom part of Figure 8.2. As explained in Chapter 7, the selected pattern is highlighted in white in the radial view, and the itemset can be seen in the hierarchy tree in the top left part of the figure.

We can easily see the gaps in the periodicity, but we can also see that the gaps appear quite regularly. This means that the application is unable to keep up the expected mixing rate. Thus, by using our approach, the search space for the problem was reduced substantially.

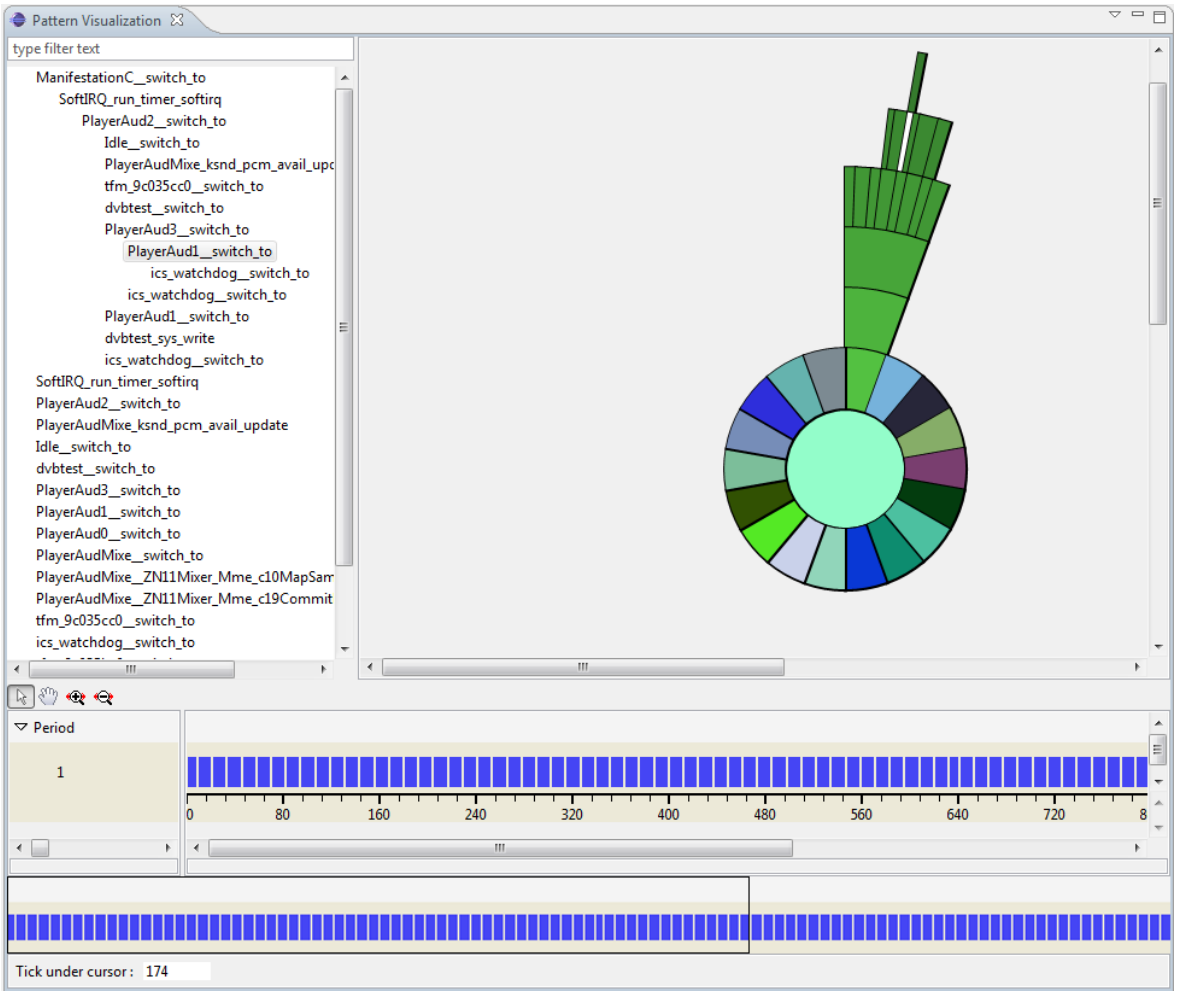


Figure 8.2: Trace visualization

The application developers investigated these *gaps* and found that there was a bug in the calculation of an interrupt period. This interrupt was in charge of flushing a buffer of samples read from memory, but it was being generated too late. Moreover, this caused buffer overflows, which in consequence caused higher level drivers to underflow when operating double buffered.

8.3 Conclusions

In this chapter, we have presented two use cases based on the analysis of real multimedia application traces. In both cases, our approach has been applied following the methodology presented in Chapter 6. Therefore, for each use case we have justified the decisions taken for the preprocessing of the execution traces as well as the minimum support threshold value used during the mining process. The postprocessing of the mining results have shown that our approach allows developers to quickly discover certain problems in the execution of their applications by automatically analyzing their execution traces, that otherwise would have taken a long time to discover by manually analyzing execution traces.

Concretely, in the first use case, *HNDTest* application, we have discovered a conflict between the test application and the system by postprocessing the mining results using the competitors finder tool presented in Section 6.3. Then, in the second use case, *GStreamer* application, we have helped developers in reducing the search space by postprocessing the mining results using *CPCViewer*, presented in Chapter 7. Moreover, a detailed analysis of the "gaps" in the periodicity of the application revealed an error in the programming of an interrupt that was causing the mixer to underrun.

It is important to say that our approach can also help in the verification of the performance of an application by checking that the expected periodicities are being respected. Therefore, our approach allows a finer-grained analysis of the performance of the application than a performance profiler.

Part III

Related Works

9 Related Works

In this chapter, we study previous works carried out regarding the main topics concerned by this thesis. Concretely, in Section 9.1 we present previous works on periodic pattern mining. In Section 9.2, we study previous works on pattern mining on trace analysis. Finally, in Section 9.3, previous works on pattern visualization are presented.

9.1 Periodic Pattern Mining

Ozden et al. defined the problem of discovering cyclic association rules in transactional databases in [ORS98]. An association rule captures relationships between sets of items. For example, on a market basket problem a discovered association rule might be represented by "milk ⇒ cereal (support 10%, confidence 60%)" which means that in 10% of the transactions customers bought both milk and cereal, and that in 60% of transactions where customers bought milk, they also bought cereal. A cyclic association rule includes an extra source of information: the time. For example, a cyclic association rule might state that milk and cereal are bought together mainly in the morning (9am to 10am).

More formally, the authors defined a cycle of an association rule as a tuple (length, origin) where length states a length in terms of time units, e.g. 24 hours with a time unit being 1 hour, and the offset states the first time the cycle occurs, e.g. 9AM, where the association rule holds in every instance of the cycle. For example, the cycle (24,9) of the association rule "'milk ⇒ cereal" would state that the association rule holds during the time interval 9am-10am every day (every 24 hours).

This type of periodicity is called **partial periodicity** since not all the time units contribute to the periodicity, e.g. in the cycle (24,9) presented above only the time unit 9am-10am contributes to the periodicity while the rest of the time units do not contribute. Nevertheless, the authors only considered **perfect periodicity**, i.e. the pattern holds in all cycles, which is very restrictive since no irregularities or gaps in the periodicity are allowed.

Han et al. [HGY98] [HDY99] introduced several algorithms to mine partial and **imperfect periodic patterns** over symbol sequences. The so called imperfectness was limited to allow missing some occurrences in an otherwise perfect periodicity setting. Moreover, this approach, like Ozden et al., considered *synchronous* periodic patterns, where all occurrences must follow

the same alignment. Such approaches are not resistant to disturbances such as "phase shift", which changes the alignment of occurrences without changing the period.

One of the first studies to consider *asynchronous* periodic patterns was carried out by Ma et al. [MH01]. Ma et al. proposed a definition to mine sets of events that are found periodically together on a time-based sequence. In their work, the authors introduced irregular gaps in the periodicity that they call "off-segments". An off-segment is of arbitrary size, and during an off-segment the pattern is either absent or not periodic. They also introduced a windowing technique in order to resist noise on the timing of events.

Yang et al. [YWY03] considered asynchronous periodic patterns when mining sequences in time series data. Similarly to Ma et al., during valid segments perfect repetitions of the frequent sequence are found and segments are separated by disturbances. The authors defined the minimum number of repetitions min_rep of a sequence in order to be considered a valid segment, but also the maximum number of symbols between two segments max_dis in order to differentiate random noise from a change of system behavior.

Yang et al. returned for each frequent sequence found, the longest subsequence of the time series data supporting this sequence. Huang et al. [HC04] argued that returning only the longest subsequence, a part of the system behavior might remain hidden. Therefore, the authors extended the framework of [YWY03] by returning all subsequences whose global number of repetitions is greater than a given global repetition threshold $global\_rep$.

Sequences are useful in multitude of contexts, including execution trace analysis. In our context, we want to help developers to understand and validate their software by automatically analyzing execution traces. Sequences present too many constraints and moreover they are very computationally expensive to mine. Moreover, when dealing with concurrent software, the scheduler of the operating system can change the order of the events in the trace and therefore it is more difficult to mine sequences. Therefore, we are interested in mining periodic itemsets.

In the context of transactional databases, the research on periodic pattern mining has been centered on mining periodic-frequent itemsets, introduced by Tambeer et al. in [TAJL09]. In this approach, only one period per pattern is considered which is equal to the maximum inter-arrival distance of the pattern. For instance, if a pattern occurs in transactions $\{t_1, t_3, t_5, t_{15}, t_{17}, t_{18}\}$, the inter-arrival distances would be $(1, 2, 2, 2, 10, 2, 1, 0)$, and thus the period would be 10. Then, a pattern is considered periodic-frequent if its support is greater than a minimum support threshold min_sup and its period is smaller than a maximum periodicity threshold max_per.

The problem of this approach is that there is a danger that the period of a pattern will not be representative of the real periodicity of the pattern. An inter-arrival time bigger than the rest can have as a consequence that the associated pattern is not considered periodic-frequent because the period is bigger than max_per. Considering the example presented in the previous paragraph, if max_per is smaller than 10, then the pattern would not be considered periodic-frequent while it is clear that its periodicity is 2. Kiran et al. tried to solve this issue in [KR11] by adding an extra constraint called minimum periodic ratio ($MinPr$). This new constraint considers only the inter-arrival distances that are smaller than max_per and calculates their periodicity ratio. If the calculated periodicity ratio is bigger than $MinPr$, then the pattern is considered periodic-frequent.

We believe that periodic-frequent patterns do not reflect well the periodicity of the patterns. In our context, a more representative information about the periodicity of each pattern is needed. Moreover, it would be difficult to specify a maximum period without generating a combinatorial explosion of the results since no efforts are put in eliminating redundancy.

Therefore, in this thesis we have exploited a definition of frequent periodic pattern (see Definition 3.3) very similar to the one presented in [MH01], which is well suited to the variety of frequent periodic patterns that can be found in execution traces. The minor difference is that in our case the windowing is performed as a preprocessing step, so the definition is directly based on windows of events and not on raw events.

Coping with Combinatorial Explosion

As other exhaustive frequent pattern mining problems, periodic pattern mining suffers from combinatorial explosion that leads to a huge number of results with high redundancy among these results.

Sequences present this problem in a higher scale, authors try to limit the number of subsequences returned for a frequent sequence, either by returning only the longest one [YWY03] or by requiring a minimal number of repetitions [HC04].

Regarding the periods, which can also suffer from this problem, previous studies have tried to reduce the set of possible periods. This was done, in the brute-force case by using a unique period [TAJL09] [KR10], in the simple case by putting lower and upper bound limits to the period value [ORS98] [YWY03], or, in a more complex case, by calculating a set of candidate periods using the *Fast Fourier Transform* [BVA$^+$02], or the chi-squared test [MH01].

The first study to consider the redundancy of periods while mining the whole set of possible periods was carried out by Yang et al. [YL04]. The authors mined synchronous partial periodic patterns from time series databases. In their algorithm a period p is pruned if is contained by another period p', i.e. the occurrences of p are included in the occurrences of p'. Indeed, in this aspect, our approach is similar to this approach. However, the authors do not consider redundancy regarding the itemsets which we consider that would reduce considerably the combinatorial explosion of periodic pattern mining algorithms.

Regarding the itemsets, Kiran et al. proposed a solution to the "rare item problem" in [KR10], i.e. items with low frequency being generated without the combinatorial explosion of a very low minimum support threshold, by specifying a minimum support threshold per item. This way, a pattern is frequent only if it is present in as many transactions as the maximum value of minimum support threshold of its items.

This is an interesting approach that does not consider closed or maximum frequent patterns, i.e. the list of frequent patterns is fully generated, but manages to reduce the number of patterns generated while keeping the ones the user is interested in. The drawback of this approach is the fact that it is necessary to specify a minimum support threshold value for each item. This is manageable when the number of different items is small but it is not conceivable with large number of items different items. On the other hand, closed frequent patterns offer a condensed representation of the set of frequent patterns so that a low minimum support threshold can be used without the consequential combinatorial explosion.

Amphawan et al. [ALS09] proposed to mine the top-k periodic-frequent patterns on a transactional database, i.e. the set of k patterns with highest support. Obviously, this reduces substantially the result set of patterns, but in our case this is not a viable solution since what we are looking for are irregularities and this approach would give us a subset of the most regular patterns.

As far as we know, there are no previous studies that consider the frequent itemset redundancy problem in the context of periodic pattern mining. In this thesis, the redundancy of the periods is also taken into account in a similar way than in [YL04], although our technique for computing of the periods and the successive pruning strategies are more complex since we consider asynchronous partial periodic patterns.

Furthermore, to the best of our knowledge our study is the first one to present a condensed representation [PBTL99] [CG02] of the set of frequent periodic patterns. Condensed representations such as closed [PBTL99] or non-derivable [CG02], generally exploit closure techniques based on Galois connections. But, since periodic pattern mining are based on ternary relations, existent techniques could not be exploit. Therefore, we propose in this thesis a condensed representation based on a triadic approach. Furthermore, the nature of periodic patterns makes possible for the proposed condensed representation to go further than the standard triadic approach that proposes triadic concepts [CBRB09].

9.2 Execution Trace Analysis through Pattern Mining

Pattern mining is starting to play an important role in system analysis, even more in embedded systems, where tracing is widely used in order to analyze the system while avoiding high intrusiveness.

First uses of pattern mining in system analysis focused on detecting bugs, e.g. introduced by copying-and-pasting kernel source code [LLMZ06] or caused by the violation of programing rules [LZ05], or more generally detecting systemic problems [LXM08]. Lo et al. [LCH$^+$09] studied how to classify software behaviors, obtained by pattern mining of known normal and failing execution traces, in order to detect failures in future executions of the system.

Regarding program validation, Lo et al. [LcKL07] proposed to mine association rules from program execution traces in order to help developers in better understanding program behaviors and facilitate the verification of their programs. Also, Chang et al. [CW10] worked on system verification using pattern mining in order to extract assertions from simulated traces of the system being validated. In terms of performance analysis, Zou et al. [ZXHW10] used pattern mining to reduce vast amounts of hardware sample data into a set of easier-to-analyze frequent instruction sequences, in order to help to analyze the performance of the system.

Most related work are focused on itemsets, association rules or sequences, but there exist other types of more complex patterns such as graphs that have also been explored. As an example, Liu et al. [LYY$^+$05] and Di Fatta et al. [DFLS06] used graph mining techniques to mine call graphs from program execution traces.

Nevertheless, to the best of our knowledge, none of the previous studies have applied periodic pattern mining to trace analysis. We believe that more complex patterns than itemsets or sequences could give more specific information about program behavior and help

in the discover of bugs.

9.3 Pattern Visualization

Even if there exists a vast variety of pattern mining algorithms, all of them are focus on performance and do not pay much attention to usability. Hence, the output of pattern mining algorithms is usually a huge text file with a list of mined patterns. Taking into account that the end user of these algorithms is not necessarily familiar with data mining techniques, the analysis of the mined patterns using a visualization tool would considerably help to facilitate the analysis of the results.

Previous studies have been centered on itemset visualization and to the best of our knowledge, there does not exist any periodic pattern visualization tool. Therefore, in this section we are going to present frequent itemsets visualization tool since an important part of a periodic pattern is the itemset.

Parallel Coordinates

Li Yang proposed in [Yan03] [Yan05] an itemset and association rule visualization tool. The proposed tool transforms the items and the relationships (in form of itemset or association rule) into the visual elements of parallel coordinates. The set of items are listed on the vertical axis, and this is repeated until there are enough vertical axes to host the longest frequent itemset or association rule. Then, the pattern is visualized by a polyline (a series of connected line segments) between consecutive vertical axes joining all items of the pattern. An association rule is visualized as two polylines showing its left-hand-side (LHS) and right-hand-side (RHS) joined by an arrow. Attributes such as support or confidence of the pattern can be visualized using graphical features such as the color or the width of the polyline.

An example is shown in Figures 9.1 and 9.2, in Figure 9.1 frequent itemsets *adbe*, *cdb* and fg are shown and in Figure 9.2 association rule $ab \Rightarrow cd$ is shown. When two or more itemsets or association rules have items in common, e.g. *adbe* and *cdb* in Figure 9.1, polynomial curves can be used instead of polylines.

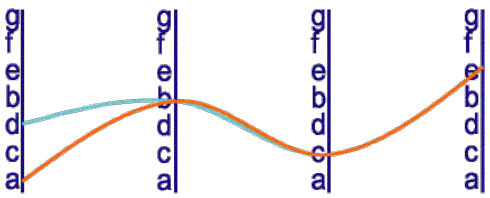


Figure 9.1: Frequent Itemset Visualization

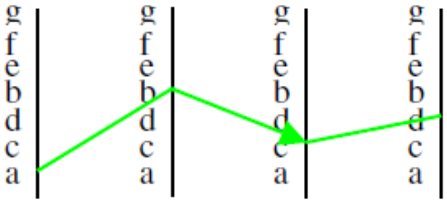


Figure 9.2: Association Rule Visualization

The set of items can be quite big so in the cases where an item taxonomy is available the authors propose to show the tree taxonomy on the vertical axis instead of the plain list of items. An example is shown in Figure 9.3.

In our opinion, this visualization is not scalable since when many patterns need to be visualized, there are too many lines crossing to be able to follow any of them. Moreover,

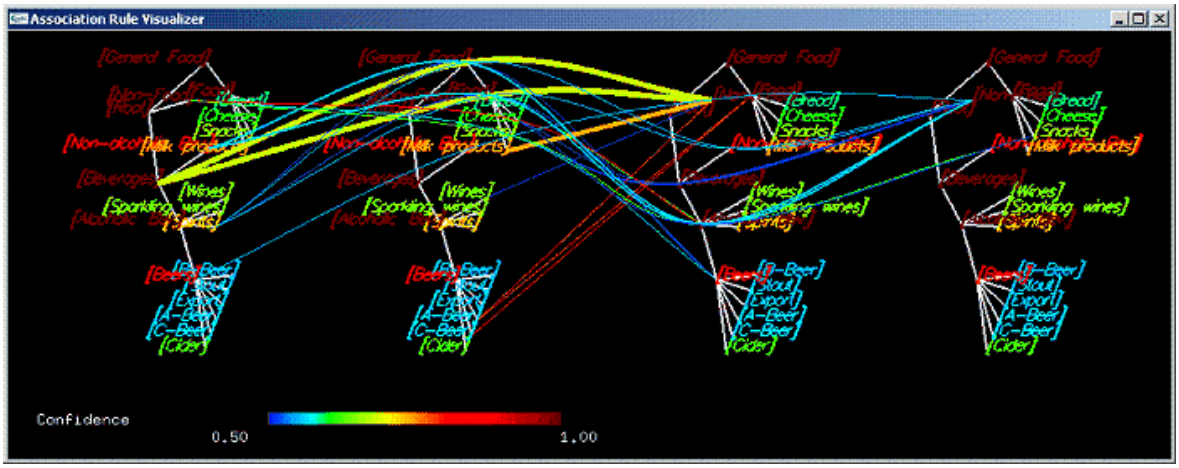


Figure 9.3: Visualizing association rules with item taxonomy

the aggregation of an extra level of visualization by adding the taxonomy to the pattern visualization complicates the analysis instead of facilitating it.

Power Set Viewer

Munzner et al. [MKN+05] proposed a frequent itemset visualization tool, called PowerSetViewer (PSV). The first element of PSV is a visualization module, shown on Figure 9.4, that shows on a rectangular layout the powerset of the dataset's alphabet, i.e. the set of items. The power set of any set I is the set of all its subsets. The layout is divided on sections where the top section corresponds to itemsets of length 1, followed by the section with itemsets of length 2 and so on, each section having a different background color for an easier visualization.

The second element of PSV is a mining engine that receives constraints specified by the user on the visualization module and the raw dataset, and generates the set of frequent itemsets and then sends it to the visualization module. Each itemset has a specific position on the layout and its box is colored blue if the itemset is frequent. When there is not enough room on an area to draw one box per itemset, multiple itemsets are represented by a single box. The more itemsets there are in a box, the darker the box is.

The constraints that the user can specify on the visualization module are: the frequency, aggregation or containment constraints. Aggregation constraints consists on applying an aggregation on an item's attribute. Containment constraints consists on filtering the set of frequent itemsets according to a given itemset.

This approach is useful to have a general idea of the concentration of the patterns towards a zone of the powerset, but is not practical for the analysis of individual itemsets since the itemset itself or its support are not visualized.

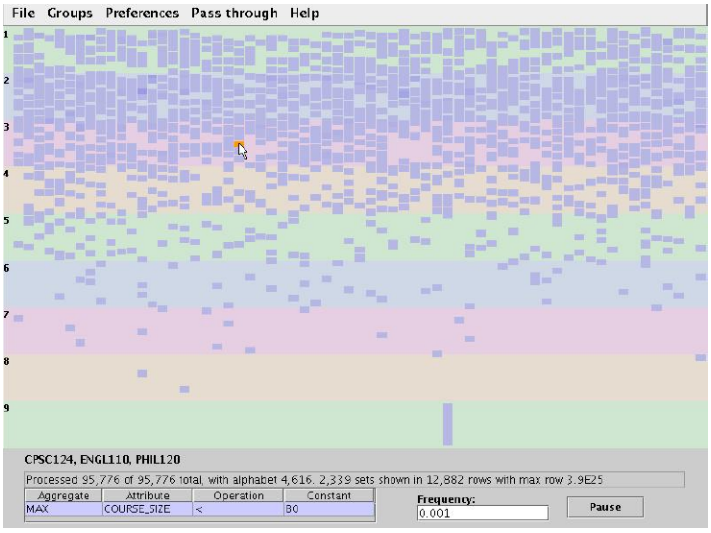


Figure 9.4: PowerSetViewer

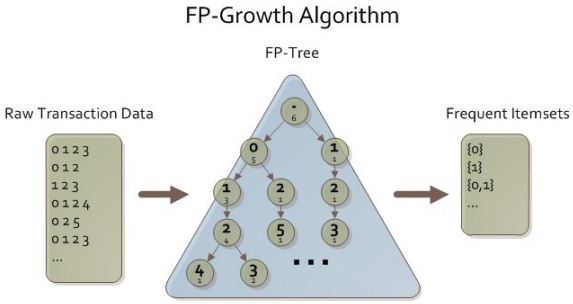


Figure 9.5: FP-growth

FP-Viz

Keim *et al.* [KSS05] proposed a frequent itemset visualization tool, called FP-Viz, based on a radial visual layout that is an extended version of the tree used in FP-growth mining method [HPY00]. FP-growth creates a tree where each node contains an item, and a path of the tree represents an itemset, as can be seen on Figure 9.5. The frequency of each item is represented by the tree level, the lower on the tree the lower the frequency of the item. But instead of representing directly the tree generated by the mining algorithm, FP-Viz represents the tree using a radial visualization method as can be seen in Figure 9.6.

The root of the radial visualization is a circle in the center that initially does not contain any item. Nodes of the tree are represented by circle segments which order of placement depends on the frequency of the corresponding items. Therefore, each pattern contained in the tree is represented by a sequence of circular segments from the root to the end level given by the length of the pattern.

Regarding the support, an color scale is shown on the bottom left corner of the visualization and then each circular segment is colored with the color corresponding to the support

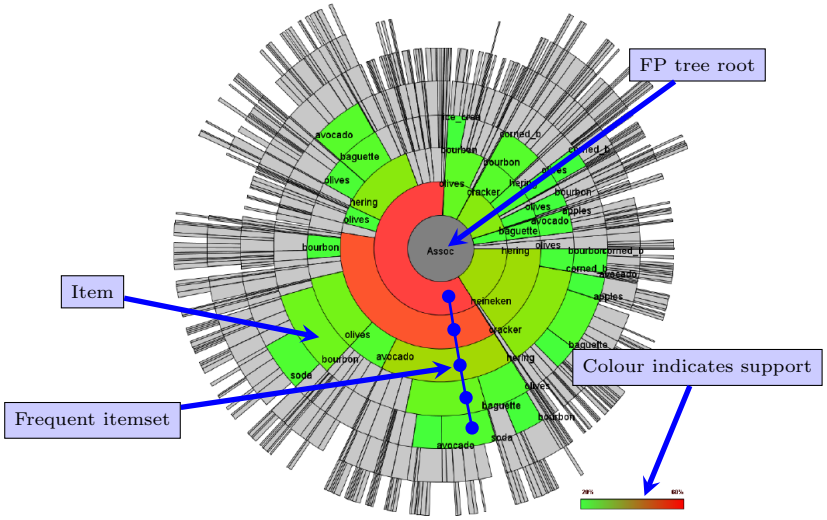


Figure 9.6: FP-Viz

of the represented item.

On this tool, the user can modify the minimum support threshold and see the effect of the modification interactively which is very useful when the user is not sure about the value. Also, the user can select an item and analyze only frequent itemsets containing that item.

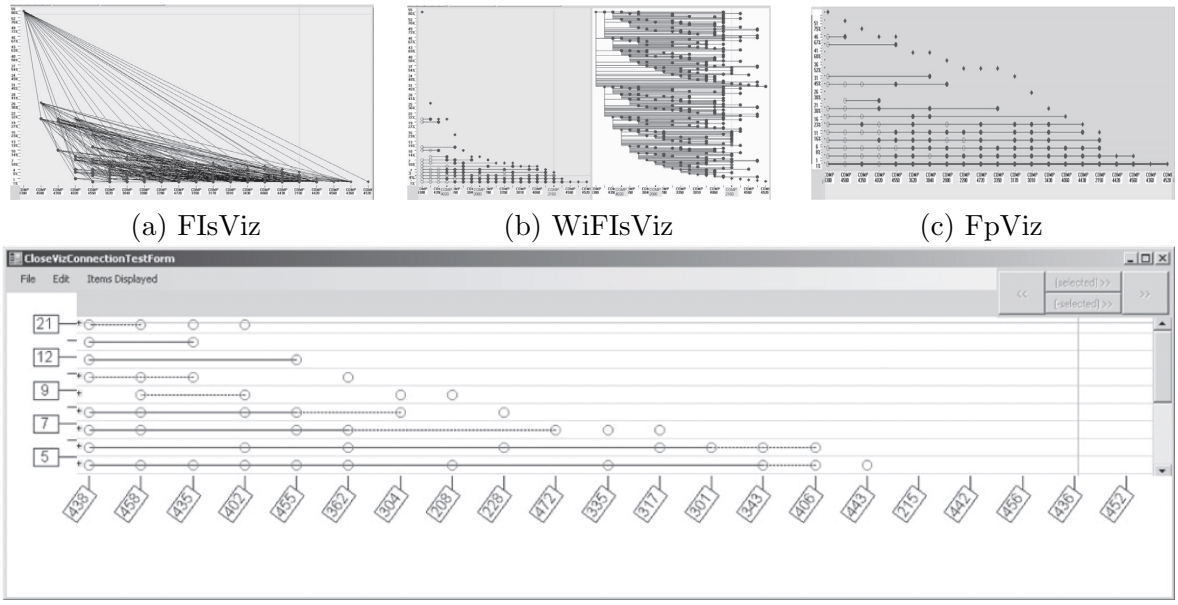
Each segment of the circle is labeled with the corresponding item which is clear when the segment is big but becomes difficult to read when the space between the segments does not allow a correct visualization of the label. Also, the concrete support of each pattern is known approximatively thanks to the color of the segment but it is not possible to know its exact value.

This approach is interesting because it gives a global idea of the whole set of frequent patterns. Nevertheless, it shows all levels of the hierarchy which can be a bit overwhelming for the user. Moreover, the labels are not readable in most of the cases as can be seen in Figure 9.6. Therefore, in our opinion an extra support to visualize the labels of the items would be of much help.

Indeed, we have reused these technique, with a few modifications, as part of our periodic pattern visualization tool *CPCViewer*. Concretely, we have limited the number of hierarchy levels shown simultaneously in order to facilitate the analysis. Also, the itemsets are shown in a tree hierarchy side by side with the radial view in order not to overload the user with information.

CloseViz

CloseViz [CL10] is a frequent pattern visualization tool. This tool propose some improvements to other visualization tools, FIsViz [LIC08a], WiFIsViz [LIC08b] and FpViz [LC09], previously published by one of the authors, Carson Kai-Sang Leung from the University of Manitoba. All four visualization tools are shown in Figure 9.7.



(a) FIsViz (b) WiFIsViz (c) FpViz

(c) CloseViz

Figure 9.7: CloseViz evolution

FIsViz represents the set of frequent patterns on a two dimensional space. The Y-axis contains the support of the patterns and the X-axis contain the different items of the patterns. Each pattern is represented by a polyline connecting n nodes for a n-itemset, i.e. an itemset of length n. The main drawback of this visualization tool is that the polylines can cross each other which may lead to confusion.

WiFisViz and FpViz represent the itemsets using a horizontal line instead of a polyline, which improves readability of the set of patterns. In order to reduce the list of patterns visualized they make use of some compression techniques such as representing two patterns on the same line if one is the prefix of the other one, e.g. $\{a, b, c\}$ is a prefix of $\{a, b, c, d\}$ so they are represented on the same line. The use of filled and unfilled circles allows the user to know when there has been a compression.

CloseViz shows only *closed* patterns which reduces the set of patterns shown and it simplifies the representation by using only unfilled circles. Some extra compression techniques are used to further reduce the set of patterns shown by the tool such as collapsing two patterns with the same support and common prefixes, e.g. $\{c, d, e\}$ and $\{c, d, f\}$ share the prefix $\{c, d\}$ so if they have the same support they will be represented on the same line.

This is an interesting approach that summarizes considerably the amount of information contained in a set of frequent itemsets. However, it groups the patterns according to their support. In our opinion, grouping them by similarity would better show the hidden relationships between the different itemsets of the set.

10

Conclusion and Future Work

Nowadays, multimedia embedded systems populate the market of consumer electronics with products such as set-top boxes, tablets, smartphones and MP4 players. This highly competitive industry pressures semiconductor manufacturers to design *better* products, always providing new features, and before the competitors (short time-to-market).

An observable consequence is the increase in complexity in both the software and the underlying platform hardware, which in turn lengthens the debugging and validation phases of product development. In this context, execution trace analysis presents itself as a comprehensive debugging technique for embedded systems. Soon, the evolution in embedded system tracing techniques will offer execution traces with a so much information that its manual analysis will become unmanageable.

Therefore, for an efficient analysis of execution traces, we believe that automatic analysis techniques such as data mining are the right solution. Moreover, multimedia applications present a periodic behavior based on frame treatment. Therefore, our contributions were focused on the usage of periodic pattern mining techniques for the analysis of multimedia applications execution traces.

10.1 Contributions

The first part of this thesis was dedicated to our pattern mining contributions. First, in Chapter 3, a definition of frequent periodic pattern, adapted from the definition proposed by Ma et al. [MH01], was presented. We then proposed a condensed representation of the set of frequent periodic patterns, called *core periodic concept* by adopting a triadic approach. The proposed condensed representation was focused not only on the redundancy of itemsets but also on the redundancy of periods. To the best of out knowledge, no previous work exists that considers both redundancies at the same time. Moreover, we studied the nature of the proposed condensed representation and identified connectivity properties, based on the fact that the set of transactions of a core periodic concept is deeply related to the set of periods of the same core periodic concept.

These properties allowed us to implement an efficient algorithm for mining core periodic concepts, called PerMiner, presented in Chapter 4. PerMiner algorithm is based on the

state of the art polynomial-delay and polynomial-space enumeration techniques [UAUA04]. Indeed, a version of LCM's enumeration strategy [UAUA04] was adapted by PerMiner algorithm, taking into account the fact that several core periodic concepts can be generated in each step of the enumeration.

In Chapter 4, we first presented a brief explanation of LCM's enumeration strategy [UAUA04] before explaining what modifications were carried out in order to adapt it to the mining of core periodic concepts. Then, we proved that PerMiner algorithm presents polynomial space and polynomial delay time complexity, which make PerMiner algorithm scalable in terms of mining time and memory usage.

Nowadays, multicore processors are a standard component of general purpose computers. Therefore, in order to exploit the parallelism offered by these processors, a parallel version of PerMiner algorithm was presented in Section 4.3. Finally, we proved that PerMiner algorithm returns the complete set of core periodic concepts (completeness), that all patterns returned by the algorithm are core periodic concepts (soundness), and that no duplicates core periodic concepts are generated.

In order to evaluate the scalability of PerMiner algorithm, we carried out a comparative analysis with synthetically generated data. The scalability of PerMiner algorithm was evaluated against three parameters: the number of distinctive items, the number of transactions and the minimum support threshold. PerMiner algorithm's efficiency was compared to a naive algorithm presented in [LCBT$^+$12].

The results showed that PerMiner algorithm is constantly faster than *3-STEP* by two to three orders of magnitude. Moreover, an analysis of the performance of PerMiner algorithm was carried out over a real dataset obtained from an execution trace of a video and audio decoding application, where we showed that PerMiner algorithm is polynomial in time with respect to the number of core periodic concepts and that PerMiner algorithm can efficiently handle real datasets. Besides, we showed that PerMiner presents excellent parallel scaling capabilities.

The second part of this thesis was dedicated to the embedded system contributions. Since most software developers are not familiar with data mining theory and techniques, they need some guidance in order to use any proposed data mining technique during software development. Therefore, in Chapter 6 we proposed a first step towards a methodology to use periodic pattern mining to analyze multimedia application traces.

The presented methodology gives guidelines about the three phases of the analysis: the preprocessing of the execution traces, the mining process and the postprocessing of the mining results. Moreover, several propositions were made on the preprocessing of the execution traces and the postprocessing of the mining results that involved trace splitting methods, a visualization tool and a competitors finder tool. We concluded that domain specific knowledge is an important part of any pattern mining analysis, not only in the preprocessing and postprocessing of the data but also in the parameters chosen during the mining process.

Pattern mining algorithms output the mined patterns in text format. In the case of periodic patterns, this format complicates the search for relationships between the different patterns and the analysis of the periodicity of the patterns. Therefore, in Chapter 7, we proposed a core periodic concepts visualization tool, called *CPCViewer*. We believe that the analysis of the set of core periodic concepts starts with the analysis of the itemsets forming

part of the core periodic concepts and that, once an itemset has been found interesting, then the analysis of its periodicity is carried out. Therefore, the presented visualization tool splits the information in itemsets and periodicity. *CPCViewer* allows a quick analysis of the set of core periodic concepts that would have been a much longer process by using directly the text format.

Finally, the methodology presented in Chapter 6 was used in the analysis of two multimedia applications' execution traces, in Chapter 8. In the first use case, the competitors finder tool presented in Chapter 6 was used to discover a conflict between the multimedia application and the communication port USB. Then, in the second use case, the visualization tool *CPCViewer* was used to discover an anomaly in the periodicity of the decoding of audio frames, which a further analysis showed that it was caused by a buffer overflow. Therefore, through these two use cases, we showed our approach can help in the debugging process of multimedia applications.

10.2 Future Work

The analysis of execution traces is a interesting research domain that is becoming critical with the rapid increase in computational power and parallelism. Below, we explain several research possibilities identified during this thesis, which are divided into three categories: pattern mining, analysis and visualization of execution traces.

Pattern Mining

- *CPC enumeration strategy.* As we have shown in Chapter 4, the enumeration strategy is exclusively focused on itemset enumeration. In consequence, the periods need to be computed in each node of the enumeration tree. We believe that a more comprehensive enumeration strategy focused on the enumeration of both items and periods, with the definition of an efficient first parent test also based on itemsets and periods, would allow the implementation of an even more efficient CPC mining algorithm.

- *Explore different types of patterns: sequences, graphs, etc.* Different types of patterns can be used to discover different kinds of behavioral information. In this thesis, we have focused on the discovery of periodic behaviors of a set of events. However, the only information given by these sets is that the events forming part of them are executed in the same time interval. But it is well known that there are certain software operations that need to be carried out in order, and when this order is not respected, the system might be affected. Therefore, sequences might help in discovering these ordered behaviors.

- *Include context information in order to automatically classify the results of the pattern mining techniques.* As it has been shown in this thesis, results obtained by pattern mining techniques usually contain behavioral patterns well known by the developers as well as patterns that might represent anomalies in the system. During the debugging process, developers are only interested in anomalies. Therefore, including context information into the pattern mining techniques, expected patterns can be early pruned, and only patterns presenting some kind of anomaly would be presented to the developer.

Such approaches are currently being investigated by LIG laboratory and STMicroelectronics as part of the collaborative SOC-TRACE project [Soc].

Analysis

- *Automatic detection of anomalies.* In the same way context information helps classifying pattern mining results, it would be very useful for developers to be able to automatically detect anomalies in the software. As an example, association rules could be generated and used against execution traces to detect anomalies, i.e. where the association rules are not respected. These association rules could be generated by pattern mining algorithms and/or by the analysis of context information given by application developers [FDBM06].

- *Definition of a full methodology.* As has been said, developers are generally not familiar with data mining theory and techniques. Therefore, in this thesis, a first step towards a methodology that makes use of periodic pattern mining to analyze multimedia applications execution traces has been defined. Nevertheless, a full methodology should be defined in collaboration with multimedia applications and software developers in order to make our approach usable.

Visualization

Data visualization is a research domain on its own, and it can be a very powerful tool to analyze data in an intuitive way. Regarding trace visualization, we consider that it is necessary to organize the information given by the trace in ways where a visualization can be useful. Standard trace visualization, i.e. timeline chart, is not scalable to the big quantities of data contained in current execution traces. With the introduction of multicore architectures this lack of scalability is becoming more and more obvious. Therefore, new techniques are needed to support the analysis of execution traces. Below, some ideas are presented:

- Usually, execution traces include events of *different levels* (Hardware, operating system, application) and different parts of the system (communication, synchronization, memory access). By applying a series of filters it would be possible to have a different visualization depending on the level being analyzed.

- *3D visualizations* have not really won their place into visualization for analysis. Nevertheless, the potential is high, and we consider this option should be considered. Anyhow, the visualization should be kept relatively simple since a complex visualization would disturb more than help in the analysis.

- Nowadays object oriented languages are wining their space into soft real-time software. Therefore, in our opinion, the developer would appreciate having a view of the objects in the system, including activation and destruction, possibly with different colors indicating their status, e.g. red indicating an error has happened. Moreover, in the near future the use of components in software development might become more important, and therefore, due to the similarity between the two concepts, object visualization could be adapted to component visualization.

Indeed, a new CIFRE Thesis (Oleg Iegorov), in collaboration with STMicroelectronics, is going to extend some of the contributions of this thesis, specially the definition of a full methodology thanks to contacts with software developers at STMicroelectronics.

Acronyms

3D Three-Dimensional.

CEA Commissariat à l'énergie atomique et aux énergies alternatives.

CPC Core Periodic Concept.

DAG directed acyclic graph.

DSP Digital Signal Processor.

HD high-definition.

HDMI High-Definition Multimedia Interface.

HEVC High Efficiency Video Coding.

HPC High Performance Computing.

ICE In Circuit Emulator.

IDTEC Integrated Development Tools Expertise Center.

IP Internet Protocol.

IP Intellectual Property.

LHS left-hand-side.

MPSoC Multiprocessor System on Chip.

NFS Network File System.

NoC Network on Chip.

PID Process Identifier.

QoS Quality of Service.

RHS right-hand-side.

SoC System on Chip.

USB Universal Serial Bus.

Bibliography

[ALS09] Komate Amphawan, Philippe Lenca, and Athasit Surarerks. Mining top-k periodic-frequent pattern from transactional databases without support threshold. In Borworn Papasratorn, Wichian Chutimaskul, Kriengkrai Porkaew, and Vajirasak Vanijja, editors, *Advances in Information Technology*, volume 55 of *Communications in Computer and Information Science*, pages 18–29. Springer Berlin Heidelberg, 2009.

[ARMa] Cortex-A9 processor. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`.

[ARMb] Mali-400mp. `http://www.arm.com/products/multimedia/mali-graphics-hardware/mali-400-mp.php`.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[AU09] Hiroki Arimura and Takeaki Uno. Polynomial-delay and polynomial-space algorithms for mining closed sequences, graphs, and pictures in accessible set systems. In *SIAM International Conference on Data Mining*, pages 1087–1098, 2009.

[BDRL05] E. Bouix, M. Dalmau, P. Roose, and F. Luthon. A multimedia oriented component model. In *19th International Conference on Advanced Information Networking and Applications*, volume 1 of *AINA'05*, pages 3–8, March 2005.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference (ISSCC 2008)*, pages 88–598, February 2008.

[BHPW07] Mario Boley, Tamás Horváth, Axel Poigné, and Stefan Wrobel. Efficient closed pattern mining in strongly accessible set systems (extended abstract). In *Proceedings of the 11th European conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD'07, pages 382–389, Berlin, Heidelberg, 2007. Springer-Verlag.

[BMR10] E. Bezati, M. Mattavelli, and M. Raulet. RVC-CAL dataflow implementations of MPEG AVC/H.264 CABAC decoding. In *Conference on Design and Architectures for Signal and Image Processing*, DASIP'10, pages 207–213, October 2010.

[Bou05] Jean-Francois Boulicaut. Condensed representations for data mining, 2005. Encyclopedia of Data Warehousing and Mining, J. Wang Editor, Idea Group Reference, pp. 207-211.

[BVA+02] Christos Berberidis, Ioannis P. Vlahavas, Walid G. Aref, Mikhail J. Atallah, and Ahmed K. Elmagarmid. On the discovery of weak periodicities in large time series. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '02, pages 51–61, London, UK, UK, 2002. Springer-Verlag.

[CBRB09] Loïc Cerf, Jérémy Besson, Céline Robardet, and Jean-François Boulicaut. Closed patterns meet n-ary relations. *ACM Transactions on Knowledge Discovery Data*, 3(1):3:1–3:36, March 2009.

[CCH+99] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew J. McNelly, and Lee Todd. *Surviving the SOC revolution: a guide to platform-based design.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[CG02] Toon Calders and Bart Goethals. Mining all non-derivable frequent itemsets. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '02, pages 74–85, London, UK, UK, 2002. Springer-Verlag.

[CGMM+11] M. Castro, K. Georgiev, V. Marangozova-Martin, J.-F. Mehaut, L.G. Fernandes, and M. Santana. Analysis and tracing of applications based on software transactional memory on multicore architectures. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 199 –206, feb. 2011.

[CL10] Christopher L. Carmichael and Carson Kai-Sang Leung. Closeviz: visualizing useful patterns. In *Proceedings of the ACM SIGKDD Workshop on Useful Patterns*, UP'10, pages 17–26, New York, NY, USA, 2010. ACM.

[cor] Coresight. `http://www.arm.com/products/system-ip/coresight/index.php`.

[CW10] Po-Hsien Chang and Li.-C Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 607–612, Piscataway, NJ, USA, 2010. IEEE Press.

[DFLS06] Giuseppe Di Fatta, Stefan Leue, and Evghenia Stegantova. Discriminative pattern mining in software fault detection. In *In Proceedings of the 3rd international workshop on Software quality assurance*, SOQUA '06, pages 62–69, New York, NY, USA, 2006. ACM.

[FDBM06] Clment Faure, Sylvie Delprat, Jean-Francois Boulicaut, and Alain Mille. Iterative bayesian network implementation by using annotated association rules. In *Proceedings 15th International Conference on Knowledge Engineering and Knowledge Management EKAW'06*, LNAI, pages 326–333. Springer, October 2006.

[FH00] P. Faraboschi and F. Homewood. ST200: A VLIW Architecture for Media-oriented applications. In *Microprocessor Forum*, San Jose, CA, October 2000.

[Gel89] David Gelernter. Multiple tuple spaces in Linda. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer Berlin Heidelberg, 1989.

[gst] Gstreamer debugging tool. `http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/section-checklist-debug.html`.

[HC04] Kuo-Yu Huang and Chia-Hui Chang. Asynchronous periodic patterns mining in temporal databases. In *Databases and Applications*, pages 43–48, 2004.

[HCvW07] D. Holten, B. Cornelissen, and J.J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT 2007, pages 47–54, 2007.

[HDY99] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of the 15th International Conference on Data Engineering*, ICDE '99, pages 106–115, Washington, DC, USA, 1999. IEEE Computer Society.

[HGY98] Jiawei Han, Wan Gong, and Yiwen Yin. Mining segment-wise periodic patterns in time-related databases. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 214–218, 1998.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. "mining frequent patterns without candidate generation". In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[IBM] IBM Quest Synthetic Data Generator. `http://sourceforge.net/projects/ibmquestdatagen`.

[JW05] A.A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, February 2005.

[KAL] KALRAY. Multi-Purpose Processor Array. `http://www.kalray.eu/products/mppa-manycore/mppa-256/`.

[Kan02] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[KMW12] Christian Koehler, Albrecht Mayer, and Maximilian Wurm. Combined hardware and software tracing of real and virtual embedded system parts. In *Mixed Design of Integrated Circuits and Systems (MIXDES), 2012 Proceedings of the 19th International Conference*, pages 340 –345, may 2012.

[kpt] Kptrace. `http://www.stlinux.com/devel/traceprofile/kptrace`.

[KR10] R. Uday Kiran and P. Krishna Reddy. Mining periodic-frequent patterns with maximum items' support constraints. In *Proceedings of the Third Annual ACM Bangalore Conference*, COMPUTE '10, pages 1:1–1:8, New York, NY, USA, 2010. ACM.

[KR11] R.Uday Kiran and P.Krishna Reddy. An alternative interestingness measure for mining periodic-frequent patterns. In JeffreyXu Yu, MyoungHo Kim, and Rainer Unland, editors, *Database Systems for Advanced Applications*, volume 6587 of *Lecture Notes in Computer Science*, pages 183–192. Springer Berlin Heidelberg, 2011.

[Kri05] R. Krishnakumar. Kernel korner: KProbes-A kernel debugger. *Linux J.*, 2005(133):11–, May 2005.

[KS05] Youngsoo Kim and Suleyman Sair. Designing real-time h.264 decoders with dataflow architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 291–296, New York, NY, USA, 2005. ACM.

[KSS05] Daniel A. Keim, Jrn Schneidewind, and Mike Sips. FP-Viz: Visual Frequent Pattern Mining. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '05), Poster Paper*, Minneapolis, USA, 2005.

[KWK10] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: lessons learned from five industrial projects. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 315–329, Berlin, Heidelberg, 2010. Springer-Verlag.

[LC09] Carson Kai-Sang Leung and Christopher L. Carmichael. Fpviz: a visualizer for frequent pattern mining. In *Proceedings of the ACM SIGKDD Workshop on Visual Analytics and Knowledge Discovery: Integrating Automated Analysis with Interactive Exploration*, VAKD '09, pages 30–39, New York, NY, USA, 2009. ACM.

[LCBT+12] Patricia Lopez-Cueva, Aurélie Bertaux, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. Debugging Embedded Multimedia Application Traces Through periodic pattern mining. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2012*, Tampere, Finland, October 2012.

[LCH+09] 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*, KDD '09, pages 557–566, New York, NY, USA, 2009. ACM.

[LcKL07] David Lo, Siau cheng Khoo, and Chao Liu. Mining temporal rules from program execution traces. *International Workshop on Program Comprehension*, 2007.

[LCSZ04] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 173–186, Berkeley, CA, USA, 2004. USENIX Association.

[LIC08a] Carson Kai-Sang Leung, Pourang Irani, and Christopher L. Carmichael. FIsViz: A Frequent Itemset Visualizer. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, PAKDD'08, pages 644–652, May 2008.

[LIC08b] Carson Kai-Sang Leung, Pourang P. Irani, and Christopher L. Carmichael. Wifisviz: Effective visualization of frequent itemsets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 875–880, Washington, DC, USA, 2008. IEEE Computer Society.

[LLMZ06] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[ltt] Linux trace toolkit - next generation. `https://lttng.org/`.

[LW95] Fritz Lehmann and Rudolf Wille. A Triadic Approach to Formal Concept Analysis. In *Proceedings of the Third International Conference on Conceptual Structures: Applications, Implementation and Theory*, volume 954 of *ICCS '95*, pages 32–43, London, UK, UK, 1995. Springer-Verlag.

[LXM08] Christopher LaRosa, Li Xiong, and Ken Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 880–885, New York, NY, USA, 2008. ACM.

[LYY+05] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *In Proceedings of the 5th International Conference on Data Mining*, SDM'05, 2005.

[LZ05] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.

[MBF12] Diego Melpignano, Luca Benini, and Eric Flamand. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, 2012.

[MC03] Grant Martin and Henry Chang, editors. *Winning the SoC revolution : experiences in real design*. Kluwer Academic Publishers, Boston, London, 2003.

[Meh02] Katharina Mehner. Javis: A uml-based visualization and debugging environment for concurrent java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 163–175, London, UK, UK, 2002. Springer-Verlag.

[MH01] Sheng Ma and Joseph L. Hellerstein. Mining Partially Periodic Event Patterns with Unknown Periods. In *Proceedings of the 17th International Conference on Data Engineering*, pages 205–214, Washington, DC, USA, 2001. IEEE Computer Society.

[MKN+05] Tamara Munzner, Qiang Kong, Raymond T. Ng, Jordan Lee, Janek Klawe, Dragana Radulovic, and Carson K. Leung. Visual Mining of Power Sets with Large Alphabets. Technical Report TR-2005-25, Department of Computer Science, The University of British Columbia, Vancouver, BC, Canada, 2005.

[mpe] Mpeg-2 standard. `http://mpeg.chiariglione.org/standards/mpeg-2`.

[MT96] Heikki Mannila and Hannu Toivonen. Multiple uses of frequent sets and condensed representations (extended abstract). In *In Proceedings of the 2nd International Conference on Knowledge Discovery in Databases*, KDD'96, pages 189–194, Portland, USA, 1996. AAAI Press.

[mtt] Multi-target trace api. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00053272.pdf`.

[NAW+96] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[NTMU10] B. Negrevergne, A. Termier, J. Mehaut, and T. Uno. Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses. In *International Conference on High Performance Computing and Simulation*, HPCS'10, pages 521–528, 2010.

[ORS98] Banu Özden, Sridhar Ramaswamy, and Abraham Silberschatz. Cyclic association rules. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 412–421, Washington, DC, USA, Feb 1998. IEEE Computer Society.

[PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Inf. Syst.*, 24(1):25–46, March 1999.

[PHM00] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[PHMa+01] 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 *Proceedings of the 17th International Conference on Data Engineering*, ICDE'01, pages 215–224, 2001.

[PRRR+09] 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*, Sophia de Antipolis, France, September 2009.

[PTBS09] Serge De Paoli, Marc Titinger, Roland Bohrer, and Miguel Santana. Enabling new Device Software Optimization features thanks to System Trace Module. In *Conference on System Software, SoC and Silicon Debug S4D 2009*, Sophia de Antipolis, France, September 2009.

[Sem] NPX Semiconductors. PNX4008. `http://forum2.mobile-review.com/attachment.php?attachmentid=8276&d=1178712994`.

[Shn86] Ben Shneiderman. *Designing the user interface: strategies for effective human-computer interaction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Sno] Snowball. `http://www.igloocommunity.org/`.

[Soc] SOC-TRACE project. `http://www.minalogic.com/TPL_CODE/TPL_PROJET/PAR_TPL_IDENTIFIANT/2717/15-annuaire-innovations-technologiques-nanotechnologie-systeme-embarque.htm`.

[sta09] Stapi-skd v.a18, release notes, 2009.

[STL] STWorkbench. `http://www.stlinux.com/stworkbench/`.

[STMa] STMicroelectronics. ST40 processor. `http://www.st.com/web/en/catalog/mmc/FM141/SC1714/LN1030`.

[STMb] STMicroelectronics. STi7200-MBoard platform. `http://www.st.com/web/catalog/mmc/FM131/SC999/SS1629/PF160130?s_searchtype=partnumber`.

[TAJL09] Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong, and Young-Koo Lee. Discovering periodic-frequent patterns in transactional databases. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD '09, pages 242–253, Berlin, Heidelberg, 2009. Springer-Verlag.

[tra] Trace32. `http://www.lauterbach.com/publications/instrumentation_trace.pdf`.

[UAUA04] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In Einoshin Suzuki and Setsuo Arikawa, editors, *Discovery Science*, volume 3245 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[VKR+08] Bart Vermeulen, Rolf Kühnis, Jeff Rearick, Neal Stollon, and Gary Swoboda. Overview of debug standardization activities. *IEEE Design Test of Computers*, 25(3):258–267, May 2008.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[Wil95] Rudolf Wille. The Basic Theorem of triadic concept analysis. *Order*, 12:149–158, 1995.

[WJM08] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.

[Wol01] Wayne Wolf. *Computer as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, San Francisco, California, USA, 2001.

[Wol04] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 681–685. ACM, 2004.

[wPOH09] Hae woo Park, Hyunok Oh, and Soonhoi Ha. Multiprocessor SoC design methods and tools. *Signal Processing Magazine, IEEE*, 26(6):72–79, November 2009.

[Yan03] Li Yang. Visualizing frequent itemsets, association rules, and sequential patterns in parallel coordinates. In Vipin Kumar, MarinaL. Gavrilova, Chih-JengKenneth Tan, and Pierre LEcuyer, editors, *Computational Science and Its Applications ICCSA 2003*, volume 2667 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2003.

[Yan05] Li Yang. Pruning and visualizing generalized association rules in parallel coordinates. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):60–70, January 2005.

[YL04] Wenpo Yang and Guanling Lee. Efficient partial multiple periodic patterns mining without redundant rules. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, volume 1 of *COMPSAC 2004*, pages 430–435, 2004.

[YWY03] Jiong Yang, Wei Wang, and Philip S. Yu. Mining asynchronous periodic patterns in time series data. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):613–628, March 2003.

[ZjH02] Mohammed J. Zaki and Ching jui Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining*, pages 457–473, April 2002.

[ZXHW10] Jia Zou, Jing Xiao, Rui Hou, and Yanqi Wang. Frequent instruction sequential pattern mining in hardware sample data. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 1205–1210, Washington, DC, USA, 2010. IEEE Computer Society.

Abstract

Increasing complexity in both the software and the underlying hardware, and ever tighter time-to-market pressures are some of the key challenges faced when designing multimedia embedded systems. Optimizing software debugging and validation phases can help to reduce development time significantly. A powerful tool used extensively when debugging embedded systems is the analysis of execution traces. However, evolution in embedded system tracing techniques leads to execution traces with a huge amount of information, making manual trace analysis unmanageable. In such situations, pattern mining techniques can help by automatically discovering interesting patterns in large amounts of data. Concretely, in this thesis, we are interested in discovering periodic behaviors in multimedia applications. Therefore, the contributions of this thesis are focused on the definition of periodic pattern mining techniques for the analysis of multimedia applications execution traces.

Regarding periodic pattern mining contributions, we propose a definition of periodic pattern adapted to the characteristics of concurrent software. We then propose a condensed representation of the set of frequent periodic patterns, called Core Periodic Concepts (CPC), by adopting an approach originated in triadic concept approach. Moreover, we define certain connectivity properties of these patterns that allow us to implement an efficient CPC mining algorithm, called PerMiner. Then, we perform a thorough analysis to show the efficiency and scalability of PerMiner algorithm. We show that PerMiner algorithm is at least two orders of magnitude faster than the state of the art. Moreover, we evaluate the efficiency of PerMiner algorithm over a real multimedia application trace and also present the speedup achieved by a parallel version of the algorithm.

Then, regarding embedded systems contributions, we propose a first step towards a methodology which aims at giving the first guidelines of how to use our approach in the analysis of multimedia applications execution traces. Besides, we propose several ways of preprocessing execution traces and a competitors finder tool to postprocess the mining results. Moreover, we present a CPC visualization tool, called *CPCViewer*, that facilitates the analysis of a set of CPCs. Finally, we show that our approach can help in debugging multimedia applications through the study of two use cases over real multimedia application execution traces.

Résumé

La conception des systèmes multimédia embarqués présente de nombreux défis comme la croissante complexité du logiciel et du matériel sous-jacent, ou les pressions liées aux délais de mise en marche. L'optimisation du processus de débogage et validation du logiciel peut aider à réduire sensiblement le temps de développement. Parmi les outils de débogage de systèmes embarqués, un puissant outil largement utilisé est l'analyse de traces d'exécution. Cependant, l'évolution des techniques de traçage dans les systèmes embarqués se traduit par des traces d'exécution avec une grande quantité d'information, à tel point que leur analyse manuelle devient ingérable. Dans ce cas, les techniques de recherche de motifs peuvent aider en trouvant des motifs intéressants dans de grandes quantités d'information. Concrètement, dans cette thèse, nous nous intéressons à la découverte de comportements périodiques sur des applications multimédia. Donc, les contributions de cette thèse concernent l'analyse des traces d'exécution d'applications multimédia en utilisant des techniques de recherche de motifs périodiques fréquents.

Concernant la recherche de motifs périodiques, nous proposons une définition de motif périodique adaptée aux caractéristiques de la programmation paralléle. Nous proposons ensuite une représentation condensée de l'ensemble de motifs périodiques fréquents, appelée Core Periodic Concepts (CPC), en adoptant une approche basée sur les relations triadiques. De plus, nous définissons quelques propriétés de connexion entre ces motifs, ce qui nous permet de mettre en oeuvre un algorithme efficace de recherche de CPC, appelé PerMiner. Pour montrer l'efficacité et le passage à l'échelle de PerMiner, nous réalisons une analyse rigoureuse qui montre que PerMiner est au moins deux ordres de grandeur plus rapide que l'état de l'art. En plus, nous réalisons un analyse de l'efficacité de PerMiner sur une trace d'exécution d'une application multimédia réelle en présentant l'accélération accompli par la version parallèle de l'algorithme.

Concernant les systèmes embarqués, nous proposons un premier pas vers une méthodologie qui explique comment utiliser notre approche dans l'analyse de traces d'exécution d'applications multimédia. Avant d'appliquer la recherche de motifs fréquents, les traces d'exécution doivent être traitées, et pour cela nous proposons plusieurs techniques de pré-traitement des traces. En plus, pour le post-traitement des motifs périodiques, nous proposons deux outils : un outil qui trouve des pairs de motifs en compétition ; et un outil de visualisation de CPC, appelé *CPCViewer*. Finalement, nous montrons que notre approche peut aider dans le débogage des applications multimédia à travers deux études de cas sur des traces d'exécution d'applications multimédia réelles.