Data Mining Approach to Temporal Debugging of Embedded Streaming Applications

Oleg Iegorov, Alexandre Termier, Vincent Leroy, Jean-François Méhaut, Miguel Santana

Data Mining Approach to Temporal Debugging of Embedded Streaming Applications

Oleg Iegorov
STMicroelectronics and
Université de Grenoble Alpes,
LIG, Grenoble, France
oleg.iegorov@imag.fr

Vincent Leroy
Université de Grenoble Alpes,
LIG and CNRS,
Grenoble, France
vincent.leroy@imag.fr

Alexandre Termier
Université de Rennes 1, IRISA
and INRIA Rennes Bretagne
Atlantique, Rennes, France
alexandre.termier@irisa.fr

Jean-François Méhaut
Université de Grenoble Alpes,
LIG and CEA LETI,
Grenoble, France
jean-francois.mehaut@imag.fr

Miguel Santana
STMicroelectronics, France
miguel.santana@st.com

ABSTRACT

One of the greatest challenges in the embedded systems area is to empower software developers with tools that speed up the debugging of QoS properties in applications. Typical streaming applications, such as multimedia (audio/video) decoding, fulfill the QoS properties by respecting the real-time deadlines. A perfectly functional application, when missing these deadlines, may lead to cracks in the sound or perceptible artifacts in the image.

We start from the premise that most of the streaming applications that run on embedded systems can be expressed under a dataflow model of computation, where the application is represented as a directed graph of the data flowing through computational units called actors. It has been shown that in order to meet real-time constraints the actors should be scheduled in a periodic manner. We exploit this property to propose *SATM* – a novel approach based on data mining techniques that automatically analyzes execution traces of streaming applications, and discovers significant breaks in the periodicity of actors, as well as potential causes of these breaks. We show on a real use case that our debugging approach can uncover important defects and pinpoint their location to the application developer.

1. INTRODUCTION

Designing applications for embedded systems, such as set-top boxes or smartphones, is one of the most challenging areas of software development. With each hardware generation, more powerful and complex Systems-on-Chip (SoC) are released, and developers must constantly strive to adapt their applications to these new platforms. Streaming applications require particular attention to ensure that strict QoS properties are preserved. For example, a multimedia player should neither show artifacts in the video nor cracks in the sound.

Debugging QoS properties proves to be tricky, as these properties are not related to the functional correctness of the code for which traditional debuggers are designed. Their violation can result from complex interactions between the application and the system, or other applications: the complete execution context must be taken into account for debugging. The usual solution is to capture a *trace* of the execution, and to analyze it afterwards to understand what went wrong. However, such traces can have a large volume, and understanding them requires data analysis skills that are currently out of the scope of developer's education.

In this paper, we propose SATM (**S**treaming **A**pplication **T**race **M**iner) – a novel approach to help understand the violations of QoS properties in streaming applications. SATM is based on the premise that such applications are designed under the *dataflow* model of computation. In this model, an application is represented as a directed graph where the nodes are called *actors* and the data flows between actors along the arcs. In such a setting, as we explain in Section 2, the actors must be scheduled in a periodic way in order to meet QoS properties expressed as real-time constraints, such as displaying 30 video frames per second. In Section 3, we show that an actor which does not eventually respect its period at runtime causes the violation of application's real-time constraints. These two sections form the theoretical foundation of SATM.

In practice, SATM is a *data analysis workflow* combining statistical approaches and data mining algorithms. It provides an automatic solution to the problem of temporal debugging of streaming applications (Fig 1). Given an execution trace of a streaming application exhibiting low QoS as well as a list of its actors, SATM firstly determines exact actors' invocations found in the trace (Section 4.1). It then discovers the actors' periods, as well as parts of the trace in which the periods are not respected (Section 4.2). Those parts are further analyzed to extract patterns of system activity that differentiate them from other parts of the trace (Section 5). Such patterns can give strong hints on the origin of the problem and are returned to the developer. We demonstrate SATM's ability to detect both an artificial per-

turbation injected in an open source multimedia framework, and a complex temporal bug of an industrial application (Section 6).

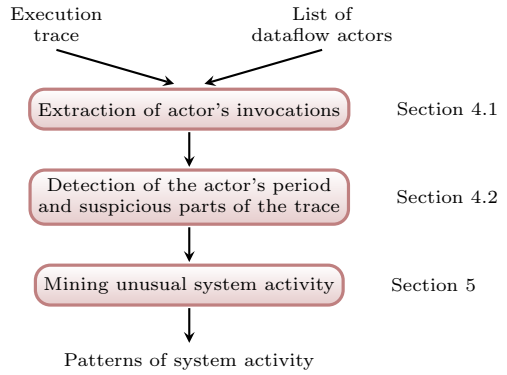

Figure 1: SATM data analysis workflow

2. BACKGROUND ON EMBEDDED STREAMING APPLICATIONS

In this section, we introduce domain knowledge of streaming applications and the difficulties of debugging their QoS properties. We conclude with our problem statement.

2.1 Design of Streaming Applications

Embedded applications are expected to run efficiently on various chips with minor manual code rewriting. To answer this demand, Model-of-Computation based design has emerged as a widely used standard to express the semantics of the interaction between the components of an embedded application. Different types of Models of Computation (MoC) include Finite State Machine, Dataflow, Discrete Event and others [7]. An embedded application described using a specific MoC is used as a part of specification in electronic system-level (ESL) design tools [12], whose goal is to automatically generate code from MoC-based description to be executed on a particular SoC.

The widely adopted MoC to design streaming applications is Dataflow [18]. With Dataflow MoC, the program is modeled as a directed graph where vertices (or *actors*) process data units (or *tokens*), and edges (or *communication channels*) transfer data, with the requirement that vertices cannot share data. This last requirement makes dataflow particularly suitable to express the parallelism of streaming applications. Popular Dataflow MoCs include Synchronous Dataflow (SDF) and Cyclo-Static Dataflow (CSDF) [18].

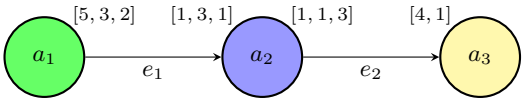

Figure 2: Example CSDF graph

Figure 2 presents a simple CSDF graph G containing three actors a_1, a_2, a_3 and two communication channels e_1, e_2. Each time an actor is invoked, it produces a specific number of tokens to its outbound communication channel and consumes a specific number of tokens from the inbound communication channel, as specified in the square brackets. Consider actor a_2. The first time a_2 is invoked, it consumes 1 token and produces 1 token. The next time it is invoked, it consumes 3 tokens and produces 1 token, the next time - consumes 1 token and produces 3 tokens, then - 1 token is consumed and 1 token is produced again, and so on.

2.2 Hard Real-Time Constraints

Multimedia streaming applications conceived to run in modern consumer electronic (CE) devices process high volumes of arriving data streams with sophisticated algorithms in order to provide high quality output. Indeed, the customer wants her set-top box to decode a high-definition video streaming from Internet TV provider and to reproduce it smoothly and perfectly synchronized with audio and subtitles (required QoS properties). One approach to meet these strict requirements would be the use of hardware solutions designed to execute a specific application. However, the diversity of applications the modern CE device must support (flexibility requirement), and the desire of application developers to use their software on a wide set of multimedia platforms (openness requirement), require the use of software-based solutions. This implies that all the complexity of multimedia streaming algorithms and application scheduling must be addressed at the software level. Dataflow MoC provides an efficient way of designing parallel multimedia streaming applications. Such applications must run under hard real-time constraints, so that application's output is never delivered later than a specific deadline, in order to meet QoS of modern CE devices (constant bit-rate, no jitter, and others) [5]. Indeed, there is an emerging evidence of using software based solutions that require hard real-time performance [9].

The majority of hard real-time scheduling algorithms deal with applications modeled as a set of periodic or sporadic independent tasks [9]. Such model, however, does not apply to dataflow applications where actors are not independent, but have data-dependency constraints. On the other hand, it was analytically proven that embedded streaming applications modeled with Dataflow MoC can be scheduled as a set of *strictly periodic tasks* [2] [3], meaning that each actor is scheduled as a task which is invoked at strict moments of time defined by its period [1]. This means that a variety of hard real-time scheduling algorithms can be applied on an application, given its dataflow model and worst-case execution time (WCET) of each of its actors (computed with static analysis tools or profiling on the target platform [19]).

2.3 Temporal Debugging

A typical usage scenario of a CE device includes several concurrent real-time applications sharing the SoC resources. For example, a set-top box decodes a video stream (to be shown on the screen) and simultaneously encodes it (to be stored in a different format on an external USB hard drive). That is why embedded multimedia applications usually run in general-purpose environment. On the other hand, due to concurrent activity of applications and operating system, it can happen that estimated WCETs of tasks are not respected. Application's hard real-time requirements, thus, are violated resulting in a drop in application QoS, and debugging must be performed in order to resolve problematic

[1]In the rest of the paper, the modeling term *actor* and its scheduling counterpart term *task* will be used interchangeably, if not stated otherwise

system behavior.

Debugging QoS properties is not a trivial task. On one hand, traditional debugging tools focus on functional correctness (i.e. accurate output values), but do not address temporal correctness (i.e. output is available no later than a given deadline). On the other hand, the process of debugging should not be intrusive, otherwise it will alter application's temporal behavior. Temporal debugging using execution traces answers both of these problems: it allows to analyze the whole system activity, checking the time elapsed between any system events; at the same time, trace collection imposes minimal overhead thanks to dedicated tracing hardware, like Embedded Trace Macrocell in modern ARM processors [2].

Traces are usually hard to analyze. First of all, an execution trace contains a sequence of timestamped system events with no application model semantics attached to them. There is no generic way to know if `processA` followed by `processB` in the trace have some semantic meaning, or they are completely independent and happen in succession because of OS scheduling. Moreover, there is no starting point of debugging in the trace, as the QoS observed at a given moment of time may be caused by some earlier system activity. Finally, a trace can easily contain more than a million of events for just a few minutes of system tracing. As the result, programmers are often overwhelmed with the amount of raw data in execution traces. A promising solution is to exploit *data mining* techniques to automatically extract relevant information from such large volumes of data.

2.4 Problem Statement

The goal of this paper is to facilitate the temporal debugging of embedded streaming applications by applying data mining techniques on application execution traces. We now formally state the problem addressed in this paper:

Having an execution trace and a dataflow model of an embedded multimedia application that does not meet its QoS requirements, find out automatically which system activity is responsible for application's faulty temporal behavior.

3. PROPAGATION OF EXECUTION DELAY IN DATAFLOW GRAPHS

In this section, we argue that a streaming application with low QoS, modeled with the dataflow MoC and scheduled under hard real-time constraints, must contain at least one actor that does not respect its period at runtime. We introduce the notion of *execution delay propagation* that justifies this argument.

Hard real-time scheduling of a dataflow graph G consists in finding out how to execute graph actors $a_i, i = 1..N$ as a set of strictly periodic tasks τ_i. Each τ_i is defined by a tuple $\tau_i = (S_i, D_i, P_i)$, where $S_i \geq 0$ is the start time of τ_i, $D_i \geq WCET_i$ is its deadline and $P_i \geq D_i$ is its period, so that τ_i is invoked at $t = S_i + mP_i, m = 0, .., \infty$ and executes for no longer than D_i.

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

$$q_1P_1 = q_2P_2 = \cdots = q_{N-1}P_{N-1} = q_NP_N = \alpha \text{ [2]}, \quad (1)$$

while all the q_i can be directly found from production and/or consumption properties of actors in dataflow graph G [14] [4]. The product q_iP_i designates the duration of actor a_i's *iteration*, and, as can be seen from Equation 1, has the same value α for all the actors in G. Let's denote by a_{i+1} the actor that consumes the tokens produced by a_i. A valid periodic schedule guarantees that if

$$S_{i+1} = S_i + q_iP_i \quad (2)$$

then a_{i+1} will always find the required number of input tokens each time it is invoked [2].

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

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

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

$$t_i^{k+1} = t_i^k + P_i. \quad (3)$$

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

It can be seen from Equations 1 and 2 that the start time of a_{i+1}'s p^{th} iteration is equal to the start time of a_i's $(p+1)^{th}$ iteration. Thus, if a_i's $(p+1)^{th}$ iteration is delayed, then a_{i+1}'s p^{th} is delayed as well. The same applies to all the a_j, $i \leq j \leq N$, which means, that the p^{th} iteration of a_N is shifted by $delay_i^e$, hence, application's output is delayed by $delay_l^e$ □

Figure 4 helps to understand the notion of execution delay propagation using the schedule from Figure 3. As a_1's e^{th} invocation took longer than P_1 to execute, $delay_1^e = exec\_time_1^e - P_1$ was introduced to the pipeline, which propagated through a_2 and a_3 actors resulting in the delayed output of the application.

The proposition shows that in order to explain application's delayed output, it is essential to find the first actor in application dataflow graph that occasionally does not respect its period.

4. DISCOVERING ACTOR PERIOD FROM APPLICATION EXECUTION TRACE

In order to identify delayed actor's invocations, one needs first to determine the actor's period. Although it is possible that the person who performs the debugging is aware of the periods of all the dataflow actors (e.g. the scheduling is hard-coded into the program, or the information from

[2] `http://arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php`

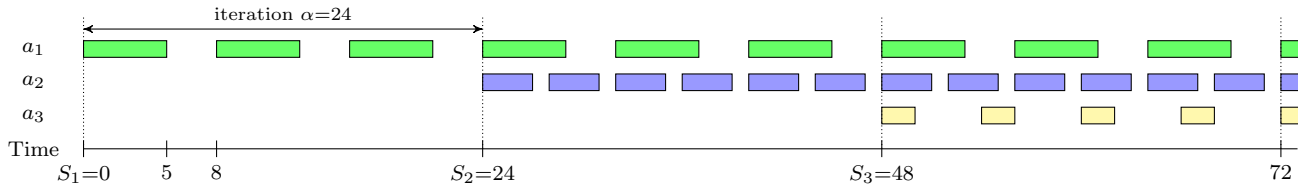

Figure 3: Valid periodic schedule for CSDF graph from Figure 2

the scheduler is easily accessible), we argue that automatic extraction of the period from the execution trace provides multiple benefits. Firstly, it relieves the need to manually look for the parts of the trace where the period of a given actor is not respected. Secondly, it makes SATM suitable to the users who are oblivious of the scheduling decisions, which is often the case in industry where the performance debugging teams are separated from the developers. Therefore, SATM mines actors' periods from the execution trace. The first actor exhibiting significant delay in its period is signaled for further analysis.

We firstly provide the necessary formalism. Let $\mathcal{E} = \{e_1, e_2, .., e_m\}$ be the set of all system and application *events* that the user has decided to trace (for example, *Interrupt168* or *sys_read[process23]*). Each line of a trace file contains a *timestamped event* of a particular type e that started to execute on the CPU at a specific timestamp ts: $evt = (ts, e)$; $ts \in \mathbb{N}, e \in \mathcal{E}$. A *trace*, thus, can be represented as a sequence of timestamped events $\langle evt_1, evt_2, \ldots, evt_l \rangle = \langle (ts_1, e_1), (ts_2, e_2), ..., (ts_l, e_l) \rangle$ where $\forall i \in [1, l]$ $ts_i \in \mathbb{N}, e_i \in \mathcal{E}$, and the trace events are ordered by increasing order of timestamp. For example, $\langle$(1, *Interrupt*168),(4, *sys_read* [*process*23]), (6,*Interrupt*168)$\rangle$ is a simple trace of length 3.

The mapping between a dataflow actor and the corresponding event in the execution trace is done in the following way. If the given actor consists of a single function, then each appearance of this function in the trace is considered as actor invocation. In case when the actor represents a set of functions or a specific software module, then the appearance of the first function (the one which reads the actor's input from the communication channel) is considered as actor invocation.

The first difficulty of discovering an actor period from a trace comes from the observation that we can not directly compare the time elapsed between the consecutive actor occurrences; the reason being preemptive multitasking operating system. As actor execution can be preempted in favor of other processes and system tasks, the trace may contain several actor occurrences, even if semantically they belong to one actor invocation. Consider Figure 5 which presents the visualization of an excerpt from system execution trace. We observe 3 invocations of actor a_i at timestamps 152, 177 and 202 with the period equal to 25ms (hollow rectangles), while execution trace contains 11 occurrences of a_i (filled rectangles), as the operating system preempted it 3 times during first execution, then 2 and 3 times during second and third execution respectively. If we directly compare all the time intervals between consecutive actor occurrences that is {3,5,3,14,6,4,15,5,3,3}, the lack of constant value among them would signal non-periodic invocation rate of actor a_i, which is wrong. A preprocessing is thus needed in order to group actor occurrences in the trace into semantic invocations.

4.1 Temporal clustering of actor occurrences

Grouping similar objects is the goal of cluster analysis, a branch of the data mining field. In our particular case, the objects are the occurrences in the trace of a specific actor, and the similarity measure of two occurrences consists of only one dimension: the elapsed time between them. This way, the less time has elapsed between two occurrences, the more similar they are. The requirement that we impose on clustering is its fully automatic behavior, i.e. no user involvement is required. Interestingly, the vast body of clustering algorithms deals with high-dimensional data and various manually set thresholds, leaving automatic one-dimensional clustering without proper attention. In [8] Cooper et al. propose an algorithm for unsupervised temporal clustering of digital photo collections which they prove to be accurate and efficient. With only a slight enhancement, we were able to apply their unsupervised time-based similarity analysis to accurately and automatically cluster actor invocations, as explained below.

The first step of clustering actor occurrences consists in the construction of a similarity matrix M, which entries $M[m][n]$ contain the similarity measure between the m^{th} and the n^{th} actor occurrences. Pairs of occurrences that are closer to each other in time will have bigger similarity val-

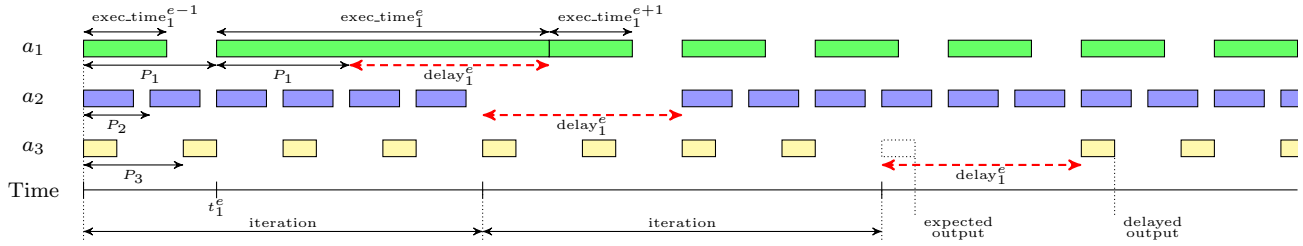

Figure 4: Propagation of execution delay

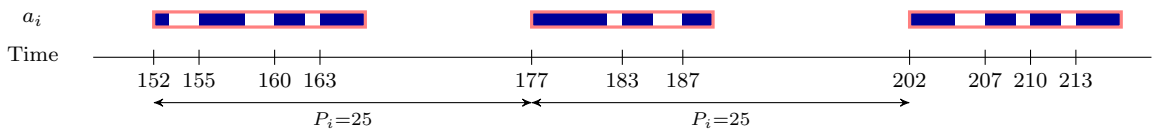

Figure 5: Results of the operating system preemption of an actor a_i

ues. Consider the similarity matrix for the case of Figure 5 depicted on Figure 6a. The darker color denotes the higher similarity value, hence the smaller time distance between the pairs of actor occurrences. The three clusters which correspond to three actor invocations are clearly visible along the main diagonal.

The second step deals with the detection of potential cluster borders using *novelty scores* computed for each actor occurrence. The novelty score quantifies the dissimilarity of the groups of occurrences both before and starting at the target actor occurrence. A big value of novelty score thus means that the new cluster starts at the given actor occurrence. With regard to Figure 5, the novelty score of occurrences 1, 5 and 8 will be greater than those of other occurrences, as shows Figure 6b.

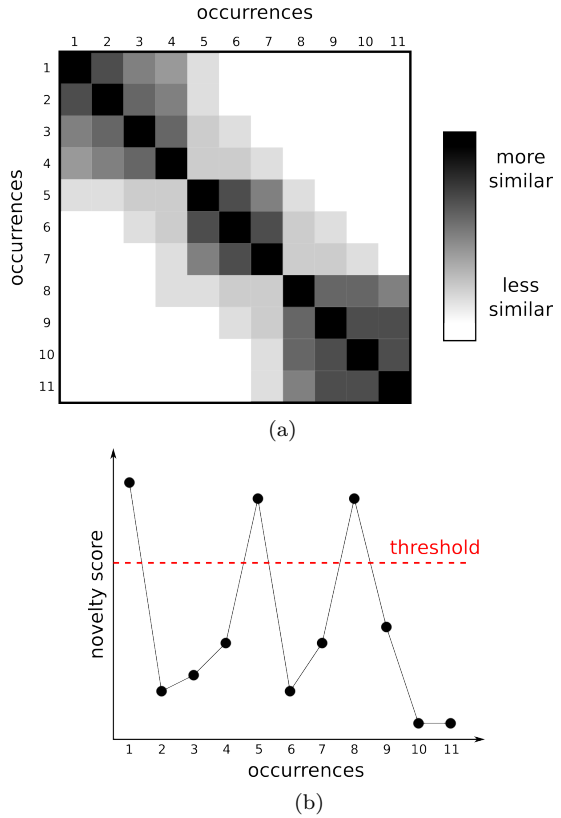

Figure 6: Similarity matrix M (a) and novelty scores (b) for actor occurrences from Figure 5

The last step concerns the detection of actor occurrences that correspond to actor invocations. Indeed, as Figure 6b shows, the threshold, that classifies novelty scores as "big enough" or "not big enough", should be chosen. We have enriched the time-based similarity analysis of [8] to do this step automatically, using the Otsu's threshold selection method [17]. Otsu's state-of-the-art method allows to accurately transform any grayscale image into its black and white counterpart. It allows to decide whether a given grayscale pixel is closer to "white" or "black" by automatically finding a threshold that minimizes the intra-class variance for both "white" and "black" pixel classes. Otsu's method is thus directly applicable to find out if a particular occurrence's novelty score is closer to "cluster-defining" or to "non-cluster-defining" class.

In practice, SATM clusters actor's invocations with high accuracy. The reason of this is twofold. On one hand, the actor's period is normally quite bigger than its worst-case execution time. On the other hand, valid use cases usually do not overutilize the SoC's resources. Hence, no heavy preemption takes place, and the actor's occurrences related to a single invocation are compactly grouped on the temporal axis.

4.2 Discovering actor period

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

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

$$QCoD = \frac{Q_3 - Q_1}{Q_3 + Q_1}, \tag{4}$$

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

In case if the value of $QCoD$ for the distribution of time intervals between actor invocations is small enough to consider the actor as periodic, we proceed to detect the gaps in actor's periodicity, i.e. time intervals that are much bigger than actor's period. The precise "much bigger" value can be

obtained from the inter-quartile rule for outliers: distribution elements that fall above $Q_3 + 1.5 * IQR$ are outliers, where $IQR = Q3 - Q1$ is the inter-quartile range.

Consider the following (clustered) timestamps of actor's a_i invocations extracted from an execution trace: $t_i^1 = 45, t_i^2 = 75$, $t_i^3 = 104$, $t_i^4 = 134$, $t_i^5 = 164$, $t_i^6 = 352$, $t_i^7 = 382$, $t_i^8 = 413$, $t_i^9 = 443$, $t_i^{10} = 538$, $t_i^{11} = 568$. The corresponding distribution of time intervals between its invocations is

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

and the median equals to 30. Next, we decide if this value can be considered as the actor's period, i.e. if most of the time intervals tend to be equal to the median, using QCoD dispersion measure (Equation 4). $Q_1 = 30$, $Q_3 = 31$, and $QCoD = 1/61 \approx 0.016$. The small value of $QCoD$ implies that distribution σ is well centered on its median, even with the presence of two big outliers. Hence, we infer that the actor a_i is periodic with the period equal to 30. We then apply the inter-quartile rule, which detects 95 and 188 to be outliers in σ, as these values fall above $Q_3 + 1.5 * IQR = 32.5$.

Assuming that a_i is the first actor from the dataflow model which has outliers in its distribution of occurrences in the execution trace, it is essential to discover the cause of the presence of these outliers which, as proved in Section 3, result in reduced QoS.

Having a_i's timestamps and the list of its outliers, the execution trace D can be split into two sets: D_{neg}, where the target actor is invoked periodically (outlier-negative), and D_{pos}, where its period is broken (outlier-positive). Using the example above, $D_{neg} = \{ [0, t^1]; [t^1,t^2]; [t^2,t^3]; [t^3,t^4]; [t^4,t^5]; [t^6,t^7]; [t^7,t^8]; [t^8,t^9]; [t^{10},t^{11}]; [t^{11},t^{last}] \}$, $D_{pos} = \{[t^5,t^6]; [t^9,t^{10}]\}$ where $[t^i, t^j]$ denotes the part of the trace between the timestamps t^i and t^j and is called a *subtrace*, while t^{last} is the timestamp of the last event in the trace D.

5. MINING UNUSUAL SYSTEM ACTIVITY

As described in the previous section, SATM discovers the most upstream actor a_i with delayed invocations, and splits the execution trace D into two sets of subtraces: D_{neg} containing the parts of D between a_i's invocations that respect the period, and D_{pos}, containing the parts of D between a_i's invocations that do not respect the period. At the next stage, SATM determines what makes D_{pos} different from D_{neg}.

In data mining, discovering fine-grained differences between two or more datasets of categorical elements can be addressed by emerging pattern mining [10]. An *emerging pattern* is a group of data elements that appear much more frequently in one dataset than in another. If one considers trace events as data elements, and D_{pos} with D_{neg} as sets of trace events grouped into subtraces, emerging patterns can give a valuable insight into what makes these two datasets different. In fact, emerging patterns can be viewed as a concise representation of system activity observed more frequently while the period of the target actor was being violated, hence, such activity is unusual to the temporally correct execution of the target actor and, therefore, is correlated to the temporal bug. The user expertise is then needed in order to derive semantical meaning from the emerging patterns and decide if they truly explain the cause of the temporal bug. We now formally define the problem of mining emerging patterns in execution traces.

5.1 Emerging patterns in execution traces

Using the notation introduced in Section 4, a *pattern* is a group of events: $P = \langle e_1, e_2, \ldots, e_m \rangle$, where $\forall j \in [1, m]\ \ e_j \in \mathcal{E}$. For example, $\langle sys\_read[process23], Interrupt168 \rangle$ is a pattern of length 2. Note that a pattern has no timestamps attached to the events. The goal of emerging pattern mining is thus to discover *all the patterns* that are observed more frequently in the dataset D_{pos} than in the dataset D_{neg}. As D_{pos} and D_{neg} are used only to extract patterns, we can drop all the timestamps from them while preserving the order of the events which essentially makes subtraces in D_{pos} and D_{neg} sequences of events.

The given definition of a pattern is rather general, as it does not allow to define an occurrence of a pattern in a subtrace. We rely on the following observations on execution traces that provide sufficient information to decide if a given pattern is present in a given subtrace. Firstly, the order of events in a pattern is important, as a pattern ⟨readBuffer,writeBuffer⟩ has a different meaning from ⟨writeBuffer,readBuffer⟩ in the execution trace context. An ordered list of elements is called a *sequential pattern*. This way, a pattern P occurs in a subtrace $T \in D_{pos}$ (or $T \in D_{neg}$) if P is a subsequence of T ($P \subset T$); more formally, if $T = \langle e_1, e_2, \ldots, e_n \rangle$, then $P \subset T$ if $P = \langle e_{i_1}, e_{i_2}, \ldots, e_{i_m} \rangle$ such that $1 \leq i_1 < i_2 < \cdots < i_m \leq n$. Next, we expect the events that are far from each other in the trace to not have a direct connection. On the other hand, due to general-purpose nature of the execution environment, two logically connected events A and B (e.g. A calls B in the application's code) can be separated by some unrelated events in the execution trace. Therefore, the difference $i_{k+1} - i_k$, $k \in [1, m-1]$ must be upper-bounded with some g, $g \ll n$. In other words, if $P \subset T$, we would like P's elements to be close to each other in T, but not necessarily subsequent. We call g a maximum gap constraint. Summing up, $P \subset T = \langle e_1, e_2, \ldots, e_n \rangle$ if $P = \langle e_{i_1}, e_{i_2}, \ldots, e_{i_m} \rangle$ such that $1 \leq i_1 < i_2 < \cdots < i_m \leq i_n$ where $i_{k+1} - i_k \leq g$, and P is called a g-gap constrained sequential pattern.

Given a set of sequences D, a sequential pattern P and a maximum gap constraint g, the count of P in D with g-gap constraint, denoted as $count_D(P)$ is the number of sequences $T \in D$ in which P appears as a subsequence fulfilling the g-gap constraint. The support of P in D with g-gap constraint is defined as $sup_D(P, g) = count_D(P, g)/|D|$, where $|D|$ is the number of sequences in D. Given a positive threshold δ, if $sup_D(P, g) \geq \delta$ we say P is *frequent* in D. Otherwise, P is *infrequent* in D.

Finally, given two sets of sequences D_{pos} and D_{neg}, two support thresholds δ and α, and a maximum gap g, a pattern P is called an emerging pattern from D_{neg} to D_{pos} (abbreviated simply as *emerging pattern*) if $sup_{D_{pos}}(P, g) \geq \delta$ and $sup_{D_{neg}}(P, g) \leq \alpha$.

Having two emerging patterns P and Q, such that $P \subset Q$, P characterizes the differences between D_{pos} and D_{neg} better than Q, as it is more concise, hence, easier to analyze. An emerging pattern that does not have any emerging subpatterns is called a *minimal* emerging pattern. We are, therefore, interested in mining minimal emerging patterns.

As an example, having $\mathcal{E} = \{A, B, C, D, E, X\}$ consider the following D_{pos} and D_{neg}:

D_{pos}	ABXCD, ABXCED
D_{neg}	AXBCD, AXBECD, ABCED, AXBD

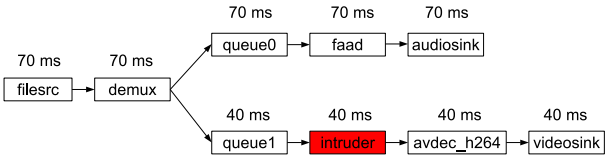

Figure 7: GStreamer dataflow graph with actors' periods detected by SATM

With $\delta = 100\%$, $\alpha = 0\%$ and $g = 1$, there are six emerging patterns: $\langle A, B, X\rangle$, $\langle A, B, X, C\rangle$, $\langle A, B, X, C, D\rangle$, $\langle B, X\rangle$, $\langle B, X, C\rangle$ and $\langle B, X, C, D\rangle$, but only $\langle B, X\rangle$ is minimal.

In [13] Ji et al. proposed an algorithm *ConSGapMiner* which mines minimal emerging gap-constrained sequential patterns from two classes of sequences. Being the only algorithm to mine this type of patterns, we have integrated ConSGapMiner as the last stage of SATM.

6. EXPERIMENTAL RESULTS

We evaluate SATM using manually perturbed GStreamer traces and a case of an industrial embedded application with low QoS. SATM is composed of an implementation of algorithms presented in Section 4 written in Perl, and of a multithreaded implementation of the ConSGapMiner algorithm written in Java.

6.1 GStreamer use case

GStreamer is an open source multimedia framework implementing the dataflow model. We build a pipeline composed of a standard set of GStreamer actors, with the addition of *intruder* actor (Fig 7). The role of *intruder* is to randomly introduce delay into the pipeline, and therefore perturb the video output of the application. This is done in the following way. *Intruder* implements 3 random bit generators: A, B and C. Each of them calls a specific function depending on the value of the generated bit (e.g. $A1$, $B0$). If a sequence "010" is generated (i.e. $A0$, $B1$ and $C0$ are called), *intruder* sleeps for a significant amount of time, so that the delay propagates through the subsequent actors resulting in late frame display, thus reducing GStreamer's QoS. Using this setup, we play 1 minute of video and record the execution trace.

6.1.1 Period computation

SATM successfully mined actors' periods, as shown in Fig. 7. The distribution of inter-occurrence intervals' lengths of *demux* and *avdec_h264* actors is presented in Figures 8 and 9 respectively. Note that given their position in the GStreamer graph, *demux* is not affected by the perturbations of *intruder*, while *avdec_h264* is. Although *demux* first appears to be aperiodic, the clustering eliminates small intervals, and SATM detects that *demux* occurs with a period of 70ms. The low dispersion value ($QCoD = 0.6\%$) confirms that the execution of *demux* is not perturbed. In practice, the values of $QCoD$ smaller than 10% indicate that the distribution is well centered around its median. Having $QCoD = 7.2\%$, *avdec_h264* is also a periodic actor, with a period of 40ms. However, its higher $QCoD$ value indicates that there is more dispersion around the median. As Figure 9 shows, the system preemption is not the cause of this dispersion, because the clustering does not remove the outliers. Hence, *avdec_h264* requires further investigation to identify the origin of the perturbation.

6.1.2 Mining Unusual System Activity

We now evaluate the ability of ConSGapMiner algorithm used at the last stage of SATM to identify the injected sequence of events $\langle A0, B1, C0\rangle$ which perturbs periodicity of *avdec_h264* actor, and thus of all the downstream actors resulting in delayed output delivery; we will call this sequence as *target* further on. The ideal output of this stage would be a single minimal emerging sequence *target*.

Fig 10 presents a heat map showing the number of minimal emerging sequences mined by ConSGapMiner for the fixed gap value $g = 1$ and varying δ and α.

We can pinpoint four areas of interest on this heat map, corresponding to different output quality. When $\delta \geq 94\%$ or $\alpha \leq 2\%$, *target* is not found. As most data mining approaches, SATM makes a few classification errors when building D_{pos} and D_{neg}. Hence, in this case, 2% of the subtraces in D_{neg} contain *target*. Similarly, a few subtraces in D_{pos} do not contain *target*, or contain it with a gap larger than 1. This area can easily be ignored by decreasing δ and increasing α accordingly. Then, for a large fraction of the parameter space: $\delta \in [82\%; 92\%]$ and $\alpha \in [2\%; 12\%]$ ("only target" region on the figure), SATM finds a single result, *target*, which corresponds to the ideal situation.

For $\alpha \geq 14\%$, the output of SATM exhibits anomalies coming from the perturbation process, as we explain below. The period of *intruder* actor being equal to the period of *avdec_h264* actor, each subtrace in D_{pos} and D_{neg} contains one execution of the *intruder* actor. Out of the 8 possible executions of *intruder* ($\langle A0,B0,C0\rangle$, $\langle A0,B0,C1\rangle$,..., $\langle A1,B1,C1\rangle$), D_{neg} is defined by 7 of them (all except $\langle A0, B1,C0\rangle$) that do not introduce a delay into the pipeline. Consequently, a sequence that differs from *target* by one el-

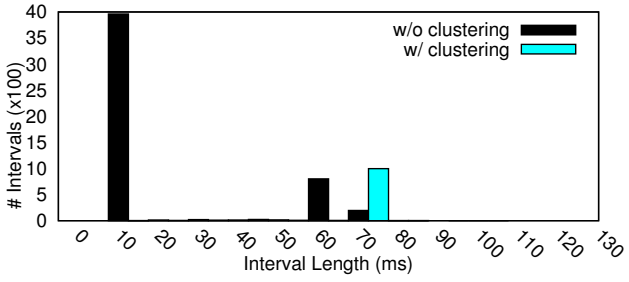

Figure 8: Inter-occurrence interval distribution for *demux*

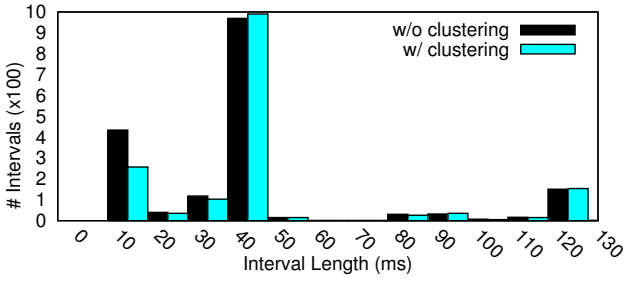

Figure 9: Inter-occurrence interval distribution of *avdec_h264*

ement, e.g. $\langle A1, B1, C0\rangle$, appears in 14% (1/7) of sequences of D_{neg}. As α reaches 14%, ConSGapMiner starts considering sequences like $\langle B1, C0\rangle$ as emerging ones: being a subsequence of $\langle A0, B1, C0\rangle$, $\langle B1, C0\rangle$ has a high frequency in D_{pos}, and being a subsequence of $\langle A1, B1, C0\rangle$, $\langle B1, C0\rangle$ has a frequency of 14% in D_{neg}. At first, with $\alpha = 14\%$ ("target+noise" region), SATM returns *target* with up to 5 other sequences of the form $\langle X, B1, C0\rangle$ with X being some sequence of trace events that usually precede the execution of *intruder*. Then, for $\alpha > 14\%$ ("no target" region), subsequences of the form $\langle B1, C0\rangle$ become emerging, and *target* is no longer returned as it is not minimal.

Note that mining *maximal* emerging patterns would not solve this problem for real use cases. Due to the highly similar nature of the sequences constituting D_{pos} and D_{neg}, maximal emerging sequences can be very long, with hundreds or thousands of elements, with *target* buried indistinguishably somewhere in them.

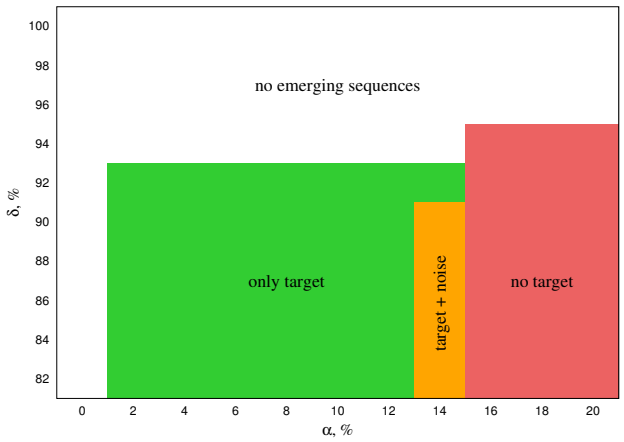

Figure 10: Heat map showing the output of ConSGapMiner algorithm for maximum gap $g = 1$

We evaluate the impact of the gap parameter g in Table 1. These results show the minimum values of δ and α for which *target* is found, which corresponds to the coordinates of the top-left corner of the "only target" region in Figure 10. As g increases, δ can be chosen closer to 100%. Indeed, by increasing g, ConSGapMiner tolerates a larger gap when searching for *target* in D_{pos}, which makes SATM more resilient to system preemptions polluting the trace, at the expense of the running time of ConSGapMiner algorithm. Note that a high value of δ increases the analyst's confidence that discovered emerging sequences are correlated with the bug, since they are shown to characterize a larger fraction of D_{pos}.

g	$sup_{D_{pos}}(\langle A0, B1, C0\rangle, g)$	$sup_{D_{neg}}(\langle A0, B1, C0\rangle, g)$
0	85.3 %	1.5 %
1	92.2 %	1.7 %
2	93.6 %	1.7 %
3	93.6 %	1.7 %

Table 1: Impact of the value of the maximum gap g on the support of the sequential pattern $target = \langle A0, B1, C0\rangle$.

This experiment illustrates the process of parameterizing ConSGapMiner in SATM. The user is encouraged to start with extreme values of δ and α (100% and 0% respectively), and then gradually adjust them to obtain relevant sequences. The results demonstrate that SATM can identify gap constrained sequences which characterize the cause of the temporal bug without requiring complex parameter tuning.

6.2 Industrial use case

This use case is a real QoS problem that occurred at STMicroelectronics. The target application is called *ts_record*, and is designed for set-top boxes equipped with STiH416 MPSoC to record a video stream received from the network onto secondary storage such as an external USB hard drive. The reported defect is that there exist parts of the recorded high-definition video that have poor quality due to dropped frames. *ts_record* is modeled with a single actor that reads data from network buffers, and transfers it to the secondary storage.

SATM discovered the period of 10ms of the *ts_record* actor. There were, however, numerous time intervals of several hundreds of milliseconds between *ts_record* invocations. While mining emerging sequences in these parts of the trace, SATM was able to return the following emerging sequence for the parameters as extreme as $\delta = 94\%$ $\alpha = 3\%$ $gap = 2$: $\langle$ *Interrupt182(GICehci_hcd:usb3), __switch_to-45(usb-storage)-0, __switch_to-0-45(usb-storage), Interrupt182(GICehci_hcd:usb3), __switch_to-45(usb-storage)-0*$\rangle$.

A graphical representation of one part of the trace where the emerging sequence occurs is presented in Fig. 11 (a screenshot of STMicroelectronics KPTrace Viewer). This sequence consists of events implicated in the transfer of data to the external USB hard drive.

The found emerging sequence allowed the developers at STMicroelectronics to recognize a somewhat notorious behavior of the Linux page cache: the data is not written to the secondary storage immediately after calling *write* system call, but is put into the part of main memory called page cache; it is then the responsibility of the *pdflush* kernel thread (*1055(flush-8:0)* in Fig. 11), invoked by default by the operating system every 5 seconds, to initiate the writing of the data stored in page cache to the secondary storage. The peculiarity of *pdflush* operation is the complete blocking of page cache, so that *ts_record* can't transfer the newly received data from network buffers to page cache while *pdflush* is active. The amount of space required to store 5 seconds of high-definition video can easily exceed the size of network buffers, therefore we lose some network packets due to the network buffers overflow, hence, some video frames are dropped in the recorded stream. A possible fix of this problem is to reduce the *pdflush* invocation interval [3].

When looking at our emerging sequence that allowed to understand the problem, *pdflush* itself is a rare event and was not captured in the emerging sequence. However, *pdflush* triggers numerous calls to *interrupt182(GICehci_hcd:usb3)* which further lead to the context switches reported in the emerging sequence. The developers were thus able to use our results to isolate *pdflush* as the source of the problem.

7. RELATED WORK

Entrialgo et al. [11] approached the problem of temporal debugging of real-time systems by analyzing the execution

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

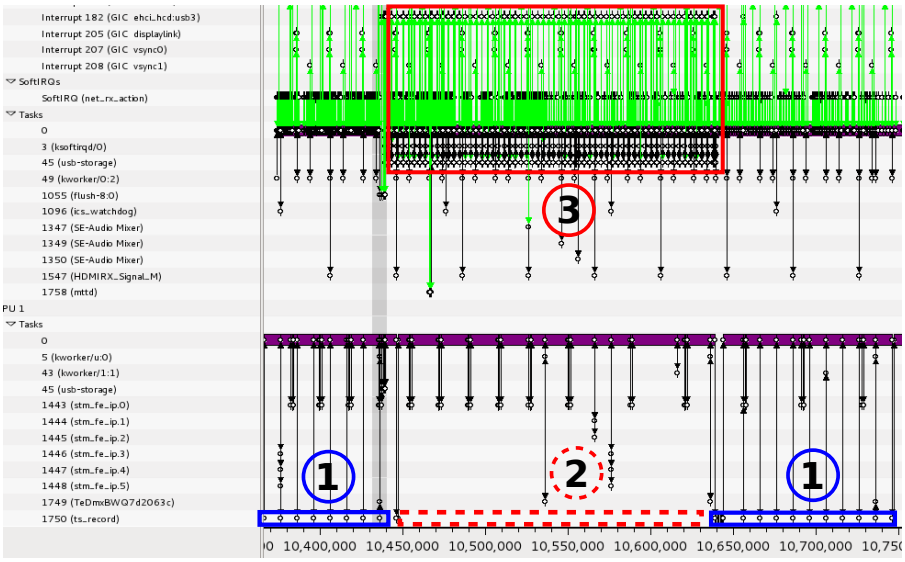

Figure 11: Visualization of *ts_record* execution trace: ① part of the trace where the period is respected; ② part of the trace where the period is broken; ③ emerging sequence.

time of application's tasks modeled with stochastic techniques. They assumed that non-respect of a task's deadline at runtime is caused solely by the incorrect model of this task's execution time. We, however, consider deadline misses as symptoms, and provide methods to discover the cause of application's temporal bug represented by some external to application system activity. Therefore, we rely on the application model and its scheduling on the target platform to always respect the tasks deadlines at runtime if no other system processes are running in parallel.

Albertsson [1] addresses the difficulty of debugging temporal bugs in multimedia applications running in general purpose environment by complete system simulation. The proposed gdb-like debugger running on simulated target platform is capable to pinpoint a missed deadline if the user knows the expected timing characteristics of all the application routines. In contrast to simulator-based debugging, we benefit from the low-intrusive trace recording solutions available in modern embedded systems. Hence, no simulation is needed, and the recorded trace can be analyzed post-mortem, i.e. after the application has finished its execution.

The recent work of Yu et al. [20] presents an approach to debug application performance problems which is conceptually similar to ours. They introduce the notion of *cost propagation* in systems consisting of interacting components, akin to our delay propagation, in order to discover the component which is responsible for the observed performance bug. Their approach mines a number of execution traces capturing both "good" and "bad" application performances, to find event patterns that are inherent to the "bad" traces and that originate from the set of predefined "suspicious" components. In contrast to [20], our approach makes no assumptions on the location of the performance bug, and does not require collection of a call stack attached to each event in the trace. Moreover, we target the embedded software by including its specificities (periodic execution, real-time deadlines) to the core of our approach.

The work of López Cueva et al. [15] is the closest to the spirit of our approach as it also applies data mining algorithms on execution traces for temporal debugging of embedded multimedia applications. Their PerMiner algorithm allows to mine all the sets of events that occur together in the execution trace in a periodic manner, called periodic patterns. At the same time, PerMiner finds time intervals where the period of the chosen pattern is not respected. It then feeds this information to the analysis tool called CompetitorsFinder, that is capable of mining a competitive periodic pattern which is present in the trace only during the specified time intervals. The problem of PerMiner is its generality, as it mines all the periodic patterns in the trace expecting the user to manually analyze them, and lack of precision in pattern's period computation. With PerMiner, one pattern can have many co-prime periods, which makes no sense with respect to the multimedia application real-time scheduling. At the same time, CompetitorsFinder is capable of mining only periodic competitor patterns.

Our temporal debugging approach for streaming embedded applications closely corresponds to the generic debugging methodology for parallel dataflow applications proposed in [6] (non-italics text describes the corresponding steps of our approach):

1. *Identify the portions of the program whose behavior will be monitored.* As explained in Section 4, we monitor actor functions that read input.
2. *Specify the expected execution behavior of the set of nodes that will be monitored.* We expect chosen functions to be invoked in a strict periodic manner.
3. *Determine where the actual and expected events first diverge.* We determine the first actor that occasionally misses its deadline and look for the parts of the trace where its periodic behavior was not respected.
4. *Define corrective action.* We look for the system activity that is related to the temporal bug, and leave the choice of corrective action to the application developer.

8. CONCLUSIONS AND FUTURE WORK

The complexity of modern multiprocessor SoC coupled with the application performance requirements make temporal debugging a major issue for embedded software developers. Temporal bugs impacting application's QoS are hard to debug, firstly, because of the lack of adequate tools, and secondly, because such bugs have a tendency to show up in the last stages of application development cycle where time is running short. In this paper, we propose a novel approach to help application developers detect possible causes of temporal bugs in embedded streaming applications in an automatic manner. This approach is based on data mining techniques applied to execution traces. The experiments have shown that our approach is robust and allowed to solve a real QoS problem at STMicroelectronics.

This work opens many interesting research directions. One of them is the usage of patterns based on partial orders instead of the strict order enforced in this paper [16]. This would allow to capture more complex patterns of system activity. Another direction would be to take into account the execution time of actors, so that the user is able to detect abnormal timing behavior of actors which does not necessarily result in breaks in periodicity. It will be interesting then to apply our enhanced temporal debugging approach on other real streaming applications.

9. REFERENCES

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

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

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

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

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

[6] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, 3(1):75–83, 1995.

[7] W.-T. Chang, S. Ha, and E. A. Lee. Heterogeneous simulation – mixing discrete-event models with dataflow. *Journal of VLSI signal processing systems for signal, image and video technology*, 15(1-2):127–144, 1997.

[8] M. Cooper, J. Foote, A. Girgensohn, and L. Wilcox. Temporal event clustering for digital photo collections. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 1(3):269–288, 2005.

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

[10] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. pages 43–52, 1999.

[11] J. Entrialgo, J. García, J. L. Díaz, and D. F. García. Tools and stochastic metrics for debugging temporal behaviour of real-time systems. *J. UCS*, 15(8):1563–1588, 2009.

[12] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1517–1530, 2009.

[13] X. Ji, J. Bailey, and G. Dong. Mining minimal distinguishing subsequence patterns with gap constraints. *Knowledge and Information Systems*, 11(3):259–286, 2007.

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

[15] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana. Debugging embedded multimedia application traces through periodic pattern mining. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 13–22. ACM, 2012.

[16] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

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

[18] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan. Dataflow model of computation. In *Physical Layer Multi-Core Prototyping*, pages 53–75. Springer, 2013.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[20] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.