HAL
open science

Distance-Based Trace Diagnosis for Multimedia Applications: Help Me TED!

Christiane Kamdem Kengne, Noha Ibrahim, Marie-Christine Rousset, Maurice Tchuente

To cite this version:

Christiane Kamdem Kengne, Noha Ibrahim, Marie-Christine Rousset, Maurice Tchuente. Distance-Based Trace Diagnosis for Multimedia Applications: Help Me TED!. [Research Report] RR-LIG-045, LIG. 2013. hal-00923547v2

HAL Id: hal-00923547

https://inria.hal.science/hal-00923547v2

Submitted on 25 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distance-based Trace Diagnosis for Multimedia Applications: Help me TED!

Christiane Kamdem Kengne*†, Noha Ibrahim*
*University of Grenoble
LIG
681 rue de la passerelle
38400 Saint Martin d'Hères
France
{surname.name}@imag.fr

Marie-Christine Rousset*, Maurice Tchuente†
†University of Yaounde I
LIRIMA, Equipe IDASCO
BP 812 Yaoundé, Cameroun
UMI 209 UMMISCO
BP 337 Yaoundé, Cameroun
Maurice.Tchuente@ens-lyon.fr

***Abstract*—Execution traces have become essential resources that many developers analyze to debug their applications. Ideally, a developer wants to quickly detect whether there are anomalies on his application or not. However, in practice, size of multimedia applications trace can reach gigabytes, which makes their exploitation very complex. Usually, developers use visualization tools before stating a hypothesis. In this paper, we argue that this solution is not satisfactory and propose to automatically provide a diagnosis by comparing execution traces. We use distance based models and conduct a user case to show how TED, our automatic trace diagnosis tool, provides semantic added-value information to the developer. Performance evaluation over real world data shows that our approach is scalable.**

***Index Terms*—Execution traces; Diagnosis; Audio/Video decoding; Multimedia applications.**

I. Introduction

With the proliferation of embedded systems (from home boxes to tablets and smartphones) providing an everywhere access to multimedia contents, the development of multimedia applications is an area of high competition in which every second lost by a developer to debug the application amounts a financial loss for companies.

The analysis of execution traces, that are sequences of timestamped events, is at the core of the optimization and debugging of applications. When the developer has a reference trace (which can be produced by a simulator), a technique for detecting possible anomalies within an execution trace is to compare it with the reference trace using a suitable distance [1]. However, although there is an abundant literature about distances on sequences ([2]–[4]), very few distances take into account the temporal aspect that is crucial in execution traces. More generally, designing an appropriate distance for a meaningful comparison between multimedia execution traces is a difficult task. Indeed, it requires to capture and combine within a single numerical function, several aspects that are specific to multimedia execution traces. Whatever the quality of a distance for suggesting the existence of a bug in an execution trace, based on its numerical comparison with a reference trace, the results of the distance calculation are inherently difficult to interpret by human developers, in particular for finding the actual cause of the bug.

In this paper, we propose to replace a black-box approach encapsulated in a single complex distance by a glass-box approach based on a fine-grained analysis of problems that are likely to occur in multimedia applications. The idea is that anomalies in multimedia applications usually have visible effects such as desynchronization of sound with the picture or subtitles, the interruption of a video streaming or the loss of some frames (a frame being an image rendered during a known time interval).

We make the following contributions:

1) We have identified a family of anomalies likely to occur in multimedia applications and that are visually perceptible when a user is watching a video.
2) For each type of anomaly, we have designed a specific distance which measures appropriately the amplitude of the corresponding anomaly.
3) Based on these distances, we have designed a diagnosis tool able to detect degraded execution traces and to identify the causes of such a degraded behaviour.

The rest of the paper is organized as follows: Section II provides some background and states the problem that we consider in this paper. In Section III we present the general approach that we propose to solve this problem. In Section IV, we describe our TED tool and illustrate it on a use case. Section V summarizes experiments conducted using TED. Finally, Section VI briefly surveys related work, and concludes with some perspectives.

II. Background and Problem Statement

In this section, we explain how execution traces are obtained and described as timestamped sequences. We also introduce three well-known types of anomalies occurring in video streaming. Finally, we state the trace diagnosis problem.

A. Execution Traces Generation and Description

Embedded systems directly integrate hardware tracing support to collect events generated by applications or perform a post-mortem analysis of their execution. These techniques minimize intrusiveness, i.e tracing an application has a minimal impact on its behaviour, allowing complex

interactions to be shown in real-time applications such as video decoding. In some cases, software tracing solutions are provided by the operating system. For instance, on an ST40 core [5], applications run on a Linux distribution for STMicroelectronics products. This operating system provides a tracing tool based on KProbes [6], which registers system and application events: interrupts, context switches, function calls, system calls, etc. In general there are toolkits for managing multimedia data, from source acquisition to treatment and diffusion. The source can be a network stream, a webcam or a file on the hard disk. The treatment concerns for instance audio/video effects and encoding. The architecture of such toolkits are generally modular, pipeline-based and composed of a *gstreamer* [7] core and different plugins. The pipeline connects a number of elements and each element is provided by a plug-in. To play a stream containing both video and audio data for instance, one thread is used for each output. In this case, the pipeline has essential components as *audio decoder, video decoder, demuxer* or *filesrc* [7].

Based on our previous work [8], we formalize the execution traces that are generated as sequences of timestamped events as depicted in Fig. 1, where, for instance, 1965720232 is a timestamp of the event $ffmpeg : gst_ffmpegdec_chain :' Received$.

```
1965720232 ffmpeg:gst_ffmpegdec_chain:'Received
1965979119 ffmpeg:gst_ffmpegdec_frame:'data
1966226423 ffmpeg:gst_ffmpegdec_do_qos:'QOS
1966476796 ffmpeg:gst_ffmpegdec_video_frame:'stored
1966705008 ffmpeg:gst_ffmpegdec_release_buffer:'default
1966864741 ffmpeg:gst_ffmpegdec_get_buffer:'getting
1966987954 ffmpeg:gst_ffmpegdec_get_buffer:'dimension
1967152363 ffmpeg:gst_ffmpegdec_get_buffer:'direct
1967277565 ffmpeg:gst_ffmpegdec_get_buffer:'linsize
1967389606 ffmpeg:gst_ffmpegdec_get_buffer:'data
1969901891 ffmpeg:gst_ffmpegdec_video_frame:'after
1970029153 ffmpeg:gst_ffmpegdec_video_frame:'pts
1970202702 ffmpeg:gst_ffmpegdec_video_frame:'picture
1970441718 ffmpeg:gst_ffmpegdec_video_frame:'picture
1970675868 ffmpeg:gst_ffmpegdec_video_frame:'picture
1970878638 ffmpeg:gst_ffmpegdec_video_frame:'picture
1971219806 ffmpeg:gst_ffmpegdec_video_frame:'picture
1971466389 ffmpeg:gst_ffmpegdec_video_frame:'picture
1971677487 ffmpeg:gst_ffmpegdec_video_frame:'repeat_pict
1971887683 ffmpeg:gst_ffmpegdec_video_frame:'interlaced
1972094822 ffmpeg:check_keyframe:'current
1972329704 ffmpeg:get_output_buffer:'get
1972557530 ffmpeg:get_output_buffer:'clip
1972786880 ffmpeg:alloc_output_buffer:'alloc
1972995748 ffmpeg:alloc_output_buffer:'calling
```

Fig. 1. An execution trace

More formally, let Σ be a set of events. A *timestamped event* is a pair (t, e) where $t \in \mathbb{N}$ is a timestamp and e is an event. A *trace* is a sequence of timestamped events. The length of a trace T, denoted $|T|$, is the number of its timestamped events.

B. Audio/Video decoding Anomalies Description

While streaming a video, some common anomalies can occur. These anomalies are well known in the community of A/V developers ([9], [10]) and almost always have visual and sound effects on the video streaming. They can even be simulated using existing tools that are able to inject those perturbations.

We have chosen to detect three of the common errors that a developer encounters in his video players:

P_1**: Audio/video/subtitle desynchronization anomaly**: This anomaly reflects a desynchronization in time between audio, video or subtitles. The audio may be slower than the video or the subtitle may not appear at the right moment.

P_2**: Player crash anomaly**: The player stops abruptly at a random execution time, without any reason.

P_3**: Slow streaming anomaly**: Visually, video is very slow. In this case the audio/video/subtitles are synchronized but take much more time than in a normal execution.

C. Trace Diagnosis Problem Statement

The general trace diagnosis problem can be decomposed into two sub-problems.

1) Detect whether an execution trace presents some anomalies reflecting an abnormal behaviour of the application under supervision, and if this is the case
2) Identify the cause or at least the type(s) of these anomalies.

The two sub-problems are difficult to solve in general, i.e. without exploiting some additional knowledge or without restricting their scope in order to exploit some domain-specific characteristics.

Our approach to address the first sub-problem is to exploit error-free *reference* traces that can be obtained by a simulator, and to compare them with real execution traces using suitable distances. Detecting whether a real trace execution is abnormal consists in a distance-based comparison with the reference trace obtained by the simulator ran on the same video.

Addressing the second sub-problem is crucial for trace debugging since the developers need to understand what is going wrong in order to identify the anomalies revealed by trace comparison. Our approach is to focus on the identification of pre-established types of domain-specific anomalies, namely those mentioned in Section II-B and referred to as P_1, P_2 and P_3 respectively.

The trace diagnosis problem that we consider in this paper can then be stated as follows:

Given an execution trace T and a reference trace T_r, how to automatically detect whether T contains anomalies of type P_1, P_2 or P_3, using a distance-based comparison with T_r.

III. Distance-based diagnosis

In this section, we explain our general approach for solving the trace diagnosis problem stated above, using appropriate distances.

A distance d between two objects is a numerical measure of how far apart these objects are [11]. There exist many distance definitions in the literature. For every three objects T_1, T_2 and T_3, the following relations must hold:

$$\begin{cases} d(T_1,T_2) \geq 0 \\ d(T_1,T_2) = 0 \quad only\ if \quad T_1 = T_2 \\ d(T_1,T_2) = d(T_2,T_1) \\ d(T_1,T_2) + d(T_2,T_3) \geq d(T_1,T_3) \end{cases}$$

Instead of defining a single distance as a black-box to detect various anomalies, our glass-box approach defines multiple distances that are appropriate to the types of anomalies we want to detect.

The procedure we follow to define our distances is the following. First, we decode a movie video with $gstreamer$ and obtain a reference trace. Then, we inject in the streaming, perturbations corresponding to the three types of anomalies and we obtain for each anomaly the corresponding abnormal execution traces. Finally, for each type of anomaly, we manually analyze the reference trace and the execution trace, and extract the differences that are relevant for each distance.

Let us now present our three distances. The first distance is the *occurrence distance*, suitable for detecting an anomaly of type P_1 when applied to an execution trace and the corresponding reference trace. The second distance is the *dropping distance*, appropriate to identify anomalies of type P_2. Finally, we introduce the *temporal distance* designed to detect anomalies of type P_3. For each distance, we give a formal definition and an algorithm for its computation.

A. Occurrence distance

For P_1 anomaly, when examining the traces, one can detect different numbers of occurrences of some events in the simulated trace and the abnormal one.

We first define the *occurrence ratio* of an event in two traces.

Definition 1: Let T_1 and T_2 be two execution traces. Let $nb_occur(e,T)$ be the number of occurrences of event e in trace T. The *occurrence ratio* of an event e in the two traces T_1 and T_2 is defined as follows:

$$occ_ratio(e,T_1,T_2) = \frac{Min\{nb_occ(e,T_1), nb_occ(e,T_2)\}}{Max\{nb_occ(e,T_1), nb_occ(e,T_2)\}}$$

Note that e should appear in traces. A value of $occ_ratio(T1,T2)$ close to zero, means that event e occurs in one of the two traces much more frequently than in the other one. Such a situation is related to an anomaly P_1 because a desynchronization in time between audio, videao and/or subtitles induces many abnormal events.

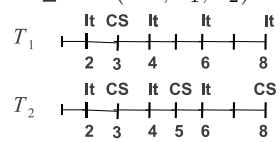
That is why we define the *occurrence distance* between two traces as the number of events that have an occurrence ratio less than or equal to a given threshold. This distance is appropriate to retrieve P_1, A/V/S desync. anomaly, (see section II-B) because it measures the number of events that differentiate T_1 from T_2. The formal definition of this distance, thereafter denoted d_1 is the following:

Definition 2: Let T_1 and T_2 be two execution traces. The ***occurrence distance*** between T_1 and T_2 is:

$$d_1(T_1,T_2) = |\{e \mid occ_ratio(e,T_1,T_2) \leq \theta\}|$$

where θ is a given threshold.

Example 1: consider the traces T_1 and T_2 below, and let $\theta = 0.5$. $d_1(T_1,T_2) = 1$ with $occ_ratio(It,T_1,T_2) = 3/4 = 0.75$, $occ_ratio(CS,T_1,T_2) = 1/3 = 0.33$.



B. Dropping distance

For P_2 anomaly, when comparing the simulated and abnormal traces, we found that some events seem to appear only in one trace and not in the other one.

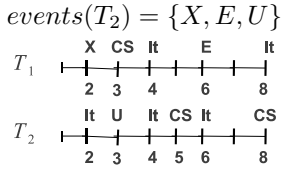
The corresponding *dropping distance* refers to the number of distinct events that belong only to one trace.

Definition 3: Let *events(T)* be the set of distinct events in T. The ***dropping distance*** between T_1 and T_2 is the size of the symmetric difference between *event(T_1)* and *event(T_2)*.

$$d_2(T_1,T_2) = |events(T_1) \triangle events(T_2)|$$

This distance is appropriate to retrieve P_2, i.e. Player crash anomaly (see section II-B).

Example 2: for traces T_1 and T_2 below, $events(T_1) = \{X, CS, It, E\}$, $events(T_2) = \{CS, It, U\}$; $events(T_1) \triangle events(T_2) = \{X, E, U\}$ and $d_2(T_1,T_2) = 3$.



C. Temporal distance

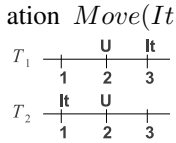
For P_3 anomaly, the duration and the order of some events differ in the two traces. In the abnormal trace, some events durations are much longer than in the simulated trace.

The temporal distance that we propose is an adaptation of the distance model of *Mannila et. al* [12] which is an edit-distance taking into account temporal aspects. It uses three basic operations:

- *Ins(e,t)* that inserts an event e at time t
- *Del(e,t)* that deletes an event e at time t
- *Move(e,t,t')* that moves an event e from t to t'.

Note that the Move operation is order-preserving. This means that if $t(e) = t < t' = t(e')$ and e, e' are not deleted than one cannot have $Move(e,t,t_1)$ and $Move(e',t',t'_1)$ for $t_1 > t'_1$.

Example 3: For instance, in the example below, the operation $Move(It,1)$ that transforms T_1 into T_2 is forbidden.



A cost $c(o)$ is associated with each operation o and $c(Ins(e,t)) = c(Del(e,t)) = w(e)$ where $w(e)$ is a weight

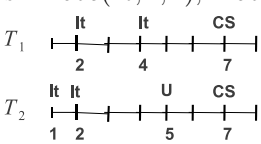
associated with event e.
$c(Move(e,t,t')) = V|t'-t|$ where V is a constant such that $V \leq 2.w(e)$. Without this condition, it would always be better to do a deletion and an insertion of an event e, instead of moving e from t to t'.
The cost of a sequence of operations can then be deduced. Let $O = o_1 \dots o_k$ be a sequence consisting of k transformations. The cost of O is:

$$c(O) = \sum_{i=1}^{k} c(o_i)$$

The distance $d(T_1, T_2)$ is defined as the cost of the cheapest sequence of operations that transform T_1 into T_2. Thus, if Θ is the set of operation sequences that transform T_1 into T_2, then:

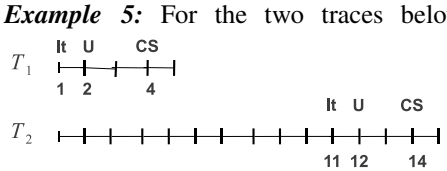
$$d(T_1, T_2) = \underset{O \in \Theta}{Min} c(O)$$

Example 4: For traces T_1 and T_2 below, the cheapest order-preserving sequence of operations that transforms T_1 into T_2 is $Move(It,2,1), Move(It,4,2), Ins(U,5)$= $3V + w(U)$.



The beginning timestamp in two traces is not always the same. Consequently, results obtained with this method are not satisfactory. We explain this problem in example 4.

Example 5: For the two traces below, $d(T_1, T_2) \neq 0$.



This is not satisfactory because T_1 and T_2 have exactly the same events, and the same time intervals between events. Clearly, such traces should be considered as similar. Therefore, we adapt the Mannila distance model in order to have $d_3(T_1, T_2) = 0$ when T_2 is obtained from T_1 by a time shift.
Definition 4: Let $T_1 = (e_1, \dots, e_n)$ and $T_2 = (f_1, \dots, f_m)$ two execution traces, and let $r(i,j)$ denote the minimum cost of the operations needed to transform the first i events of T_1 into the first j events of T_2. The ***temporal distance*** between T_1 and T_2 is:

$$d_3(T_1, T_2) = r(n, m)$$

where $r(i,j)$ is computed according to the following dynamic programming algorithm:

$$\begin{aligned} r(0,0) &= 0 \\ r(i,0) &= r(i-1,0) + w(e_i) \\ r(0,j) &= r(0,j-1) + w(f_j) \\ r(i,j) &= min\ \{\ r(i-1,j) + w(e_i), \\ & \quad r(i,j-1) + w(f_j), r(i-1,j-1) + cost(i,j)\} \end{aligned}$$

$w(e_i)$ is the cost of deleting event e_i at position i. $w(f_j)$ the cost of inserting event f_j at position j and

$$cost(i,j) = \begin{cases} w(e_i)+w(f_j) & if\ e_i \neq f_j \\ V.|(t_i - t_{i-1}) - (t'_j - t'_{j-1})| & if\ e_i = f_j\ and\ i = j \\ V.|t_i - t'_j| & if\ e_i = f_j\ and\ i \neq j \end{cases}$$

The application of this *Temporal distance* in the traces of example 2 gives $d_3(T_1, T_2) = 0$. Hence, this distance is appropriate to retrieve P_3, i.e. slow streaming anomaly (see section II-B).

D. Distance computation algorithms

For each distance defined above the output is a value in $\mathbb{R}^+$. In order to better interpret the results, it is important to normalize the output. We use a non-linear transformation g, in order to normalize the distances:

$$\begin{aligned} g \ : \ \mathbb{R}^+ &\rightarrow [0,1] \\ d &\mapsto d/(1+d) = g(d) \end{aligned}$$

The computation of *occurrence distance* (Subsection III-A) and *dropping distance* (Subsection III-B) are done in linear time complexity since a simple scan of traces is necessary. With the dynamic programming algorithm presented above, the computation of *temporal distance* (Subsection III-C), has a quadratic complexity $O(m \times n)$, where m and n are the lengths of the two traces. [13] proposed some improvements with a $O(np)$ time complexity, where $p = D/2 - (n-m)/2$ with D being the length of a shortest edit script (consisting of insertions and deletions) between the two sequences to compare; we suppose $n \geq m$.

It is important to emphasize that each of these distances can be computed at different levels of granularity. Each trace can be decomposed into blocks of events related to a specific plugin of the pipeline (Subsection II-A). When comparing sequences of events by plugin, we can detect in which plugin the anomaly that takes place.

IV. The TED tool illustrated on a use case

In this section, we describe TED, our TracE Diagnosis tool (Fig. 2), and illustrate its functioning on two use cases.

A. TED Architecture

TED handles two main phases. The *Preprocessing and trace generation* phase takes as input - a reference trace and a source file to generate an execution trace T via the *multimedia Toolkit*. The traces are preprocessed. This step is very important for a successful outcome of the analysis as a non cleansed and non normalized data can lead to spurious and meaningless results [2]. A parsed trace (c.f. figure 3) T_p (respectively T_r) is obtained from T (respectively reference trace), by removing some redundant informations or by modifying others. If needed, we can abstract traces via the

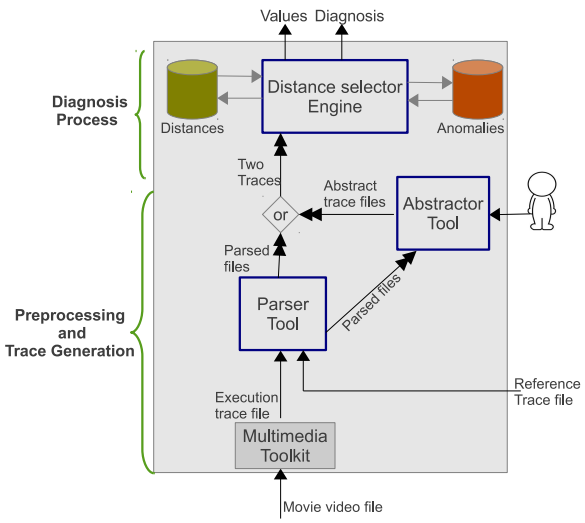


Fig. 2. TED Architecture

abstractor tool. We further explain in section V the utility of such abstraction and how our distance-based algorithms can be adapted to such traces.

The *Diagnosis process*, is the second and core phase of TED. The *distance selector engine* chooses an appropriate distance from the *Distances* database and applies it to the anomaly it needs to detect. For instance, if we want to detect a desynchronization anomaly, the *distance selector engine* applies the occurrence distance on T^p and the reference trace T^r.

B. Use cases

We consider the following scenario. A user is watching a video and (a) the video streaming becomes very slow or, (b) the sound is desynchronized with images.

In the *Preprocessing and Trace Generation* phase, we decode the movie with *gstreamer* to obtain the reference trace T_r. We use a gstreamer element *identity* [7], with property *sleep-time*, to obtain a A/V/S desync. anomaly (scenario b). The abnormal trace obtained is T. We generate another abnormal trace, with a slow streaming anomaly (scenario a) by a stress of CPU and memory in the system. T_r and T have the format of Fig. 3(a). In order to reduce the size of the dataset for easier processing by temporal distance, we keep only four events columns, which correspond to *timestamps, Debug level, function* and the first argument of the *message*. As a result, the dataset was reduced to $26,5\%$ of its original size (Fig. 3(b)).

In the *Diagnosis process* phase, the developer uses TED as follow:

- The developer has an idea of the anomaly and just want to verify if his hypothesis is true or not. He selects the distance to apply and TED gives the diagnosis. In Fig. 4(a), *temporal distance* is used (scenario a). The developer suspects a slow streaming anomaly (P3). TED detects the anomaly and returns the value of temporal distances between the two traces per plugins. TED points out the *audioresample* plugin to be the one with the most dissimilar events between the two traces.

- The developer has no idea of what is happening and would like to find if there exists an anomaly in T. He selects the choice *find anomaly*, and TED applies successively all the distances, and stops when one of them gives a non-zero value (Fig. 4(b)). In scenario b, dropping and occurrences distances have been tested and a A/V/S desync. anomaly was detected.

- The developer wants to find all potential anomalies in T (choice *all tests*). Indeed, it is possible to have simultaneously a A/V/S desync. and a player crash anomaly.

By using TED, a developer analyzing an execution trace is notified of anomalies, their types and where they appear in the trace (the plugin concerned). TED is a time saver for developers as they can quickly detect anomalies in their execution traces and fix them.

V. EXPERIMENTS

We conducted a set of experiments to demonstrate the quality and efficiency of our proposed execution trace diagnosis tool. First we use standard distance algorithms to compare traces and show the semantic added-value brought by TED. We also show how helpful this automatic tool can be for developers, by an evaluation of TED scalability and precision. Finally, we discuss the importance of trace abstraction and show how to adapt TED to take into account abstract traces.

System configuration: Our prototype system is implemented in Python 3.2. The experiments were run on an Intel Xeon E5-2650 at 2.0GHz with 32 Gigabytes of RAM with Linux.

Data Set: We use traces from two real applications, described below:
Gstreamer application: Gstreamer [7] is a powerful open source multimedia framework for creating streaming applications, used by several corporations as Intel, Nokia, STMicroelectronics and many others. For these experiments we decoded several movies using Gstreamer on a Linux platform, with the $ffmpeg$ plugin for video decoding.
GSTapps application: It is a test video decoding application for STMicroelectronics development boards. This application is widely used by STMicroelectronics developers. The execution trace contains both application events and system-level events. It is generated from a $ST40$ core of the SoC, which is dedicated to application execution and device control.
Table I gives a description of reference traces.

Comparison with standards sequence distances: We used existing implementations of two well known sequence dis-

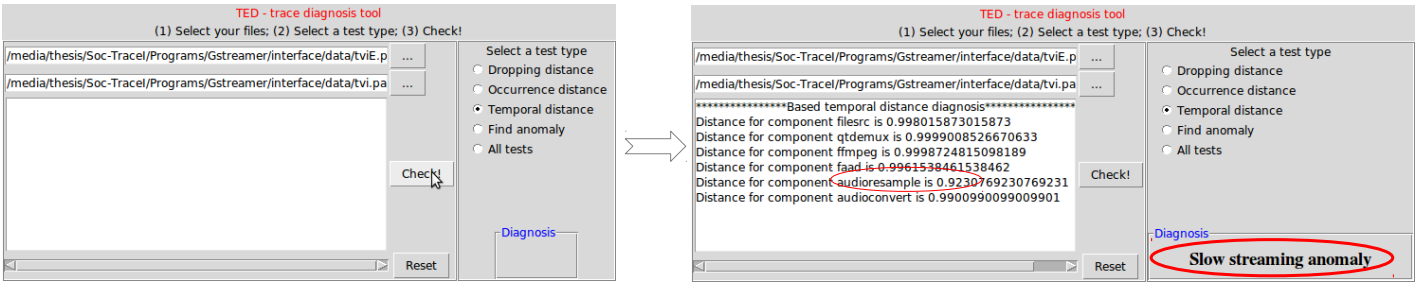
```
1073846790;9002;0x1e6a370;DEBUG;;ffmpeg;gstffmpegdec.c;1747;gst_ffmpegdec_video_frame;ffdec_h2640;'stored opaque
values idx 14'
1073914700;9002;0x1e6a370;DEBUG;;ffmpeg;gstffmpegdec.c;1112;gst_ffmpegdec_release_buffer;ffdec_h2640;'default
release buffer'
1073972663;9002;0x1e6a370;DEBUG;;ffmpeg;gstffmpegdec.c;1005;gst_ffmpegdec_get_buffer;ffdec_h2640;'getting buffer'
1074027123;9002;0x1e6a370;LOG;;ffmpeg;gstffmpegdec.c;1021;gst_ffmpegdec_get_buffer;ffdec_h2640;'dimension 640x360,
coded 640x360'
1074030908;9002;0x1eeb800;LOG;;queue;gstqueue.c;656;apply_buffer;queue0;'last_stop updated to 0:00:00.696598639'
1073583066;9002;0x1eeb850;LOG;;queue;gstqueue.c;656;apply_buffer;queue1;'last_stop updated to 0:00:01.600000000'
1074086213;9002;0x1e6a370;LOG;;ffmpeg;gstffmpegdec.c;1023;gst_ffmpegdec_get_buffer;ffdec_h2640;'direct rendering
disabled, fallback alloc'
1074223050;9002;0x1eeb800;LOG;;queue;gstqueue.c;588;update_time_level;queue0;'sink 0:00:01.578956916, src
0:00:00.696598639'
1074305235;9002;0x1eeb850;LOG;;queue;gstqueue.c;588;update_time_level;queue1;'sink 0:00:01.600000000, src
0:00:00.600000000'
1074381176;9002;0x1e6a370;LOG;;ffmpeg;gstffmpegdec.c;1027;gst_ffmpegdec_get_buffer;ffdec_h2640;'linsize 672 336 336'
1074498987;9002;0x1eeb850;LOG;;qtdemux;qtdemux.c;3377;gst_qtdemux_combine_flows;demuxer;'flow return: ok'
1074551207;9002;0x1e6a370;LOG;;ffmpeg;gstffmpegdec.c;1030;gst_ffmpegdec_get_buffer;ffdec_h2640;'data 0 4178467352
4178537688'
1074560949;9002;0x1eeb800;LOG;;faad;gstfaad.c;756;gst_faad_handle_frame;faad0;'382 bytes consumed, 2048 samples
decoded'
1074602875;9002;0x1eeb850;LOG;;qtdemux;qtdemux.c;3407;gst_qtdemux_combine_flows;demuxer;'combined flow return:
ok'
1074831787;9002;0x1eeb850;LOG;;qtdemux;qtdemux.c;5512;qtdemux_parse_samples;demuxer;'parsing samples for stream
fourcc avc1, pad video_00'
1074905182;9002;0x1eeb850;DEBUG;;qtdemux;qtdemux.c;5523;qtdemux_parse_samples;demuxer;'parsing up to sample 40'
1074972621;9002;0x1eeb850;LOG;;qtdemux;qtdemux.c;5549;qtdemux_parse_samples;demuxer;'sample 40 has size 1858'
```

(a) original trace

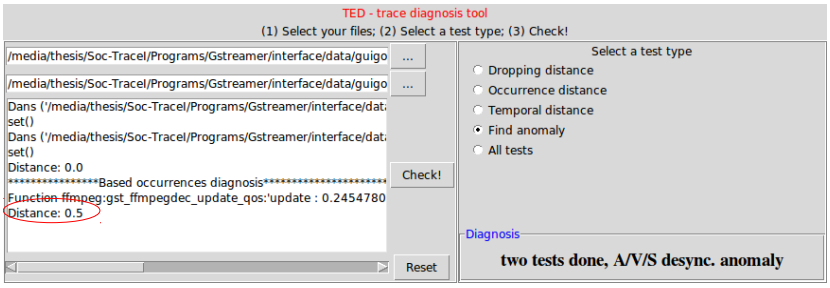
```
1073846790 ffmpeg:gst_ffmpegdec_video_frame :'stored
1073914700 ffmpeg:gst_ffmpegdec_release_buffer:default
1073972663 fmpeg:gst_ffmpegdec_get_buffer:'getting
1074027123 ffmpeg:gst_ffmpegdec_get_buffer:'dimension
1074030908 queue:apply_buffer:'last
1073583066 queue:apply_buffer:'last
1074086213 ffmpeg:gst_ffmpegdec_get_buffer:'direct
1074223050 queue:update_time_level:'sink
1074305235 queue:update_time_level:'sink
1074381176 ffmpeg:gst_ffmpegdec_get_buffer:'linsize
1074498987 qtdemux:gst_qtdemux_combine_flows:'flow
1074551207 ffmpeg:gst_ffmpegdec_get_buffer:data
1074560949 faad:gst_faad_handle_frame:382
1074602875 qtdemux:gst_qtdemux_combine_flows:'combined
1074831787 qtdemux:qtdemux_parse_samples:parsing
1074905182 qtdemux:qtdemux_parse_samples:'parsing
1074972621 qtdemux:qtdemux_parse_samples:'sample
```

(b) parsed trace

Fig. 3. Example of data preparation



(a) execution trace with a *slow streaming anomaly*. The developer selects the distance to apply (scenario a)



(b) TED finds and detects one anomaly: A/V/S desync. anomaly (scenario b)

Fig. 4. TED's help

TABLE I
EXPERIMENTAL DATASET

Video	Duration	Nb. of events	Size
generic	$5s$	$15,110$	$2.9M_o$
pub	$30s$	$74,510$	$14.3M_o$
movie	$3628s$	$12,423,095$	$2457,6M_o$
SDK2	$335s$	$2,382,720$	$73.2M_o$

TABLE II
DTW AND LCS DISTANCES

	DTW	LCS
(T_r, T_1)	509069	28035
(T_r, T_2)	504472	28086
(T_r, T_3)	920600	18377

tances *DTW* [14] and *LCS* [4]. These implementations are given by mlpy [15], a Python module for Machine Learning built. For our experimentations, the events of execution traces were coded as integers, as required by *mlpy*. $LCS(x, y)$ returns the length of the longest common sequence of x and y. We then obtain distance between x and y by $d(x, y) = |x|+|y|-2*LCS(x, y)$. Table II shows the values of distances obtained w.r.t to two execution traces given as input.

T_r is the reference trace of *generic video*; T_1 is obtained by using the gstreamer element *identity* before the video decoding plugin, with property $sleep - time = 30000$. With $sleep - time = 5000$, we obtained T_2 and a visual degradation slighter than those related to T_1, not really perceptible. Naturally, we expect that $d(T_r, T_1) > d(T_r, T_2)$. It is the case with *DTW* distance ($509069 > 504472$), but not with *LCS* distance. T_3 is obtained with property *error-after*.

An error occurs during the video streaming, after a given number N of buffers. $N = 500$. We obtained for instance $dtw(T_r, T_3) = 920600$.
The observation is that T_1, T_2 and T_3 are far from T_r. With standard distance algorithms, we can only compute distance values but we have no idea which type of anomalies are in the traces.

In our proposal, for T_1, TED diagnoses a slow streaming problem. He gives 132090.5 as $d_3(T_r, T_1)$, and 131525 as $d_3(T_r, T_2)$ which confirm our expectation of $d(T_r, T_1) > d(T_r, T_2)$, and the fact that the video execution of T_1 is slower than the one of $T2$. For T_3, TED diagnoses a player crash anomaly in addition to giving a distance value between T_3 and T_r.

Running time and Scalability: Fig. 5 reports the wall clocks of TED for *occurrence* and *dropping* distance, when varying events number of execution traces. Horizontal axis represent the maximum number of events of the two compared traces. In practice, we consider as $\theta = 0.25$, as threshold of occ_ratio. One can notice that, for traces of more than $1G_o$, corresponding to approximatively $4,000,000$ events, TED can give a diagnosis in less than $10s$. For the *pub* video of table II, an output is obtained in $0.12s$. The experiments showed that the proposed methods can scale to real application traces. This makes TED suitable for analysis of real traces.

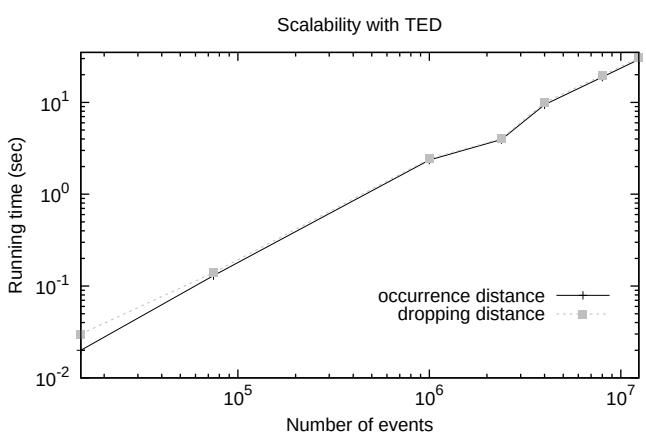


Fig. 5. Running time

Precision: In order to evaluate the accuracy of the diagnosis done by TED, we run TED on a sample of 300 execution traces as shown in Table III. The first observation is that all execution traces initially considered as normal were diagnosed as such by TED. However, the tool gave 14 *false-true* which are execution traces considered by TED as normal but which contain anomalies. Thus, TED has a precision of 95.33%. A reason of this lack of precision can be the value of threshold for *occurrence distance*. We fixed it at $\theta = 0.25$ but we will surely gain to adapt the threshold value to the length of the video decoded. We are currently testing the correlation between the video length and the threshold value.

TABLE III
TED PRECISION

Nb. traces	Initially	With TED
Sample of 300 traces	normal: 130	normal: 144
	abnormal: 170	abnormal: 156

Discussion about abstract traces:
One way to bypass the problem raised by multimedia applications in which raw execution traces are very large (more than a gigabyte for few minutes of video decoding [16], [17]) is to abstract traces. The abstraction process produce more compact traces and facilitate the readability of traces for human programmers. An abstract trace example is given Fig. 7.

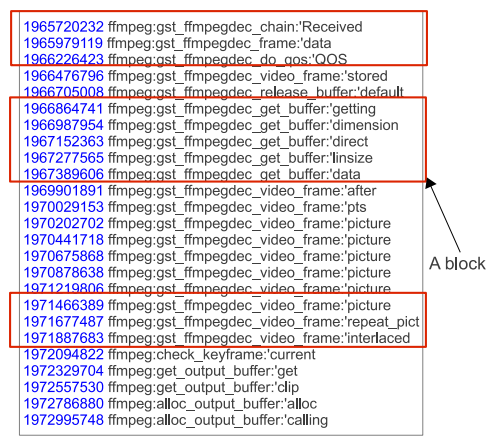


Fig. 6. A block in an execution trace

We define as *timestamped block* a pair (t, B) where B is a block and $t \in \mathbb{N}$, is the timestamp of the first event of B. A *abstracted trace* is a sequence of timestamped blocks. The length of a abstracted trace T denoted $|T|$ is the number of its blocks (c.f. fig. 6). The size of a sequence S, denoted by $\|S\|$, is the total number of events that it contains. For an execution trace T, $|T| = \|T\|$ whereas for an abstracted trace T^a, described by blocks, $|T^a| \neq \|T^a\|$ (except when blocks are singletons of events).
Fig 7 is an example of abstracted trace obtained by *FrameMiner* [8] on *pub* video.
Our approach gains to be generic i.e. applicable to execution traces described at different levels of abstractions: on raw execution traces that are sequences of time-stamped low-level events, as well as on sequences of time-stamped *blocks*, in which (subsequences of) low-levels events have been abstracted into blocks [8] more meaningful to the programmer. In order to apply TED on abstract traces, a first idea would be to consider occurrence of a block as a strict

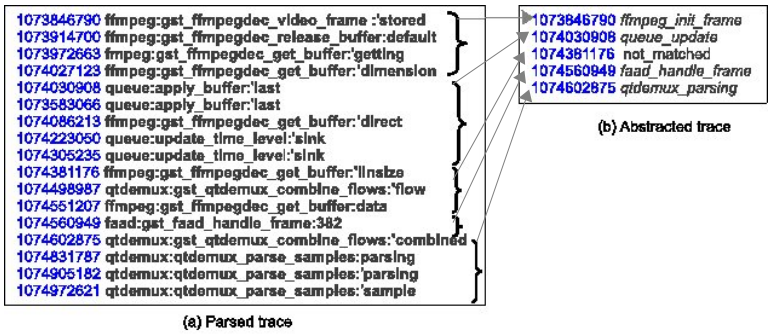


Fig. 7. An abstracted trace obtained with *FrameMiner*

sequence of events and to apply our distances not on events but on blocks. The adaptation of TED to abstracted traces is currently under development.

VI. Conclusion

To analyse traces of finished events, and fix bugs, programmers use several tools such as trace visualizers ([18]–[21]) and techniques such as tracepoints on the execution traces. These techniques need to have an expert to interpret the graphical representation. In contrast, our work based on distances develops a technique which limits the developer intervention.

There is an abundant literature about distances. For distances between sequences, an edit distance model is used in [22] to approximate matching of timed strings; [23], [24] propose to represent each sequence in a suitable form, before computing distance. However, very few distances take into account the temporal aspect. We propose a temporal distance that is adapted for trace comparison. But the most distinguishing point of our approach is that our method is the first, to the best of our knowledge which returns a diagnosis to the user, added to the effective values of distance.

Our approach diagnoses anomalies in an execution trace of multimedia application, by comparison with a reference trace. We use distances as models of comparison and specifically design three distinct distances in order to tackle well-known anomalies of the multimedia domain. We experimentally show the originality of our solution compared to existent distances and show that our proposed approach scales well to real huge application traces. Distances defined in our approach allow to identify a specific problem and give a semantic added-value level to the analysis. Moreover, as all distances, they also provide insights of how far an abnormal trace is from a correct one. We also present a use case on how TED performs the analysis of a trace and conduct some experiments to evaluate TED scalability and accuracy.

We have three research directions. The first direction is to adapt our distances to abstract traces so that our proposal be as generic as possible. The second direction is to enlarge TED to other types of anomalies for instance the image is completely fuzzy, upside down and/or cut in half. The strength of our contribution is that it is easily extensible to other types of anomalies. For each new anomaly, we only need to follow the same methodology as explained in the paper to find the best suitable distance capable of clearly detecting the anomaly. There is no need to do any changes in TED existing architecture. Finally, additional constraints can be introduced such as parallel execution traces and the challenge is to identify, for example, streams of different execution and take them into account for the computation of distances.

Acknowledgment

This work is supported by French FUI project SoCTrace.

References

[1] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection for discrete sequences: A survey,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, no. 5, pp. 823–839, 2012.

[2] F. Mörchen, “Time series knowledge mining,” 2006.

[3] R. Tavenard, L. Amsaleg, and G. Gravier, “Estimation de similarité entre séquences de descripteurs à l’aide de machines à vecteurs supports,” in *Proc. Conf. Base de Données Avancées, Marseille, France*, 2007.

[4] L. Bergroth, H. Hakonen, and T. Raita, “A survey of longest common subsequence algorithms,” in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. IEEE, 2000, pp. 39–48.

[5] Stlinux website. [Online]. Available: http://www.stlinux.com/

[6] R. Krishnakumar, “Kernel korner: kprobes-a kernel debugger,” *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[7] Gstreamer website. [Online]. Available: http://www.gstreamer.net

[8] C. K. Kengne, L. C. Fopa, A. Termier, N. Ibrahim, M.-C. Rousset, T. Washio, and M. Santana, “Efficiently rewriting large multimedia application execution traces with few event sequences,” in *KDD Industrial Track (To appear)*, 2013.

[9] Discussion page: Troubleshooting guide. [Online]. Available: http://www.cccp-project.net/wiki/index.php?title=Troubleshooting_Guide

[10] Faq: Play an audio or video file. [Online]. Available: http://windows.microsoft.com/en-us/windows7/play-an-audio-or-video-file-frequently-asked-questions

[11] T. Pang-Ning, M. Steinbach, and V. Kumar, “Introduction to data mining,” 2006.

[12] H. Mannila and P. Ronkainen, “Similarity of event sequences,” in *Proceedings of the 4th International Workshop on Temporal Representation and Reasoning (TIME ’97)*, ser. TIME ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 136–.

[13] S. Wu, U. Manber, G. Myers, and W. Miller, “An o(np) sequence comparison algorithm,” *Inf. Process. Lett.*, vol. 35, no. 6, pp. 317–323, Sep. 1990.

[14] Y. Sakurai, C. Faloutsos, and M. Yamamuro, “Stream monitoring under the time warping distance,” in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 1046–1055.

[15] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello, “mlpy: Machine learning python,” 2012.

[16] X. gurin, "Approche Efficace de Développement de Logiciel Embarqué pour des Systèmes Multiprocesseurs sur Puce," Ph.D. dissertation, 2010.
[17] C. Prada-Rojas, M. Santana, S. De-Paoli, X. Raynaud *et al.*, "Summarizing embedded execution traces through a compact view," in *Conference on System Software, SoC and Silicon Debug S4D*, 2010.
[18] B. D. O. Stein, "Pajé trace file format," 2003.
[19] J. Roberts, "Tracevis: an execution trace visualization tool," in *In Proc. MoBS 2005*. Citeseer, 2005.
[20] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (vet): an interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User interface conference - Volume 50*, ser. AUIC '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 153–160.
[21] J. Seyster, "Techniques for visualizing software execution," Citeseer, Tech. Rep., 2008.
[22] S. Dobrisek, J. Zibert, N. Pavesic, and F. Mihelic, "An edit-distance model for the approximate matching of timed strings," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, no. 4, pp. 736–741, 2009.
[23] O. Kostakis, P. Papapetrou, and J. Hollmén, "Distance measure for querying sequences of temporal intervals," in *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*. ACM, 2011, p. 40.
[24] O. E. Gundersen, "Toward measuring the similarity of complex event sequences in real-time," in *Case-Based Reasoning Research and Development*. Springer, 2012, pp. 107–121.