Enhancing the Analysis of Large Multimedia Applications Execution Traces with FrameMiner

C. Kamdem K.*†, L. C. Fopa*† N. Ibrahim*, A. Termier*, M.-C. Rousset*
*University of Grenoble
LIG
681 rue de la passerelle
38400 Saint Martin d'Hères
France
{surname.name}@imag.fr

†University of Yaounde I
LIRIMA, Equipe IDASCO
Faculté des Sciences
Département d'Informatique
BP 812 Yaoundé, Cameroun
UMI 209 UMMISCO
BP 337 Yaoundé, Cameroun

T. Washio‡
‡Institute of Scientific and Industrial Research
Osaka University
8-1 Mihogaoka, Ibaraki, Osaka, 567 Japan
washio@ar.sanken.osaka-u.ac.jp

***Abstract*—The analysis of multimedia application traces can reveal important information to enhance program comprehension. However traces can be very large, which hinders their effective exploitation. In this paper, we study the problem of finding a *k-golden* set of blocks that best characterize data. Sequential pattern mining can help to automatically discover the blocks, and we called *k-golden set,* a set of k blocks that maximally covers the trace. These kind of blocks can simplify the exploration of large traces by allowing programmers to see an abstraction instead of low-level events. We propose an approach for mining golden blocks and finding coverage of frames. The experiments carried out on video and audio application decoding show very promising results.**

***Keywords*-Data mining; Trace Analysis; Program Comprehension; Software Engineering;**

I. Introduction

The use of embedded systems like smartphones, tablets and controllers has been expanded in many fields of our everyday life. This situation increases the needs to develop applications for these systems. One of the most used are multimedia applications in which video and audio decoding are the important tasks. A multimedia decoding is the process of rendering images and sounds on a screen, and the result must be of good quality, without interruption between images or any delay between picture and sound. This process deals with computations over *frames*. A frame is an image rendered during a known time interval. An anomaly or an unusual execution in an application decoding video (or audio) can waste a lot of time and a lot of money in industry. Increasingly, the analysis techniques of applications use execution traces, which are chronological sequences of couples made up by a timestamp and an event, to efficiently uncover bugs causing such faulty behaviors ([1]–[3]); however, the challenge in this case is the size of these traces that can easily reach gigabytes for only few minutes of decoding (for instance, the tool Parallel MJPEG [4] can produce a trace file of 7 Gigabytes for less than 5 minutes of video decoding).

To analyse traces of finished events, and fix bugs, programmers use several tools such as trace visualizers ([5,6]) and techniques such as tracepoints on the execution traces. One problem is that *it is difficult to know where and what to look* within a trace in order to detect particular behaviour or observe something interesting, because of the amount of data. Various studies have examined the techniques to reduce the volume of traces ([7,8]) which propose sampling methods. These techniques can obtain a reduced execution trace but not always representative of the entire trace [9]. Pirzadeh et al. introduced in [2] that, the general consensus in the trace analysis community is to emphasise the work towards effective trace abstraction techniques, such as [10]. For reducing the volume of events in a trace to be analysed by the programmer, we propose to abstract series of low-level events as *blocks* that are meaningful to the programmer, and then to further abstract the trace as series of blocks. Fig. 1 illustrate a real trace with frames and blocks. The candidate blocks that can serve for rewriting traces into series of blocks can be provided by the user or automatically discovered by some given measures. In our proposed algorithm *FrameMiner*, the candidate blocks are automatically discovered by mining the frames in the trace file. The candidate blocks are then obtained as the result of a sequence mining algorithm returning the set of consecutive events that occur frequently in the set of frames in a trace file, where the frames can be easily identified because they are delimited by two *start* and *end* events.

The problem is then to rewrite each frame into a short description with a minimum set of blocks. In fact, this problem is an optimization problem to find k blocks that provide the best coverages of a set of frames. For this purpose it is first interesting to see how a single frame can be covered with blocks. That is the reason why we will consider a frame as granularity level, and each block ought to have a meaning in the frame decoding process.

In our proposed algorithm *FrameMiner*, this NP-hard prob-

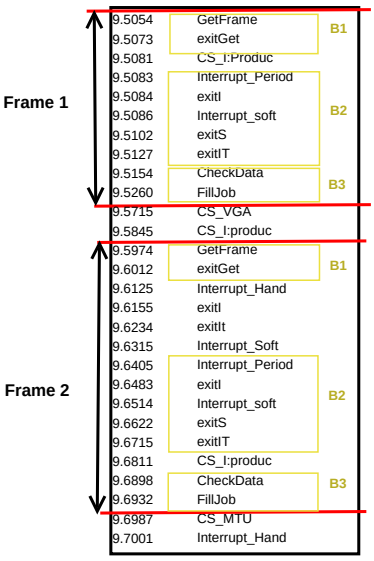


Figure 1. A real trace with the *start event* GETFRAME, the *end event*, FILLJOB and blocks B_1, B_2, B_3

lem is solved using a greedy algorithm to find approximately largest coverages of frames using predefined number, k, of blocks called *golden blocks*. We have conducted preliminary experiments on a set of frames coming from a trace of a SoC multimedia video decoding program and obtained their promising quantitative and qualitative results.

This paper is organized as follows: Section II states the problem and briefly gives some notations and important definitions. In Section III, we present our greedy algorithm at the core of *FrameMiner*. Section IV reports on experiments done on SoC multimedia video decoding program, followed by a discussion. Section VI is an overview of the related work. We end in Section VII by a conclusion and future work.

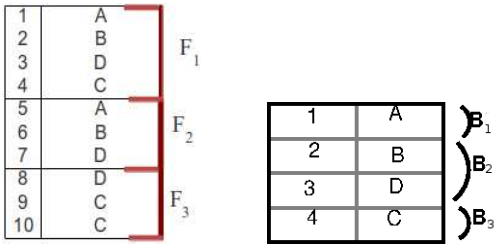
II. PRELIMINARIES AND PROBLEM STATEMENT

In this section we give the notations and definitions necessary to model our problem.

A. Notations

Let Σ be a set of events. A *block* is a non empty sequence of events. A *timestamped event* is a pair (t, e) where $t \in \mathbb{N}$, is a timestamp and e is an event. *Frames* are sequences of timestamped events and a *trace* is a sequence of frames ordered by timestamps. The size of a sequence S, denoted by $\|S\|$, is the total number of events that it contains.

Example: For the trace in Fig. 2, $\|F_1\| = 4$. $B_2 = \langle B, D \rangle$ is a block of two events B and D.



(a) - A trace with 3 frames F_1, F_2, F_3

(b) - The frame F_1 with 3 blocks: $B_1 = \langle A \rangle, B_2 = \langle B, D \rangle, B_3 = \langle C \rangle$

Figure 2. Example of trace, frames and blocks

B. Definitions

The first definition that we introduce is the occurrence time of a block in a frame.

Definition 1: Let $B = \langle e_B^1, \ldots, e_B^v \rangle$ be a block and let $F = \langle (t_1, e_F^1), \ldots, (t_n, e_F^n) \rangle$ be a frame. B occurs in F (denoted $B \sqsubseteq F$) between timestamps i and $i + v$ iff:

$$\forall j \in [i, i+v], \quad e_F^j = e_B^{j-i+1}.$$

i is then called the occurrence time of B in F.

Example: In Fig.2(a), $B_1 = \langle B, D \rangle$ occurs in F_1 between timestamps 2 and 3; it occurs in F_2 between 6 and 7.

As we said before, our granularity level is a frame, so we don't need to have a block greater than a frame, in order to avoid an overlap between frames. We then define the notion of *local coverage of a frame* by a sequence of blocks.

Definition 2: Given a frame F, a sequence of blocks $C = \langle B_1, \ldots, B_m \rangle$ is a local coverage of F, if and only if all blocks in C occur in F in non-overlapping manner, and by following the order in C.

More strictly, for each B_i, let ϕ_i be the occurrence time of B_i in F, the following relation holds:

$$\forall i \in [1, m-1], \quad \phi_i + \|B_i\| \leq \phi_{i+1}$$

Example: In Fig.2(b), $C = \langle B_1, B_2 \rangle$ occurs in non-overlapping manner and by following this order in F_1, and so it is a local coverage of F_1.

With the above definition, a *coverage* over a set of frames is dependant of *locale coverage* of each frame of the set. We define a *coverage* over $\mathcal{F} = \{F_1, \ldots, F_l\}$ using a set of candidate blocks S as a set of the local coverages of the frames.

Definition 3: Let S be a set of candidate blocks $\{B_1, \ldots, B_n\}$ and $\mathcal{F} = \{F_1, \ldots, F_l\}$ be a set of frames. A coverage of $\mathcal{F}$ using S is a set $\{C_1, \ldots, C_l\}$ such that $\forall i \in [1, l]$, C_i is a local coverage of F_i using blocks in S.

Note that in the above definition, there may exist frames F_i such as their local coverage C_i is the empty sequence.

These frames cannot be covered with blocks in S at all.

The covering degree of a coverage is the proportion of the number of events in the frames of a trace file that are covered by the blocks in the coverage.

Definition 4: Let $\mathcal{C} = \{C_1, \ldots, C_l\}$ be a coverage of a set of frames $\mathcal{F} = \{F_1, \ldots, F_l\}$.The covering degree of $\mathcal{C}$ over $\mathcal{F}$ is defined as follows:

$$coverDegree(\mathcal{C}, \mathcal{F}) = \frac{\sum_{i=1}^{l} \sum_{j=1}^{v_i} \|B_j^i\|}{\sum_{j=1}^{l} \|F_i\|}$$

where B_j^i is the *j-th* block in the *i-th* local coverage C_i in $\mathcal{C}$.

Example: For a given set of candidate blocks $S = \{\langle A, B\rangle, \langle B, D\rangle, \langle D, C\rangle\}$, a coverage of $\mathcal{F} = \{F_1, F_2, F_3\}$ in Fig.3 is $\mathcal{C} = \{C_1, C_2, C_3\}$, with $C_1 = \langle\langle B, D\rangle\rangle$, $C_2 = \langle\langle B, D\rangle\rangle$, and $C_3 = \langle\langle D, C\rangle\rangle$. Its covering degree $coverDegree(\mathcal{C}, \mathcal{F})$ is $\frac{2+2+2}{10} = 0.6$

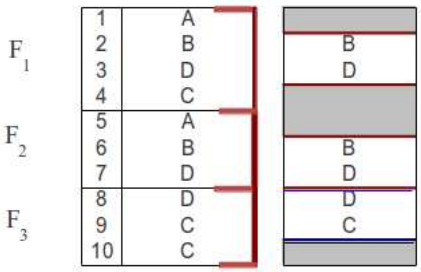


Figure 3. A set of frames with a coverage:$\{\langle\langle B, D\rangle\rangle, \langle\langle B, D\rangle\rangle, \langle\langle D, C\rangle\rangle\}$

A set of candidate blocks S may lead to many coverages of a set of frames. We define the coverage rank of S on F as the maximum degree of all the coverages that can be built from the set S.

Definition 5: Let S be a set of blocks, $\mathcal{F}$ be a set of frames and $\{\mathcal{C}_1, \ldots \mathcal{C}_p\}$ be the set of all coverages of $\mathcal{F}$ using blocks in S, the *coverage rank* of S on $\mathcal{F}$ is defined as follows:

$$coverRank(S, \mathcal{F}) = \underset{i \in [1,p]}{Max} \quad coverDegree(\mathcal{C}_i, \mathcal{F})$$

Example: The *coverage rank* of S on the set of frames of Fig. 3 is 0.8 with the coverage $\{\langle\langle A, B\rangle, \langle D, C\rangle\rangle, \langle\langle B, D\rangle\rangle, \langle\langle D, C\rangle\rangle\}$

Remark: $\forall\ S, \mathcal{F},\ 0 \leq coverRank(S, \mathcal{F}) \leq 1$

Given a set of frames $\mathcal{F}$, we can compare the coverage ranks of different S having a fixed size k, and choose S maximizing the coverage rank. Such a set of blocks, with size k are called *k-golden set*, and their elements, the *golden blocks*. The golden blocks in a *k-golden set* provide the maximum power of coverage on the set of frames for any combination of k blocks.

Definition 6: In a family $\{S_1, \ldots, S_q\}$ of sets of blocks where all sets have an identical size k, a *k-golden set* is a set S_i, satisfying $coverRank(S_i, \mathcal{F}) = \underset{j \in [1,p]}{Max} \quad coverRank(S_j, \mathcal{F})$.

Example: Assuming that $\langle C\rangle$, $\langle A, B\rangle$, $\langle B, D\rangle$, and $\langle D, C\rangle$ are frequent consecutive events for the set of frames in Fig. 3, let us consider the following sets consisting of 3 blocks: $S_1 = \{\langle C\rangle, \langle B, D\rangle, \langle D, C\rangle\}$, and $S_2 = \{\langle C\rangle, \langle A, B\rangle, \langle D, C\rangle\}$, $S_3 = \{\langle C\rangle, \langle A, B\rangle\, \langle B, D\rangle\}$, and $S_4 = \{\langle D, C\rangle, \langle A, B\rangle\, \langle B, D\rangle\}$ $coverRank(S_1, \mathcal{F}) = 0.8$; $coverRank(S_2, \mathcal{F}) = 0.9$; $coverRank(S_3, \mathcal{F}) = 0.7$; $coverRank(S_4, \mathcal{F}) = 0.8$. S_2 is then the *3-golden set*.

C. Problem statement

Though our proposed framework is generic so that it can treat blocks provided from a trace data under various measures, we assume here that the blocks we are looking for are frequents, since in the context of multimedia application debugging, particularly video decoding, a frame decoding generally follows the same procedure.

Given a set $\mathscr{S}$ of candidate blocks (obtained by a frequent sequence mining algorithm [11]–[13] applied on a set of frames in a trace file), the problem addressed in this paper is to find a *k-golden set* of blocks that provide the best coverage of the frames in the trace file. This problem can be seen as a submodular function maximization problem [14] which has been studied in particular for resource allocation. More precisely, it is a monotone submodular maximization, $max\{f(S) : |S| \leq k\}$, where f is the function of coverRank (section II-B, Def. 6), which is a submodular function of a subset of the blocks set $\mathscr{S}$. This problem is known to be NP-Hard. But it was proven that the lower bound of any local optimum f, any local optimum coverRank in our case, provided by a simple greedy search, is not less than 63% of the global maximum, and it is reported in many studies that the greedy algorithm provides nearly optimum values in various problems [15]–[18].

In the next section, we provide a greedy algorithm that we have implemented in FrameMiner to compute an approximated golden blocks. We compare its results with a baseline algorihm that computes an exact solution but with an exponential time computation.

III. Coverage with golden blocks in FrameMiner

A naive idea to obtain the golden set is to first generate all sets consisting of k blocks respectively, and then find among them the set that maximizes the $coverRank$. The algorithm

Algorithm 1 BaselineFrame

Input: A given set of blocks $\mathscr{S}$, the set of frames $\mathcal{F}$, the constraint k

Output: The complete set of k-golden sets

1: $SS \leftarrow \{S_i | S_i \subseteq \mathscr{S}, |S_i| = k\}$ {where $|S_i|$ means the number of blocks in S_i}
2: $max \leftarrow 0$
3: $G \leftarrow \{\emptyset\}$
4: **for** each $S_i \in SS$ **do**
5: $d \leftarrow coverRank(S_i, \mathcal{F})$
6: **if** $d > max$ **then**
7: $G \leftarrow \{S_i\}$
8: $max \leftarrow d$
9: **else if** $d = max$ **then**
10: $G \leftarrow G \cup \{S_i\}$
11: **end if**
12: **end for**
13: **return** G

for this simple method, termed *BaselineFrame*, is presented in Algorithm 1.

Although BaselineFrame Algorithm is simple and ensures that we obtain all exact solutions, it has an exponential time complexity : the number of subsets in SS in line 1 is C_n^k where $|S| = n$. Therefore, we introduce a greedy algorithm depicted in Algorithm 2 to avoid this costly enumeration of all subsets candidates.

Algorithm 2 GreedyFrame

Input: A given set of blocks $\mathscr{S}$, the set of frames $\mathcal{F}$, the constraint k

Output: An approximated k-golden set S_a

1: Randomly pick a block b in $\mathscr{S}$
2: $S_a \leftarrow \{b\}$
3: $\mathscr{S} \leftarrow \mathscr{S} - \{b\}$
4: **while** $|S_a| \neq k$ **do**
5: $b \leftarrow argmax_{b \in \mathscr{S}}(coverRank(S_a \cup \{b\}, \mathcal{F}))$
6: $S_a \leftarrow S_a \cup \{b\}$
7: $\mathscr{S} \leftarrow \mathscr{S} - \{b\}$
8: **end while**
9: **return** S_a

Algorithm 2 first randomly pick a block b in $\mathscr{S}$ (line 1) and initializes the solution S_a with this block. b is then excluded from $\mathscr{S}$ to avoid duplicating blocks in the result. At each iteration, the algorithm looks for the best block b such that the obtained set covers the maximum number of events in the frames (line 5). This block is added to the solution S_a in line 6 and excluded from future choices in line 7. The algorithm stops when the solution has k blocks, and returns the solution found.

In this algorithm, a problem could be the quality of the solution found. The algorithm has only k iterations, but there is no guarantee that it finds an optimal solution. The result is only an approximation of a golden set. However, in case of submodular functions it has been proven [14] that if S_{opt} is a k-golden set, then the following inequality holds :

$$coverRank(S_a) \geq (1 - \frac{1}{e}) coverRank(S_{opt})$$

This means that the $coverRank$ of S_a is guaranteed to be more than 63% of the $coverRank$ of S_{opt}. Moreover, as mentioned before, many past work pointed that the greedy algorithm provide a solution having the nearly optimum value in many problems. Accordingly, the resultant set S_a is expected to achieve a reasonable coverage w.r.t. to its much lower computational cost.

The complete mining algorithm of FrameMiner, starting from the set of frames, the predefined size k and a threshold ε, is described in Algorithm 3.

Algorithm 3 FrameMiner

Input: A set of frames $\mathcal{F}$, the constraint k, a minimum support threshold ε

Output: An approximated k-golden set S_a

1: $\mathscr{S} \leftarrow profspan(\mathcal{F}, \varepsilon)$
2: $S_a \leftarrow GreedyFrame(\mathscr{S}, \mathcal{F}, k)$
3: **return** S_a

FrameMiner starts by computing all the frequent sequential patterns occurring in the frames with a classical sequence mining algorithm (here $profspan$ [12]), with an user-defined minimum support threshold ε. The patterns obtained are the blocks that are given as input to $GreedyFrame$, which returns as output an approximated k-golden set.

IV. Experiments

We have performed preliminary experiments of our FrameMiner tool on a real trace of an embedded multimedia application. The computer used for the experiments has an Intel Xeon X4760 processor at 2.66 GHz and 64 Gigabytes of RAM. Our trace comes from an execution of a video and audio decoding test application, run on a Linux distribution. This trace was recently used by Lopez et al. [19], it is a real trace but of small size (240 frames). This small size helped us to conduct a first series of qualitative experiments to assess the interest of golden patterns. A preprocessing step is first applied to split the raw trace into frames with the two given *start* and *end* events; After this first step of preprocessing, the dataset contained 240 frames with 123575 events. At the end of this step some frames were strictly identical, they were grouped together in order to reduce the amount of redundant information. The non-redundant frame

set obtained has 99 frames and 1974 events. We then applied *FrameMiner* on non-redundant frame set, with a support threshold of 20% and different values of k. The frequent sequence mining step, using the *profspan* algorithm, returned 95 frequent sequences with a 20% support threshold.
In the rest of this section, the quality of our approximation of golden patterns with $greedyFrame$ is evaluated. Then a subjective assessment of the interest of golden patterns is presented, and a quantitative measure of the amount of information reduction is shown.

Qualitative assessment: To evaluate the quality of the golden set approximation found by $greedyFrame$, we compared it with the optimal solution found by $baselineFrame$. Due to the exponential computation time of $baselineFrame$, it couldn't compute optimal solutions on the frame set having 99 frames. We thus considered two smaller samples of 20 and 30 frames. Using $profspan$ with 20% threshold, the first sample gave 16 frequent patterns, and the second sample gave 18 frequent patterns. Table I below shows coverRank of the *4-golden* set and *6-golden* set, for approximated and optimal solution, on first samples. For the overall set of 99 frames, optimal *4-golden* set and *6-golden* set are only presented for $greedyFrame$, as they couldn't be computed by $baselineFrame$. The table shows also the lower bound claimed.

Table I
COVERRANK OF GOLDEN SETS AND APPROXIMATED GOLDEN SETS

Nb. frames	k	Optimal	FrameMiner	Lower bound
Set of 20 frames	4	0.776	0.556	0.489
	6	0.858	0.830	0.540
Set of 30 frames	4	0.710	0.533	0.447
	6	0.812	0.720	0.511
Set of 99 frames	4	N/A	0.527	N/A
	6	N/A	0.656	N/A

For each set of frames in Table I, we performed ten executions of *FrameMiner* and the value represented is the lowest value obtained for all executions. We observed in the experiments that, the approximated value obtained is always much better than the lower bound. By example, for 30 frames, k = 6, the coverRank of the approximated solution is 0.72, which is greater than $> 0.63 * optimal = 0.511$.
Figure 4 compares the computation time of *greedyFrame* and *baselineFrame* for $k = 4$ and $k = 6$. Overall, the running time grows with the number of frames and the constraint k. For example, with a set of 30 frames, for $k = 4$, $baselineFrame$ takes $6.2s$ against $0.13s$ in average for greedyFrame; for $k = 6$, $baselineFrame$ takes $17.2s$ against $0.17s$ in average for greedyFrame. With these results, we observe that the approximated solution is almost 90% of the optimal solution, with an execution two to three orders of magnitude faster.

Subjective ranking: To evaluate the usefulness of the golden blocks obtained, we subjected 95 frequent patterns to expert determination, without revealing which one were golden blocks. The expert is a programmer of multimedia applications on SoC and he is familiar with our dataset. He had to classify the patterns into three categories. *Rank A* are very interesting patterns that the programmer would like to see. *Rank B* are somewhat interesting patterns, the programmer might use it in debugging. *Rank C* corresponds to uninteresting patterns, the programmer would not use it in debugging. Figure 5 shows the proportion of each of these rank for all frequent patterns on the left, and only for golden blocks on the right.

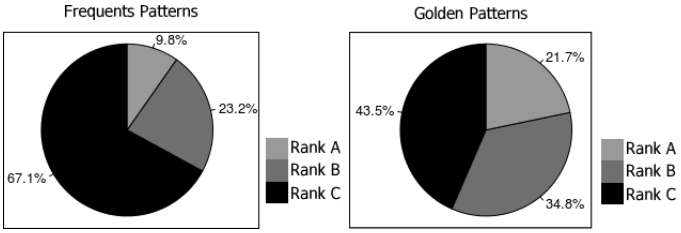


Figure 5. Subjective ranking of golden blocks

We are able to view that the expert finds more than 50% of golden blocks very or somewhat interesting. With the 95 frequent patterns, he found 67.1% of uninteresting patterns. The percentage of interesting patterns was enriched in golden blocks, confirming the interest of our method.

Reduction: Our method significantly reduce the amount of trace to observe. After a rewriting of the overall non-redundant frame set of 99 frames, a programmer has only 502 *events* to observe instead of 1974. This represents a *70%* reduction in the information to handle. As an example, Fig. 6 shows three frames and their rewriting with golden blocks. In this rewriting, gray boxes represent golden blocks, and white ones represent "gaps", i.e. part of the frame which was not covered by golden blocks.
With Figure 6, programmer can more quickly see that, frames 6(a) and 6(b) are similar, but in 6(b), an event occurs between the *start* block `[GetFrame,exitGet]` and the *interrupt* block `[Interrupt_period, exitI, Interrupt_soft, exitS, exitIT]`. In 6(c), an *interrupt* block is expected after the *start* block, but the "gap" allows to observe interruption called `[Interrupt_Handler]`, which is particular in this context. In visualization before golden blocks, it was not easy to make a difference between what was interesting to look or not. Even less between which event or block is expected or not. The programmer can now focus on "blocks" instead of individual events. It has fewer areas of traces to observe and may plan to analyse only areas not covered by golden blocks.

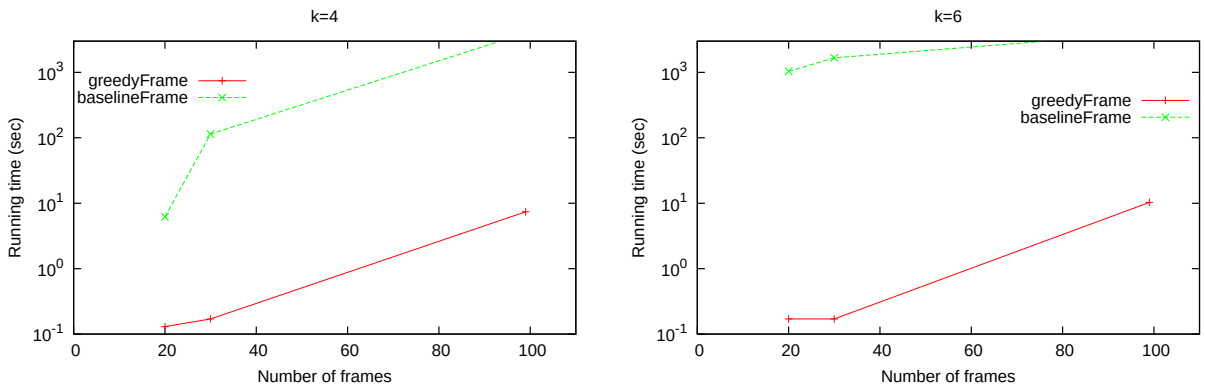


Figure 4. Computation times of $greedyFrame$ and $baselineFrame$ for $k = 4$ and $k = 6$

V. Discussion

Figure 5 shows that golden blocks are interesting patterns for understanding application behaviour. Several patterns have been said *unexpected* by the expert, it was in fact patterns containing events not being directly related to the decoding but to the scheduling done by the operating system. These last could be considered noise. According to the expert, the golden blocks found represent different phases of the video decoding. We can cite few golden blocks such as *start decoding frame* represented by the block `[GetFrame, exitGet]`, which represents a call to the user function of video decoding start, or the *keys interrupt* `[Interrupt_period, exitI, Interrupt_soft, exitS, exitIT]`. The expert says that these interruptions are necessary during the processing of the frame.

As much as the golden blocks are important, as much as the ”gaps” are, in the context of trace analysis. Indeed, the rewriting of all the frames helps to quickly find blanks. In theory, these empty blocks mark a fairly uncommon sequence, probably of interest in order to investigate potential dysfunctions. For example, through our rewriting, the programmers were able to search a ”gap“ when they expected a precise sequence. They therefore found that it occurred two particular interruptions `[Interrupt_timer]` and `[Interrupt_Handler]` that they were not expecting.

VI. Related work

Existing work on execution traces addressed several issues such as reducing the volume of data by abstraction or by division of the trace. Pirzadeh et al. in [2] use analysis of execution traces to understand behavioural aspects of complex software systems. They can divide the content of a trace into meaningful trace segments called execution phases, and their slicing is done using Gestalt laws. Kim et al. studied in [20] the problem of finding a minimum set of signature patterns such that each object in datasets has at least one signature pattern that explains all data. The signature patterns, which are a kind of discriminative patterns, can be mainly used in hardware design as a verification methodology. The authors assume that signature patterns are infrequent and propose a novel pattern enumeration method. In the other hand, traces need to be visualized in order to be analysed. Several works like [5,21] and [6], present various tools in this direction. However, the programmer has to go through a series of pages in order to find a specific information. Our work divides an input trace into relevant blocks, mutually independents, without any information on some execution part, unlike [2,10]. The visualization step becomes now easier to manage through information grouping, and the analysis is improved. Other authors such as Hoffman et al. in [10] use more formal abstraction techniques and build views that represent various levels of abstraction in the executing program; these views are linked each other. In a context of interprocess communication traces, authors in [22] focus on detecting communication patterns and then propose an abstraction of traces. As said before, one have to deal with huge volume of traces and the size explosion problem was largely described in [1]. Different techniques to reduce traces were proposed, as those in [7,8], and Pirzadeh in [9] introduces a notion of trace size reduction to obtain a representative sample of the original trace; however the last approach does not take into account the sequentiality of events, essential to understand multimedia application traces. The same author gives an idea in [23] to mimic the psychological processes in order to deal with huge volume of visual data. The abstraction that we realised will also reduce the amount of traces, and contrary to [20], we assume that the blocks that we are looking for, result of a sequential pattern mining algorithm.

VII. Conclusions and future work

In this work, we presented an original approach for automatically detecting golden blocks, which are frequent sequences

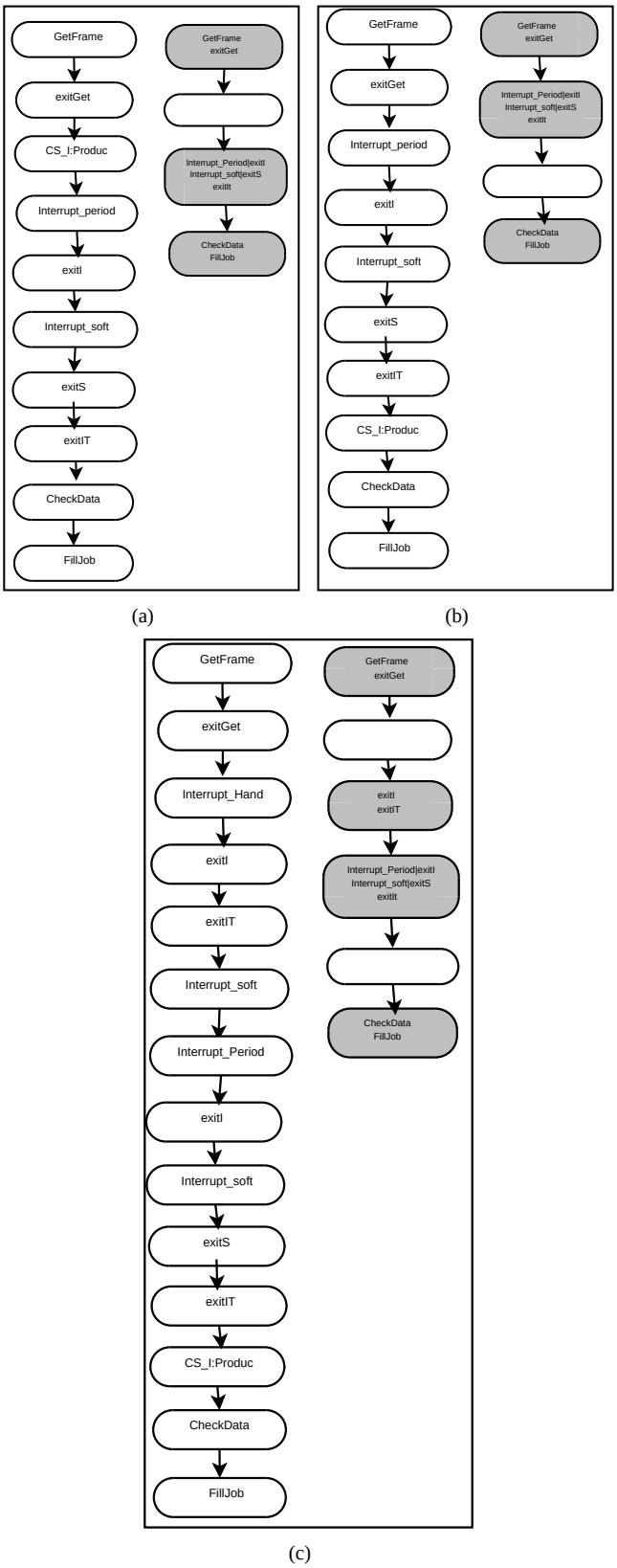


Figure 6. Initial frame and rewriting with golden blocks

having an important coverage in an execution trace. We argue that these blocks can greatly simplify trace exploration and thus debugging through trace analysis of embedded systems.

We formally define the problem of finding golden blocks and show that computing an exact solution is computationally expensive. We thus provide an efficient greedy algorithm that computes an approximation of the golden blocks. Experimental results show that the automatically found blocks are interesting for experts. The rewriting of frames using these blocks allows a significant reduction of the execution trace volume.

In future work, we are thinking on using semantic to automatically label these golden blocks. By that mean, we can refer to ontologies for describing the trace abstractions, as well as automatic detection of faults. Several metrics will be taken into account like duration of events. In this paper we focused on single core traces, but we plan to apply our methods to multi-core traces.

Acknowledgments

This work is supported by French FUI project SoCTrace.

References

[1] A. Hamou-lhadj and T. C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques,” 2004, pp. 1–14.

[2] H. Pirzadeh and A. Hamou-Lhadj, “A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension,” in *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*. Ieee, Apr. 2011, pp. 221–230.

[3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, “Understanding Execution Traces Using Massive Sequence and Circular Bundle Views,” *15th IEEE International Conference on Program Comprehension (ICPC ’07)*, pp. 49–58, 2007.

[4] X. gurin, “Approche Efficace de Développement de Logiciel Embarqué pour des Systèmes Multiprocesseurs sur Puce,” Ph.D. dissertation, 2010.

[5] B. D. O. Stein, “Pajé trace file format,” 2003.

[6] Z. Weg and R. Henschel, “Introducing OTF / Vampir / VampirTrace,” *Memory*.

[7] P. Dugerdil, “Using trace sampling techniques to identify dynamic clusters of classes,” in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research - CASCON ’07*. New York, New York, USA: ACM Press, 2007, p. 306.

[8] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying, “Scaling an object-oriented system execution visualizer through sampling,” in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 237–.

[9] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, and A. Mehrabian, "The Concept of Stratified Sampling of Execution Traces," in *2011 IEEE 19th International Conference on Program Comprehension*. Ieee, 2011, pp. 225–226.

[10] K. J. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware trace analysis," in *ACM SIGPLAN Notices*, vol. 44, no. 6, May 2009, p. 453. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1543135.1542527

[11] Q. Zhao and S. S. Bhowmick, "Sequential Pattern Mining: A Survey," *Database*, no. 2003118, 2003.

[12] J. Zou, J. Xiao, R. Hou, and Y. Wang, "Frequent Instruction Sequential Pattern Mining in Hardware Sample Data," *2010 IEEE International Conference on Data Mining*, pp. 1205–1210, Dec. 2010.

[13] T. Ball, B. Laboratories, and L. Technologies, "The Concept of Dynamic Analysis," *Analysis*.

[14] L. W. Gi. NEMHAUSER* and M. L. FISHER, "An analysis of approximations for maximizing submodular set functions," vol. 14, pp. 265–294, 1978.

[15] A. Kulik, H. Shachnai, and T. Tamir, "Maximizing submodular set functions subject to multiple linear constraints," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '09. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009, pp. 545–554. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496770.1496830

[16] J. Vondrak, "Optimal approximation for the submodular welfare problem in the value oracle model," in *Proceedings of the 40th annual ACM symposium on Theory of computing*, ser. STOC '08. New York, NY, USA: ACM, 2008, pp. 67–74. [Online]. Available: http://doi.acm.org/10.1145/1374376.1374389

[17] G. Goel and A. Mehta, "Online budgeted matching in random input models with applications to adwords," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '08, 2008, pp. 982–991.

[18] M. Streeter and D. Golovin, "An online algorithm for maximizing submodular functions," 2007.

[19] P. L. Cueva, A. Bertaux, A. Termier, J. Mehaut, and M. Santana, "Debugging Embedded Multimedia Application Traces through Periodic Pattern Mining," in *Proceedings of International Conference on the Embedded Software*.

[20] H. Kim, S. Im, T. Abdelzaher, J. Han, D. Sheridan, and S. Vasudevan, "Signature Pattern Covering via Local Greedy Algorithm and Pattern Shrink," *2011 IEEE 11th International Conference on Data Mining*, pp. 330–339, Dec. 2011.

[21] "Visual trace explorer." [Online]. Available: http://vite.gforge.inria.fr/

[22] L. Alawneh and A. Hamou-Lhadj, "Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication," in *2011 15th European Conference on Software Maintenance and Reengineering*. Ieee, Mar. 2011, pp. 211–220. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5741263

[23] H. Pirzadeh and A. Hamou-lhadj, "A Software Behaviour Analysis Framework Based on the Human Perception Systems (NIER Track)," pp. 948–951.