Efficiently Rewriting Large Multimedia Application Execution Traces with few Event Sequences

C. Kamdem Kengne, L. C. Fopa
University of Grenoble, LIG
surname.name @imag.fr
University of Yaounde I, LIRIMA
Yaounde, Cameroon

A. Termier, N. Ibrahim, M.-C. Rousset
University of Grenoble, LIG
Grenoble, France
surname.name @imag.fr

T. Washio
Institute of Scientific and Industrial Research
Osaka University, Japan
washi@ar.sanken.osaka-u.ac.jp

M. Santana
STMicroelectronics
Crolles, France
miguel.santana@st.com

ABSTRACT
The analysis of multimedia application traces can reveal important information to enhance program execution comprehension. However, typical size of traces can be in gigabytes, which hinders their effective exploitation by application developers. In this paper, we study the problem of finding a set of sequences of events that allows a reduced-size rewriting of the original trace. These sequences of events, that we call blocks, can simplify the exploration of large execution traces by allowing application developers to see an abstraction instead of low-level events.

The problem of computing such a set of blocks is NP-hard and naive approaches lead to prohibitive running times that prevent analysing real-world traces. We propose a novel algorithm that directly mines the set of blocks. Our experiments show that our algorithm can analyse real traces of up to two hours of video. We also show experimentally the quality of the set of blocks proposed, and the interest of the rewriting to understand the actual trace data.

Categories and Subject Descriptors
H.2.8 [Database Applications]: Data mining; H.3.1 [Content Analysis and Indexing]: Abstracting methods

Keywords
Pattern mining, Combinatorial optimization, Execution traces, Multimedia applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD ’13 Chicago, Texas USA
Copyright 20XX ACM X-XXXX-XX/XX/XX ...$15.00.

1. 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 need 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 for audio/video decoding solution providers such as STMicroelectronics. Increasingly, the analysis techniques of applications use execution traces, which are sequences of timestamped events produced by an application or a system, to efficiently uncover bugs causing such faulty behaviors ([2, 4, 10]).

The challenge in this case is the size of these traces that can easily reach gigabytes for only a few minutes of decoding. For instance, the STMicroelectronics video decoding application DVBTest can produce a trace file of 1 Gigabyte for less than 10 minutes of playback. Various studies have proposed techniques to reduce the volume of traces ([13, 16]) with sampling methods. These techniques can obtain a reduced execution trace but not always representative of the entire trace [11]. Pirzadeh et al. introduced in [10] that the general consensus in the trace analysis community is to emphasise the work towards effective trace abstraction techniques, such as [5].

In this paper, we investigate an approach based on covering frames by blocks that are sequences of low-level events. More precisely, given a set of frames, the problem is to discover a given (small) number of blocks that cover as much as possible each frame of the input set, thus making possible to rewrite them using blocks. Fig. 1 illustrates a trace with frames and blocks.

In fact, this maximal covering problem is a variant of the packing problem which is NP-hard, and for which no generic
algorithm leading to local or global solution is known. We thus propose and experimentally compare several greedy algorithms. They differ in the way they discover candidate blocks, either as a preliminary step independent of the coverage test, or combined with the coverage test. As a result we show that two of our proposed greedy algorithms scale to gigabyte-sized traces.

For the trace in Fig. 2,

Example: of a block in a frame.

The first definition that we introduce is the occurrence time

defined by timestamps. The size of a sequence

Q

of timestamped events and a

trace

is a timestamp and

e

timestamped event

is a pair

B

⟨

D

e

B

1

,...,e

v

B

⟩

is a block of two events

and

B

is a block of two events

Example: In Fig.2(b),

F

occurs in

F

between timestamps 2 and 3; it occurs in

F

blocks of the covering can only occur inside frames. The

global coverage of the set of frames can thus be expressed through a series of local coverages of each of the frames. A local coverage is a sequence of blocks taken from a given large set of blocks, and verifying the constraints stated below.

DEFINITION 2. Given a frame

F

and a set of blocks

S
,

a sequence of blocks

C

= (B₁,...,Bₘ), with

m ≤ |S|

and

∀ i ∈ [1,m] B_i ∈ S

is a local coverage of

F

if and only if all blocks in

C

occur in

F

in a non overlapping manner, and by following the order in

C
.

More formally, for each

B_i ∈ C
,

let

φ_i

be the occurrence time of

B_i

in

F
,

the following relation holds:

∀ i ∈ [1,m − 1],

φ_i + |B_i| ≤ φ_{i+1}

Note that the

B_i

are not necessarily distinct blocks: the same block can appear several times in a local coverage. Moreover, for a given

F

and

S
,

there may be many local coverages satisfying the definition.

Example: In Fig.2(b),

C

= (B₁,B₂)

occurs in non overlapping manner and by following this order in

F₁
,

and so it is a local coverage of

F₁

when considering

S

= (B₁,B₂,B₃).

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

F

= \{F₁,...,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

{B₁,...,Bₘ}

and

F

= \{F₁,...,F_l\}

be a set of frames. A coverage of

F

using

S

is a set

{C₁,...,C_l}

such that

∀ i ∈ [1,l], C_i

is a local coverage of

F_i

using blocks in

S
.

Figure 1: An example trace with blocks

This paper is organized as follows: Section 2 states the problem and briefly gives some notations and important definitions. In Section 3, we present our approaches based on greedy algorithms. Section 4 reports on experiments done on real traces of multimedia applications. Section 5 is an overview of the related work. We end in Section 6 by a conclusion and future work.

2. PRELIMINARIES AND PROBLEM STATEMENT

In this section we give the notations and definitions necessary to model our problem. This formalism comes from our previous work [6].

2.1 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 ∈ \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

Q

, denoted by

∥Q∥

, is the total number of events that it contains. Example: For the trace in Fig. 2,

∥F₁∥ = 4. B₂ = (B,D)

is a block of two events

B

and

D.

2.2 Definitions

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

DEFINITION 1. Let

B = ⟨e₁,.,eₙ⟩

be a block and let

Figure 2: Example of trace, frames and blocks

F = ((t₁,e₁),.,(tₙ,eₙ))

be a frame. B occurs in

F

denoted

B ⊆ F

between timestamps

i

and

i+v

iff:

∀ j ∈ [i,i + v],

e_j = e^B_{i+j+1}.

i

is then called the occurrence time of

B

in

F.

Example: In Fig.2(a),

B₁ = (B,D)

occurs in

F₁

between timestamps 2 and 3; it occurs in

F₂

blocks 1, 2, 3, 4, and 5 of the covering can only occur inside frames. The global coverage of the set of frames can thus be expressed through a series of local coverages of each of the frames. A local coverage is a sequence of blocks taken from a given large set of blocks, and verifying the constraints stated below.

DEFINITION 2. Given a frame

F

and a set of blocks

S
,

a sequence of blocks

C

= (B₁,...,Bₘ), with

m ≤ |S|

and

∀ i ∈ [1,m] B_i ∈ S

is a local coverage of

F

if and only if all blocks in

C

occur in

F

in a non overlapping manner, and by following the order in

C.

More formally, for each

B_i ∈ C
,

let

φ_i

be the occurrence time of

B_i

in

F
,

the following relation holds:

∀ i ∈ [1,m − 1],

φ_i + |B_i| ≤ φ_{i+1}

Note that the

B_i

are not necessarily distinct blocks: the same block can appear several times in a local coverage. Moreover, for a given

F

and

S
,

there may be many local coverages satisfying the definition.

Example: In Fig.2(b),

C

= (B₁,B₂)

occurs in non overlapping manner and by following this order in

F₁
,

and so it is a local coverage of

F₁

when considering

S

= (B₁,B₂,B₃).

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

F

= \{F₁,...,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

{B₁,...,Bₘ}

and

F

= \{F₁,...,F_l\}

be a set of frames. A coverage of

F

using

S

is a set

{C₁,...,C_l}

such that

∀ i ∈ [1,l], C_i

is a local coverage of

F_i

using blocks in

S.


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

$$\text{coverDegree}(C, F) = \frac{\sum_{i=1}^{l} \sum_{j=1}^{n_i} \|B_i^j\|}{\sum_{i=1}^{l} \|F_i\|}$$

where $B_i^j$ is the $j$-th block in the $i$-th local coverage $C_i$ in $C$.

**Example:** For a given set of candidate blocks $S = \{(A, B), (B, D), (D, C)\}$, a coverage of $F = \{F_1, F_2, F_3\}$ in Fig.3 is $C = \{C_1, C_2, C_3\}$, 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 $\text{coverDegree}(C, F)$ is $\frac{2 \times 1.2}{10} = 0.24$.

![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\]$](image)

Because a set of candidate blocks $S$ may lead to many local coverages for a single frame (Def. 2), it may also lead to many coverages for 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, $F$ be a set of frames, and $\{C_1, \ldots, C_p\}$ be the set of all coverages of $F$ using blocks in $S$; the coverage rank of $S$ on $F$ is defined as follows:

$$\text{coverRank}(S, F) = \max_{i \in \{1, p\}} \text{coverDegree}(C_i, F)$$

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

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

Given a set of frames $F$, 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 $k$, is called $k$-most representative block set (denoted $k$-MRBS), and its elements, the most representative blocks (noted MR-blocks). The most representative blocks in a $k$-most representative block 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 $k$-most representative block set is a set $S_i$, satisfying:

$$i = \arg\max_{j \in [1, q]} \text{coverRank}(S_j, F)$$

**Example:** Assuming that $\langle\langle C\rangle\rangle, \langle\langle A, B\rangle\rangle, \langle\langle B, D\rangle\rangle, \langle\langle D, C\rangle\rangle$ are frequent consecutive events for the set of frames in Fig. 3, let us consider the following sets consisting of 3 blocks: $S_1 = \{\langle\langle C\rangle\rangle, \langle\langle B, D\rangle\rangle, \langle\langle D, C\rangle\rangle\}$, and $S_2 = \{\langle\langle C\rangle\rangle, \langle\langle A, B\rangle\rangle, \langle\langle D, C\rangle\rangle\}$, $S_3 = \{\langle\langle C\rangle\rangle, \langle\langle A, B\rangle\rangle, \langle\langle B, D\rangle\rangle\}$, and $S_4 = \{\langle\langle D, C\rangle\rangle, \langle\langle A, B\rangle\rangle, \langle\langle B, D\rangle\rangle\}$.

$\text{coverRank}(S_1, F) = 0.8; \text{coverRank}(S_2, F) = 0.9; \text{coverRank}(S_3, F) = 0.7; \text{coverRank}(S_4, F) = 0.8$. $S_2$ is then the 3-most representative block set (3-MRBS).

### 2.3 Problem statement

The problem that we consider is the following: given as input a set of frames $F$ and a number $k$, our goal is to output a $k$-most representative block set $S$ that maximizes the coverage of $F$.

This problem is a variant of the packing problem [3], which consists of maximally filling a space with $k$ types of pieces. In our case the space is the set of frames, and the pieces are the blocks. An additional constraint in our setting is that the location of each type of piece is constrained: a block can only cover specific places of the frames. An additional difficulty of our case is that the pieces, i.e. the blocks, are not given as input, and must be computed from the data. The packing problem is a NP-hard problem. There is no known generic algorithm for global optimization of this problem. In the next section, we propose several approaches to solve our problem, based on greedy algorithms.

### 3. APPROACH

The problem of finding a limited set of blocks allowing to maximal cover a set of frames can be decomposed into two subproblems:

- find a large set $S_0$ of “candidate” blocks that are subsequences of frames
- find $S \subset S_0$ such that $|S| = k$ and the coverage of the frames is maximized by the blocks of $S$

#### 3.1 Baseline approaches

A naive approach, that we exposed previously in [6], consists in solving these two subproblems separately. A good heuristic for the discovery of the blocks is to assume the blocks are frequent sequences of the set of frames. Such frequent sequences of large support (i.e. appearing in many frames) and of sufficient length are likely to exhibit important coverage values. Standard pattern mining algorithms can be applied to discover the complete set $S_0$ of frequent sequences [19]. Then, a simple greedy algorithm can be used to choose
the $k$ frequent sequences of $S_0$ that maximize coverage. We call this approach NaiveBaseline. The disadvantage of this approach is that it has a prohibitive computation time. Computing the frequent sequences has a time complexity exponential in the number of events in the frames, and can output thousands, even millions of frequent sequences. The greedy algorithm is then confronted with a very large combinatorial search space, thus requiring long computation. In our experiments, finding blocks for rewriting a small set of 200 frames (less than 10 seconds of video) took more than 10 hours on a standard computer. This simple approach thus do not scale to real world multimedia traces having tens of thousands of frames, and cannot be exploited to help multimedia application developers.

To address this limitation, we propose several approaches, which are based on the following ideas: 1) the greedy algorithm should have a considerably smaller search space, i.e. receive several orders of magnitude less frequent sequences to choose from; 2) reducing the number of frequent sequences should be done by considering coverage constraints.

An aggressive reduction in the number of input frequent sequences given to the greedy algorithm may prevent to find a solution with $k$ elements. All the approaches that we propose are thus based on an iterative structure, where in each iteration a set of frequent sequences is generated, then passed to the greedy algorithm. If the solution found has $k$ blocks the algorithm stops, else it continues to further add extra blocks having large coverages until the number of blocks reaches $k$.

In order to illustrate this structure, consider the pseudo code of Algorithm 1, which consists in our RandomBaseline approach.

\begin{algorithm}
\caption{RandomBaseline}\label{alg:random}
\textbf{Input}: A set of frames $F$, an integer $K$, a frequency threshold $\varepsilon$, minimum block size $m$, size of greedy algorithm input $\ell$
\textbf{Output}: A set $S$ of frequent sequences optimizing coverage over $F$, with $|S| = k$

1: $S \leftarrow \emptyset$
2: AllFreqSeq $\leftarrow$ computeFrequentSequences($F, \varepsilon, m$)
3: \textbf{while} $|S| < k$ and AllFreqSeq $\neq \emptyset$ \textbf{do}
4: \hspace{1em} PatPool $\leftarrow$ randomly get $\ell$ frequent sequences from AllFreqSeq
5: \hspace{1em} $S_{new} \leftarrow$ greedyChooseBlocks(PatPool, $F, k - |S|$)
6: $S \leftarrow S \cup S_{new}$
7: $F \leftarrow$ Remove all blocks of $S_{new}$ from $F$
8: AllFreqSeq $\leftarrow$ AllFreqSeq \setminus PatPool
9: \textbf{end while}
10: \textbf{return} $S$
\end{algorithm}

This algorithm is still a baseline that it only exploits intuition 1) above when $\ell << \text{AllFreqSeq}$. As in the NaiveBaseline approach, the complete set of frequent sequences AllFreqSeq is computed beforehand in line 2. Then in the iteration of lines 3-9, a set PatPool of fixed size $\ell$ (user given) is taken from AllFreqSeq (line 4). This set is fed into the greedy algorithm, which produces (a part of) the solution in line 5. Blocks in the solution are removed from the frames (line 7), and if the solution do not have $k$ blocks the algorithm continues. Note that in some rare cases (for example when $\text{AllFreqSeq}$ is small, or when $k$ is set too high), the algorithm may not find a solution. Although such cases are unlikely to happen on real data, should they happen, the user would have to decrease $k$ and/or decrease the support threshold $\varepsilon$.

We briefly review the function greedyChooseBlocks, presented in Algorithm 2. It is a standard greedy algorithm: the algorithm is given a target number of blocks $k'$, and iterates as long as its solutions has less than $k'$ blocks and it has not exhausted patterns of PatPool. At each iteration it chooses the block $B'$ giving best coverage in line 4 and add it to the solution $S_{new}$. The algorithm then suppresses all blocks of PatPool overlapping $B'$ (they can’t be part of the solution any longer), and all instances of $B'$ from a projection of the frames in order to avoid doing computations for already covered parts.

\begin{algorithm}
\caption{greedyChooseBlocks}\label{alg:greedy}
\textbf{Input}: A set of frequent sequences PatPool, a set of frames $F$, an maximal number of blocks $k'$
\textbf{Output}: A set $S_{new} \subseteq \text{PatPool}$ of frequent sequences that optimize coverage on the parts of $F$ not already covered by the blocks of $S$, with $|S_{new}| \leq k'$

1: $F' \leftarrow F$
2: $S_{new} \leftarrow \emptyset$
3: \textbf{while} $\text{PatPool} \neq \emptyset$ and $|S_{new}| < k'$ \textbf{do}
4: $B' \leftarrow \arg\max_{B \in \text{PatPool}}(\text{coverRank}(\{B\}, F'))$
5: \hspace{1em} // by definition of coverRank, $B'$ is non-overlapping with all blocks of $S_{new}$
6: $S_{new} \leftarrow S_{new} \cup \{B'\}$
7: $OB \leftarrow \{P \mid P \in \text{PatPool}, \text{overlap}(P, B')\}$
8: $\text{PatPool} \leftarrow \text{PatPool} \setminus OB$
9: $\text{Remove from }F' \text{ all instances of } B'$
10: \textbf{end while}
11: \textbf{return} $S_{new}$
\end{algorithm}

This algorithm guarantees the non-overlapping of the blocks in $S_{new}$: they will make a proper coverage of $F$ according to Def. 3. However, it is not guaranteed that this coverage will have the highest coverRank value, as the coverage is only estimated on the new block being added at each iteration, and not globally on the set of blocks. The heuristic of adding first blocks of highest coverage has good practical results, as experimentally shown in Section 4. Moreover, it avoids the huge computational price of an exhaustive computation of the best coverRank.

\subsection{One step approaches}
We now have the necessary material to present our contribution. First, recall that the main difference between NaiveBaseline and RandomBaseline is that RandomBaseline does not consider all possible frequent patterns at once in the greedy algorithm: it proceed iteratively, considering at each iteration a small set PatPool $\subseteq \text{AllFreqSet}$. This should improve the computation time of the greedy algorithm, but because patterns of PatPool are chosen at random, the coverage of the solution output may be far from optimal.

Our contribution thus consists in two approaches, coined OneStepMultiSon and OneStepOneSon, which follow an iterative structure similar to RandomBaseline, but where, by exploiting intuition 2) above, the choice of PatPool is improved. In these approaches, PatPool is guaranteed to con-
tain blocks that all have high coverage, and that are already known to participate together in at least one local coverage. The pseudo-code for OneStepMultiSon is given in Algorithm 3. The pseudo-code for OneStepOneSon is identical, expect for line 5 where the call to getFramePatternsMS is replaced by a call to getFramePatternsOS.

Algorithm 3 OneStepMultiSon

Input: A set of frames $F$, an integer $k$, a frequency threshold $\varepsilon$, minimum block size $m$

Output: A set $S$ of frequent sequences optimizing coverage over $F$, with $|S| = k$

1: $S \leftarrow \emptyset$
2: $\forall f \in F$ $f$.mark = false
3: while $|S| < k$ and $\exists f \in F$ s.t. $f$.mark = false do
4:   $f \leftarrow \text{random}\{f \in F \mid f$.mark = false\}$
5:   $\text{PatPool} \leftarrow \text{getFramePatternsMS}(f, F, \varepsilon, m)$
6:   $S_{\text{new}} \leftarrow \text{greedyChooseBlocks}(\text{PatPool}, F, k - |S|)$
7:   $S \leftarrow S \cup S_{\text{new}}$
8:   $F \leftarrow \text{Remove all blocks of } S_{\text{new}} \text{ from } F$
9:   $f$.mark $\leftarrow$ true
10: end while
11: return $S$

Before elaborating on the two different ways of computing $\text{PatPool}$, we focus on the common parts of OneStepMultiSon and OneStepOneSon, and position them w.r.t. RandomBaseline. The non-baseline algorithm are “one step”, in the sense that they don’t need to compute the whole set of frequent sequences beforehand. A reduced set of frequent sequences is computed at each iteration by getFramePatternsMS / getFramePatternsOS and feeds the greedy algorithm. The reduction comes from two points: first, the coverage constraint is taken into account during frequent sequence generation. Second, at each iteration of the algorithm, only sequences belonging to a selected random frame can be generated. This last point means that our approach is based on a sampling of frames: the blocks output by OneStepMultiSon/OneStepOneSon will be blocks appearing in a small set of randomly chosen frames (one random frame per iteration of the algorithm). This comes from the observation that usually multimedia application have a very regular execution, thus the sequences of events of the frames will be quite similar. When mining frequent sequences that should occur in most of the trace (support threshold $> 50\%$), taking a few sample frames is likely to quickly give enough blocks to get a good coverage of the whole trace.

Note that this is different from the RandomBaseline approach, where at each iteration a random sample of blocks is selected, but there are no constraints on where do this blocks come from: they may all come from different frames, possibly never appearing together in local coverages. In the algorithm, this is realized by first setting all frames as “unmarked” in line 2. In each iteration, a random sample frame $f$ in selected in line 4, which is then passed as input to the frequent sequence mining algorithm. At the end of an iteration, frame $f$ is marked in order to avoid selecting it again.

We first present the approach used in OneStepMultipleSon, by explaining function getFramePatternsMS, whose pseudo-code is given in Algorithm 4. This function is very similar to a classical pattern growth algorithm. However, there are two key differences from traditional pattern growth, that come from the fact that the goal of the patterns is to be arranged together to cover frames by a greedy algorithm later, and that constitute part of our contribution:

- all the patterns found are necessarily rooted in a random frame $f$, which severely restricts the search space
- for a given “seed” pattern of pattern growth (see below), the output is not all the frequent patterns extending this seed pattern, but only those extensions that provide better coverage than the patterns they extend.

The first step, shown in line 2, is to find pattern growth “seeds”. It is done by computing all sequences of length $m$ of the frame $f$. For each of these seeds, its extension is computed by the procedure pattGrowth called in line 4. This procedure is shown in lines 7-21. It takes as input a pattern $P$, and modifies the final output $\text{PatPool}$. First the frequency of the $P$ is tested (line 9). If $P$ is frequent, its extensions in $f$ are computed (line 11), i.e. all occurrences of $P$ plus one event $e$ are computed in $f$. The extensions that have a higher or equal coverage than $P$ (line 12) are explored recursively in line 15. If none exist, that $P$ is added to the final result $\text{PatPool}$. This way pattGrowth guarantees its maximality condition.

Algorithm 4 getFramePatternsMS

Input: A frame $f \in F$, a set of frames $F$, a frequency threshold $\varepsilon$, minimum block size $m$

Output: A set $\text{PatPool}$ of coverage-maximal frequent sequences (each of length $\geq m$) occurring in $f$ and frequent in $F$

1: $\text{PatPool} \leftarrow \emptyset$
2: $\text{Pool}_m \leftarrow \text{set of all sequences of consecutive events of } f \text{ of length } m$
3: for all $P \in \text{Pool}_m$ do
4:   $\text{pattGrowthMS}(P, \varepsilon, F, \text{PatPool})$
5: end for
6: return $\text{PatPool}$
7: procedure $\text{pattGrowthMS}(\text{in } P, \varepsilon, F, \text{in/out } \text{PatPool})$
8: begin
9: if isFrequent($P, \varepsilon, F$) = true then
10:   $c_P \leftarrow \text{coverRank}(\{P\}, F)$
11:   $\Ext_P \leftarrow \{e \in f \mid P + e \text{ is a sequence in } f\}$
12:   $\Child_P \leftarrow \{P + e \mid e \in \Ext_P \text{ s.t. } \text{coverRank}(P + e, F) \geq c_P\}$
13: if $\Child_P \neq \emptyset$ then
14:   for all $P' \in \Child_P$ do
15:     $\text{pattGrowthMS}(P', \varepsilon, F, \text{PatPool})$
16:   end for
17: else
18:   $\text{PatPool} \leftarrow \text{PatPool} \cup \{P\}$
19: end if
20: end if
21: end // procedure pattGrowthMS

The method used in getFramePatternsMS is close to a full fledged pattern mining algorithm. Especially, it may explore
Algorithm 5 getFramePatternsOS

Input: A frame $f \in \mathcal{F}$, a set of frames $\mathcal{F}$, a frequency threshold $\varepsilon$, minimum block size $m$

Output: A set $PatPool$ of cover maximal frequent sequences (each of length $\geq m$) occurring in $f$ and frequent in $\mathcal{F}$

1: $PatPool \leftarrow \emptyset$
2: $Pool_{ms} \leftarrow$ set of all sequences of consecutive events of $f$ of length $m$
3: for all $P \in Pool_{ms}$ do
4:     pattGrowthOS($P, \varepsilon, \mathcal{F}, PatPool$)
5: end for
6: return $PatPool$
7: procedure pattGrowthOS($in, P, \varepsilon, \mathcal{F}, in/out PatPool$)
8: begin
9: if isFrequent($P, \varepsilon, \mathcal{F}$) = true then
10:     $c_p \leftarrow$ coverRank($\{P\}, \mathcal{F}$)
11:     $Ext_P \leftarrow \{e \in f \mid P + e$ is a sequence in $f\}$
12:     BestExt $\leftarrow \arg\max_{e \in Ext_P}$ (coverRank($\{P + e\}, \mathcal{F}$) s.t. coverRank($\{P + $BestExt$\}, \mathcal{F}$) $\geq c_p$
13: if BestExt exists then
14:     pattGrowthOS($\{P + $BestExt$\}, \varepsilon, \mathcal{F}, PatPool$)
15: else
16:     $PatPool \leftarrow PatPool \cup \{P\}$
17: end if
18: end if
19: end // procedure pattGrowthOS


and even return a number of frequent sequences exponential with the size of input frame $f$, due to the way it explores most subsequences of $f$. The approach used in function getFramePatternsOS, presented in Algorithm 5, is a slight variation, which relaxes the exhaustiveness in search of traditional pattern mining algorithms. Here, instead of choosing a set of possible extensions in line 12, only the extension BestExt leading to the best coverage is retained. It leads to a better coverage than original pattern $P$, then a single recursive call is performed on $\{P + $BestExt$\}$. The consequence is that in getFramePatternsOS, in the worst case the number of frequent sequences examined is $O(|f|^3)$: $Pool_{ms}$ has less than $|f|$ elements, for each of these elements there can’t be more than $|f|$ extensions to check in line 12, and the number of recursive calls to pattGrowthOS it generates is bounded by $|f|$. getFramePatternsOS is thus polynomial in the size of the input frame. This was not the case in getFramePatternsMS, where it was possible to have one recursive call to pattGrowthMS per extension, leading to a worst case complexity of $O(2^{|f|})$. Thus, OneStepOneSon should exhibit better execution times than OneStepMultSon, possibly with a minor degradation of coverage value of the solution. Comparing the performances of NaiveBaseline, RandomBaseline, OneStepMultSon and OneStepOneSon in terms of computation time and of coverage value is the objective of the next section. We will also show the interest of the $k$-most representative block sets obtained on real execution traces.

4. EXPERIMENTS

In this experimental section, our goal is first to evaluate the scalability on large real world traces of the four approaches presented above. For each approach, we will also evaluate the average coverage given by a solution, in order to evaluate the quality of the solution found. We will also show how real traces can be rewritten as sequences of most representative blocks, and show how helpful it can be for application developers.

4.1 Experimental settings

We implemented the algorithms of Section 3 in Python 3. The frequent sequence mining algorithm used is our implementation of ProScan [19]. The experiments were run on an Intel Xeon E5-2650 at 2.0GHz with 32 Gigabytes of RAM with Linux. The parameters of the algorithms are fixed to $k = 10$, $\varepsilon = 75\%$, $m = 2$ and $\ell = 300$. The datasets used are traces from two real applications, described below.

Gstreamer application: Gstreamer [1] is a powerful open source multimedia framework for creating streaming applications, used by several corporations as Intel, Nokia, STMicroelectronics and many others. It is modular, pipeline-based and open source. For our experiments we decoded a movie of 2 hours using Gstreamer on a Linux platform, with the ffmpeg plugin for video decoding. The execution trace obtained has a size of 1 Gigabyte. This trace comprises 131,340 frames, for a total of 5,120,973 events.

DVBTest application: It is a test video decoding application for STMicroelectronics development boards. This application is widely used by STMicroelectronics developers. In our trace, the application is run on a STT298 SoC, which is used in high definition set-top boxes produced by STMicroelectronics. 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. This trace has a size of 1.2 Gigabytes, contains 13,224 frames for a total of 18,208,938 events.

4.2 Comparison of scalability

Fig. 4 reports the wall clock time of the four algorithms presented in Section 3, when varying the number of frames given as input. Each point represents the average of 10 executions. One can notice that both OneStepMultSon and OneStepOneSon are always faster than NaiveBaseline and RandomBaseline. For the GStreamer dataset, both OneStep approaches are one order of magnitude faster than RandomBaseline and two orders of magnitude faster than NaiveBaseline. For the DVBTest dataset, the difference is less important for small number of frames, but quickly jumps to more than one order of magnitude for 5,000 frames. Note that in both datasets, the baseline approaches could not output results for more than 5,000 frames even after more than 10 hours of computation. This comes from the much bigger search space that they have to explore. On the other hand both OneStepMultSon and OneStepOneSon can output results even for the 131,340 frames of the GStreamer dataset within 3 hours. This makes them more suitable for analysis of real traces.
4.3 Comparison of coverage

Fig. 5 shows the coverage of the set of blocks obtained, w.r.t. the number of frames given as input. The first observation from the DVBTest dataset is that the coverage value of the solutions given by all approaches decreases with the number of frames given as input. The reason is that we fixed $k = 10$, which is small and thus prefers blocks that appear in many frames, i.e. with large support. The frames in this dataset tend to have many events with some variety between the frames, especially because of system-level events. For small number of frames, interesting frequent blocks with good coverage can be found. However, with more frames, blocks with very high support tend to have small size and thus bad coverage.

Oppositely, in the GStreamer dataset, there are only application level events, leading to frames with less events and less inter-frame variability. Thus the coverage values for this dataset stay high whatever the number of frames considered. When comparing the approaches, one can notice that the random selection of $PatPool$ in RandomBaseline does not give good results, as this approach has the lowest coverage of all. On the other hand, both OneStep approaches achieve coverage results similar to NaiveBaseline even if they don’t have access to as many candidate blocks. This validates the interest of our iterative greedy algorithm approach.

Last, the OneStepOneSon approach, which generates smaller $PatPool$ than OneStepMultSon, achieves similar coverage results. This is interesting as it means that few well selected patterns in $PatPool$ are enough to allow the greedy algorithm to find a good solution, and that the selection of this pattern can be done with aggressive pruning compared to traditional pattern mining methods. Overall OneStepOneSon is the most interesting tradeoff, as it presents the best computation time and near best coverage values.

4.4 Practical trace analysis

The previous experiments showed that the methods we proposed can scale to real application traces, and allow to find most representative blocks. We now present how such blocks could be of interest for application developers.

A first simple point is information reduction. In the case of the GStreamer dataset, here reduced to its first 100 frames, usually a developer would have to analyze manually or with graphical tools a trace having 3,915 events. Rewriting this trace as a sequence using 10-most representative blocks (10-MRB) extracted by one of our algorithms and special blocks for regions not covered leads to a trace of 320 blocks, which gives a 92% reduction factor.

Such rewriting is easier to represent graphically than the original trace. Consider Fig. 6 which shows a rewriting of the 60 first frames of the GStreamer dataset. The frames are the horizontal lines in the picture. Each frame is composed of blocks represented as rectangles, where each of the 10-MRB has a different shade of grey and the parts of the
frame not covered by blocks are in black. The length of a block corresponds to the number of events of this block. One can notice that most frames have similar number of events, except for some of them having more events. A developer can quickly notice two things with this representation: first, the regular structure of computation of the frames is exhibited by the regular sequencing of blocks across the frames. Especially, the middle and end parts of the frames is very regular and should not require too much attention. Second, some irregularities can quickly be spotted, either by not covered parts of the trace or by MR-blocks that do not appear as often as the others. The developer can quickly check that these irregularities come mostly at the beginning of frames. MR-blocks arising in these irregularities can give good hints of what is going on, and suggest that the irregularities they participate in are not anomalies but more likely operations that do not need to appear in all frames. Not covered sections, on an other hand, may be beneficial for the developer to investigate.

![Figure 6: Global View](image)

Fig. 7 shows a detailed view of the fourth frame, which has an uncovered region at its beginning. The figure shows the frame rewritten with MR-blocks. For convenience, the events have been written inside the blocks on this figure. The developer can quickly identify in the uncovered region a rare call to the function `gst_ffmpegdec_chain::resized`, Such call signal that after receiving new data it was necessary to resize the buffer, an operation usually unneeded as buffers are supposed to be of sufficient size for handling frame data. Memory operations being critical, the developer, without looking at the whole frame, immediately knows that he has to investigate if this buffer resizing caused problems or not.

To summarize, the MR-blocks allow to rewrite the frame as a sequence of blocks of limited size, which is much more manageable that a large sequence of events. Such sequence of blocks can for instance easily be displayed by graphical tools, and shows irregular parts of the traces. The developer can then delve into the analysis of a single frame, and in last resort to the events arising at some point of this frames, allowing to quickly pinpoint possible problems.

5. RELATED WORK

The problem we consider falls in two different domains of Computer Science. The first is Combinatorial Optimisation, as our problem is a variant of the packing problem. This problem is of special use in logistics, for example in order to fill shipment containers with rectangular boxes of different sizes [3]. This problem has no known general algorithm for computing either global or local solutions that we could use in our case. Moreover, in traditional combinatorial optimization settings the elements filling the container (i.e. the blocks in our case) are given as input, whereas in our problem they must be discovered as well.

Our work is closer to Data Mining works where a small set of patterns best describing the data has to be computed. Such patterns are often qualified of “useful”, “descriptive” or “summarizing” patterns. Older approaches are “two-step” approaches that first compute the complete set of frequent patterns, then postprocess this set to compute a small set of descriptive patterns. More recent approaches focus on more efficient “one step” approaches which directly mines the small set of descriptive patterns. Our OneStepMultSon/OneStepOneSon algorithms also follow this approach.

Among existing approaches, high utility pattern mining refers to the discovery of itemsets with “utilities” higher than a user-specific minimum utility threshold, where utility is a numerical value associated to items in input data. Many studies have been proposed for mining high utility pattern sets (HUI) in two step as [8,15,17], but recently some works were proposed to discover HUI without candidate generation. For instance, J. Liu et. al [9] proposed an efficient pruning of the search space based on estimated utility values for itemsets. These techniques focus on itemsets, so they can not be applied in our setting where the sequencing of events matters. Some studies have been done to integrate utility into sequential pattern mining, and the most recent is USpan [18] which defines the problem of mining high utility sequential patterns, but the approach used is a two-step approach, which may have difficulties to scale on very large datasets.

Another trend is to mine “descriptive” or “informative pat-
terns” which are in general small sets of patterns, containing no redundancy, which describe datasets. Among those are approaches that return a set of patterns allowing to compress maximally the dataset according to the MDL principle [14] and [12]. Their approaches is of great practical interest, but compared to our case, the number of desired patterns in output cannot be specified, and as these approaches do not take coverage into account they will typically output much more than the dozen of patterns that a developer accepts to see.

Our work also falls within the more general problem of analysing execution traces. Many existing approaches, such as [5,7,10] need some external information to proceed to trace size reduction or pattern extraction. On the other hand, our approach is purely combinatorial and does not need such information. It is thus better adapted for a first processing of unknown traces where no information are available. Another point is that many approaches [4,7,11] focus on frequent itemsets as patterns, whereas our approach finds frequent sequences. In multimedia application where a strict frequent sequences. In multimedia application where a strict

6. CONCLUSION AND FUTURE WORK

In this paper, we presented the problem of finding a small set of representative blocks of events that can maximally cover an execution trace of a multimedia application. This problem is a variant of the packing problem, a NP-hard problem for which no general algorithm is known. We thus present several greedy approaches, and show experimentally that our best approaches scale well to real application traces up to gigabyte size. We presented a detailed case study on how to analyze a trace with such representative blocks. Our approach allow to drastically reduce the quantity of information a developer has to handle, and is appropriate for graphical visualization. We show with such visualization how a developer can with few operations spot unusual behaviors in the trace, and understand the reason of such behavior. We think that this approach is promising to help application developers in their everyday debugging or optimization tasks.

We have two research directions. The first one is about the labeling of blocks: for now blocks are simply sequences of events, and the developer has to find out himself what is the block about. Integrating some domain knowledge could be better adapted.

7. ACKNOWLEDGMENTS

This work is supported by French FUI project SoCTrace.

8. REFERENCES