Model-based constraints over execution traces to analyze multi-core and real-time systems

Journal:	*Transactions on Modeling and Performance Evaluation of Computing Systems*
Manuscript ID	TOMPECS-2016-0028
Manuscript Type:	Original Manuscript
Date Submitted by the Author:	14-Jun-2016
Complete List of Authors:	Beamonte, Raphaël; Ecole Polytechnique de Montreal, Computer and Software Engineering Department Dagenais, Michel; Ecole Polytechnique de Montreal, Computer and Software Engineering Department
Keywords:	Linux, Real-time systems, Performance analysis, Tracing, Automatic analysis

Z

Model-based constraints over execution traces to analyze multi-core and real-time systems

RAPHAËL BEAMONTE, Polytechnique Montréal
MICHEL R. DAGENAIS, Polytechnique Montréal

As part of the model-driven development methodology, modeling is used to represent the application workflow or automatically generate source code. This methodology is convenient for developers, particularly to create or improve real-time applications embedded in complex mechanical systems.

Multi-core systems are difficult to monitor because of the multiple concurrently running processes that can interfere with each other. When such a system has to be real-time compliant, timing constraints add to the system complexity, invalidating results as soon as a deadline is missed. In such circumstances, tracing is usually the most accurate and reliable tool available to study and understand the behavior of those applications. However, the interpretation of voluminous detailed execution traces requires from its users a deep understanding for the operating system and application behavior, and a long time to dig through the millions of trace events.

In this paper, we present the use of model-based constraints on top of user-space and kernel traces to provide weighted analysis results. We describe our model-based constraints system and how constraints are validated or invalidated. We then introduce how to extract, organize, compare and weight data from the trace. We then explain how we assign responsibilities to unwanted behaviors. Our algorithms have been applied to multiple input traces showing common problems for multi-core real-time systems, while varying the number of trace segment of interest instances, duration of the instances, position of the instances as well as the number of events. The experimental results show that our algorithm can quickly identify many different types of problems, while its runtime stays in the order of a few tens of seconds, even for traces with millions of events, thus helping to save time when analyzing thousands of trace events for complex systems.

CCS Concepts: •**Computer systems organization** → **Real-time system specification;** •**Software and its engineering** → **State systems; Consistency; Software defect analysis;** *Real-time schedulability; Dynamic analysis;* •**General and reference** → Performance; Verification;

Additional Key Words and Phrases: Linux, Real-time systems, Performance analysis, Tracing, Modeling, Automatic analysis

ACM Reference Format:
Raphaël Beamonte, Michel R. Dagenais, 2016. Model-based constraints over execution traces to analyze multi-core and real-time systems *ACM Trans. Model. Perform. Eval. Comput. Syst.* X, Y, Article Z (June 2016), 30 pages.
DOI: 0000001.0000001

1. INTRODUCTION

To quickly diagnose problems, in complex time-sensitive applications, developers usually require system analysis tools. Tracers are particularly interesting, as they do not freeze the application, while still providing a lot of information on its runtime, and the status of the system throughout the traced period. Every tracer has characteristics such as its overhead, intrusiveness, precision level and whether it can trace the kernel, user-space, or both. Yet, all

This research is supported by Ericsson, EfficiOS, and the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant CRD-468687-14.
Author's addresses: R. Beamonte (raphael.beamonte@polymtl.ca) and M. R. Dagenais (michel.dagenais@polymtl.ca) are with the Computer and Software Engineering Department, Polytechnique Montreal, C.P.6079, Station Downtown, Montréal, Québec, Canada, H3C 3A7

DOI: 0000001.0000001

of those tracers require significant and tedious human intervention to analyze the detailed and extensive information contained in traces. Moreover, sufficient expertise to understand those events is also necessary to take advantage of such information.

The workflow of an application, and the quantitative constraints to satisfy, can easily be defined by technical as well as non-technical users through modeling. Modeling is also widely used in the real-time community for formal verification [Aceto et al. 1998]. Using models and traces together allows to automatically check specifications against the real application behavior. These traces take into account the influence of other running applications on an application's workflow. However, uncovering specification violations just provides a general idea of the non-compliance area, and the detailed trace information may be used to further analyze what happened when a constraint is invalid. The automation of this further analysis is the focus of the present work.

This paper describes a new approach to use kernel and user-space traces as well as model-based constraints to automatically identify the origin of an unwanted behavior in applications by means of comparison. It also demonstrates this approach on common problems encountered in real-time and multi-core applications. The analysis results save time by pointing directly to the system activities being the most probable cause, such as the most likely process responsible for preemption, the state in which the application is staying too long, or the system call that should not have happened. Further analysis can then be started automatically to achieve a more precise diagnostic. Our main contribution is to provide an automated analysis of the probable origins of problems. Those are discovered thanks to system-side events such as process preemption, system calls and scheduling.

We present the related work in section 2. The model-based constraints approach is presented in section 3. We then explain our algorithms for data extraction, organization and comparison, as well as how we assign responsibilities in section 4. Results, computation time and scalability for our approach are shown in section 6. Suggestions for future work and the conclusion are in section 7.

2. RELATED WORK

This section presents the related work in the two main areas relevant for this paper, software tracers for both kernel and user-space, and tools for the analysis of traces using model-based constraints.

2.1. Software tracers for both kernel and user-space

In order to extend the checking and analysis of the specifications of an application, we first need to acquire trace data at both the application and operating system levels. It is also necessary to prioritize the low disturbance and high precision of the tracer used to generate those traces. In this section, we thus present the characteristics of currently available software tracers, that provide both user-space and kernel tracing capabilities, with the ability to interface with the Linux kernel `TRACE_EVENT()` macro.

`Perf` [Thornburg 2009] is a built-in Linux kernel tracer. It was at first designed to access the processors' performance counters, but its use has later been extended to interface with the `TRACE_EVENT()` macro and the Linux kernel tracepoints. However, the main orientation of `perf` is for sampling and, even if it is possible to use it as a regular tracer, it has not been optimized for this. Sampling is interesting for high performance systems as it allows for low-overhead, but it loses in accuracy as compared to regular tracing. Besides, an interrupt is at the origin of the collection process, making it invasive and costly. Finally, `perf`'s scalability for multi-core is limited [Desnoyers 2010].

The *Function Tracer* (`ftrace`) is a set of built-in Linux kernel tracers [Edge 2009]. Its primary aim was to determine the bottlenecks in the kernel by monitoring the relative cost of the called functions. Since then, it has evolved to include other analysis modules, such as scheduling or latency analysis [Rostedt 2008]. The `debugfs` pseudo-filesystem allows

to manage `ftrace` and to activate and deactivate its tracers. The *event tracer* of `ftrace` allows the use of the `TRACE_EVENT()` macro [Rostedt 2010]. To save analysis time on the tracer side, `ftrace` only collects data defined in this macro using the `TP_printk` macro. However, this removes the ability to add system context information to trace events. Since Linux kernel 3.5, `ftrace` can also use UProbes to connect to user-space applications. Still, UProbes uses interruptions for its instrumentation, adding unacceptable overhead for high performance and real-time applications.

`SystemTap` [Eigler 2006] is a Linux monitoring system that was created for system administrators. It can use both static and dynamic instrumentation with `TRACE_EVENT()` or KProbes respectively [Krishnakumar 2005]. Since Linux kernel 3.8, it can also instrument user-space applications using UProbes. In either case, the instrumentation is written in a special scripting language, then compiled to a kernel module. Some data processing is done in-flight, inside the instrumentation, and the results can be printed at regular interval. There are no special facilities included to write events to stable storage. Moreover, KProbes as well as UProbes, incur an interrupt for each tracepoint. Like `ftrace`, UProbes being the only option on the user-space side, the interrupt overhead cannot be avoided, which is problematic for real-time and high performance applications.

`LTTng-UST` is the user-space component of `LTTng`. It provides macros to add statically compiled tracepoints to programs. An external process consumes the produced events and writes them to disk. `LTTng-UST` allows the use of arbitrary event types, thanks to the Common Trace Format [Desnoyers 2011]. `LTTng-UST` was designed for high performance, scalability and wait-free properties for event producers, by using per-CPU ring-buffers. Moreover, atomic operations are preferred to locking for the update of the ring-buffer control variables. Read-copy update (RCU) data structures are used to protect important tracing variables, thus avoiding the cache-line exchanges between readers that happen when using read-write lock schemes [McKenney and Walpole 2008; Desnoyers et al. 2012]. The kernel level uses an equivalent architecture. Also, kernel and user-space tracers use the same clock for timestamps, allowing an easy correlation of events across layers at the nanosecond scale. Finally, `LTTng` can trace real-time applications with low latency [Beamonte and Dagenais 2015]. `LTTng` is therefore the best candidate to trace real-time and multi-core applications at both the kernel and user-space levels.

2.2. Automatic data extraction tools and model-checking for traces

In this section, we present different tools and methodologies used for data extraction and model-checking for traces.

`Tango` [Ezust 1995; Ezust and Bochmann 1995] allows to automatically generate backtracking trace analysis tools. Using formal specifications, it will generate tools that are specific to a given model. The tools will then allow to verify that any execution trace complies to the specifications. `Tango` was mainly designed to validate protocol specifications and does not allow to specify constraints based on the system's state. This limitation also prevents from further analysis of a specification invalidation using other resources.

The Logic of Constraints checker [Chen et al. 2004] also allows to generate an executable checker from formulas written in a formal quantitative constraint language. An evaluation report is generated by running this executable on simulation traces. This report will inform of any constraint violation. This tool is, however, using text-format traces, making it very sensitive to any change in the trace format. Moreover, no further analysis than the constraint verification is offered.

`Scalasca` [Geimer et al. 2010] provides a way to identify bottlenecks using execution traces. It allows to analyze aggregated statistical runtime summaries to identify which process consumes CPU time and how much. An analysis of event traces is also available, and used for a deeper study of the concurrency of programs. The traces are used to identify wait states and linked performance properties. `Scalasca` then generates a pattern-analysis

report containing performance analysis metrics for each function and system resource. Yet, `Scalasca` does not allow to provide our own specifications.

`SETAF` [Peiris and Hill 2012] is a framework that allows to add properties for the analysis and validation of the QoS in system execution traces and dataflow models. In its workflow, `SETAF` requires the user to manually analyze the execution trace to identify what are the missing properties, and thus provide the related adaptation patterns. This pattern will then be used to add those properties and create the necessary causality relations. By this workflow, `SETAF` requires the user to have a deep understanding of the trace format. Also, no further analysis than the specifications invalidation is provided.

`Trace Compass` [The Eclipse Foundation 2014] is an `Eclipse` graphical interface for `LTTng`. It provides multiple views to show specific analysis of the traces. Some of these views are of particular interest for our research, such as the analysis for real-time applications [Rajotte and Dagenais 2014] and the analysis of the system-level critical path of applications [Giraldeau 2015]. The later allows to recover segments of execution that affect the waiting time of a given computation. `Trace Compass` also support different trace formats. Finally, it provides methods to create state system attribute trees, and to store metrics that vary along the time axis in the State History Tree database. This database allows to query efficiently the modeled state, for any given point in time.

To our knowledge, existing model analysis tools do not exploit all the available information and limit their analysis to the detection of invalidated constraints. Using model analysis and trace analysis tools, and by applying statistical methods and algorithms, we take advantage of this unused information to provide more insight to the user. This not only provides further details on the invalidated constraints, but also possible reasons explaining those invalidations.

2.3. Statistics algorithms for relation analysis

In order to take advantage of the available data, we based our work on common methods and algorithms used to compare and associate data. Those are reviewed in this section.

The edit distance is a method to quantify how similar or dissimilar two entities are to one another, such as strings [Ristad and Yianilos 1998] or trees [Bille 2005]. To do so, it counts the minimum number of operations required to transform one entity into the other, amongst a defined limited set of operations. Multiple implementations exist that differ only in the set of operations. For instance, the Levenshtein distance allows the removal or insertion of a single element, as well as the substitution of one element for another.

The *association rule learning* is a data mining method that aims to find interesting relations between variables in large databases. These relations are identified using different mesures of interestingness [Piatetsky-Shapiro 1991]. Association rules were first introduced to discover consumers regularities in supermarkets purchases [Agrawal et al. 1993]. Such application would then allow to identify that when a consumer buys items A and B, there is a high chance that he will buy an item C, and thus help in making decisions for products placement for instance. There are multiple algorithms implementing association rule learning using different approaches. One of the most used is the Apriori algorithm, using a breadth-first search to identify the frequent individual items, before extending them to larger sets, until these item sets do not appear often enough anymore [Agrawal and Srikant 1994].

3. USING MODEL-BASED CONSTRAINTS

The developers usually have specific expectations when designing high performance applications. They know in which order the different operations should be performed, and have estimates for the values of different metrics and their evolution during the application runtime. With these expected values, a developer can check that the application is behaving as planned, and thus adjust accordingly the length of the debugging and tuning period.

Model-based constraints have been used in the past to specify and validate the workflow of an application or system using different metrics [Dasarathy 1985]. This has been extended with success in previous work to include constraints based on more sophisticated metrics, extracted from operating system tracing [Beamonte and Dagenais 2016]. This section provides an overview of such constraints, which is the basis for our work on automatic problem identification analysis.

3.1. Internal structure

The representation used is based on four different elements: the states, representing the multiple states of the application, the transitions, representing the movement from one state to itself or to another state, the variables, used to get and store the value of the metrics we aim to verify, and the constraints, used to express the specification of what we expect for the application execution.

The modeled state of the operating system and applications is maintained in an attribute tree called the state system. The state system is based on the state attribute tree of `Trace Compass`, for which the state history database is built to contain the different metrics needed for access later during the analysis. The values for those metrics are stored during the first reading of the kernel trace and can be accessed later by querying the state history database.

We distinguish three main categories of variables used to verify constraints: variables not dependent on the current state (state system free) , the counter variables – or counters – and the timer variables – or timers. The variables are categorized as such depending on the number of queries needed in the state system to compute their next value. It is necessary to make respectively 0, 1 and 2 queries to the state history database to compute the value of a state system free variable, counter variable and timer variables at a given time.

Each constraint is linked to a transition and is being checked when this transition happens. The constraint will then either be valid, invalid, or uncertain if data is missing to check that constraint. The linked transition will then be valid if all its constraints are valid, invalid if at least one constraint is invalid, or uncertain if at least one constraint is uncertain and none are invalid.

The succession of states and transitions is finally used to follow the multiple instances of the application that will be checked at each step, i.e. when we use a transition to move to a next state.

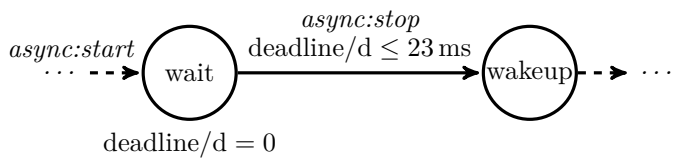
3.2. Model representation

Figure 1(a) shows the representation of a part of a state machine, for an application where we would want to verify some metrics. The two states of the application in the Figure are called "wait" and "wakeup". *wait* represents the state that starts the period to check, and *wakeup* represents the state that ends that period. To follow the workflow of the application, user-space events are used: "async:start" is the event received that will signal us to enter state *wait*, while "async:stop" will signal to move from state *wait* to enter state *wakeup*.

The initializations are all the different variables initializations that will appear at a given state, to allow the future verification of our metrics. A variable is initialized by being set to 0. That initialization will create the given variable of the given type to be checked later in a constraint. For a variable d of type *deadline*, the initialization would look like the following: "deadline/d = 0", as shown in the Figure.

The verifications are all the constraints that will be checked when following that transition. A constraint for a variable d of type *deadline*, that needs to be less or equal to a value *23ms*, would look like the following: "deadline/d $\leq$ 23 ms", as shown in the Figure.

Figure 1(b) shows a State Chart XML representation of the same section, but considering only that section as entry and exit points. Such files can automatically be used by the analysis application to build the state machine and its constraints.



(a) Graphical representation of the period of `JACK2`

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0">
    <initial>
        <transition event="async:start" target="wait"/>
    </initial>

    <state id="wait">
        <onentry>
            <assign location="deadline/d" expr="0"/>
        </onentry>

        <transition event="async:stop" target="wakeup" cond="deadline/d &lt;=
            ↪  23ms"/>
    </state>

    <state id="wakeup">
    </state>
</scxml>
```

(b) XML representation of the critical real-time task of `tk-preempt` using State Chart XML

Fig. 1: State machine representation of `tk-preempt`'s real-time process using constraints to check that our process was not preempted during its critical real-time task

4. ALGORITHMS FOR DATA ORGANIZATION AND EXTRACTION

Once the instances of matched patterns of the application have been built from the traces, using the process described in 3, where the different constraints were verified simultaneously, we end up with information about constraints validation, uncertainty or invalidation for each step. Even if that information allows us to pinpoint the location (tracepoint location and time) of a given problem, for instance a missed deadline or a preemption, it does not necessarily reveal the origin of that problem. To know the origin of the problem, someone would thus have a given location and time point to look at (tracepoints), and would also know the kind of unexpected behavior (constraint violation) to look for, but the subsequent investigation work would still have to be done by hand. Moreover, understanding the problem origin may require a good understanding of the application and of the operating system internal behavior.

As we work towards automating the developer's work to save time and allow more informed decisions, this section presents our approach to organize the data, resulting from that previous analysis, to perform a more thorough analysis and extract more informative results. We first present the general data organization, before explaining the extraction process and the algorithms used.

4.1. Data organization

Data analysis is achieved by looking at what happened when encountering unexpected situations, and comparing it with what normally happens, identifying what is common in those unexpected cases. In the context of traces, this means identifying events or lists of

events that can be significant, and in some cases comparing them with events or lists of events that we expected to appear, in order to highlight the discrepancies.

When a trace has been analyzed using the model-based constraints checking described earlier, we get a list of instances of the application, according to the workflow defined in the model used. These instances come along with the list of all the states encountered, and the events that brought us there; these are called instances steps. These steps contain themselves the list of all the constraints that have been verified when arriving there, the validation status for each of those constraints, as well as the step at which the verified variables were last initialized.

In order to obtain our list of significant elements, we first need to separate the validated cases from the invalidated ones, for each constraint. The cases for which the validation is considered uncertain are not considered, as we cannot be sure of the actual outcome for the concerned period. Our aim is to build, for each constraint that has been invalid at least once, two lists containing respectively all the invalid and valid cases for this constraint. We thus browse the constraints, for each instance step of each instance, and store the constraints validation information in either the valid or invalid list. This means that an instance that has been invalid for one constraint can still be used as a valid instance for another constraint. Once the data is organized in this way, we run a variable-specific analysis, corresponding to the type and category of the variable used in the constraint, for all the constraints having at least one invalid occurrence.

This analysis will receive both the valid and invalid lists of instances, that can then be used for the identification of significant elements to compare.

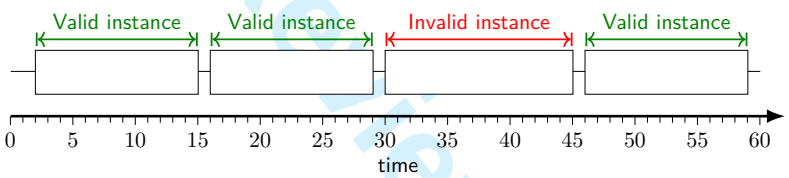


Fig. 2: Example of a trace containing 4 instances, following a model specifying that the instance execution duration should be of 13 units of time; 3 of the instances are valid, and 1 is invalid with a duration of 15 units of time.

Figure 2 gives an example for which we would end up with 3 instances in the list of valid instances, and 1 instance in the list of invalid ones.

4.2. Data extraction

The data extraction process follows multiple stages for each variable that has to be analyzed. These stages are the instances selection, the extraction of elements of interest from these instances, and the assignation of responsibilities.

The work done in some of these stages depends on the algorithm chosen to do the analysis. Two algorithms are available, the partial analysis algorithm and the full analysis algorithm. The algorithm choice is based on the kind and value of the constraint to analyze. If our constraint is absolute, meaning that any change to the variable value during the analyzed period of time is prohibited, we will chose the partial analysis to directly compute responsibilities for each element leading to a variable value change. However, if changes to the variable value are authorized, a full analysis must be done in order to compare what is happening during valid and invalid instances, extract the discrepancies and then compute responsibilities. For instance, when analyzing a counter, if the constraint is not to have any increment to this counter, any increment to that value is responsible for the constraint violation. However, if we allow two increments to that counter but in some situations we end

up with three increments, it is not clear which increment is responsible for the violation. Different computations are therefore needed for these different cases, hence the two analysis processes.

4.2.1. Selection of the instances. A large number of instances can be found in a trace, depending on the trace size and the model duration. Finding the source of violated constraints for a large number of instances can thus take a time linearly proportional to the number of instances. This can therefore impact the scalability of our analysis.

However, when the instances are regrouped as valid or invalid for a constraint, they will most likely follow a finite number of different patterns according to the elements of interest that we will next extract. This is because, for a real-time application, the tasks that have to be done are highly likely to be similar. This means that instead of checking all of the valid and invalid instances, we could determine a sample big enough in each case to consider that the different patterns are represented. This sample would then allow us to perform a much faster analysis with minimal precision loss.

A correct sample size for a given list of instances can be determined by using the following formula, which takes into account the finite population correction [Suresh and Chandrashekara 2012]:

$$S = \left\lceil \frac{\frac{z^2 \times p(1-p)}{e^2}}{1 + \left(\frac{z^2 \times p(1-p)}{e^2 N}\right)} \right\rceil \tag{1}$$

Where S is the computed sample size, N is the population size, e the margin of error, p the initial probability and z the z-score corresponding to the expected confidence level. In our case, the population size is the number of instances in the list in which we will select the sample. We set the initial probability as 50 %, which is the worst case scenario, leading to the biggest sample size. Finally, the margin of error and confidence level follow the industry standard of respectively 5 % and 95 %, which corresponds to a z-score of 1.96.

Once we determined the sample size, we need to determine which instances will compose the tested sample. In order for our results to be statistically correct, we need for the instances to each have an equal chance to be part of the sample, which means random sampling. Among the methods for random sample selection, we chose to use *reservoir sampling*, which allows the equality of chances for each member of the population to be in the selected sample. Reservoir sampling is originally derived from the Fisher-Yates shuffle [Fisher and Yates 1949] and exists in multiple implementations of different complexity [Vitter 1985; Li 1994]. The one we use in our analysis is the efficient algorithm for sequential random sampling [Vitter 1987], which allows for fastest sample selection on big data sets.

4.2.2. Extraction of elements of interest. Once the instances to analyze are selected, we need to identify the key elements that will provide us with the source of the invalid constraint. We call these the elements of interest, or elements. They differ depending on the kind of variable used for the constraint.

Counter and timer variables directly follow values stored in our state system. This allows us to directly request the timestamps of the variable's value changes along the duration of the instance under consideration. Once these timestamps are extracted, we can directly read from the trace the events that triggered those value changes, and extract the data needed. For instance, the extracted data could be the name of the system call for a system calls counter, or the status of the process being unscheduled for a CPU usage timer.

The extraction process for state system free variables depends on all that can influence that variable. For instance, in the case of the deadline variable, we created a state system attribute that follows the state of the process throughout the trace. Then, each change of the process state during the analyzed period is extracted as being of interest.

Independently of the constraint, elements are then stored as element durations. Element durations contain the element, and the exact value increment caused by the appearance of that element at that moment, for the variable used by the constraint. The element is thus the duration key, while the increment is the duration value. All the durations found during a period of time are then merged in an element duration set. Element duration sets thus contain a number of durations that may or not be about the same element. A duration set can be empty, which means that during the analyzed period, no element was found.

$$[a, b, c] = [b, c, a] = [c, a, b] = [c, b, a] = [a, c, b] = [b, a, c]$$

Fig. 3: When merging duration sets, only the content is considered and not the order, which means that all the *keymaps* presented here are considered identical

For a given list of instances, similar element duration sets are then merged into element interval sets. We consider that two element duration sets are similar if they share the same *keymap*, i.e. the same number of durations about the same elements. This means that when merging duration sets into interval sets, we only use their content without considering the order of appearance. For instance, two duration sets containing only one occurrence of each a, b and c, no matter the order, will be merged in the same interval set, as shown in Figure 3. The interval sets will therefore share the same *keymap* as the duration sets that were used to compose them. It will moreover have minimum and maximum durations (for timers and state system free variables) or minimum and maximum values (for counters).

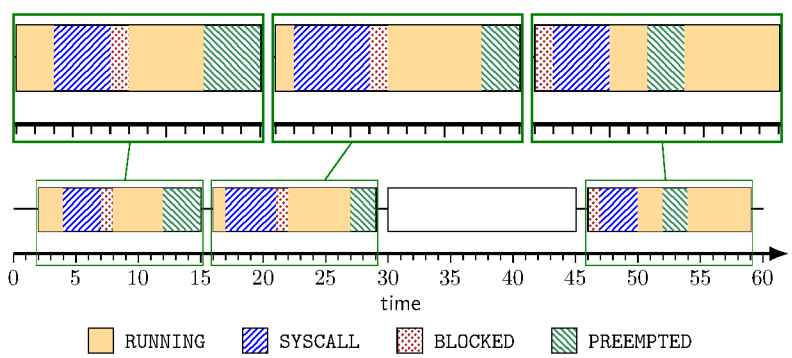


Fig. 4: Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the valid instances which are in equal number for each element, but with different order or duration.

Returning to the example shown in Figure 2, we could consider a small real-time application which shares its time between computation, requesting a resource via a system call, and being preempted by a higher priority task. In such case, Figure 4 shows the content of the different valid instances and Figure 5 the different duration sets extracted from these instances. We can see that even if the order and duration for each element is different from an instance to another, we still have two occurrences of element `RUNNING` and one occurrence of each element `SYSCALL`, `BLOCKED` and `PREEMPTED`. The duration sets' *keymap* of each valid instance thus being the same, the duration sets of theses instances will be merged into one unique valid interval set.

```
DurationSet: [
    [RUNNING, 2, 6], // durations 2 and 4
    [SYSCALL, 1, 3],
    [BLOCKED, 1, 1],
    [PREEMPTED, 1, 3],
]
```

(a) The duration set of the first valid instance contains two durations for the element `RUNNING` that can be summed to 6 units of time, one duration for the element `SYSCALL` for 3 units of time, one duration for the element `BLOCKED` for 1 units of time and one duration for the element `PREEMPTED` for 3 units of time

```
DurationSet: [
    [RUNNING, 2, 6], // durations 1 and 5
    [SYSCALL, 1, 4],
    [BLOCKED, 1, 1],
    [PREEMPTED, 1, 2],
]
```

(b) The duration set of the second valid instance contains two durations for the element `RUNNING` that can be summed to 6 units of time, one duration for the element `SYSCALL` for 4 units of time, one duration for the element `BLOCKED` for 1 units of time and one duration for the element `PREEMPTED` for 2 units of time

```
DurationSet: [
    [RUNNING, 2, 7], // durations 2 and 5
    [SYSCALL, 1, 3],
    [BLOCKED, 1, 1],
    [PREEMPTED, 1, 2],
]
```

(c) The duration set of the third valid instance contains two durations for the element `RUNNING` that can be summed to 7 units of time, one duration for the element `SYSCALL` for 3 units of time, one duration for the element `BLOCKED` for 1 units of time and one duration for the element `PREEMPTED` for 2 units of time

Fig. 5: The three duration sets extracted from the three valid instances presented in Figure 4; the durations contained in the set are shown in the simplified form of the number of durations for a given element and the sum of these durations

When creating an interval set from a duration set, we obtain an interval set containing a number of intervals with identical minimum and maximum values. When merging a duration set into an interval set, the element durations of the duration sets will be matched with the element intervals of the interval set in order to keep the resulting intervals as short as possible. For instance, if for an element `RUNNING` we have two intervals $[1, 2]$ and $[4, 5]$ and two durations 6 and 2 to merge into, the 2 will be merged in the interval $[1, 2]$ and the 6 in the interval $[4, 5]$. Doing so, the second interval will only become larger by 1, while the first interval will stay untouched.

Considering this whole process, we can deduce that if the duration set used to create an interval set was empty, the interval set will also be empty.

Figure 6 represents the valid interval set of our example, shown in the simplified form of the number of intervals about a given element, and the minimum and maximum values of the smaller interval containing all of the intervals for the element. We can observe that its *keymap* is the same as the duration sets' one, and that the different durations have been merged into the smallest intervals.

4.2.3. Assignation of responsibilities. The final stage of the analysis is to assign responsibilities to the elements extracted in order to identify clearly which element or elements have the most to do with the constraint violation.

When using the partial analysis algorithm, each and every element extracted is partly responsible, which means that all these have to be used to assign responsibilities. If we consider I to be the extracted invalid interval set and $I_{j,vmin}$ and $I_{j,vmax}$ respectively the minimum value and maximum value of the interval for the element j, then the minimum

```
IntervalSet: [
  [RUNNING, 2, [1, 5]], // [1, 2]&[4, 5]
  [SYSCALL, 1, [3, 4]],
  [BLOCKED, 1, [1, 1]],
  [PREEMPTED, 1, [2, 3]],
]
```

Fig. 6: Interval set resulting from the merge of the three duration sets in Figure 5; it contains two intervals for the element `RUNNING` that are both between 1 and 5 units of time, one interval for the element `SYSCALL` between 3 and 4 units of time, one interval for the element `BLOCKED` for 1 unit of time and one interval for the element `PREEMPTED` for 2 to 3 units of time

```
DurationSet: [
  [RUNNING, 2, 4], // durations 1 and 3
  [SYSCALL, 1, 4],
  [PREEMPTED, 1, 2],
  [IRQ, 2, 5], // durations 2 and 3
]
```

Fig. 7: The duration set extracted from the invalid instance presented in Figure 9; it contains two durations for the element `RUNNING` that can be summed to 4 units of time, one duration for the element `SYSCALL` for 4 units of time, one duration for the element `PREEMPTED` for 2 units of time and two duration for the element `IRQ` that can be summed to 5 units of time

responsibility $r_{e,vmin}$ and maximum responsibility $r_{e,vmax}$ of the element e in the constraint violation can be formulated this way:

$$r_{e,vmin} = \begin{cases} \dfrac{I_{e,vmin}}{\sum_{j \in I} I_{j,vmin}} & \text{if } \sum_{j \in I} I_{j,vmin} \neq 0 \\ r_{e,vmax} & \text{if } \sum_{j \in I} I_{j,vmax} \neq 0 \\ \dfrac{1}{\#I} & \text{else} \end{cases} \tag{2}$$

$$r_{e,vmax} = \begin{cases} \dfrac{I_{e,vmax}}{\sum_{j \in I} I_{j,vmax}} & \text{if } \sum_{j \in I} I_{j,vmax} \neq 0 \\ r_{e,vmin} & \text{if } \sum_{j \in I} I_{j,vmin} \neq 0 \\ \dfrac{1}{\#I} & \text{else} \end{cases} \tag{3}$$

Which makes it possible to compute the mean responsibility r_e for element e, which is the one that will be shown to the user, as follows:

$$r_e = \frac{r_{e,vmin} + r_{e,vmax}}{2} \tag{4}$$

However, when using the full analysis algorithm, some of the extracted elements are expected to be there. The situation is thus more complex to analyze, since we cannot directly consider each element as incorrect, and responsible for the constraint violation. We thus need to identify the elements that are not there in valid instances, but appear in invalid ones.

To do that, we first select the valid instances to analyze and perform the extraction of elements of interest. As explained in 4.2.2, this stage leaves us with some element interval sets, each with different *keymaps*, called thereafter valid element interval sets. These are the baseline that will be used for comparison purposes.

We do the same thing with the selected invalid instances, from which we obtain a number of invalid interval sets. For each of these interval sets, we need to identify the closest valid interval sets in order to get the most accurate comparison. This is done by calculating, for each pair of valid and invalid interval sets, a weight that takes into account two distinct metrics: the distance between the interval sets in consideration and the number of occurrences of the valid one. We consider that the valid interval set closest to the invalid one has the highest probability of being the desired behavior, instead of the invalid behavior that lead to the violation. Furthermore, the probability increases with the number of occurrences of that valid interval set.

Distance between interval sets. The distance between the current invalid interval set and a given valid interval set i is noted d_i and represents the differences in the invalid interval set that separate it from the valid one. The distance d_i is computed as being the average of the distances from the duration sets that compose the invalid interval set to the valid interval set.

$[a, a, b, c]$	to $[a, a, b]$	$(d = 1)$	remove c
	to $[a, a]$	$(d = 2)$	remove b
	to $[a]$	$(d = 3)$	remove a
	to $[\,]$	$(d = 4)$	remove a
	to $[e]$	$(d = 5)$	insert e

Fig. 8: Distance calculation step by step, showing the distance for intermediate sets

The distance calculation is largely based on an edit distance that only takes into account additions and deletions, as the order is not considered. An example of such approach, and intermediate distances when moving from a set of non unique elements to another, is given in Figure 8. We however consider the directional component of the analyzed constraint: when an element appears in the valid interval set but is not in the invalid one, if the constraint was for the value to be less than a given limit, this element will not add to the distance. Respectively, if the constraint was for the value to be higher than a limit, only the elements appearing in the valid interval set and not in the invalid duration set will add to the distance. Finally, when the constraint is for the value to be equal or different than a given target, all the discrepancies have to add to the computed distance.

Algorithm 1 gives the pseudocode of the algorithm used to compute the distance between a single element duration and a single element interval. We can see that the directional component is taken into account while computing the distance.

Algorithm 2 uses Algorithm 1 in order to compute the full distance between a distance set and an interval set. The `closestPermutation` function called in Algorithm 2 returns the list of durations and interval in an order that allows for the shortest cumulated distance. This calculation can easily end up being of combinatorial complexity, when trying to select the x elements that will generate the smallest distance among y available elements. As we consider a statistical approach, another sample is selected randomly here, using the same process as used for the selection of instances. This allows to keep the algorithm scalable. The size of the returned distance duration set is considered as being the distance value.

Following on our example, we can see the content of the invalid instance in Figure 9. This instance contains occurrences of the element `IRQ` that was not existent in valid instances, but does not contain any occurrence of element `BLOCKED` that was in valid instances. Moreover, we can see in the extracted duration set presented in Figure 7 that the number and values

ALGORITHM 1: Distance calculation between an *interval* and a *duration*, taking into account the direction of the constraint C, the result being a numerical value

```
function distance(duration, interval, C)
    d ← duration.value()
    i_min ← interval.minValue()
    i_max ← interval.maxValue()
    if d ≥ i_min and d ≤ i_max then
        return 0
    end
    if C in [<, ≤] then
        return max(d − i_max, 0)
    else if C in [=, ≠] then
        return max(i_min − d, 0)
    end
    return min(|i_min − d|, |d − i_max|)
end
```

ALGORITHM 2: Distance calculation between an interval set *intSet* and a duration set *durSet*, taking into account the direction of the constraint C, the result being a duration set representing the distance per element

```
function distanceSet(durSet, intSet, C)
    distSet ← new duration set
    forall element in durSet do
        D ←
          durSet.getDurations(element)
        I ← intSet.getIntervals(element)
        if isEmpty(I) then
            if C in [=, ≠, <, ≤] then
                distSet.addAll(D)
            end
            continue
        end
        D, I ← closestPermutation(D, I)
        min ← minSize(D, I)
        forall i in 0..min do
            dist ← distance(D, I, C)
            if dist > 0 then
                distSet.add(element, dist)
            end
        end
        if C in [=, ≠, <, ≤] then
            s ← size(D)
            distSet.addRange(D, min..s)
        end
        if C in [=, ≠, >, ≥] then
            s ← size(I)
            distSet.addRange(I, min..s)
        end
    end
    return distSet
end
```

ALGORITHM 3: Substraction of a valid interval set *validIntSet* from an invalid one *invalidIntSet*, taking into account the direction of the constraint C, the result being a differential interval set

```
function substract(invalidIntSet, validIntSet,
 C)
    DIntSet ← new differential interval set
    forall element in invalidIntSet do
        I ← invalidIntSet.getIntervals(element)
        V ← validIntSet.getIntervals(element)
        N_I ← size(I)
        min_I ← sumMinValueOfAll(I)
        max_I ← sumMaxValueOfAll(I)
        if isEmpty(V) then
            if C in [=, ≠, <, ≤] then
                DIntSet.addInterval(N_I, min_I,
                  max_I)
            end
            continue
        end
        N_D ← N_I− size(V)
        min_D ← min_I− sumMinValueOfAll(V)
        max_D ← max_I− sumMaxValueOfAll(V)
        if C in [<, ≤] then
            N_D ← max(0, N_D)
            min_D ← max(0, min_D)
            max_D ← max(0, max_D)
        else if C in [>, ≥] then
            N_D ← max(0, −N_D)
            min_D ← max(0, −min_D)
            max_D ← max(0, −max_D)
        end
        if N_D ≠ 0 or min_D ≠ 0 or max_D ≠ 0 then
            min_D, max_D ← minMax(min_D, max_D)
            DIntSet.addInterval(N_D, min_D,
              max_D)
        end
    end
    if C in [=, ≠, >, ≥] then
        forall element in validIntSet do
            if element in DIntSet then
                continue
            end
            V ←
              validIntSet.getIntervals(element)
            N_V ← − size(V)
            min_V ← − sumMinValueOfAll(V)
            max_V ← − sumMaxValueOfAll(V)
            DIntSet.addInterval(N_V, min_V,
              max_V)
        end
    end
    return DIntSet
end
```

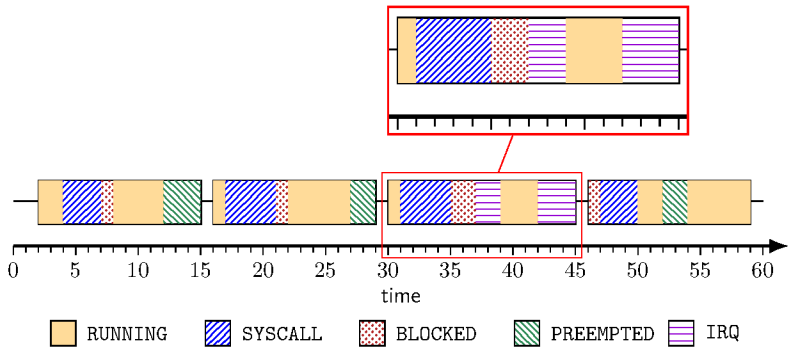


Fig. 9: Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the invalid instance, that differ in number or in duration when compared to valid instances.

of the durations for the common elements are also different between this invalid instance and the valid ones seen in Figure 5.

Table I: Computed distance sets and values between the invalid duration set represented in Figure 7 and the valid interval set in Figure 6 for different constraints on a target value x

Constraint	Distance	
	Duration set	Value
$= x$ or $\neq x$	`DurationSet: [ [RUNNING, 1, 1], [BLOCKED, 1, 1], [PREEMPTED, 1, 2], [IRQ, 2, 5], // 2 and 3 ]`	5
$< x$ or $\leq x$	`DurationSet: [ [BLOCKED, 1, 1], [IRQ, 2, 5], // 2 and 3 ]`	3
$> x$ or $\geq x$	`DurationSet: [ [RUNNING, 1, 1], [PREEMPTED, 1, 2], ]`	2

Table I shows the different computed distances for different constraints that could be encountered in the form of both the distance duration set and the resulting distance value. The case "$> x$ or $\geq x$" is shown for the purpose of highlighting the consideration of the directional element, as we would not end up verifying such constraint while having valid instances not validating the constraint themselves. We can see for the different cases that the distance value does not take into account the value of each contained element duration. The distance value is actually the number of durations in the distance duration set, which means that two different distance duration sets could lead to the same distance value. The different computed distance duration sets also show that our algorithms' output adapt correctly to the directional element; only the relevant elements add to the distance.

Weight calculation. Once we know the distance between our interval sets, we compute a weight that aims to reflect the chances for the valid interval set to be the one that should have happenned instead of our invalid one. We call W_{ri} the relative weight of the valid interval set i for our currently analyzed invalid interval set. To put emphasis on the distance, over the number of occurrences, we chose to use the following formula:

$$W_{ri} = \frac{O_i}{\sum_{d_j \leq d_i} O_j} \times \frac{d_i}{\max(1, s)} + \frac{s - d_i}{\max(1, s)} \tag{5}$$

Where O_i is the number of occurrences of the valid interval set i, d_i the distance between this interval set and the invalid one, $\sum_{d_j \leq d_i} O_j$ the sum of occurrences of all the valid interval sets that have a distance lower or equal to d_i, and s the size – or number of intervals – of the invalid interval set.

However, this formula does not take into account the uncertainty of the results, that grows with the number of different valid interval sets available. We defined the maximum uncertainty as the confidence factor F_C:

$$F_C = 0.1 \tag{6}$$

This factor thus represents the maximum penalty that will be substracted from our weight results. The penalty P is defined by the following formula:

$$P = F_C \times \left(1 - \frac{1}{N_{valid}}\right) \tag{7}$$

Where N_{valid} is the number of different valid interval sets that have been found during the analysis. The penalty P will thus be null when there is only one valid interval set found, and tend to F_C while the number of valid interval sets grows. By substracting that penalty from our relative weight W_{ri}, we calculate the weight W_i:

$$W_i = W_{ri} - P \tag{8}$$

If a valid interval set ends up with a weight W_i under 0, this set will simply be dropped. The growing penalty therefore allows to automatically exclude interval sets with a relative weight W_{ri} under the value of P, i.e. with probabilities too low given the number of valid interval sets.

Differential interval set. After weighting the valid interval sets against an invalid one, the next step is to identify the discrepancies. For this purpose, we will compute a differential interval set for each valid interval set with a weight greater of equal to 50 %. If no such valid interval set is found, the valid interval set with the highest weight will be the only one considered. The local differential interval set is computed by substracting the valid interval set from the invalid one. This is done by comparing the intervals in both sets for each element, and computing new intervals that embody the difference in terms of number of intervals, and minimum and maximum value, of all the intervals. Algorithn 3 gives the pseudocode of the `substract` function used for this end. We can see that the resulting differential interval set takes into account the directional component of the constraint.

When there is more than one computed differential interval set, i.e. when we have more than one valid interval set with a weight greater or equal to 50 %, we merge them in order to get only one differential interval set that represents the smallest discrepancies between the invalid interval set and the closest valid interval sets. The merge process is

called `interUnion`, as it does both an intersection on the contained elements, and an union on the interval values. The intersection part of the process allows to eliminate elements that could be considered discrepancies while looking at one valid interval set, but are actually present in another one. The union process allows to widen the values of the interval to the smallest interval containing all the differential intervals for the given element.

Finally, we can use the equation (4) to identify the responsibility r_e of each element e in our differential interval set. If multiple invalid instances sets were identified, i.e. invalid instances sets with different *keymaps*, the computed responsibilities for each element in each invalid case are merged to form the global responsibilities. For instance, consider that we have two invalid cases. The computed responsibilities for the first case are of 80 % for element a, 15 % for element b and 5 % for element c. However, the responsibilities for the second case are of 50 % for element a, 40 % for element b and 10 % for element d. Therefore, the final merged responsibilities will be of 65 % for element a, 27.5 % for element b, 2.5 % for element c and 5 % for element d.

4.2.4. Diagnostic. For counters and timers, the data extraction process will commonly give the user enough information to understand what happened in the system. However, when working with state system free variables, the origin of the problem is usually not as clear. If we take as example a deadline constraint, once the analysis has been done on the process' state, the underlying problem, when the process is spending too much time being blocked on a system call, or even running, is not necessarily straightforward.

In such cases, other specific analysis can be run, that in turn can run other analysis in order to pinpoint as precisely as possible the origin of the problem. In order to consider only the elements with the highest probabilities for further analysis, an element will be considered only if its responsability is greater or equal to the minimum responsibility for consideration R_{min} as expressed in (9).

$$R_{min} = \max_{e \in r} r_e - \sigma\left(\forall_{e \in r} r_e\right) \tag{9}$$

Among those analysis, we can count the critical path analysis that allows to retrieve the critical path of the process in the system during a period of time, for which the same data extraction and comparison processes were done. Section 5 presents multiple analysis cases and the different analysis that were automatically triggered in order to give a thorough report to the user.

5. CASE STUDIES

This section presents different case studies of common real-time problems on which we evaluated our automated analysis approach to pinpoint the origin of the problem encountered.

5.1. Too low priority

5.1.1. Situation. `JACK2` is the `C++` version of the JACK Audio Connection Kit sound server. This application is real-time and can run with a periodic workload. When started, `JACK2` configures the ALSA driver to sample at a configurable frequency and to accumulate a specific number of frames before raising an interrupt. That whole period can thus be identified and instrumented. Tracepoints were added around the working period of the application, allowing to identify in the trace when `JACK2` started to wait to be awakened, and when it actually woke up.

`JACK2` already detects when its period is not met, which is called an *xrun* and generates a log message when this situation happens. This report is useful as it provides a way to validate the results we get from analyzing a trace of the running application.

Taking advantage of the way `JACK2` works, we simulated a priority problem as was done in [Desfossez 2016]. We first started the `JACK2` daemon with a low real-time latency priority,

```
$ taskset -c 0 jackd -P 1 -v -d alsa -H
...
creating alsa driver ... hw:0|hw:0|1024|2|48000|0|0|hwmon|swmeter|-|32bit
configuring for 48000Hz, period = 1024 frames (21.3 ms), buffer = 2 periods
...
```

Fig. 10: `ALSA` driver being configured by the `JACK2` daemon process after it was started with a low real-time priority of 1 and pinned on CPU 0 [Desfossez 2016]

and we pinned it on a given CPU, as shown in Figure 10. We can see that the `ALSA` driver is being configured to sample at 48 kHz and accumulate 1024 frames. This period has an expected runtime of 21.3 ms. We then ran a CPU-intensive task on CPU 0, with a higher prority, in order to disrupt the period of `JACK2`. Once that task stopped, `JACK2` reported an *xrun*, meaning the application was indeed preempted for a sufficient time to miss its period.

Using the tracepoints reached before and after the period, and the duration of that period, we can easily build an analysis model for that case, as presented in Figure 1. The deadline constraint is for a period having a duration of less than 23 ms to take into account the waking time of the process. We then used that model to perform an analysis of a trace during which `JACK2` reported an *xrun* of "at least" 353.963 ms.

5.1.2. Automatic analysis. Figure 11 presents details of the work performed to identify the source of the problem using a process state analysis. Figure 11(a) shows an invalid interval set, which represents the intervals of time taken by different process states during the analyzed invalid instances. In our case, this set contains the data of only one instance, hence the interval having the same duration for both minimum and maximum. Figure 11(b) shows the weighted results between the invalid interval set presented in Figure 11(a) and the different valid interval sets computed using valid instances. The results show both the distance and occurrences of each valid interval set that were used to compute its weight against our invalid one. Figure 11(c) finally shows the computed difference after the weighting step: each valid interval set with a weight greater than or equal to 50 % is substracted from the invalid one, generating an interval set of the states in excess in the invalid interval set.

The intervals of time added by each process state are then computed as a percentage of the total time added by all of the process states in the difference interval set. Figure 12(a) thus shows these results. We can see that the prominent state is *WAKING*, meaning that our application spent 96.65 % of its time waiting to be woken up. A critical path analysis is then triggered by those results. The analysis process is the same, but follows the critical path of the process instead of its own states. Figure 12(b) shows the differential results between the critical path of the invalid instances of `JACK2` and the valid ones. This analysis shows that the `JACK2` daemon process was preempted for 96.51 % of the time. A CPUTop analysis is finally triggered on the period for which `JACK2` was identified as preempted, and on the CPU on which this process was running. Figure 12(c) shows the results of the CPUTop analysis, on which we can see that process `cpuburn` was the one running on the process while `JACK2` was preempted. We can also see in the results that the priority of the `cpuburn` process during that period of time was of -61 (or a real-time priority of 60), while the priority of our process was of -2 (or a real-time priority of 1).

These results thus correspond to the setup of the application and the problem we induced.

5.2. In-kernel wake lock priority inversion

5.2.1. Situation. Priority inversions are a common problem of real-time systems, happening when a process with a low priority prevents another process with a higher priority to run. Such situations usually involve a shared resource with a limited access.

```
IntervalSet(1): [
    [BLOCKED on poll, 3, [4.1705323E7, 4.1705323E7]]
    [RUNNING, 14, [238189.0, 238189.0]]
    [SYSCALL futex, 1, [10094.0, 10094.0]]
    [Interrupted by IRQ29 (snd_hda_intel), 1, [977.0, 977.0]]
    [WAKING, 3, [5.85536417E8, 5.85536417E8]]
    [SYSCALL poll, 6, [58995.0, 58995.0]]
    [SYSCALL ioctl, 5, [103256.0, 103256.0]]
    [SYSCALL write, 5, [31727.0, 31727.0]]
]
```

(a) Set of intervals of time spent in the different process states for the invalid instances of `JACK2`

```
Valid list                                          Distance   Occurrences   Weight
IntervalSet(105): [                                     30          105      93.33%
    [BLOCKED on poll, 2, [2.0703525E7, 2.2803038E7]]
    [RUNNING, 3, [31941.0, 111629.0]]
    [WAKING, 2, [10907.0, 85443.0]]
    [SYSCALL poll, 4, [34946.0, 90288.0]]
]
IntervalSet(2): [                                       31            2      13.28%
    [BLOCKED on poll, 2, [2.0767886E7, 2.17529E7]]
    [RUNNING, 3, [61252.0, 87261.0]]
    [WAKING, 2, [15856.0, 23237.0]]
    [SYSCALL poll, 5, [43815.0, 76711.0]]
    [Interrupted by HRTIMER, 1, [5848.0, 15496.0]]
]
IntervalSet(4): [                                       33            4       9.62%
    [BLOCKED on poll, 2, [2.0757138E7, 2.1794678E7]]
    [RUNNING, 4, [46599.0, 115511.0]]
    [WAKING, 2, [13054.0, 24173.0]]
    [SYSCALL poll, 4, [32867.0, 61412.0]]
    [Interrupted by HRTIMER, 1, [7908.0, 12598.0]]
]
```

(b) Weighting of the valid interval sets against the invalid one in Figure 11(a), showing the number of occurrences of each interval set, its distance to the invalid interval set and its computed weight

```
Diff: [
    [RUNNING, 11, [126560.0, 206248.0]]
    [BLOCKED on poll, 1, [1.8902285E7, 2.1001798E7]]
    [WAKING, 1, [5.85450974E8, 5.8552551E8]]
    [SYSCALL futex, 1, [10094.0, 10094.0]]
    [SYSCALL ioctl, 5, [103256.0, 103256.0]]
    [SYSCALL poll, 2, [0.0, 24049.0]]
    [SYSCALL write, 5, [31727.0, 31727.0]]
    [Interrupted by IRQ29 (snd_hda_intel), 1, [977.0, 977.0]]
]
```

(c) Set of intervals of time considered to be in excess in the invalid interval set compared to the valid interval sets with a weight of at least 50 %

Fig. 11: Algorithm intermediate results for the analysis of `JACK2`

```
State                                   Responsibility for added time
WAKING                                    96.65%
BLOCKED on  poll                           3.29%
RUNNING                                    0.03%
SYSCALL ioctl                              0.02%
SYSCALL write                              0.01%
SYSCALL poll                               0.00%
SYSCALL futex                              0.00%
Interrupted by IRQ29 (snd_hda_intel)       0.00%
Minimum responsibility for a case to be considered: 64.83%
```

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process waits to be woken up

```
Critical path state          Responsibility for added time
jackd PREEMPTED                96.51%
swapper/0 RUNNING               1.88%
swapper/0 PREEMPTED             1.55%
jackd RUNNING                   0.05%
irq/29-snd_hda_ PREEMPTED       0.01%
irq/29-snd_hda_ RUNNING         0.00%
Minimum responsibility for a case to be considered: 60.79%
```

(b) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process is preempted

```
Analyzed CPUs: 0
Analyzed timerange: [11:54:37.711 896 155, 11:54:38.319 519 477]
Per-TID Usage   Process                    Migrations   Priorities
96.34%          cpuburn (11371)                   0         [-61]
 0.29%          bash (11375)                      1          [20]
 0.13%          bash (11376)                      2          [20]
 0.11%          bash (11377)                      0          [20]
 0.10%          /usr/bin/x-term (3154)            2            []
Priorities of the PREEMPTED process during that interval: -2
```

(c) CPUTop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 0, where `JACK2` was running, during the period of time for which `JACK2` was preempted

Fig. 12: Sections of the analysis report generated by our analysis for `JACK2`

We can take for instance the setup presented in Figure 13. In this case, both a low real-time priority process (`lowprio1`) and a higher priority process (`highprio0`) require the same resource, and are both pinned on different CPUs. The low priority process requests the lock on the resource first, leaving the high priority process waiting for its turn to access it. However, if the low priority process is preempted by another process with a higher priority than its own (`highprio1`), pinned on the same CPU, then it is not able to finish using the resource and release its lock. In such situations, the Linux kernel uses the priority inheritance process to allow the low priority process to take over the CPU in order to finish its work with the resource. The priority inheritance process means that the priority of the high priority process waiting for the resource (`highprio0`) is given to the low priority process for the time needed to release the lock. That is what we can see in Figure 13. This process with the raised priority will keep it until it releases the lock on the resource, then its old priority is restored. In the case shown in the Figure, this means that `lowprio1` will again be preempted by `highprio1`, but this time without holding the resource. With the lock released, `highprio0` will be able to do its work with the resource.

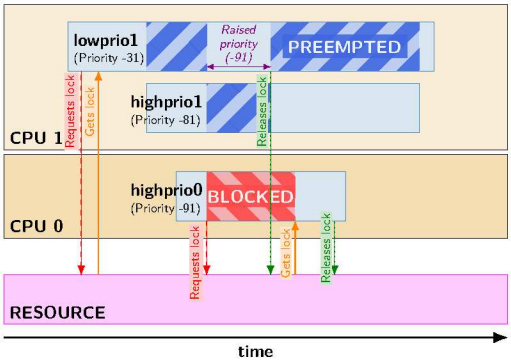


Fig. 13: Schema of a priority inheritance while waiting for a resource: the `lowprio1` process takes the lock of the resource and is preempted by a higher priority process `highprio1`, but can finish its task and free the lock when inheriting temporaryly the priority of process `highprio0`

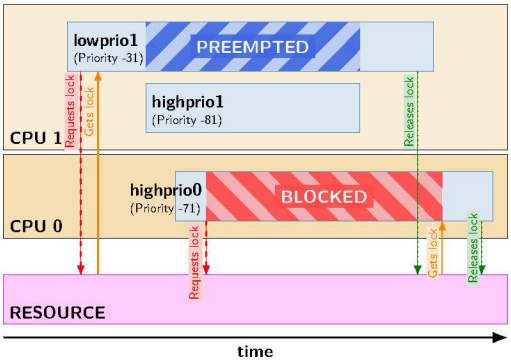


Fig. 14: Schema of an in-kernel wake lock priority inversion: the `lowprio1` process takes the lock of the resource and cannot free it for the higher-priority `highprio0` process because `lowprio1` is currently preempted by the `highprio1` process, which has a higher priority than `highprio0`

Figure 14 shows the schematization of an in-kernel wake lock priority inversion situation. The difference with Figure 13 is the priority of the `highprio0` process, which is now "70" instead of "90", and thus lower than the priority of `highprio1`. Therefore, in this case, the process preempting the low priority one has a higher priority than the high priority process waiting for the resource. We thus end up in a situation where a low priority process prevents a high priority process from running by holding the lock on a needed resource.

Table II: Real-time priorities of the processes in the setup used to generate the trace for the in-kernel wake lock priority inversion. An empty priority means that the process was not started.

	Real-time priority			
Process	Setup 1	Setup 2	Setup 3	Setup 4
lowprio1		30	30	30
highprio1			80	80
highprio0	70	70	70	90

A kernel module and small application have been created to simulate this problem. The kernel module creates a file that can be opened only once at a time using a `rtmutex`. The application starts the multiple threads presented in Figures 14 and 13. We ran that application with different setups in order to trace both working and non-working scenarios and analyze the generated traces automatically. These setups are presented in Table II.

5.2.2. Automatic analysis. The results of the automatic analysis are presented in Figure 15. Figure 15(a) shows the results of the state analysis of the application simulating an in-kernel wake lock priority inversion. We can see in those results that, for the invalid instances, the process spends most of its excess time being blocked on an `open` system call. This thus triggers both the priority inheritance analysis and the critical path analysis, for which results are respectively presented in Figure 15(b) and Figure 15(c). The former shows that, in invalid instances, the priority inheritance stays active for an abnormal amount of time

State	Responsibility for added time
BLOCKED on open	100.00%
RUNNING	0.00%
SYSCALL open	0.00%
SYSCALL gettid	0.00%

Minimum responsibility for a case to be considered: 56.70%

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on an `open` system call

```
Duration of active 'sched_pi_setprio'...
... in valid instances (maximum): 170.909us
... in invalid instances (average): 329.534us

Active 92.81% more time in invalid instances than in valid instances.

Verdict: Very high probability of a priority inversion
```

(b) Priority inheritance analysis triggered by the state analysis, comparing the average time where the priority inheritance was active in invalid instances to the maximum time in valid instances

Critical path state	Responsibility for added time
lowprio1 PREEMPTED	100.00%

Minimum responsibility for a case to be considered: 100.00%

(c) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process waits after the `lowprio1` process, which is preempted

Analyzed CPUs: 1
Analyzed timerange: [21:40:35.867 825 012, 21:40:36.033 045 371]

Per-TID Usage	Process	Migrations	Priorities
99.18%	highprio1 (26408)	0	[-100, -81]
0.82%	lowprio1 (26407)	0	[-71]
0.00%	lttng-consumerd (26369)	0	[20]
0.00%	lttng-consumerd (26370)	0	[20]
0.00%	Cache2 I/O (3011)	0	[20]

Priorities of the PREEMPTED process during that interval: -71

(d) CPUTop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 1, where `lowprio1` was running, during the period of time for which `lowprio1` was preempted

Fig. 15: Sections of the analysis report generated by our analysis for the in-kernel wake lock priority inversion

compared to valid instances, resulting in a verdict of very high probability of a priority inversion. The latter shows that the the added time in invalid instances was spent waiting for the `lowprio1` process which was preempted. The CPUTop analysis is then triggered by these last results. As we can see in Figure 15(d), the CPUTop analysis shows that the `lowprio1` process was preempted by the `highprio1` process, which has a greater priority.

These analysis results thus allow to directly identify the in-kernel wake lock priority inversion problem, and its origin.

5.3. Bad userspace code

5.3.1. Situation. When working on high performance applications, all the algorithms are important, as a poorly written source code can induce latency. In some cases, however, a portion of code that should not be run frequently is thus not optimized. In such a situation, the cause of a missed deadline is not external to our task, preventing us from analyzing what is happening outside of the studied task. However, it is still possible to give an helpful insight into what is happening in the application, thanks to the model.

Figure 16 shows the model of an application that has been built to have such a problem. In this application, we know that the global time taken between the entry in state "step1" and the exit of step "step2" should be less than 200 µs. In the application, in between each state of the model, we do a busy loop. Though, using a random number generator, we double the number of loop iterations executed by the application while in the "step3" state, once in a while, making the application miss its deadline. Using multiple runs of that application, we were able to generate a trace with working and non-working instances to analyze with our automatic analysis. This closely resembles an actual problem encountered in an industrial real-time application.

5.3.2. Automatic analysis. The results of the automatic analysis are presented in Figure 17.

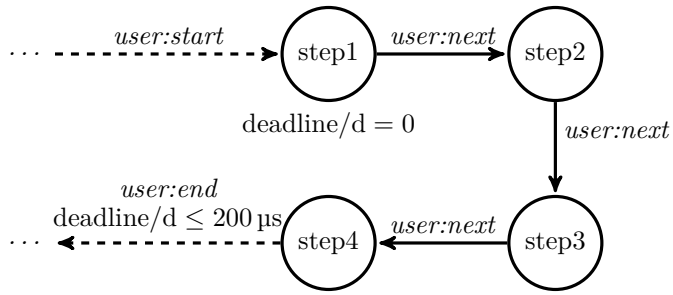


Fig. 16: Graphical representation of the period within our userspace application

State	Responsibility for added time
RUNNING	99.89%
Interrupted by HRTIMER	0.04%
Interrupted by SOFTIRQ_TIMER	0.03%
Interrupted by SOFTIRQ_RCU	0.02%
SYSCALL write	0.02%
Interrupted by SOFTIRQ_SCHED	0.00%
SYSCALL gettid	0.00%

Minimum responsibility for a case to be considered: 64.94%

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly running

Model state	Responsibility for added time
step3	99.92%
step1	0.05%
step4	0.03%
step2	0.01%

(b) State machine state analysis triggered by the state analysis, showing the responsibility of each state of the model for the added time in the invalid instances: in this case, the process spends most of its time in the "step3" state

Fig. 17: Sections of the analysis report generated by our analysis for the bad userspace code

Figure 17(a) presents the results of the state analysis of our userspace application where invalid instances are due to a busy loop. We can see that the state analysis highlights the running state of the application. This means that the excess time spent in invalid instances was while the application was running. The state analysis results thus trigger a state machine state analysis, as shown in Figure 17(b). This analysis compares the time spent in each machine state for valid and invalid instances. Such analysis allows to pinpoint the place in the source code where the excess time is spent for invalid instances. In this case, we can see that the application was spending most of its excess time in the "step3" state.

These analysis results thus allow to recognize that the problem in this case was linked to the way the application was written, and to get a fix on where in the code to look for correcting the problem.

5.4. Frequency scaling

5.4.1. Situation. With the advent of real-time systems, general purpose computers are often able to run soft real-time applications. However, the frequency of processors is usually configured to scale with the work intensity, to optimise power consumption.

The frequency of a running application can also change when there is a CPU migration and the two processors do not have the same frequency. Some embedded architectures rely on this same principle, such as the `big.LITTLE` architecture, which takes advantage of using simultaneously a very low-power processor (*LITTLE*) as the main microprocessor, and several higher-power processors (*big*) activated when there is a need for high computing power.

These situations can thus lead to instances of the modeled application taking more or less time throughout the trace, even without any internal or external interference apart from the CPU frequency change. In such cases, analysis results can be altered from what we would expect, and might be useless.

5.4.2. Automatic analysis. While working on the different case studies, we encountered this problem. We thus worked on a frequency scaling analysis to include at the beginning of the automatic generated report.

```
CPU frequency for valid instances:

| CPU Frequency | Percent of time |
|---------------|-----------------|
| 2661.00 MHz   | 83.09%          |
| 2660.00 MHz   | 11.85%          |
| 2527.00 MHz   |  5.06%          |

Average frequency: 2654.10 MHz

CPU frequency for invalid instances:

| CPU Frequency | Percent of time |
|---------------|-----------------|
| 1596.00 MHz   | 39.77%          |
| 2527.00 MHz   | 24.97%          |
| 2661.00 MHz   | 18.36%          |
| Unknown       |  8.55%          |
| 2660.00 MHz   |  8.36%          |

Average frequency: 2161.23 MHz

22.81% higher average frequency in valid than in invalid instances.

Verdict: Probability of a frequency scaling problem
```

Fig. 18: Frequency scaling analysis, showing the CPU frequencies for valid and invalid instances, and comparing their average frequencies in order to identify if there is a probability of a frequency scaling problem

Figure 18 shows the results of such analysis, in a case where there was a frequency scaling problem while a deadline constraint was verified. We can see, in the analysis, the two histograms showing the different frequencies while valid and invalid instances were running. An average frequency is then computed for each situation, and those averages are finally compared. In this case, the average CPU frequency was 22.81 % higher in valid instances than in invalid instances, leading to a verdict of probability of a frequency scaling problem.

5.5. Preempted waker

5.5.1. Situation. The `cyclictest` tool allows to verify the software real-time performance of a system. It does so by executing a periodic task with multiple processes. Each process runs on a different CPU, and each task can be set with a different period. The aim of the task is to be woken up during the given period. The performance is then evaluated by measuring the discrepancy between the desired period and the real wake up time.

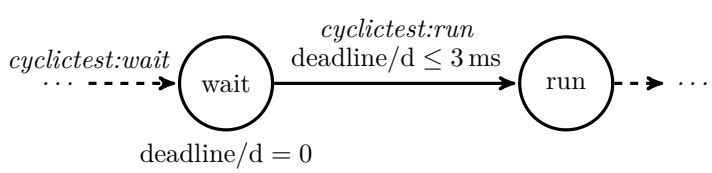


Fig. 19: Graphical representation of the period of `cyclictest`

We instrumented `cyclictest` to have tracepoints around its periodic task, which leads to the model representation in Figure 19. We then ran it with a task on each CPU, a real-time priority of 99, and the `SCHED_FIFO` scheduler. The experience was done on an NVIDIA Jetson TK1 while tracing in snapshot mode with `LTTng`. The snapshot mode allows to write the trace buffers to disk as soon as a specific event, high latency in our case, is detected by the application.

At some point, the tool reported an abnormal latency of 7 ms, and the tracer saved the snapshot of the trace buffers to analyze. Without any insight on the origin of the problem, we used our automatic analysis to understand what happened.

5.5.2. Automatic analysis. The results of the automatic analysis of the `cyclictest` run are presented in Figure 20.

State	Responsibility for added time
BLOCKED on rt_sigtimedwait	99.95%
RUNNING	0.03%
SYSCALL getcpu	0.01%
SYSCALL clock_gettime	0.01%

Minimum responsibility for a case to be considered: 56.68%

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on the `rt_sigtimedwait` system call

Critical path state	Responsibility for added time
ktimersoftd/3 PREEMPTED	99.68%
ktimersoftd/3 RUNNING	0.24%
cyclictest PREEMPTED	0.08%

Minimum responsibility for a case to be considered: 52.77%

(b) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process mostly waits after the `ktimersoftd/3` process, which is preempted

Analyzed CPUs: 3
Analyzed timerange: [15:55:53.528 529 561, 15:55:53.536 012 105]

Per-TID Usage	Process	Migrations	Priorities
99.73%	irq/154-hpd (162)	0	[-51]
0.26%	irq/25-54200000 (163)	0	[-51]
0.01%	ktimersoftd/3 (32)	0	[-2]
0.00%	irq/22-Tegra PC (160)	0	[-51]
0.00%	irq/388-eth0 (611)	0	[-51]

Priorities of the PREEMPTED process during that interval: -2

(c) CPUTop analysis triggered by the critical path analysis, showing the processes that were running on CPU 3, during the period of time for which `ktimersoftd/3` was preempted

Fig. 20: Sections of the analysis report generated by our analysis for `cyclictest`

Figure 20(a) presents the results of the state analysis for our `cyclictest` execution. We can see that the `cyclictest` instance that missed the deadline was blocked on a `rt_sigtimedwait` system call. These results trigger a critical path analysis that is shown in Figure 20(b). This analysis puts forward that in invalid instances, the `cyclictest` process spent time waiting to be woken up by the `ktimersoftd/3` process, which was preempted. The CPUTop analysis is finally triggered by the results of the critical path analysis. As we can see in Figure 20(c), while the `ktimersoftd/3` process was preempted, the `irq/154-hpd` process was running on the CPU. The preempted process had a real-time priority of 1, while the interrupting process had a real-time priority of 50.

The `irq/154-hpd` process corresponds to the interrupt handler for the hot plug detect feature. This feature is a communication mechanism between two devices allowing to be aware when one is connected to or disconnected from the other [National Instruments 2015]. This mechanism is supported by the HDMI protocol. Knowing that, we were able to identify that the latency was caused when we plugged an HDMI screen to the NVIDIA Jetson TK1 while trying to see the current `cyclictest` results.

6. RUNNING TIME AND SCALABILITY

This section presents the timing results of our analysis as well as its scalability according to different factors. All the tests presented were executed on a test system that consists of an Intel® Core™ i7-4790 CPU at 3.6 GHz, with 32 GiB of DDR3 RAM at 1600 MHz. Hyperthreading was disabled, as well as the idling and turbo modes of the CPU.

6.1. Running time

Table III: Number of events and sizes of the traces used to measure our analysis running time

Case	Number of events		Duration of trace (s)	Size (MiB)
	UST	Kernel		
jackd	321	419 164	3.144	25.09
wakelock	42	194 997	1.605	11.92
userspace	29	89 840	0.6353	5.456
cyclictest	41 677	208 489	12.90	21.63

To present the running time of our analysis, we use the different cases presented in 5. Table III presents the information about the traces that were used to analyze our different cases. We can see that all those traces are different in terms of duration and number of events.

Table IV: Results of the instances duration analysis for the traces of the cases studied, which computes information about the similar instances steps in the valid and invalid instances

Instances	Count	Instances Steps Execution Duration (in ms)			
		Minimum	Average	Maximum	St. Dev.
jackd					
Valid	112	20.857	21.324	22.062	0.427
Invalid	1	627.685	627.685	627.685	0.000
wakelock					
Valid	3	0.025	113.541	170.952	80.270
Invalid	1	329.575	329.575	329.575	0.000
userspace					
Valid	2	0.070	0.074	0.078	0.004
Invalid	1	21.463	21.463	21.463	0.000
cyclictest					
Valid	20 834	0.830	1.734	2.686	0.559
Invalid	1	7.562	7.562	7.562	0.000

Table IV reports the results of the instance duration analysis performed when working on a deadline constraint. It allows to identify the variation in the duration of the instances of the application, as well as to have an idea on the number of valid and invalid instances in the trace that will be analyzed.

Table V shows the process state analysis execution duration and results with and without sampling. Only the `JACK2` and `cyclictest` cases are presented here as they are the only ones of the four cases taking advantage of sampling, as the sample size starts to be smaller than the list size when reaching a cardinality of 21. We can thus observe that the number of analyzed instances is very different when sampling is used. While looking at both Tables IV and V, we can see that the number of analyzed instances for `cyclictest` is different than the total number of instances found in the trace. This is justified by the fact that `cyclictest`

Table V: Statistical comparison between the use of sampling or not for the process state analysis, in terms of number of instances analyzed, analysis execution duration and result (percentage of responsibility for the highest responsible state for added time); the highest responsible state was, in all cases, identical within 0.1% whether sampling was used or not; statistics were computed with 100 runs of the analysis for each situation.

		Process state analysis				
Case	# instances analyzed	Execution duration (ms)		Most responsible (%)		
jackd						
Without sampling	113	23 930	± 1 552	96.65	±	0.
With sampling	88	15 310	± 1 038	96.65	±	0.002 222
Difference	−25	−8 615		−0.000 590 7		
cyclictest						
Without sampling	4 191	103 100	± 4 609	99.99	±	0.
With sampling	379	7 125	± 432.1	99.91	±	0.063 78
Difference	−3 812	−95 990		−0.077 95		

Note: The number of analyzed instances can be different than the total number of instances, even when sampling is disabled, as instances steps are dropped when data is missing.

was traced in snapshot mode. In this mode, when buffers are full, the oldest data is dropped in favor of the new one. This means that data can be missing to analyze instances, hence the analysis dropping these.

We can also see in Table V that while the analysis execution duration is drastically reduced, the results are highly similar, still leading to the right conclusions. Using sampling in our analysis can thus be considered as an approach to retain, as it allows to process a much reduced but sufficient number of instances while keeping a good precision.

Table VI presents the benchmark of the different steps of the analysis, per step of the analysis and per case analyzed.

We can see for instance that the *Instances duration* analysis, which goes through all the instances in the trace, is taking much more time for the `cyclictest` case than for the three others. This is easily explained by the number of instances in this case versus the others. This is also the situation for the *CPU Frequency* and the *Priority inheritance* analysis.

The *Process state* analysis is the initial analysis performed with our algorithms. We can observe that this analysis is taking more time for the `jackd` case than for `cyclictest`, despite the large number of instances. This is justified by the fact that `jackd` instances, even valid ones, are taking much more time than `cyclictest` ones, as we can see in Table IV. The main influence of the instance duration when performing a *Process state* analysis is that when instances take more time, more state changes will happen, leading to more data to analyze.

The *Critical path* analysis as well suffers partly of the instance duration, but is less influenced by the number of instances as the closest instances that are used for comparison have been identified during the *Process state* analysis. This means that this analysis is mostly impacted by the duration of the invalid instances.

The *CPUTop* analysis is an external python analysis part of the `lttng-analyses` python scripts. Its duration is mostly limited by the period of time identified for further analysis by the *Critical path* analysis. It however takes time as it has to read the trace from the beginning until it reaches the period of time to be analyzed. This is thus highly dependent on the number of events and the position of the period to analyze in the trace.

This last statement can be extended to all the steps of the analysis, as the more events there is in the trace, the more events there is to read to reach the ones that are of importance to the analysis.

Table VI: Statistics on the execution duration of the different parts of our analysis process, computed with 100 runs of the analysis for each case studied in 5

Analysis	Analysis execution duration (in ms)			
	Minimum	Average	Maximum	St. Dev.
Instances duration				
jackd	0.833 6	0.947 7	1.228	0.067 26
wakelock	0.613 2	1.374	8.734	1.497
userspace	0.628 9	1.609	6.917	1.311
cyclictest	24.59	25.51	35.18	1.099
CPU Frequency				
jackd	21.32	24.67	29.10	1.372
wakelock	4.966	11.90	34.50	6.727
userspace	5.107	11.99	37.03	5.944
cyclictest	14 810	15 850	17 130	431.8
Process state				
jackd	12 740	15 310	18 380	1 038
wakelock	264.6	402.1	591.0	82.76
userspace	185.2	250.7	358.3	29.10
cyclictest	6 068	7 125	8 389	432.1
State machine state				
userspace	0.484 7	0.680 0	3.799	0.399 8
Priority inheritance				
wakelock	0.690 9	0.990 7	4.629	0.591 7
cyclictest	8 360	8 725	8 960	135.0
Critical path				
jackd	1 092	1 289	1 632	101.7
wakelock	758.5	990.8	1 163	96.49
cyclictest	155.1	520.8	656.8	107.1
CPUTop				
jackd	4 919	5 020	5 322	59.38
wakelock	2 500	2 558	2 667	29.97
cyclictest	7 980	8 201	8 398	84.29

6.2. Scalability

In order to test and verify the scalability of our approach according the different points raised in 6.1, we created a simple program performing a parametrable number of *chmod* system calls. We ran and traced the execution of that program while varying different parameters such as the number of instances or the number of system calls performed for a valid or an invalid instance. In each case, only one invalid instance was created to limit the variation of the experience parameters. We then ran our analysis on these multiple traces in order to generate the graphs presented in Figures 21 and 23. These graphs reflect the evolution of the analysis duration for full and partial analysis, with more or less events enabled while tracing the application.

Figure 21 shows the evolution of our analysis duration according to the number of instances in the trace.

We can see in the graph that, for a partial analysis, the analysis time is relatively constant. This can be explained by the fact that, when performing a partial analysis, only the invalid instances are considered, and only one invalid instance was created when running the program.

The full analysis is following a different pattern where the duration of the analysis is proportional to the number of instances in the trace. That can be explained by the number of instances to analyze in order to extract the right information, which is increasing with the number of instances available in the trace. This tendency is shown in Figure 22, which represents the size of the sample to analyze according to the number of instances in the traces used for these tests. We can see however that, even if the sample size stabilizes, the

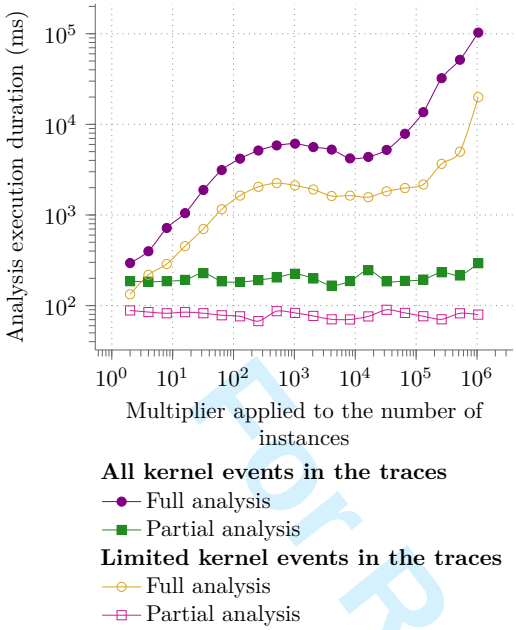


Fig. 21: Execution duration of the initial variable analysis according to the number of instances in the trace, when only one of these instances is invalid; the original number of instances is 2; each data point is the average of 20 runs

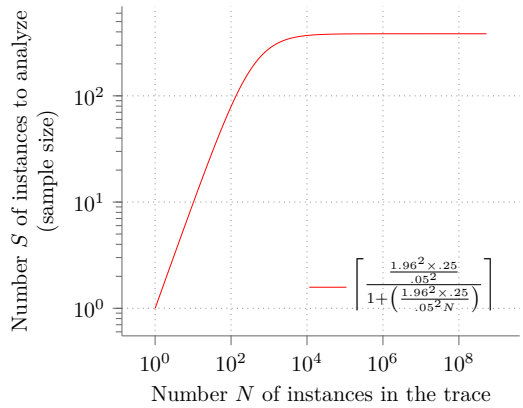


Fig. 22: Number of instances to analyze according to the number of valid or invalid instances in the trace, using (1) with $p = 50\%$, $z = 1.96$ and $e = 5\%$

full analysis duration continues to increase with the number of instances to analyze. This is also explained by the number of events to read in order to reach the instances to analyze.

Figure 23 shows the evolution of our analysis duration according to the number of variable changes during an instance. The traces used contained only two instances, one of which was invalid. We can see that the more variable value changes we have for an instance, the more time the analysis will take. This is easily explained by the fact that for each variable value change, an event has to be extracted from the trace to perform the analysis. Moreover, the higher the number of system calls generated by the invalid instance, the longer the instance will take, and the higher the number of external events in the trace will be, also adding to the analysis time when searching for the specific event that provoked a change to the variable value.

Figure 24 shows the influence of the position of the instances on the analysis duration. We can see that for the partial analysis, the analysis duration is always the same and does not depend on the position of the invalid instance in the trace. This is due to the fact that only the invalid instance will be analyzed. Therefore, once we reach the position of the instance in the trace, we will not have to go back in time in the trace for another one. However, this is not the case for the full analysis, where we also have to analyze some of the valid instances. This is why we can see variations in the duration of the full analysis.

Finally, we can see in Figures 21, 23, and 24 that if the number of events in the trace does not impact the global tendency of the analysis duration, it still impacts its value. We

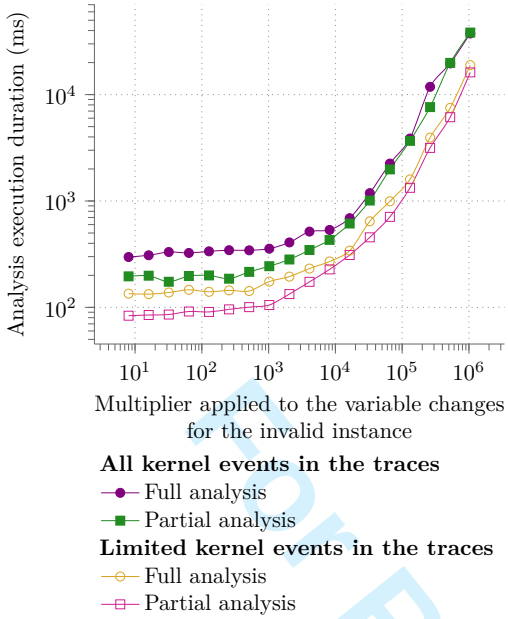


Fig. 23: Execution duration of the initial variable analysis according to the number of variable value changes during an instance; the number of variable value changes for a valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs

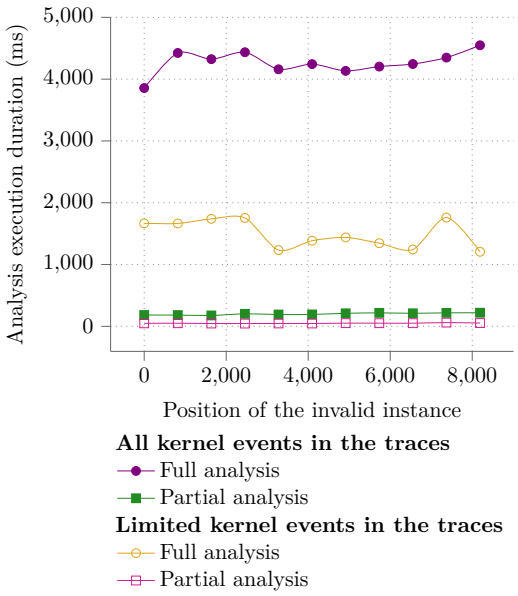


Fig. 24: Execution duration of the initial variable analysis according to the position of the invalid instance among the 8192 instances in the trace; the number of variable value changes for an valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs

can thus observe that when there are fewer events in the trace, it takes less time to perform the analysis.

All those tests confirm that our analysis can be executed in time proportional to the number of events in the trace, the number of instances of the application, and the number of variable value changes during an instance. However, the position of the instances in the trace has no direct impact on the duration of the analysis.

7. CONCLUSION AND FUTURE WORK

We proposed a new approach to use model-based constraints and kernel and user-space traces to automatically analyze the origin of unexpected behaviors in real-time and multi-core applications. We presented a brief overview of the model representation used to follow the workflow of the application in the trace, part of previous work. We then detailed the organization and extraction of the interesting data for a given invalidated constraint. This extraction is followed by an assignation of responsibility that provides the user with a list of potential causes for the constraint violation. Depending on the type of cause identified, further analysis can be automatically started to obtain more precise information for the user. We applied our new approach to common real-time and multi-core problems, and provided examples of analysis reports that helped us to pinpoint the cause of encountered problems. We finally characterized the execution time as well as the scalability of our implementation,

as a function of the number of instances in the trace, the duration of the instances, the position of the instance and the number of events in the trace.

This approach is novel and an important step to automate performance analysis and problem detection, in order to save time and make tracing more accessible to people without a deep system knowledge. We intend to pursue our work in order to propose an approach to automatically set values to the constraints when analyzing a trace.

References

Luca Aceto, Augusto Bergueño, and Kim Guldstrand Larsen. 1998. Model Checking via Reachability Testing for Timed Automata. In *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Bernhard Steffen (Ed.), Vol. 1384. Springer-Verlag, Gulbenkian Foundation, Lisbon , Portugal, 263–280.

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining Association Rules Between Sets of Items in Large Databases. *SIGMOD Record* 22, 2 (1 June 1993), 207–216. DOI:http://dx.doi.org/10.1145/170036.170072

Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB '94)*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 487–499.

Raphaël Beamonte and Michel R. Dagenais. 2015. Linux Low-Latency Tracing for Multicore Hard Real-Time Systems. *Advances in Computer Engineering* 2015 (18 Aug. 2015), 8. DOI:http://dx.doi.org/10.1155/2015/261094 Article ID 261094.

Raphaël Beamonte and Michel R. Dagenais. 2016. Detection of common problems in real-time and multi-core systems using model-based constraints. *Scientific Programming* 2016 (16 March 2016), 18. DOI:http://dx.doi.org/10.1155/2016/9792462 Article ID 9792462.

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1–3 (9 June 2005), 217–239. DOI:http://dx.doi.org/10.1016/j.tcs.2004.12.030

Xi Chen, Hsieh Harry, Felice Balarin, and Yosinori Watanabe. 2004. Logic of constraints: a quantitative performance and functional constraint formalism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 8 (Aug. 2004), 1243–1255. DOI:http://dx.doi.org/10.1109/TCAD.2004.831575

Balakrishnan Dasarathy. 1985. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them. *IEEE Transactions on Software Engineering* SE-11, 1 (Jan. 1985), 80–86. DOI:http://dx.doi.org/10.1109/TSE.1985.231845

Julien Desfossez. 2016. Monitoring real-time latencies. (6 Jan. 2016). Retrieved June 9, 2016 from https://lttng.org/blog/2016/01/06/monitoring-realtime-latencies/

Mathieu Desnoyers. 2010. A new unified Lockless Ring Buffer library for efficient kernel tracing. (11 Aug. 2010). Retrieved June 9, 2016 from http://www.efficios.com/pub/linuxcon2010-tracingsummit/presentation-linuxcon-2010-tracing-mini-summit.pdf

Mathieu Desnoyers. 2011. Common Trace Format (CTF) Specifications. (2011). Retrieved June 9, 2016 from http://git.efficios.com/?p=ctf.git; a=blob_plain; f=common-trace-format-specification.txt

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (Feb. 2012), 375–382. DOI:http://dx.doi.org/10.1109/TPDS.2011.159

Jake Edge. 2009. A look at ftrace. (13 March 2009). Retrieved June 9, 2016 from http://lwn.net/Articles/322666/

Frank Ch. Eigler. 2006. Problem Solving With Systemtap. In *Proceedings of the Linux Symposium*, Vol. 1. Ottawa, Ontario, Canada, 261–268.

S. Alan Ezust. 1995. *Tango: The Trace ANalysis GeneratOr*. Master's thesis. McGill University, Québec, Canada.

S. Alan Ezust and Gregor V. Bochmann. 1995. An Automatic Trace Analysis Tool Generator for Estelle Specifications. In *Proceedings of ACM SIGCOMM 95 Conference*. 25–4.

Ronald Aylmer Fisher and Frank Yates. 1949. *Statistical tables for biological, agricultural and medical research* (3rd ed., rev. and enl. ed.). Oliver and Boyd, London.

Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice & Experience* 22, 6 (April 2010), 702–719. DOI:http://dx.doi.org/10.1002/cpe.v22:6

Francis Giraldeau. 2015. *Analyse de performance de systèmes distribués et hétérogènes à l'aide de traçage noyau*. Ph.D. Dissertation. École Polytechnique de Montréal, Québec, Canada.

R. Krishnakumar. 2005. Kernel Korner: Kprobes-a Kernel Debugger. *Linux Journal* 2005, 133 (May 2005), 11–.

Kim-Hung Li. 1994. Reservoir-sampling Algorithms of Time Complexity O(N(1 + Log(N/N))). *ACM Trans. Math. Software* 20, 4 (Dec. 1994), 481–493. DOI:http://dx.doi.org/10.1145/198429.198435

Paul E. McKenney and Jonathan Walpole. 2008. Introducing Technology into the Linux Kernel: A Case Study. *ACM SIGOPS Operating Systems Review* 42, 5 (July 2008), 4–17. DOI:http://dx.doi.org/10.1145/1400097.1400099

National Instruments. 2015. Test Engineer's Guide to HDMI 1.4. (17 Aug. 2015). Retrieved June 9, 2016 from http://www.ni.com/white-paper/12680/en/

T Manjula Peiris and James H Hill. 2012. Adapting System Execution Traces for Validation of Distributed System QoS Properties. In *IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, IEEE, 162–171. DOI:http://dx.doi.org/10.1109/ISORC.2012.29

Gregory Piatetsky-Shapiro. 1991. Discovery, analysis and presentation of strong rules. In *Knowledge Discovery in Databases*, Gregory Piatetsky-Shapiro and William J. Frawley (Eds.). AAAI/MIT Press, Cambridge, MA, 229–248.

François Rajotte and Michel R. Dagenais. 2014. Real-Time Linux Analysis Using Low-Impact Tracer. *Advances in Computer Engineering* 2014 (5 June 2014), 8. DOI:http://dx.doi.org/10.1155/2014/173976 Article ID 173976.

Eric Sven Ristad and Peter N. Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (May 1998), 522–532. DOI:http://dx.doi.org/10.1109/34.682181

Steven Rostedt. 2008. ftrace - Function Tracer. (2008). Retrieved June 9, 2016 from https://www.kernel.org/doc/Documentation/trace/ftrace.txt

Steven Rostedt. 2010. Ftrace: Now and Then. (2010). Retrieved June 9, 2016 from http://people.redhat.com/srostedt/trace-cmd-linuxcon-2010.odp

Kumar P. Suresh and S. Chandrashekara. 2012. Sample size estimation and power analysis for clinical research studies. *Journal of Human Reproductive Sciences* 5, 1 (Jan. 2012), 7–13. DOI:http://dx.doi.org/10.4103/0974-1208.97779

The Eclipse Foundation. 2014. Trace Compass. (2014). Retrieved June 9, 2016 from https://projects.eclipse.org/proposals/trace-compass

Harry Thornburg. 2009. Perfcounters added to the mainline. (1 July 2009). Retrieved June 9, 2016 from http://lwn.net/Articles/339361/

Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Software* 11, 1 (March 1985), 37–57. DOI:http://dx.doi.org/10.1145/3147.3165

Jeffrey Scott Vitter. 1987. An Efficient Algorithm for Sequential Random Sampling. *ACM Trans. Math. Software* 13, 1 (March 1987), 58–67. DOI:http://dx.doi.org/10.1145/23002.23003

Received June 2016; revised xxxx 2016; accepted xxxx 2016