Declarative Framework to Stateful Analysis of Execution Traces

Florian Wininger
Ecole Polytechnique Montreal
Montreal, Quebec h3t 1j4
florian.wininger@polymtl.ca

Naser Ezzati-Jivan
Ecole Polytechnique Montreal
Montreal, Quebec h3t 1j4
n.ezzati@polymtl.ca

Michel R. Dagenais
Ecole Polytechnique Montreal
Montreal, Quebec h3t 1j4
michel.dagenais@polymtl.ca

***Résumé*—With newer complex multi-core systems, it is important to understand applications' run-time behaviour to be able to debug their executions, detect possible problems and bottlenecks and finally identify potential root-causes. Execution traces usually contain precise data about applications' execution, with which analysis and abstraction at multiple levels, they can provide valuable information and insights about the applications' run-time behaviour. However, with multiple abstraction levels, it becomes increasingly difficult to find the exact location of detected performance or security problem. Tracing tools provide various analysis views to help understand these problems. However, these views are not somehow enough to uncover all aspects of the underlying issues. The developer is in fact the one who best knows his application. Therefore, a declarative approach that enables users to specify and build their custom analysis based on their knowledge, requirements and problems can be more useful and effective. In this paper we propose a generic declarative trace analysis framework to analyze, comprehend and visualize execution traces. This enhanced framework builds custom analysis based on a specified modelled state, extracted from a system execution trace and stored in a special purpose database. The proposed solution enables users to first define their different analysis models based on their application and requirements, then visualize these models in many alternate representations (Gantt chart, XY chart, etc.), and finally filter the data to get some highlights or detect some potential patterns. Several sample applications with different operating systems are shown using trace events gathered from Linux and Windows kernel and user-space levels.**

I. Introduction

Debugging applications in distributed and multi-core environments and finding their performance bottlenecks and runtime problems are difficult and almost imposable using only the static data (e.g., source codes, documents and other software artifacts). Instead, dynamic analysis is mostly used to debug complex application, for which execution traces provide a highly detailed data.

The principle behind execution tracing is to insert trace points or probes at specific locations in the source code or binary of an application. Those trace points are executed and trace events/logs are generated when encountered during program execution. The LTTng tracer (Linux Tracing Toolkit Next Generation) [1], [2], DTrace [3], SystemTap [4] are some of modern Linux operating system tracers which are referred or used in this research.

Although tracing tools generate useful and precise data about the runtime behaviour of a program, the collected data may become very large and difficult to follow, when a system with several nodes and multiple cores are traced. Therefore, it is essential to have efficient analysis and filtering tools in order to highlight the important portions of the execution, extract useful information, detect problems and identify their possible root causes.

There are several tools,e.g., LTTV (Linux Tracing Toolkit Viewer)[5], Jumpshot [6], Triva [7], Trace Compass [1] available to analyze trace events and give graphical representations of different run-time aspects. However, a limitation of these tools is that they are only available for a particular trace type generated by a specific tracer. Another limitation is that they only cover the most typical contexts and in fact give less flexibility, forcing users to only use the available shipped analysis, views and features.

Nevertheless, as problems are often complex and unique, it is most likely that these default analyses do not help targeting them sufficiently. We propose in this paper a tool architecture to allow the users to easily extend the available analysis tools according to the application's custom characteristics and needs. This approach is data-driven and potentially results in different analysis models and views each time a new declarative specification is chosen.

Using this approach, users can declaratively define the way they deal with the input trace events, the type and quantity of aggregated information they want to keep track, and also the way they aim to represent the results. In comparison with the previous works and tools, the proposed solution :

- increases **useability** of the existing tools by making it easier to create new analysis and views for custom users problems,
- increases **expressiveness** by replacing the current hard-coded analysis and views by high-level declarative analysis and views,
- increases **flexibility** of the existing tools by supporting different trace types and formats,
- increases **maintainability** of the existing tools, by replacing some parts of system (analysis modules) by declarative modules, therefore less code to maintain,
- preserves and improves the **performance** of the tools by compiling the declarative models to low-level codes and

[1] http://www.eclipse.org/linuxtools/projectPages/lttng/

executing these codes in the run-time (will be explained and experimented in the later sections).

The remainder of the paper is organized as follows : after reviewing related work and the existing infrastructure, we present the specification of the proposed declarative language and detail the implementation. Then, we discuss several possible analyses and visualization views using an enhanced state model, and validate the flexibility and performance of this solution. Finally, we conclude and outline possible future work.

II. Architecture

As mentioned earlier, our proposed solution is a generic and flexible solution which supports different trace formats. In this section, we present the architecture of our declarative solution which is shown in Figures 1, fig :arch2. In the following, we explain the different modules of the architecture and the way each module works. But, to give a sense of what a trace can look like, we first define the general format of the traces that our solution can support.

A trace is actually defined as a sequence of timestamped events e_1, e_2,...,e_i,...,e_n, in which each event $e_i t_i$,r_1,...,r_n,v_1,...,v_m composed of a timestamp t_i, a set of system resources r_1 to r_n (i.e., machine, CPU, process, file, function name, etc.) and a set of values v_1 to v_m (i.e., count, return value, output, etc.). Each trace event is in fact the lowest observable log unit to depict the system behaviour at a specific time point (t_i the timestamp of the event). Therefore, a trace (i.e., a set of events) represents underlying system behaviour during the time duration of its events : [t_1 (timestamp of the first event) ,t_n(the timestamp of the last event)].

Common approach of trace analysis is gathering the trace events from different distributed machines, parsing and analyzing and then aggregating them to some high-level models, e.g., states [8], synthetic events [9], compound events [10], as shown in Figure 1. These high-level models are fed, in turn, into visualization process to display the analysis results to users.

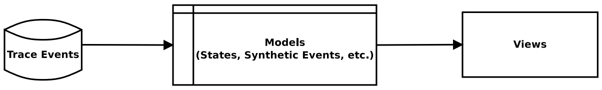


Figure 1. Common trace analysis approach.

Performing these two actual steps (trace -¿ models and models-¿views) in most previous works are commonly hardcoded. The main reason is that the trace type/format is known in advance and the analysis models and views are also generally defined previously. However, a declarative approach is used in our proposed architecture, which is depicted in Figure 2.

In the architecture shown in Figure 2, both the analysis steps (trace -¿ models and models-¿views) are delegated to users. Users can define the way they handle their trace events, they way they extract the high-level notions and models, and the way they visualize and display the models. This approach gives more flexibility to users. User in fact is the only person that knows its application and the possible problematic points very well, therefore, (s)he can easily customize the analysis and the output, based on his application's needs and requirements. In the following we explain different parts of the above architecture.

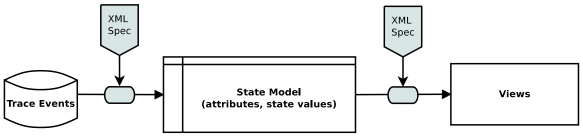


Figure 2. Architecture of the proposed declarative trace analysis approach.

A. Data Model

Since we aim to propose a generic solution without forcing users to use a specific trace format or pre-defined analysis, the analysis model should be as generic as possible. In other words, users will be able to define their own trace analysis, probably customized for a particular problem and tuned to focus on exactly what you're looking for.

To do so, we define notion of "state model" to let users keep track of the status of different under interest parameters of the system/application within the trace duration. The "state model" is in fact a generic "state" container. Each state refers to a "key", a set of "state values" and a time duration (a start time and an end time). The key, named "attribute", is to indicate an entity for which the values will be kept track. An attribute can be any system resources or any metrics or any custom user-defined entities for which users aim to store data in the model. "State values" in turn refer to different possible values assigned to the referred system elements (i.e., attributes) at different time points during the execution of the system.

For example, if a user wants to study the status of a CPU, in the above state model he can define an attribute as "CPU state" and the different possible status of that CPU during the system execution as "state values", whether it is assigned/busy or idle. Another example is defining the state for a process (i.e., attribute), whether it is running, blocked, waiting for CPU or waiting for IO (i.e., different state values). The definition of state is not limited to only the system resources. Users can define the state to keep track of status of any conceptual entities. For example to see if a status of network connection is connecting, established or closed. The state can also be used to store the statistics values of a metric. For example, the number of opened files, or number of established connections, number of bytes read or written, number of functions called, etc. are all examples of the metrics than can be stored as different states in the state model.

B. State Provider : Extract States from Events

Once the logical state model is constructed at a high level, the next step is to read the input trace events and extract the required information and populate the state model from the input data (Figure 2). State provider, which also contains a mapping between events and states, is responsible to build the state model. Trace events are passed chronologically through this state provider and determine what *changes* to the model are caused by each event. In a simple case, each event may change only a few states in the model, so a simple mapping table between events and states can be used. For instance, a ”file open” event changes the state of a file to ”opened”. In other cases, a series of events following each other in a particular order may be required to make a change in the model. For instance, a group of socket/network events with a particular order may be required to change the state of a network connection to ”half-opened” connection. In the latter case, a pattern of events might be required. The state provider designed in this system supports both types of the conversion.

In summary, state provider uses a simple or complex *state change* patterns to extract the state model from the input events. This mechanism called *state change* which is one of the main elements of designed language. In the most common case, the state changes can be seen as transitions in a finite state machine which specify what changes to the model are caused by each input event.

C. Visualization : Populate Views from the State Model

The state model constructed by the state provider can then be used by users to query the analysis information and provide an aggregated view of the underlying trace. It can also be used to populate the various data-driven views to display the outputs of the data analysis processes or the user supplied queries.

In this system, two generic Gantt-chart and XY-chart graphs are defined and users can populate these graphs dynamically from their models using data-driven specifications.

D. Model Container

We use the SHT (state history tree) data structure proposed in a previous work [11], to store the proposed data model. SHT is a special purpose disk-based database designed to store a huge amount of intervals for incrementally-arriving trace data [11]. This data structure is optimized for fast accesses on a rotational disks allowing fast search queries (with logarithmic time) on the stored interval for any given time [8].

Since SHT stores the data in interval format, it can be used as a generic model container to store the user-defined models, as long as the including data elements can be stored in interval format. In the proposed state model each state value between two consecutive state changes is modeled as an interval and can be stored in the SHT container. Figure 3 shows an example of how to store two consecutive state changes as an interval value.

As shown in Figure 3, Event e1 (t1) makes a ”state change” of the attribute atr1 from S0 to S1. At a later time, e2 (t2) changes the value of the same attribute Atr1 to S2. Since the state value for the Atr1 between t1 and t2 is S1, so it can be stored in an interval like [Atr1, S1, t1,t2]. This interval in fact indicates that the value for atr1 between t1, and t2 is S1. Figure 3 shows also the other intervals [Atr1,S0,0,t1) and [Atr1,S2,t2,T) for the other time ranges of the graph.

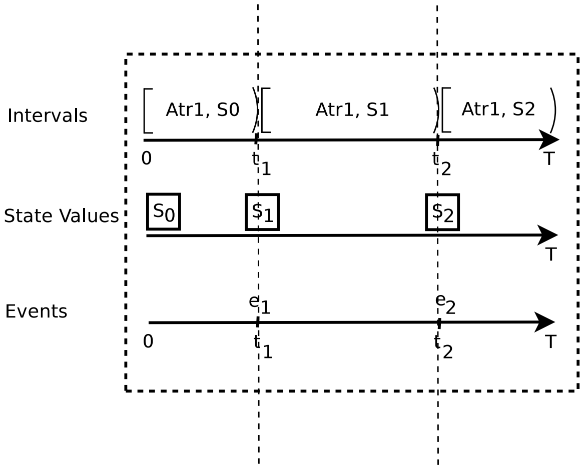


FIGURE 3. An example of state value changes and resulting intervals.

In addition to the interval tree to store the data intervals, SHT uses another tree structure called ”attribute tree” to organize the attributes. In this abstract data structure, attributes are accessible through their own specific paths, like in a file system, (for example ”`/CPUs/CPU0/ Current Thread`”). This allows the analysis to make easy accessing the attributes. An example of an *attribute tree* is shown in Figure 4.

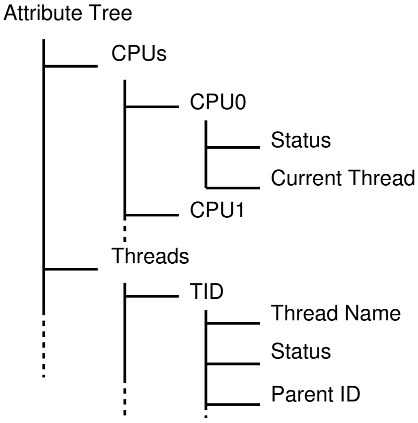


FIGURE 4. Example of an attribute tree

Please note that using the SHT interval container as the data store for our model does not force any modifications in the high-level models. Users define their own logical state model and the conversion between events and states without having to worry about the underlying container. The conversion from state changes to the internal intervals is a low-level task performed by the module, not the users.

III. LANGUAGE SPECIFICATION

As mentioned earlier, the proposed architecture lets user to define their own custom models for specifying the behaviour of an application or operating system. In this section, the detailed specification of the proposed language is provided.

To facilitate future functionality extensions of the language, it was decided to use XML with XSD schema for the syntax definitions. XML is extensible, widely used and capable of being easily integrated with other existing tools. A graphical user interface might be needed later to aid users in creating the models in relatively high level using graphical elements and generating XML specifications from these graphical models.

The developed descriptive language is able to create states from the input events and store them in the state model, and provides new analysis in a specific context.

A. Basic Definitions

In this section, we define the preliminary language operations necessary to define a analysis model. To clarify the definitions, traces format and events of LTTng kernel tracer [2] is used.

1) Access to attribute values: In order to access to a particular attribute, we use a path such as the following :

```
/Threads/100/Status
```
(1)

In Expression 1 The number 100 is in fact the thread number and the whole path specifies the status of the thread 100. Here, we only define the logical path of the attribute and do not discuss its possible values (which can be RUNNING, CRITICAL, WAITING, etc.). The possible values will be defined later in the state provider section.

Sometimes in the actual path of an attribute, it might be required to make a query and refer to another attribute. For example to call the current running thread of a specific CPU, we may write a path like is shown in Expression 2.

```
/Threads/${/CPUs/1/CurrentThread}/Status
```
(2)

Expression `${}` is therefore used as a path component to make a query on another attribute and replace the expression by the result, as shown in Expression 2. The final result, after replacing the query of the `${}`, will be a path like is shown in Expression 1.

In addition to make a query for another attributes, it is also possible to use an input event field as a part of an attribute path. An example is shown in Expression 3 in which `event/cpu_id` is used to access the cpu_id field of the input event, in the context of kernel traces.

$$\begin{aligned}&/Threads/\$\{/CPUs/\$\{event/cpu_id\}/...\\&...CurrentThread\}/Status\end{aligned}\tag{3}$$

In practice, for kernel traces, some information such as the thread id is not available in all events. It is thus necessary to use the context switch events to extract this information and store them in the state model of each CPU core, for later accesses. It is then possible to extract the current thread of each event by simply knowing its CPU's number and making a query to the state model. Expression 3 indicates in fact the path to the status of the current running thread of the event's CPU (i.e., current thread' status).

2) Assignment: Another possible operation is the assignment of a value to an attribute. This operation changes the value of an attribute, ending the previous state interval and starting a new one with the new value (remember the example shown in Figure 3).

```
/CPUs/${event/cpu_id}/Status = RUN_IN_USERMODE
```
(4)

The value can be a constant, as in Expression 4, a reference to another path in the model, or an event field, as in Expression 5.

```
/Threads/${event/tid}/Exec_name = /event/exec_name
```
(5)

3) Condition: Sometimes, we want to change a state value if a certain condition is met. So to make a complex model it is required to define conditional statements. A basic condition is based on the event type to specify the changes each event can make to the model. This condition type is somehow necessary to sort the different state changes by event type, to allow the user to easily correlate the changes with a trace event, in the state provider declarations.

Conditions can also be based on a field of event or another state value of the model. To do so, the same syntax to access the variables is used, with classical boolean operators `AND`, `OR`, and `NOT` for conditions. The condition shown in Expression 6 checks if the status of a specific file (event/fd) is OPEN and the filename is ".passwd".

```
/File/${event/fd}/Status == OPEN and ...
        .../event/filename == ".passwd"
```
(6)

It then becomes possible to choose the conditions, based on either the information contained in an event or the information already contained in the state model.

B. State Provider

As mentioned earlier, the state provider is the part that defines the way to extract state values from the input events. To be generic enough, a reference to the trace type, the name of the state model and some other information are included in the header of the state provider.

1) Locations, Constant Values and Variables: To identify the possible state values which correctly describe the model, users can define constant and variable values and use them later in the language. Example 7 shows two constant-value definitions. Values can be abstract values, e.g., `OPEN, CLOSED, RUNNING, STOPPED`, or a string that contains a payload, e.g., the executable name of a process.

```
<stateValue name="RUNNING" value="1" />
<stateValue name="STOPPED" value="2" />
```
(7)

`Location` element is used to define a shortcut name for a frequently used path of the attribute tree. Although not mandatory, these shortcuts may be used in state change declarations for conciseness and clarity purposes. Code 1 shows an example of the `Location` element. It actually corresponds to the logical attribute path of Expression 8.

In most cases, user wants to store a list of indexed properties, like the status of all CPUs, all running threads, or all opened files. To do so, we use a path with a wildcard, like `/Thread/*/Status`, where each possible value represented by $\star$ is a unique index. Here, the `tid` index is obtained from the input event field.

$$\texttt{/Threads/\$\{event/tid\}/Status} \tag{8}$$

Listing 1. Example of Location element

```
<location id=”CurrentThreadStatus”>
    <attribute constant=”Threads” />
    <attribute eventfield=”tid” />
    <attribute constant=”Status” />
</location>
```

2) Event Handler: While the events type could have been yet another field subject to conditions, it was decided to have an explicit *event handler*, a top-level structure that defines a namespace for each vent type. This choice simplifies the addition of rules for new trace events as well as helps to quickly specify what types of events are needed for an analysis. It may also give a feedback to the tracer to only collect data about some particular trace events.

Event Handler is a container for state changes, as shown in Listing 2 :

Listing 2. Example of Event Handler element

```
<eventHandler eventname=”sched_switch”>
  <stateChange>
    <attribute location=”CurThreadStatus”/>
    <value int=”$RUNNING” />
  </stateChange>
  <stateChange>
    <attribute location=”PrevThreadStatus”/>
    <value int=”$STOPPED” />
  </stateChange>
</eventHandler>
```

In this example, the *sched_switch* event causes two changes. It first updates the status of the current thread to ”running” and then changes the status of the previous thread to stopped.

3) State Change: The last part of the state provider is the transcription of the state changes, for which an example was shown previously. This construction contains a path and a value, possibly with a condition. For example :

$$\texttt{/Threads/\$\{event/tid\}/exec\_name= /event/execname} \tag{9}$$

Listing 3. Example of State Change element

```
  <stateChange>
      <attribute constant=”Threads” />
      <attribute eventfield=”tid” />
      <attribute constant=”exec_name” />
      <value eventfield=”execname” />
  </stateChange>
```

A condition can also be added, which will be shown as a complete example in the following section.

4) Example: Here is a simple example with traces generated from LTTng-UST (user-space) instrumentation [12]. The objective is to debug an application to know the duration it works and is active. We add two trace points : one at the beginning called `application:start`, and one at the end called `application:end`.

In our state model, we define two states : `RUNNING` and `STOPPED`. We know in advance that there will be several instances of the application. Therefore we define the attribute `Application/*/Status` path to access the state values.

Listing 4. Complete example of a trace analysis

```
<stateprovider analysisid=”app.ust”>
   <head>
      <tracetype id=”ust.ctf” />
      <view id=”gant.view” />
   </head>

   <stateValue name=”RUNNING” value=”1” />
   <stateValue name=”STOPPED” value=”0” />

   <location id=”App_Status”>
      <attribute constant=”application” />
      <attribute eventfield=”pid” />
      <attribute constant=”Status” />
   </location>

   <eventHandler eventname=”app:start”>
      <stateChange>
         <attribute location=”App_Status” />
         <value int=”$RUNNING” />
      </stateChange>
   </eventHandler>
   <eventHandler eventname=”app:end”>
      <stateChange>
         <attribute location=”App_Status” />
         <value int=”$STOPPED” />
      </stateChange>
    </eventHandler>
</stateprovider>
```

Gantt Chart view is used to display the analysis output which is shown in Figure 5. You can see in green the active duration of processes and in grey the stopped ones.

C. Filtering

For managing a large volume of data in the state model, filtering may be used to highlight the most interesting part of the data. Filtering can be used to display the only important data that obey the data-driven filtering criteria. It actually

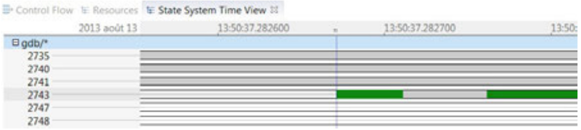


FIGURE 5. Example of a UST instrumentation

works by minimizing the volume of analysis by discarding the irrelevant information and retrieving only the desired data.

The proposed language supports filtering elements to help users to navigate easily the constructed system model as well as to specify triggers to debug the applications, or to detect the potential performance or security attacks. In addition, filtering can be used to add bookmarks in the trace, helping the user to navigate directly where an interesting behaviour is occurred (e.g., to bookmark the point that a problem is detected). Since the filtering patterns are also similar to the patterns used to convert the events to states, the same pattern syntax and processing engine is used for the filtering.

These filters create in fact new virtual states which help to explain the state intervals defined by the state provider. We don't use a persistent storage to store the filtering results, so the filters must be reexecuted and recalculated at every reload of the viewer.

The following example shows a filtering pattern to find when a specific application is preempted because of a lack of CPU resources. The new virtual state BLOCKED can be then used to highlight the interesting portion of the trace. Since the virtual states have the same characteristics as state intervals, we can use the same views to display them.

Listing 5. Example of filtering a trace state model

```
<filter name="filter_1">
    <if>
        <attribute location="App_Thread" />
        <attribute constant="Status" />
        <value int="$STATUS_WAIT_FOR_CPU" />
    </if>
    <then>
        <attribute location="Filter" />
        <attribute constant="Blocked" />
        <value int="$BLOCKED">
    </then>
    <else>
        <attribute location="Filter" />
        <attribute constant="Blocked" />
        <value int="$UNBLOCKED" />
    </else>
</filter>
```

D. Views

As mentioned previously, the proposed architecture supports two declarative visualization views to display the analysis results : Gantt charts and XY charts. The Gantt chart can be used to visualize activities along the execution time, for example, CPUs or Threads activities at different time points.

In other hand, XY charts can be used to display the statistics about the underlying system. Several useful metrics are extracted from the state model [13]. For instance, to measure the time spent by a process in state *"wait for a CPU"* or the amount of memory a specific process or a group of processes consume during a particular time period or during the the whole trace.

In the specification scripts, users can identify which parameters and which part of the state model can be used to display which graphical elements of the view. The colors, tool-tips are specifiable through the script as well. The following script (Listing 6) reveals the way to specify a Gantt chart view. An example of the result view was previously shown in Figure 5.

Listing 6. Specification of a graphical view in the proposed system

```
<timeGraphView id="controlflow">
  <head>
    <analysis id="kernel.linux.sp" />
    <label value="Thread Activity" />
  </head>
  <!-- Control Flow View -->
  <entry path="Threads/*">
    <display type="constant" value="Status" />
    <parent type="constant" value="PPID" />
    <name type="constant" value="Exec_name" />
  </entry>
</timeGraphView>
```

E. Language Limitation

The proposed language allows some operations to access and assign the memory of the state model. It is also possible to define conditions in the views. However, it is not included unrestricted conditional or unconditional branching. This prevents looping and accordingly infinite loops, insuring that the processing time is finite. This only allows a finite number of state changes for each event. Because of this limitation, our descriptive language is not Turing complete.

IV. APPLICATIONS AND PERFORMANCE ANALYSIS

In this section, we will examine applications that have been achieved with the proposed tool. The proposed tool, with the declarative language and generic use of the state model and views, is implemented in Trace Compass, which is publicly available on web [2]. The tests to validate the performance of the implemented tools have been performed under Ubuntu Linux 12.04, on a dual quad-core Intel Xeon E5405 2Ghz with 8GiB of RAM. In this project, LTTng [2] is used to generate trace events for the Linux applications. This tracer, available for the Linux (Kernel and user-space), is optimized for low overhead and collects kernel and user-space events.

A. Performance Analysis

We were able to compare the conciseness and performance between the declarative and hardcoded Java versions. Since the

[2] https ://projects.eclipse.org/projects/tools.tracecompass

Linux Kernel model was our starting point, it was expected that the required expressiveness would be provided.

1) Construction time: In this benchmark, we propose to evaluate whether there is a performance degradation between a version implemented using Java (hard-coded way) and a version using the proposed XML syntax (declarative way).

For this, we used two kernel traces : a 13.4 MiB trace available as a CTF sample in LTTng website[3] and a 100 MiB kernel trace. The tests were repeated 25 times to get an average value and standard deviation.

Trace 13.4 MiB	Java	XML
Average time (s)	8.687	8.979
Standard Deviation (s)	0.218	0.277
Min (s)	8.263	8.387
Max (s)	9.141	9.797

TABLE I
CONSTRUCTION TIME OF THE STATE MODEL FOR A 13.4 MIB KERNEL TRACE.

Trace 100 MiB	Java	XML
Average time (s)	49.359	50.025
Standard Deviation (s)	1.034	1.140
Min (s)	47.054	44.325
Max (s)	52.670	52.427

TABLE II
CONSTRUCTION TIME OF THE STATE MODEL FOR A 100 MIB KERNEL TRACE.

The results in Tables I and II show that the XML version is very slightly slower. However, the difference is smaller than the standard deviation between the different tests. Variations between instances are mainly associated with the garbage collection of the necessary objects to create and store state intervals. The main reason is to the (almost) similarity of the results is that in the proposed model, the declarative language is compiled and converted to internal JAVA modules and the trace events are examined by this internal modules. So, the possible time differences equals to the time required to parse, compile and convert the input specification (written in XML language) to the internal JAVA modules.

B. Generic Kernel Model

we mentioned that this work is generic and can be used for any trace format. It is because the language does not force any limitation on the format and name of the events that users can define. The only important part is that the users should mention in the language the way the system should deal with each input event and the way this event should be converted to states and stored in the state model. To prove this we have tested our model with trace events come from different tracers running in different operating systems (i.e., Linux and Windows kernel tracers).

For the Linux, we were able to easily represent the Linux Kernel model with our XML syntax. More impressive was the fact that the new declarative language was used to parse *Event Tracing for Windows (ETW)* kernel traces. Therefore, the method supports also Windows operating system kernel traces with the same level of information as Linux kernel traces. An example of both trace analyzes will be provided in the coming sections. The support of both Linux and Windows trace formats is already added to the Trace Compass tool and is usable by the public.

1) An Illustrative Example : Linux and Windows Comparison: The way to represent an operating system operations with a Gantt chart view, and to describe threads activities, is fairly common. However the strength of our model is that it can be easily interfaced to all platforms with different tracers.

By studying the ETW tracer on Microsoft Windows, we have noticed that this tracer has equivalent events and can be used to model the system in the same way. It was then possible, with a simple revision of the XML file, to get the same views, already available for Linux, with Microsoft Windows. This actually shows the independence of the work with the operating system and input trace.

This independence of the work with the input trace data/format is a big gain and can lead to use the same trace analysis tool for the different traces, different analysis with different purposes. In addition to that, it makes possible to compare different executions from different operating systems and different applications. For example, the tool can be used to compare the differences of how the same (or different) application behaves in different execution environments (e.g., different operations systems or different loads). For instance, to see the comparison of the Chrome browser executions in two different operating systems.

In this way, we have designed a simple test to compare the behaviour of the two operating systems : Windows and Linux. In this example. at every second we start a new process that makes a *CPU burn* and the objective is to see how these increasingly numerous threads are distributed on a single computer with 4 CPUs. The result is shown in Figures 6 and 7. The interesting part is that the both of these views are defined and generated decoratively without writing of even one line of JAVA code.

C. Multi-level Tracing

A second usage scenario for the proposed tool was to define more complex models based on the generic kernel model. For this objective, we have tried the same application but running in different levels (Kernel and User space) to integrate different sources of trace events.

1) Kernel and User Space Traces: We chose an already instrumented application, Google Chromium [14], whose architecture behaviour is only visible with both user-space and kernel traces. Indeed, it uses both numerous system-level threads and user-level task queues. The UST (User Space Trace) model of Chromium has already been defined and is used by the Chrome internal tracer[4]. This model use two event types to know the beginning and end of a code portion. Not all functions are instrumented, but only the key functions of the application.

[3] http://lttng.org/download

[4] chrome://tracing in the Chromium browser

In our experiment, we want to use these events to maintain a stack of the functions running for each thread. We use the *Begin* event to push the function name on the stack, and the *End* event to pop this stack.

By combining the kernel and UST data, we know what functions are running in each thread we well as when threads are preempted or which system calls are executed for each function. In Figure 8, we see the current stack in the ”State System Explorer” view, a view to debug the state model and see the values of different states of the different attributes.

As shown in Figure 8, we see that the stack depth is 4, and the top is *OnDispatchMessage*. The name of this function is also used as label in the green portion in the Gantt chart view.

2) Virtual Machines Monitoring: Another application is the instrumentation of a physical server which is hosting some virtual machines. The host and virtual machines are computers that can be traced independently with the separate tracers. However, there is a potential gain of information by grouping together the different traces, which is possible in our framework and shown in this experiment.

A simple example is to add the information of the CPU resources of the host in the virtual machine. We can then see if the virtual machine is running or preempted in the host machine. This information is then added to the model of the virtual machine. In this case, we see if the threads, thought to be running on the virtual CPUs, have real accesses to the physical CPUs, or if they are in fact preempted.

An example of virtual machines monitoring is displayed in Figure 9 (the idea of this example is taken from one previous work [15]). It shows CPU statuses of two virtual machines hosted in the same physical machine. There are three general states in this figure : when the virtual CPU is active while the second one is preempted, the first virtual CPU is preempted while the second one is active and running and finally the case that both virtual CPUs are preempted. The first two cases show the fact that there is competition with the virtual CPUs to acquire the main CPU (the shared resource) to execute their codes. The last case, however, indicates the fact that there is possibly another thread competing with these two virtual machines on that shared resource (the physical CPU).

Creating such analysis from the different sets of trace events (gathered from Linux, Windows or even mobile devices) is completely possible using the declarative expressions in our proposed framework, without needing to have a pre-support of such analysis in the trace tool.

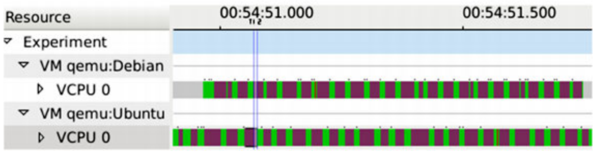


FIGURE 9. Competing of two virtual CPUs to be executed in the physical CPU.

3) Anomaly Detection: As mentioned earlier, it is possible to use the proposed method to detect system problems that can not be detected (or difficult) with the existing tools. Here we show an example of the way we use our method to find a bug in Google Chromium application.

Multilevel trace analysis is one of the advantages of our work with respect to the other previous works (as well as the Chromium built-in tracer). For instance, in comparison with the Chromium built-in tracer, our tool can use the events of operating system kernel level, e.g., information about system calls, CPU schedules, disk blocks, etc. in addition to what the Chromium tracer can provide for the analysis.

The Chromium architecture is a highly parallel architecture, in which more than a dozen of executions are executed by each process. However, from the operating system level view, there is at most one thread of execution which is assets per processor at each time. Kernel traces can help us to see the active thread as well as to see if a particular thread is preempted by other processes at every moment of the trace.

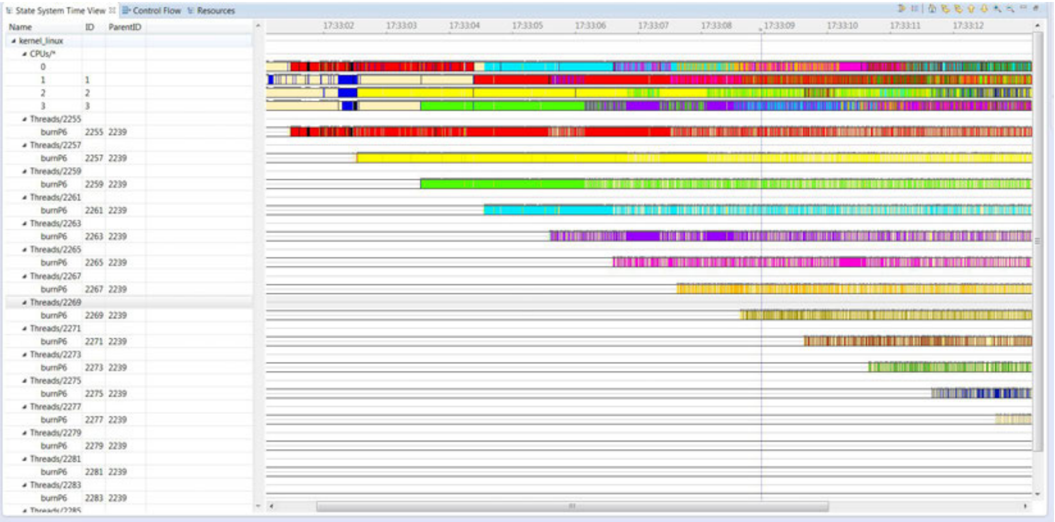
This information is not available at the application level. For example, with only using the Chromium trace events, one will not be able to see what is really happening between the start and end points of a function which may prevent to detect some design or execution problems.

To prove this, we use traces has made with the help of a virtual machine with two virtual processors. It will therefore be two active simultaneous executions at each time (i.e., one for each CPU). The expected behaviour is to never block the Chromium browser by the system calls of disk IO (reading/writing) tasks. To achieve this goal, the main thread crRenderMain calls the ChildIOThread to perform the input/output requests (Figure 10). This IO thread is executing in a lower priority rather than the main execution thread. This example is shown in Figure 10, in which green and blue shows the active states of the threads, while the yellow is waiting and orange is preempted states.

In this representation, the main execution thread has priority over the one who manages the inputs/outputs. The latter may be executed only when the other does not have more to do. Otherwise, it is preempted. However, under certain conditions, the behaviour of Chromium is not the one that is expected. Figure 11 shows an execution in which each time the main execution thread sends a message to start a task, it is preempted by the IO maintainer thread. This type of behaviour is very penalizing for the user interface. Although it has no impact on the overall calculation time of the software, this behaviour slows the display and negatively affects the perception of the UI fluidity.

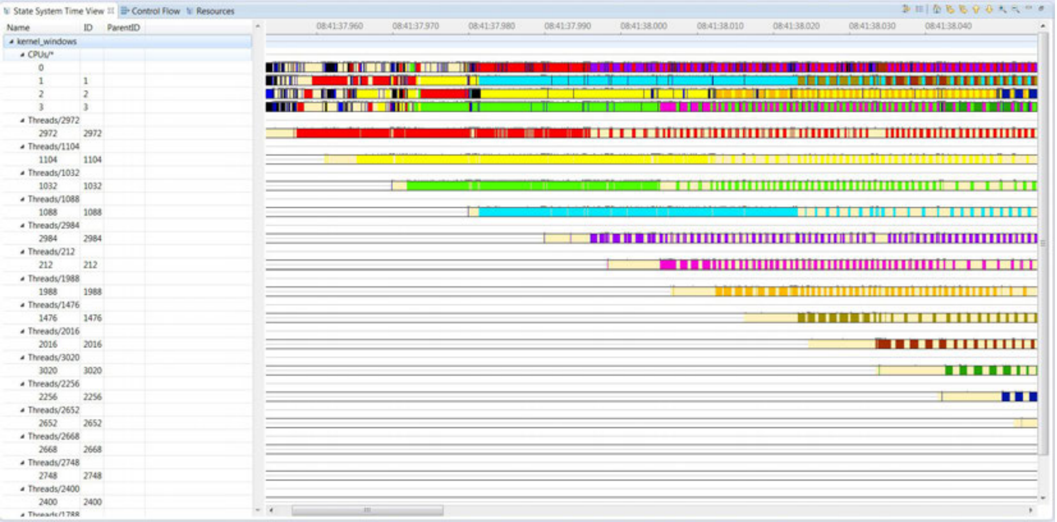
It’s easy to automatically detect such problems by using filters on state model in our system. Just specify a shape condition : if CrRenderMain == Preempted and also Chrome_ChildIOThread == Current Active Thread.

A Design Challenge: Combining multiple kernel traces together presents a new challenge for the state model. Indeed, the number of attributes increases very rapidly with the number of virtual machines. Moreover, there query time cost in performance increases linearly with the number of attributes, as shown in Figure 12. This is why it is necessary to consider the



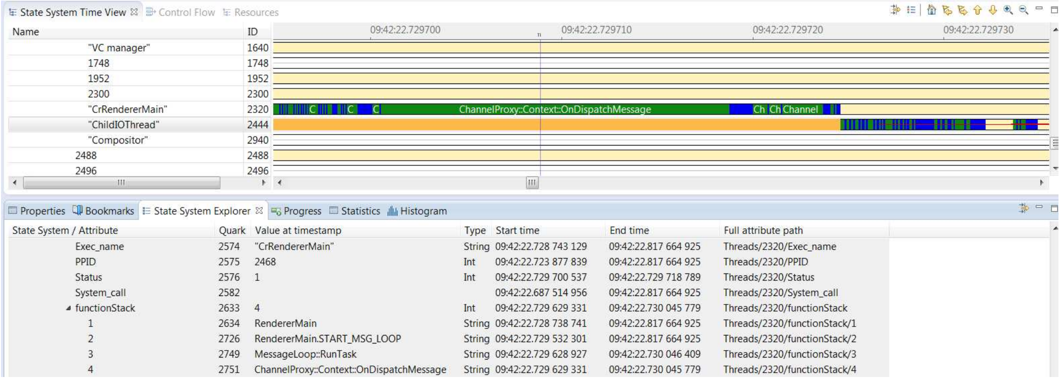
Name	ID	ParentID
kernel_linux		
CPUs/*		
0		
1	1	
2	2	
3	3	
Threads/2255		
burnP6	2255	2239
Threads/2257		
burnP6	2257	2239
Threads/2259		
burnP6	2259	2239
Threads/2261		
burnP6	2261	2239
Threads/2263		
burnP6	2263	2239
Threads/2265		
burnP6	2265	2239
Threads/2267		
burnP6	2267	2239
Threads/2269		
burnP6	2269	2239
Threads/2271		
burnP6	2271	2239
Threads/2273		
burnP6	2273	2239
Threads/2275		
burnP6	2275	2239
Threads/2277		
burnP6	2277	2239
Threads/2279		
burnP6	2279	2239
Threads/2281		
burnP6	2281	2239
Threads/2283		
burnP6	2283	2239
Threads/2285		

FIGURE 6. Thread activities view in Linux.



Name	ID	ParentID
kernel_windows		
CPUs/*		
0		
1	1	
2	2	
3	3	
Threads/2972		
2972	2972	
Threads/1104		
1104	1104	
Threads/1032		
1032	1032	
Threads/1088		
1088	1088	
Threads/2984		
2984	2984	
Threads/212		
212	212	
Threads/1988		
1988	1988	
Threads/1476		
1476	1476	
Threads/2016		
2016	2016	
Threads/3020		
3020	3020	
Threads/2256		
2256	2256	
Threads/2652		
2652	2652	
Threads/2668		
2668	2668	
Threads/2748		
2748	2748	
Threads/2400		
2400	2400	
Threads/1788		

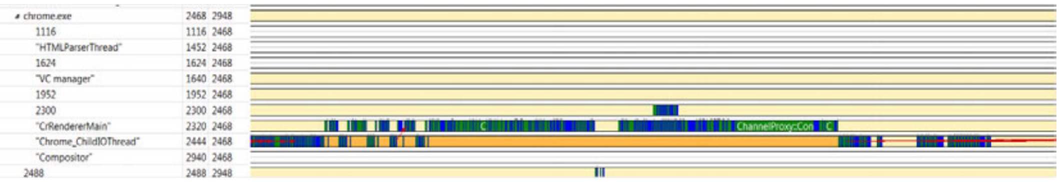
FIGURE 7. Thread activities view in Windows.



Name	ID
"VC manager"	1640
1748	1748
1952	1952
2300	2300
"CrRendererMain"	2320
"ChildIOThread"	2444
"Compositor"	2940
2488	2488
2496	2496

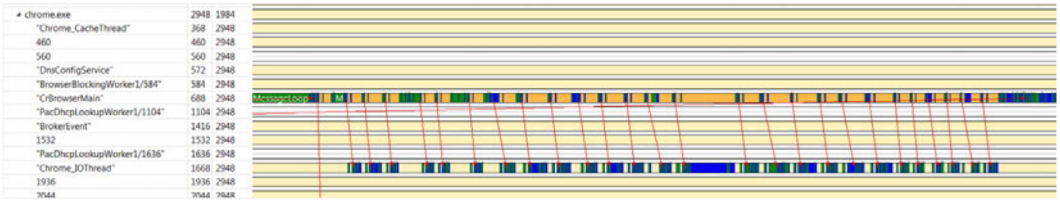
State System / Attribute	Quark	Value at timestamp	Type	Start time	End time	Full attribute path
Exec_name	2574	"CrRendererMain"	String	09:42:22.728 743 129	09:42:22.817 664 925	Threads/2320/Exec_name
PPID	2575	2468	Int	09:42:22.723 877 839	09:42:22.817 664 925	Threads/2320/PPID
Status	2576	1	Int	09:42:22.729 700 537	09:42:22.729 718 789	Threads/2320/Status
System_call	2582			09:42:22.687 514 956	09:42:22.817 664 925	Threads/2320/System_call
functionStack	2633	4	Int	09:42:22.729 629 331	09:42:22.730 045 779	Threads/2320/functionStack
1	2634	RendererMain	String	09:42:22.728 738 741	09:42:22.817 664 925	Threads/2320/functionStack/1
2	2726	RendererMain.START_MSG_LOOP	String	09:42:22.729 532 301	09:42:22.817 664 925	Threads/2320/functionStack/2
3	2749	MessageLoop::RunTask	String	09:42:22.729 628 927	09:42:22.730 046 409	Threads/2320/functionStack/3
4	2751	ChannelProxy::Context::OnDispatchMessage	String	09:42:22.729 629 331	09:42:22.730 045 779	Threads/2320/functionStack/4

FIGURE 8. UST instrumentation combined with kernel events.



chrome.exe	2468	2948
1116	1116	2468
"HTMLParserThread"	1452	2468
1624	1624	2468
"VC manager"	1640	2468
1952	1952	2468
2300	2300	2468
"CrRendererMain"	2320	2468
"Chrome_ChildIOThread"	2444	2468
"Compositor"	2940	2468
2488	2488	2948

FIGURE 10. Google Chrome execution threads



chrome.exe	2948	1984
"Chrome_CacheThread"	368	2948
460	460	2948
560	560	2948
"DnsConfigService"	572	2948
"BrowserBlockingWorker1/584"	584	2948
"CrBrowserMain"	688	2948
"PacDhcpLookupWorker1/1104"	1104	2948
"BrokerEvent"	1416	2948
1532	1532	2948
"PacDhcpLookupWorker1/1636"	1636	2948
"Chrome_IOThread"	1668	2948
1936	1936	2948
2044	2044	2948

FIGURE 11. Fault identification example for Google Chrome application

way to split the model into several separate internal structures. Please note that this splitting operation is a low level task which i performed by the tool and is hidden from the high level users.

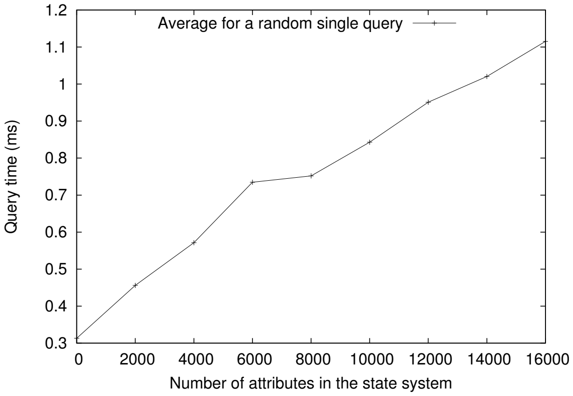


FIGURE 12. Average time to query a random interval in function of the number of different attributes in the state system.

The attribute tree is divided into groups of attributes (examples : CPUs, Threads, Files...). In the case of virtual machines monitoring, we use the name of the computer node as the first level (see Figure 13). Then, each group may be stored in a separate SHT tree to divide the problem size. A strategy to define folders as `mount` points, like in POSIX filesystems, could easily be added in the XML state provider header. This would allow the user to choose the backend used for each subfolder in the state model.

D. Query Optimization

Another interesting performance challenge is with queries for the views. In order to have a good performance level, it is essential to minimize the number of queries in the system. Several query types are supported, and the performance depends on the information that we want to display. We detail a few use cases in this section.

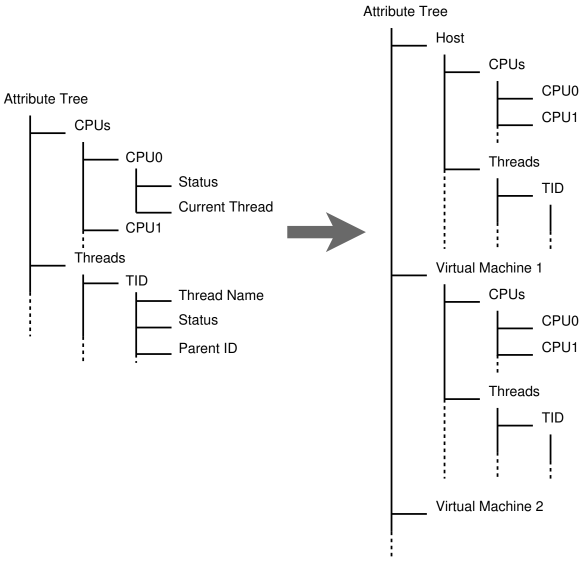


FIGURE 13. Adaptation of the structure of the attribute tree for the VM's application.

1) Complete Query: The most expensive case is when we try to put bookmarks in places where we have detected an anomaly. Since it is necessary to check all state intervals for an attribute, possibly scanning the whole state model, these queries can be long and time consuming, especially if the attribute often changes its state. For example, CPUs may change scheduling state many times within a single second in a kernel trace.

Initially, we do not have much information on the nature of the filter, so we cannot easily predict the time needed to get the query information. But, if users can predict in advance the filters they will need (which is the case in some applications), it would be possible to integrate the filter into the state model construction and pre-construct the required model. Then, the filter results will be available quickly in the user interface.

2) Resolution Query: Another optimization is the strategy implemented to quickly populate the Gantt chart view. The Gantt chart is generally rendered by reading the whole information stored for an attribute along the time axis. But reading all of the information for an attribute (like a ”CPU state” attribute that may be changed a lot during only a fraction of a second) may consume much of display time.

However, in the reality we can prevent reading all state information of an attribute. This is achieved by adding the notion of resolution. The size of the visible screen (the whole displaying time duration) or the number of available pixels for the same duration of a trace can play a key role to decide how to query the underlying model. Suppose the case that the screen is used to display only 5 seconds of execution. Of course the amount and dept of information that is queried and displayed is completely different with the case that the same screen is assigned to display the 5 hours of execution. In the former case, a pixel of screen is assigned to a small range of trace data, while in the latter case it is assigned for a very larger trace duration. Therefore, the same querying and rendering algorithm should not be used for both cases, otherwise the performance of the view will be degraded for the large execution duration.

To solve this problem and achieve almost the same query time for all trace duration and all display sizes, we add the notion of resolution. In this optimization, the algorithm goes pixel by pixel, queries only a few states within the duration of each pixel (e.g., the starting and ending states of the pixel, when the resolution value of a pixel is 2), and ignores the remaining state intervals. If these queries can specify the view color and value for that pixel it is passed and processing of the next pixel is started, otherwise a black dot is put in the view, indicating that there are some more information and values existing in that pixel. This black dot notify users that they can zoom in on this pixel to dig into that and get more information. This strategy is used to have a quick and significant overview of the trace at high level views without querying the whole state model.

This strategy may reduce the precision of the view and the amount and even the correctness of information a view can naturally display, but the gain is a speed rendering of the view (because the query time and display time is reduced). So, we do not in fact suggest this strategy for any kind of views. However, in some use cases this strategy can be very useful, specially when a quick displaying of an overview of the underlying model can help a lot and guide users directly to the problematic or interesting points of the execution very quickly (e.g., statistics use cases).

As an example, suppose we use the state model to store and render some statistics about our application of interest. Bu defining resolution values, it is not necessary to query all underlying state intervals to display the values. The resolution value in fact defines a sampling ratio to estimate characteristics of the whole interval. For example, to display a bar chart about the statistics of a thread execution (if the thread time is consumed by user space, it is executed in Kernel mode or it is in Blocked or Waiting states), we can query a fixed number of state intervals (based on the pre-defined resolution value) instead of querying all stored values within the state model, and define a confidence interval for this metric.

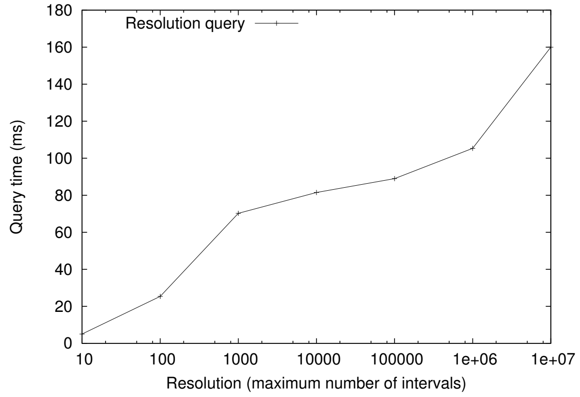


FIGURE 14. Average query time in function of the maximum number of intervals.

This optimization gives a logarithmic performance gain for the query time, depending on the maximum number of intervals we want to query in total, as shown in Figure 14. The resolution provides information for every iteration step. However, if the state interval is longer than the iteration step, we don't make a query for each step. This way, on average, we have a logarithmic gain, as shown in Figure 14.

3) Partial query with a time Range: The last case is when the user wants to display the results of a filter (virtual states) in the Gantt chart view to highlight some certain sections. A possible optimization is to calculate and query only the filtered time ranges. This technique is very responsive and is already used to populate the Gantt chart view when we use the zoom.

This query type is useful to reduce the query time. We have a linear improvement of the query time performance, see Table III and Figure 15. In addition, it is possible to combine this optimization with the resolution optimization.

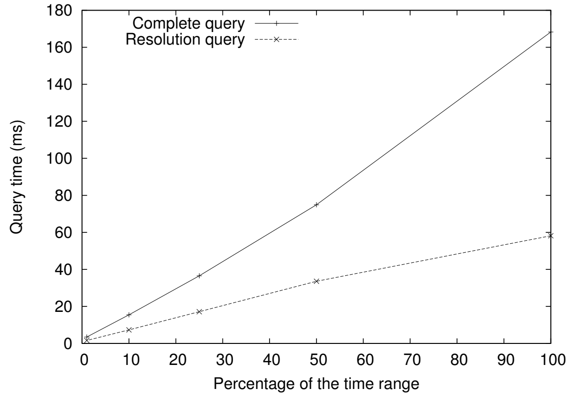


FIGURE 15. Average query time in function of the time range percentage.

Query	Complete (ms)	Resolution (ms)
Full range	168.3	57.38
50% range	74.90	33.60
25% range	36.52	17.16
10% range	15.44	7.32
1 % range	3.46	1.62

TABLE III
AVERAGE QUERY TIME IN FUNCTION OF THE THE TIME RANGE PERCENTAGE.

V. LITERATURE REVIEW

In this architecture, we use a stateful approach to model the system and give the user a comprehensive image of the application runtime behaviour. For example, the state of a process may change over time between states *start*, *running*, *waiting* and *stopped*. These changing states of a process is stored in an "attribute" that is named *status of the process*. The approach to index and retrieve the system state history has been used previously [16], [17]. The states selected to model the system are very important, and depend on the system and the problem we want to investigate. To study a performance degradation for example, we should track states of important resources (e.g. what are the CPU usage, the currently running thread, the files being accessed, or the network usage). Such metrics can help administrators understand the problem and possibly find a way to eliminate the underlying cause.

An approach to model the state from a system trace has already been studied [8] and implemented in the Eclipse Tracing and Monitoring Framework, Trace Compass[5]. It is based on a state manager and a special purpose database, State History Tree (SHT), used to efficiently store, navigate and display the state in the trace analysis softwares.

State History Tree (SHT)

With the huge trace data size, a special purpose database was designed to store all produced state intervals on hard disk [11]. The general idea of the approach is to incrementally extract and store the information of each relevant trace event and create different interval values in a custom designed database called State History Tree (SHT). This *State History Tree* allows the state system to make fast queries for any system parameter and attribute at any time during the trace. Furthermore, all state intervals are inserted by sorted end time. The State History Tree uses this property to optimize its layout for fast access on a rotational disk. This property obviates the need for re-balancing the tree, but preserves the property of logarithmic search. As a result, this data structure is well optimized to be used with trace files as large as 1TB.

A. Trace Analysis Tools

Several tools exist to analyze and visualize execution traces. Viewers like LTTV (Linux Tracing Toolkit Viewer)[5], Jumpshot [6] or Triva [7] display different analysis metrics of the underlying system execution (CPU usage, memory consumption, critical path analysis, etc.). Trace Compass[6] is another tool used to perform different trace analysis on traces collected from different sources (e.g., LTTng Traces, Network packet traces or custom defined traces, etc). Trace compass supports the aforementioned State History Tree, to manage the states of the system parameters. It provides various views like statistics view, Gantt charts and histograms. You can see in Figure 16 the data representation for Linux kernel traces : CPU usage, threads activities, statistics for the number of events, etc.

However, a limitation of these tools is that it is only available for a particular trace type generated by a specific tracer. Moreover, they only offer some specific views forcing users to use only specific set of analysis. Users are not able to define their own custom analysis based on their data and based on their specific requirements. However, In the new proposed architecture, the event-to-trace conversion (state provider), underlying state model and the display views are completely generic, and easily definable and customizable.

B. Descriptive Languages

There are many types of languages dedicated to system analysis. Interesting reviews of trace analysis systems are available from Matni [18] and Waly [19]. Declarative languages for patterns in network traces and logs are used by SNORT [20] or SECnology [21].

SNORT is an open-source Network Intrusion Detection System based on a collection of rules. This software provides a simple declarative syntax for defining intrusions in network connection packet traces. Nonetheless, by looking at each packet in isolation, this technique alone is not very efficient for defining complex analysis such as those in State Providers.

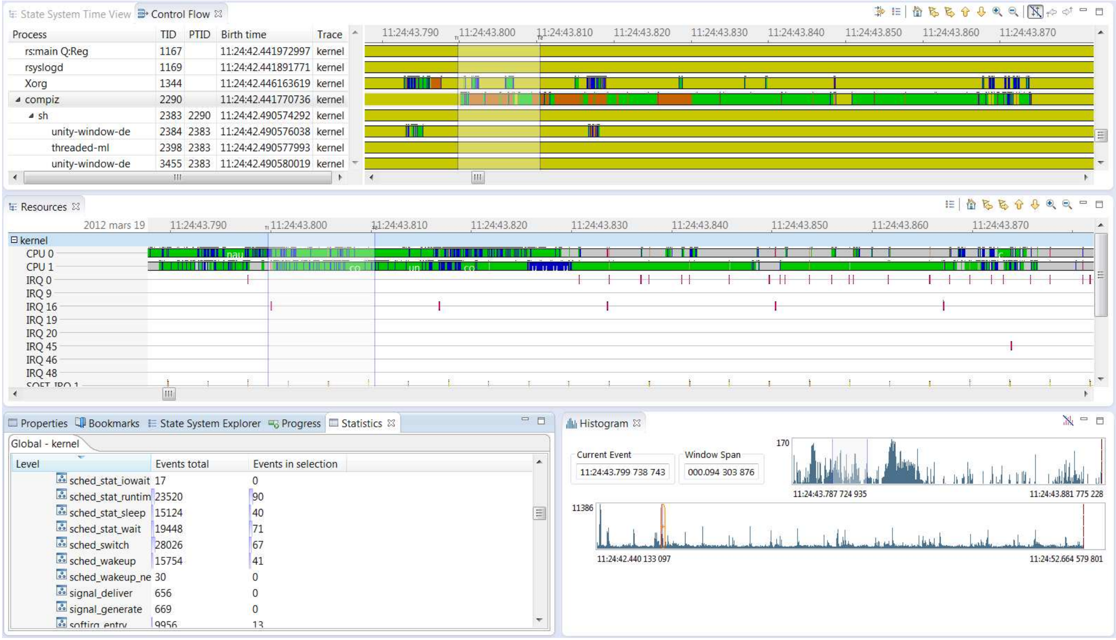
Imperative languages like RUSSEL (RUle-baSed Sequence Evaluation Language) [22] offer better expressiveness. Rules can trigger other rules. If rules are viewed as procedures, it is similar to procedural languages.

Another language is the D language, designed by DTrace [3] to dynamically define the instrumentation probes. However, this is more of a generic imperative language. SystemTap [4], another Linux kernel tracer, also provides a similar imperative scripting language, triggered by kernel-level events.

Automata-based languages are closer to the requirements of defining state transitions from events. This kind of language uses a finite state machine to describe the problem, with states, transitions and actions. STATL (State Transition Analysis Technique Language) [23] is a good example of a generic state machine diagram language that is extensible and usable by different applications in intrusion detection field. However, STATL is not completely declarative because the users should detail the transitions and the way they performed.

However, these languages are not necessarily adapted to use with a backend data model like a state model, which is closer to a database. They are also designed to specific domains and not to work with any trace data. On the other hand, query languages like SQL are limited to only tabular data rather than un/semi structured trace data. What we actually propose in this paper is a generic trace-specific language to extract users specific and custom purpose model from the trace data

[5] http://www.eclipse.org/linuxtools/projectPages/lttng/
[6] http://www.eclipse.org/linuxtools/projectPages/lttng/



Process	TID	PTID	Birth time	Trace
rs:main Q:Reg	1167		11:24:42.441972997	kernel
rsyslogd	1169		11:24:42.441891771	kernel
Xorg	1344		11:24:42.446163619	kernel
compiz	2290		11:24:42.441770736	kernel
sh	2383	2290	11:24:42.490574292	kernel
unity-window-de	2384	2383	11:24:42.490576038	kernel
threaded-ml	2398	2383	11:24:42.490577993	kernel
unity-window-de	3455	2383	11:24:42.490580019	kernel

Global - kernel

Level	Events total	Events in selection
sched_stat_iowait	17	0
sched_stat_runtim	23520	90
sched_stat_sleep	15124	40
sched_stat_wait	19448	71
sched_switch	28026	67
sched_wakeup	15754	41
sched_wakeup_ne	30	0
signal_deliver	656	0
signal_generate	669	0
softirq_entry	9956	13

Current Event: 11:24:43.799 738 743

Window Span: 000.094 303 876

FIGURE 16. Multiple data views for a Linux Kernel Trace display with TMF

and use this model to analyze and display the desired outputs, used in system performance analysis as well as attack and intrusion detection purposes.

VI. Conclusion and Future Work

There is a tremendous amount of data available in execution traces and logs. However, it remains difficult for the developer or the system administrator to extract the right information to find the causes of his problems. Trace analysis software and trace viewers help to have meaningful analysis and graphical representations, but these analysis and representations are often designed for specific purposes, and not very adaptable to other usages and contexts.

In this paper, we presented a new tool architecture based on a generic declarative specification. This framework allows the developer to put his knowledge of the product directly inside a model that can be used by the viewer to display synthetic information. The framework proposes a generic way enabling users to define their input trace events, their custom model, the effect of each event in their model and finally the way of displaying the analysis outputs.

We have shown throughout this paper many successful applications of this proposed architecture. This work has generalized the way to model the state information of a system. It is now possible to obtain a detailed view of the operating system/application internals and compare them with different systems of completely unrelated origin, like Linux and Windows. In addition, we demonstrated the use of this approach to model more complex systems with multi-level traces, by combining User-space tracing and Kernel tracing, or Kernel tracing in several virtual and physical computers.

However, the possibilities are even greater. This declarative specification describes generic models and events. We can use it to create models with network events, telephony servers, financial records, etc. Moreover, the XML syntax is extensible. Thereby, the next step is to add more features, like critical path analysis and more visual view types. Another possibility for future work is to optimize the framework by applying a parallel way of event parsing and construction of the users defined models.

Références

[1] M. Desnoyers and M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium) 2006*, pp. 209–224, 2006.

[2] M. Desnoyers and M. Dagenais, “Lttng : Tracing across execution layers, from the hypervisor to user-space,” in *Linux Symposium*, p. 101, 2008.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC 04, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2004.

[4] F. C. Eigler and R. Hat, “Problem solving with systemtap,” in *Proc. of the Ottawa Linux Symposium*, pp. 261–268, Citeseer, 2006.

[5] J.-H. Deschênes, M. Desnoyers, and M. R. Dagenais, “Tracing time operating system state determination,” *Open Software Engineering Journal*, vol. 2, pp. 40–44, 2008.

[6] O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward scalable performance visualization with jumpshot,” *Int. J. High Perform. Comput. Appl.*, vol. 13, pp. 277–288, Aug. 1999.

[7] L. M. Schnorr, G. Huard, and P. O. A. Navaux, “Towards visualization scalability through time intervals and hierarchical organization of monitoring data,” in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID 09, (Washington, DC, USA), pp. 428–435, IEEE Computer Society, 2009.

[8] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data,” in *Runtime Verification* (A. Legay and S. Bensalem, eds.), vol. 8174 of *Lecture Notes in Computer Science*, pp. 219–234, Springer Berlin Heidelberg, 2013.

[9] N. Ezzati-Jivan and M. R. Dagenais, “Stateful synthetic event generator from kernel trace events,” *Advances in Software Engineering*, January 2012.

[10] A. Hamou-Lhadj, S. S. Murtaza, W. Fadel, A. Mehrabian, M. Couture, and R. Khoury, “Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction,” in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS ’13, (New York, NY, USA), pp. 328–335, ACM, 2013.

[11] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, “State history tree : an incremental disk-based data structure for very large interval data,” in *2013 ASE/IEEE International Conference on Big Data*, 2013.

[12] J. Blunck, M. Desnoyers, and P.-M. Fournier, “Userspace application tracing with markers and tracepoints,” in *Proceedings of the Linux Kongress*, 2009.

[13] N. Ezzati-Jivan and M. R. Dagenais, “A framework to compute statistics of system parameters from very large trace files,” *ACM SIGOPS Operating Systems Review*, vol. 47, pp. 43–54, Jan. 2013.

[14] “Trace event profiling tool, http ://www.chromium.org/developers/how-tos/trace-event-profiling-tool.”

[15] M. Gebai, F. Giraldeau, and M. R. Dagenais, “Fine-grained preemption analysis for latency investigation across virtual machines,” *Journal of Cloud Computing : Advances, Systems and Applications*, vol. 3, no. 1, p. 41, 2014.

[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 105–118, oct 2005.

[17] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, “Correlating instrumentation data to system states : a building block for automated diagnosis and control,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation -Volume 6*, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[18] G. Matni and M. Dagenais, “Automata-based approach for kernel trace analysis,” in *Canadian Conference on Electrical and Computer Engineering, 2009. CCECE 09.*, pp. 970–973, May 2009.

[19] H. Waly, “A complete framework for kernel trace analysis,” Master’s thesis, Laval University, 2011.

[20] M. Roesch *et al.*, “Snort : Lightweight intrusion detection for networks.,” in *LISA*, vol. 99, pp. 229–238, 1999.

[21] “Secnology, http ://www.chromium.org/developers/how-tos/trace-event-profiling-tool.”

[22] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu, “Asax : Software architecture and rule-based language for universal audit trail analysis,” in *Computer Security—ESORICS 92*, pp. 435–450, Springer, 1992.

[23] S. Eckmann, G. Vigna, and R. Kemmerer, “Statl : An attack language for state-based intrusion detection,” *Journal of Computer Security*, vol. 10, no. 1/2, pp. 71–104, 2002.