Hindawi Publishing Corporation
Advances in Computer Engineering
Volume 2016, Article ID 9467181, 12 pages
http://dx.doi.org/10.1155/2016/9467181

Research Article

Problem Detection in Real-Time Systems by Trace Analysis

Mathieu Côté and Michel R. Dagenais

Department of Computer and Software Engineering, École Polytechnique de Montréal, P.O. Box 6079, Station Downtown, Montreal, QC, Canada H3C 3A7

Correspondence should be addressed to Mathieu Côté; mathieu.cote@polymtl.ca

Received 29 July 2015; Revised 24 November 2015; Accepted 17 December 2015

Academic Editor: Valeriy Sukharev

This paper focuses on the analysis of execution traces for real-time systems. Kernel tracing can provide useful information, without having to instrument the applications studied. However, the generated traces are often very large. The challenge is to retrieve only relevant data in order to find quickly complex or erratic real-time problems. We propose a new approach to help finding those problems. First, we provide a way to define the execution model of real-time tasks with the optional suggestions of a pattern discovery algorithm. Then, we show the resulting real-time jobs in a Comparison View, to highlight those that are problematic. Once some jobs that present irregularities are selected, different analyses are executed on the corresponding trace segments instead of the whole trace. This allows saving huge amount of time and execute more complex analyses. Our main contribution is to combine the critical path analysis with the scheduling information to detect scheduling problems. The efficiency of the proposed method is demonstrated with two test cases, where problems that were difficult to identify were found in a few minutes.

1. Introduction

Real-time systems are characterized by their timing constraints. They are composed of real-time tasks that will each generate a sequence of jobs with a priority and a deadline.

The moment at which a new job has to be executed is called the arrival time, and the moment at which a job actually starts to be executed is called the start time. If the jobs arrive at fixed interval, the task is called periodic; otherwise it is called sporadic. Periodic tasks are often driven by timer, like the processing of video frames. On the other hand, sporadic tasks are often driven by interrupts, like the response to a user action. In both cases, there will be deadlines, but it will be deadlines relative to the start time for the sporadic tasks and absolute deadlines for the periodic ones.

To avoid unwanted consequences, those deadlines must be met by the real-time jobs. However, when only a few deadlines are missed, it can be hard to identify the underlying cause, due to the numerous components involved in the systems and their interactions. Because of this intermittent problem occurrence, profiling tools may have difficulty to pinpoint the source of the problem. The numerous jobs that went according to the specifications will be taken into account in the resulting statistics and hide the rare problematic ones.

In that situation, tracing can be useful or even essential. It consists in collecting selected events during the execution of a program and the time at which they occurred. It is then possible to analyse the interesting parts of the resulting trace. However, the trace can be very large and it can be difficult to identify those interesting parts. This motivates the need for specialized tools to help developers, by guiding them to efficiently find the problems.

Our objective is to develop such a tool. To test our concepts, we used a Linux kernel with the PREEMPT_RT patch. This was shown to provide excellent real-time response with the Linux kernel. In addition, we used the LTTng tracer, characterised by a very low overhead [1]. This tracer has already been tested on a PREEMPT_RT patched Linux kernel with good results [2]. We implemented our analysis as a plug-in within Trace Compass, an open source and flexible trace visualiser in the Eclipse framework. Trace Compass is highly scalable and provides a powerful infrastructure, including interesting analyses like the critical path computation [3]. More precisely, we implement a new view structure to show elements in a Comparison View instead of a linear time based

view, we add four new views that will be explained in the next sections, and we add a new analysis to populate our views. We also ensure that our work was interacting correctly with the actual code, mostly in terms of views synchronisation.

Because they are frequent in real-time systems, we decided to focus on problems related to scheduling and priority. The scheduler is the component that selects the threads that will be executed next on the CPUs. Each thread has a scheduling policy and a static priority that are used by the scheduler to take decisions. To be able to analyse the various tasks, we need a method that can support the different scheduling policies available on Linux. In addition, we want to detect priority inversions, when a higher priority task is needlessly waiting on a lower priority task, usually because the latter is holding a lock while waiting on a third task of medium priority. We also want to support the different protocols used to avoid this inversion, like the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) [4]. In PIP, lower priority threads inherit priority from higher priority threads waiting on them, while in PCP, threads priority is boosted when they obtain a resource, to the highest priority of the threads that can obtain it.

We first present related work in the fields of real-time tracing and pattern discovery. We then describe our new approach to efficiently solve scheduling problems with a real-time task in four steps. The first step is to let the users define the execution model of the task jobs with the optional help of a pattern discovery algorithm. Based on that model, the second step is to locate all the corresponding jobs in the trace. The third step is to select interesting jobs from a Comparison View that highlight those that are problematic. The last step is to execute different analyses on the corresponding trace segments. These analyses include our main contribution which is to combine the critical path analysis with the scheduling information to quickly detect scheduling problems.

Thereafter, the paper continues with the different options offered to the users and the implications in terms of execution time. We also show the different views that have been prototyped to display the results. Then, we present different examples, typical of industrial problems, that can be efficiently solved using our tool. We conclude on the possible next steps for future work.

2. Related Work

In this section, we will review the studies that focus precisely on trace analysis for real-time systems. First, a pattern language is defined in [5] and the trace is iterated until the pattern is found. This is used to detect security attacks. However, this is not to find problems but to detect occurrences when you already know the representation of the problematic situation in the trace.

In [6], the authors present a method to identify periodic patterns using the gap between events of the same type. Then, they try to see if those patterns are in conflict by correlating the perturbation in the periodicity of a certain pattern, with the activity of the others. This works well with high-level events, like a function entry, that can be associated with a specific real-time task. However, the same low-level event type, like a scheduling event, may be used to define many different task types. Another limitation is that it only works for periodic tasks and thus will not handle sporadic ones.

A method to extract useful metrics based on kernel traces is defined in [7]. The authors used different state machines for the metrics, like the usage rate of system resources, the running time, or the response time. However, there are multiple constraints, because the different state changes will depend on the scheduling policies and the method used to avoid priority inversions. In the present situation, their work will only compute valid results for one specific combination. It will also provide some metrics but not pinpoint problematic executions.

A similar method is used in [8]. However, in addition to the metrics, the authors also retrieve the task intervals. Then, they present these in a comparative view, sorted by the longest interval time, to help viewing the differences. Users will typically focus on the analysis of the jobs with the longest times first, since these are more likely to be problematic for the real-time performance. However, there are some limitations. Because it is a fixed model, this will again only work well with a specific scheduling policy and method used to avoid priority inversions. In that work, it was geared towards the SCHED_FIFO policy and the Priority Inheritance Protocol.

Multiple visual tools have been developed to display traces, like Tracealyser [9], TuningFork [10], WindView [11], Vampir [12], Zinsight [13], KernelShark [14], or Trace Compass [15]. Most of them will focus on a timeline view to show the traces. It is convenient to follow the flow of a program to understand its behaviour. However, it is not very practical to compare different parts of the trace. Also, to the best of our knowledge, only Trace Compass presents a critical path analysis based on kernel traces, and none of these systems exploit or extend this critical path analysis to add useful information for real-time systems.

3. Pattern Discovery

To find the real-time jobs in the trace, users must provide the corresponding definition in term of a list of tracing events that occur in order. When users do not know what events are involved in the execution of a real-time task, the first step is to suggest them some possible definitions in a graphical interface. This is the pattern discovery step. The users will then select a pattern that will later be used to find all the jobs.

Before explaining the algorithm, we will start with few definitions illustrated in Figure 1. An event has an event type, a content and a timestamp corresponding to the time at which it was generated. A sequence is composed of multiple ordered events. A subsequence is a subgroup of events from the same sequence. An episode is a group of event definitions that need to occur in order. A subepisode is a subgroup of event definitions from the same episode. An occurrence is a sequence corresponding to an episode. The support is the maximum frequency of an episode in a given sequence, thus

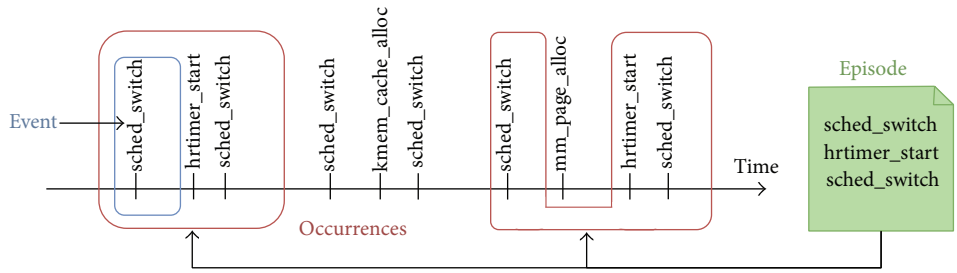

Figure 1: Graph showing events in the trace as a function of time, as well as the two occurrences of a given episode.

2 in the given example. Finally, a pattern is an episode that we will later be used to find the jobs in the trace.

Some algorithms work with the timestamps to find periodic patterns. However, because we want to support sporadic tasks, we wanted an algorithm based on the events order and not on a specific period. Also, to simplify the problem and increase the robustness, we choose to force the pattern to be on a specific thread and not on the whole trace. Real-time tasks divided among several different threads, and even processes, are relatively rare and follow much more complex patterns. On the other side, the algorithms are much more complex when some events are considered to have occurred in parallel, which is often the case with multiple threads.

3.1. MANEPI Algorithm. Based on the previous criteria, we decided to use the *MANEPI* algorithm [16] that discovers patterns using minimal and nonoverlapping occurrences. Indeed, we are interested in nonoverlapping pattern occurrences, since it is the case for real-time tasks within a specific thread. Also, the minimal occurrences, which means that the corresponding episode cannot be found in a subsequence of the occurrence, would result in more precision in the calculation of episode frequency.

In a few words, the *MANEPI* algorithm will find episodes that are more frequent than a given support threshold, which represents the minimum number of repetitions that is needed to validate a pattern. It starts by finding all frequent elements, which means they have more occurrences than the threshold. They will be considered as the basic elements. Then, the algorithm uses the fact that all subepisodes of a valid pattern must also be supported. This first implies that there is no need to consider the elements that are not frequent. It is also possible to start with episodes made of only 1 basic element and increment them to find larger valid episodes. More precisely, for each valid episode, the algorithm checks if the episode is still supported with the addition of each frequent element, until it is no longer supported. The algorithm is depth-first, which means that once the end of a branch is reached, it is possible to release the memory used to get the result and thus lower the maximum memory consumption, as compared to a breath-first search. At each stage, the offset of the occurrences of the current episode is stored to calculate its support. The complex part of the algorithm resides in the calculation of the minimal and nonoverlapping occurrences efficiently.

3.2. Algorithm Modification. As explained, the algorithm takes a sequence and a support threshold to output frequent episodes. In our case, we have complex events with fields and timestamps. To convert them to an ordered sequence of elements, we simply preserve the order of the events and drop the timestamps. These latter can provide useful information but, as previously explained, we want to support sporadic tasks, and we prefer to use the timestamps only in the analysis phase, once all the jobs are found.

Moreover, because we are wanting simple elements to compare, we decide to use only the event types. In fact, we have additional information available since each event can carry a payload in the event fields. While in some cases the event fields can denote a subtype (e.g., sys_read or sys_write instead of syscall), in other cases they can specify an instance (e.g., which timer was set or just expired) or simply some useful statistics (e.g., number of bytes transferred). It would have been interesting to also use the information from the various event fields, but the difficulty is to automatically identify the relevant ones. Moreover, the interesting fields can vary depending on the context. For example, the *id* field of the *hrtimer* event will be relevant only in situations where the usage of the same timer is important. We may eventually attempt to uncover automatically the relevant fields or let users specify which and how event fields should be matched, but this will be more complex and was left for future work.

Because the trace can be very large, we also add a maximum number of events for which the patterns are searched. Only the intervals in the trace of this number of events will be kept in memory. This will result in a faster search and will prevent memory problems. In practice, the interesting sequences are rather short and there is a huge gap between the number of events needed to have a few repetitions and the default maximum number. This approach will thus lead to a valid result, despite this limitation, as long as the maximum number of events is reasonably large.

3.3. Support Threshold. To use the algorithm, we must define a support threshold. We offer two options to the users. First, they can directly define the minimum number of repetitions. That way, users do not have to know how many events are in the trace and how many events are included in the pattern. They only need to have an idea of how many jobs of the tasks are present in the trace segment, in order to specify a lower bound. The higher this bound is, the faster

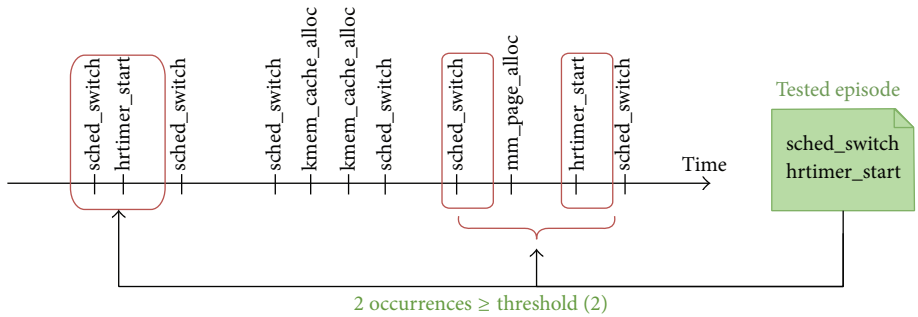

Figure 2: Graph showing events in the trace as a function of time, as well as the two occurrences of a given episode that should be reported as the used support threshold has a value of 2.

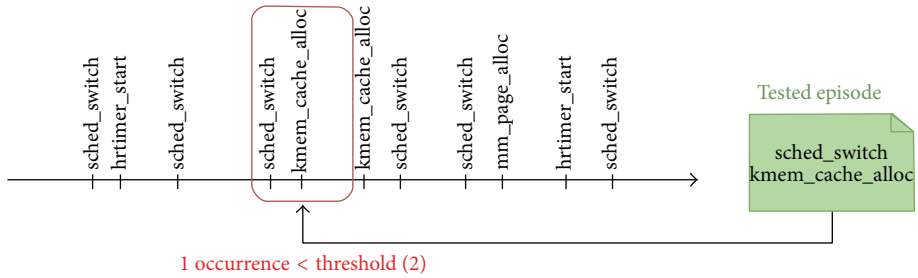

Figure 3: Graph showing events in the trace as a function of time, as well as the only occurrence of a given episode that should not be reported as the used support threshold has a value of 2.

Table 1: Example of a list of occurrences of events with their considered status according to a support threshold of 2.

Count	Element (event type)	Status
6	sched_switch	Frequent
2	kmem_cache_alloc	Frequent
2	hrtimer_start	Frequent
1	mm_page_alloc	Under the threshold: no need to test

the algorithm will be. Indeed, the episodes will be dropped more quickly because their support will sooner be under the support threshold, and there will be fewer frequent elements to iterate at each step. We can see in Table 1 an example with a threshold of 2 that leads to three frequent elements. We can also visualize its impact in Figures 2 and 3 that represent, respectively, episodes with support above and below it.

If the user does not know the number of repetitions, he can also define directly the number of frequent basic elements (i.e., episode of one event type) to be included by the algorithm. We also add a mode to force the pattern to start with a *sched_switch* event. As this event occurs when a thread is scheduled in, it is often the first event in the pattern definition. With this option, the support threshold must obviously be at most the number of *sched_switch*. It would have been interesting to let the users customize the starting events, but it was left for future work.

Despite the threshold option, this algorithm can lead to exponential computation time growth if too few branches are removed. Indeed, if the support threshold is defined sufficiently high, the episodes will soon be discarded, which means that there will be fewer matches. However, if it is not the case, this can lead to some problems with the calculation time. To avoid that, we add a computation time limit that can be set by the users. The results up to that point will still be available.

The presentation of the resulting patterns was one of the challenges. It needed to let the user easily select one of them and load it in the pattern matching interface for the next step. There are some cases where the same event is really frequent in the trace. This results in many discovered episodes with almost only this event type. This was usually not relevant and was harder to present. To avoid this, we allow only one element of each type in an episode. The users can add more afterwards, in the pattern matching phase. Also, to avoid having too many results, we present only the largest patterns. Obviously, if a longer episode is supported, its subepisodes are also supported. The users will just have to select the containing episode and delete the unwanted events with the pattern editing interface in the next step.

4. Pattern Matching

The pattern matching is the phase where all occurrences of a pattern in a given trace will be found. It will normally correspond to the real-time jobs, but as it is not always the case, we will use the term *executions* instead of *jobs*. Then, those executions will be presented in a Comparison View. The goal is to later analyse the ones that have taken more time or present irregularities.

To define the execution model, the first possibility is to load the pattern discovered with the previous phase, if the user is not familiar with the events that define a task. Otherwise, there is an interface to define or edit the pattern. There are two main options offered to the users in this pattern matching dialog. The first option is the same TID mode, which is selected when the executions must start and end on the same thread. It is the most frequent case as real-time tasks are usually a simple task on a single thread. The other option is the different TIDs mode, which means that the executions can start and end on different threads. This case can be useful when a parent thread is creating a child and the execution will end on the child thread, like a real-time timer. Even in the same TID mode, it is possible to support executions on multiple threads, but each execution will stay on the same thread. This will be useful in case there is a thread pool. To define the start and end TIDs set, users can supply them directly or click on the corresponding lines in the main view of Trace Compass showing the different threads.

The events are defined using the event name and the event fields. The model definition also supports some basic operations. First, the keyword $tid will mean the execution TID. For example, the event *sched_switch, next_tid=123* will be matched with the definition *next_tid=$tid* and the execution with TID equal to 123 even if the event occurs on a different TID. Otherwise, the TID on which the event occurs must match the execution TID. The use of the token *&* can also be useful in the case of flags. For exemple, if the flag *1* must be set and *2* must not, we can write *param_name&3=1*. On their side, starting and ending threads are usually defined using TIDs, but the process name is also a supported way to define them.

4.1. Same TID Mode. For the case where the start and end TIDs are the same, there is a graphical interface to add, remove, and change the order of the events in the definition. Each valid TID will have its own state machine instance to detect executions. Those instances are stored within *hashmap* with TIDs as keys to process each event in constant time. While iterating the trace, the state machines are created upon the first encounter of the corresponding valid TID. When an event of the trace is processed, the machine in charge of finding executions for that thread processes the event and compares it to its next definition. If the last state definition is reached and matched, then the execution is registered in the list of valid executions, and the state machine returns to its starting state.

In case there is one or more *$tid* tokens in the pattern, there is a phase to process those definitions, retrieve the TID, and then send the event to the corresponding state machine in addition to the one corresponding to the current thread. Also, in case we encountered a sched_process_free event, which means that it was the last event for this TID, we remove it from the *hashmap* to reduce memory usage. This can be useful if there are thousands of threads.

4.2. Different TIDs Mode. For the case where the start and end TIDs can be different, only start and end events definitions are supported. In addition, there are two different lists for the TIDs, one for the start and one for the end. While iterating over the trace events, we will first try to match the start definition and then the end definition. Depending on if we are matching the start or end, we will discard the event read if its TID is not in the corresponding TIDs list. Once both events are matched, the execution will be added to the list of valid executions and we will restart the pattern.

To know on which thread an event occurs, we need to keep some information. In fact, because the events in LTTng are collected by CPU core, and not by threads, we keep the running TIDs by CPU in a table. Each time a *sched_switch* event is received, this table is updated with the *next_tid* field. Then, the TIDs of the events are retrieved based on the event *processor field* which is always available.

4.3. Options. By default, the complete trace is processed, but it is also possible to process only a segment of it. To do so, the users have two choices. First, they can determine directly the time range, either selecting it graphically or typing it. Only the events within the time range will be processed. Otherwise, they can select the maximum number of executions to detect. The first method is usually preferred when users can identify an interesting portion of the trace and the second method to avoid having memory problems for very large traces. Furthermore, the two can be combined and used simultaneously.

Predefined models are offered to help the users to write the matching definitions. Those include an option to include all events for a TID in a single execution. This can be useful to obtain statistics and execute the analyses on a manually selected trace segment corresponding to the complete thread execution. Other predefined definitions include the *running sequence* and the *blocking sequence*.

Another useful option is to have nested executions, to define events to match at different levels. For example, first level executions can be defined by *sched_switch* and will be displayed. Then, if the second level is defined by calls to *futex*, only the *futex* calls that happen with the same TID, within a higher level execution, will be displayed.

4.4. Complexity. We will define an *event matching* as the comparison of the event type names (represented by a unique id) and then, depending on the case, the comparison of some of the event fields. The complexity of the pattern matching algorithm will be of one *event matching* per *sched_switch* event to compute the running TIDs (current thread on each CPU core). In the same TID mode, there will be an additional *event matching* for each event. Otherwise, it is automatically sent to the matching machine and only one *event matching* will be done. It should be noted that the events are read and decoded only once in this process to increase performance.

5. Views

Once we have all the trace segments corresponding to the execution of the real-time jobs, they are displayed in the *Comparison View*. The goal is to easily identify which executions

present irregularities. The *Time Perspective View* will also be useful to find strange behaviour in a more global perspective. Then, the users will select executions they want to further analyse. The *Critical Path Analysis* will be performed on those executions and the *Critical Path Complement View* will be used to present relevant scheduling information. This is a powerful approach because it will graphically display the synchronization dependencies among the different states involved in the relevant executions.

5.1. Comparison View. This first view, shown in Figure 12, presents the various executions to facilitate the identification of problematic ones. As the events in the trace are collected with timestamps, it would be natural to show the trace in a timeline view. This is in fact a common view in Trace Compass. However, for comparing real-time tasks, we superpose the different jobs executions using the same time scale. That way, it is much easier to compare executions instead of having to look at them along the time axis, where problematic executions could be few and far apart in a trace timeline. Also, the segments of the trace between executions are not shown, to facilitate the visual analysis by avoiding irrelevant information.

By default, the executions are sorted by duration, starting with the longest. This metric is based on the elapsed time and includes the time when the thread is not running, either blocked (waiting for some resources) or preempted (it could run, but other tasks are running). This facilitates the search for problems by starting to analyse the executions that take the most time first. Otherwise, it is also possible to sort the executions by total running time, total preempted time, or starting time. The running time can be useful if it is a low priority task and it is normal to be preempted. On the contrary, the preempted time can be preferred if the running time varies, but the task is of high priority and expected not to be preempted. Finally, the starting time can be used to see the difference between consecutive executions. Those times are calculated with the *sched_switch* events. Indeed, the event *sched_switch* contains a field *prev_state* that indicates if the thread was still running when scheduled out (state *TASK_RUNNABLE*). If it is the case, the thread was preempted. Otherwise, the thread was blocked.

The view is also synchronized with the other views in Trace Compass. That way, it is easy to click on an execution and see what was happening at that time on the system. For example, the *Control Flow View* will show the state of the various threads in the system and the *Resources View* what were the threads running on each CPU.

5.2. Time Perspective View. This view, illustrated in Figure 9, also helps to identify problematic executions, but using a global perspective. It will show the duration of the job executions as a function of their starting time. This can be useful to see if there is a pattern in the distribution of the running time. For example, the longest executions could all occur one after the other. In that case, it is better to check for some special condition happening at that moment. On the other hand, if the perturbations occur with a fixed period, it is probably due to a problematic interaction with another periodic task. In addition, this view allows clicking on the dot associated with an execution to synchronize the other views with that time point and highlighting the corresponding execution in the *Comparison View*. There is also an option to specify a deadline and to show in red the executions that missed their deadline.

5.3. Critical Path Complement View. Once the user finds a suspicious or problematic execution, the goal is to further analyse it. The first step is to use the critical path analysis in Trace Compass which provides useful information about the significant dependencies of a thread. When the analysed thread is blocked, the view shows the resources or threads after which it waits. When a thread on the critical path is preempted, it may be complex to retrieve the priorities of the different threads running during each preemption. You can however check the *Resources View* to find what threads where running and to look for the different events that can result in a priority change. The *Critical Path Complement View*, used for that analysis, is displayed in Figure 14.

Without the critical path analysis, it would still be possible to show the other threads running when the execution of interest is preempted. However, in combination with the critical path, it is also possible to see the running threads when the various threads involved in the critical path are preempted. This means that if the analysed thread was waiting for a resource, and the thread owning this resource is preempted, it will be possible to analyse this scheduling. For example, if the execution thread was waiting for a message and the thread that would eventually send the message is preempted, then the running threads at that moment will be shown with their priorities. If the priority of a running thread is lower then the priority of the analysed thread, it will be displayed in a different colour to show that there is a priority inversion. There is also an option to select the CPUs of interest for the running threads. This can be useful if the system uses different groups of cores, *cpusets*, for specific tasks. Finally, the running threads are sorted to show first the ones that affect the analysed thread the most. This facilitates the search to understand the problem and find a better system configuration.

To know the scheduling priorities of the threads, we keep a list for each TID of priority changes in the form of ordered timestamps with corresponding priority. We build that list at the same time as searching for the execution patterns, to avoid the cost of reading the trace twice. This is mainly done with the *sched_switch* events that store this information in the *next_priority* field and the *sched_pi_setprio* events that report the priority in the case of a priority inheritance. To retrieve a priority for a given timestamp, we do a binary search (of $\log n$ complexity, n being the number of priority changes for the corresponding thread).

Another mode of this view is to show all the threads that interact with the execution thread. This can be useful to understand the system without looking at all the threads that are not related. Internally, it uses the dependencies graphs calculated with the critical path in Trace Compass. There can be more than one graph in the case where the threads are not all linked.

Two options are offered. It is first possible to show the threads that interact directly with the selected one. For example, a thread can be wakeup because another thread releases the futex it was waiting for. The other option is to also show the indirect relations. For example, if thread A is interacting with thread B that is interacting with thread C, then thread C will be shown as indirectly related to A.

To populate this information from the dependencies graph, we first get the graph containing the selected thread. Then, in the case of direct interactions, we just add threads linked from the selected thread within the execution time range. For the indirect interactions, we cannot take all the threads in the graph because this covers more than just the interactions within the time range of interest. Instead, we populate sets with related threads. When a thread A is interacting with thread B in the time range of interest, we check if A or B is in existing sets. If not, we create a new set with the two threads. If only one of them is in a set, we add the other to the same set. If they are in different sets, we merge them.

To avoid iterating through all sets to search if a thread is present, we store the information in a *hashmap*, with the TIDs as keys and references to the sets as values. To merge sets, we add the elements from the smaller set to the larger set and update the references. That way, searching for element takes a constant time, and the total complexity is linear with the number of links. To merge, the overall worst case is when each group is initially composed of 2 elements, and they are merged with a group of the same length, recursively, until there is only a single group. That will result in a worst case of $1/2n \log 2(n) - 1/2n$ updates. Thus, the worst case complexity of the algorithm is $O(n \log n + m)$, where n is the number of TIDs and m the number of links.

5.4. Extentded Time View. It can be useful in some situations to have more information than only the critical path and related threads. The goal here is to present different kernel facilities related to a specific job execution. Those will be presented in a timeline. The time range can match an execution of the *Comparison View* or the range may be specified graphically or by typing, for example, to have a larger view of the situation. The view is shown in Figure 15.

Three options are proposed. First, there is the high resolution timer (hrtimer). It can be in the state *TIMER_INIT*, *TIMER_START*, *TIMER_EXPIRED*, or *TIMER_CANCEL*. It will be in *TIMER_INIT* state after initialization and then in *TIMER_START* state until the timer expired or is cancelled. For a short time, the timer will be in *TIMER_EXPIRED* or *TIMER_CANCEL*, that is, between the respective start and end events. Each timer will be shown in a different row if the job execution refers to more than one.

Then, there is the futex. It can be in the states *FUTEX_WAIT* or *FUTEX_WAKE*. The futex will be in those states when there is futex contention. When one or more threads are waiting on a futex, the state will be *FUTEX_WAIT* until the futex is released. Then, it will briefly be in the *FUTEX_WAKE* state by the time the wake system call is issued. Like for the timers, each futex will be shown in a different row, if the job execution refers to more than one, and the list of futex waiting will be shown in *FUTEX_WAIT* state.

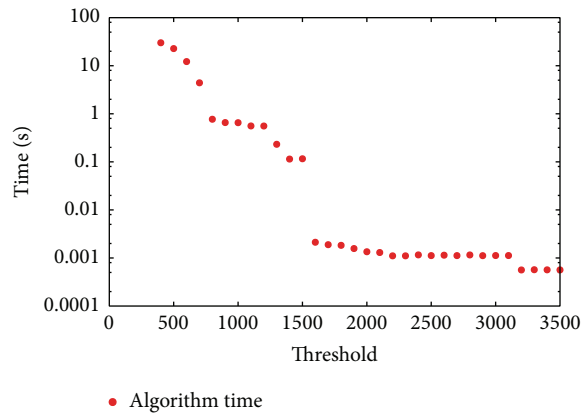

Figure 4: Average time consumed by the pattern discovery algorithm for varying thresholds for a given trace.

Finally, there is the queue. It can be in 4 different states. When a receiver tries to send a message, it will normally result in the state *SENDERS_WAITING*. However, it can also go in the state *QUEUE_FULL_WHILE_SENDERS* if the return code indicates that the queue was full and the thread was not set to wait. On the other hand, when a receiver tries to read from the queue, it will become in the state *QUEUE_EMPTY_WHILE_RECEIVERS* if it is a blocking call and the queue is empty or the state *RECEIVERS_WAITING* if the return code shows that the receivers obtained a message. Each time there is an action, the waiters and receivers are kept in memory to be displayed in the tooltips of the view. That way, it is easy to see what is happening with the queues.

To obtain the information at the start of the time range of the execution, there is an option to select the maximum number of preceeding events to process in the trace, before the start of the desired range. This is a good compromise between the analysis time and the completeness of the information. The different state machines for each resource are kept in hashmap, and each state change occurs in constant time. Thus, the time complexity grows linearly with the number of events in the trace.

6. Performance Analysis

The traces used to test the tool were generated with LTTng tracer version 2.6 with all kernel events enabled on a Linux Preempt-RT Kernel version 3.12. We used a 4-physical-core, 2.67 GHz, machine with 6 Gb of RAM. The first set of data was 5 real-time threads preempting each other, and traces from 483 k to 20.6 M events were collected. The second set of data was traces with up to 16042 different TIDs. Also, traces with various scheduling policies were collected but show no significant difference in performance.

6.1. Pattern Discovery. First, we can see in Figure 4 that the execution times for the pattern discovery algorithm are

erratic. The different traces tested all show a similar behaviour for the execution times. As explained previously, this is caused by the eligibility of new basic elements. This means that when the threshold decreases, enough to allow another element, this will lead to a jump in the execution time. Indeed, there is one additional verification for each valid episode, so more time is required to check all the branches. Moreover, more and longer sequences are kept.

The number of possibilities grows rapidly, as the factorial of the number of basic elements. That explains the general exponential trend, even if most branches are not checked due to the lack of support for the corresponding episode. For example, with the simple case presented in Table 1, a threshold between 6 and 3 leads to only 1 episode to test (*sched_switch*). With a threshold of 2, two other elements are eligible (*kmem_cache_alloc* and *hrtimer_start*) and that leads to many possible episodes. However, not all of them will be tested. Like we can visualize in Figure 3, the episode (*sched_switch, kmem_cache_alloc*) is not valid and thus, the episode (*sched_switch, kmem_cache_alloc, hrtimer_start*) will not be checked.

Figure 4 shows the time consumed by the pattern discovery algorithm for a trace where over 45000 events are considered for the selected thread, from the 4 million events in the trace. Even with that amount, up to a threshold of 800, the algorithm takes less than one second to execute. It is also important to understand that having a threshold too small will result in a huge number of results. For instance, in the case presented, with a support threshold of 1600 occurrences, there are 4 patterns returned, but with a threshold of 500 occurrences, there are over 15000 valid patterns.

6.2. Executions Detection. The detection of executions relies on various factors. The first test was to compare it with the reading of the trace. During the detection, we try to parse the events only if necessary, and we ensure we are only parsing the name and the content once. We compare the executions detection of two models with reading the events name only and with reading the name and the content. The first model is defined by the start and the end of a *nanosleep* system call and the events definitions have no fields matching. The second model represents messages exchange and the definition included events with fields to match the queue identifier.

The first model tested returns a few hundred executions and the second one returns up to 300,000 executions with the bigger trace, which has more than 20 million events. Each of those executions, as presented in Figure 5, shows that both are actually faster than reading the name and the content of each event. However, as can be expected, they take more time than only reading the name. In fact, even without matching the executions, the detection of executions algorithm reads all the names to check for *sched_switch* events and parses the content of those events to maintain the running TIDs. To give an idea, in the largest trace presented there were over 1,3 million *sched_switch* events.

The second test compares the detection to other analyses in Trace Compass. The results are shown in Figure 6. The

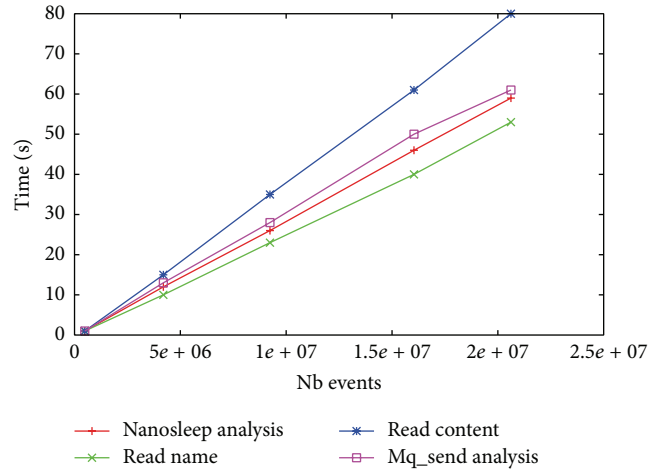

Figure 5: Time taken for our execution detection (nanosleep analysis and mq_send analysis) compared to trace reading in Trace Compass.

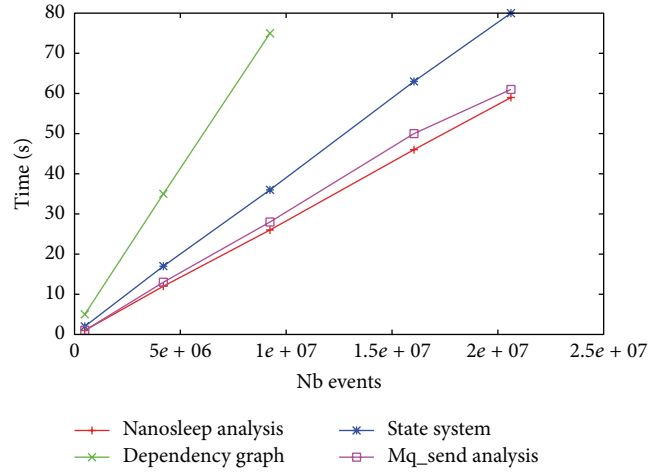

Figure 6: Time taken for our execution detection (nanosleep analysis and mq_send analysis) compared to other analyses in Trace Compass.

first analysis is building the state system used by the *Control Flow View* of Trace Compass. The second one is to build the dependencies graph. This gives us a good insight because our analysis can be complemented by both of them. They appear to be much longer, so our work will not be the bottleneck of a complete analysis. In addition, the dependencies graph construction was running out of memory when the trace was too big.

Furthermore, the fields matching appears not to significantly change the execution time, even if we need to read the content of the event. This is explained by the fact that reading the content takes approximately 30% more time but only approximately one percent of the events are concerned, which would lead to a one-third of a percent increase. This one percent is already a high percentage because, to be concerned by the field matching, an event must be on the relevant TID, in the right state, and must have matching fields. However, the

Table 2: Execution time for the two modes (same TID mode and different TIDs mode) compared to trace reading (in s).

	Read name	Read content	Same TID	Diff TID
Average	2,303	3,326	2,918	2,576
STD	0,025	0,025	0,023	0,026

worst case would be reading all events, approximately leading to a 30% increase.

The previous tests were made with the same TID mode, which is more complex than the different TIDs mode. However, we tested with a trace containing more than 8000 valid TIDs to do the matches, and the same TID mode was significantly slower, around 13%. This is because we need to create an instance by TID and to use a hashmap to match the tid with the instance. The results are shown in Table 2.

For all these tests, we compare the number of executions detected with the number of events shown by the Trace Compass *Statistics View* and each time it was possible to verify the results, we arrived at the same number of executions. This was the case for the two different modes. The execution time is fairly consistent. We can see in Table 2 a low standard deviation based on 10 repetitions for each test. This is similar to the execution time when only reading the trace.

6.3. Views. The views appear to bring another limitation. They can lead to memory problems if there are too many lines, and they can take a long time to refresh. However, there is no point in displaying thousands of executions on separate rows. Thus, the problem will only occur when the default limits are increased. All our views use the same structure, inherited from Trace Compass and follow the same trend. We can see the results in Figure 7. We can see that the time to draw the views increases faster than linearly. This is due to the sorting, being $O(n \log n)$. However, for instance, with the default maximum of 10000 executions, it takes less than 10 seconds which is still acceptable.

7. Test Cases

Two cases are presented to show the usage of the proposed tool and how it can help developers to quickly find problems.

7.1. Real-Time Timer with Higher Priority Task. In the first example, extracted from an industrial use case, a task is initiated from a real-time timer each 250 μs. Upon each timer expiration, a new thread is created to execute a given code. A few times each second, the task takes more time than usual for unknown reasons. The system was traced to find the problem. With the main view in Trace Compass, it is hard to see which executions take a longer time, because only a few missed their deadline over many thousands.

With our tool, we define the job execution as the interval between the end of the code execution of two consecutive threads created by the timer. This way, if the problem occurs either with the timer thread or with the created threads, it will be detected as we can observe in Figure 8.

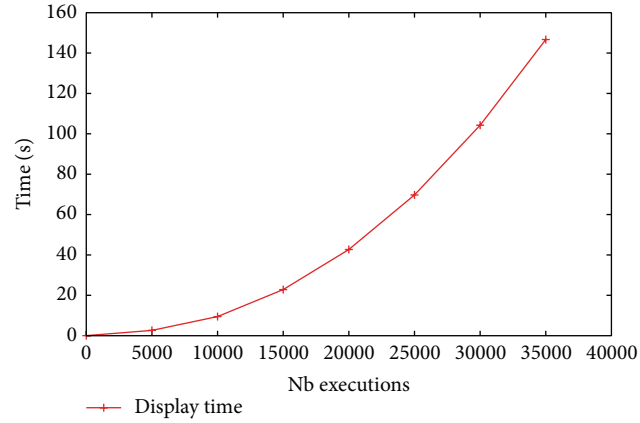

Figure 7: Time consumed in drawing Comparison View in Trace Compass as a function of increasing number of executions.

Running the analysis extracts the executions and highlights the longer executions. We can see in Figure 9 that there were only a few outliers, but they were taking up to almost 4 times the average. When clicking on one of the problematic executions, we can see that the problem is at the level of the timer thread, in Figures 10 and 11, that show, respectively, the gap between the executions and the preemption. Then, we can look at the *Resources View* of Trace Compass to see the running threads.

Alternatively, we can also display the critical path of one of the longest executions and the complementary information. That informs us that the execution was preempted because another thread had a higher priority. It was a configuration problem, because this thread was not supposed to have a higher priority in that situation. The main difficulty lied in the fact that a very large number of threads were involved, designed by different programmers. Once the problem and its origin were pinpointed by the tool, the remedy was simple to devise.

7.2. Waiting for Message. This is a synthetic case, to show the usage of more advanced features. In that case, a high priority task is waiting for a message, but the thread supposed to send the message is preempted by other tasks. There is no priority inheritance with message queues, because we only know afterwards which message goes from which thread to which other thread. In that case, it can be considered as a priority inversion, because the higher priority thread is indirectly waiting for medium priority threads which are preempting the low priority thread.

The first step is to define our model. We use the pattern discovery tool with 12 basic events. This gives us many possible patterns, including the one we where looking for, based on the high resolution timer (*syscall_exit_clock_nanosleep* to *syscall_entry_clock_nanosleep*). We load the pattern and search for executions. This returns a few hundred executions including a few ten problematic ones that we can see in Figure 12, with the running time in green. With the

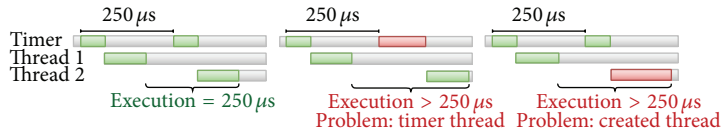

Figure 8: Definition of job execution as the interval between the end of the code execution of two consecutive threads created by the timer.

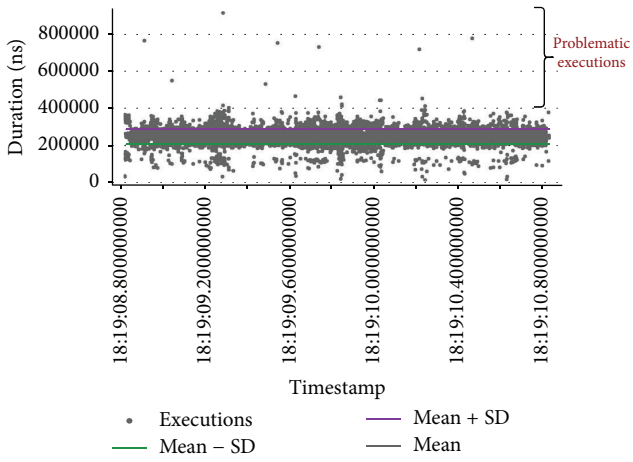

Figure 9: Perspective Time View that helps identify problematic executions using a global perspective.

longest execution, we check the critical path as shown in Figure 13. It shows us that the task thread (TID 3988) was blocked by a thread (TID 3950) that was preempted. With the *Critical Path Complement View* in Figure 14, we can see that the thread blocked was of a lower priority than other threads that preempted it. To prevent this situation, the priority of the thread sending the message should be increased.

Instead of using the critical path, another way of finding that the problem is caused by the thread with TID 3950 would have been to use the *Extended Time View*. It is shown in Figure 15 with the corresponding message queues. This can be really useful if there is a race condition to receive or send a message.

8. Discussion of Results

We show that the search of periodic executions can be useful to efficiently find problems in real-time systems. Running the analysis to find scheduling problems and computing the critical path analysis for the whole trace would have resulted in more complexity to find the interesting results. In fact, when a task with higher priority is ready to run, the short time before it is scheduled can be considered as a priority inversion, if another task is running, and returning all that information would have resulted in a lot of noise. Furthermore, the part of the trace corresponding to the repetitive task of interest could be only a subset of the events that occur on the thread. It is then easier to define the model of the task and show the results only for the outlier executions.

The two test cases presented show that the comparison view can be effective to find scheduling problems, when comparing various executions of a periodic task. None of the tools presented in Section 2 would have been able to pinpoint the problems efficiently as they are not able to detect priority inversions.

The other views developed have also been used to find problems like priority inversion and could probably be extended. For example, analyses on the cache memory or on the communication between machines would be interesting added values.

9. Future Work

The automatic detection of real-time tasks is a field that deserves further work. With many threads, it can be difficult to identify quickly which are the threads of interest. Often, the real-time tasks will have a periodic pattern and will use high resolution timers. It would be interesting to explore the detection of those patterns to allow focusing directly on the corresponding threads. Otherwise, it would also be possible to use the thread priorities to locate real-time threads. From there, there is more work to be done to let users define the job executions, without having an extensive knowledge of kernel tracing.

The present work could be refined to simplify its use for common cases not requiring the advanced functionalities. In addition, some concepts could be decoupled from specific hard-coded events, in order to generalize the procedure to use the same tool for different tracers and for custom structures.

Furthermore, the memory scalability can be problematic for very large traces, because the information concerning valid executions is kept in memory. Instead of only limiting the number of executions or events used, it could be interesting to write the data in a structure similar to the state history tree used by the *Control Flow View*.

10. Conclusion

We demonstrated that a real-time specific kernel trace analysis tool can be used to quickly find complex real-time problems. It was shown that a general model defined by the user, combined with a comparison view, can be very effective to pinpoint the problematic job executions. We also presented a case where the model ability to define a job execution, starting and ending on different threads, was useful. Moreover, we developed an approach to present various possible execution models to the user using pattern discovery. Finally, we presented some interesting avenues to extend the critical path analysis in order to detect scheduling

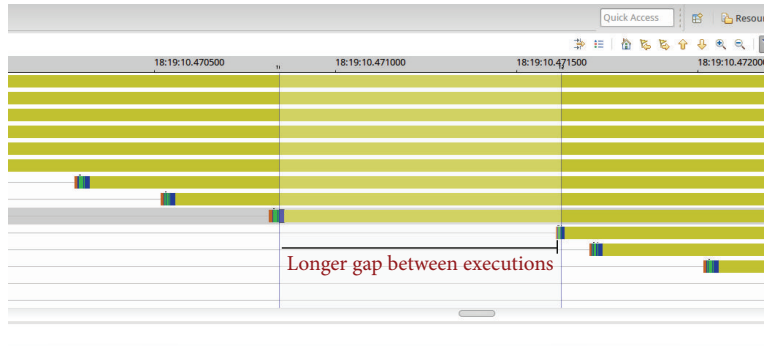

Figure 10: Control Flow View showing the gap between executions.

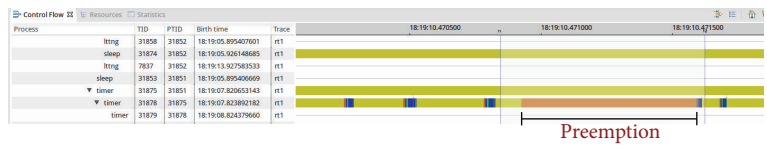

Figure 11: Control Flow View showing the preemption of the timer thread.

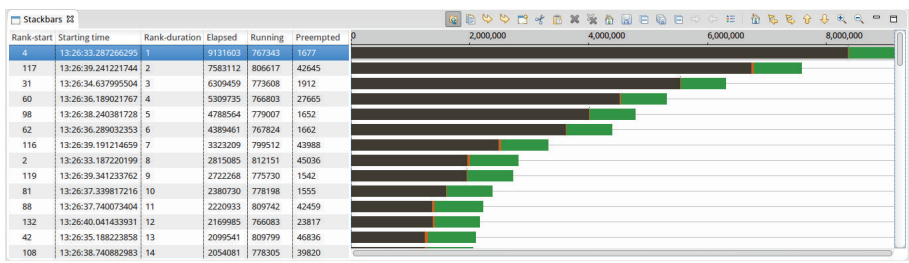

Figure 12: Comparison View in Trace Compass showing the difference in job execution times and statuses allowing the user to identify the most time consuming jobs.

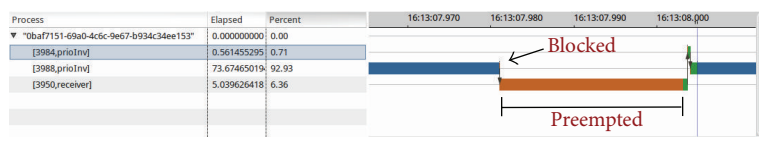

Figure 13: Critical Flow View showing that the task thread (TID 3988) was blocked by a thread (TID 3950) that was preempted.

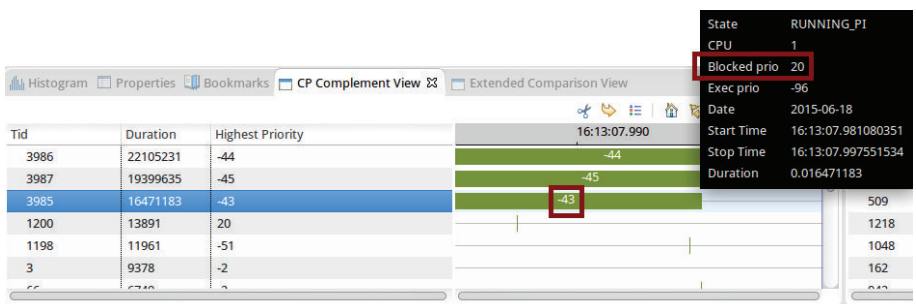

Figure 14: Critical Path Complement View showing that the thread blocked was of a lower priority than other threads that preempted it.

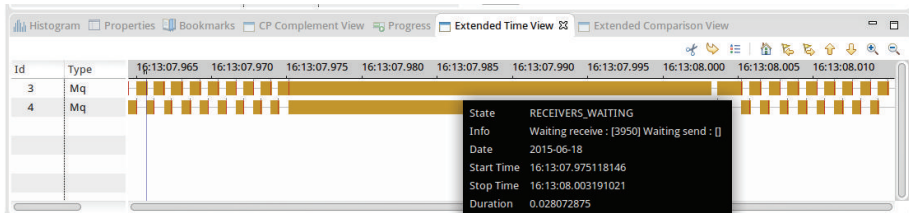

Figure 15: Extended Time View showing message queues and allowing seeing if there are multiple threads waiting to receive or send a message.

problems. All these approaches have also been tested, and the performance measurements were presented, to show in which conditions they are the most efficient.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors are grateful to Francis Giraldeau, Raphaël Beamonte, and Geneviève Bastien for the reviews and useful comments. This research is supported by OPAL-RT, CAE, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ).

References

[1] M. Desnoyers and M. R. Dagenais, "The lttng tracer: a low impact performance and behavior monitor for gnu/linux," in *Proceedings of the Ottawa Linux Symposium (OLS '06)*, pp. 209–224, Citeseer, 2006.

[2] R. Beamonte, F. Giraldeau, and M. Dagenais, "High performance tracing tools for multicore linux hard real-time systems," in *Proceedings of the 14th Real-Time Linux Workshop*, OSADL, Chapel Hill, NC, USA, October 2012.

[3] 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*, vol. 8174 of *Lecture Notes in Computer Science*, pp. 219–234, Springer, Berlin, Germany, 2013.

[4] A. Carminati, R. Silva de Oliveira, and L. F. Friedrich, "Implementation and evaluation of the synchronization protocol immediate priority ceiling in PREEMPT-RT linux," *Journal of Software*, vol. 7, no. 3, pp. 516–525, 2012.

[5] H. Waly, *Automated fault identification: Kernel trace analysis [Ph.D. thesis]*, Université Laval, Quebec City, Canada, 2011.

[6] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana, "Debugging embedded multimedia application traces through periodic pattern mining," in *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT '12)*, pp. 13–22, ACM, Tampere, Finland, October 2012.

[7] A. Terrasa and G. Bernat, "Extracting temporal properties from real-time systems by automatic tracing analysis," in *Real-Time and Embedded Computing Systems and Applications*, vol. 2968 of *Lecture Notes in Computer Science*, pp. 466–485, Springer, Berlin, Germany, 2004.

[8] F. Rajotte and M. R. Dagenais, "Real-time linux analysis using lowimpact tracer," *Advances in Computer Engineering*, vol. 2014, Article ID 173976, 8 pages, 2014.

[9] M. Holenderski, M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: tracing, visualizing and measuring the behavior of real-time systems," in *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS '10)*, pp. 37–42, Brussels, Belgium, July 2010.

[10] D. F. Bacon, P. Cheng, D. Frampton, D. Grove, M. Hauswirth, and V. T. Rajan, "Demonstration: on-line visualization and analysis of real-time systems with TuningFork," in *Compiler Construction*, vol. 3923 of *Lecture Notes in Computer Science*, pp. 96–100, Springer, Berlin, Germany, 2006.

[11] S. A. Hissam, G. A. Moreno, D. Plakosh, I. Savo, and M. Stelmarczyk, "Predicting the behavior of a highly configurable component based real-time system," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, pp. 57–68, IEEE, Prague, Czech Republic, July 2008.

[12] A. Knüpfer, H. Brunst, J. Doleschal et al., "The vampir performance analysis toolset," in *Tools for High Performance Computing*, pp. 139–155, Springer, Berlin, Germany, 2008.

[13] W. De Pauw and S. Heisig, "Zinsight: a visual and analytic environment for exploring large event traces," in *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*, pp. 143–152, ACM, Salt Lake City, Utah, USA, October 2010.

[14] S. Rostedt, "Using kernelshark to analyze the real-time scheduler," 2011, https://lwn.net/Articles/425583/.

[15] Trace Compass, Trace compass, 2015, https://projects.eclipse.org/projects/tools.tracecompass.

[16] H. Zhu, P. Wang, X. He, Y. Li, W. Wang, and B. Shi, "Efficient episode mining with minimal and non-overlapping occurrences," in *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM '10)*, pp. 1211–1216, IEEE, Sydney, Australia, December 2010.