Hindawi Publishing Corporation
Advances in Computer Engineering
Volume 2014, Article ID 173976, 8 pages
http://dx.doi.org/10.1155/2014/173976

Research Article

Real-Time Linux Analysis Using Low-Impact Tracer

François Rajotte and Michel R. Dagenais

École Polytechnique de Montréal, C.P. 6079, Station Downtown, Montréal, QC, Canada H3C 3A7

Correspondence should be addressed to François Rajotte; francois.rajotte@polymtl.ca

Received 21 March 2014; Accepted 18 May 2014; Published 5 June 2014

Academic Editor: Ying-Tung Hsiao

Debugging real-time software presents an inherent challenge because of the nature of real-time itself. Traditional debuggers use breakpoints to stop the execution of a program and allow the inspection of its status. The interactive nature of a debugger is incompatible with the strict timing constraints of a real-time application. In order to observe the execution of a real-time application, it is therefore necessary to use a low-impact instrumentation solution. Tracing allows the collection of low-level events with minimal impact on the traced application. These low-level events can be difficult to use without appropriate tools. We propose an analysis framework to model real-time tasks from tracing data recovered using the LTTng tracer. We show that this information can be used to populate views and help developers discover interesting patterns and potential problems.

1. Introduction

Real-time applications distinguish themselves from their non-real-time counterparts because of their strict timing constraints. The correct operation of a real-time system requires that it responds to stimuli in a bounded time. Real-time systems are often separated in two categories: hard and soft real-time. Hard real-time requires the response time to be bounded and never exceeded. In soft real-time systems, an exceeded response time is undesirable but does not incur the complete failure of the system.

The real-time capabilities of the Linux kernel have been improved thanks to the work done by the PREEMPT-RT patch contributors. Many tools have been developed to help demonstrate the real-time capabilities and limits of Linux systems. Previous work has also demonstrated the good real-time behavior of the LTTng tracer [1]. LTTng provides both kernel and user space instrumentation. Because of the demonstrated low impact of LTTng on real-time applications, we have chosen to use it to gather the traces required for the analysis.

In Linux, the thread is the basic unit of execution managed by the scheduler. A single real-time task can therefore easily be mapped to a thread. A task can have properties that are unknown to the kernel such as periodicity and maximum tolerated response time. Our goal is to extract these higher level concepts of real-time tasks from information collected at the kernel level.

The low overhead needed for the instrumentation means that the events are recorded with as little preprocessing as possible. In order to extract more advanced information from the events, it is possible to apply a postprocessing step on a recovered trace. Using the semantics of the events, it is possible to extract metrics such as CPU or memory usage over time [2]. Our contribution consists of an analysis framework to extract additional debugging information and metrics from a trace recorded using the LTTng tracer on a Linux system running real-time applications, without the need for manual instrumentation.

2. Related Work

This section presents closely related work on the subject of trace analysis. The techniques discussed here are divided in two categories: algorithm-based techniques and visualization-based techniques.

2.1. Existing Algorithmic Techniques. Some techniques use knowledge of the execution of a task to compute metrics and statistics. Santos and Wellings calculate blocking time experienced by a task to help identify errors in the worst case execution time assumptions [3]. This algorithm uses knowledge of the tasks' base and active priorities to identify periods of priority inversion. Modifications in the operating system were required to acquire all the information necessary for the algorithm. Terrasa and Bernat use finite state machines to extract metrics at the task and global level [4]. Simple automatons can generate metaevents to feed into larger automatons enabling the computation of more complex metrics. This approach defines four minimal events required to build the automatons. These events describe when a task becomes ready and when it is finished as well as its base priority. The final event describes context switches. Of these four events, only the context switch event is directly retrievable from a Linux kernel trace.

Data mining techniques are also useful to find periodic patterns and anomalies in a trace. Mannila et al. introduce the concept of episode to describe temporal relationships between events [5]. An algorithm is provided to find frequent episodes. The frequency of episodes is determined by splitting the trace into windows of a fixed size and measuring the number of windows that contain the episode. These frequent episodes can then be used to infer rules about the presence or absence of events in a trace. Other techniques focus on finding periodic patterns. Some of these techniques require the definition of windows or specific events to split the trace into individual worksets [6] while others can find patterns without this need [7]. Common to all these methods is the incremental approach to building the patterns. It is therefore necessary to read the trace multiple times or preprocess the trace in a different format.

2.2. Existing Visualization Techniques. Visualization techniques are also useful to organize the content of a large trace.

Zinsight provides three views to present trace data in different formats [8]. The first one places the events in rectangles in a two-dimensional plane with time on the vertical axis and a variable grouping scheme on the horizontal axis. The second view groups events by type and provides timing information on paired events such as function entry and exit. The third view calculates sequences of events leading up or following an event type of interest, presenting this information in a directed graph. This view allows a developer to see common event sequences and abnormal ones. These views have the advantage of letting user display information in many different ways but still require advanced knowledge of the trace events to find sequences of interest.

TuningFork is a framework developed specifically to help debug complex real-time systems [9]. It provides filters and aggregation functions to generate data for views. Among its generic views is the Oscilloscope view. This view allows the visualization of high frequency data by separating the trace in strips using a predefined time interval and stacking them. This view allows the observation of periodic behaviour but it requires the knowledge of the period of the task and is of limited use on tasks exhibiting a varying period, such as sporadic tasks.

2.3. TMF. The tracing and monitoring framework (TMF) is the default viewer for traces recorded using LTTng. It provides views to explore traces and display various statistics. Among these views, the Control Flow View is used to display detailed information about the states of the different threads running on the system. These states are derived from the trace events using an extensive finite state machine. An efficient storage mechanism is then used to store and retrieve the states for display without the need to recompute them from the trace.

3. Proposed Model

Whereas TMF recreates the threads' states as they are inside the Linux kernel, our approach's goal is to create a higher level understanding of the thread at the task level. We define a real-time task as a series of recurring jobs running on a single thread. If the recurring jobs arrive at constant intervals, the task is said to be periodic. An example of this kind of task is the running of a real-time simulation, such as an aircraft simulator. Periodic tasks are in charge of calculating new simulation parameters in time for the next frame. When the recurring jobs do not arrive at regular intervals, the task is said to be sporadic. Using the same example, a sporadic task could be in charge of modifying simulation parameters when the operator inputs a command or activates a switch.

In schedulability analysis [10], a periodic task is defined using a period, a relative deadline, and a worst case execution time. A sporadic task is similar to a periodic task but has a variable period. It is rather defined using a minimum arrival time, specifying the minimum time between two jobs.

A system will generally have many of these tasks running at the same time, on one or many processor cores.

The operating system is in charge of scheduling the different tasks. In Linux, the schedulable entity is the thread. We must therefore map the thread's state inside the kernel to the higher level task state we want to model.

The Linux kernel uses two major states to keep track of the status of a thread: it is either running or blocked. When a thread is in the running state, it means that the scheduler is free to schedule the task on a CPU. Even in the running state, a thread can still wait in the run queue if all the CPUs are occupied. When it is not running, a thread will be in one of the three principal blocking states. When blocked, a thread will be sleeping until a certain condition is met. Each of these blocking states describes what can wake up the thread. In interruptible sleep, a thread is woken up when the required condition is met or when an interrupt occurs or a signal is received. In uninterruptible sleep, only the required condition can wake up the thread. Killable sleep is a specialization of uninterruptible sleep that also allows the thread to wake up when a fatal signal is received.

3.1. Modeled Task States. The thread states contained within the Linux kernel represent the concerns of the operating system and do not translate directly to the realities of the application or real-time task. A recurring real-time task will follow a pattern of two major phases. It is either executing

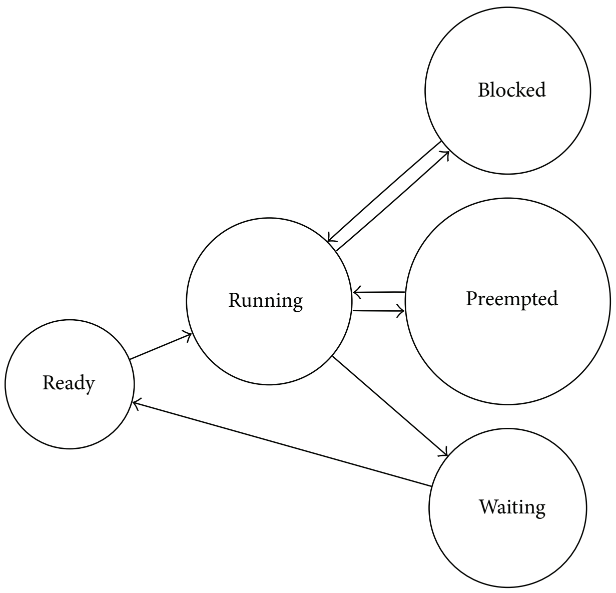

Figure 1: The states of the model and the possible transitions between them. Waiting represents the duration between the end of the previous job and the start of the following job. Ready represents the duration between a task receiving the signal to wake up (its arrival) and the actual start of the job. Blocked is reached when a task is blocked from entering a critical section. Preempted happens when a task is preemptively stopped from running because of a higher priority task entering the running state on the same processor. Running is the state in which the task executes the job.

to complete before a deadline, or waiting until its next execution. Because other real-time tasks are executing at the same time, the execution phase will generally be broken up by periods of waiting. This waiting can happen because a higher priority task must execute first, or because a necessary resource is currently held by another task.

A real-time task will therefore follow a series of waiting and execution states. Our approach models the different states that a task can be in during its execution. Most importantly, it distinguishes between the different reasons for waiting. For that purpose, we have modeled four different states describing different types of waiting. The fifth state is used to describe the running state. These states and the possible transitions between them are shown in Figure 1.

3.2. Trace Events. In order to extract these states from the trace, we have identified the kernel events that allow us to define the necessary state transitions.

When tracing a real-time system, it is important to disturb the system as little as possible. As such, we have chosen those events because they are the minimal set that allows describing the transitions of our model.

The states are built using a finite state machine using the same states as those defined in the model discussed earlier. The transitions are based on the chosen events and conditions on their fields. The two events needed are both generated from the scheduler of the Linux kernel.

The sched_wakeup event is used to know when a task becomes ready. The sched_switch event is generated when the scheduler changes the thread executing on a processor. We use this event to know when a task starts running, when it is preempted or blocked, and when it starts waiting for the next job.

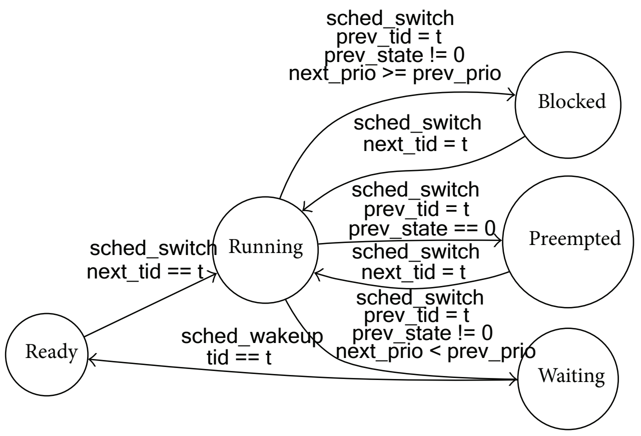

Figure 2: The state transitions are defined using specific events and their fields.

3.3. State Transitions. The running-to-preempted transition is easily covered using the prev_state field of the sched_switch event. When a thread is scheduled out while still being runnable, the prev_state field will indicate TASK_RUNNABLE. This can be directly mapped to the preempted state of our model.

The running-to-blocked and running-to-waiting transitions are trickier because in both cases the thread will be in the TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, or TASK_KILLABLE state. Using the prev_state field is not sufficient in this case because the kernel uses the same values to describe different realities at the task level. The ambiguity arises if the task uses mutexes with priority inheritance to delimit its critical sections. According to the priority inheritance protocol, in case of contention, the priority of the offending thread will be boosted to the priority of the highest priority blocked thread. We can therefore use the prev_prio and next_prio fields to distinguish the blocked and waiting states. If the next thread to run has a lower priority, we can be certain that the previous thread has transitioned to the waiting state. If the priority of the next process is equal or higher, the previous thread has transitioned to the blocked state. Figure 2 presents the graph of transitions with the required events and their fields.

This distinction can be made if all threads follow the first-in first-out scheduling (SCHED_FIFO) and have different base priorities. In the case of a higher priority thread becoming ready, it will immediately preempt the current running thread. The current task will therefore always enter the preempted state at that point. In the case of a lower priority thread becoming ready, it will not have the chance to run until all other higher priority tasks voluntarily enter the waiting state. The only ambiguous transition is when a thread of the same priority replaces the current one. Since we have as

Table 1: State transitions that start and end the accumulation of a given statistics.

Statistics	Transition begin	Transition end
Latency	Waiting-to-ready	Running-to-waiting
Running time	*Any*-to-running	Running-to-*any*
Blocking time	*Any*-to-blocked	Blocked-to-*any*
Inter-arrival time	Waiting-to-ready	Waiting-to-ready
Wakeup time	Waiting-to-ready	Ready-to-running

a restriction that all threads have different base priorities, the only way another thread would have the same priority is if its priority was boosted. The boosted priority requires that the current thread is blocked from acquiring a mutex still held by the offending thread.

3.4. Statistics Extraction. As the trace is analyzed using our model, we divide the tasks according to the individual jobs they contain. The task entering the ready state is used as the marker for a new job. Statistics can then be calculated for each individual job. Some statistics are defined using the transitions of the state machine. The transitions define when a statistics should start accumulating and when it should stop. Some transitions can occur multiple times during a job and therefore trigger the same statistics accumulation. In those cases, only the total value for a job is kept. Table 1 shows an example of common statistics and the transitions used to compute them.

4. Performance Analysis

The performance of this statistics extraction technique was tested on traces of varying sizes. The traces were generated using the LTTng kernel tracer with the sched_switch and sched_wakeup events enabled. Enabling other events populates the trace with events that are ignored by our model. The worst performance is expected when most events translate to state changes. By enabling only these two events, we ensure that the density of events of interest is high and that many events will generate state transitions in the model.

The real-time workload was generated using the cyclictest tool, part of the *rt-tests* test suite. Cyclictest is used to measure the worst case latency expected on the system. It does this by running simultaneous real-time tasks and measuring difference in time between the expected arrival time and the actual job start time.

For this test, we used cyclictest running ten threads at varying periods and priorities while tracing the kernel. Larger traces were achieved by running cyclictest for a longer period of time. The analysis was run on an Intel Core7 processor running at 2.6 GHz with 10 GB of RAM. The analysis time is presented in Figure 3. The time spent only reading the trace is also presented to better show the actual time spent generating the model.

We observe a linear progression of the time spent generating the model compared to the size of the trace. This is expected since evaluating a state change of the model requires

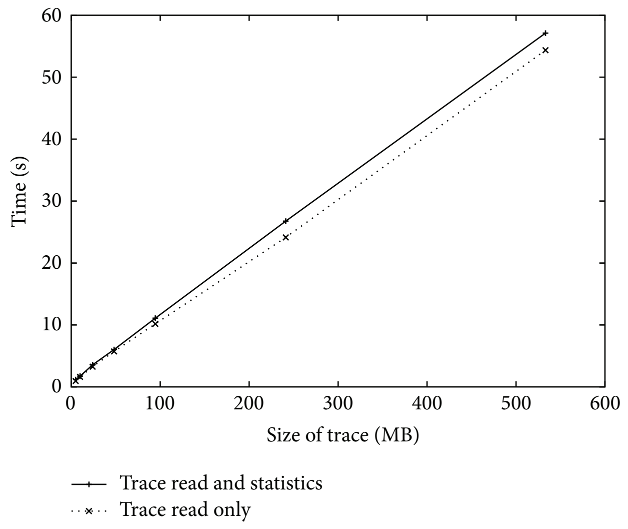

Figure 3: Time spent calculating our model for traces of varying sizes.

verifying conditions on a finite number of fields for each event. Most of the time is actually spent reading and parsing the trace file. This cost is unavoidable but is not a major issue when we take into account the fact that other analyses can be run at the same time on the same trace. The cost of reading the trace is therefore amortized over all the analyses running on the same data.

5. View

The model we presented earlier allows for fast modelling of task data from thread information contained in kernel traces. Common statistics can be extracted from the model and provide an overview of the performance of a real-time task. The use of tracing also allows for more in-depth analysis. By tracing the kernel, it is even possible to record events outside the real-time application, without the need for additional instrumentation. At this level of detail, views become important tools to quickly navigate the large quantity of information.

Since traces contain chronologically ordered events, it is typical to display the desired information on a single timeline from trace start to trace end with the possibility to zoom in and out at will. This kind of display makes it easy to follow the execution of a single thread but it becomes harder to compare two sections of the trace far apart in time at a sufficient level of detail.

We used these modeled states to separate a task into each individual job. We developed a view to show the jobs together on the same time base, synchronized on the task release time. It is therefore easier to compare them without having to scroll through the trace to compare two jobs. It is also possible to sort the jobs, according to different statistics that can be calculated from the time spent in each state or between transitions.

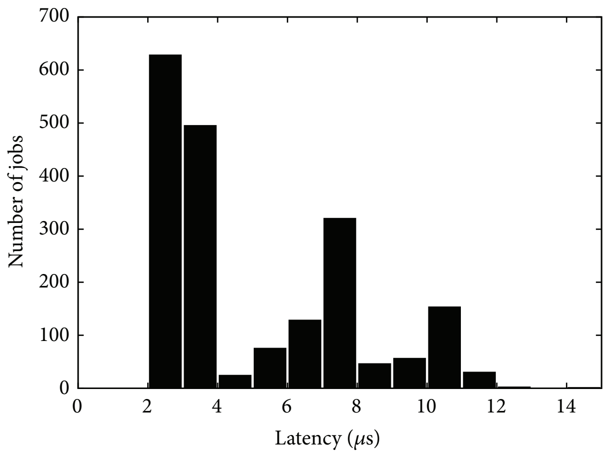

Figure 4: Histogram of the latency of a cyclictest thread running at medium priority.

6. Test Cases

In this section, we will show how this view can be used in tandem with existing tools to better understand complex interactions of real-time applications. First, a simple case will be presented to introduce basic concepts. Then, two more examples will demonstrate the additional insight provided using our approach and the help it provides to debug problems.

6.1. Basic Concepts. To introduce the view, we will use cyclictest to generate a simple trace. Cyclictest was configured to run ten threads, simulating ten real-time periodic tasks on the same CPU. Each task's period is 100 us longer than the previous one, the smallest period being 100 us. The task with the smallest period was also the one with the highest priority. The other tasks have one lower priority than the previous one, following their respective period in increasing order.

Once this run of cyclictest is traced, the analysis is performed. The gathered statistics provide a good overview of the whole run. Figure 4 presents the histogram of the latency observed for the fifth highest priority task. The maximum latency observed is 14 us but most latencies are between 2 and 3 us. Other peaks are observed at 7 and 10 us.

Statistics alone cannot explain the observed behavior, but they provide clues for potential problems. In this case, we would like to understand the cause of the peaks in latency observed. Although still acceptable, the peaks could be signs of a deeper problem. The Control Flow View of TMF presents the trace in a chronological fashion that allows zooming in on a time range and observes the interactions between threads. Such a view is presented in Figure 5. In this view, threads of higher priority are at the top. Although most jobs are executed without delay, sometimes, a higher-priority job will preempt the execution of a lower-priority one. This preemption increases the latency of lower priority jobs. We also observe that longer delays can happen when multiple higher-priority jobs need to be executed in a short interval.

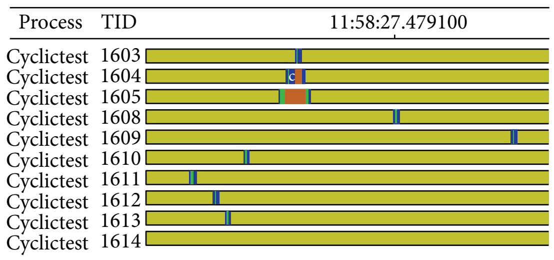

Figure 5: A zoomed-in Control Flow View of TMF showing the preemption of threads.

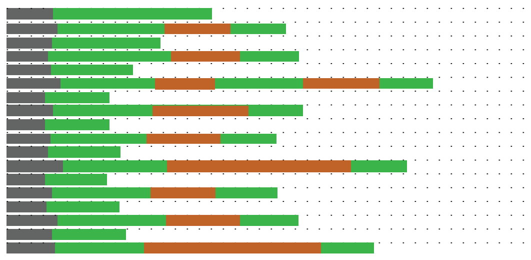

Figure 6: Our view showing individual jobs stacked and synchronized on their arrival time.

Preemption is a normal and desired feature for real-time schedulers. It allows high priority tasks to finish sooner and therefore have low latency. If we were to investigate the latency spike observed using only TMF's Control Flow View, we might dismiss the spike as normal and not a cause for concern. Using our model, we can extract jobs of a real-time task and show them in a way that would not be possible in a strictly chronologically accurate view such as the Control Flow View of TMF.

The resulting view is shown in Figure 6. Using this view, we can observe that the longer latencies are not randomly distributed but follow a pattern. Every two jobs, a higher priority preempts the current job. Every six jobs, another job also preempts it, producing an even larger delay. This is caused by the arrival times of different tasks that happen too close to one another.

6.2. Abnormal Delay. The previous section dealt with extracting knowledge from a seemingly normal execution. This section will deal with finding the source of the problem after an anomaly occurs such as a missed deadline. During our work with cyclictest, we have experienced unexpected latencies in the order of several milliseconds. This was surprising considering the fact that the program was running on an isolated core. We were able to trace the system while the latency spike happened and use our model to pinpoint the problem.

Since we want to find a missed deadline, we use our view while sorting by longest latency first (Figure 7). The worst offending job will be at the top of the view. We can further explore the source of the problem by examining the surrounding area of the trace. Since our view is synchronized with the Control Flow View, simply clicking on a job will take

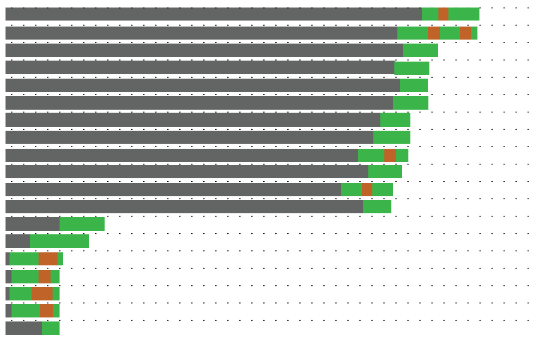

Figure 7: Our custom view showing individual jobs, sorted by longest latency.

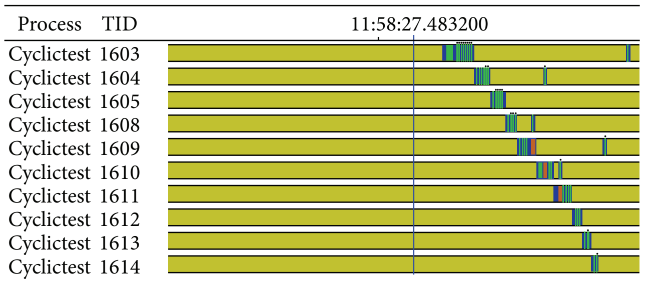

Figure 8: The delayed threads as seen in the Control Flow View of TMF.

the Control Flow View to the same location in the trace. This view is shown in Figure 8.

In that view, we can confirm the problem is plaguing all the other threads of cyclictest as well. At that location, we also find another program executing on another core. That program was executing an ioctl system call. Using the syscall_entry event from the trace, we can recover that call's arguments and discover that this particular call is tied to the graphics driver. Upon further investigation, we found out that the graphics driver executed a privileged instruction invalidating the cache of the processor. This caused subsequently the processor to stall for a few milliseconds while the cache was being repopulated, even on cores theoretically shielded from the others.

6.3. Sporadic Tasks. The previous cases dealt mainly with simple periodic tasks. The next case we present will deal with sporadic tasks and exhibit blocking and priority inheritance. The use of specific kernel events does not limit the separation of the trace according to a fixed period. It is also possible to split a trace according to a task of variable period. To demonstrate that possibility, we have implemented a simple producer-consumer application.

We have attributed higher priority to the consumer task than the producer task to keep the overall latency of the application as low as possible. The data is transferred from the producer to the consumer using a shared buffer in memory. To prevent concurrent access, this buffer is synchronized using semaphores. A third task is also running at the same time, with the highest priority. This task's purpose is to disturb the two other tasks of interest and create jitter.

Because of the way this application is designed, we can expect the behaviour to follow a pattern. At the beginning of the execution, the consumer has nothing to consume and is therefore blocked. The producer produces its first unit of data, stores it in the buffer, and indicates via a semaphore that data is ready to be consumed. The consumer wakes up and starts consuming right away because of its higher priority. It soon consumes all the data in the buffer and blocks. This, in turn, allows the producer to continue producing and the cycle begins anew until all data is processed.

We can try to verify this behaviour using the Control Flow View of TMF. In Figure 9, we can see the tasks executing one after the other as expected. However, it is not clear when each of the expected phases is active. There appears to be additional scheduling activity that was not predicted by the previous analysis. In Figure 10, we use our algorithm to split the consumer task in its individual phases.

We can observe that the additional scheduling activity is caused by a period of blocking at the end of each job. Some jobs also show a second period of blocking. Cross-referencing these jobs with the Control Flow View, we see that these extraneous periods of blocking are caused by the higher priority task interfering with the execution of the consumer, of lower priority.

However, the common period of blocking to all jobs is not caused by an external process. It is rather caused by the use of a fully preemptible Linux kernel modified with the PREEMPT_RT patch. The goal of this patch is to reduce latency inside the Linux kernel by reducing the amount of time spent in nonpreemptible code.

In our case, when the producer task wakes up the consumer to indicate that data is ready, the producer enters kernel space. However, as soon as the consumer becomes ready, the producer is preempted and prevented from leaving protected areas of the kernel. This allows the consumer to complete its work earlier and reduce latency. When all the data is consumed, the consumer tries to enter the waiting state once again but is prevented from entering protected areas of the kernel because the producer has not left them yet. The consumer is blocked while the producer's priority is boosted and can leave the protected areas. Once this is complete, the consumer wakes up yet again, this time to wait on the availability of data on the shared buffer.

The extra context switches are an example of the drawbacks of using a kernel modified to reduce latency. By reducing the overall latency of the system, the throughput of the application is affected negatively. If the reduced throughput is an important concern for the application, it might be desirable to configure the kernel to reduce its preemptibility. This allows the user to fine-tune the system between throughput and latency.

Using this real-time application as example, we are able to observe some inner workings of the Linux kernel that are not obvious when programming at the user space level. Our task-splitting algorithm can improve the comprehension of a

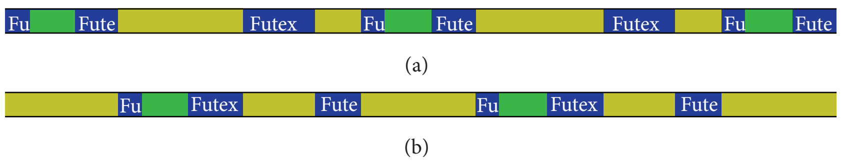

FIGURE 9: The producer (a) and consumer (b) threads as seen in TMF's Control Flow View.

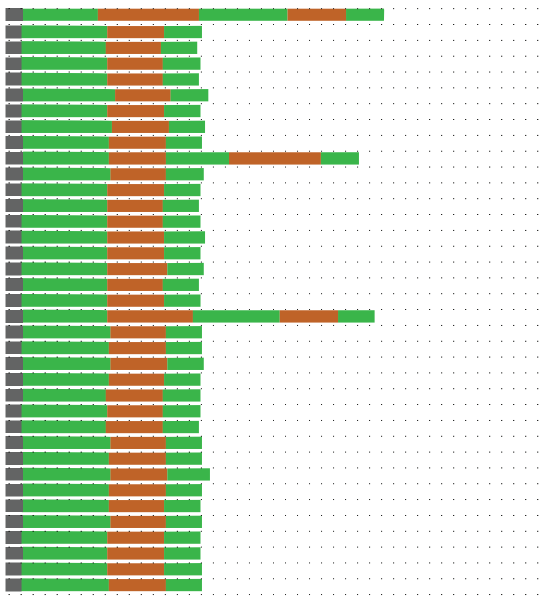

FIGURE 10: The individual jobs of the consumer thread, ordered chronologically.

trace by extracting important states and displaying them in a way that helps the user discover interesting patterns that are not obvious in a strictly linear chronological view.

7. Conclusion

This paper addresses the analysis of real-time tasks from information found in a kernel trace. Debugging real-time tasks is inherently difficult because it is not possible to use traditional debuggers to analyse complex timing interactions. Using the low-impact LTTng kernel tracer, we can gather traces of real-time tasks running on Linux while disturbing the system as little as possible. Using a kernel tracer also has the benefit of not requiring modifying the application's source code to add trace points manually. Once the trace is retrieved, we can recreate the task state from the kernel events contained in the trace.

We begin by defining our model using common states and concepts of real-time applications. These states take into account the common realities of priority-based scheduling such as preemption and priority inheritance. Next, we identify the kernel events that can be used to generate the required transitions of the model. We use knowledge of the scheduler to distinguish task states that are ambiguous at the kernel level. We then use the model to extract statistics useful in gaining insight on the application. We analyse the performance of our method and find that the time spent generating the model is very small compared to the time required to read the trace. We then use the generated model to split a task into its constituent jobs, whether they are periodic or sporadic. A view is presented in which the jobs are shown and easily compared to one another. The statistics gathered previously can be used to reorder the jobs and find areas of interest. Two real-time applications are analyzed using this approach. In the first case, we find unforeseen interactions between tasks within the application and outside the application. In the second case, we observe interactions with the kernel that are invisible at the user space level.

One area of future work is to express the transitions of the model using a generic language that would allow a greater flexibility. Users could define their own model, include the necessary instrumentation, and even define their own views. Another area of interest is the calculation of critical paths during periods of blocking. Critical paths are used to explain the duration of blocked states by following the chain of events that caused the blocked states to end. When applied to real-time tasks, critical paths can be used to find complex interactions between threads.

Conflict of Interests

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

References

[1] R. Beamonte, *Traçage de systèmes linux multi-coeurs en temps réel [Ph.D. thesis]*, École Polytechnique de Montréal, Montréal, Canda, 2013.

[2] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, "Recovering system metrics from kernel trace," in *Proceedings of the Ottawa Linux Symposium (OLS '11)*, pp. 109–116, 2011.

[3] O. M. D. Santos and A. Wellings, "Measuring and policing blocking times in real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 1, article 2, 2010.

[4] 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*, pp. 466–485, Springer, New York, NY, USA, 2004.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[6] P. L. 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, New York, NY, USA, 2012.

[7] S. Ma and J. L. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *Proceedingsof the 17th International Conference on Data Engineering*, pp. 205–214, IEEE, Heidelberg, Germany, 2001.

[8] 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, New York, NY, USA, October 2010.

[9] D. F. Bacon, P. Cheng, D. Frampton, and D. Grove, "Tuningfork: visualization, analysis and debugging of complex real-time systems," IBM Research RC24162, 2007.

[10] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling sporadic and aperiodic events in a hard real-time system," Tech. Rep., DTIC Document, 1989.