Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 9792462, 18 pages
http://dx.doi.org/10.1155/2016/9792462

Research Article

Detection of Common Problems in Real-Time and Multicore Systems Using Model-Based Constraints

Raphaël Beamonte and Michel R. Dagenais

Computer and Software Engineering Department, Polytechnique Montreal, CP 6079, Station Downtown, Montreal, QC, Canada H3C 3A7

Correspondence should be addressed to Raphaël Beamonte; raphael.beamonte@polymtl.ca

Received 25 November 2015; Revised 3 March 2016; Accepted 16 March 2016

Academic Editor: Dimitrios S. Nikolopoulos

Multicore systems are complex in that multiple processes are running concurrently and can interfere with each other. Real-time systems add on top of that time constraints, making results invalid as soon as a deadline has been missed. Tracing is often the most reliable and accurate tool available to study and understand those systems. However, tracing requires that users understand the kernel events and their meaning. It is therefore not very accessible. Using modeling to generate source code or represent applications' workflow is handy for developers and has emerged as part of the model-driven development methodology. In this paper, we propose a new approach to system analysis using model-based constraints, on top of userspace and kernel traces. We introduce the constraints representation and how traces can be used to follow the application's workflow and check the constraints we set on the model. We then present a number of common problems that we encountered in real-time and multicore systems and describe how our model-based constraints could have helped to save time by automatically identifying the unwanted behavior.

1. Introduction

System analysis tools are necessary to allow developers to quickly diagnose problems. Tracers provide a lot of information on what happened in the system, at a specific moment or interest, and also what leads to these events with associated timestamps. They thus allow studying the runtime behavior of a program execution. Each tracer has its own characteristics, including weight and precision level. Some tracers only allow tracing kernel events, while others also provide userspace tracing, allowing correlating the application's behavior to the system's background tasks. However, each of these tracers shares the fact that an important human intervention is required to analyze the information read in the trace. It is also necessary to understand exactly what the events read means, to be able to benefit from this information.

Modeling allows technical and nontechnical users to define the workflow of an application and the logical and quantitative constraints to satisfy. Modeling is also often used in the real-time community to do formal verification [1]. Models and traces could thus be used together to define specifications to satisfy and to check these against the real behavior of our application. This real behavior, reported in traces, would moreover take into account the influences of other running applications, the system resources, and the kernel tasks. Using kernel traces information, we could therefore extend the set of internal constraints, to add system-wide constraints to satisfy, such as a minimum or maximum CPU usage or a limit on the number of system calls our application can do.

This paper describes a new approach for application modeling using model-based constraints and kernel and userspace traces to automatically detect unwanted behavior in applications. It also explains how this approach could be used on top of some common real-time and multicore applications to automatically identify the problems that we encountered and when they occur, thus saving analysis time.

Our main contribution is to set constraints over system-side metrics such as resource usage, process preemption, and system calls.

We present the related work in Section 2. We explain our approach on using model-based constraints and present some specific constraints in Section 3. We then detail some common real-time and multicore application problems, as part of case studies to evaluate our proposed approach, in Section 4. Results, computation time, and scalability for our approach are shown in Section 5. Future work and the conclusion are in Section 6.

2. Related Work

This section presents the related work in the two main areas relevant for this paper, software tracing with a userspace component and analysis of traces using model-based constraints.

2.1. Existing Software Userspace Tracers. To extend the specifications checking of an application, trace data must be available at both the application and system levels. We also need to put emphasis on the precision and low disturbance of the tracer we would use to acquire these traces. In this section, we present characteristics of currently available software tracers with a userspace component and kernel tracing habilities.

Basic implementations of tracers exist that rely on blocking system calls and string formatting, such as using `printf` or `fprintf` or even that lock shared resources for concurrent writers to achieve thread-safety. Those tracers are slow and unscalable and are thus unsuitable for our research on multicore and real-time systems. They have therefore been excluded.

`Feather-Trace` [4] uses very lightweight static events. It was mainly designed to trace real-time systems and applications and is thus a low-overhead tracer. `Feather-Trace`'s inactive tracepoints only cause the execution of one statement while active ones execute two. It uses multiprocessor-safe and wait-free FIFO buffers and achieves buffer concurrency safety using atomic operations. This tracer achieves low-overhead by using its own event definitions of a fixed size. The memory mechanism for these events is based on indexed tables. However, this design choice limits overhead but makes `Feather-Trace` unable to add system context information to the events, for instance. In its current form, the tracer also cannot use the standard `TRACE_EVENT()` macro to access system events and, even with improvements, would not be able to take advantage of the different event sizes and the information it provides. Also, `Feather-Trace` does not include a writing mechanism for storing the traces on permanent storage. Finally, the timestamp source used is the `gettimeofday()` system call, limited to microsecond precision.

`Paradyn` uses dynamic instrumentation by inserting calls to tracepoints directly in the binary executables [5]. Although the instrumentation can be done at runtime [6], `Paradyn` uses a patch-based instrumentation to rewrite the binary, only imposing a low-overhead latency [7]. This method has been used to monitor and analyze the execution of malicious code. This tracer however offers limited functionality. It cannot switch to another buffer when the buffer is full nor can it store the tracing data to disk while tracing. Furthermore, it cannot support the definition of different event types. It thus is not possible to use the Linux kernel static tracepoints defined by the standard `TRACE_EVENT()` macro nor to add system context information. Also, no assurance can be given on the tracing condition for multicore systems. In addition, `Paradyn` imposes an overhead proportional to the number of instrumented locations.

`Perf` [8] is one of the built-in Linux kernel tracers which was designed to access the performance counters in the processors. Its use was however later extended to interface with the `TRACE_EVENT()` macro and thus access the Linux kernel tracepoints. Yet, `perf` is mostly oriented towards sampling. It is possible to use `perf` as a regular tracer but it has not been optimized for this use. If sampling does allow low-overhead, making it interesting for real-time systems, it does so by sacrificing accuracy. Furthermore, the collection process is based on an interrupt, which is both costly and invasive. Finally, `perf`'s multicore scalability is limited [9].

The *Function Tracer*, or `ftrace`, is a set of different tracers built into the Linux kernel [10]. It was created in order to follow the relative costs of the functions called in the kernel to determine the bottlenecks. It has since evolved to include more comprehensive analysis modules such as latency or scheduling analysis [11]`Ftrace` is directly managed through the `debugfs` pseudofilesystem and works through the activation and deactivation of its tracers. It can connect to the static tracepoints in the kernel through its `event tracer` using the `TRACE_EVENT()` macro [12]. It collects only data defined in this macro using the `TP_printk` macro, to save analysis time on the tracer side. This behavior, however, comes with the drawback of not being able to add system context information to trace events. Finally, `ftrace` can also connect to userspace applications using UProbes, since Linux kernel 3.5. This instrumentation is using interruptions though, which adds unacceptable overhead for most real-time and high performance applications and systems.

`SystemTap` [13] is a monitoring system for Linux primarily aimed at the community of system administrators. It can instrument dynamically the kernel using KProbes [14] or interface with static instrumentation provided by the `TRACE_EVENT()` macro. It can also be used to instrument userspace applications using UProbes, since Linux kernel 3.8. The instrumentation is done in both cases using a special scripting language that is compiled to a kernel module. The data analysis is directly bundled inside the instrumentation and the results can be printed at regular interval on the console. As far as we know, the analysis being done in-flight, there are no efficient built-in facilities to write events to stable storage. Moreover, userspace probes as well as kernel probes, even if they have been statically compiled in precise places, incur an interrupt to work. If this interrupt is avoidable on the kernel side by using only the static instrumentation provided by `TRACE_EVENT()`, this is not possible on the userspace side. Interrupts add overhead that can be problematic for real-time tracing.

`LTTng-UST` provides macros for adding statically compiled tracepoints to programs. Produced events are consumed using an external process that writes events to disk. Unlike `Feather-Trace`, `LTTng-UST` uses the Common Trace Format, allowing the use of arbitrary event types [15].

The architecture of this tracer is designed to deliver high performance. It allocates per-CPU ring-buffers to achieve scalability and wait-free properties for event producers. Moreover, control variables for the ring-buffer are updated using atomic operations instead of locking. Also, read-copy update (RCU) data structures are used to protect important tracing variables. This avoids cache-line exchanges between readers that occur with traditional read-write lock schemes [16, 17]. A similar architecture is available for tracing at the kernel level. Moreover, kernel and userspace timestamps use the same clock source, allowing events to be correlated across layers at the nanosecond scale. This correlation is really useful to understand the behavior of an application. Finally, previous work demonstrated `LTTng`'s ability for high performance tracing of real-time applications [18]. `LTTng` is therefore the best candidate to trace real-time and multicore systems while correlating userspace and kernel activities.

2.2. Model-Checking Analysis and Data Extraction Tools for Traces. In this section, we present different approaches used for model-checking analysis on traces. We also review interesting tools aiming at extracting data from traces.

`Tango` [19, 20] is an automatic generator of backtracking trace analysis tools. It works using specifications written in the `Estelle` formal description language. It is based on a modified `Estelle-to-C++` compiler. `Tango` generates tools that are specific to a given model and that allow checking the validity of any execution trace against the specifications, using a number of checking options. However, `Tango` can only be used for single-process specifications and needs a NIST X Windows Dingo Site Server to do its analysis. Moreover, it was mainly designed to validate protocol specifications and therefore does not provide a way to specify constraints based on the system's state.

Other algorithms to automatically generate trace checkers are presented in [21]. These algorithms follow the same idea as `Tango` as they use formulas written in a formal quantitative constraint language, Logic of Constraints, in correlation with traces. They can thus analyze a traced simulation for functional and performance constraint violations. The specifications file is converted to `C++` source, which is then compiled to generate an executable checker. Using simulation traces, the executable will produce an evaluation report mentioning any constraint violation. However, this tool uses text-format traces and is thus very sensitive to any change in the trace format.

`Scalasca` [22] aims to simplify the identification of bottlenecks using execution traces. It offers analysis using both aggregated statistical runtime summaries and event traces. The summary report gives an overview of which process, in which call-path, consumes times and how much. The event traces are used for a deep study of the concurrent behavior of programs. `Scalasca` analyzes the traces at the end of the execution to identify wait states and related performance properties. It then produces a pattern-analysis report with performance metrics for every function call-path and system resource. If it allows extracting interesting metrics from the runtime of an application, `Scalasca` does not allow providing our own specifications.

`SETAF` [23] is a framework to adapt the system execution traces and dataflow models to have the required properties for analysis and validation of the QoS. `SETAF` works using `UNITE`, which describes a method to use system execution traces in order to validate the distributed system QoS properties [24]. `SETAF` acts as an overlay used by `UNITE` to transform the traces and provide the missing information. To do so, it requires the user to first manually analyze the execution trace to identify what properties need to be added to the dataflow model and thus provide the correct adaptation pattern. This adaptation pattern will allow adding information leading to the creation of a valid execution flow and new causality relations between log formats in `UNITE` but requires the user to have deep understanding of the trace format and `UNITE` requirements.

`Trace Compass` [25] is a graphical interface in `Eclipse` for the `LTTng` tracing tools. It supports multiple types of trace formats and provides different views showing specific analysis of the traces. Amongst these views, `Trace Compass` provides analysis for real-time applications [26] and an analysis of the system-level critical path of applications [27]. The latter aims to recover segments of execution affecting the waiting time of a given computation. Finally, `Trace Compass` also allows the creation of state system attribute trees and storing metrics throughout time in the state history tree database. This database provides efficient queries to the modeled state of the traced system for any given point in time.

To our knowledge, model analysis is not yet exploiting all the available information. By combining model analysis and trace analysis tools, the gap of unused information can be reduced. This would allow the specified behavior of the system to be verified through its execution trace, during or after running our application. Previous work has also been done on automatic kernel trace analysis using pattern matching, through state machines [28]. This work shows that trace events could be used to follow the workflow on an application and thus link the states of a state machine to the states of a running application.

3. Using Model-Based Constraints to Detect Unwanted Behaviors

When designing a high performance application, the developers usually know what they expect their application to do. They know the order of the operations to perform and different metrics along with their average values. It is in fact these values that allow the developers to verify that their application is performing well and doing what they want it to do.

In this section, we will present our approach, which uses finite state machine models and constraints over kernel and userspace traces to detect unwanted behaviors in programs. These models will require instrumented applications to delimit the constraints application. We will first detail the

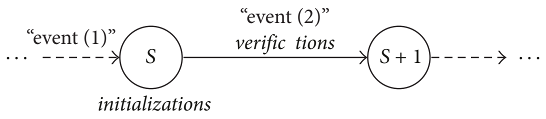


Figure 1: State machine representation that can be used to check metrics using traces.

general representation and then propose some model-based constraints that could be applied to existing applications.

3.1. General Representation. Whether it is to check a limit in terms of time or resources used by an application, metrics are usually taken between two states during the execution. We first have the start state, appearing before the application's work that we want to check. This state serves as a base to calibrate our metrics. We then have the second state, the end state for that check, at which point we can validate that we are within the limits.

Even if, during debugging, these states are usually read by the developer knowing the application, they can be fixed in the application workflow using a state machine representation. This representation can then be used to analyze constraints. Events generated from userspace tracepoints can thus be used to identify the state changes in our application.

3.1.1. Internal Structure. Our representation is based on four elements: the states, the transitions, the variables, and the constraints. The states are here to represent the different states of our application. The transitions represent the movement from a state to itself or another. The state changes in the traced application can be identified and replicated in the traced system model through the events received in the trace.

The variables are used to get and store the values of the metrics we need to verify. There are three main categories of variables: the state system free variables (not based on the state system such as those used to store timestamps or values available directly from the received events), the counter variables (or counters, such as those used to store the number of system calls throughout time), and the timer variables (or timers, such as those used to store the time spent running a process). The variables are categorized depending on the number of calls needed to get their value from our state system.

Our state system is based on the `Trace Compass` state attribute tree. We build our own state history tree database containing the different metrics that we want to keep accessing later during the analysis. These metrics and their evolution are thus saved to a file during the first analysis of the kernel trace and are thereafter accessible using simple requests to the state system. The state system free variables, counter variables, and timer variables are variables that, respectively, need 0, 1, and 2 calls to our state system to obtain their values at a given timestamp. This means that state system free variables can be read directly from the userspace trace, while counters and timers need a kernel trace to be available.

Counters do not need more than one call to our state system as their value is considered being the last one encountered: once a counter is incremented, it will keep this value until the next incrementation. On the other side, the new value of a timer is stored in the state system at the end of the activity, adding up to that timer. This means that when requesting the value of a timer at a given timestamp, we need to verify if the timer is currently running. We thus need to get the last value of the timer and its next value to interpolate the current running value.

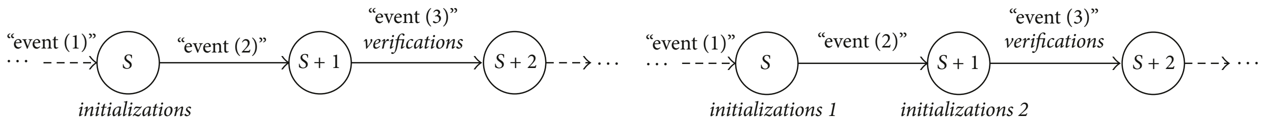
The constraints are used to express specifications of the expectations for the run of the applications. They are composed of two operands and one operator. The operands are either variables or constant values to be compared. The operator is one of the standard relational operators, equal (==), not equal (≠), greater (>), greater or equal (≥), less (<), or less or equal (≤).

Three validation status are available for the constraints: valid, invalid, and uncertain. The valid status means that the constraint was satisfied. The invalid status means that the constraint was not satisfied. In both those cases, we were able to read the variable and compare it to the requirement. In some cases, however, when there is missing information, a constraint cannot be verified. This is, for instance, the situation of constraints over counters or timers when there is no kernel trace available for the analysis and thus no state system built. In those cases, the constraint validation status is considered as uncertain.

The constraints are linked to a transition and will be checked when this transition is reached. The transition will thus have a validation status that will be the worst case of its constraints statuses. Therefore, having at least one invalid constraint is sufficient to know that the transition did not satisfy the constraints. If there is no invalid constraint, but at least one uncertain constraint, we cannot guarantee that all the requirements were met for that transition, thus making it uncertain. Finally, a transition will be valid if and only if all of its linked constraints are valid.

All those elements will allow building our model used to identify instances of our application in the traces. The instances are identified using their thread id. The variables are currently local to an instance of the application and are thus not shareable.

3.1.2. Models. Figure 1 shows a representation of a section of a state machine for an application where we would like to verify some metrics. The states in the figure are named "*S*" and "*S* + 1," respectively, for the start and end states of the zone we want to check.



(a) The verifications will use initializations that only appears at state "S" (b) The verifications will use both initializations declared at different states

Figure 2: State machine representation with late verification of constraints and a transitional state.

The "event (1)" string represents the event that would be used to enter state S of the state machine and the "event (2)" string would be the one used to move from state S to state $S + 1$.

The "*initializations*" string represents the diverse variables' initializations we would need to do in order to verify our metrics. The initialization of a variable is represented by setting this variable to 0. For instance, for a variable v of type *type*, we would write "type/$v = 0$."

Finally, the "*verifications*" string represents the list of constraints we would verify when passing from state "S" to state "$S + 1$," that is, when reading an event of type "event (2)."

Both initializations and verifications are discretionary for a state or transition, but events are still needed to follow the application workflow. That allows following a strict order of events to move forward in the application, without necessarily having metrics to check at this point.

This also allows us to initialize variables at one state but to only check them at a later state of our state machine, as shown in Figure 2(a). The period of the constraint would then only be larger than if we used a more recent initialization. This also allows checking multiple constraints at one point, while the initializations appeared at different states of the application workflow, as shown in Figure 2(b). Having larger check periods would not add up to the verification time, since constraints validations are done in a constant number of operations for a given category of constraint, as explained in Section 3.1.1.

It is also possible for a state to have two (or more) next states. It would still be possible to validate the related constraints. In such cases, different events would be used for each of the possibilities, as shown in Figure 3. When an event is reached while in state "S," we would automatically know if that event was of the type "event (2)" or "event (3)." We thus would be able to move to the right state and thus to read and execute the related verifications, if any.

Finally, in our approach, an event can be used at each junction of the model, but only once per junction. This removes any uncertainty about the flow to follow, in order to verify the constraints. Using this and the possibility to have multiple exits per node, we could, for instance, allow executing the initializations each time we encounter an event of type "event (1)," to only verify metrics between the last event of type "event (1)"and the first event of type "event (2)." Figure 4 shows a representation of this example. If we want to implement this specific example, we would not define any constraint in "verifications 1."

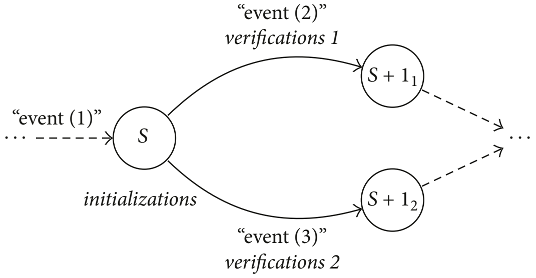


Figure 3: State machine representation with multiple next states for state "S."

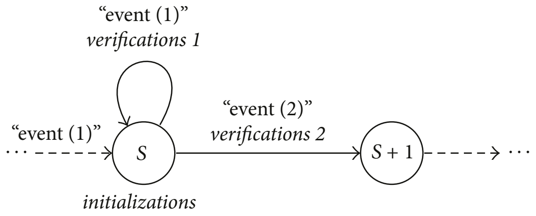


Figure 4: State machine representation using a loop, to go over the initializations when reading an event of type "event (1)."

3.2. Specific Constraints on Metrics. This section gives an overview of some constraints that we can specify on different metrics of the applications or the system. This overview extends the deadline constraint, already present in most real-time analysis tools based on constraint verification, to our new system-specific constraints. Our new constraints take advantage of the kernel-level information, about kernel internals and processes, available in our detailed execution traces.

3.2.1.Deadline Constraint. Real-time is as much about logical determinism as it is about temporal determinism. In such applications, we consider that a deadline has to be satisfied for the result to be correct, and we have to take into account the maximum allowed time to get that result.

Figure 5 gives a model representation of a constraint that could be used in that case. State "S" is the state of the application when starting the time-related task. State "$S + 1$" is the state of the application when it finishes that task. The events of types "event (1)" and "event (2)" are the events generated by tracepoints in the application when switching to those states.

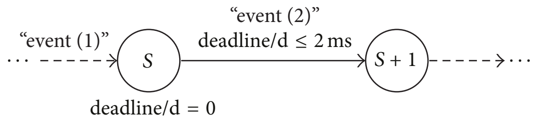


Figure 5: State machine representation of a constraint validating whether we spent at most 2 ms between the states "S" and "$S + 1$."

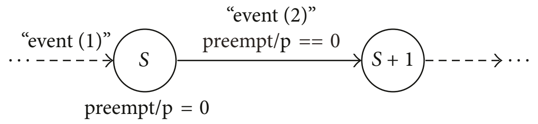


Figure 6: State machine representation of a constraint validating that our process has not been preempted between the states "S" and "$S + 1$."

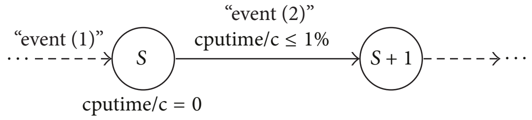


Figure 7: State machine representation of a constraint validating whether our process used at most 1% of the CPU time between states "S" and "$S + 1$."

When entering the "S" state, a timer must be initialized to know the time spent before reaching the "$S + 1$" state. That initialization is represented with the string "deadline/d = 0" on the model, initializing a deadline variable "d." Programmatically, this would be done using the read event that informs us that we enter this state. In this case, this event was of type "event (1)." We thus read the event time and set it as our base.

When entering the "$S + 1$" state, that timer must be checked to validate that we spent less than the duration limit. That verification is represented with the string "deadline/d ≤ 2 ms" on the model. Programmatically, we would use the base previously set for the "deadline/d" variable as well as the read event informing us that we enter the "$S + 1$" state. We would then compare both those values and verify that the difference between these times is less than or equal to 2 ms.

We thus would only need userspace traces to check a constraint of this type. The deadline constraint is using a system state free variable.

3.2.2. Preemption Constraint. When designing a high performance application, some tasks can be highly sensitive. In such case, any preemption could be disrupting the application work. We thus usually design our application to be able to work without being interrupted by another task, for instance, by setting a high priority.

Figure 6 gives a model representation of a constraint that can be used to limit the number of types of preemption that the process suffers during the given period, delimited by the "S" and "$S + 1$" states.

When entering the "S" state, a preemption counter has to be initialized to know how many types of preemption the process has experienced when reaching the "$S+1$" state. In the figure, we initialize a preemption counter variable "p" using the string "preempt/p = 0." When entering the "$S + 1$" state, the preemption counter variable is checked to validate that we did not have any preemption, using the string "preempt/p == 0." We could also have allowed at most one preemption, for instance, using the string "preempt/p ≤ 1."

Programmatically, we would use the "sched_switch" kernel events to know when the process is scheduled and unscheduled, using the events of types "event (1)" and "event (2)" to limit the search zone. For each "sched_switch" encountered for which we preempt our process (i.e., for which our process enters in a wait-for-CPU state), we can increment our initialized preemption counter. All this work is done directly in our state system. Once we reach the constraint, we thus only need to get the difference between the value of the preemption counter at the timestamp when we entered the "S" state and the value of the preemption counter at the timestamp when we entered the "$S + 1$" state. Using that difference, it is then possible to validate or invalidate the requirement.

This constraint is complementary to the deadline constraint. Indeed, an application could reach a given deadline while having been preempted, and in reverse an application could fail a deadline while not having been preempted. They could thus be used together to enforce a high performance condition verification. In typical cases, the deadline is ultimately the important constraint, but any preemption, even a short one that does not cause a deadline failure, may be an indication of the possibility that longer preemption could happen that would cause a deadline failure.

The preemption constraint is using a counter variable.

3.2.3. Resource Usage Constraint. Whether it is a minimum or a maximum, it can be useful to limit the usage of the resources of a system such as the CPU, raw access memory or even disk, or network input/output. Taking the example of the CPU usage, we could consider, for instance, that our application is doing a really simple job and thus should not use more than 1% of the CPU time during a given period delimited by two states. We could also consider that our application work is so important during a given period that it should be using 100% of the CPU time (no preemption or waiting).

Figure 7 gives a model representation of a constraint that could be used in that later case and could be easily changed to be used for the former.

When entering the "S" state, a CPU usage timer must be initialized to know the time spent using the CPU when reaching the "$S + 1$" state. That initialization is represented with the string "cputime/c = 0" on the model, initializing a CPU usage timer "c." When entering the "$S + 1$" state, this CPU usage timer must be checked to validate that we used at most 1% of the CPU. That verification is represented with the string "cputime/c ≤ 1%" on the model.

Programmatically, we use the events of types "event (1)" and "event (2)" to delimit the time period during which we look at the CPU usage. Using the kernel traces for the same time period, we can know which process was running on

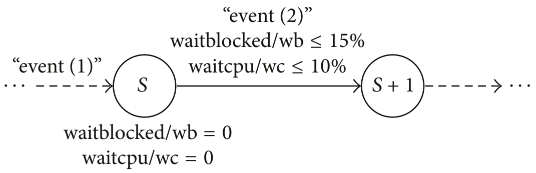


Figure 8: State machine representation of constraints validating whether the process spent at most 10% of the time period between states "S" and "$S+1$" waiting for a CPU and at most 15%of this same period being blocked.

which CPU for how long. With that information, we can sum the running durations of our process and compare that information to the total time period duration. This value is actually computed in our state system allowing getting the actual value at the time of "event (1)" and "event (2)" using only two state system calls for each. Using both those values, we can get the difference and compare it to the limit (1% in our example) to check if our constraint is validated or not.

The resource usage constraint is using a timer variable.

3.2.4. Wait Status Constraint. Following the CPU usage constraint, it could be as interesting to limit how much time a process is spending in "wait-for-CPU" or "wait-blocked" status. These constraints are thus complementary to the previous one.

Figure 8 shows a representation of both "wait-for-CPU" and "blocked" status constraints being used on a model.

On this model, the "wait-for-CPU" status constraint "wc" is initialized using the string "waitcpu/wc = 0" while the "wait-blocked" status constraint "wb" is initialized using "waitblocked/wb = 0." They are then checked using constraints to limit the "wait-for-CPU" status of the process to at most 10% of the time period between the two states and the "wait-blocked" status to at most 15% of that same time period.

Programmatically, the events of types "event (1)" and "event (2)" would allow delimiting the working time period. We would then look at the kernel events in that period to check the status of our process and compute the time percentage spent in the status we want to check, in the same way that we computed this information for the CPU usage constraint, but using the new state of the unscheduled process to know if it is now waiting for CPU or blocked. This information is directly computed in our state system; we can thus access it easily for the given interval and verify our constraint.

The wait status constraint is using a timer variable.

3.2.5. System Calls Constraint. High performance applications are sometimes designed to work only in userspace during their critical inner loop performing the real-time task. This helps remove any latency that can be caused by other processes, the hardware, or other resources in the system. This is, for instance, the case when a user process gets the proper permissions to access directly some I/O addresses, for interacting with external inputs and outputs through an FPGA card connected to the PCIe bus. In that case, these input and output operations completely avoid any interaction through the operating system. Other common cases of communications that bypass the operating system are accesses through shared memory buffers, synchronized by native atomic operations. In such cases, we would want to verify that the process remained in userspace for all its scheduled time. Using a system calls constraint could be useful in such cases.

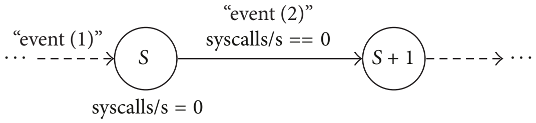


Figure 9: State machine representation of a constraint validating that our process has not done any system call between states "S" and "$S+1$."

Figure 9 gives a model representation of a constraint that can be used to limit the number of system calls issued by the process during the given period.

When entering the "S" state, a system calls counter variable is initialized. In the figure, the system calls counter variable "s" is initialized using the string "syscalls/s = 0." When entering the "$S+1$" state, we check this counter to validate that we did not have any system call since the "S" state. We use the string "syscalls/s == 0" to do so.

Programmatically, we count the number of kernel events whose name starts by "syscall_entry," using the events of types "event (1)" and "event (2)" to limit the search of these events. For each event encountered that matches our search, we can increment our system calls counter. We can then compute the difference for that counter and use that difference to check against the requirement.

The system calls constraint is using a counter variable.

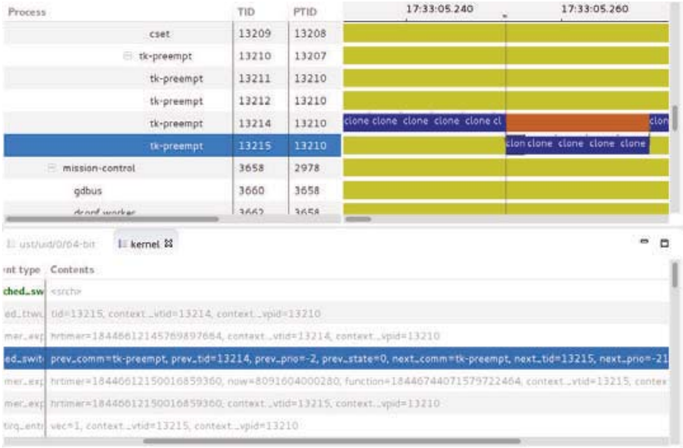
4. Case Studies

This section presents different case studies of common problems; each one is extracted from a real industrial problem that we solved using tracing.

4.1. Occasional Missing of Deadlines

4.1.1. Problem Summary. In real-time systems, we have to comply with the given deadlines for a task. That task can happen multiple times in a short period of time. In some cases that we encountered, a task happening up to 1000 times per second was missing its deadline one or two times per second. That task being a hard real-time one, those missed deadlines were not acceptable.

4.1.2. Trace Analysis Approach. Kernel and userspace traces were used to identify the task execution and see what happened on the kernel side. These traces lead to see that, for each task that did not reach its deadline in time, another process of higher priority was scheduled instead. That process was not scheduled the rest of the time, letting the other



Process	TID	PTID
cset	13209	13208
tk-preempt	13210	13207
tk-preempt	13211	13210
tk-preempt	13212	13210
tk-preempt	13214	13210
tk-preempt	13215	13210
mission-control	3658	2978
gdbus	3660	3658
dconf worker	3662	3658

17:33:05.240
17:33:05.260
clone clone clone clone clone cl
clon
clon clone clone clone clone
ust/uid/0/64-bit
kernel

nt type	Contents
ched_sw	<srch>
ed_ttwu	tid=13215, context._vtid=13214, context._vpid=13210
mer_exp	hrtimer=18446612145769897664, context._vtid=13214, context._vpid=13210
ed_swit	prev_comm=tk-preempt, prev_tid=13214, prev_prio=-2, prev_state=0, next_comm=tk-preempt, next_tid=13215, next_prio=-21
mer_exp	hrtimer=18446612150016859360, now=8091604000280, function=18446744071579722464, context._vtid=13215, contex
mer_exp	hrtimer=18446612150016859360, context._vtid=13215, context._vpid=13210
irq_entr	vec=1, context._vtid=13215, context._vpid=13210

(a) Screenshot of Trace Compass showing the preemption

```
[17:33:05.252828753] (+0.000000748) computer
     sched_switch: { cpu_id = 2 }, { vtid =
    13214, vpid = 13210 }, { prev_comm = "tk
    -preempt", prev_tid = 13214, prev_prio =
     -2, prev_state = 0, next_comm = "tk-
    preempt", next_tid = 13215, next_prio =
    -21 }
```

(b) Trace event "sched_switch" happening to do the preemption

Figure 10: Preemption of a process by another higher priority (−2 versus −21; the lower the value, the higher the priority).

tasks—having the same system priority as the ones missing their deadlines—reach their deadlines.

Figure 10(a) shows a visualization of such situation in Trace Compass. We can see that the thread of TID 13214 is running uninterrupted while the thread of TID 13215 is in wait-blocked status (yellow on the figure). As soon as the thread 13215 exits its wait-blocked status, it is scheduled on the CPU, instead of thread 13214. The latter is then in wait-for-CPU status until the former finishes its task and returns in wait-blocked status. As shown in Figure 10(b), that preemption was caused because the priority of the thread of TID 13214 was only of −2 (field "prev_prio") while the one of the thread of TID 13215 was of −21 (field "next_prio"). In this case, we need to read the priorities in reverse, meaning that thread 13215 had a higher priority and thus preempted the other while it was running.

4.1.3. Using Model-Based Constraints. The application could be represented using our model approach, setting at least two states, one for the beginning of each task subject to a deadline and one for its end. We could here use a deadline constraint to be informed each time we have not finished our task in time, limiting the search for problems to precise zones.

Depending on what we expect our application to do, we could also take advantage of other constraints like a preemption constraint or a CPU usage one to get more information as to why we do not follow the expected workflow. These constraints would, however, need kernel traces to be verified. Figure 11shows the state machine representation of our example using all three mentioned constraints.

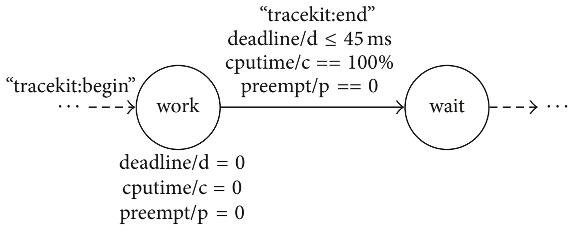


Figure 11: State machine representation of `tk-preempt`'s work using constraints to check if our process spent at most 45 ms working, used 100% of the CPU time, and was not preempted during its critical real-time task.

4.2. Priority Inversion

4.2.1. Problem Summary. Some high performance processes have to be running all the time. In such situations, the system and process are usually configured to favor that status, permanently running, by setting a high real-time priority and affinity to an isolated CPU, for instance. Still, in some instances, our high performance process is preempted when it should not.

In previous work [18], we wanted to allow tracing such applications. We thus created a minimal `UST`-traced application doing only loops and calculating their duration and saw that a delay was unfortunately added when tracing. Our application, even while being the highest priority one in the system, was unscheduled at some point while being traced.

4.2.2. Trace Analysis Approach. Using kernel traces, it is possible to see the different processes being scheduled and compare their priority.

In the LTTng case, kernel and userspace traces were used to trace the execution of a minimal UST-traced application doing only loops. The application was instrumented using three UST tracepoints, generating three different kinds of events: "start" and "stop" for, respectively, the start and stop of the application's run and "loop" for each iteration of the low-latency internal loop of the application. We thus could identify the period of time for which we had higher latencies and look at the kernel traces to see what happened on the system side. We identified that our application, using the LTTng-UST library, was at some point using sys_rt_sigpending, sys_rt_sigprocmask, and write system calls. Those calls were allowing the kernel to take control of the CPU and thus to schedule waiting kernel workers or tasks that were waiting to be executed, even if their priority is lower. This scheme is the one that created a priority inversion in our high performance situation.

Figure 12shows the high duration between the two "loop" events of the npt application. We can see in Figure 12(a)the presence of system calls in between the two events. Looking further at the trace, we can see a number of system calls running on the scheduled CPU, as listed in Figure 12(b). We can also see that "npt:loop" events are only 0.48 μs apart when there is no system call.

4.2.3. Using Model-Based Constraints. With our model approach, we can use a state identifying that our application is in a loop and for each event read that informs us we are doing another iteration of the loop ("npt:loop" in our example), a constraint would be validated. This constraint could be a CPU usage constraint, for instance, ensuring that our application had at least a high share of its CPU. We could otherwise use a preemption constraint, to limit the number of times that our application has been preempted during that iteration of the loop. Finally, if we consider that our application should only be working in userspace during the given period of time, a syscall constraint can be used. Figure 13 shows the state machine representation of our example using both a CPU usage and a syscall constraint.

4.3. Unefficient Synchronization Method

4.3.1. Problem Summary. Synchronization between the different threads and processes of a multicore application is often the hardest part of the design. In high performance applications, we want to be sure that the thread or process waiting for another will spend only the necessary amount of time waiting and be able to resume its activity as soon as possible. For some programs, however, the sleep command is used as a synchronization method and is thus adding unusual latencies in a usually efficient workflow. When using such programs, that problem is not always obvious as potentially hidden by other tasks. Moreover, using sleep as synchronization either is simply unsafe or implies that we have strict upper bounds on the duration of some portions of tasks such that the sleep duration is sufficient to finish the task we are waiting for. Furthermore, this method is based on the worst duration case and is rarely a good choice.

The application apt is the package manager used in Debian-based distributions. MongoDB is an open-source database software. They have in common that they were both, at some point, using sleep (or equivalent functions usleep or nanosleep) to do synchronization in their multithreaded tasks.

4.3.2. Trace Analysis Approach. Using kernel traces, we can identify the status of a process as wait-blocked status and use a waiting dependency analysis to identify the origin of the waiting status of our process.

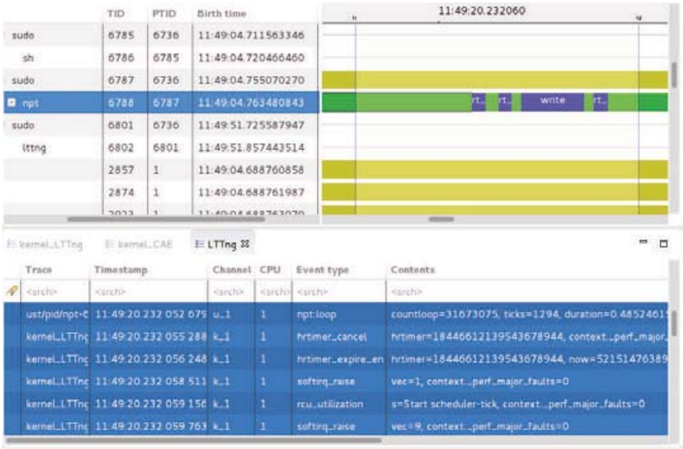
The kernel traces were used to identify what apt was waiting for in its installation process. Indeed, we found, by tracing an apt installation, that 37% of the time along the critical path was spent by the program in a wait-blocked status. This is surprising because if we are waiting for another process to produce useful results, the other process (and not the wait) becomes part of the critical path. A wait along the critical path is caused by events such as timers (sleep) or external events. Amongst the different processes created by apt, a long sleep was found along the critical path, identified using the perf toolchain as associated with a call to nanosleep, used to "give [the child process] time to actually exit and produce its results, avoiding an attempt to read the results before they were ready."

Kernel traces also identified that the same synchronization strategy was used in MongoDB. Knowing that there was an unusual, infrequent, long latency in the run of batch insert commands sent to the MongoDB server, we used tracing and trace comparison to understand what was happening for those instances. We then found that a sleep was used to wait for some delay before trying to obtain a hazard pointer to a page of data. That delay was inserted to allow another thread to finish its task, if it was trying to evict that page from the cache of MongoDB at the same time.

4.3.3. Using Model-Based Constraints. Considering that the application should normally have well-bounded delays for its tasks, we could use our model approach to represent the application normal task and use deadline constraints to verify that we are not having unduly long latencies. For the MongoDB situation, this could be set as having 1 s to 2 s deadline, since most commands run in less than 700 μs but were exceeding 3 s about once every 10 000 commands.

4.4. Wait-Blocked Processes on Multiprocessor Activity

4.4.1. Problem Summary. Processes sometimes expect high performance for multithreaded tasks on a multicore system. In these cases, cache access and synchronization are usually optimized to achieve a good scalability. However, it may



	TID	PTID	Birth time
sudo	6785	6736	11:49:04.711563346
sh	6786	6785	11:49:04.720466460
sudo	6787	6736	11:49:04.755070270
npt	6788	6787	11:49:04.763480843
sudo	6801	6736	11:49:51.725587947
lttng	6802	6801	11:49:51.857443514
	2857	1	11:49:04.688760858
	2874	1	11:49:04.688761987

Trace	Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
ust/pid/npt-6	11:49:20.232 052 675	u_1	1	npt:loop	countloop=31673075, ticks=1294, duration=0.4852461
kernel_LTTng	11:49:20.232 055 288	k_1	1	hrtimer_cancel	hrtimer=18446612139543678944, context._perf_major_
kernel_LTTng	11:49:20.232 056 248	k_1	1	hrtimer_expire_en	hrtimer=18446612139543678944, now=52151476389
kernel_LTTng	11:49:20.232 058 511	k_1	1	softirq_raise	vec=1, context._perf_major_faults=0
kernel_LTTng	11:49:20.232 059 156	k_1	1	rcu_utilization	s=Start scheduler-tick, context._perf_major_faults=0
kernel_LTTng	11:49:20.232 059 763	k_1	1	softirq_raise	vec=9, context._perf_major_faults=0

(a) Screenshot of Trace Compass showing the period between two "npt:loop" events in the application

```
[..51386] npt:loop: { cpu_id = 1 }, {
    countloop = ..3, .., duration = 0.485246
     }
[..51871] npt:loop: { cpu_id = 1 }, {
    countloop = ..4, .., duration = 0.484496
     }
[..52668] npt:loop: { cpu_id = 1 }, {
    countloop = ..5, .., duration = 0.485246
     }
[..55300] hrtimer_cancel: { cpu_id = 1 }, ..
[..56260] hrtimer_expire_entry: { cpu_id = 1
     }, ..
[..58523] softirq_raise: { cpu_id = 1 }, ..
[..59168] rcu_utilization: { cpu_id = 1 },
    ..
[..59775] softirq_raise: { cpu_id = 1 }, ..
[..60238] rcu_utilization: { cpu_id = 1 },
    ..
[..60810] hrtimer_expire_exit: { cpu_id = 1
    }, ..
[..61303] hrtimer_start: { cpu_id = 1 }, ..
[..62923] sys_rt_sigpending: { cpu_id = 1 },
     ..
[..64118] exit_syscall: { cpu_id = 1 }, ..
[..65228] sys_rt_sigprocmask: { cpu_id = 1
    }, ..
[..66368] exit_syscall: { cpu_id = 1 }, ..
[..67190] sys_write: { cpu_id = 1 }, ..
[..70615] sched_wakeup: { cpu_id = 1 }, ..
[..72728] exit_syscall: { cpu_id = 1 }, ..
[..73547] sys_rt_sigprocmask: { cpu_id = 1
    }, ..
[..74773] exit_syscall: { cpu_id = 1 }, ..
[..77392] npt:loop: { cpu_id = 1 }, {
    countloop = ..6, .., duration = 24.5571
    }
```

(b) Kernel events traced between the two "npt:loop" events showing kernel tasks running while the application is waiting to continue its work, causing latency

Figure 12: Unexpected kernel work while tracing an userspace-only application.

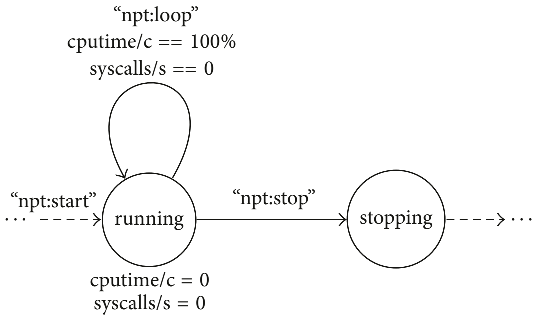


Figure 13: State machine representation of `npt`'s loop using constraints to check if our process used 100% of the CPU time between each loop iteration.

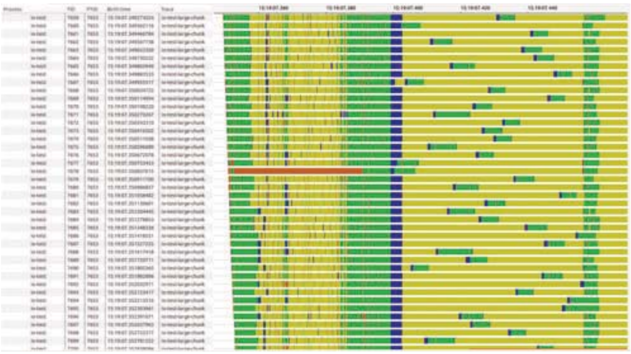


Figure 14: Screenshot of Trace Compass showing the barrier at which threads are waiting before unmapping operations, after their calls to `munmap` [2].

happen that some part of the task misses these optimizations and does not scale well, causing regressions when using parallel cores. For high scalability multithreaded processes, this behavior should be avoided.

An occurrence of that problem was encountered while we were searching the point at which a heavy I/O highly parallel application becomes I/O-bound. That application used the `mmap` system call in different threads to map different parts of a file. We were puzzled to measure that when using 64 threads on a 64-core machine, the execution time was 10 times slower than with just one thread, even if the threads are totally independent from each other.

4.4.2. Trace Analysis Approach. Using only kernel traces and looking at those with a visualizing tool, the regression appearing in that last example was identified. The processes seemed to all be waiting for the last calling thread before unmapping and ending their respective calls to `munmap`. This also appeared (but not as clearly) for the `mmap` calls.

This behavior is normally associated with the use of a barrier, as seen in Figure 14. We were able to find in the Linux kernel source code that Linux uses a global semaphore protecting the `mm_struct` data structure. We thus found a solution to circumvent the problem for our application, using a unique thread for memory mapping. The scalability of the `mmap` system call was also analyzed in [29].

4.4.3. Using Model-Based Constraints. If we consider the application as the one of highest priorities on the system, a CPU usage constraint could be efficient to know if the process is really taking advantage of the CPU. This constraint would show if the CPU usage is not sufficient, compared to our expectations. We could also use a wait-blocked status constraint, stating that our application should not spend more than a given time percentage in wait-blocked status. With one of those constraints, we would detect that situation.

4.5. Wait-Blocked Processes While Using External Resources

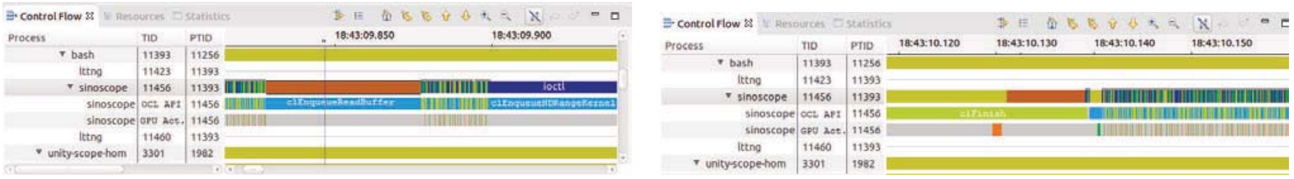
4.5.1. Problem Summary. External resources are necessary in some cases to perform specific tasks. For instance, for some highly parallel computing tasks, GPUs are becoming more and more interesting, as compared to multicore CPUs. In such cases, computation or data rendering depends on another processing unit, different from the one running our application. In high performance situations, if the CPU work is highly dependant on the GPU work and if the GPU work is not optimized, bottlenecks will appear and our process will be in wait-blocked status.

That problem was encountered while we wanted to know if an application running on a CPU and requesting GPU work was optimized.

4.5.2. Trace Analysis Approach. Kernel and userspace traces can here be useful while using a visualizing tool. In the previous example, we added userspace tracepoints in the API calls to OpenCL to get more information about what happened in the GPU. We thus were able to use the generated events to understand the origin of a latency in a given process.

In some situations, the latency was induced by CPU preemption: the GPU had finished its work but is unable to get back to the process, currently unscheduled or already busy, as shown in Figure 15(a).In some other cases, as we can see in Figure 15(b) it was linked to GPU sharing: the process was in wait-blocked status, waiting for the GPU to get back to it, while the GPU was not working on that task, being already busy on another one. As a solution for such situations, the process itself and its GPU tasks can be optimized to improve their use of the available resources. On the CPU side, it could be using a higher priority for the process to prevent preemption. On the GPU side, it could be a better division of the tasks.

4.5.3. Using Model-Based Constraints. The CPU preemption could easily be detected by using our model approach. Before calling the external resource (the GPU in this case), we could enter a "external resource call" state and once that resource answers, we could enter a "external resource answered" state, for instance. We could then use a preemption constraint or a wait-blocked status constraint to ensure that our process does not end up unscheduled from its CPU.



Process	TID	PTID
▾ bash	11393	11256
lttng	11423	11393
▾ sinoscope	11456	11393
sinoscope	OCL API	11456
sinoscope	GPU Act.	11456
lttng	11460	11393
▾ unity-scope-hom	3301	1982

(a) Unified CPU-GPU view showing the process unscheduled from its CPU causing wait on the GPU side

Process	TID	PTID
▾ bash	11393	11256
lttng	11423	11393
▾ sinoscope	11456	11393
sinoscope	OCL API	11456
sinoscope	GPU Act.	11456
lttng	11460	11393
▾ unity-scope-hom	3301	1982

(b) Unified CPU-GPU view showing the process waiting for the GPU while the GPU is still working on another task

Figure 15: Views showing wait situations while using external resources; these views do not exist in the Trace Compass mainline version yet [3].

Table 1: Number of events and sizes of the traces used to benchmark our analysis.

Name	Number of events			Size (MiB)
	UST	Kernel	Total	
tk-preempt (1)	20 015	932 778	952 793	35.40
tk-preempt (2)	800 015	10 961 881	11 761 896	508.8
modelbench (3)	102 400	13 510 489	13 612 889	352.6

The GPU sharing would however be trickier to detect with the current resources as LTTng does not implement shared CPU-GPU traces yet. If we have an idea of the duration of the GPU task, we could use a wait-blocked status constraint on our process, stating that if our process is spending more than a given time in wait-blocked status, something is probably wrong on the resource side. Having combined CPU and GPU traces could help to further detect this kind of problems.

5. Analysis Results and Time

Using the case presented in Section 4.1, the associated detection constraints, and the model represented in Figure 11 to automatically identify its unwanted behavior, we ran our analysis and benchmarked the time it took to analyze different traces detailed in Table 1. This case has the ability to provide interesting results, as we can use for it the three different categories of variables available in our approach.

5.1. Constraints Validation. Figure 16 gives an overview of the different outputs of our analysis, depending on the constraints satisfaction and available data. These results are just a section of the full report containing all the instances of the application, according to our model and their analysis results. For these examples, we used the trace 1. For all those results, the first part shows the entry in the work state when receiving a “tracekit:begin” event and the entry in the wait state when receiving the “tracekit:end” event. We also have the timestamp of the read event, and we can see the list of initialized variables and constraints, if any.

Results presented in Figures 16(a) and 16(b) were computed using both the userspace and kernel traces. In Figure 16(a), we can see that the requirements were satisfied, and thus each constraint is in the valid status. In Figure 16(b), none of the requirements were met, thus setting all constraints to the invalid status. We can see in that latter case that the computed value is shown in the report to understand why the requirements were not met. The results presented in Figure 16(c) were computed using only the userspace trace. We can see that given only the userspace trace, the analysis was not able to verify if all the constraints were satisfied and thus set the CPU usage and preempt constraints to uncertain status, while the deadline constraint has been analyzed and is, in this case, invalid as the time spent in the work state was 45.4054 ms while the maximum was 45 ms.

Figure 17 shows some invalid sections, as reported by our tool for other cases presented in Section 4. Figure 17(a) shows a section in which even though the application discussed in Section 4.2 was still scheduled and should have been running only in userspace, four system calls were executed, making the section last for more than 49 μs. Figure 17(b) shows that MongoDB, discussed in Section 4.3, took much more time than expected for running a command. Finally, Figure 17(c) shows that the application discussed in Section 4.4 spends an unexpected amount of time in the wait-blocked status.

5.2. Running Time. Switching on and off the different constraints put in the model represented in Figure 11,we benchmarked the running time of our analysis. Our test system consist of an Intel® Core™ i7-4810MQ CPU at 2.8 GHz, with 16GiB of DDR3 RAM at 1600 MHz. Table 2 shows the results for trace 1 while Table 3 shows the results for trace 2.

Given the different numbers of userspace and kernel events in each trace, we can see the different baseline times needed to build the state system and the model and verify the constraints when no constraint is active (*None*). We thus see that having around 12 times more kernel events makes the state system build time around 10 times longer. On the userspace side, having around 40 times more events makes the model build time around 35 times longer.

Amongst the constraints, we can see that for both traces the deadline constraint is the fastest to compute, followed by the preemption and finally the CPU usage. This is coherent with the fact that state system free constraints do not need complementary data to be computed, while counters need two state system calls for the interval and timers four calls.

For each trace, however, we can see in Tables 2 and 3 that the state system build time is always the same, independently of the active constraints. It thus only depends on the kernel

```
Received tracekit:begin at 18:27:53.143 850
    080
     Entering state: work
     Variables:
         - deadline/d = 0
         - cputime/c = 0
         - preempt/p = 0
Received tracekit:end at 18:27:53.173 146
    432
     Entering state: wait
     Constraints:
         - deadline/d <= 45ms [VALID]
         - cputime/c == 100% [VALID]
         - preempt/p == 0 [VALID]
```

(a) Result shown following the analysis of both kernel and userspace traces when the constraints are satisfied

```
Received tracekit:begin at 18:27:53.173 147
    428
     Entering state: work
     Variables:
         - deadline/d = 0
         - cputime/c = 0
         - preempt/p = 0
Received tracekit:end at 18:27:53.218 552
    778
     Entering state: wait
     Constraints:
         - deadline/d <= 45ms [INVALID] value
             : 45.4054ms
         - cputime/c == 100% [INVALID] value:
              99.9806%
         - preempt/p == 0 [INVALID] value: 1
```

(b) Result shown following the analysis of both kernel and userspace traces when the constraints are not satisfied

```
Received tracekit:begin at 18:27:53.173 147
    428
     Entering state: work
     Variables:
         - deadline/d = 0
         - cputime/c = 0
         - preempt/p = 0
Received tracekit:end at 18:27:53.218 552
    778
     Entering state: wait
     Constraints:
         - deadline/d <= 45ms [INVALID] value
             : 45.4054ms
         - cputime/c == 100% [UNCERTAIN]
         - preempt/p == 0 [UNCERTAIN]
```

(c) Result following the analysis of the userspace trace only to simulate a case where we would not have any kernel trace, thus making the state system unavailable for the analysis

Figure 16: Results of the analysis using the model-based constraints on userspace and kernel traces.

```
Received npt:loop at 16:53:03.116 356 658
        Entering state: running
        Variables:
                 - cputime/c = 0
                 - syscalls/s = 0
[...]
Received npt:loop at 16:53:03.116 405 899
        Entering state: running
[...]
        Constraints:
                 - cputime/c == 100% [VALID]
                 - syscalls/s == 0 [INVALID]
                      value: 4
```

(a) Invalid section for the case presented in Section 4.2 when verifying the model represented in Figure 13, during which four (4) system calls were issued

```
Received mongodb:combegin at 21:31:52.618
    207 853
        Entering state: command
        Variables:
                 - deadline/d = 0
Received mongodb:comend at 21:31:57.557 893
    474
        Entering state: wait
        Constraints:
                 - deadline/d < 1s [INVALID]
                      value: 4.9397s
```

(b) Invalid section for the case presented in Section 4.3 when verifying a deadline constraint of less than one (1) second for a task length, which lasted nearly five (5) seconds in this case

```
Instance TID: 41176
Received cache:begin at 13:31:22.023 214 053
        Entering state: mmapping
        Variables:
                 - waitblocked/wb = 0
Received cache:end at 13:31:22.231 503 275
        Entering state: waiting
        Constraints:
                 - waitblocked/wb < 10% [
                     INVALID] value:
                     15.8541%
```

(c) Invalid section for the case presented in Section 4.4 when verifying a wait-blocked constraint of less than ten percent (10%) for a task, which spent more than fifteen percent (15%) of its time being blocked in this case

Figure 17: Examples of invalid sections as reported by our tool for other cases discussed in Section 4.

trace size. This is because the state system is computed to acquire all the metrics necessary to set constraints at once, no matter which ones are actually used. While this behavior is costly for the first run, the state history tree database built is saved in stable storage to allow fast access for the following runs, as shown in Table 4.

5.3. Scalability. The last validation step of our approach has been to verify its scalability. As our model-based analysis uses both traces and models, we needed to validate scalability on those two different aspects.

In order to measure the scalability relative to trace length, we generated a number of traces containing events needed to follow the model presented in Figure 11. Each data point presented in Figures 18 and 19 is the average elapsed execution time over twenty runs of our algorithm.

Figure 18 presents the results using the different userspace traces and our different categories of constraints. We can see in the figure that, for all categories of constraints used, the complexity of the approach is proportional to the number of userspace events. We also observe that it takes about twice as much time to use timer variables as compared to counter

Table 2: Average (avg.) and standard deviation (atd. dev.) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace tk-preempt (1).

Constraints	Time (in s)		
	State system build	Model build & constraint verif.	Total
Deadline			
Avg.	3.8500	0.34352	4.1935
Std. dev.	0.23075	0.024853	0.23532
CPU usage			
Avg.	3.8592	0.72952	4.5888
Std. dev.	0.20432	0.047765	0.21079
Preemption			
Avg.	3.8271	0.46162	4.2887
Std. dev.	0.17792	0.043790	0.18840
All three			
Avg.	3.8609	1.1251	4.9860
Std. dev.	0.22394	0.042746	0.22484
None			
Avg.	3.8312	0.15789	3.9891
Std. dev.	0.17902	0.017863	0.17954

Table 3: Average (avg.) and standard deviation (std. dev.) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace tk-preempt (2).

Constraints	Time (in s)		
	State system build	Model build & constraint verif.	Total
Deadline			
Avg.	29.663	6.9751	36.638
Std. dev.	0.84278	1.1635	1.5931
CPU usage			
Avg.	29.599	41.223	70.821
Std. dev.	1.1128	1.0844	1.7550
Preemption			
Avg.	29.462	22.220	51.682
Std. dev.	1.0688	0.59945	1.2692
All three			
Avg.	29.766	58.855	88.620
Std. dev.	1.1275	1.2559	1.8220
None			
Avg.	30.049	5.2234	35.272
Std. dev.	0.98000	0.91995	1.3683

variables. Both those variables are more expensive than using a state system free variable.

Figure 19 shows the results using the different kernel traces. It shows that the time it takes to build the state system is proportional to the number of kernel events in the trace.

Table 4: Average (avg.) and standard deviation (std. dev.) of the time taken in seconds to build the state system during the first run versus to verify if it exists in the subsequent runs, computed using 100 runs of the analysis of the traces.

Trace	Time (in s)	
	Build	Access
tk-preempt (1)		
Avg.	3.8457	0.0516
Std. dev.	0.20404	0.00611
tk-preempt (2)		
Avg.	29.708	0.0539
Std. dev.	1.0463	0.00764

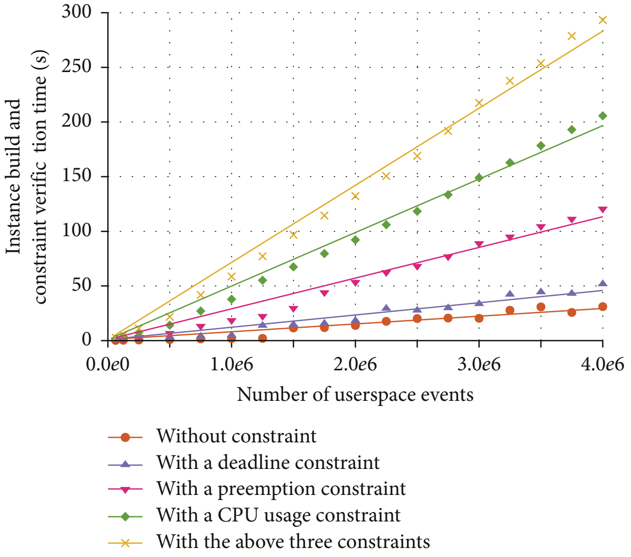


Figure 18: Time (in s) to build the instances and check their constraints as a function of the number of userspace events. Lines represent linear regressions of the data.

Also, the linear regressions in both Figures 18 and 19 show that the proportionality follows a linear pattern for both kernel and userspace traces.

To analyze the model scalability, we used the trace *model-bench (3)* that contains 100 calls to 1024 different tracepoints. Each data point presented in Figures 20, 21, and 22 is the average elapsed execution time over twenty runs of our algorithm.

We used this trace to first consider a model without constraint and with only one transition per state, to analyze the scalability according to the number of successive states in the model, as shown in Figure 20. We see that the time it takes to build the instances is proportional to the number of states involved.

We then analyzed a situation in which the number of states was fixed, but the number of transitions from one state to the other was variable. Figure 21 shows the results for this case that does not support any constraint, where the

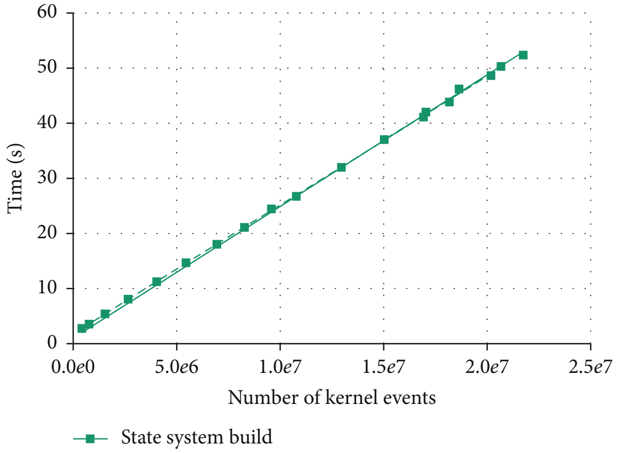


Figure 19: Time (in s) to build the state system as a function of the number of kernel events. The line represents a linear regression of the data.

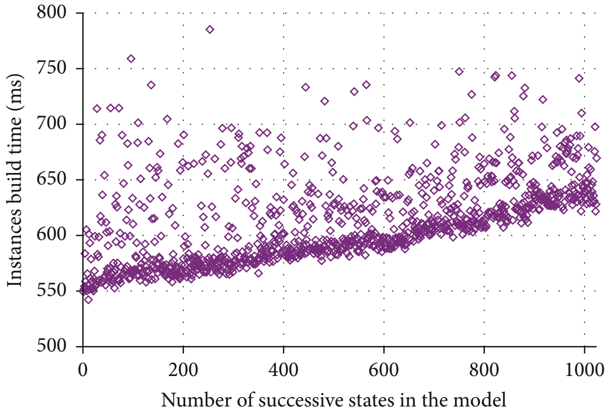


Figure 20: Time (in ms) to build the instances as a function of the number of successive states in the model.

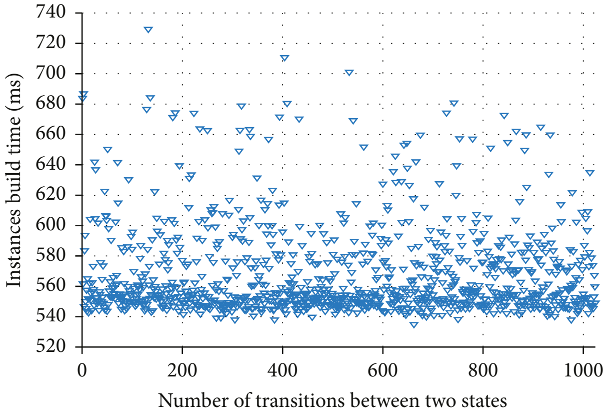


Figure 21: Time (in ms) to build the instances as a function of the number of transitions between two states.

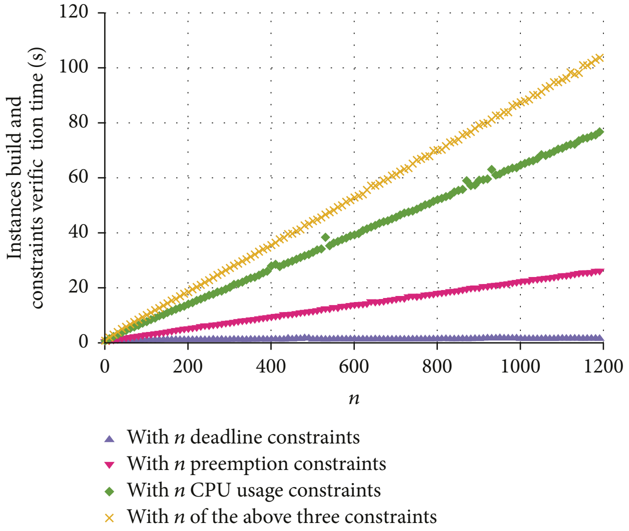


Figure 22: Time (in s) to build the instances and check their constraints, as a function of the number and categories of constraints between two states.

transitions are each based on a different trace event. These results illustrate that the number of transitions between two events does not impact the time it takes to build the instances.

Finally, we studied the constraints scalability by fixing the number of states and transitions and by varying the number of constraints on that transition. We repeated that test for the three different categories of constraints and for a case using one constraint of each category. Figure 22 shows the results of those tests. We can observe that in each case the time is linearly proportional to the number of constraints involved.

All those tests allow us to validate that our approach executes in time linearly proportional to the trace length and model size. It will thus take more time to analyze a bigger trace, as it will be longer to follow and check a model with more nodes and constraints.

6. Conclusion and Future Work

We have presented our approach for application modeling, using model-based constraints, and kernel and userspace traces, to automatically detect unwanted behavior in real-time and multicore applications. We presented how our models use tracepoints to follow the application workflow. We then proposed some constraints, using userspace and kernel traces information, to validate application behavior. We detailed multiple cases where tracing has been helpful to identify an unexpected behavior and explained how our model approach could have saved time by automatically identifying those behaviors. Finally, we presented the results produced by our approach and the associated execution time as well as its scalability relative to trace length and model complexity.

We believe that using model-based constraints on top of userspace and kernel traces has a great potential to automate performance analysis and problem detection. We intend to pursue our work to use model-based constraints not only to detect problems but also to identify their root cause. We could also use this information to allow our approach to propose simple solutions to common real-time and multicore problems, such as raising the priority of a process if it was preempted but should not have been.

Disclosure

This work represents the views of the authors and does not necessarily represent the view of Polytechnique Montreal. Linux is a registered trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

The authors are grateful to Mathieu Côté, David Couturier, François Doray, Francis Giraldeau, and Fabien Reumont-Locke for the cases studied in this paper. 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] L. Aceto, A. Burgueño, and K. G. Larsen, "Model checking via reachability testing for timed automata," in *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, March 1998*, B. Steffen, Ed., vol. 1384 of *Lecture Notes in Computer Science*, pp. 263–280, Gulbenkian Foundation, 1998.

[2] F. Reumont-Locke, *Méthodes efficaces de parallélisation de l'analyse de traces noyau [M.S. thesis]*, École Polytechnique de Montréal, Québec, Canada, 2015.

[3] D. Couturier and M. R. Dagenais, "LTTng CLUST: a system-wide unified CPU and GPU tracing tool for OpenCL applications," *Advances in Software Engineering*, vol. 2015, Article ID 940628, 14 pages, 2015.

[4] B. Brandenburg and J. Anderson, "Feather-trace: a light-weight event tracing toolkit," in *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 61–70, July 2007.

[5] B. P. Miller, M. D. Callaghan, J. M. Cargille et al., "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11,pp. 37–46, 1995.

[6] R. Wismüller, M. Bubak, W. Funika, and B. Baliś, "A performance analysis tool for interactive applications on the grid," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 305–316, 2004.

[7] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '11)*, pp. 9–16, Szeged, Hungary, September 2011.

[8] J. Edge, *Perfcounters Added to the Mainline*, 2009, http://lwn.net/Articles/339361/.

[9] M. Desnoyers, *A New Unified Lockless Ring Buffer Library for Efficient Kernel Tracing*, 2010, http://www.efficios.com/pub/linuxcon2010-tracingsummit/presentation-linuxcon-2010-tracing-mini-summit.pdf.

[10] J. Edge, *A Look at Ftrace*, 2009, http://lwn.net/Articles/322666/.

[11] S. Rostedt, *Ftrace—Function Tracer*, 2008, https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[12] Ftrace: now and then, 2010, http://people.redhat.com/srostedt/trace-cmd-linuxcon-2010.odp.

[13] F. C. Eigler, "Problem solving with systemtap," in *Proceedings of the Linux Symposium*, vol. 1, pp. 261–268, Ottawa, Canada, July 2006.

[14] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[15] M. Desnoyers, *Common Trace Format (CTF) Specifications*, 2011, http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.txt.

[16] P. E. McKenney and J. Walpole, "Introducing technology into the Linux kernel: a case study," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 4–17, 2008.

[17] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, 2012.

[18] R. Beamonte, *Traçage de systèmes linux multi-coeurs en temps réel [M.S. thesis]*, École Polytechnique de Montréal, Montreal, Canada, 2013.

[19] S. A. Ezust, *Tango: the trace analysis generator [M.S. thesis]*, McGill University, Montreal, Canada, 1995.

[20] S. A. Ezust and G. V. Bochmann, "An automatic trace analysis tool generator for estelle specifications," in *Proceedings of the ACM SIGCOMM Conference*, pp. 175–184, Cambridge, Mass, USA, 1995.

[21] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of constraints: a quantitative performance and functional constraint formalism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1243–1255, 2004.

[22] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency Computation Practice & Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[23] T. M. Peiris and J. H. Hill, "Adapting system execution traces for validation of distributed system QoS properties," in *Proceedings of the 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '12)*, pp. 162–171, IEEE, Guangdong, China, April 2012.

[24] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, "Unit testing non-functional concerns of component-based distributed systems," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST '09)*, pp. 406–415, Denver, Colo, USA, April 2009.

[25] Trace Compass, https://projects.eclipse.org/proposals/trace-compass.

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

[27] F. Giraldeau, *Analyse de performance de systèmes distribués et hétérogènes à l'aide de traçage noyau [Ph.D. dissertation]*, École Polytechnique de Montréal, Montreal, Canada, 2015.

[28] G. Matni and M. Dagenais, "Automata-based approach for kernel trace analysis," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '09)*, pp. 970–973, May 2009.

[29] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, pp. 198–207, Austin, Tex, USA, October 2009.

Advances in Multimedia
The Scientific World Journal
International Journal of Distributed Sensor Networks
Journal of Industrial Engineering
Applied Computational Intelligence and Soft Computing
Advances in Fuzzy Systems
Modelling & Simulation in Engineering
Journal of Computer Networks and Communications
Advances in Artificial Intelligence
Advances in Computer Engineering
International Journal of Computer Games Technology
International Journal of Biomedical Imaging
Advances in Artificial Neural Systems
Advances in Software Engineering
Journal of Robotics
Advances in Human-Computer Interaction
Computational Intelligence and Neuroscience
International Journal of Reconfigurable Computing
Journal of Electrical and Computer Engineering