View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

DÉTECTION DE PROBLÈMES DANS LES SYSTÈMES TEMPS RÉEL PAR L'ANALYSE DE TRACES

MATHIEU CÔTÉ
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2015

© Mathieu Côté, 2015.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DÉTECTION DE PROBLÈMES DANS LES SYSTÈMES TEMPS RÉEL PAR L'ANALYSE DE TRACES

présenté par : CÔTÉ Mathieu
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GAGNON Michel, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. ADAMS Bram, Doctorat, membre

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche Michel Dagenais de m'avoir incité à entreprendre des études supérieures. Son support, sa grande expertise et sa disponibilité ont été essentiels à l'accomplissement de ce travail.

Je tiens ensuite à reconnaître le soutien financier apporté à mon projet de recherche par OPAL-RT, CAE, le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) et le Consortium de recherche et d'innovation en aérospatiale au Québec (CRIAQ).

Je tiens finalement à remercier mes amis et collègues de laboratoire pour les activités plaisantes, la bonne ambiance, les discussions et les conseils.

RÉSUMÉ

Les systèmes temps réel ont la particularité de devoir respecter des contraintes de temps. Ils sont donc difficiles à observer correctement puisqu'il faut éviter de venir perturber leur exécution. Dans cette situation, le traçage peut s'avérer essentiel. Cela consiste à collecter de l'information lors de l'exécution d'un système avec un impact minimal. Puisque les traces produites contiennent beaucoup de données, il est ensuite nécessaire d'extraire l'information pertinente à l'aide d'outils appropriés. Dans cette optique, l'objectif du présent travail consiste à améliorer la détection de problèmes dans les systèmes temps réel en développant un outil d'analyse de traces spécialisé pour ces systèmes.

La nouvelle approche proposée consiste à permettre aux développeurs de définir le modèle de la tâche temps réel qu'ils souhaitent analyser. Puisque les tâches temps réel se produisent de manière répétitive, l'idée est de récupérer les segments de traces correspondants aux différentes exécutions de la tâche. Il est ensuite possible de comparer ces exécutions entre elles et d'analyser plus en profondeur celles qui semblent problématiques.

Pour définir le modèle d'une tâche temps réel, nous avons conçu une interface graphique permettant de spécifier les événements de la trace définissant une exécution. Comme cela peut s'avérer complexe, nous fournissons également des suggestions pertinentes. Pour ce faire, nous avons d'abord comparé différents algorithmes de reconnaissance de patrons. Puis, nous avons adapté celui qui semblait le plus prometteur pour qu'il fonctionne efficacement avec des traces d'exécution. Nous pouvons ainsi présenter les groupes d'événements qui semblent faire partie de tâches récurrentes. L'utilisateur peut ensuite choisir un des groupes pour qu'il forme la base du modèle d'exécution et l'éditer si nécessaire.

Une fois le modèle défini, nous cherchons dans la trace toutes les correspondances. Nous utilisons à cet effet des machines à état permettant de repérer les occurrences en effectuant une seule lecture des événements. Nous avons ensuite observé le fonctionnement des différents outils de visualisation de traces afin de sélectionner les meilleures représentations et de les améliorer en fonction de notre cas. Nous avons donc conçu un ensemble de vues permettant d'afficher les exécutions et de les comparer entre elles.

Ensuite, nous avons répertorié les principaux problèmes dans les systèmes temps réel. Nous avons également cherché des métriques pertinentes à collecter afin de repérer la présence de ces problèmes dans une exécution. En présentant des analyses spécifiques à certains problèmes et de l'information utile, nous sommes en mesure d'aider les développeurs à trouver pourquoi certaines exécutions présentent des anomalies. Ainsi, notre contribution principale consiste

à détecter les inversions de priorité en combinant l'analyse du chemin critique d'une tâche avec les informations d'ordonnancement.

Nous avons finalement validé notre approche à partir de résultats expérimentaux. Nous avons ainsi pu détecter la source de latences inexpliquées dans des applications industrielles de manière très efficace.

ABSTRACT

Real-time systems are characterized by their timing constraints. It means that the real time tasks must respect deadlines to avoid unwanted consequences. When only a few deadlines are missed, it can be hard to identify the underlying cause, due to the numerous components involved in the systems and their interactions. It often requires specialized tools. However, those tools must not disturb the system or slow down the tasks. In that situation, tracing can be useful or even essential. It consists in collecting selected events during the execution of a program, and the time at which they occurred. It is then possible to analyse the interesting parts of the resulting trace. However, the trace can be very large and the challenge is to retrieve relevant data in order to find quickly complex or erratic real-time problems. This motivates the goal of this work to develop a real time systems specialized tool to help developers.

We propose a new approach to efficiently find real-time problems based on the fact that the real time tasks are repetitive. We first provide a way for the users to define when those tasks are executing in term of tracing events. Then, the idea is to execute different analyses on the corresponding trace segments instead of the whole trace. This allows us to save huge amount of time and execute more complex analyses.

To define the execution model of a task, we conceive a graphical interface to let the users specify the trace events that must be include in an execution. Because some users may not know the events involved in their systems, we first proposed them predefined model of interest. We also provided some suggestions based on a pattern matching algorithm. The users can then load one of the model and edit it if necessary. This algorithm is based on an known algorithm that we adapt to make it efficient with traces.

Once the model is defined, we search in the trace all the sequences that match the model using state machines. This allow us to read the trace only once. Then, we show the resulting executions in a comparison view, to highlight those that are problematic.

Once we have selected executions that present irregularities, we execute different analyses on the corresponding trace segments. To make our analysis, we search for the common real-time problems and the relevant metrics. Our main contribution is an analysis that combine the search of the dependencies of a task with the scheduling information to detect scheduling problems.

The efficiency of the proposed method has finally been tested with multiple traces. We also successfully use our solution to identify the cause of problems in industrial applications.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CPU	Central Processing Unit
IRQ	Interrupt Request
LTTng	Linux Tracer Toolkit next generation
PCP	Priority Ceiling Protocol
PID	Process Identifier
PIP	Priority Inheritance Protocol
RAM	Random-Access Memory
RCU	Read-Copy-Update
TID	Thread Identifier

CHAPITRE 1 INTRODUCTION

Les systèmes temps réel ont la particularité de devoir respecter des contraintes de temps. Il est donc plus difficile d'analyser leur comportement puisqu'il faut éviter de les perturber. C'est particulièrement vrai si on souhaite observer une défaillance se produisant de manière erratique et qui implique un non-respect des échéances. Dans ce contexte, le traçage peut s'avérer nécessaire. Il consiste à collecter de l'information lors de l'exécution d'un système avec un impact minimal et d'analyser les données recueillies après coup. Comme les traces ainsi produites peuvent s'avérer très volumineuses, il convient d'avoir des outils pour repérer les événements d'intérêt et les présenter de manière cohérente. De plus, puisque les systèmes temps réel présentent des caractéristiques qui leur sont propres, il peut être avantageux d'utiliser des outils adaptés pour ces systèmes.

1.1 Définitions et concepts de base

Dans cette section, différents concepts de base seront présentés afin de faciliter la compréhension du mémoire.

1.1.1 Système temps réel

Les systèmes informatiques temps réel présentent la particularité de devoir tenir compte de contraintes de temps. En effet, dans ces systèmes, il ne convient pas de donner uniquement des résultats exacts, mais il faut également que ces résultats soient fournis en respectant les délais.

Types de systèmes temps réel

On peut regrouper les systèmes temps réel en trois types selon les conséquences occasionnées par le non-respect des échéances (voir Srinivasan et al., 1998). Les systèmes temps réel stricts (*hard real-time*) occasionnent de graves conséquences quand une échéance n'est pas respectée, les systèmes temps réel fermes (*firm real-time*) n'occasionnent pas de graves conséquences, mais le résultat n'a plus aucune utilité lorsque l'échéance n'est pas respectée et les systèmes temps réel souples (*soft real-time*) voit la valeur du résultat se dégrader lorsque l'échéance est dépassée. On peut citer en exemple respectivement un appareil médical, une vidéoconférence et la lecture d'un enregistrement vidéo.

Tâches temps réel

Une tâche temps réel (*task*) peut être caractérisée par un temps de calcul, une échéance et optionnellement une période, alors que l'exécution d'une tâche peut être appelée une instance (*job*). On retrouve trois types de tâches (voir Isovic and Fohler, 2000) soit :

– Les tâches périodiques : il s'agit de tâches qui sont démarrées à intervalle régulier et qui ont un délai plus petit ou égal à leur période.
– Les tâches sporadiques : il s'agit des tâches qui peuvent arriver à des moments arbitraires dans le temps, bien qu'il y ait souvent un temps minimal entre les occurrences. Elles ont un délai relatif à leur temps de démarrage.
– Les tâches apériodiques : il s'agit des tâches qui ne rentrent pas dans les deux autres catégories. On retrouve ici, entre autres, les tâches qui sont exécutées une seule fois à un moment connu.

1.1.2 Ordonnancement

L'ordonnanceur est la composante du système d'exploitation qui se charge de décider quel fil d'exécution (*thread*)s'exécute sur quel processeur. Pour choisir les prochains fils d'exécution à s'exécuter, l'ordonnanceur utilise les politiques d'ordonnancement associées à chaque fil d'exécution de même que leur priorité. Dans la plupart des systèmes d'exploitation, il est possible de préempter un fil d'exécution qui s'exécute, ce qui signifie de le remplacer par un autre alors qu'il est encore exécutable.

Sous Linux, on peut distinguer 3 types de priorité. Les priorités temps réel sont assignées aux fils d'exécution ayant une politique d'ordonnancement temps réel. Il s'agit des priorités les plus grandes. Les autres fils d'exécution se voient plutôt assigner une priorité statique qui va rester constante. Ces fils d'exécution ont également une priorité dynamique basée sur leur priorité statique, mais qui varie selon le temps d'exécution du fil d'exécution et qu'il est possible de modifier par un appel à *nice*, *setpriority* ou *sched_setattr*. C'est généralement la priorité dynamique qui est utilisée pour ordonnancer les fils d'exécution dans les politiques qui ne concernent pas le temps réel.

1.1.3 Politiques d'ordonnancement

Politiques régulières

Il existe six politiques d'ordonnancement sous Linux (voir Kerrisk et al., 2014). Il y en a trois régulières :

– SCHED_OTHER : Il s'agit de la politique d'ordonnancement par défaut qui fonctionne de manière à répartir le temps de équitablement entre les fils d'exécution qui utilisent cette politique. Plus précisément, les fils d'exécution se voient assigner une priorité statique de 0. On modifie ensuite la priorité dynamique des fils d'exécution qui sont prêts, mais ne parviennent pas à s'exécuter afin qu'ils puissent être sélectionnés. Un temps d'exécution maximal appelé quantum va être attribué aux processus afin de permettre la rotation.
– SCHED_BATCH : Il s'agit d'une politique d'ordonnancement pour exécuter des fils d'exécution en augmentant les temps pour lesquels ils s'exécutent et en pénalisant les changements de contexte. C'est utile lorsqu'on a un ensemble de tâches qui ne sont pas dépendantes les unes des autres et qui ne sont pas interactives.
– SCHED_IDLE : Il s'agit d'une politique d'ordonnancement pour exécuter des fils d'exécution uniquement quand il n'y a pas d'autres fils d'exécution en attente d'être exécuté sauf ceux qui utilisent également cette politique. En effet, les fils d'exécution se voient attribuer la plus basse priorité possible.

Politiques temps réel

Il y a également trois politiques temps réel :
– SCHED_FIFO : Il s'agit d'une politique d'ordonnancement temps réel basé sur le principe de file (First In First Out). Les fils d'exécution ainsi ordonnancés ont une priorité statique plus grande que les fils d'exécution n'utilisant pas de politique temps réel et vont donc toujours les préempter. Plus précisément, les fils d'exécution sont classés en liste par priorité et l'ordonnanceur exécute les fils d'exécution se retrouvant au début de la liste de plus haute priorité non vide. Lorsqu'un nouveau fil d'exécution se voit attribuer cette politique, il se retrouve au début de la liste correspondant à sa priorité et va donc préempter un fil d'exécution de priorité inférieure ou égale. Lorsqu'un fil d'exécution redevient exécutable ou fait appel à *sched_yield*, il se retrouve à la fin de la liste correspondant à sa priorité. Finalement, lorsqu'un fil d'exécution est préempté, il va au début de sa liste et reprend donc son exécution lorsque le fil d'exécution l'ayant préempté se retrouve bloqué ou a terminé.
– SCHED_RR : Il s'agit de la même chose que SCHED_FIFO, mais les fils d'exécution ont en plus un temps maximal d'exécution avant d'aller à la fin de la liste correspondant à leur priorité. Cela constitue donc un système *Round Robin* pour les fils d'exécution de même priorité.
– SCHED_DEADLINE : Il s'agit d'une politique d'ordonnancement où il faut définir un temps d'exécution, un délai et une période pour le fil d'exécution. Le temps d'exécution est ici le temps maximal que peut prendre la tâche. Avec ces informations, l'ordonnan-

ceur calcule s'il est possible d'ordonnancer correctement la tâche et retourne une erreur autrement lors de l'assignation de la politique. Puisqu'il s'agit de la plus grande priorité possible, les fils d'exécution qui ont cette politique préemptent tous les autres. Afin de respecter toutes les échéances, les fils d'exécution sont priorisés selon le temps restant avant leur échéance.

1.1.4 Correctif PREEMPT_RT

Le correctif PREEMPT_RT (voir McKenney, 2005) permet d'améliorer les performances temps réel du noyau Linux. Pour ce faire, on réduit les parties du code du noyau qui ne peuvent pas être préemptées en rendant les sections critiques préemptibles. De cette manière, un fil d'exécution sera remplacé par un autre de plus haute priorité même s'il se trouve dans une section critique. De plus, on remplace les *interrupts handler* par des fils d'exécution noyau (*kernel threads*) qui peuvent être préemptés. On implémente également l'héritage de priorité pour les sémaphores et les verrous du noyau. Cela réduit donc au minimum les moments où une tâche ne pourra pas être préemptée. L'héritage de priorité garantit ainsi un temps d'exécution pour une tâche qui requiert une échéance stricte.

Les changements apportés constituent les nouveaux comportements par défaut, mais de nouvelles primitives implémentant les anciens comportements sont tout de même fournies pour les cas particuliers. Ainsi, on peut par exemple utiliser les *raw_spinlock* pour rendre une section critique non préemptible, les *compat_semaphores* pour ne pas avoir d'héritage de priorité ou *SA_NODELAY* dans la *struct irqaction* pour ne pas que l'interruption soit traitée par un fil d'exécution noyau. L'utilisation de ces primitives est toutefois découragée puisque cela nuit à la prédictibilité du système.

1.1.5 Traçage

Le traçage est une méthode utilisée pour obtenir de l'information sur l'exécution d'un programme. Contrairement au débogage durant lequel on exécute le programme pas à pas, on collecte plutôt des événements sans arrêter l'exécution. Pour ce faire, il convient tout d'abord d'instrumenter le code. Cela peut être fait de manière statique ou dynamique. Ensuite, un traceur doit enregistrer les événements désirés. Cela ressemble beaucoup à un système de journalisation, mais on enregistre généralement davantage d'événements et ceux-ci peuvent être de bas niveau. Vu la quantité de données récoltées qui peut s'avérer énorme, il est souvent nécessaire d'utiliser un outil pour analyser les événements collectés. L'analyse peut se faire a posteriori ou en temps réel, selon les traceurs et les outils d'analyse utilisés.

Événement

Un événement peut être défini comme survenant à un instant précis. Il est donc caractérisé par son type et le moment auquel il a été généré. On peut également enregistrer des informations supplémentaires utiles. Par exemple, l'événement *sched_switch* est généré lors de l'ordonnancement d'un nouveau fil d'exécution et contient comme supplément d'information les identifiants des fils d'exécution, leur priorité et l'état du fil d'exécution sortant.

Point de trace

Un point de trace est un endroit dans le code d'une application ou du système d'exploitation pouvant mener à la génération d'un événement. Ceux-ci peuvent être activés ou désactivés. Lorsqu'ils sont rencontrés et actifs, des fonctions permettant l'enregistrement par les traceurs d'information spécifique peuvent être appelées.

1.2 Éléments de la problématique

Les systèmes temps réels sont essentiels dans de nombreux domaines comme l'aéronautique, les télécommunications ou le domaine médical. Ceux-ci doivent avoir des mécanismes permettant de garantir que les échéances des différentes tâches seront respectées. Le développement du correctif *PREEMPT_RT* a permis l'utilisation de Linux comme système d'exploitation pour ces systèmes. En effet, Linux est très performant et polyvalent. Avec l'application du correctif, on y ajoute des caractéristiques permettant de satisfaire à des contraintes temps réel strictes. De plus, plusieurs des modifications apportées par le correctif ont depuis été intégrées à la version standard du noyau. Cela permet, entre autres, d'exécuter des applications temps réel à contraintes souples en obtenant de bonnes performances.

Ces divers développements ont mené à un nombre grandissant d'applications temps réel utilisant Linux. Il y a donc un besoin pour des outils de qualité permettant d'observer le comportement de ces applications. Or, le débogage des systèmes temps réel peut s'avérer difficile. En effet, il faut éviter de venir altérer le comportement du système, par exemple en bloquant une application pour analyser son état. Cela peut mener à un non-respect des échéances ou modifier le comportement de l'application. De plus, l'analyse s'avère encore plus complexe quand seulement une faible proportion des délais ne sont pas respectés. Dans ces cas, le profilage non plus n'est généralement pas d'une grande utilité puisque les nombreuses exécutions qui se déroulent correctement viennent masquer les statistiques des quelques exécutions problématiques.

À l'inverse, le traçage est une excellente solution. Comme mentionné précédemment, cela consiste à collecter des événements spécifiques et le moment auquel ils se sont produits. Or, il est possible de tracer en ayant un impact suffisamment minime sur le système et d'enregistrer des renseignements relatifs autant au noyau du système d'exploitation qu'à une application donnée. On peut ensuite analyser l'information recueillie en utilisant une autre machine ou après la fin de l'exécution. Cela constitue toutefois une importante quantité de données qu'il convient de traiter pour identifier la source des problèmes.

Il serait théoriquement possible d'analyser manuellement les traces d'exécution, mais cela prendrait énormément de temps et demanderait une bonne connaissance des événements générés, entre autres, par le noyau du système d'exploitation. C'est particulièrement vrai dans le cas de systèmes multicoeurs qui sont généralement plus difficiles à comprendre puisque plusieurs événements peuvent se produire simultanément sur des coeurs différents. Il est donc préférable d'utiliser des outils pour présenter les éléments pertinents dans la trace. De même, dans le cas de problèmes qui se produisent rarement, une complexité supplémentaire provient de la recherche des segments de trace présentant des anomalies.

Dans les systèmes temps réels, on retrouve souvent des problèmes typiques qui ne sont pas nécessairement présents lorsqu'une application n'a pas de contraintes de temps. On peut donner comme exemple les cas d'ordonnancements problématiques qui viennent ralentir des tâches prioritaires. Il serait donc intéressant de créer un outil d'analyse de traces qui permette de repérer efficacement les problèmes associés à ces systèmes.

Plus précisément, on retrouve généralement des tâches qui s'exécutent plusieurs fois, que ce soit de manière sporadique ou périodique. Il peut donc s'avérer pertinent de pouvoir repérer les segments de trace correspondant aux différentes exécutions afin de pouvoir les comparer. Cela demande donc d'abord de pouvoir définir un modèle d'exécution pour ensuite rechercher les correspondances dans la trace. Il serait également intéressant de pouvoir utiliser des informations pertinentes comme celle concernant la priorité des différentes tâches afin de pouvoir aider les développeurs dans l'identification de la source de problèmes. On peut également mentionner qu'il serait bien d'avoir un outil fonctionnant, peu importe les politiques d'ordonnancement utilisées.

1.3 Objectifs de recherche

L'objectif général de la présente recherche est d'améliorer la détection de problèmes et de leurs sources dans les systèmes temps réel en développant un outil d'analyse de traces.

De manière plus spécifique, les objectifs sont :

1. Identifier les informations pertinentes à collecter dans une trace pour analyser des systèmes temps réel
2. Développer une méthode permettant à un utilisateur de définir le modèle d'exécution d'une tâche temps réel spécifique dans une trace
3. Développer une méthode permettant de repérer dans une trace les exécutions correspondant à un modèle défini
4. Développer une méthode permettant de repérer si une exécution présente un problème
5. Présenter les exécutions problématiques dans une vue
6. Valider l'approche à partir de résultats expérimentaux

1.4 Plan du mémoire

Le chapitre 2 présente d'abord une revue critique de la littérature dans les domaines du traçage, de l'analyse de trace, de la découverte de patrons et des problèmes dans les systèmes temps réels. Puis, le chapitre 3 fait état de la méthodologie utilisée pour atteindre nos objectifs de recherche et présente la pertinence de notre article, "Problem detection in real-time systems by trace analysis", qui est reproduit au chapitre 4. Une discussion sur l'ensemble des travaux suivra au chapitre 5. Finalement, le chapitre 6 conclura sur les limitations de notre solution et les améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

Cette section passe en revue les articles pertinents et l'état de la technologie concernant l'analyse de traces dans les systèmes temps réel. Plus précisément seront présentés les différents outils de traçage, les principaux problèmes dans les systèmes temps réel, les outils d'analyses de traces et les méthodes de détection de patrons dans une trace.

2.1 Traçage sous Linux

Comme mentionné précédemment, le traçage est une méthode utilisée par les développeurs logiciels pour obtenir de l'information sur l'exécution d'un programme. Pour ce faire, il convient tout d'abord d'instrumenter le code. Cela peut être fait de manière statique ou dynamique.

2.1.1 Instrumentation statique

L'instrumentation statique consiste à insérer des points de trace dans le binaire d'une application avant son exécution. Pour instrumenter de manière statique le noyau Linux, la macro TRACE_EVENT() a été développée (voir Rostedt, 2010). Elle permet d'ajouter un point de trace et de créer une fonction de rappel (*probe*) associée à ce point de trace qui peut enregistrer des informations quand il est rencontré. De plus, la fonction de rappel a accès aux variables locales dont elle peut enregistrer la valeur. Les points de trace ainsi générés sont fonctionnels pour différents traceurs dont perf, SystemTap, Ftrace et LTTng. De plus, les points de trace peuvent être activés ou désactivés dynamiquement. L'instrumentation statique présente l'avantage d'être plus rapide à l'exécution que l'instrumentation dynamique, mais cela demande de recompiler le programme instrumenté.

2.1.2 Instrumentation dynamique

À l'inverse, l'instrumentation dynamique se fait à l'exécution. Outre le fait de ne pas avoir à recompiler, cela permet d'ajouter des points de trace dans les parties qui semblent problématiques en cours d'exécution.

Tout d'abord, les *Kernel Probes* (voir Corbet, 2009) permettent d'ajouter des *probes* et des *probes handler* dans le noyau. Plus spécifiquement, il existe trois types de *kernel probes*, soient *kprobe*, *jprobe* et *kretprobe*, qui permettent d'instrumenter respectivement n'importe quelle instruction, les entrées de fonction, et les retours de fonctions. Cela fonctionne en remplaçant

l'instruction instrumentée par un point d'arrêt (*breakpoint*). Puis, quand le *breakpoint* est atteint, on exécute le *pre_handler*, l'instruction et finalement le *post_handler*. Un mécanisme similaire existe au niveau utilisateur soit les *uprobes* (voir Corbet, 2012) qui fonctionnent eux aussi en remplaçant l'instruction d'un programme par un point d'arrêt.

2.2 Traceurs pour Linux

Une fois le code instrumenté, il faut enregistrer les événements désirés lors de l'exécution. Pour ce faire, il est important d'avoir un traceur avec un impact minimal afin que la collecte ne dégrade pas les performances du système et que les informations collectées ne soient pas altérées. De même, il est préférable d'avoir une bonne mise à l'échelle dans les environnements avec plusieurs fils d'exécution.

2.2.1 LTTng

LTTng (Linux Tracing Toolkit next generation) est un ensemble d'outils de traçage pour Linux qui permet de tracer en espace noyau et en espace utilisateur (voir The LTTng Project, 2014). Un de ses principaux avantages est sa performance. Il présente en effet un impact minimal sur les applications tracées, en plus d'avoir le plus petit surcoût parmi les autres solutions équivalentes offertes sous Linux.

Plus précisément, LTTng est composé de trois composantes principales. D'abord, lttng-modules est un ensemble de modules noyau qui permet de tracer le noyau Linux en fournissant des fonctions de rappel, une implémentation de tampon circulaire pour enregistrer les données et le traceur. Ensuite, lttng-UST est une librairie de traçage en espace utilisateur. Finalement, lttng-tools est un ensemble de composants qui permet de contrôler les sessions de traçage. Il est possible de ne pas installer les modules noyau si on souhaite tracer simplement en espace utilisateur, ou à l'inverse de ne pas installer les librairies de traçage pour l'espace utilisateur. Cependant, un des avantages de LTTng consiste au fait qu'il peut corréler les traces prises en espace utilisateur et celles prises en espace noyau. Cela permet d'obtenir une information globale qui peut s'avérer utile pour des problèmes complexes.

À l'instar de plusieurs autres solutions de traçage, LTTng utilise l'instrumentation présente dans le noyau grâce à la macro TRACE_EVENT. À cela, il ajoute un niveau d'adaptation pour ajouter des fonctionnalités. On n'a donc pas à recompiler le noyau et cela permet d'avoir plusieurs *probes* pour un même point de trace. Si plus d'instrumentation est nécessaire, les *kprobes* peuvent également être utilisés pour ajouter des points de trace de manière dynamique dans le noyau.

Pour ce qui est du traçage avec LTTng-UST, la première option consiste à utiliser *tracef* comme un *printf* pour rapidement créer un point de trace. Autrement, un point de trace plus complexe peut être généré en utilisant l'utilitaire *lttng-gen-tp* ou encore en éditant manuellement les fichiers. Cela permet principalement d'ajouter des champs à l'événement généré. Dans tous ces cas, il est nécessaire de recompiler l'application puisqu'il s'agit d'instrumentation statique.

Pour arriver à avoir une aussi bonne performance, LTTng utilise différents mécanismes. D'abord, on utilise un tampon circulaire par processeur. Cela limite les échanges de données et permet donc de limiter les mécanismes de synchronisation. On peut toutefois noter que cela pourrait être amélioré. En effet, les systèmes temps réel utilisent souvent l'isolation des processeurs. Il serait intéressant de pouvoir tracer uniquement un groupe de processeurs et également de pouvoir attribuer des tampons de tailles différentes par processeur.

LTTng permet quand même une flexibilité pour la taille des tampons. En effet, il utilise un *session deamon* pour contrôler les sessions de traçage et il permet d'utiliser différents canaux pour tracer différents événements. Ces canaux peuvent avoir des tailles de tampons et un nombre de sous-tampons différents. Ensuite, un consommateur est utilisé pour le traçage noyau et un pour le traçage en espace utilisateur qui ont pour tâche de collecter les événements.

Parmi les techniques utilisées au niveau utilisateur, LTTng utilise la librairie Userspace RCU qui permet de lire et de mettre à jour des données de manière simultanée. C'est donc plus rapide puisqu'il n'est pas nécessaire d'avoir un verrou. Cela évite un retour dans le noyau. Ensuite, les événements sont écrits en mémoire partagée et un tube de contrôle est utilisé pour avertir le consommateur que le tampon est plein.

Un autre point positif est le format de la trace, soit CTF (Common Trace Format) (voir EfficiOS, 2015). C'est un format qui est flexible et permet facilement de modifier le type d'information que l'on veut enregistrer. De plus, la trace est écrite en binaire. Cela prend donc un décodeur, mais c'est beaucoup plus rapide à écrire et cela permet d'enregistrer plus d'information pour une trace de même grandeur. Pour décoder la trace, la première option est d'utiliser l'outil babeltrace qui vient avec LTTng et qui permet de visualiser la trace dans la console. Autrement, un outil graphique externe peut être utilisé pour l'analyser.

Ce désir de minimiser l'impact sur le système se manifeste également dans la façon de gérer les surplus d'événements. En effet, lorsque le traceur n'est pas capable d'enregistrer tous les événements, il laisse tomber des événements plutôt que de bloquer le système. Cela peut mener à la perte d'information pertinente puisque c'est souvent dans les moments où beaucoup d'événements arrivent que les systèmes présentent des problèmes. On peut toutefois

mentionner que LTTng présente un mode *flight recorder* qui permet d'enregistrer les événements en mémoire dans un tampon circulaire en écrasant continuellement les événements les plus anciens, et d'enregistrer le tampon dans une trace uniquement à la réception d'une commande de saisie d'un cliché (*snapshot*). Cela peut donc s'avérer utile particulièrement lorsque le *snapshot* est déclenché à la détection d'un problème.

On peut également mentionner qu'il est possible d'envoyer la trace par le réseau, ce qui peut s'avérer pertinent dans divers cas comme lorsque le système tracé a peu de mémoire, ou qu'on désire observer le comportement de diverses machines à partir d'un même poste.

LTTng a été testé dans le cadre du traçage de systèmes temps réel avec l'application *npt* (Beamonte, 2013). Pour ce faire, les auteurs ont exécuté une boucle avec un point de trace en espace utilisateur afin de mesurer l'impact sur l'application. Cela a donné une latence maximale de 35 µs avec le traceur standard. Cependant, avec le mode read-timer qui remplace le tube de contrôle par de l'écoute active, on arrive avec une latence maximale de 7 µs, ce que les auteurs jugent suffisamment faible pour utiliser le traçage dans la plupart des systèmes temps réel. Ces tests ont été effectués en isolant LTTng et les tâches temps réel sur des coeurs différents. On peut également mentionner que l'intervalle pour l'écoute du mode read-timer est paramétrisable. Avec un temps trop petit, cela demande beaucoup de ressources, alors qu'avec un temps trop grand, il est possible de perdre des événements qu'on ne perdrait pas autrement.

2.2.2 FTrace

FTrace (Function Tracer) est un ensemble d'outils de traçage inclus dans le noyau Linux depuis la version 2.6.27 (voir Rostedt, 2008). Il utilise lui aussi l'instrumentation statique présente dans le noyau, mais il permet également d'ajouter des points de trace de manière dynamique avec kprobes. Comme son nom l'indique, il s'agissait à la base d'un traceur de fonction. Cependant, ses fonctionnalités ont été étendues pour inclure la latence entre l'activation et la désactivation des interruptions, la latence entre l'activation et la désactivation de la préemption, le temps maximal pour être ordonnancé suite à un réveil et le temps maximal pour être ordonnancé suite à un réveil pour les tâches temps réel seulement.

Un des avantages de FTrace est qu'il existe de nombreux filtres qui permettent de sélectionner ce qu'on désire tracer. On peut entre autres filtrer par événements, par fonction, par module, par identifiant du processus ou par processeur. De plus, la trace peut être vue en temps réel, ou il est possible de la consulter a posteriori. Celle-ci utilise le système de fichiers debugfs. Quant au format, la trace peut être enregistrée en format texte pour être directement lisible, ou en format brut pour faciliter le scriptage. FTrace présente de bonnes performances. On

peut cependant souligner qu'il s'agit principalement d'un traceur pour le noyau et qu'il faut utiliser de l'instrumentation dynamique pour tracer en espace utilisateur, ce qui est moins performant. De plus, il ne peut y avoir qu'une session de traçage à la fois.

2.2.3 SystemTap

SystemTap permet de tracer en espace noyau tout comme en espace utilisateur (voir SystemTap Editor Group, 2015). Pour cela, il utilise de l'instrumentation dynamique, soit *uprobes* et *kprobes*. La particularité de SystemTap est qu'il fournit un langage de script permettant beaucoup de flexibilité. Les scripts sont ensuite compilés dans un module et chargés dans le noyau pour le traçage noyau. Pour le traçage en espace utilisateur, on utilise plutôt un *user-space mutator* DynInst (Paradyn Project, 2015) pour arriver au même résultat. L'outil en ligne de commande *stap* permet d'accomplir cela facilement. *SystemTap GUI* peut quant à lui être utilisé pour éditer les scripts. Une autre particularité intéressante est que SystemTap possède un mode guru permettant d'insérer du code en C. On peut également ajouter qu'il est possible d'utiliser les points de trace définis par DTrace et qu'on peut aussi tracer à distance.

Au niveau des performances, on peut toutefois noter qu'il s'avère plus lent que les solutions qui utilisent un traçage statique. En effet, on note dans ce banc de tests (Desnoyers and Desfossez, 2011) des performances de plus de 20 fois moins bonnes pour le traçage utilisateur que LTTng pour un fil d'exécution. Cela se dégrade avec le nombre de fils d'exécution pour atteindre plus de 200 fois plus lent avec 8 fils d'exécution. On peut donc noter que ce traceur présente une grande flexibilité, mais qu'il n'est pas adapté pour la collecte d'un important nombre d'événements.

2.2.4 Perf

Perf est un outil d'analyse de performance (voir Gregg, 2015). Il est intégré dans le noyau Linux depuis 2.6.31. En plus de supporter l'instrumentation statique du noyau, il supporte aussi l'instrumentation dynamique et les compteurs de performance sur les principales architectures modernes. De plus, *perf* présente un petit surcoût et ne nécessite pas de *deamons*.

Les compteurs de performance matériels couvrent les fautes de cache, les instructions exécutées et les branchements. Les compteurs de performance logiciels incluent les fautes de page mineures et majeures, les changements de contexte et les migrations de CPU. Il est également possible d'obtenir les piles d'appels avec DWARF.

Perf permet aussi d'échantillonner le temps passé dans chaque fonction en spécifiant la fréquence de la prise de données pour avoir une approximation plus ou moins précise. Il est même possible de descendre d'un niveau et d'avoir le pourcentage par instruction avec *perf annotate*. Afin de limiter la quantité de données recueillies, on peut également filtrer selon le programme ou le processeur.

Cela dit, ce n'est pas vraiment efficace pour trouver des problèmes qui ne surviennent pas régulièrement, puisque l'échantillonnage ne montre pas s'il y a des exécutions durant lesquelles des fonctions prennent beaucoup plus de temps. Il est possible d'utiliser *perf-top* pour voir en temps réel, ou depuis le dernier rafraîchissement, où le temps est passé, ce qui peut s'avérer utile dans certains cas, mais cela ne remplace pas une analyse approfondie. Or, il n'y a pas vraiment d'analyse complexe disponible avec l'outil *perf*. La trace produite peut toutefois être convertie en format CTF afin d'utiliser un outil qui supporte ce format pour en effectuer l'analyse.

2.2.5 DTrace

DTrace est un traceur développé par Sun qui servait à la base pour tracer le système d'exploitation Solaris (voir Zannoni and Van Hees, 2012). Il est cependant maintenant disponible comme module noyau pour Linux. Il est possible de créer des scripts en langage D pour indiquer ce qu'on désire tracer. Le langage D comporte des éléments du C, mais ajoute des fonctionnalités propres au traçage. Cela permet donc une grande flexibilité.

Bien que ce soit par défaut de l'instrumentation dynamique, DTrace supporte également l'instrumentation statique. De plus, il permet de tracer l'espace utilisateur, tout comme l'espace noyau. On peut toutefois noter que le traçage en espace utilisateur avec DTrace est plus lent qu'avec LTTng-UST, soit environ 10 fois selon un banc de test (Brosseau, 2011).

2.2.6 KTap

KTap est similaire à SystemTap mais, contrairement à celui-ci, il ne dépend pas du compilateur GCC (voir Corbet, 2013). Il permet le traçage dynamique en utilisant des scripts qui sont basés sur du code binaire. Cela ne demande donc pas de recompiler un module noyau pour chaque script. Cela peut s'avérer utile pour des logiciels embarqués puisque ceux-ci ne contiennent pas nécessairement l'infrastructure pour compiler rapidement un module. On peut toutefois noter que cet outil en est encore à ses débuts et qu'il manque pour le moment plusieurs fonctionnalités présentes dans les alternatives.

2.2.7 Sysdig

Sysdig est une autre solution de traçage pour Linux (voir Sysdig Cloud, 2015). Il permet d'utiliser des scripts pour analyser les traces produites et d'en écrire des nouveaux. Un accent semble être mis sur l'infonuagique et la réseautique. Il existe par exemple plusieurs scripts pour afficher de l'information sur la virtualisation ou sur les connexions réseau. On peut toutefois mentionner que cet outil est nouveau et ne présente pas encore toutes les fonctionnalités disponibles avec les autres traceurs. Par exemple, la liste d'événements tracés est encore réduite, et il n'est pas possible de tracer en espace utilisateur ou d'ajouter des points de trace facilement.

2.2.8 Paradyn Trace Tool

Paradyn est un outil pour analyser les performances de systèmes à grande échelle (voir Paradyn Project, 2015). Il a été testé avec des systèmes comprenant des centaines de milliers de fils d'exécution, s'exécutant sur des dizaines de processeurs. Pour éviter de se retrouver avec une quantité énorme de données, Paradyn utilise de l'instrumentation dynamique uniquement sur les parties du programme qui semblent problématiques. Il analyse donc les données en temps réel pour ajouter et enlever des points de trace. De plus, ces données sont principalement prises par échantillonnage, plutôt qu'à chaque exécution d'un point de trace, pour encore une fois réduire le volume d'information récoltée. Cela peut être une bonne approche dans certains cas, mais la possibilité d'ajouter des points de trace de manière dynamique diminue la prédictibilité du système dans le cas de systèmes temps réel.

2.3 Problèmes dans les systèmes temps réel

Afin de développer des outils pertinents pour analyser les systèmes temps réel, il convient d'abord d'avoir une idée des principaux problèmes rencontrés dans ces systèmes.

2.3.1 Inversion de priorité

L'inversion de priorité est un problème bien étudié qu'il est important de chercher à éviter (Helmy and Jafri, 2006). Cela consiste en un fil d'exécution prioritaire qui se retrouve à attendre indirectement sur un fil d'exécution de moindre priorité. En effet, lorsqu'un fil d'exécution prioritaire est bloqué par un fil d'exécution de faible priorité, celui-ci peut se faire préempter par un fil d'exécution de priorité intermédiaire, si aucun mécanisme n'est mis en place. Deux solutions principales peuvent être mises en place pour contrer ce pro-

blème. Premièrement, le protocole de priorité plafond (priority ceiling protocol) consiste à augmenter la priorité d'un fil d'exécution lorsqu'il entre dans une section critique à la valeur de celle du fil d'exécution de plus haute priorité pouvant accéder à cette ressource. Cela présente comme inconvénient d'augmenter inutilement la priorité de certains fils d'exécution, et comme avantage de limiter les changements de contexte. Ceci est donc principalement utilisé quand la tâche de haute priorité est celle qui utilise le plus souvent la ressource. La deuxième méthode est l'héritage de priorité (priority inheritance protocol). Avec celle-ci, les fils d'exécution bloquant un fil d'exécution de plus grande priorité voient leur priorité augmenter temporairement à la valeur de celle du fil d'exécution bloqué. Cela cause des changements de contexte additionnels, mais les fils d'exécution ne s'exécutent pas à une haute priorité sans nécessité. C'est donc généralement la méthode privilégiée quand le fil d'exécution de haute priorité n'est pas celui qui accède à la ressource le plus souvent.

2.3.2 Calcul du pire temps d'exécution

Comme il a déjà été mentionné précédemment, la connaissance du pire temps d'exécution (Worst Case Execution Time, WCET) d'une tâche s'avère très utile dans le but de valider un système temps réel et de spécifier des contraintes propres à chaque tâche.

On cherche donc à mesurer les pires temps d'exécution pour chaque tâche. Pour ce faire, il est nécessaire de calculer les pires temps d'exécution pour chaque fil d'exécution, les temps maximums où les tâches de plus basse priorité peuvent être préemptées par des tâches de plus haute priorité et les temps maximums où les tâches de plus haute priorité peuvent être bloquées par des tâches de plus basse priorité qui accèdent à des ressources communes (Santos and Wellings, 2010).

Plusieurs méthodes existent pour calculer les WCETs. Il s'agit généralement d'analyses statiques du code, d'analyses basées sur les temps d'exécution de chaque partie du code ou d'une combinaison des deux.

Or, il peut arriver que ce calcul ne soit pas valide. Plusieurs facteurs ont été identifiés par Burns and Wellings (2001). Au niveau des outils, il se peut que l'analyse des WCETs comporte une erreur et que les temps ne soient pas valides ou encore que l'analyse des ordonnancements comporte une erreur. Au niveau des spécifications, il se peut que les temps de blocage aient été sous-estimés, que les hypothèses concernant l'ordonnancement ne soient pas valides, ou encore que les spécifications du système soient trop optimistes. Par exemple, une tâche sporadique pourrait se produire plus fréquemment que prévu initialement.

Cela donne lieu au non-respect des échéances dans le programme. Dans dos Santos and Wellings (2008), on identifie cinq chaînes d'événements menant à une échéance manquée. On pourrait les résumer comme étant :

- un mauvais calcul du WCET qui mène directement à une échéance manquée,
- un mauvais calcul du WCET qui mène à une échéance manquée d'une autre tâche de plus basse priorité par propagation,
- une mauvaise estimation de la fréquence d'une tâche qui mène directement à une échéance manquée,
- une mauvaise estimation de la fréquence d'une tâche qui mène à une échéance manquée d'une autre tâche de plus basse priorité par propagation et
- une erreur sur une tâche de basse priorité qui mène à une échéance manquée d'une tâche de plus haute priorité qui partage une ressource.

Pour pouvoir prévenir correctement ces différents scénarios, cela demande d'identifier les différentes situations indésirables qui mènent à des problèmes, et pas seulement les échéances manquées. On peut mentionner ici les WCET manqués, les événements sporadiques qui arrivent plus souvent que prévu et les ressources partagées utilisées plus longtemps que prévu.

Dans l'article Perathoner et al. (2007), on effectue une comparaison des différentes méthodes d'analyse du code pour trouver le WCET. Plus précisément, les auteurs testent plusieurs outils basés sur les approches d'ordonnancement holistique, soit l'utilisation de différentes méthodes statiques, les approches compositionnelles (calculer pour de petits segments et remonter pour trouver le WCET global), et les approches utilisant des automates temporisés. Les auteurs en arrivent à la conclusion qu'aucune approche n'est meilleure dans tous les cas et qu'on se retrouve souvent avec un résultat beaucoup trop pessimiste. Devant ce constat, des analyses de l'exécution du système en fonctionnement sont requises pour tester que l'implémentation réelle du système correspond à son modèle idéal.

2.3.3 Ordonnancement

Les différentes politiques d'ordonnancement disponibles ont été présentées précédemment. Il convient de mentionner que chacune présente des avantages et des inconvénients. Ainsi, on peut mentionner qu'avec SCHED_DEADLINE, il n'est pas possible de spécifier des tâches prioritaires par rapport à d'autres, puisque les tâches se font ordonnancer selon la proximité de leur échéance. En cas de dysfonctionnement du système, il n'est donc pas possible de prioriser une tâche. À l'inverse, avec les politiques SCHED_FIFO et SCHED_RR, une tâche de basse priorité peut manquer son échéance même s'il existait un ordonnancement valide du

système. De plus, il est possible d'obtenir une grande quantité de changements de contexte qui auraient pu être évités sans mener à des échéances manquées.

Il existe des solutions alternatives pour l'ordonnancement des tâches dont plusieurs sont présentées dans l'article Buttazzo et al. (2013). En effet, si on ne limite pas la préemption, bien que cela permette de s'assurer que les tâches de plus haute priorité s'exécutent dans les temps, cela peut augmenter le surcoût et rendre plus difficile l'analyse du WCET.

D'abord, on propose le *Preemption Thresholds Scheduling* (PTS) où chaque tâche a une priorité pour s'exécuter et une priorité pour se faire préempter. Ainsi, si deux tâches sont en attente d'être exécutées, la tâche prioritaire est sélectionnée, mais si la tâche la moins prioritaire s'exécute déjà, une priorité plus grande que son seuil est nécessaire pour la préempter.

Ensuite, avec le Deferred Preemptions Scheduling (DPS) , chaque tâche a un temps alloué durant lequel elle ne peut pas se faire préempter. On analyse dans l'article Bril et al. (2009) comment on peut calculer le WCET pour cette méthode. Or, les auteurs concluent qu'il s'avère très difficile de prédire le temps d'exécution, excepté pour la tâche avec la plus basse priorité, puisque celle-ci ne peut pas être bloquée, mais seulement préemptée.

Finalement, le Fixed Preemption Points (FPP) consiste à avoir des sections de code où on ne peut pas se faire préempter. Il s'agit de la méthode la plus prédictible, mais également la plus complexe à implémenter, puisque cela demande d'ajouter des instructions dans le code.

2.3.4 Autres défis

En plus des problèmes courants, il peut être pertinent de mentionner les principaux défis dans les systèmes temps réel. Outre de contrôler les politiques d'ordonnancement correctement, on mentionne dans l'article de Niz et al. (2008) que ce serait de réserver des lignes de cache de manière pertinente et de partitionner correctement la cache pour optimiser le temps d'exécution. Il serait donc intéressant de voir comment on peut instrumenter ces différents cas afin d'offrir des outils efficaces aux développeurs.

2.4 Outils pour l'analyse de trace

Vu la quantité de données collectées qui peut s'avérer énorme, il est souvent nécessaire d'utiliser un outil pour analyser les événements générés par le traçage d'un système. Or, il existe un bon nombre d'outils de visualisation de traces d'exécution. L'accent sera donc mis sur ceux qui présentent des caractéristiques uniques ou qui sont spécifiques aux systèmes temps réel.

2.4.1 Trace Compass

Trace Compass est un outil de visualisation de trace qui supporte de multiples formats, dont le format CTF utilisé par LTTng (voir Trace Compass, 2015). Il comporte plusieurs analyses de base présentées dans différentes vues. On retrouve entre autres une vue de l'usage des processeurs, une vue des ressources, une vue de la pile d'appels, une vue détaillée des événements et une vue des statistiques. Ensuite, il permet de définir soi-même des états et des transitions en XML ou en java et d'afficher une vue correspondante. Il s'agit d'une vue des états pour les différents fils d'exécution en fonction du temps qui est présentée horizontalement. Par défaut, une vue des différents états d'un fil d'exécution dans le noyau Linux est présentée.

Ces états sont conservés dans un arbre avec les attributs correspondants (Montplaisir et al., 2013). Cela permet de naviguer rapidement dans la trace une fois l'arbre généré. En effet, l'état courant pourra être retrouvé avec une seule requête dans l'arbre puisque les intervalles sont conservés de manière à ce que l'intervalle correspondant à chaque noeud soit compris dans l'intervalle de son parent et que, pour un même niveau de l'arbre, les intervalles soient disjoints.

Ensuite, Trace Compass présente plusieurs vues expérimentales intéressantes. Parmi celles-ci, on pourrait noter une vue du chemin critique d'un fil d'exécution qui peut être défini comme les segments d'exécution dont la réduction de la durée réduit le temps d'exécution global. Cette vue permet de suivre le chemin critique d'un fil d'exécution même sur plusieurs machines différentes en synchronisant les traces. Il existe cependant des limitations concernant les algorithmes utilisés. En effet, si on calcule le chemin critique comme étant le plus long chemin dans le graphique de dépendance, en comptant les blocages comme valant 0, il se peut qu'on tombe sur une tâche qui n'était pas vraiment reliée, mais détenait un verrou qui nous a bloqués à un moment comme on peut le voir à la Figure 2.1. La totalité de l'exécution de cette tâche pourrait donc se retrouver dans le chemin critique. Une des solutions proposées est plutôt de calculer le chemin critique uniquement pour les périodes de blocages. On suit donc les événements de notification de fin de blocage, *sched_wakeup*, et on remonte jusqu'à ce que la période de blocage de la tâche principale soit couverte. À l'inverse de la première technique, il se peut ici qu'il manque de l'information sur ce qui est arrivé avant la période de blocage, mais qui devrait normalement faire partie du chemin critique. On pourrait également ajouter qu'il manque des informations pertinentes pour les systèmes temps réel. Ainsi, quand un fil d'exécution sur le chemin critique est préempté par d'autres qui ont une plus grande priorité, ce n'est pas possible de savoir facilement ce qui se passe comme on peut le

voir à la Figure 2.2. Il convient donc d'ajouter de l'information pertinente pour les systèmes temps réel, comme les fils d'exécution s'exécutant lors de la préemption et leur priorité.

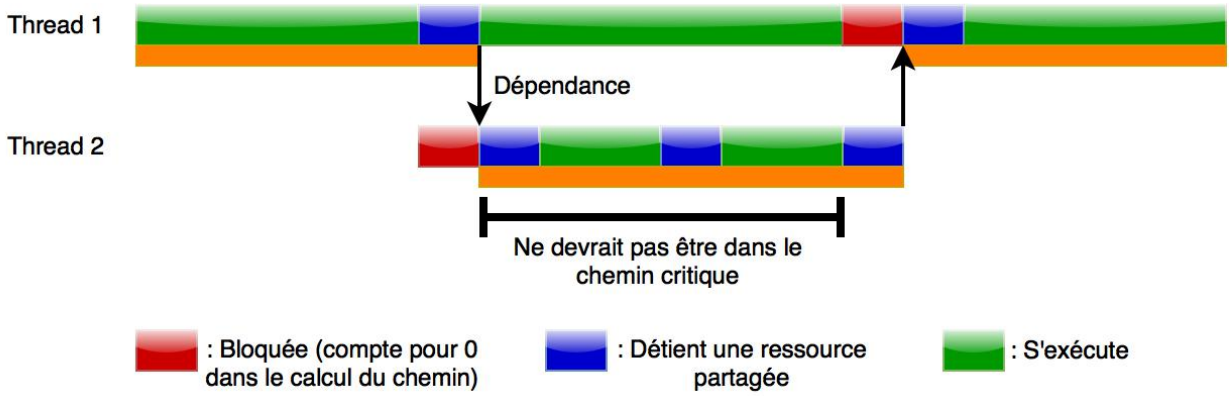

Figure 2.1 Deux fils d'exécution selon le temps avec le chemin critique du premier fil d'exécution en orange montrant comment des segments d'exécution du deuxième fil peuvent se retrouver dans le chemin critique de manière non désirée.

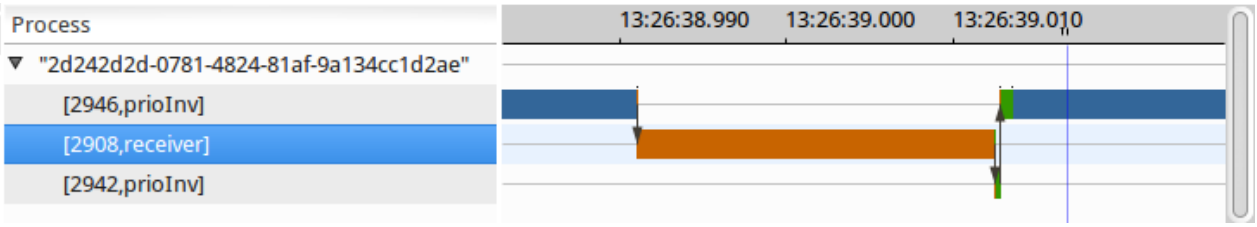

Figure 2.2 Capture d'écran de la vue du chemin critique dans Trace Compass où il y a une préemption en orange et où on remarque qu'il n'y a pas d'information sur ce qui se passe pendant celle-ci.

2.4.2 KernelShark

KernelShark utilise les traces générées par *trace-cmd* qui fait partie du traceur FTrace (Rostedt, 2011). Les différentes caractéristiques de ce traceur ont déjà été mentionnées précédemment. Pour ce qui est de l'outil de visualisation, deux vues s'offrent aux utilisateurs soit une vue des tâches et des CPU en fonction du temps et une vue des événements. Différents filtres sont également disponibles. Il s'agit d'un bon complément au traceur, mais aucune analyse complexe n'est présente.

2.4.3 Zinsight

ZInsight est un outil de traçage pour les *mainframes* d'IBM (IBM Research, 2015). Comme ces machines ont plusieurs dizaines de processeurs, l'outil est adapté pour cette situation. La *flow view* présente les événements avec une échelle de temps verticale. Ceux-ci sont séparés selon le module duquel ils proviennent ou selon le processeur. Une fonctionnalité intéressante est de montrer les migrations de fils d'exécution seulement ou de suivre un fil d'exécution.

Ensuite, la *event type view* permet de voir le nombre d'événements selon le type. Elle fournit également des statistiques pour les événements qui peuvent former des paires. On aura donc, par exemple, le temps d'exécution entre les événements représentant le début et ceux représentant la fin de l'exécution d'une fonction.

Finalement, la *sequence view* permet de voir toutes les séquences d'événements survenus entre deux événements sélectionnés, avec des chiffres sur les traits représentants le nombre de fois où ce chemin a été pris. De plus, il est possible de voir des petites barres montrant le temps pour traquer les plus longs délais comme on peut le voir à la Figure 2.3. Le principe des barres est cependant fort peu efficace lorsqu'il y a plusieurs dizaines d'exécutions et cela prendrait plutôt un outil de comparaison.

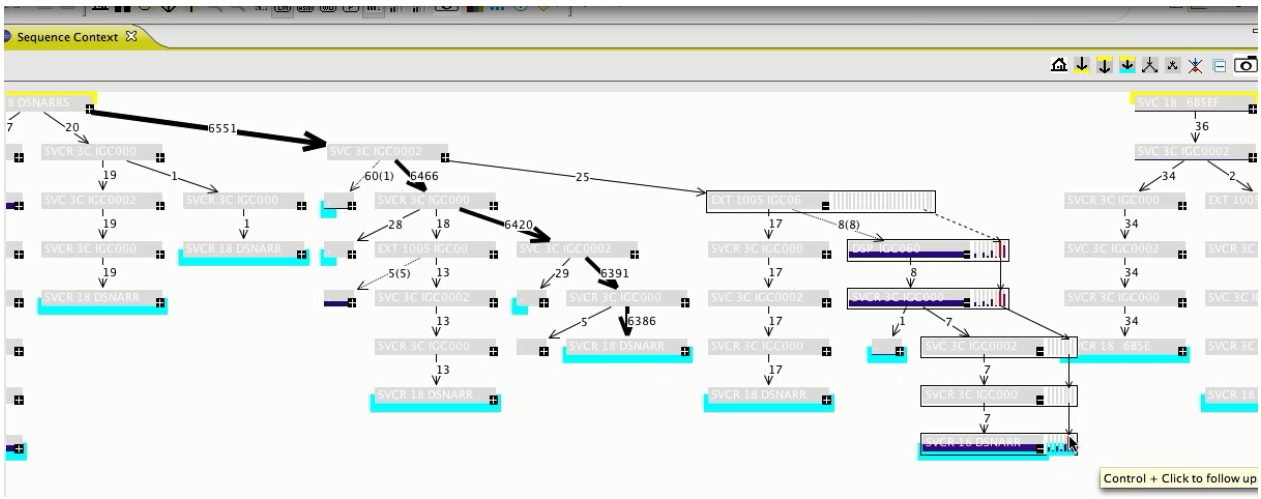

Figure 2.3 Vue de séquences dans Zinsight montrant des barres pour indiquer les plus long délais.

2.4.4 Vampir

L'outil *Vampir* est axé sur l'analyse de traces provenant d'applications de haute performance (GWT-TUD GmbH, 2015). Il supporte les traces dans le format *Open Trace Format* et celles

produites par le traceur *VampirTrace*. Celui-ci permet de tracer un programme et inclut une librairie qui permet de tracer les appels à la librairie *Message Passing Interface* (MPI).

La vue de base est la *Master Timeline* qui présente, pour tous les processus qui se sont exécutés, différentes informations concernant les fonctions exécutées et la synchronisation. Plus précisément, on voit les échanges faits entre les divers processus et dans quelle fonction se trouve le programme selon une ligne du temps horizontale.

Plutôt que de montrer les processus à l'horizontale, la vue *Process Timeline* présente plutôt la pile d'appels d'un processus et les différents événements selon le niveau d'appel auquel ils se sont produits. Cela peut s'avérer utile pour repérer dans quelle fonction se trouve un problème. L'arbre des appels avec le temps passé dans chaque fonction peut également être affiché.

La vue *Counter Data Timeline* présente quant à elle l'évolution d'une valeur en fonction du temps. Elle permet de définir l'échelle de temps en question. Toutefois, le plus intéressant est qu'il est possible de définir graphiquement les métriques à afficher. Il est également possible de superposer ces métriques à la vue principale avec une certaine opacité correspondant à la valeur.

2.5 Outils pour l'analyse de système temps réel

2.5.1 Visual Analysis of Distributed Real-time systems (VADR)

Bien que cet outil ne soit plus en développement, il est intéressant de mentionner les idées présentes (Harmon and Klefstad, 2006). En effet, les chercheurs ont développé des vues en 3D permettant de suivre les différents liens entre les tâches. Celles-ci sont représentées par des sphères. Plus une tâche passe près de ne pas respecter son échéance, plus la sphère correspondante s'approche du rouge. De même, les liens entre les tâches sont de grosseur variable selon l'importance. Bien que cette approche soit originale et permette d'avoir une vue d'ensemble, cela n'est pas facile d'utilisation et une approche en deux dimensions est probablement à privilégier.

2.5.2 TuningFork

TunningFork est un outil d'analyse de traces pour les systèmes temps réel développé par IBM et basé sur Eclipse (Bacon et al., 2006). Il supporte différents formats de trace de même qu'une combinaison de différentes traces pour pouvoir utiliser des traces de différents niveaux. Plusieurs fonctions sont disponibles comme de jouer la trace, de voir l'évolution

des différentes vues, de revenir dans le temps et d'arrêter le visionnement de la trace. Cela permet de voir l'état du système à différents moments. De plus, certaines vues sont plutôt axées sur une analyse d'une portion de la trace. L'outil a été développé pour offrir de bonnes performances graphiques. Il était important de pouvoir naviguer facilement dans la trace et de manière fluide. On peut noter que plusieurs filtres ont été ajoutés pour pouvoir afficher uniquement l'information pertinente.

Au niveau des vues offertes, on retrouve une vue d'histogramme où il est possible de séparer les barres en différentes couleurs pour, par exemple, montrer le temps d'exécution d'une tâche dans la période donnée avec la barre et, à l'intérieur de celle-ci, la proportion de temps passé dans chaque fonction avec les couleurs. Il est également possible d'afficher des histogrammes superposés avec différents intervalles pour voir facilement si une métrique se stabilise à plus grande échelle.

Ensuite, une vue présentant la mémoire utilisée et les tâches s'exécutant permet de voir visuellement quelle tâche s'exécutait lors de l'allocation de mémoire.

Finalement, on présente également une vue en oscilloscope qui permet de comparer des exécutions et qui est donc particulièrement utile pour le temps réel. Cette vue présente la trace sur plusieurs lignes. Les exécutions peuvent donc être alignées afin de déterminer visuellement celles qui ont pris le plus de temps. Il est également possible de superposer les lignes supplémentaires aux lignes déjà présentes pour voir facilement si c'est stable au fil du temps. Il aurait été plus pratique de pouvoir trier ces exécutions en terme de temps d'exécution puisque, de la manière présentée, il est difficile de comparer des exécutions arrivant à plusieurs exécutions d'intervalle. On peut également souligner que cet outil ne semble plus être en développement.

2.5.3 Tracealyser

Tracealyser est un outil développé par Percepio (Percepio, 2015). Il supporte des traces provenant de divers systèmes d'exploitation temps réel, dont VxWorks et FreeRTOS. De plus, il supporte les traces provenant de LTTng pour tracer sur Linux. Puisque plus de 20 vues sont présentes, seulement les plus pertinentes pour l'analyse de systèmes temps réel seront présentées.

La vue principale est basée sur une échelle de temps verticale. Les différentes colonnes représentent les tâches du système. Il est également possible d'afficher les événements sur cette vue avec une multitude de filtres permettant entre autres d'afficher les événements du noyau, de l'espace utilisateur ou d'un certain type. Il s'agit d'une façon intéressante de présenter

les événements, puisqu'il est facile de voir les changements apportés aux états par chaque événement comme on peut le voir à la Figure 2.4. On peut toutefois souligner que c'est difficilement utilisable si on a beaucoup de tâches qui s'exécutent en même temps, puisqu'il faudra naviguer horizontalement. C'est donc conçu principalement pour des systèmes qui ont peu de tâches. Ensuite, lorsque la trace est dézoomée, il devient difficile d'utiliser les options de visualisation des événements. De plus, les filtres pour les événements sont sous forme de case à cocher en arbre, ce qui manque de flexibilité.

Figure 2.4 Capture d'écran de la vue des événements dans Tracealyzer montrant également les filtres à droite.

Puisque la plupart des autres vues sont avec une ligne de temps horizontale, une vue de l'ordonnancement des tâches à l'horizontale est disponible. On ne retrouve toutefois pas les événements dans cette vue, mais celle-ci présente l'avantage de se synchroniser avec les autres vues horizontales. Parmi ces dernières, on retrouve une vue de l'utilisation de la mémoire, une vue de l'utilisation des processeurs et une vue des signaux.

La vue des CPU est intéressante. En effet, comme dans l'outil TuningFork, on présente un histogramme de la charge dans un intervalle avec les portions correspondant aux différentes tâches de différentes couleurs.

Ensuite, le *Communication flow* présente un graphique des dépendances de chaque tâche. Pour un petit système, cela peut être intéressant pour comprendre le fonctionnement, mais cela devient rapidement difficile à lire pour un système complexe.

Parmi les autres vues intéressantes, on retrouve également une vue qui montre le temps de réponse (entre le sched_wakeup et le sched_switch) ou le temps d'exécution (entre les sched_switch) pour une tâche en fonction du temps dans la trace. Cela permet donc d'aller voir dans la trace aux endroits qui semblent problématiques. Il aurait cependant été intéressant de pouvoir personnaliser davantage ce que constitue l'exécution d'une tâche.

Une autre vue intéressante est la vue de différents objets soient les mutex, les sémaphores et les descripteurs de fichier. On présente pour chacun les différents événements reliés. Dans le cas des mutex et des sémaphores, cela demande une instrumentation supplémentaire pour la librairie *pthread*, qui est disponible avec LTTng, si on désire avoir toute l'information. On peut également mentionner que cela fait simplement lister les événements, alors qu'il aurait été intéressant de pouvoir voir cela de manière graphique et de pouvoir comparer des états.

Une autre vue intéressante montre un histogramme du nombre d'exécutions qui se sont produites, selon leur temps d'exécution séparé en différents intervalles. C'est bien pour donner une idée de la répartition du temps que prennent les tâches. C'est cependant plus difficile de synchroniser cela avec une vue en fonction du temps.

2.5.4 RapiTime et RapiTask

Il s'agit de deux outils développés par *Rapita Systems* et qui utilisent le traceur fourni par la compagnie (Rapita Systems, 2015). *RapiTime* est utilisé pour calculer le temps d'exécution dans le pire cas. Pour ce faire, l'outil collecte des métriques concernant la couverture du code et le temps d'exécution de chaque segment. Il utilise ensuite ces métriques pour trouver les endroits qu'il conviendrait d'optimiser afin de réduire le *WCET*. Les vues montrent donc les différentes données obtenues pour les différentes sections du code.

RapiTask affiche plutôt des vues correspondant aux activités d'ordonnancement des tâches. Il est également possible de corréler les analyses des deux outils afin de fournir une meilleure vue d'ensemble et de tenter d'identifier les fonctions qui s'exécutaient lors de problèmes d'ordonnancement. Il aurait toutefois été intéressant d'offrir des méthodes pour repérer ces problèmes, puisque cela peut s'avérer laborieux de manière visuelle lorsque la trace est de grande taille.

2.5.5 WindRiver WindView

WindView est un outil d'analyse de trace pour VxWorks (Wind River Systems, 2015). Il n'y a qu'une seule vue qui montre l'état des différentes tâches selon le temps. L'outil présente tout de même de nombreux filtres dont un pour la priorité. Le filtrage par priorité peut s'avérer intéressant lorsqu'on souhaite sélectionner seulement les tâches plus prioritaires qu'un certain niveau. Pour le reste, l'outil est assez limité.

2.5.6 QNX System Profiler

Cet outil d'analyse de trace sert pour le système d'exploitation QNX (QNX Software Systems, 2014). La vue principale est selon le temps et présente quatre options soit la vue des événements par processeur, la vue de l'usage des processeurs, la vue selon les états et la vue des événements. Ceux-ci sont également disponibles dans une vue secondaire. Il est aussi possible de filtrer selon les fils d'exécution. Bien qu'il soit assez simple, cet outil a quand même la particularité de présenter une vue des statistiques où on retrouve les temps totaux passés dans chaque état et en combien de fois.

2.6 Découverte de patrons

La découverte de patrons peut s'avérer fort utile pour repérer des tâches dans une trace d'exécution. En effet, les développeurs ne sont pas nécessairement en mesure d'identifier correctement les événements correspondant à une exécution et il peut donc s'avérer pertinent de leur faire des suggestions. Par exemple, la suite de type d'événements *sched_switch*, *syscall_exit_clock_nanosleep*, *syscall_entry_clock_nanosleep* peut être considérée comme un patron permettant de repérer les tâches délimitées par un *sleep* haute résolution. Ainsi, chaque fois que des événements correspondants sont rencontrés dans la trace dans l'ordre, mais pas nécessairement consécutifs, l'intervalle de temps entre le premier de ces événements et le dernier est considéré comme une exécution.

Afin de bien comprendre les différents algorithmes présentés, il convient de présenter quelques définitions :

- Un événement est composé d'un type d'événement et d'un temps d'exécution.
- Une séquence est composée de plusieurs événements ordonnés.
- Une sous-séquence est un sous-groupe d'événements appartenant à une même séquence.
- Un épisode est un ensemble de types d'événements qui doivent arriver dans l'ordre.
- Un sous-épisode est un sous-groupe de types d'événements appartenant à un même épisode.
- Une occurrence est une séquence qui correspond à un épisode.

– Un patron est un épisode auquel on peut rajouter d'autres contraintes comme des contraintes de temps.

Un exemple est présenté à la Figure 4.1.

On retrouve plusieurs algorithmes pour effectuer de la recherche de patrons et qui utilisent différentes approches. Certains sont basés sur le principe de transaction, c'est-à-dire qu'ils considèrent que certains groupes d'éléments se sont produits en même temps. Cela s'applique mal au traçage d'un fil d'exécution puisque les événements se produisent de manière séquentielle, mais pourrait correspondre à un modèle d'exécution en parallèle. Dans tous les cas, il convient de présenter les approches de base des algorithmes qui fonctionnent par transaction, puisque les mêmes concepts sont utilisés dans les algorithmes séquentiels.

2.6.1 Apriori

Tout d'abord, l'algorithme *Apriori* (Borgelt and Kruse, 2002) utilise le fait que, pour qu'un groupe d'éléments soient fréquents, c'est-à-dire présents dans un grand nombre de transactions, tous les sous-groupes d'éléments le constituant doivent également être fréquents. On calcule donc les éléments simples qui sont fréquents, puis les groupes de longueur 2 dont les éléments qui les constituent sont fréquents et ainsi de suite jusqu'à ce qu'on identifie des patrons de longueur suffisante. Un grand nombre d'algorithmes développés subséquemment partent de ce principe et présentent des améliorations pour réduire le nombre de candidats ou pour accélérer l'identification des groupes d'éléments fréquents.

2.6.2 WinEpi

Pour les traces d'exécution, il convient plutôt d'utiliser des algorithmes basés sur la découverte de séquences d'événements. Pour ce faire, l'algorithme WinEpi (Mannila et al., 1995) utilise une fenêtre glissante. L'algorithme prend en entrée une taille de fenêtre et un incrément pour le pas de déplacement de la fenêtre. Pour chaque fenêtre obtenue, une approche semblable à l'algorithme Apriori est utilisée pour repérer les séquences les plus fréquentes, en calculant le nombre de fenêtres dans lesquelles la séquence apparaît au moins une fois. Cette approche comporte quelques inconvénients. Il convient d'abord de choisir une taille de fenêtre appropriée. Si la taille est trop grande, on risque d'avoir un grand nombre de patrons qui ne sont pas significatifs. À l'inverse, si la taille est trop petite, on risque de manquer certains patrons avec une taille plus grande.

2.6.3 MinEpi

L'algorithme MinEpi (Mannila and Toivonen, 1996) tente de contourner ce problème en utilisant plutôt les occurrences minimales. Une occurrence est minimale lorsque dans aucune de ses sous-séquences on ne retrouve l'épisode qui lui correspond. Ici aussi, un seuil de fréquences pour retenir les candidats est utilisé. Ainsi, on note d'abord tous les éléments qui se produisent fréquemment. Puis, en utilisant une approche similaire à l'algorithme Apriori, on sélectionne les occurrences minimales composées de deux éléments et ainsi de suite. On peut également mentionner que l'algorithme EpiBF (Casas-Garriga, 2003) ajoute une distance maximale entre les événements, ce qui permet de réduire le nombre de candidats. L'algorithme MinEpi enlève la contrainte de la fenêtre, mais cela demande beaucoup plus de mémoire puisqu'on ne réduit pas la quantité d'information conservée en calculant les données par fenêtre. Ce n'est donc pas très efficace pour des traces d'exécution de grande taille.

2.6.4 NonEpi

Les trois algorithmes présentés précédemment retournent des épisodes qui peuvent être entrecoupés. L'algorithme NonEpi (Laxman et al., 2007) recherchent plutôt des épisodes qui ne s'entrecoupent pas. Cette technique est beaucoup plus efficace. En effet, il suffit de créer un automate par épisode pour compter les fréquences. Les épisodes sont ensuite combinés avec une approche comme celle de l'algorithme Apriori. Comme beaucoup moins d'occurrences sont ainsi trouvées, cela demande moins de ressources. On se retrouve cependant avec des séquences qui ne sont pas minimales, c'est-à-dire que le patron peut se retrouver dans une sous-séquence.

2.6.5 ManEpi

L'algorithme ManEpi (Zhu et al., 2010) retourne quant à lui des épisodes qui comportent plus d'occurrences qu'un certain seuil. Ces occurrences doivent être minimales et ne pas s'entrecouper. Plus précisément, il calcule un arbre de préfixe et cherche à compléter l'arbre pour trouver les épisodes fréquents. Les enfants de la racine de l'arbre sont les épisodes fréquents de longueur 1, c'est-à-dire les éléments qui sont présents plus de fois que le seuil. Ensuite, on effectue une fouille en profondeur de l'arbre. À chaque étape, on tente d'ajouter les différents éléments fréquents de longueur 1 aux épisodes déjà trouvés et on calcule le nombre d'occurrences. On continue ainsi jusqu'à ce que le nombre d'occurrences soit plus petit que le seuil. En effectuant une fouille en profondeur, l'espace mémoire utilisé peut être réduit par rapport aux algorithmes présentés précédemment, puisqu'on peut supprimer les

sections de l'arbre qui n'ont pas mené à des épisodes intéressants. Le temps d'exécution de l'algorithme est donc en fonction du nombre d'épisodes fréquents de taille 1 et augmente linéairement selon la taille des données. Il serait intéressant de voir comment cette technique pourrait être utilisée dans le cadre d'analyse de traces d'exécution. En effet, il conviendrait probablement d'utiliser uniquement le type des événements plutôt que le contenu complet, pour éviter d'avoir trop d'éléments différents, ou du moins de sélectionner quelques champs clés des événements pour limiter le temps de calcul.

2.6.6 Dans des traces d'exécution

Un essai (LaRosa et al., 2008) a déjà été effectué pour tenter de détecter automatiquement des patrons dans des traces d'exécution du noyau Linux. Pour y arriver, les auteurs ont modifié l'approche de l'algorithme WinEpi. Ils ont d'abord utilisé les ensembles d'items plutôt que les séquences afin de voir les relations entre les processus, peu importe l'ordonnancement. Cela signifie qu'un patron correspond à un groupe d'événements se produisant souvent dans la même fenêtre.

Ensuite, ils ont ajusté l'algorithme pour avoir moins de fenêtres à calculer et ont ajouté des filtres pour diminuer le nombre d'événements évalués. Cela donne un résultat moins précis, mais la mémoire utilisée et le temps de calcul sont ainsi grandement diminués.

Avec cela, ils ont réussi à repérer un patron démontrant que les événements associés à une application étaient corrélés aux fonctions d'affichage du système puisqu'ils se produisaient régulièrement ensemble. Cela constitue une preuve de concept qu'il est possible de trouver des patrons dans des traces noyau, bien que les auteurs aient utilisé un filtre approprié et qu'ils aient ajusté leur seuil de détection et la taille de leur fenêtre pour pouvoir repéré le patron qu'ils cherchaient. Une approche n'utilisant pas de fenêtre permettrait probablement d'augmenter les chances de trouver un patron sans aide de l'utilisateur, puisqu'il ne resterait que le seuil à ajuster. Il serait donc possible de progressivement réduire ce dernier de manière automatique jusqu'à ce qu'un patron intéressant soit trouvé.

2.7 Repérage de patrons

Il se peut également que les développeurs de systèmes temps réel soient au courant des événements de bas niveau générés par les tâches présentes dans leurs systèmes. Il peut donc s'avérer utile de détecter la présence de patrons représentant l'exécution d'une tâche. Pour ce faire, plusieurs algorithmes ont été développés afin d'améliorer l'approche naïve consistant à lire tous les éléments séquentiellement pour voir si le patron peut être trouvé.

2.7.1 Avec écarts

Dans l'article Wu et al. (2014), on présente une approche pour détecter les patrons qui permet un écart entre les événements et qui permet à plusieurs exécutions du patron de se superposer. Dans le cas des systèmes temps réel, cela peut s'avérer intéressant dans certaines situations où plusieurs exécutions peuvent survenir en même temps, bien que ça ne soit généralement pas le cas. Le principal inconvénient est ici que la complexité est en fonction de l'écart maximal autorisé entre les événements, multiplié par le nombre total d'événements. Si on ne met pas de restriction d'écart, on se retrouve donc avec une complexité de l'ordre du carré du nombre d'événements, ce qui s'avère généralement trop long à calculer avec une grande trace.

2.7.2 Avec un index

L'article Fatehpuria and Goyal (2014) présente un algorithme simple pour efficacement détecter la présence d'un patron. Il itère une fois sur tous les éléments et note dans une structure de données les différentes occurrences pour chaque élément. Ensuite, il utilise cette information pour repérer rapidement les patrons, en faisant des sauts aux éléments pertinents en utilisant les index. Cette approche permet également de traiter une seule fois les données et de pouvoir détecter par la suite rapidement plusieurs patrons. Toutefois, bien que cet algorithme puisse être utile dans le cas de chaînes de caractères, cela présente quelques problèmes dans le cas de traces d'exécution. En effet, cette approche utilise premièrement beaucoup de mémoire. Sa principale lacune réside toutefois dans son manque de flexibilité. En effet, il n'est pas possible d'utiliser l'ensemble du contenu des événements dans la définition des patrons, tout en voulant faire une seule phase de traitement pour plusieurs patrons différents qui ne soient pas connus au préalable. Il conviendrait donc de traiter uniquement le nom des événements et de comparer le reste du contenu de l'événement lors de la phase de détection. Cela demanderait toutefois des accès à la trace qui ne soit pas consécutifs. Il serait intéressant de comparer cette méthode avec celles où la détection se fait en lisant la trace.

2.8 Métriques pertinentes pour l'analyse des systèmes temps réel

Il est pertinent de repérer les différentes exécutions d'une tâche, mais il faut ensuite savoir quelles métriques peuvent valoir la peine d'être calculées et présentées aux développeurs.

2.8.1 Performance

Dans l'article Hillary (2005), on présente différentes métriques pertinentes pour s'assurer de la performance d'un système temps réel. On peut mentionner le temps de réponse aux événements, le calcul du temps d'exécution par tâche et par fonction, de même que le temps avant l'expiration des délais. On met également l'accent sur les enjeux liés à la mémoire. Il est ainsi recommandé de mettre les fonctions qui vont souvent ensemble près les unes des autres pour qu'elles soient chargées ensemble dans la cache. Il est également recommandé de faire attention à la fragmentation de la mémoire au niveau de la cache pour atteindre une meilleure performance.

2.8.2 Chemin critique

Dans l'article Brandner et al. (2014), on amène plutôt la notion de *criticality*. Il s'agit d'attribuer une pondération aux différents chemins plutôt que d'uniquement calculer le chemin critique. En effet, il se peut que deux chemins soient pratiquement aussi longs et que l'effort pour réduire le temps que prend le chemin le plus long ne diminue pas significativement le temps global puisqu'il ne sera rapidement plus le chemin critique de l'application. Il peut donc s'avérer pertinent de savoir quels segments de code sont les plus avantageux à optimiser pour améliorer les performances. Pour ce faire, les auteurs utilisent des analyses statiques du code. Il serait cependant intéressant d'utiliser une approche similaire dans le calcul du chemin critique utilisant les événements du noyau. On peut toutefois mentionner qu'on ne pourrait pas utiliser les événements *sched_wakeup* pour voir les relations entre les fils d'exécution qui ne causent pas de contention et que le calcul s'avérerait ainsi beaucoup plus complexe.

2.8.3 Qualité

Dans l'article Garcia-Martinez et al. (1996), on présente plutôt différentes métriques à calculer pour évaluer la qualité d'un système d'exploitation temps réel (RTOS). Ces métriques sont classées en trois catégories, soit la réponse aux événements extérieurs, la synchronisation entre les tâches, et le partage de ressources ainsi que le transfert de données entre les tâches. On va, par exemple, retrouver comme mesure le temps que prend l'incrémentation d'un sémaphore, le temps que prend une tâche pour être informée qu'un mutex a été libéré, le temps d'exécution d'une interruption, ou encore le temps que prend un message pour être envoyé et lu. On peut cependant ajouter que, bien que cela peut être utile pour comparer les différents RTOS entre eux, la plupart des mesures varient selon divers paramètres comme la

charge du système. Cela peut quand même donner une idée de métriques qu'il devrait être possible de mesurer en utilisant un outil de traçage.

2.9 Analyse de traces d'exécution de systèmes temps réel

Cela nous amène à différents projets ayant traité de l'analyse de traces d'exécution de systèmes temps réel ou de l'utilisation de patrons dans les traces d'exécution.

2.9.1 Langage de définition de patrons

Dans Waly (2011), on présente un outil pour avertir un administrateur de la présence d'une attaque informatique. Pour ce faire, on analyse des traces et on tente de repérer des patrons précédemment définis. Lorsqu'une section de la trace correspond à un des patrons, l'information est envoyée à un module d'affichage dans le but de fournir des détails à l'utilisateur. L'outil effectue d'abord un prétraitement sur les événements dans le but de supporter de multiples traces. Cela permet aussi d'envoyer les événements en temps réel ou après coup de manière transparente au module de détection. Cela demande toutefois d'implémenter le module de prétraitement pour chaque trace. Dans le cadre des tests, le traceur LTTng a été utilisé.

Pour détecter, on utilise des scénarios qui sont des ensembles de règles devant être suivies dans l'ordre. Pour définir ceux-ci, un langage a été développé. Outre les différents filtres sur un événement, on retrouve également la définition des délais. Une règle pourra donc être de rencontrer A fois l'événement B avec le paramètre C dans les D prochaines nanosecondes. À chaque événement reçu, la règle courante de chacun des scénarios est testée. Si une règle est validée, le scénario correspondant va passer à la prochaine règle ou, s'il est arrivé à la fin, va avoir repérer le patron. À l'inverse, si la règle n'est pas validée, le scénario peut rester stable ou encore devenir invalide lorsqu'on ne respecte pas les contraintes de temps. On peut toutefois mentionner qu'il n'est pas possible de lire une seule fois la trace, et de ne pas conserver les événements, tout en calculant le scénario d'un nombre d'événements devant se produire dans un certain temps. En effet, si cela prend par exemple trois fois l'événement A en 10 secondes, et qu'il y a un premier A puis, après 9 secondes, 3 A arrivant à une seconde d'intervalle, il conviendrait de discarter le premier A seulement. Il n'est cependant pas mentionné comment ce problème est résolu.

2.9.2 Détection de patrons périodiques

Dans López Cueva et al. (2012), les chercheurs présentent une technique pour identifier différents patrons périodiques dans une trace et trouver ceux qui sont en conflit entre eux. Cela signifie que le temps entre certaines exécutions d'un patron ne suivra pas sa période et que cela pourra être corrélé à la présence d'une autre tâche périodique. Pour ce faire, ils modifient d'abord les temps d'occurrence des événements pour augmenter la présence de périodes fixes. Dans un de leurs tests, les chercheurs ont par exemple considéré les événements arrivés dans un intervalle de 0.1 ms comme ayant le même temps. Ensuite, ils identifient toutes les séquences d'un même événement qui se répète, avec leurs périodes et leurs points de départ. Puis, ce jeu de données est réduit en groupant pour chaque événement les séquences qui ont une période identique ou qui sont un multiple d'une autre période, mais qui ont des débuts différents. Cela donne donc les événements qui se répètent avec une même période, mais en tolérant des écarts d'une durée quelconque. Ensuite, on compare les différents patrons pour voir si les écarts en question pourraient être expliqués par un autre patron. Pour ce faire, pour chaque patron, on mesure la présence des autres patrons dans les moments où il est inactif. Si la présence est plus grande qu'un certain seuil ajustable, on suppose que les deux patrons sont en compétition. Avec cette technique, les chercheurs ont réussi à trouver des problèmes dans des applications multimédias.

C'est très intéressant de pouvoir détecter des problèmes sans que l'utilisateur n'ait à identifier lui-même ce qui constitue le patron d'une exécution périodique normale. Cependant, l'approche présente quelques inconvénients. Tout d'abord, le prétraitement du temps des événements peut poser problème. En effet, cela demande d'avoir une idée de la précision de la période des tâches. Par exemple, si une tâche se produit toutes les 9 à 11 ms, il faudra prendre un intervalle d'au moins 2 ms pour éviter d'être constamment considéré dans un écart. Cependant, comme c'est le même traitement pour tous les événements, cela peut poser problème si on a des tâches dont les périodes ont des précisions différentes.

Ensuite, cette approche fonctionne bien avec des événements de haut niveau comme une entrée de fonction, mais un même type d'événement de bas niveau pourrait faire partie de plusieurs tâches différentes, alors que ce serait ici considéré comme faisant partie de la même tâche. Il pourrait être intéressant de tenir compte des fils d'exécution pour tenter de séparer des événements du même type qui ne sont pas reliés, ou de chercher à les exclure des calculs. Une autre limitation évidente est que ça fonctionne uniquement avec des séquences qui se répètent avec une période fixe, et non pour des tâches qui se produisent de manière sporadique.

2.9.3 Extraction de métriques

Dans Terrasa and Bernat (2004), on montre une méthode pour extraire des métriques intéressantes à partir d'une trace du système d'exploitation. Pour ce faire, on utilise des machines à états qui reçoivent les événements et enregistrent les données. Les métriques calculées sont le pourcentage d'utilisation du système, le temps de réponse (c'est-à-dire le temps à partir duquel la tâche est prête jusqu'à la fin de son exécution), le temps de calcul et le temps en blocage.

Un exemple d'automate pour le temps de réponse est de passer dans l'état "Ready", en notant le temps à la réception d'un événement indiquant que la tâche est prête, et ensuite de passer à l'état ""Finished" en soustrayant le temps précédent au temps courant à la réception d'un événement indiquant que la tâche est terminée. Cette approche est intéressante, mais elle manque de flexibilité. En effet, pour être fonctionnelles, les tâches doivent être ordonnancées avec la politique SCHED_FIFO, utiliser le Priority Ceiling Protocol et avoir des priorités différentes. La raison pour laquelle on doit utiliser des priorités différentes est qu'on utilise les priorités pour déterminer les blocages. Cela donne des résultats intéressants, mais il aurait été préférable de pouvoir éditer les machines à états pour étendre le modèle. De la même manière, il aurait été intéressant de pouvoir identifier les sections de la trace qui présentent des exécutions problématiques.

2.9.4 Détection d'exécutions problématiques

Dans Rajotte (2014), les auteurs ont utilisé une approche similaire pour définir un modèle d'exécution des tâches temps réel en utilisant uniquement des traces noyau LTTng. Pour ce faire, ils ont défini cinq états soient : BLOCKED, PREEMPTED, WAITING, RUNNING et READY. Pour définir les changements d'état, seulement les événements *sched_wakeup* et *sched_switch* ont été utilisés. Ils ont cependant ajouté des conditions pour que leur modèle soit valide : les priorités temps réels assignées pour chaque tâche doivent être différentes et on doit utiliser des mécanismes d'héritage de priorité. Dans ces conditions, lorsque la priorité du fil d'exécution nouvellement ordonnancé est la même que celui qui s'exécutait précédemment, on sait que ce dernier était bloqué et qu'il y a eu héritage de priorité. Ensuite, si l'état précédent du fil d'exécution était TASK_RUNNABLE, cela signifie qu'il a été préempté.

Outre les limitations inhérentes au modèle, il convient de mentionner que les tests ont été effectués sur un seul processeur et qu'il présente quelques lacunes sur plusieurs processeurs. En effet, si un fil d'exécution s'exécute sur un processeur et se retrouve bloqué par un autre fil d'exécution s'exécutant en parallèle sur un autre processeur, c'est plutôt l'événement

sched_pi_setprio qui est généré pour indiquer l'héritage de priorité. Dans le modèle, on pourrait se retrouver dans l'état WAITING au lieu de BLOCKED.

On peut également souligner que le modèle n'est pas compatible avec la politique d'ordonnancement SCHED_DEADLINE puisque les fils d'exécution ainsi ordonnancés se voient tous attribuer la priorité maximale pouvant être définie par un utilisateur et donc la même priorité. Finalement, cela ne supporte pas plusieurs tâches exécutées sur un même fil d'exécution, puisqu'on ne pourra pas faire la distinction entre celles-ci. Cela nous empêche donc d'utiliser ce modèle dans diverses situations courantes comme quand un nouveau fil d'exécution est créé à chaque exécution de la tâche, ou quand un patron *thread pool* est utilisé.

Une autre innovation intéressante de ce projet est de trier les exécutions obtenues par ordre de temps d'exécution, dans une vue qui est synchronisée avec les autres vues de l'outil Trace Compass. Il est donc très rapide d'identifier les exécutions problématiques et d'obtenir plus de détails sur celles-ci. Ce modèle a d'ailleurs permis aux auteurs de trouver rapidement des zones d'intérêt à analyser plus en profondeur qu'il n'était pas facile de voir en utilisant les autres vues. De plus, on peut mentionner que la génération de ce modèle présente un très petit surcoût par rapport à la lecture de la trace uniquement.

2.10 Conclusion de la revue de littérature

D'abord, différents outils pour tracer sous Linux et analyser les traces d'exécution ont été présentés, mais aucun ne semble constituer une solution parfaite pour trouver des problèmes spécifiques aux systèmes temps réel. Or, plusieurs de ces problèmes ont été relevés, comme les problèmes d'ordonnancement et de gestion de la mémoire cache, de même que des métriques à analyser. C'est pourquoi il semble pertinent de voir s'il serait possible d'améliorer les outils existants et d'en développer de nouveau en se concentrant sur l'analyse de ces problèmes.

Ensuite, différentes méthodes pour repérer et découvrir des patrons dans une séquence ont été présentées. Il serait également bien de tester et d'adapter ces méthodes pour les traces d'exécution de manière à limiter les connaissances nécessaires pour trouver des problèmes et pour faciliter la tâche des développeurs qui désirent analyser leurs systèmes.

CHAPITRE 3 MÉTHODOLOGIE

Ce chapitre présente les aspects méthodologiques du présent projet de recherche. Plus précisément, l'environnement de travail utilisé sera d'abord présenté. Viendra ensuite une explication du choix des différents outils sur lesquels cette recherche est basée. Finalement, les différents programmes testés seront introduits.

3.1 Environnement de travail

Comme il a été mentionné précédemment, les systèmes temps réel nécessitent un environnement de travail avec un grand déterminisme. Idéalement, la variation de latence observée devrait être minimale afin de s'assurer de toujours respecter les échéances. Cela facilite également le repérage d'exécutions problématiques puisqu'on peut ainsi aisément les distinguer par rapport aux autres exécutions.

3.1.1 Matériel

Les spécifications de la machine utilisée sont présentées dans le tableau 3.1. Celle-ci a été utilisée à la fois pour collecter les traces et pour les analyser. De plus, des outils de vérification ont été utilisés pour s'assurer que le matériel ne présentait pas de problèmes et permettait d'avoir une latence stable.

3.1.2 Logiciel

Un noyau Linux 3.12 avec le correctif PREEMPT_RT a été utilisé afin d'obtenir des caractéristiques permettant un meilleur contrôle de la latence. Ensuite, l'*Hyper-threading*, qui permet d'avoir deux coeurs virtuels par coeur physique, a été désactivé puisque cela cause de la variation dans les temps d'exécution. De même, la fréquence des CPU a été configurée

Tableau 3.1 Spécifications matérielles de la machine de test utilisée

Composant	Description
Carte mère	Intel Corporation DX58SO
Processeur	Intel Core i7-920 à 2,67 GHz
Coeurs physiques	4
Mémoire vive	3 x 2 Gio Kingston DDR3 1067 MHz

en mode performance de manière à rester constamment à la fréquence maximale et d'ainsi offrir plus de déterminisme qu'en étant ajustée selon la charge.

3.2 Outils

Les différents traceurs et outils d'analyse de trace pertinents ont été présentés lors de la revue de littérature. Il convient maintenant de présenter et justifier nos choix.

3.2.1 LTTng

Le traceur LTTng a été utilisé pour collecter les traces d'exécution. En effet, il présente le plus petit surcoût parmi les autres solutions équivalentes offertes sous Linux et présente une excellente mise à l'échelle avec plusieurs coeurs utilisés. De plus, comme il a été mentionné précédemment, il a déjà été testé avec des applications temps réel afin de s'assurer qu'il présente un impact minimal sur les applications tracées.

3.2.2 Trace Compass

Dans le cadre de cette recherche, nous avons développé des analyses pour les traces de systèmes temps réel. Pour ce faire, nous avons utilisé l'outil de visualisation et d'analyse Trace Compass basé sur le *framework* Eclipse. Il s'agit d'abord d'un logiciel libre avec une communauté active qui l'améliore constamment. Ensuite, il présente de bonnes caractéristiques de mise à l'échelle et fournit une infrastructure puissante permettant d'ajouter facilement de nouveau module d'analyse. On y retrouve d'ailleurs déjà de nombreuses analyses intéressantes comme celle permettant d'obtenir le chemin critique d'une application en utilisant uniquement des événements noyau.

3.3 Programmes de test

De nombreux programmes de test ont été utilisés afin de tester nos analyses. Il fallait d'abord s'assurer de son bon fonctionnement avec les différentes politiques d'ordonnancement disponibles. Ensuite, il fallait trouver des cas avec des problèmes suffisamment complexes pour être présentés.

3.3.1 Politiques d'ordonnancement

Pour s'assurer du fonctionnement correct de nos analyses, peu importe la politique d'ordonnancement utilisée, nous avons développé un programme simple composé de tâches pério-

diques. Les multiples tâches se font ensuite attribuer une période, une politique d'ordonnancement et une priorité différente. Puisqu'il y a plus de tâches que de coeurs disponibles, elles vont se préempter. Des centaines de traces ont ainsi été générées afin de couvrir les différents cas.

3.3.2 Problèmes courants

Plusieurs programmes simples ont été développés de manière à générer des problèmes typiques. Premièrement, nous avons fait trois programmes générant occasionnellement des inversions de priorité. Dans tous les cas, un fil d'exécution de haute priorité dépend d'un fil d'exécution de basse priorité, alors que plusieurs fils d'exécution de priorité intermédiaire s'exécutent périodiquement. La variation entre les programmes vient de la dépendance qui est un *pthread_mutex* partagé et utilisant le protocole *PTHREAD_PRIO_NONE* n'implémentant pas l'héritage de priorité, un échange de messages avec une *POSIX message queue* ou encore un envoi de signal. Ainsi, si le fil d'exécution de basse priorité se fait préempter par ceux de priorité intermédiaire avant d'avoir terminé son exécution, le fil d'exécution de haute priorité se retrouve bloqué, par exemple en attendant le mutex ou un message. Puisqu'il doit attendre la fin de l'exécution des fils d'exécution de priorité intermédiaire sans que ceux-ci ne le bloquent directement, il s'agit d'une inversion de priorité.

Nous avons également testé les différents protocoles permettant de résoudre cette inversion. Nous avons donc tracé des programmes utilisant le PIP et le PCP expliqués à la section 2.3.1. Cela nous a permis de valider nos techniques de reconnaissance des exécutions et de récupération des priorités.

Nous avons aussi généré des traces avec des échanges de messages, des fautes de page, des ressources partagées et différents mécanismes de synchronisation.

3.3.3 Applications industrielles

Nous avons aussi visité trois entreprises de manière à tester notre outil sur des applications réelles et à obtenir des commentaires constructifs permettant de l'améliorer. Les cas intéressants ont ensuite été reproduits pour être présentés.

3.3.4 Traçage

Nous avons utilisé les *cpusets* permettant de spécifier sur quel CPU les différentes tâches doivent s'exécuter. Nous avons ainsi isolé les CPU 1, 2 et 3 pour exécuter les applications temps réel et garder le CPU 0 pour les autres tâches.

Il aurait été possible d'attribuer un CPU au traceur, mais cela n'aurait laissé que deux coeurs pour les applications temps réel. Nous avons plutôt décidé d'augmenter la grosseur des tampons de LTTng tel que présenté dans notre revue de littérature afin de s'assurer de ne pas rejeter d'événements lors d'une charge temporaire trop importante.

3.4 Présentation de l'article

L'article reproduit au chapitre 4 présente notre démarche pour concevoir un outil permettant d'aider les développeurs à analyser les systèmes temps réel. La première étape consiste à laisser les utilisateurs définir le modèle d'exécution des tâches avec l'aide d'un algorithme de découverte de patrons. La deuxième étape permet de localiser les segments de trace correspondant au modèle et de les afficher. Puis, la troisième étape est de sélectionner les exécutions qui semblent problématiques afin de finalement les analyser. L'analyse permet entre autres de combiner la recherche du chemin critique avec les informations d'ordonnancement afin de fournir des renseignements utiles pour repérer la source de problèmes.

Les différents programmes de test élaborés ont donc pu être testés avec l'outil développé. Cela a permis de s'assurer qu'il soit fonctionnel avec les différentes politiques d'ordonnancement et qu'il soit utile pour trouver une vaste gamme de problèmes.

CHAPITRE 4 ARTICLE 1 : PROBLEM DETECTION IN REAL-TIME SYSTEMS BY TRACE ANALYSIS

Authors

Mathieu Côté
Polytechnique Montréal
mathieu.cote@polymtl.ca

Michel R. Dagenais
Polytechnique Montréal
michel.dagenais@polymtl.ca

Submitted to Hindawi Advances in Computer Engineering

4.1 Abstract

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

4.2 Introduction

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

The moment at which a new job has to be executed is called the arrival time, and the moment at which a job actually starts to be executed is called the start time. If the jobs arrive at

fixed interval, the task is called periodic, otherwise it is called sporadic. Periodic tasks are often driven by timer, like the processing of video frames. On the other hand, sporadic tasks are often driven by interrupts, like the response to an user action. In both cases, there will be deadlines, but it will be deadlines relative to the start time for the sporadic tasks, and absolute deadlines for the periodic ones.

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

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

Our objective is to develop such a tool. To test our concepts, we used a Linux kernel with the PREEMPT_RT patch. This was shown to provide excellent real-time response with the Linux kernel. In addition, we used the LTTng tracer, characterised by a very low overhead. This tracer has already been tested on a PREEMPT_RT patched Linux kernel with good results (Beamonte, 2013). We implemented our analysis as a plug-in within Trace Compass, an open source and flexible trace visualiser in the Eclipse framework. Trace Compass is highly scalable and provides a powerful infrastructure, including interesting analyses like the critical path computation (Trace Compass, 2015).

Because they are frequent in real-time systems, we decided to focus on problems related to scheduling and priority. The scheduler is the component that selects the threads that will be executed next on the CPUs. Each thread has a scheduling policy and a static priority that are used by the scheduler to take decisions. To be able to analyze the various tasks, we need a method that can support the different scheduling policies available on Linux. In addition, we want to detect priority inversions, when a higher priority task is needlessly waiting on a lower priority task, usually because the latter is holding a lock while waiting on a third task of medium priority. We also want to support the different protocols used to avoid this inversion, like the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) (POSIX Programmer's Manual, 2003). In PIP, lower priority threads inherit priority

from higher priority threads waiting on them, while in PCP, threads priority is boosted when they obtain a resource, to the highest priority of any threads that can obtain it.

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

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

4.3 Related Work

In this section, we will review the studies that focus precisely on trace analysis for real-time systems. First, a pattern language is defined in (Waly, 2011) and the trace is iterated until the pattern is found. This is used to detect security attacks. However, this is not to find problems, but to detect occurrences when you already know the representation of the problematic situation in the trace.

In (López Cueva et al., 2012), the authors present a method to identify periodic patterns using the gap between events of the same type. Then, they try to see if those patterns are in conflict by correlating the perturbation in the periodicity of a certain pattern, with the activity of the others. This works well with high-level events, like a function entry, that can be associated with a specific real-time task. However, the same low-level event type, like a scheduling event, may be used to define many different task types. Another limitation is that it only works for periodic tasks, and thus will not handle sporadic ones.

A method to extract useful metrics based on kernel traces is defined in (Terrasa and Bernat, 2004). The authors used different state machines for the metrics, like the usage rate of system resources, the running time or the response time. However, there are multiple constraints, because the different state changes will depend on the scheduling policies and the method

used to avoid priority inversions. In the present situation, their work will only compute valid results for one specific combination. It will also provide some metrics, but not pinpoint problematic executions.

A similar method is used in (Rajotte, 2014). However, in addition to the metrics, the authors also retrieve the task intervals. Then, they present these in a comparative view, sorted by the longest interval time, to help viewing the differences. Users will typically focus on the analysis of the jobs with the longest times first, since these are more likely to be problematic for the real-time performance. However, there are some limitations. Because it is a fixed model, this will again only work well with a specific scheduling policy and method used to avoid priority inversions. In that work, it was geared towards the SCHED_FIFO policy and the Priority Inheritance Protocol.

Multiple visual tools have been developed to display traces, like Tracealyser (Percepio, 2015), TuningFork (Bacon et al., 2006), WindView (Wind River Systems, 2015), Vampir (GWT-TUD GmbH, 2015), Zinsight (IBM Research, 2015), KernelShark (Rostedt, 2011) or Trace Compass (Trace Compass, 2015). Most of them will focus on a timeline view to show the traces. It is convenient to follow the flow of a program to understand its behaviour. However, it is not very practical to compare different parts of the trace. Also, to the best of our knowledge, only Trace Compass presents a critical path analysis based on kernel traces, and none of these systems exploit or extend this critical path analysis to add useful information for real-time systems.

4.4 Pattern discovery

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

Before explaining the algorithm, we will start with few definitions illustrated in Figure 4.1. An event has an event type, a content and a timestamp corresponding to the time at which it was generated. A sequence is composed of multiple ordered events. A sub-sequence is a sub-group of events from the same sequence. An episode is a group of event definitions that need to occur in order. A sub-episode is a sub-group of event definitions from the same episode. An occurrence is a sequence corresponding to an episode. The support is the

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

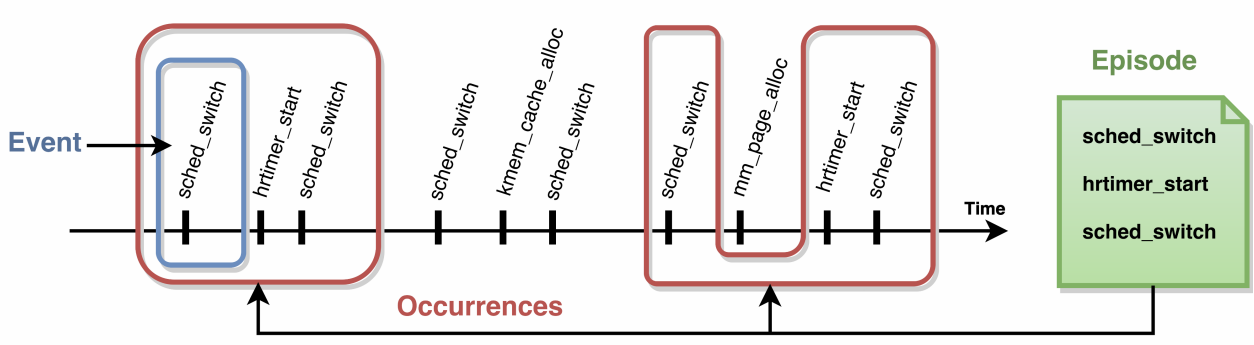

Figure 4.1 Graph showing events in the trace as a function of time, as well as the two occurrences of a given episode.

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

4.4.1 MANEPI algorithm

Based on the previous criteria, we decided to use the *MANEPI* algorithm (Zhu et al., 2010) that discovers patterns using minimal and non-overlapping occurrences. Indeed, we are interested in non-overlapping pattern occurrences, since it is the case for real-time tasks within a specific thread. Also, the minimal occurrences, which means that the corresponding episode can't be found in a sub-sequence of the occurrence, would result in more precision in the calculation of episode frequency.

In a few words, the *MANEPI* algorithm will find episodes that are more frequent than a given support threshold, which represents the minimum number of repetitions that is needed to validate a pattern. Formal descriptions of the main parts of the algorithm are presented in Table 4.1 and Table 4.2. It starts by finding all frequent elements, which means they have more occurrences than the threshold. They will be considered as the basic elements. Then,

the algorithm uses the fact that all sub-episodes of a valid patterns must also be supported. This first implies that there is no need to consider the elements that are not frequent. It is also possible to start with episodes made of only 1 basic element and increment them to find larger valid episodes. More precisely, for each valid episode, the algorithm checks if the episode is still supported with the addition of each frequent element, until it is no longer supported. The algorithm is depth-first, which means that once the end of a branch is reached, it is possible to release the memory used to get the result, and thus lower the maximum memory consumption, as compared to a breath-first search. At each stage, the offset of the occurrences of the current episode are stored to calculate its support. The complex part of the algorithm resides in the calculation of the minimal and non-overlapping occurrences efficiently.

4.4.2 Algorithm modification

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

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

Because the trace can be very large, we also add a maximum number of events for which the patterns are searched. Only the intervals in the trace of this number of events will be kept in memory. This will result in a faster search and will prevent memory problems. In practice, the interesting sequences are rather short and there is a huge gap between the number of

Table 4.1 MANEPI algorithm with modifications to adapt to tracing events

Algorithm 1 : MANEPI with modifications (T, min sup, nb frequent)

Input: T: a trace <(EventType1, Timestamp1, other fields), (ET2, t2), ... , (ETs, ts)>
min sup: a support threshold (infinite if not defined)
nb frequent : number of frequent 1-episode to consider (infinite if not defined)
Output: a tree which stores all episodes in T whose support >= min sup

1. Create a tree with simply a root node
2. Create a HashMap with event types as keys and ArrayLists of indexes as values
3. Start with an index of 0
4. Read the trace and for each event do
    5. If the event TID is the one we are searching for
        6. Get the event type and insert the index in the corresponding ArrayList
        7. Increment index
8. Sort by size of ArrayLists
9. *min sup* = Minimum(*min sup*, size of the ArrayList of the event type at the position corresponding to the *nb frequent* parameter)
10. For each frequent 1-episode e do
    11. Create node Ne as a child of the root (that include the ArrayList of the indexes corresponding to the first event of each occurrences)
    12. MineGrow(Ne)

Table 4.2 MineGrow algorithm from Zhu et al. (2010)

Algorithm 2 :MineGrow(Na)

Input: Na: the node to be expanded
Output: the tree which contains all frequent episodes with prefix a

1. For each frequent 1-episode e do
    2. Let mo(B)= ComputeMO(Na.mo,mo(e)) -> minimal occurrences
    3. Let mano(B)=ComputeMANO(mo(B)) -> minimal and non-overlapping occurrences
    4. If sup(B) = size (mano(B)) >= min sup
        5. Create node NB as a child of node Na
        6. MineGrow(NB)

events needed to have a few repetitions and the default maximum number. This approach will thus lead to a valid result, despite this limitation, as long as the maximum number of events is reasonably large.

4.4.3 Support threshold

To use the algorithm, we must define a support threshold. We offer two options to the users. First, they can directly define the minimum number of repetitions. That way, users do not have to know how many events are in the trace, and how many events are included in the pattern. They only need to have an idea of how many jobs of the tasks are present in the trace segment, in order to specify a lower bound. The higher this bound is, the faster the algorithm will be. Indeed, the episodes will be dropped more quickly because their support will sooner be under the support threshold, and there will be fewer frequent elements to iterate at each step. We can see in the Table 4.3 an example with a threshold of 2 that leads to three frequent elements. We can also visualize its impact in the Figure 4.2 and the Figure 4.3 that represent respectively episodes with support above and below it.

Table 4.3 Exemple of a list of occurrences of events with their considered status according to a support threshold of 2.

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

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

Despite the threshold option, this algorithm can lead to exponential computation time growth if too few branches are removed. Indeed, if the support threshold is defined sufficiently high, the episodes will soon be discarded, which means that there will be fewer matches. However, if it is not the case, this can lead to some problems with the calculation time. To avoid that,

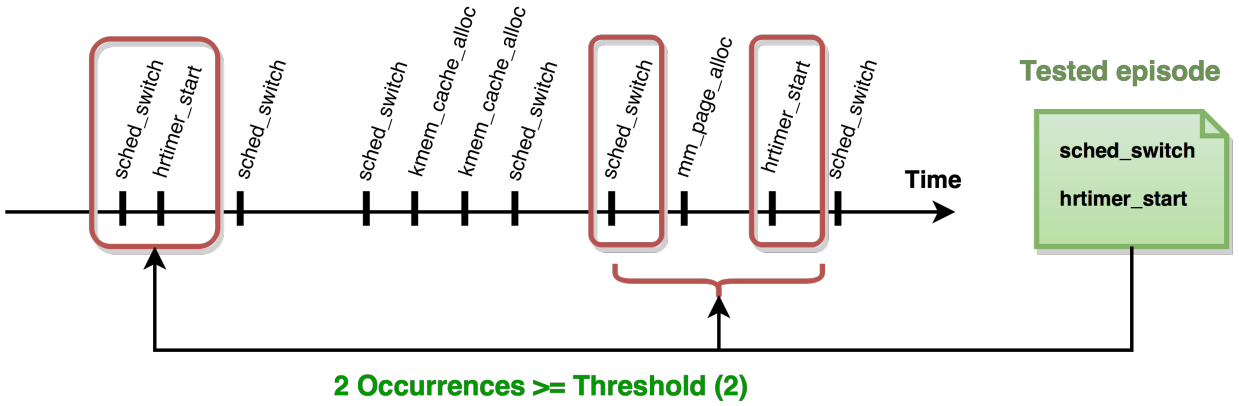

Figure 4.2 Graph showing events in the trace as a function of time, as well as the two occurrences of a given episode that should be reported as the used support threshold has a value of 2.

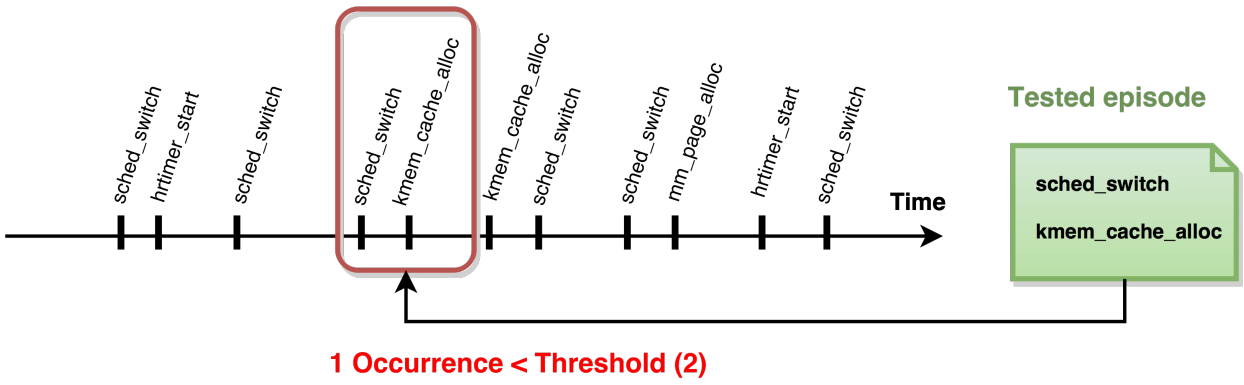

Figure 4.3 Graph showing events in the trace as a function of time, as well as the only occurrence of a given episode that shouldn't be reported as the used support threshold has a value of 2.

we add a computation time limit that can be set by the users. The results up to that point will still be available.

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

4.5 Pattern matching

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

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

The events are defined using the event name and the event fields. The model definition also supports some basic operations. First, the keyword $tid will mean the execution TID. For example, the event *sched_switch, next_tid=123* will be matched with the definition *next_tid=$tid* and the execution with TID equal to 123 even if the event occurs on a different

TID. Otherwise, the TID on which the event occurs must match the execution TID. The use of the token *&* can also be useful in the case of flags. For exemple, if the flag *1* must be set and *2* must not, we can do *param_name&3=1*. On their side, starting and ending threads are usually defined using TIDs, but the process name is also a supported way to define them.

4.5.1 Same TID mode

For the case where the start and end TIDs are the same, there is a graphical interface (see Figure 4.4) to add, remove and change the order of the events in the definition. Each valid TID will have his own state machine instance to detect executions. Those instances are stored within an *hashmap* with TIDs as keys to process each event in constant time. While iterating the trace, the state machines are created upon the first encounter of the corresponding valid TID. When an event of the trace is processed, the machine in charge of finding executions for that thread processes the event and compares it to its next definition. If the last state definition is reached and matched, then the execution is registered in the list of valid executions, and the state machine returns to its starting state.

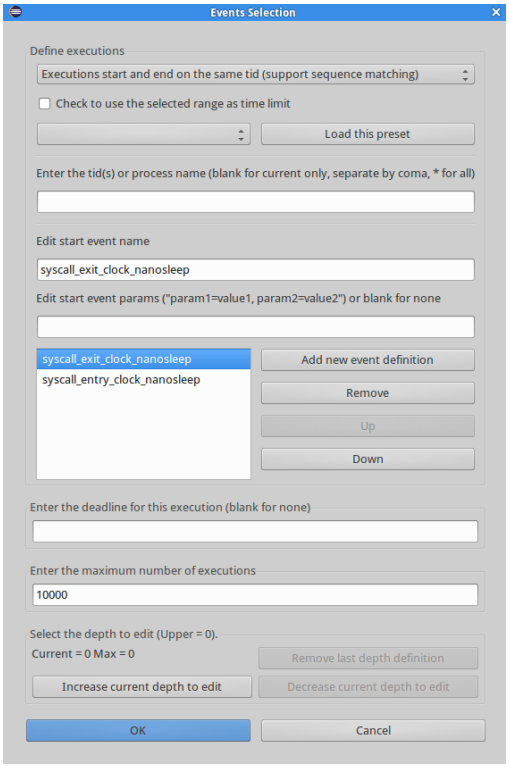

Figure 4.4 Graphical interface to add, remove and change the order of the events in the definition.

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

4.5.2 Different TIDs mode

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

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

4.5.3 Options

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

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

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

4.5.4 Complexity

We will define an *event matching* as the comparison of the event type names (represented by a unique id) and then, depending on the case, the comparison of some of the event fields. The complexity of the pattern matching algorithm will be of one *event matching* per *sched_switch* event to compute the running TIDs (current thread on each CPU core). There will be an additional *event matching* for each event that occurs on the analyzed threads. It should be noted that the events are read and decoded only once in this process to increase performance.

4.6 Views

Once we have all the trace segments corresponding to the execution of the real-time jobs, they are displayed in the *Comparison View*. The goal is to easily identify which executions present irregularities. The *Time Perspective View* will also be useful to find strange behaviour in a more global perspective. Then, the users will select executions they want to further analyze. The *Critical Path Analysis* will be performed on those executions and the *Critical Path Complement View* will be used to present relevant scheduling information. This is a powerful approach because it will graphically display the synchronization dependencies among the different states involved in the relevant executions.

4.6.1 Comparison View

This first view, shown in Figure 4.13, presents the various executions to facilitate the identification of problematic ones. As the events in the trace are collected with timestamps, it would be natural to show the trace in a timeline view. This is in fact a common view in Trace Compass. However, for comparing real-time tasks, we superpose the different jobs executions using the same time scale. That way, it is much easier to compare executions instead of having to look at them along the time axis, where problematic executions could be few and far apart in a trace timeline. Also, the segments of the trace between executions are not shown, to facilitate the visual analysis by avoiding irrelevant information.

By default, the executions are sorted by duration, starting with the longest. This metric is based on the elapsed time and includes the time when the thread is not running, either

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

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

4.6.2 Time Perspective View

This view, illustrated in Figure 4.10, also helps to identify problematic executions, but using a global perspective. It will show the duration of the job executions as a function of their starting time. This can be useful to see if there is a pattern in the distribution of the running time. For example, the longest executions could all occur one after the other. In that case, it is better to check for some special condition happening at that moment. On the other hand, if the perturbations occur with a fixed period, it is probably due to a problematic interaction with another periodic task. In addition, this view allows to click on the dot associated with an execution to synchronize the other views with that time point, and highlight the corresponding execution in the *Comparison View*. There is also an option to specify a deadline and to show in red the executions that missed their deadline.

4.6.3 Critical Path Complement View

Once the user finds a suspicious or problematic execution, the goal is to further analyze it. The first step is to use the critical path analysis in Trace Compass which provides useful information about the significant dependencies of a thread. When the analyzed thread is blocked, the view shows the resources or threads after which it waits. When a thread on the critical path is preempted, it may be complex to retrieve the priorities of the different threads running during each preemption. You can however check the *Resources View* to find

what threads where running and to look for the different events that can result in a priority change. The *Critical Path Complement View*, used for that analysis, is displayed in Figure 4.15.

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

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

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

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

To populate this information from the dependencies graph, we first get the graph containing the selected thread. Then, in the case of direct interactions, we just add threads linked from

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

To avoid iterating through all sets to search if a thread is present, we store the information in a *hashmap*, with the TIDs as keys and references to the sets as values. To merge sets, we add the elements from the smaller set to the larger set and update the references. That way, searching for element takes a constant time, and the total complexity is linear with the number of links. To merge, the overall worst case is when each group is initially composed of 2 elements, and they are merged with a group of the same length, recursively, until there is only a single group. That will result in a worst case of *1/2 n (log2 (n) – 1)* updates, because each time half the elements must be updated. Thus, the worst case complexity of the algorithm is $O(n\log n + m)$ where n is the number of TIDs and m the number of links.

4.6.4 Extentded Time View

It can be useful in some situations to have more information than only the critical path and related threads. The goal here is to present different kernel facilities related to a specific job execution. Those will be presented in a timeline. The time range can match an execution of the *Comparison View* or the range may be specified graphically or by typing, for example to have a larger view of the situation. The view is shown in Figure 4.16.

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

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

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

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

4.7 Performance Analysis

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

4.7.1 Pattern discovery

First, we can see in Figure 4.5 that the execution times for the pattern discovery algorithm are erratic. The different traces tested all show a similar behaviour for the execution times. As explain previously, this is caused by the eligibility of new basic elements. This means that when the threshold decreases, enough to allow another element, this will lead to a jump in the execution time. Indeed, there is one additional verification for each valid episode, so more time is required to check all the branches. Moreover, more and longer sequences are kept.

The number of possibilities grows rapidly, as the factorial of the number of basic elements. That explains the general exponential trend, even if most branches are not checked due

to the lack of support for the corresponding episode. For example, with the simple case presented in Table 4.3, the support thresholds between 6 and 3 lead to only 1 episode to test (*sched_switch*). With a threshold of 2, two other elements are eligible (*kmem_cache_alloc* and *hrtimer_start*) and that leads to many possible episodes. However, not all of them will be tested. Like we can visualize in Figure 4.3, the episode *sched_switch, kmem_cache_alloc* is not valid and thus, the episode *sched_switch, kmem_cache_alloc, hrtimer_start* will not be checked.

The Figure 4.5 shows the time consumed by the pattern discovery algorithm for a trace where over 45000 events are considered for the selected thread, from the 4 millions events in the trace. Even with that amount, up to a threshold of 800, the algorithm takes less than one second to execute. It is also important to understand that having a threshold too small will result in a huge number of results. For instance, in the case presented, with a support threshold of 1600 occurrences, there are 4 patterns returned, but with a threshold of 500 occurrences, there are over 15000 valid patterns.

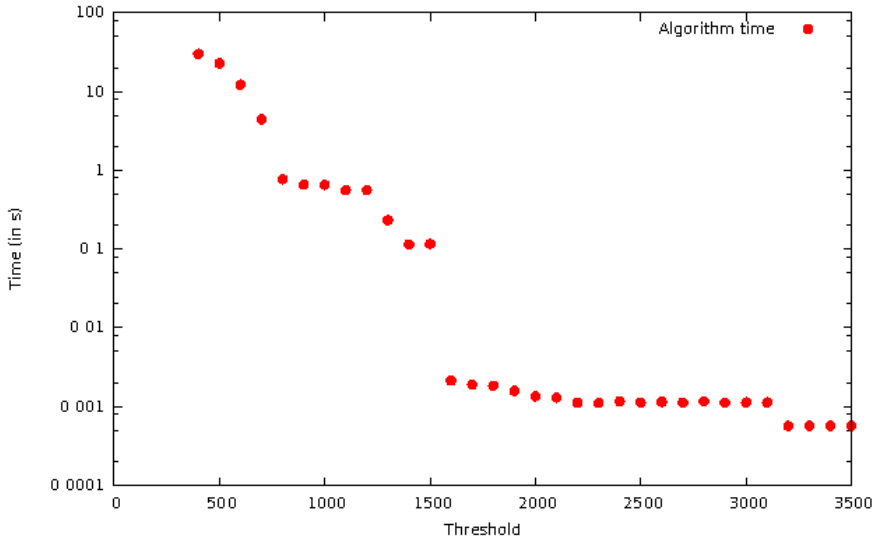

Figure 4.5 Average time consumed by pattern discovery algorithm for varying thresholds for a given trace.

4.7.2 Executions detection

The detection of executions relies on various factors. The first test was to compare it with the reading of the trace. During the detection, we try to parse the events only if necessary, and we ensure we are only parsing the name and the content once. We compare the executions detection of two models with reading the events name only and with reading the name and

the content. The first model is defined by the start and the end of a *nanosleep* system call and the events definitions have no fields matching.The second model represent messages exchange and the definition included events with fields to match the queue identifier.

The first model tested returns a few hundreds executions and the second one returns up to 300,000 executions with the bigger trace. The results, shown in Figure 4.6, show that both are actually faster than reading the name and the content of each event. However, as can be expected, they take more time than only reading the name. In fact, even without matching the executions, the detection of executions algorithm reads all the names to check for *sched_switch* events and parses the content of those events to maintain the running TIDs. To give an idea, in the largest trace presented there were over 1,3 million *sched_switch* events.

The second test compares the detection to other analyses in Trace Compass. The results are shown in Figure 4.7. The first analysis is building the state system used by the *Control Flow View* of Trace Compass. The second one is to build the dependencies graph. This give us a good insight because our analysis can be complemented by both of them. They appear to be much longer, so our work will not be the bottleneck of a complete analysis. In addition, the dependencies graph construction was running out of memory when the trace was too big.

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

The previous tests where made with the same TID mode, which is more complex than the different TIDs mode. However, we tested with a trace containing more than 8000 valid TIDs to do the matches, and the same TID mode was significantly slower, around 13%. This is because we need to create an instance by TID and to use a hashmap to match the tid with the instance. The results are shown in Table 4.4.

Table 4.4 Execution time for the two modes (Same TID mode and Different TIDs mode) compared to trace reading (in s)

	Read Name	Read Content	Same TID Mode	Diff TID Mode
Average	2,303	3,326	2,918	2,576
STD	0,025	0,025	0,023	0,026

For all these tests, we compare the number of executions detected with the number of events shown by the Trace Compass *Statistics View* and, each time it was possible to verify the results, we arrived at the same number of executions. This was the case for the two different modes. The execution time is fairly consistent. We can see in Table 4.4 a low standard deviation based on 10 repetitions for each test. This is similar to the execution time when only reading the trace.

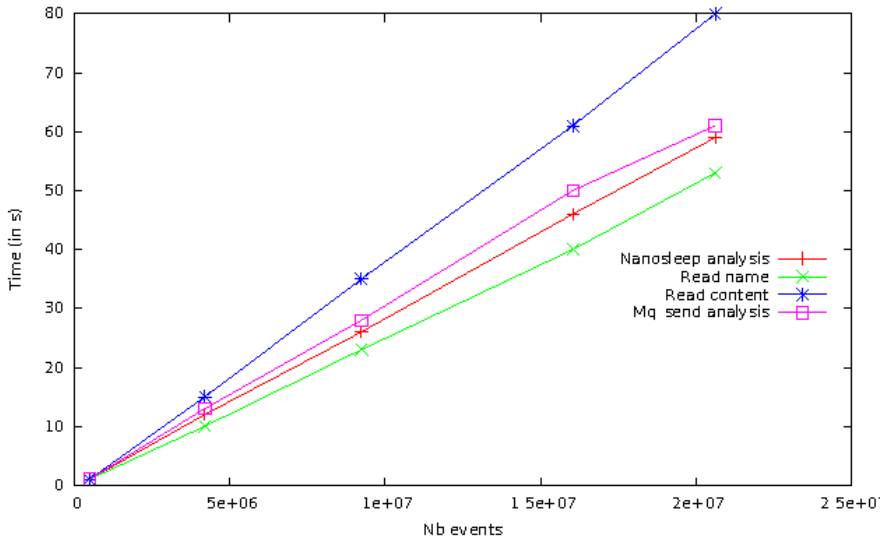

Figure 4.6 Time taken for our execution detection (nanosleep analysis and mq_send analysis) compared to trace reading in Trace Compass.

4.7.3 Views

The views appear to bring another limitation. They can lead to memory problems if there are too many lines, and they can take a long time to refresh. However, there is no point in displaying thousands of executions on separate rows. Thus, the problem will only occur when the default limits are increased. All our views use the same structure, inherited from Trace Compass, and follow the same trend. We can see the results in the Figure 4.8. We can see that the time to draw the views increases faster than linearly. This is due to the sorting, being O(n log n). However, for instance, with the default maximum of 10000 executions, it takes less than 10 seconds which is still acceptable.

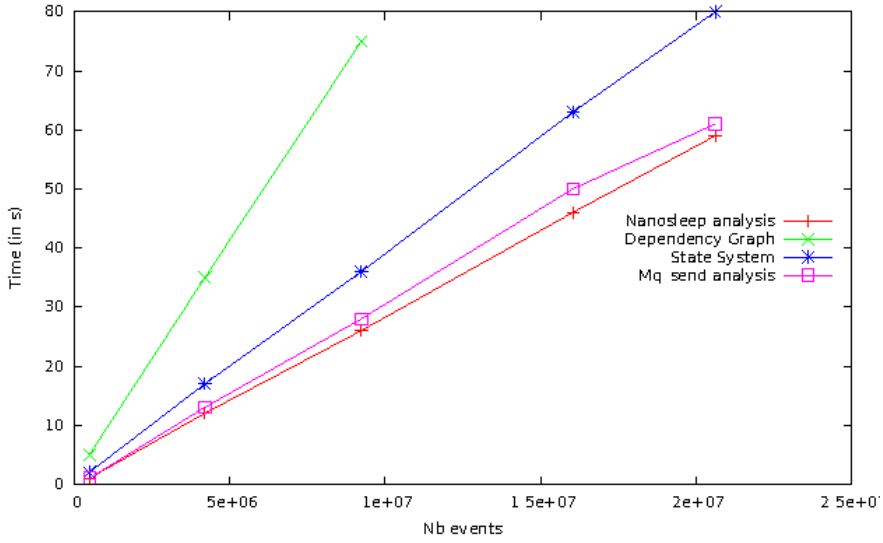

Figure 4.7 Time taken for our execution detection (nanosleep analysis and mq_send analysis) compared to other analyses in Trace Compass.

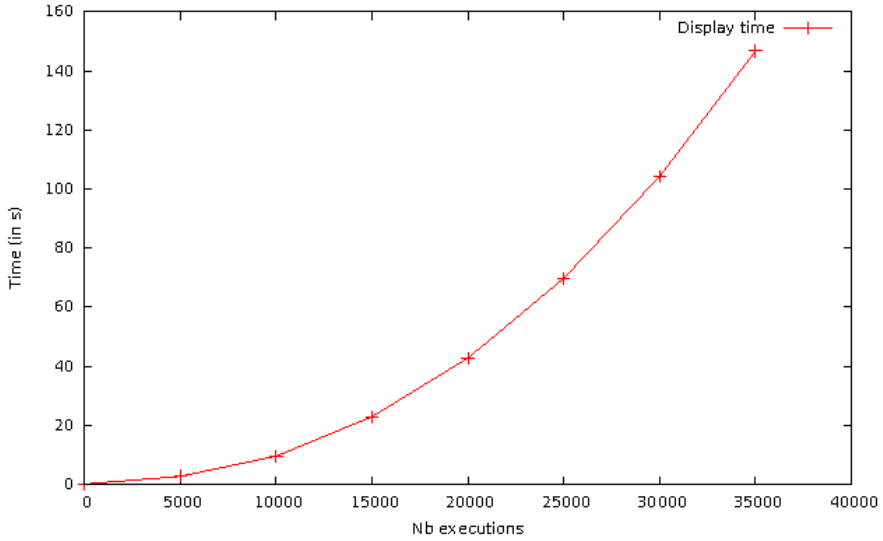

Figure 4.8 Time consumed in drawing Comparison View in Trace Compass as a function of increasing number of executions

4.8 Test cases

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

4.8.1 Real-time timer with higher priority task

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

With our tool, we define the job execution as the interval between the end of the code execution of two consecutive threads created by the timer. This way, if the problem occurs either with the timer thread or with the created threads, it will be detected as we can observe in Figure 4.9.

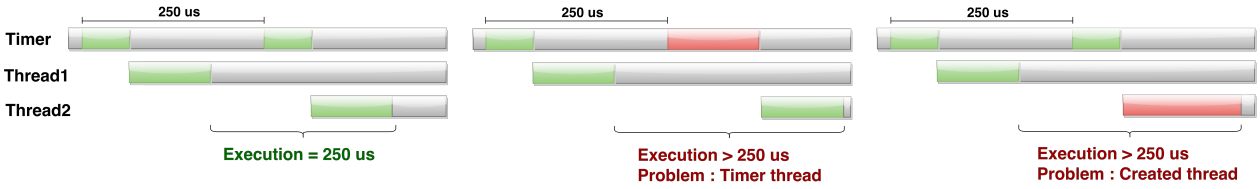

Figure 4.9 Definition of job execution as the interval between the end of the code execution of two consecutive threads created by the timer.

Running the analysis extracts the executions and highlights the longer executions. We can see in the Figure 4.10 that there were only a few outliers, but they were taking up to almost 4 times the average. When clicking on one of the problematic executions, we can see that the problem is at the level of the timer thread, in the Figure 4.11 and Figure 4.12, that show respectively the gap between the executions and the preemption. Then, we can look at the *Resources View* of Trace Compass to see the running threads.

Alternatively, we can also display the critical path of one of the longest executions, and the complementary information. That informs us that the execution was preempted because another thread had a higher priority. It was a configuration problem, because this thread was not supposed to have a higher priority in that situation. The main difficulty lied in the fact

that a very large number of threads were involved, designed by different programmers. Once the problem and its origin were pinpointed by the tool, the remedy was simple to devise.

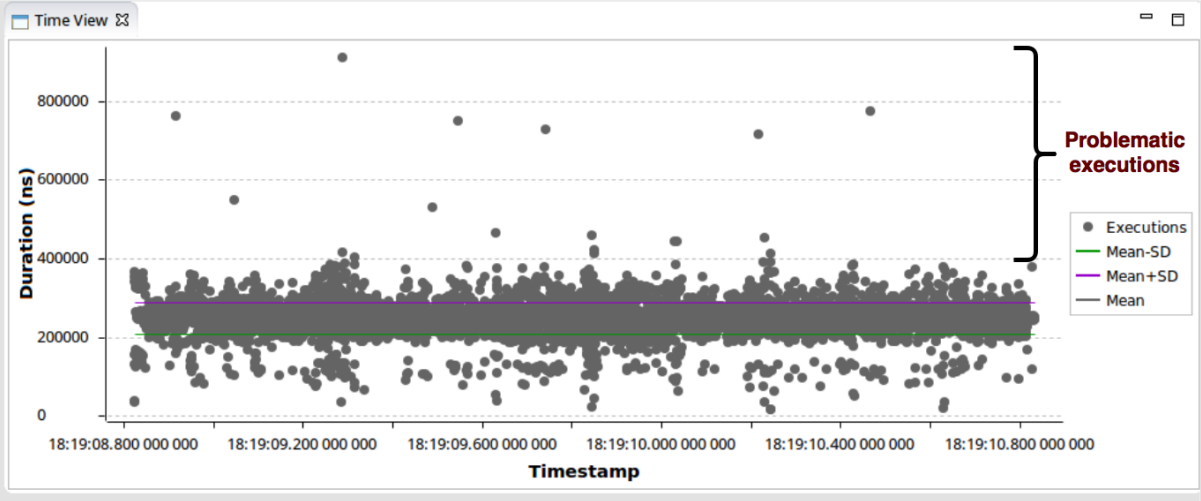

Figure 4.10 Perspective Time View that helps identify problematic executions using a global perspective

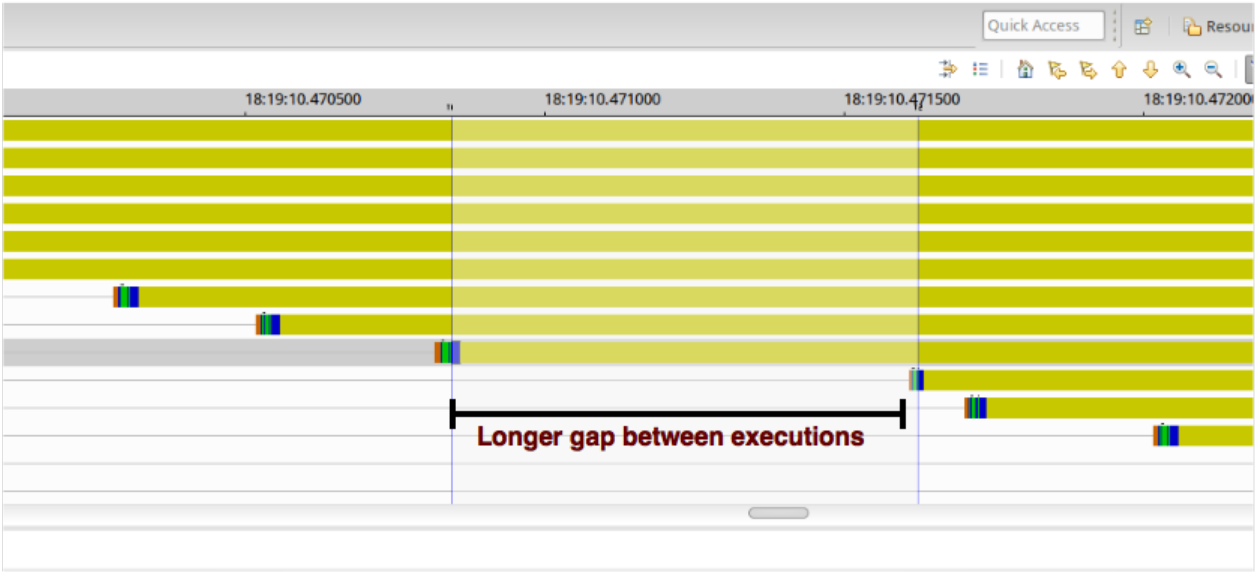

Figure 4.11 Control Flow View showing the gap between executions.

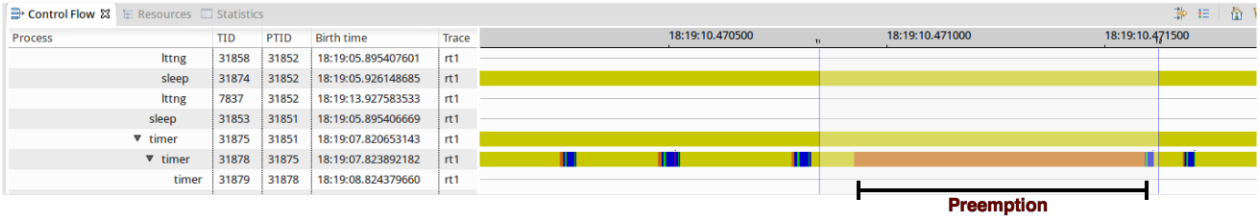

Figure 4.12 Control Flow View showing the preemption of the timer thread.

4.8.2 Waiting for message

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

The first step is to define our model. We use the pattern discovery tool with 12 basic events. This gives us many possible patterns, including the one we where looking for, based on the high resolution timer (*syscall_exit_clock_nanosleep* to *syscall_entry_clock_nanosleep*). We load the pattern and search for executions. This returns a few hundred executions including a few ten problematic ones that we can see in Figure 4.13, with the running time in green. With the longest execution, we check the critical path as shown in Figure 4.14. It shows us that the task thread (TID 3988) was blocked by a thread (TID 3950) that was preempted. With the *Critical Path Complement View* in Figure 4.15, we can see that the thread blocked was of a lower priority than other threads that preempted it. To prevent this situation, the priority of the thread sending the message should be increased.

Instead of using the critical path, another way of finding that the problem is caused by the thread with TID 3950 would have been to use the *Extended Time View*. It is shown in Figure 4.16 with the corresponding message queues. This can be really useful if there is a race condition to receive or send a message.

4.9 Discussion of results

The pattern discovery algorithm can give interesting patterns in a short time to help the users to define the execution models of the real-time tasks. However, they need to know on which thread to search and the pattern are search for only one thread. This can be a

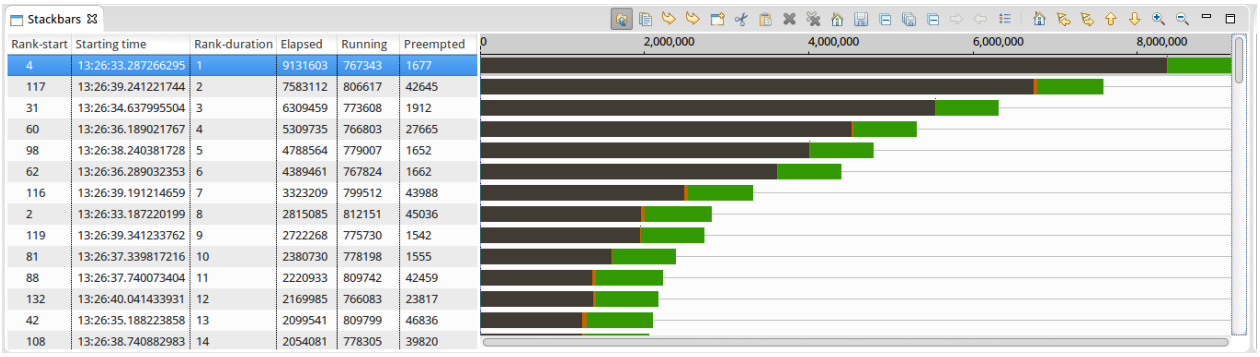

Figure 4.13 Comparison View in Trace Compass showing the difference in job execution times and statuses allowing the user to identify the most time consuming jobs.

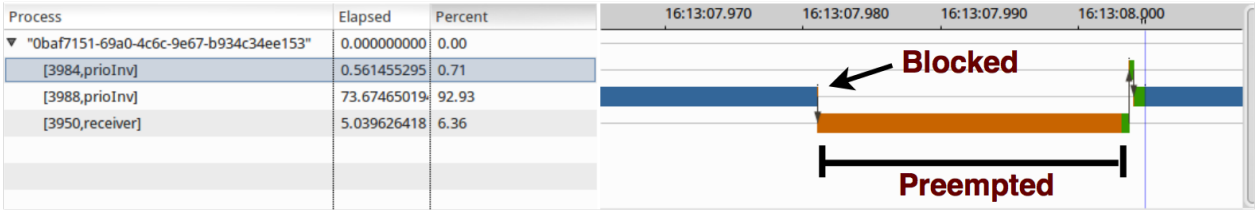

Figure 4.14 Critical Flow View showing that the task thread (TID 3988) was blocked by a thread (TID 3950) that was preempted.

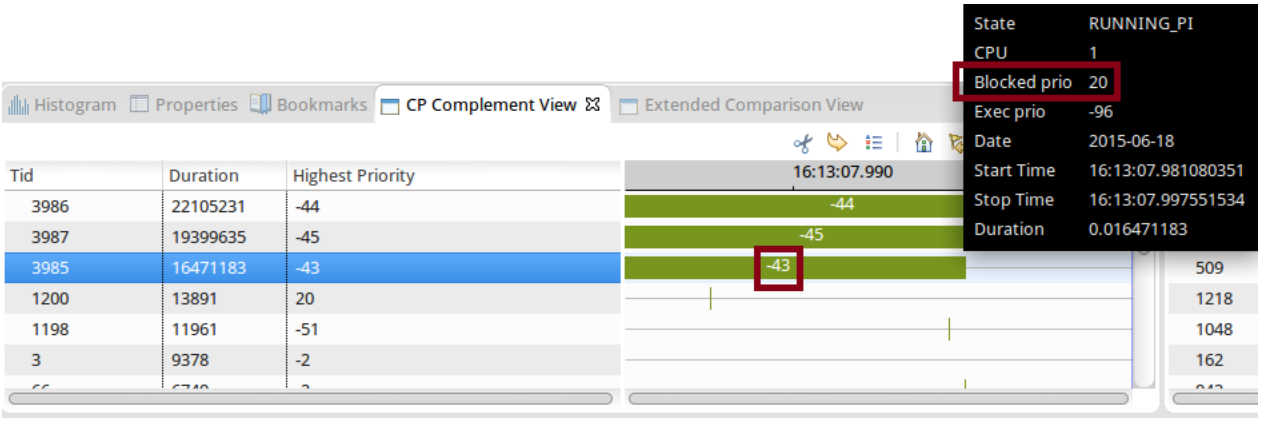

Figure 4.15 Critical Path Complement View showing that the thread blocked was of a lower priority than other threads that preempted it.

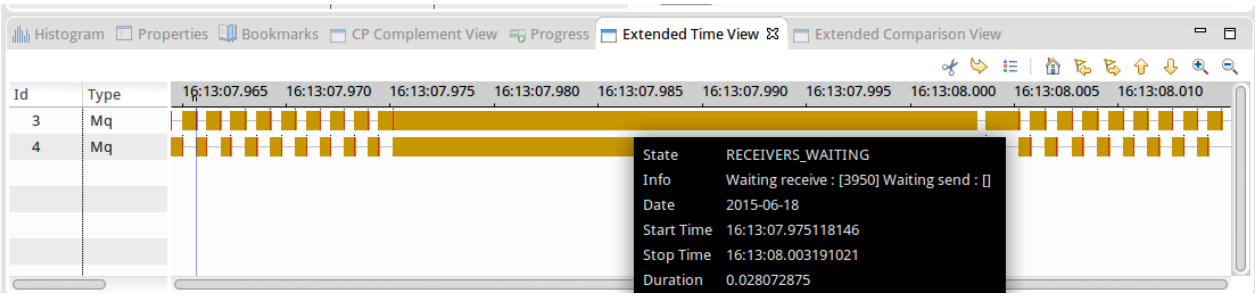

Figure 4.16 Extended Time View showing message queues and allowing to see if there are multiple threads waiting to receive or send a message.

limitation when there is an event occurring on an other thread that define the start or the end of the periodic executions.

The pattern matching algorithm works for very large trace. We show that the searching of periodic executions can be useful to efficiently find problems in real-time systems. To work, some events must be activated in tracing like the *sched_switch* and the one in the definitions of the models.

The two test cases show that the comparison view can be effective to find scheduling problems when comparing various executions of a periodic tasks. However, the display can be slow if there are more than few thousands of executions displayed, but there is an option to limit this number.

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

4.10 Future Work

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

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

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

4.11 Conclusion

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

4.12 Acknowledgements

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

4.13 Conflict of Interests

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

CHAPITRE 5 DISCUSSION GÉNÉRALE

Ce chapitre vise à discuter des résultats présentés dans l'article au chapitre 4 et à les comparer avec la revue critique de la littérature présentée au chapitre 2.

5.1 Découverte de patrons

Dans la revue de littérature à la section 2.6.6, une méthode pour trouver des patrons d'intérêt dans une trace d'exécution a été présentée. Tout d'abord, les auteurs utilisent la technique de la fenêtre glissante pour compter le nombre d'occurrences. Cela consiste premièrement à choisir un intervalle de temps correspondant à une fenêtre. Puis, il faut choisir dans combien de fenêtres un groupe d'événements doit être présent pour être considéré comme fréquent. Finalement, on retourne les groupes pertinents en calculant le nombre de fenêtres où ils sont présents. Il faut donc paramétriser la durée des fenêtres et le seuil correspondant au nombre de fenêtres dans lesquelles un patron doit se retrouver.

Une problématique importante est le choix de la durée des fenêtres. Si on choisit une fenêtre trop grande, on se retrouve avec une quantité énorme de patrons. À l'inverse, si on choisit une fenêtre trop petite, on peut manquer des patrons d'intérêt puisqu'ils peuvent empiéter sur plusieurs fenêtres.

Dans notre cas, nous avons plutôt choisi d'utiliser la technique des occurrences minimales. De cette manière, peu importe la période d'un patron, il sera retourné s'il est suffisamment fréquent. En effet, chaque occurrence se retrouve à être bornée par le début de celle qui suit et non par un temps défini. On peut toutefois noter que notre technique fonctionne uniquement lorsque les événements sont ordonnés. C'est donc valide pour des tâches s'exécutant sur un même fil d'exécution puisque les événements sont générés dans l'ordre, mais cela ne fonctionnerait pas pour des tâches complexes présentes sur plusieurs fils d'exécution. À l'inverse, avec la fenêtre glissante, on peut considérer que tous les événements d'une même fenêtre se produisent de manière simultanée et ainsi tolérer une variation dans l'ordre des événements.

Ensuite, la technique de la fenêtre glissante fait également en sorte que les occurrences des épisodes peuvent se chevaucher puisque les fenêtres se chevauchent, ce qui ne correspond pas au modèle normal des applications temps réel.

5.2 Définition du modèle d'exécution

A la section 2.9.1 de la revue de littérature, un article présentant un langage pour repérer des anomalies dans une trace a été présenté. On y décrivait des patrons comme de rencontrer A fois l'événement B avec le paramètre C dans les D prochaines nanosecondes. La notion de temps n'est cependant pas appropriée dans notre cas. En effet, on ne pourrait ainsi pas détecter une tâche exécutée sur un fil d'exécution qui se fait préempter un long moment et qui se termine plus tard puisque cela ne satisferait pas à la contrainte de temps. C'est toutefois précisément ce qu'on cherche à détecter.

Outre cela, nous avons inclus diverses fonctionnalités qui ne sont pas présentes dans leur langage et qu'il aurait fallu inclure. Par exemple, dans notre outil, une tâche peut être définie comme commençant et se terminant sur des fils d'exécution différents. De plus, il est possible d'avoir des exécutions imbriquées pour supporter des modèles plus complexes.

5.3 Analyse spécifique

Deux articles concernant l'analyse de traces dans les systèmes temps réel ont été présentés aux sections 2.9.3 et 2.9.4. Dans les deux cas, les auteurs utilisent des machines à état pour collecter différentes métriques utiles comme les temps où une tâche est bloquée. Cela nécessite toutefois un modèle prédéfini qui est uniquement valide avec de multiples contraintes comme l'utilisation de politiques d'ordonnancement particulière. Nous avons préféré laisser les utilisateurs définir le modèle correspondant à une exécution afin de supporter une plus vaste gamme de cas. C'est généralement possible de les représenter avec un modèle assez simple, mais il serait autrement possible d'utiliser des points de trace en espace utilisateur. De plus, nous avons inclus les modèles proposés dans les deux articles mentionnés dans des choix prédéfinis qu'il est possible de sélectionner. Cela permet aux utilisateurs d'utiliser ces modèles qui peuvent s'avérer intéressants.

Ensuite, nous avons reproduit les vues les plus pertinentes qu'on retrouve dans les différents outils de visualisation présentés. Ainsi, nous avons ajouté une vue présentant les exécutions dans le temps et une vue comparant les exécutions à l'outil Trace Compass comprenant déjà de nombreuses vues d'intérêt.

Nous pensons également être les premiers à avoir utilisé l'analyse du chemin critique d'une tâche pour repérer les inversions de priorité survenant sur l'ensemble du chemin critique. Il s'agit d'un complément intéressant à cette analyse pour les systèmes temps réel.

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

Ce chapitre conclut notre mémoire. On y présente une synthèse des travaux, les limitations de notre solution et les améliorations possibles.

6.1 Synthèse des travaux

Dans ce mémoire, nous avons étudié l'utilisation du traçage pour détecter des problèmes dans les systèmes temps réel. Le but était de développer un outil d'analyse de trace permettant de repérer des situations anormales de manière efficace.

Le premier objectif était d'identifier les informations pertinentes à collecter pour analyser les systèmes temps réel, ce que nous avons fait dans notre revue de littérature. On peut entre autres mentionner les informations d'ordonnancement, la priorité des fils d'exécution, les temps de réponse, les temps d'exécution et les informations sur la mémoire cache. Dans notre outil, nous nous sommes davantage concentrés sur les problèmes d'ordonnancement et de blocage et les informations reliées. Les tâches périodiques vont nécessairement se faire ordonnancer sous Linux et les problèmes occasionnés viennent dégrader les temps d'exécution qui sont particulièrement importants dans le temps réel.

Le deuxième objectif était de développer une méthode permettant à un utilisateur de définir le modèle d'exécution d'une tâche temps réel spécifique. Nous avons donc développé une interface graphique permettant de définir les événements de la trace devant être présents dans une exécution. De plus, nous avons ajouté de nombreuses fonctionnalités pour éditer facilement le modèle ainsi défini. Il est ainsi possible de repérer des exécutions se déroulant sur plusieurs fils d'exécution ou encore de définir le contenu des champs des événements.

Ensuite, nous sommes allés plus loin pour aider les utilisateurs à définir un modèle d'exécution en tentant de leur suggérer des choix intéressants. Pour ce faire, nous avons modifié un algorithme de recherche de patrons pour l'adapter aux traces d'exécution. Les résultats sont sous la forme d'une suite de définitions d'événements qui arrivent fréquemment en séquence. Il est ensuite possible de les charger dans l'interface présentée précédemment pour y apporter des modifications ou pour directement passer à la recherche des correspondances dans la trace.

Le troisième objectif était de développer une méthode permettant de repérer dans une trace les exécutions correspondant à un modèle défini. Pour y arriver, nous utilisons des machines à

état. Cela nous permet d'itérer une seule fois sur la trace pour récupérer toutes les occurrences du modèle.

Le quatrième objectif était de développer une méthode permettant de repérer si une exécution présente un problème. Nous avons d'abord tenté de localiser les inversions de priorité. Pour ce faire, nous avons utilisé l'analyse du chemin critique. Cela nous fournit les fils d'exécution ayant bloqué notre exécution. Il est ensuite possible d'obtenir les moments où ces fils d'exécution ont été préemptés. Puis, on peut obtenir les informations sur les fils d'exécution ayant causé ces préemptions. Il est finalement possible de comparer la priorité de ces derniers avec celle de notre exécution. Si elle est plus faible, cela signifie qu'il y a eu inversion de priorité.

Il aurait été intéressant de détecter automatiquement d'autres problèmes typiques . Pour le moment, il est cependant nécessaire de comparer visuellement les exécutions entre elles et d'utiliser les statistiques disponibles pour identifier la cause des autres problèmes. Il est également possible de se servir des autres vues disponibles dans Trace Compass avec lesquelles les vues développées sont synchronisées.

Le cinquième objectif était de présenter les exécutions problématiques dans une vue. Nous avons donc choisi de montrer les exécutions de deux manières. Premièrement, une vue de comparaison permet de voir les exécutions une par-dessus l'autre avec une même échelle de temps. Elles peuvent être triées par durée totale, temps de début, durée en exécution ou encore durée en préemption. Ensuite, une autre vue montre la durée totale des différentes exécutions selon leur temps de début de même que les moyennes et écarts-types des données. Cela permet de voir des patrons globaux selon la distribution des exécutions pouvant indiquer une perturbation ponctuelle ou périodique.

Le sixième objectif était de valider l'approche à partir de résultats expérimentaux. Nous avons donc généré de nombreux cas de test et utilisé des programmes industriels. Cela nous a permis d'établir les paramètres optimaux et les cas d'utilisation où notre outil apporte une aide essentielle.

6.2 Limitations de la solution proposée

Tout d'abord, certains types d'événements doivent être présents dans la trace. Il est ainsi essentiel d'activer les points de trace correspondants aux *sched_switch* puisque ce type d'événement est utilisé pour connaître les fils d'exécution s'exécutant sur les différents CPU de même que la priorité des fils d'exécution. Ensuite, l'événement *sched_pi_setprio* est nécessaire pour obtenir la priorité dans le cas d'un héritage de priorité. Il convient également de

mentionner que l'événement *sched_ttwu* permet à l'analyse du chemin critique d'établir les dépendances. Finalement, tous les événements définis dans le modèle par l'utilisateur doivent également être présents dans la trace pour obtenir des résultats.

Ensuite, l'algorithme de découverte de patrons permet d'identifier les patrons se déroulant lors de l'exécution d'un seul fil d'exécution. On peut d'abord mentionner que certains événements pertinents se produisent avant que le fil d'exécution s'exécute. Par exemple, on peut retrouver l'échéance d'un *timer* qui va réveiller un fil d'exécution et c'est seulement ensuite que celui-ci va commencer son exécution. Il serait donc nécessaire de venir ajuster le modèle avant de démarrer la recherche de l'ensemble des occurrences.

Une autre limitation est qu'il est seulement possible de définir les événements de début et de fin pour un modèle où ceux-ci se retrouvent sur des fils d'exécution différents. Cela est dû au fait qu'on ne sait pas d'avance sur quel fil d'exécution l'exécution va finir. Il est par exemple possible avec notre outil de chercher sur tous les fils d'exécution. Cela demanderait donc de conserver une machine à état pour chaque paire de fils d'exécution afin de valider les définitions intermédiaires.

On peut également mentionner les problèmes de mémoire qui ont été discutés dans notre article. Des limites concernant le nombre d'exécutions affichées ont été imposées afin de conserver une certaine fluidité dans l'affichage. De même, les informations sur les exécutions sont conservées en mémoire et peuvent donc poser problème. Dans le cas de trace immense où seules quelques exécutions sont problématiques sur par exemple des centaines de milliers, il aurait pu être intéressant de conserver après un certain seuil seulement les exécutions présentant certaines caractéristiques inhabituelles comme un temps d'exécution beaucoup plus long. Une autre solution aurait été d'écrire les données dans un fichier malgré le ralentissement occasionné par cette solution.

6.3 Améliorations futures

Premièrement, la détection automatique des tâches temps réel mérite d'être investiguée davantage. Lorsqu'un système comporte beaucoup de fils d'exécution, il peut d'abord être ardu de repérer ceux qui présentent de l'intérêt. Il serait possible d'utiliser des caractéristiques souvent présentes dans le temps réel comme une périodicité ou encore les priorités et les politiques d'ordonnancement afin de sélectionner les fils d'exécution susceptibles d'intéresser les utilisateurs.

Ensuite, il serait intéressant d'améliorer les suggestions de modèle faites. On peut notamment mentionner les événements se produisant quand le fil d'exécution ne s'exécute pas. Il

serait également pertinent de réduire les connaissances nécessaires afin de définir le modèle particulièrement au niveau des types d'événements générés par le noyau.

La présente solution pourrait d'ailleurs être raffinée pour faciliter son utilisation dans le cas où un utilisateur ne voudrait pas des fonctionnalités avancées. De plus, les analyses pourraient être généralisées. En effet, il serait intéressant de pouvoir personnaliser les événements utilisés par celles-ci afin de pouvoir utiliser l'outil avec différents traceurs et des structures personnalisées.

Finalement, le repérage de plusieurs problèmes présents dans les systèmes temps réel aurait intérêt à être ajouté à l'analyse. Par exemple, les cas de mauvaise utilisation de la mémoire cache pourraient être détectés. Ces nouvelles analyses s'intégreraient facilement à nos travaux et nous rapprocheraient encore davantage d'un outil permettant de trouver les problèmes de manière complètement automatique.

RÉFÉRENCES

D. F. Bacon, P. Cheng, D. Frampton, D. Grove, M. Hauswirth, et V. Rajan, "Demonstration : On-line visualization and analysis of real-time systems with tuningfork", dans *Compiler Construction*. Springer, 2006, pp. 96–100.

R. Beamonte, "Traçage de systèmes linux multi-coeurs en temps réel", Mémoire de maîtrise, École Polytechnique de Montréal, 2013.

C. Borgelt et R. Kruse, "Induction of association rules : Apriori implementation", dans *Compstat*. Springer, 2002, pp. 395–400.

F. Brandner, S. Hepp, et A. Jordan, "Criticality : static profiling for real-time programs", *Real-Time Systems*, vol. 50, no. 3, pp. 377–410, 2014.

R. J. Bril, J. J. Lukkien, et W. F. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption", *Real-Time Systems*, vol. 42, no. 1-3, pp. 63–119, 2009.

Y. Brosseau. (2011) A userspace tracing comparison : Dtrace vs lttng ust. En ligne : `http://www.dorsal.polymtl.ca/fr/blog/yannick-brosseau/userspace-tracing-comparison-dtrace-vs-lttng-ust`

A. Burns et A. J. Wellings, *Real-time systems and programming languages : Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

G. C. Buttazzo, M. Bertogna, et G. Yao, "Limited preemptive scheduling for real-time systems. a survey", *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 3–15, 2013.

G. Casas-Garriga, *Discovering unbounded episodes in sequential data*. Springer, 2003.

J. Corbet. (2009) Dynamic probes with ftrace. En ligne : `http://lwn.net/Articles/343766/`

——. (2012) Uprobes in 3.5. En ligne : `https://lwn.net/Articles/499190/`

——. (2013) Ktap — yet another kernel tracer. En ligne : `http://lwn.net/Articles/551314/`

D. de Niz, J. Hansson, J. Hudak, et P. H. Feiler, “3 performance challenges of modern hardware architectures for real-time systems”, *Results of SEI Independent Research and Development Projects*, p. 8, 2008.

M. Desnoyers et J. Desfossez. (2011) Lttng-ust vs systemtap userspace tracing benchmarks. En ligne : `http://lists.lttng.org/pipermail/lttng-dev/2011-February/003949.html`

O. M. dos Santos et A. Wellings, “Run time detection of blocking time violations in real-time systems”, dans *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA’08. 14th IEEE International Conference on.* IEEE, 2008, pp. 347–356.

EfficiOS. (2015) Common trace format (ctf). En ligne : `http://www.efficios.com/ctf`

S. Fatehpuria et A. Goyal, “A very unique, fast and efficient approach for pattern matching (the jumping algorithm)”, dans *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on.* IEEE, 2014, pp. 1241–1245.

A. Garcia-Martinez, J. F. Conde, et A. Vina, “A comprehensive approach in performance evaluation for modern real-time operating systems”, dans *EUROMICRO 96. Beyond 2000 : Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference.* IEEE, 1996, pp. 61–68.

B. Gregg. (2015) perf examples. En ligne : `http://www.brendangregg.com/perf.html`

GWT-TUD GmbH. (2015) Vampir - performance optimization. En ligne : `https://www.vampir.eu/`

T. Harmon et R. Klefstad, “Automatic performance visualization of distributed real-time systems”, dans *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on.* IEEE, 2006, pp. 8–pp.

T. Helmy et S. S. Jafri, “Avoidance of priority inversion in real time systems based on resource restoration.” *IJCSA*, vol. 3, no. 1, pp. 40–50, 2006.

N. Hillary, “Measuring performance for real-time systems”, *Freescale Semiconductor, November*, 2005.

IBM Research. (2015) Zinsight - ibm mvs system trace analyzer. En ligne : `http://researcher.watson.ibm.com/researcher/view_group.php?id=613`

D. Isovic et G. Fohler, “Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints”, dans *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE.* IEEE, 2000, pp. 207–216.

M. Kerrisk, P. Zijlstra, et J. Lelli. (2014) sched - overview of scheduling apis. En ligne : `http://man7.org/linux/man-pages/man7/sched.7.html`

C. LaRosa, L. Xiong, et K. Mandelberg, “Frequent pattern mining for kernel trace data”, dans *Proceedings of the 2008 ACM symposium on Applied computing.* ACM, 2008, pp. 880–885.

S. Laxman, P. Sastry, et K. Unnikrishnan, “A fast algorithm for finding frequent episodes in event streams”, dans *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2007, pp. 410–419.

P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, et M. Santana, “Debugging embedded multimedia application traces through periodic pattern mining”, dans *Proceedings of the tenth ACM international conference on Embedded software.* ACM, 2012, pp. 13–22.

H. Mannila *et al.*, “Disocvering frequent episodes in event sequences”, dans *Proc. First Conf. on Knowledge Discovery and Data Mining (KDD95)*, 1995.

H. Mannila et H. Toivonen, “Discovering generalized episodes using minimal occurrences.” dans *KDD*, vol. 96, 1996, pp. 146–151.

P. McKenney. (2005) A realtime preemption overview. En ligne : `https://lwn.net/Articles/146861/`

A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data”, dans *Runtime Verification.* Springer, 2013, pp. 219–234.

Paradyn Project. (2015) Paradyn tools project. En ligne : `http://www.paradyn.org/index.html`

S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, et M. G. Harbour, “Influence of different system abstractions on the performance analysis of distributed real-time systems”, dans *Proceedings of the 7th ACM & IEEE international conference on Embedded software.* ACM, 2007, pp. 193–202.

Percepio. (2015) Tracealyser. En ligne : `http://percepio.com/tz/#Views`

POSIX Programmer's Manual. (2003) pthread_mutexattr_setprotocol(3) - linux man page. En ligne : http://linux.die.net/man/3/pthread_mutexattr_setprotocol

QNX Software Systems. (2014) Analyzing your system with kernel tracing. En ligne : http://www.qnx.com/developers/docs/6.3.0SP3/ide_en/user_guide/sysprof.html

F. Rajotte, "Analyse de systèmes temps-réel par traçage", Mémoire de maîtrise, École Polytechnique de Montréal, 2014.

Rapita Systems. (2015) Rapitime. En ligne : http://www.rapitasystems.com/products/rapitime

S. Rostedt. (2008) ftrace - function tracer. En ligne : https://www.kernel.org/doc/Documentation/trace/ftrace.txt

——. (2010) Using the trace_event macro(). En ligne : http://lwn.net/Articles/379903/

——. (2011) Using kernelshark to analyze the real-time scheduler. En ligne : https://lwn.net/Articles/425583/

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

B. Srinivasan, S. Pather, R. Hill, F. Ansari, et D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software", dans *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE.* IEEE, 1998, pp. 112–119.

Sysdig Cloud. (2015) Sysdig examples. En ligne : http://www.sysdig.org/wiki/sysdig-examples/

SystemTap Editor Group. (2015) Systemtap wiki. En ligne : https://sourceware.org/systemtap/wiki

A. Terrasa et G. Bernat, "Extracting temporal properties from real-time systems by automatic tracing analysis", dans *Real-Time and Embedded Computing Systems and Applications*. Springer, 2004, pp. 466–485.

The LTTng Project. (2014) The lttng documentation. En ligne : http://lttng.org/docs/

Trace Compass. (2015) Trace compass. En ligne : https://projects.eclipse.org/projects/tools.tracecompass

H. Waly, “Automated fault identification : Kernel trace analysis”, Thèse de doctorat, Université Laval, 2011.

Wind River Systems. (2015) Windview 2.0. En ligne : http://read.pudn.com/downloads37/sourcecode/embed/125286/START/06-WindView2.0.pdf

Y. Wu, S. Fu, H. Jiang, et X. Wu, “Strict approximate pattern matching with general gaps”, *Applied Intelligence*, pp. 1–15, 2014.

E. Zannoni et K. Van Hees. (2012) Dtrace on linux. En ligne : https://events.linuxfoundation.org/images/stories/pdf/lfcs2012_zannoni_hees.pdf

H. Zhu, P. Wang, X. He, Y. Li, W. Wang, et B. Shi, “Efficient episode mining with minimal and non-overlapping occurrences”, dans *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1211–1216.