Titre: Title:	Analyse de systèmes temps-réel par traçage
Auteur: Author:	François Rajotte
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rajotte, F. (2014). Analyse de systèmes temps-réel par traçage [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1388/

UNIVERSITÉ DE MONTRÉAL

ANALYSE DE SYSTÈMES TEMPS-RÉEL PAR TRAÇAGE

FRANÇOIS RAJOTTE
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)
AVRIL 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DE SYSTÈMES TEMPS-RÉEL PAR TRAÇAGE

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

M. LANGLOIS J.M. Pierre, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. BELTRAME Giovanni, Ph.D., membre

À mes parents,
qui m'ont toujours supporté.

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche, Michel Dagenais, pour son support, du début à la fin du projet. Il a su inculquer en moi sa passion pour la recherche et la quête de connaissance.

J'aimerais aussi remercier tous ceux qui ont apporté un soutien financier à mon projet de recherche : CAE, OPAL-RT, 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 ne voudrais surtout pas oublier mes collègues du laboratoire DORSAL. Grâce à eux, l'ambiance de travail a toujours été fraternelle et propice au partage de connaissances. Un merci tout particulier, aussi, à mes voisins de bureau, Simon, Adrien et Raphaël, avec qui j'ai partagé bien de bons moments.

Finalement, j'aimerais remercier mes parents et amis qui sont toujours restés à mes côtés et qui ont fait de moi qui je suis.

RÉSUMÉ

Le traçage est une technique qui permet de récupérer de l'information très précise sur l'exécution d'un système avec un impact minime. Afin de mieux comprendre l'exécution d'une application, la technique habituelle consiste à y attacher un débogueur et interrompre l'application, afin d'inspecter la valeurs de certaines variables, par exemple. Cette approche est mal adaptée aux applications qui ont des interactions fréquentes avec le système lui-même ou avec d'autres applications. Les applications temps réel sont un type d'application qui possède ce genre d'interactions. L'aspect temporel de leur exécution rend l'utilisation d'un débogueur inutile dans plusieurs cas. L'impact minimal du traçage sur l'exécution d'une application lui confère un atout important pour mieux comprendre les interactions complexes qui peuvent agir au sein d'un système temps réel. Afin de minimiser l'impact de l'instrumentation, il est souhaitable de réduire la quantité d'information récupérée. Il est donc important de bien identifier l'information minimale nécessaire à l'analyse, qui ne causera pas de latences indues. Dans un même ordre d'idées, le traceur ne peut pas se permettre d'effectuer un traitement coûteux avant l'enregistrement des informations. Les événements récupérés devront donc contenir une information brute sur l'exécution du système.

L'objectif de cette recherche est de montrer que l'information récupérée lors du traçage d'un système temps réel peut être utilisée pour extraire de l'information permettant de mieux comprendre des comportements propres aux systèmes temps réel.

L'hypothèse de ce travail est que le traçage permet de récupérer de l'information sur l'exécution d'une application temps réel et que les informations de traçage ainsi récupérées peuvent être utilisées pour diagnostiquer des problèmes difficilement observables.

Nous étudions d'abord les différents outils de traçage en fonction de leurs fonctionalités et de leur impact sur les systèmes temps réel. Ensuite, nous comparons les différents outils d'analyse de trace selon deux grandes catégories : les approches algorithmiques et les techniques de visualisation.

Des problèmes typiques des applications temps réel sont initialement identifés et serviront de base pour guider notre recherche. Un algorithme est développé afin d'analyser la trace et retrouver ces problèmes typiques. L'algorithme est testé sur des traces générées à partir de cas de tests et sa performance sera évaluée.

La première contribution de ce travail consiste en la mise au point d'un algorithme permettant de générer un modèle à haut niveau d'une application temps réel à partir de la trace de son interaction avec le noyau du système d'exploitation. Ce modèle utilise la sémantique particulière des événements afin de produire une machine à états simple et rapide. Les évé-

nements nécessaires et minimaux à l'analyse sont identifiés de façon à limiter l'impact du traçage sur l'application. La deuxième contribution consiste en l'élaboration d'un outil de visualisation permettant de comparer directement les différentes phases d'exécution d'une application temps réel. Cet outil utilise le modèle généré à partir des informations de traçage afin d'identifier les différentes phases et d'extraire plusieurs statistiques utiles à la compréhension globale de l'exécution. Finalement, une structure de stockage des statistiques est améliorée de façon à récupérer efficacement des statistiques qui évoluent de façon continue dans le temps.

Le résultat final est un outil qui permet de diagnostiquer les problèmes des applications temps réel grâce aux informations contenues dans une trace noyau, en plus de faciliter la découverte de patrons d'intérêt.

ABSTRACT

Tracing is a technique to gather precise information about the execution of a system with minimal impact. In order to better understand the execution of an application, the usual technique consists in attaching a debugger and interrupting the application, to inspect the value of certain variables, for example. This approach is ill-suited for applications that are tightly coupled with the system itself. Real-time applications are a type of applications that exhibit this sort of interactions. The temporal aspect of their execution nullifies the use of a debugger. The low impact of a tracer on the execution of an application is therefore an important aspect providing better understanding of complex interactions in real-time systems. Even then, it is important to minimize even further the impact of tracing by reducing the amount of gathered information. It is therefore important to identify the minimal information that is necessary for the analysis that will not cause undue latency. Similarly, the tracer cannot afford complex processing while collecting the information. It must write the raw events as fast as possible.

The objective of this research is to show that the information gathered during the tracing of a real-time system can be used to extract additional information that can be used to better understand the behaviour of real-time systems.

We will first study the different tracing tools according to their functionalities and impact on real-time systems. Then, we will compare different trace analysis tools according to two main categories : algorithmic approaches and visualization techniques.

Typical real-time application problems will be identified and used as a baseline to guide our research. An algorithm will then be developed to analyse the trace and find these typical problems. The algorithm will be tested on traces generated from test cases and its performance evaluated.

The hypothesis of this work is that tracing allows gathering information about the execution of real-time applications and that this tracing information can be used to diagnose problems that are otherwise difficult to observe.

The result of this work is the creation of a model allowing the extraction of statistics and the generation of visualizations from kernel traces gathered on a real-time system. This model uses event semantics to produce a finite-state machine that is both simple and fast. The minimal and necessary events for the analysis are identified in order to limit the impact of tracing on the application. Finally, a statistics storage structure is improved in order to retrieve efficiently statistics that are continuously variable through time.

The final result is a tool to diagnose real-time application problems using the information stored inside a kernel trace and aid in the discovery of interesting patterns.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CTF	Common Trace Format
GDB	GNU Debugger
LTTng	Linux Trace Toolkit next generation
LTTV	Linux Trace Toolkit Viewer
PID	Process identifier
RCU	Read-Copy-Update
TID	Thread identifier
TMF	Tracing and Monitoring Framework
URCU	Userspace Read-Copy-Update

CHAPITRE 1

INTRODUCTION

Les systèmes temps réel sont, par leur nature même, difficiles à surveiller et à observer. Les contraintes temporelles auxquelles ils sont soumis rendent toute instrumentation susceptible de perturber significativement leur comportement. Ces perturbations peuvent rendre le système inopérant ou encore rendre l'observation du problème d'intérêt impossible. Le traçage présente une solution intéressante qui permet de récupérer de l'information sur le déroulement d'une application temps réel tout en minimisant l'impact de l'instrumentation sur le système. Contrairement au débogage qui permet l'inspection interactive du comportement d'une application, le traçage récupère la totalité de l'information sur le déroulement du système. Ces traces peuvent donc contenir une grande quantité d'événements et des outils spécialisés doivent être utilisés afin d'extraire l'information pertinente à la résolution d'un problème particulier.

1.1 Définitions et concepts de base

Dans cette section, les concepts utilisés dans ce mémoire sont présentés et définis.

1.1.1 Le temps réel

Les applications temps réel

Dans un système à usage général, une application est jugée correcte si la logique qu'elle contient génère les bonnes sorties en fonction d'une certaine entrée. En plus de cette contrainte de correctitude, les applications temps réel doivent également obéir à des contraintes temporelles. Le résultat du traitement doit être prêt à l'intérieur d'un délai limite, ou échéance (*deadline*). L'ensemble des échéances de toutes les applications forme les contraintes temporelles du système. Le manquement d'une échéance entraîne le non-respect des contraintes. Les systèmes temps réel sont séparés en deux catégories selon l'impact que cause le dépassement d'une échéance.

– Les systèmes temps réel à contraintes dures dépendent du respect des échéances à tout moment. Le non-respect d'une contrainte entraîne une défaillance du système. Afin de maintenir le bon fonctionnement du système, un déterminisme accru est nécessaire. Ce genre de système est commun dans les applications touchant la sécurité du public.

– Les systèmes temps réel à contraintes souples, quant à eux, ne dépendent pas absolument du respect des contraintes temporelles. Le dépassement d'une échéance entraîne la dégradation du service mais le fonctionnement global du système n'est pas compromis. Par exemple, le décodage et l'affichage d'un flux vidéo doit fournir une nouvelle trame à un intervalle régulier. Si une trame n'est pas décodée à temps, l'affichage du flux vidéo sera interrompu brièvement mais la lecture pourra continuer par la suite.

Les tâches temps réel

Une tâche temps réel consiste en une entité qui s'acquitte d'une seule responsabilité au sein d'une application. Chaque tâche doit répondre à un stimulus à l'intérieur d'un délai. Ce stimulus peut provenir d'une autre tâche de l'application, du système lui-même ou éventuellement de l'extérieur du système. À son tour, une tâche peut alors entraîner le déclenchement d'une autre tâche.

Lors de la mise en place d'une application temps réel, une analyse théorique d'ordonnancement permet de vérifier que les tâches sont ordonnançables, c'est-à-dire qu'il existe une façon d'exécuter les tâches pour qu'elles respectent toutes leurs contraintes temporelles. À cette fin, on distingue deux grands types de tâches selon la fréquence à laquelle elles sont déclenchées :

– Les tâches périodiques sont déclenchées à intervalle régulier. Le déclenchement de la tâche provient alors d'une source qui est elle-même périodique, par exemple une autre tâche périodique, une minuterie ou alors un événement extérieur qui est lui-même périodique.
– Les tâches sporadiques ne possèdent pas de période régulière selon laquelle elles sont déclenchées. Pour ces tâches, un temps minimum entre deux déclenchements est plutôt défini. Sans la définition de ce temps minimum, une tâche sporadique pourrait se déclencher continuellement et interdire aux autres tâches la chance de s'exécuter. Les tâches sporadiques peuvent se charger de répondre à des événements fréquents comme le changement des paramètres du système, lors de l'appui d'un bouton, ou alors à des événements rare comme un senseur dont la valeur quitte la plage de valeurs normales.

1.1.2 Les systèmes d'exploitation

Les primitives de synchronisation

Les processus s'exécutent généralement en symbiose avec d'autres processus. Dans les systèmes multi-processeurs, les échanges entre processus doivent être protégés afin d'éviter qu'un processus n'accède à une donnée partiellement écrite par un processus qui s'exécute

sur un autre processeur. Dans un système d'exploitation préemptif, le même problème peut survenir sur un seul processeur. Afin d'éviter ces erreurs, des sections de code sont protégées à l'aide de primitives de synchronisation. Le sémaphore permet de limiter l'accès à un nombre limité de ressources à l'aide d'un compteur interne. Le mutex limite à un seul processus l'accès à une section de code. Puisqu'un seul processus peut entrer dans une section protégée par le même mutex, on dit que le processus est alors en possession du mutex.

La préemption

L'ordonnanceur peut décider d'interrompre l'exécution d'une tâche afin de donner le processeur à une autre tâche. Dans le cas des systèmes à usage général, la préemption permet d'assurer que tous les processus aient la chance de s'exécuter en un temps raisonnable et qu'un processus n'utilise pas le système en entier à lui seul. Pour les systèmes temps réel, la préemption est nécessaire afin de s'assurer que les tâches urgentes puissent s'exécuter sans délai.

L'ordonnanceur

Lorsque plusieurs tâches veulent s'exécuter en même temps sur un système, l'ordonnanceur décide quelles tâches auront cette chance en premier. Plusieurs stratégies d'ordonnancement existent selon l'utilisation prévue du système. Dans le cas des systèmes temps réel, l'ordonnanceur doit être conçu de façon à s'assurer que toutes les tâches s'exécutent à l'intérieur des délais prévus. Un ordonnancement statique consiste à définir à l'avance l'ordre dans lequel les processus s'exécutent. Une technique plus flexible consiste à laisser l'ordonnanceur choisir les processus à exécuter selon certains critères. Une stratégie simple d'ordonnancement dynamique consiste à attribuer des niveaux de priorité aux processus du système. L'ordonnanceur exécute alors les tâches à priorité élevée en premier.

1.2 Éléments de la problématique

Un intérêt grandissant se manifeste autour de l'utilisation du noyau Linux pour des applications temps réel. Les systèmes d'exploitation basés sur GNU/Linux sont largement utilisés et possèdent une grande communauté active. Il n'est donc pas surprenant que des membres de la communauté veuillent utiliser ce système d'exploitation pour exécuter leurs applications temps réel. Le correctif `PREEMPT_RT` a été créé afin d'améliorer les performances temps réel du noyau Linux. En particulier, ce correctif réduit les périodes où la préemption est désactivée. Au cours des années, plusieurs des modifications de ce correctif ont été incorporées dans le noyau linux de base.

De par leur nature même, les applications temps réel sont difficilement observables. Les techniques de débogage traditionnelles utilisent des points d'interruption pour donner au développeur la chance d'inspecter le comportement fonctionnel d'une application. Pour le cas des applications temps réel, il ne suffit pas de vérifier la logique interne du code. Il faut également s'assurer que les contraintes temporelles soient respectées. Afin d'observer le comportement temporel d'une application, il faut utiliser des outils qui ne pertubent pas l'exécution et qui conservent le déterminisme de l'application.

Le traçage présente une solution intéressante à ce problème. En plaçant des points de trace à des endroits stratégiques dans le code de l'application, il est possible de récupérer de l'information sur son exécution. Un traceur à faible impact permet de générer ces événements avec un effet mineur sur la latence subie par l'application. En utilisant un traceur en espace noyau, il est aussi possible d'observer les interactions d'une application avec le système d'exploitation et les autres processus du système.

Le traçage est déjà utilisé pour récupérer des informations précises sur le minutage d'une application. En plaçant des points de trace aux branchements dans le code d'une application, il est possible de mesurer précisément le temps d'exécution de chaque segment de code. Lors de la conception d'une application, cette information permet de mieux prédire le temps d'exécution dans le pire cas et donc de garantir un meilleur déterminisme. Cette approche est cependant mal adaptée à l'observation des interactions entre les applications et le système.

Afin d'observer ces interactions, les points de trace doivent contenir davantage d'informations que l'information de minutage à certains endroits du code d'une application. Cependant, à cause des contraintes de déterminisme et de faible latence imposées par l'application temps réel, le prétraitement à l'intérieur du point de trace doit être minimal. Une analyse plus poussée doit donc être effectuée une fois la trace extraite du système. Avec suffisamment de points de trace, les événements récupérés à des instants ponctuels peuvent ainsi être utilisés pour regénérer l'état du système à n'importe quel autre moment de la trace.

1.3 Objectifs de recherche

Maintenant que la problématique est établie, il est possible de définir les objectifs de recherche.

- Identifier les concepts propres aux systèmes temps réel qu'il serait intéressant d'extraire en utilisant le traçage.
- Déterminer les événements de trace pertinents qui permettent de caractériser un système temps réel.

– Développer un algorithme permettant d'extraire les concepts aidant à la compréhension d'un système temps réel.
– Générer des traces expérimentalement de façon à pouvoir tester les capacités et l'efficacité de l'algorithme d'analyse.

L'objectif global est de créer un outil qui permette de faciliter l'exploration et la compréhension de traces récupérées sur un système temps réel. Cet outil devra être en mesure de s'intégrer aux outils déjà existants tout en ayant une performance suffisante pour conserver l'interactivité de l'application d'analyse.

1.4 Plan du mémoire

Une revue de littérature est présentée au chapitre 2. Celle-ci présente l'état de l'art dans le domaine de l'analyse de trace et des différents traceurs. Le chapitre 3 présente l'ensemble des particularités de l'approche présentée dans l'article au chapitre 4. Par la suite, des résultats complémentaires seront présentés au chapitre 5. Ceux-ci contiennent certaines améliorations particulières qui n'ont pas été abordées au chapitre 4. Au chapitre 6, une discussion mettra en relation les éléments importants des travaux. Finalement, une conclusion est présentée au chapitre 7 et fait une synthèse des travaux en plus d'inclure des pistes de réflexion vers des travaux futurs.

CHAPITRE 2

REVUE DE LITTÉRATURE

Cette section a pour objectif de présenter l'état de l'art des différents outils de traçage disponibles pour Linux et les problèmes qu'ils permettent de détecter. De plus, des techniques d'analyse plus poussées sont présentées sous deux grands thèmes : la détection automatique de patrons et les approches visuelles d'exploration de traces.

2.1 Le traçage

Le traçage permet de récupérer de l'information sur l'exécution d'un programme sans devoir interrompre son flot d'exécution. Des points de trace sont insérés à des endroits stratégiques où il est possible d'enregistrer des valeurs d'intérêt et de les conserver pour une analyse future. Dans cette section, les différentes stratégies d'insertion de points de trace seront abordées et les principaux outils de traçage sous Linux seront décrits.

2.1.1 Instrumentation statique

L'instrumentation statique consiste à insérer les points de trace directement dans le binaire de l'application lors de la compilation. Puisque l'instrumentation statique est toujours présente dans le fichier binaire, l'activation d'un point de trace est très rapide. Le désavantage d'une telle approche est qu'il faille recompiler le code pour définir et ajouter des points de trace additionnels. Dans le noyau Linux, la macro `TRACE_EVENT()` est utilisée pour ajouter des points de traces [1]. Celle-ci permet de définir un point de trace qui peut être activé et utilisé par n'importe quel traceur qui supporte son format.

2.1.2 Instrumentation dynamique

L'instrumentation dynamique permet d'insérer des points de trace dans un binaire sans devoir recompiler le code source, au prix d'une complexité plus grande et d'un surcoût plus élévé à l'exécution. Dans le noyau Linux, l'instrumentation dynamique est possible à l'aide de `kprobes` [2]. Ceux-ci fonctionnent en modifiant une instruction pour la remplacer par l'instruction `int3` qui permet l'interruption du flot d'exécution afin d'exécuter le code du point de trace. Au niveau de l'espace utilisateur, GDB est également en mesure d'insérer des points de trace dynamiquement [3]. L'instruction `int3` est alors utilisée pour interrompre

l'exécution du programme. Lorsque l'instruction remplacée est suffisamment grande, GDB peut insérer à la place un simple saut vers une autre partie du programme contenant le code du point de trace. Dyninst permet également d'insérer ce genre de points de trace rapides [4]. Il permet également de modifier le contenu d'un fichier binaire pour inclure un point de trace sans devoir le recompiler à partir de la source.

2.1.3 Outils de traçage pour Linux

SystemTap

SystemTap est un outil de traçage qui combine à la fois la génération de traces et l'analyse des résultats [5]. L'attrait principal de SystemTap est l'utilisation de scripts qui permettent de définir des actions lorsqu'un point de trace est atteint. Ces scripts sont compilés en modules noyau qui sont chargés au besoin. Le langage utilisé pour les scripts permet de facilement utiliser des fonctions d'aggrégation afin de calculer des statistiques qui pourront être affichées à l'utilisateur. Voici un exemple de script qui permet de mesurer les temps d'attente occasionés par l'utilisation du mécanisme de synchronisation de type `futex`.

```
#! /usr/bin/env stap

global FUTEX_WAIT = 0 /*, FUTEX_WAKE = 1 */
global FUTEX_PRIVATE_FLAG = 128 /* linux 2.6.22+ */
global FUTEX_CLOCK_REALTIME = 256 /* linux 2.6.29+ */

global lock_waits # long-lived stats on (tid,lock) blockage elapsed time
global process_names # long-lived pid-to-execname mapping

probe syscall.futex.return {
  if (($op & ~(FUTEX_PRIVATE_FLAG|FUTEX_CLOCK_REALTIME)) != FUTEX_WAIT) next
  process_names[pid()] = execname()
  elapsed = gettimeofday_us() - @entry(gettimeofday_us())
  lock_waits[pid(), $uaddr] <<< elapsed
}

probe end {
  foreach ([pid+, lock] in lock_waits)
    printf ("%s[%d] lock %p contended %d times, %d avg us\n",
            process_names[pid], pid, lock, @count(lock_waits[pid,lock]),
            @avg(lock_waits[pid,lock]))
}
```

Dans cet exemple, l'instrumentation est insérée à la sortie de l'appel système `futex` et enregistre la durée de l'appel dans un tableau indexé selon le processus appelant et l'adresse

du futex concerné. On observe également l'utilisation de fonctions d'aggrégation intégrées au langage (`@avg`, `@count`). Lors de l'exécution de ce script, le résultat est directement écrit à la console et a la forme suivante :

```
# stap futexes.stp
^C
liferea-bin[3613] lock 0x0a117340 contended 1 times, 71 avg us
java_vm[8155] lock 0x09baa45c contended 9 times, 1004013 avg us
java_vm[8155] lock 0x09d6f9ac contended 186 times, 55964 avg us
xmms[16738] lock 0xbfe29eec contended 777 times, 69 avg us
xmms[16738] lock 0xbfe29ecc contended 119 times, 64 avg us
xmms[16738] lock 0xbfe29ebc contended 111 times, 68 avg us
xmms[16738] lock 0xbfe29ef0 contended 742 times, 91 avg us
xmms[16738] lock 0xbfe29ed0 contended 107 times, 101 avg us
xmms[16738] lock 0xbfe29ec0 contended 107 times, 74 avg us
firefox-bin[21801] lock 0x096d16f4 contended 24 times, 12 avg us
firefox-bin[21801] lock 0x096d1198 contended 2 times, 4 avg us
firefox-bin[21801] lock 0x096d16f8 contended 150 times, 64997 avg us
named[23869] lock 0x41ab0b84 contended 1 times, 131 avg us
named[23869] lock 0x41ab0b50 contended 1 times, 26 avg us
```

SystemTap est un traceur noyau qui s'interface avec les macros `TRACE_EVENT()`. Il est également possible de définir des scripts à l'entrée ou à la sortie de fonctions. Dans ce cas, les points de trace dynamiques fournis par `kprobe` sont utilisés. Les processus en espace utilisateur peuvent également être tracés grâce aux points de trace dynamiques fournis par `uprobe` [6]. Ces points de trace permettent de transférer le contrôle de l'exécution au noyau lorsqu'il sont atteints.

À cause de l'utilisation de scripts qui s'exécutent pendant le traçage, SystemTap est davantage utile aux administrateurs qui veulent récupérer une statistique rapidement lorsqu'un problème particulier survient. Afin de conserver une latence faible lors de l'exécution, il serait souhaitable de repousser l'analyse de la trace après sa récolte. Cependant, il n'existe pas d'option pour différer l'exécution des scripts à un moment ultérieur.

Ftrace

Ftrace est un traceur intégré au noyau Linux dont l'objectif est de trouver des problèmes de performance à l'intérieur du noyau [7]. Il était conçu à l'origine comme traceur de fonctions mais il peut maintenant aussi s'attacher aux points de trace définis avec la macro `TRACE_EVENT()`. Afin d'utiliser le traceur de fonctions, le noyau doit être compilé avec l'option `CONFIG_FUNCTION_TRACER`. Cette option indique au compilateur d'ajouter un préambule

à l'entrée de chaque fonction. Ftrace peut s'attacher à ces préambules lors du traçage pour être averti lors de l'entrée d'une fonction.

Ftrace possède plusieurs configurations de traçage prédéfinies dans des modules d'analyse. Ces modules définissent les points de trace à activer afin d'effectuer une analyse particulière. L'analyse est également effectuée au fur et à mesure du traçage et le résultat est placé dans un fichier de résultats. Par exemple, l'analyse `irqsoff` permet de mesurer le temps maximal pendant lequel les interruptions étaient désactivées. La gestion de l'activation des divers modules d'analyse passe à travers le système de fichier debugfs. Un outil en ligne de commande est également disponible afin de faciliter l'utilisation.

Les résultats obtenus par Ftrace sont écrits dans des fichiers en texte brut dans un format facilitant la lecture. Chaque type d'analyse possède son propre format. Voici un exemple de fichier de résultat obtenu par l'analyse `sched_switch` :

```
# tracer: sched_switch
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-3997  [01]   240.132281:   3997:120:R   +  4055:120:R
            bash-3997  [01]   240.132284:   3997:120:R ==>  4055:120:R
           sleep-4055  [01]   240.132371:   4055:120:S ==>  3997:120:R
            bash-3997  [01]   240.132454:   3997:120:R   +  4055:120:S
            bash-3997  [01]   240.132457:   3997:120:R ==>  4055:120:R
           sleep-4055  [01]   240.132460:   4055:120:D ==>  3997:120:R
            bash-3997  [01]   240.132463:   3997:120:R   +  4055:120:D
            bash-3997  [01]   240.132465:   3997:120:R ==>  4055:120:R
          <idle>-0     [00]   240.132589:      0:140:R   +     4:115:S
          <idle>-0     [00]   240.132591:      0:140:R ==>     4:115:R
     ksoftirqd/0-4     [00]   240.132595:      4:115:S ==>     0:140:R
          <idle>-0     [00]   240.132598:      0:140:R   +     4:115:S
          <idle>-0     [00]   240.132599:      0:140:R ==>     4:115:R
     ksoftirqd/0-4     [00]   240.132603:      4:115:S ==>     0:140:R
           sleep-4055  [01]   240.133058:   4055:120:S ==>  3997:120:R
```

Cette analyse permet de lister tous les changements de contexte et réveils des processus du système. On remarque que le fichier est formatté afin de faciliter la lecture par un humain. Le désavantage de cette approche est que chaque module d'analyse utilise son propre format pour présenter les résultats et qu'il n'existe pas de parseur permettant de récupérer facilement l'information afin de réaliser des analyses supplémentaires. Étant intégré au noyau, il est également plus complexe d'écrire ses propres analyses.

Perf

Le traceur Perf est une autre traceur intégré au noyau Linux [8]. Les processeurs modernes possèdent des compteurs de performance. L'objectif original de Perf était de faciliter l'accès à ces compteurs. Depuis, Perf est également en mesure de s'interfacer avec les points de traces statiques et dynamiques du noyau.

Perf possède également un outil en ligne de commande qui s'interface avec le traceur. Cet outil permet de lancer le traçage d'une application selon plusieurs analyses prédéfinies. Les analyses de base retournent l'évolution des compteurs de performance pendant l'exécution d'un programme. Des analyses plus poussées utilisent l'échantillonnage afin d'obtenir des informations plus précises sur l'exécution d'un programme. Une interruption peut être déclenchée lorsqu'un compteur atteint une certaine valeur. Perf utilise cette interruption pour récolter de l'information lors de l'exécution d'un programme. En se connectant au compteur d'instruction, par exemple, perf peut identifier les fonctions dans lesquelles le plus d'instructions sont exécutées. Voici un extrait d'un rapport généré par perf, grâce à un échantillonage à tous les mille cycles d'horloge, et qui permet d'identifier les commandes les plus gourmandes en temps processeur :

```
# Events: 1K cycles
#
# Overhead          Command                    Shared Object
# ........  ...............  ...........................
#
   28.15%      firefox-bin  libxul.so
    4.45%           swapper  [kernel.kallsyms]
    4.26%           swapper  [kernel.kallsyms]
    2.13%      firefox-bin  firefox-bin
    1.40%  unity-panel-ser  libglib-2.0.so.0.2800.6
```

Perf offre une variété intéressante d'outils d'analyse. Cependant, les résultats offerts demeurent des statistiques de l'exécution globale d'un programme ou du système en entier. Ces statistiques peuvent être utiles pour détecter l'existence d'un problème mais, puisque les analyses ne permettent pas de creuser plus en profondeur, il est difficile de déterminer la raison de ce problème.

Paradyn

Paradyn est un outil d'analyse de performance axé sur les sytèmes parallèles [9]. Paradyn offre une solution de traçage complète allant de l'instrumentation jusqu'à l'affichage de vues pour analyse. L'instrumentation est effectuée grâce à Dyninst. Grâce à l'instrumentation dynamique offerte par Dyninst, Paradyn peut ajouter davantage de points de trace lorsqu'un

problème de performance est détecté, afin de récupérer plus d'information. Une fois les observations complétées, les points de trace peuvent être retirés afin de réduire l'impact du traçage sur le système.

Les traces récupérées ne sont pas écrites sur disque mais plutôt immédiatement envoyées au module d'analyse qui se charge d'extraire les statistiques. Cette approche permet de récupérer les statistiques de plusieurs systèmes tout en limitant la quantité d'information totale à analyser. Paradyn utilise par la suite les statistiques récupérées afin de guider le traçage en insérant des points de trace au besoin.

L'instrumentation offerte par Paradyn n'est cependant pas appropriée pour l'analyse d'un système temps réel. En effet, l'ajout et le retrait de points de trace dynamiquement cause des délais dans l'exécution, qui sont incompatibles avec le déterminisme attendu pour les applications temps réel. De plus, l'analyse automatique ne permet pas d'inspecter avec détail la trace et d'observer les interactions complexes entre les applications.

Le traceur `LTTng`

Linux Trace Toolkit next generation (LTTng) est un traceur dont l'architecture a été conçue afin de minimiser son impact sur le système traçé [10]. LTTng possède à la fois un traceur en espace noyau et un traceur en espace utilisateur. Un outil commun, `lttng-tools`, permet de contrôler ces deux traceurs à partir de la ligne de commande. Contrairement à SystemTap et Ftrace qui sont des parties intégrantes du noyau Linux, le traceur noyau de LTTng est un module qui est chargé lors du démarrage des outils de traçage. En plus de supporter les points de trace statiques définis avec la macro `TRACE_EVENT()`, LTTng supporte aussi l'activation des points de trace définis avec `kprobes`.

Puisque LTTng cherche à diminuer au maximum son impact sur le système, il utilise une architecture qui minimise le blocage dû à la synchronisation. Des tampons circulaires sont alloués sur chaque processeur de façon à éliminer le partage d'espaces mémoire entre les processeurs. Lorsque des échanges doivent être faits, des structures de données Read-Copy-Update (RCU) sont utilisées encore une fois pour éliminer les sources potentielles de blocage. Le mécanisme RCU permet à un processus écrivain de modifier une structure de données sans qu'un lecteur potentiel en soit conscient. Lorsqu'un écrivain modifie la structure de données, les anciennes valeurs ne sont pas encore effacées. En effet, le lecteur conserve une ancienne version qui sera éventuellement réclamée lorsque tous les lecteurs l'auront libérée.

D'une même façon, le traceur en espace utilisateur évite les blocages en utilisant des tampons qui sont alloués pour chaque processus. La librairie Userspace Read-Copy-Update (URCU) est également utilisée pour l'échange de structures plus complexes. Cette librairie fonctionne sous le même principe que le mécanisme RCU mais est implémentée entièrement

en espace utilisateur et ne nécessite pas d'appels au noyau. Grâce à cela, l'interaction avec le noyau est limitée aux demandes d'écriture lorsqu'un tampon est plein.

La performance de LTTng a été évaluée dans un environnement temps réel [11]. Dans cette étude, une application a été développée afin de mesurer la latence ajoutée par le traçage. Cette application exécute une boucle rapide contenant un point de trace. Toute latence observée entre deux tours de boucle provient donc du délai causé par le point de trace. Les résultats obtenus par cette étude montrent que les latences causées par le traceur LTTng sont inférieures à 30 microsecondes. Une modification proposée au traceur en espace utilisateur permet de réduire davantage la latence observée. Au lieu que l'application indique au traceur qu'un tampon est plein, le traceur vide lui-même les tampons périodiquement. Cette modification réduit le couplage entre l'application et le traceur et permet donc d'abaisser davantage l'impact du traceur sur l'application.

Les traces récoltées par LTTng sont enregistrées dans le format Common Trace Format (CTF). La spécification CTF permet à une trace adhérant à ce format de décrire elle-même sa structure [12]. Ainsi, un lecteur de trace qui implémente la spécification CTF peut lire n'importe quelle trace CTF, peu importe son origine. L'outil `babeltrace` est utilisé pour imprimer le contenu d'une trace à la console. Cet outil permet la lecture directe de la trace mais n'effectue pas d'analyses plus poussées. Voici un exemple de la sortie du programme `babeltrace` :

```
[14:37:18.889020245] (+0.000002168) sched_switch: { cpu_id = 0 }, {
   prev_comm = "irq/19-uhci_hcd", prev_tid = 80, prev_prio = -78,
   prev_state = 1, next_comm = "irq/19-ata_piix", next_tid = 65, next_prio
    = -51 }
[14:37:18.889027913] (+0.000007668) softirq_raise: { cpu_id = 0 }, { vec =
    4 }
[14:37:18.889028297] (+0.000000384) sched_wakeup: { cpu_id = 0 }, { comm =
    "ksoftirqd/0", tid = 3, prio = 98, success = 1, target_cpu = 0 }
[14:37:18.889029371] (+0.000001074) softirq_entry: { cpu_id = 0 }, { vec =
    4 }
[14:37:18.889030410] (+0.000001039) block_rq_complete: { cpu_id = 0 }, {
   dev = 8388608, sector = 940481696, nr_sector = 8, errors = 0, rwbs =
   17, _cmd_length = 0, cmd = [ ] }
[14:37:18.889033946] (+0.000003536) sched_stat_iowait: { cpu_id = 0 }, {
   comm = "lttng-consumerd", tid = 3387, delay = 24598 }
[14:37:18.889034376] (+0.000000430) sched_wakeup: { cpu_id = 0 }, { comm =
    "lttng-consumerd", tid = 3387, prio = 120, success = 1, target_cpu = 3
    }
```

La librairie `libbabeltrace` est également fournie pour permettre à d'autres programmes de lire les traces LTTng. Le Linux Trace Toolkit Viewer (LTTV) est un programme léger qui

utilise `libbabeltrace` pour afficher les traces dans une interface graphique. Le Tracing and Monitoring Framework (TMF), quant à lui, est un autre visualisateur de traces LTTng, mais est basé sur la plateforme Eclipse. Ces deux visualisateurs permettent de naviguer librement dans une trace en affichant une vue représentant l'état des processus tracés en fonction du temps.

La suite de vérification Rapita

La suite d'outils offerte par Rapita Systems fournit à la fois la solution de traçage ainsi que deux outils d'analyse [13]. L'objectif de cette suite d'outils est de fournir les informations de minutage d'une application temps réel afin de caractériser son comportement temporel dans divers cas d'utilisation. Grâce à ces informations de minutage, une analyse plus précise du pire temps d'exécution peut être réalisée.

La solution de traçage est compatible avec plusieurs systèmes d'exploitation temps réel. En plus d'offrir une solution implémentée complètement en logiciel, les traces peuvent également être récupérées en utilisant les composants matériels trouvés sur certains processeurs et ainsi réduire le surcoût du traçage. Il est également possible d'utiliser un simulateur du système testé afin de récupérer les informations de traçage.

Les points de trace sont automatiquement insérés lors de la compilation aux points de branchement de l'application. Chaque point de trace permet de récupérer une estampille de temps au moment où il est atteint ainsi qu'un identifiant unique de ce point de trace. Les traces ainsi récoltées fournissent l'information nécessaire pour deux outils d'analyses : RapiTime et RapiCover.

L'outil d'analyse RapiTime utilise l'information de minutage contenue dans la trace afin d'extraire le pire temps d'exécution de chaque segment de code de l'application. Un modèle du code de l'application est utilisé afin de combiner le pire temps d'exécution de plusieurs segments et obtenir le pire temps global, même si toutes les conditions menant à ce pire cas n'ont pas été rencontrées lors d'un même cas de test. Malgré cet avantage, la qualité du résultat obtenu demeure dépendante de l'exhaustivité des cas de test utilisés.

L'outil d'analyse RapiCover, quant à lui, vérifie quelles sections de code ont été ou n'ont pas été exécutées par un certain banc d'essai. En plus de vérifier que chaque ligne de code est exécutée, cet outil permet aussi de vérifier que toutes les possibilités d'une condition sont bien rencontrées au moins une fois.

2.2 La découverte de patrons

Une trace peut facilement contenir des millions d'événements. Il n'est pas simple de trouver les événements d'intérêt manuellement. Plusieurs algorithmes existent pour analyser ce genre d'information et extraire automatiquement les séquences d'intérêt. La découverte de patrons consiste à trouver des relations entre les éléments qui sont vraies sur l'ensemble d'un jeu de données.

Dans le cas de base, le jeu de données consiste en une liste d'éléments appelés items déjà groupés en transactions indépendantes. Dans ce cas, la recherche de patrons consiste à trouver des relations entre les items qui sont à l'intérieur d'une même transaction.

Dans un autre cas, les éléments ne sont pas groupés en transaction mais sont plutôt ordonnés linéairement à l'intérieur d'une séquence unique. Dans ce cas, il n'existe pas de groupement privilégié entre les éléments d'une séquence et la nature des patrons retrouvés est variable selon la technique de groupement utilisée.

2.2.1 Les ensembles d'items les plus fréquents

Considérons une base de données contenant des transactions. Chacune de ces transactions contient un certain nombre d'items. La recherche des ensembles d'items les plus fréquents consiste à trouver les items qui sont fréquemment retrouvés dans les mêmes transactions [14].

L'algorithme *Apriori*

L'algorithme *Apriori* [15] a été décrit pour la première fois pour découvrir des règles d'association entre des items d'une base de donnée. Ces règles d'association permettent de décrire la présence d'un certain type d'items étant donné la présence d'un autre item dans le même groupe d'items. Afin de pouvoir émettre ces règles, la première étape consiste à trouver les items qui surviennent souvent en même temps. L'algorithme *Apriori* permet simplement de déterminer les groupes d'items qui surviennent souvent en même temps. Le principe de cet algorithme est basé sur une intuition assez simple : des groupes d'items sont fréquents seulement si les items les composant sont eux-aussi fréquent. L'algorithme *Apriori* se divise en deux phases qui se répètent pour former des groupes d'items de plus en plus gros. La première étape consiste à générer les candidats à considérer. Ceux-ci sont formés des groupes d'items obtenus de l'itération précédente auxquels sont ajoutés un nouvel item à considérer. La deuxième étape consiste à vérifier le support de chaque candidat en effectuant une lecture de la base de données. Lorsque, pendant une itération, tous les candidats sont rejetés, l'algorithme se termine.

L'algorithme *FP-growth*

L'algorithme FP-growth a été développé afin d'accélérer la génération de candidats de l'algorithme Apriori. En effet, la liste de candidats peut devenir très grande et à chaque nouvelle liste de candidats il est nécessaire d'effectuer une nouvelle passe de la base de donnée. L'algorithme FP-growth a été créé afin d'éliminer l'étape de génération de candidats [16]. Dans cette approche, une structure est créée afin de contenir l'ensemble des items fréquents. Cette structure présente les items fréquents sous forme d'arbre avec les items les plus fréquents près de la racine. Ainsi, en parcourant cet arbre, on retrouve les items fréquents de chaque transaction originale. Des arbres secondaires sont créés pour chaque item et ces arbres résultants sont concaténés pour retrouver les patrons les plus longs. Ainsi, les sous-patrons communs à plusieurs patrons ne sont calculés qu'une seule fois.

2.2.2 Les séquences les plus fréquentes

La découverte de séquences les plus fréquentes est une spécialisation de la découverte d'ensembles d'items. Dans ce cas, les items d'une transaction sont ordonnés. Le problème consiste à découvrir des séquences d'items qui sont fréquentes parmi l'ensemble des transactions.

Approche de base

L'algorithme *Apriori* a été adapté pour trouver des patrons séquentiels [17]. Dans l'algorithme original, chaque groupe d'item est indépendant. Dans la recherche de patrons séquentiels, un groupe d'items peut en suivre un autre dans le temps. On ne cherche plus seulement à savoir lorsque deux items apparaissent en même temps mais aussi si la présence d'un item annonce la présence d'un autre item dans un groupe d'items subséquent. Cette approche modifiée fonctionne sous le même principe que l'approche originale : une étape de génération de candidats suivie d'une étape d'élimination par manque de support. Une étape additionnelle est ajoutée afin de générer des séquences. Encore une fois, ces séquences sont générées à partir de séquences candidates déjà fréquentes. Les candidats qui n'ont pas le support désiré sont éliminés.

L'algorithme PrefixSpan

Tout comme l'algorithme FP-growth, l'algorithme PrefixSpan cherche à réduire l'impact de trouver de nouveaux candidats pour former des séquences plus longues [18]. Le principe de base de cet algorithme est de diviser la base de données en fonction des préfixes communs des séquences. Cette division est effectuée récursivement en projetant les séquences sur des préfixes de plus en plus longs. La projection conserve uniquement les séquences qui

contiennent le préfixe concerné. Ainsi, cette projection réduit l'espace de recherche et élimine automatiquement les candidats non valides. Cet algorithme se divise en trois grandes étapes. La première étape consiste à trouver les séquences fréquentes de longueur 1. Chacune de ces séquence formera le préfixe de séquences plus longues. La deuxième étape consiste à projeter la base de données pour chaque préfixe trouvé. Finalement, les séquences formées à partir des préfixes trouvés à la première étape sont extraites de cette projection. Les séquences plus longues sont obtenues récursivement en reprojetant la base de données sur les séquences nouvellement trouvées.

Les épisodes

Les algorithmes basés sur la technique *Apriori* ont comme limitation de fonctionner sur des bases de données déjà séparées en transactions. Or, l'information récoltée n'est pas nécessairement regroupée en transactions. Le concept d'épisode a été inventé afin de décrire la relation entre des événements qui surviennent proches les uns des autres dans une séquence d'événements [19]. Les épisodes sont simplement définis comme un ensemble d'événements qui peuvent être ordonnés selon un ordre partiel. Deux types d'épisodes de base sont décrits. L'épisode sériel consiste en un couple d'événements dont l'un deux se trouve toujours avant l'autre dans une séquence. L'épisode parallèle consiste quant à lui en un couple d'événements qui surviennent sans ordre précis. Tout ce que l'épisode parallèle décrit est que ses événements surviennent ensemble mais pas toujours dans le même ordre. Des épisodes plus complexes peuvent être formés en combinant à la fois des épisodes sériels et des épisodes parallèles.

Les algorithmes *Winepi* et *Minepi*

L'extraction d'épisodes fréquents permet ainsi de découvrir des patrons complexes. Les algorithmes Winepi et Minepi décrivent deux techniques pour extraire ces patrons [19]. L'algorithme Winepi consiste d'abord à diviser la séquence en fenêtres de tailles identiques. Par la suite, les épisodes fréquents sont extraits grâce à une technique itérative semblable à celle de l'algorithme Apriori. Des candidats pour des épisodes plus complexes sont générés à partir des épisodes trouvés à l'itération précédente. Les candidats qui n'ont pas de support adéquat (qui ne se retrouvent pas dans un nombre suffisant de fenêtres) sont éliminés. Minepi est une modification de Winepi qui évite l'utilisation de fenêtres sur la séquence. Cet algorithme se base plutôt sur la notion d'occurence minimale. L'occurence minimale d'un épisode est définie comme un intervalle de la séquence qui contient cet épisode mais dont tous les sous-intervalles ne le contiennent pas. Ainsi, le support n'est plus défini comme le nombre de fenêtres qui

contiennent un épisode donné mais plutôt le nombre d'occurences minimales d'un épisode dans la séquence.

Les approches hybrides

La découverte d'épisodes sériels et parallèles ajoute une complexité supérieure au problème de découverte de groupes d'items fréquents. Des techniques hybrides ont donc été développées pour simplifier le problème. Une de ces techniques consiste à considérer un intervalle autour d'un événement comme une fenêtre où tous les événements à l'intérieur de celle-ci sont survenus en même temps. Cette réduction du problème permet d'éliminer toute notion d'épisode sériel puisque tous les événements d'une même fenêtre sont considérés comme étant simultanés [20]. Une fois le problème ainsi réduit, il est possible d'extraire les groupes d'items les plus fréquents en utilisant la technique de son choix [21]. Le choix de la taille de la fenêtre reste un problème. Une fenêtre trop petite et les patrons complexes ne sont pas trouvés. Une fenêtre trop grande et trop de patrons sont trouvés.

2.2.3 Les patrons périodiques

Dans certains cas, les patrons d'intérêt se répètent à intervalles réguliers. Des techniques existent pour détecter spécifiquement ces comportements réguliers [22].

Période connue

Lorsque la période du patron recherché est connue, il est possible d'appliquer une segmentation simple à la séquence et d'utiliser les algorithmes de découverte de séquences traditionnels. Une technique simple consiste à vérifier si un patron séquentiel est présent dans un nombre suffisant de segments. Il est possible d'accélérer cette vérification en considérant que les sous-parties d'un patron cyclique doivent également être cycliques [23]. Ainsi, il n'est nécessaire que de vérifier les segments qui sont des multiples de la période des sous-parties d'une séquence.

Période inconnue

Lorsque la période du patron recherché n'est pas connue, il n'est pas si simple de segmenter la séquence en intervalles. Une recherche exhaustive de toutes les longueurs de segments possibles n'est pas envisageable. Afin d'éviter de devoir vérifier toutes ces périodes différentes, une technique consiste à précalculer les périodes existantes dans une séquence en utilisant une transformée de Fourier [24]. Une autre technique consiste à conserver les informations sur

la fréquence des symboles uniques dans la séquence. Ces informations permettent de guider la recherche en éliminant les périodes impossibles [25].

Les patrons partiellement périodiques

La période d'un patron n'est pas nécessairement stable pendant toute une séquence. Des variations de la période d'un patron peuvent survenir alors que la séquence évolue dans le temps. De plus, dans certains cas, les patrons périodiques ne sont pas toujours actifs. Un comportement périodique peut être présent pendant un moment mais être absent le reste du temps. Ce genre de patron ne possède donc pas de période régulière. Pour décrire ce genre de comportement, la notion de patrons partiellement périodique a été définie [26]. Un patron partiellement périodique est un patron périodique divisé en segments actifs et en segments inactifs. Pendant un segment actif, le patron est présent mais pendant un segment inactif, le patron est absent. Une technique pour déterminer la période d'un tel patron consiste à mesurer le temps écoulé entre deux patrons successifs. Par la suite, une vérification est faite pour s'assurer que la distribution des temps récupérés est significativement différente d'une distribution aléatoire.

2.2.4 L'utilisation de la connaissance du domaine

Les techniques et algorithmes génériques permettent de trouver des patrons dans des jeux de données quelconques. Lorsque le domaine est contraint, il est possible d'utiliser les particularités de ce domaine afin de restreindre l'espace de recherche.

La description de patrons

Lorsque le patron recherché est connu, il est possible de le décrire afin de le retrouver rapidement dans une trace. Par exemple, plusieurs attaques informatiques nécessitent une suite d'actions connues. L'outil AFI (*Automated Fault Identification*) fournit à la fois un langage de description et un engin de détection afin de repérer des patrons dans une trace [27]. Ce langage de description permet de définir une suite d'événements appelée scénario. Des conditions sont appliquées sur les champs des événements afin de spécifier quels événements font progresser l'état du scénario. Il est possible de spécifier un temps maximal entre les événements d'un scénario afin de limiter la recherche de patron à un intervalle de temps délimité. Une fois les scénarios définis, l'engin de détection lit linéairement la trace et vérifie si chacun des événements fait progresser l'état d'un des scénarios. Si un scénario est détecté, une action appropriée peut alors être déclenchée.

Les machines à états

Pour des patrons plus complexes qu'une simple suite d'événements, il est possible d'utiliser des machines à états. Les machines à états sont une technique efficace afin de retrouver des séquences de lettres dans un texte ou encore pour décrire un langage régulier [28]. Elles ont comme avantage de nécessiter une seule lecture du jeu de données. Les machines à états permettent également de recréer l'état du système à partir d'événements simples. Le Tracing and Monitoring Framework (TMF) utilise une machine à états pour recréer l'état des processus d'un système à partir des événements d'une trace noyau. L'état des processus peut alors être affiché à n'importe quel moment de la trace.

Une approche basée sur les machines à états a aussi été utilisée pour récupérer des statistiques et propriétés des systèmes temps réel [29]. Des événements spécifiques sont utilisés pour indiquer le déclenchement et la terminaison de chaque tâche, en plus de déterminer la tâche couramment en exécution. Grâce à ces événements, les temps de réponse et de calcul de chaque tâche peuvent être récupérés en plus de permettre le calcul des temps d'interférence et de blocage.

2.3 Les approches visuelles

D'autres techniques se concentrent plutôt sur la présentation de la trace à un utilisateur. L'utilisateur a alors la liberté d'explorer la trace à sa guise. Nous présentons ici les outils qui permettent de visualiser et d'analyser des traces.

2.3.1 L'outil Zinsight

Zinsight est un visualisateur de traces développé par IBM pour leur gamme de produits basés sur System z [30]. Cet outil présente la trace à l'aide de trois vues distinctes : une vue de flot des événements, une vue de statistiques et une vue de contexte de séquence.

La vue de flot des événements

La vue de flot des événements présente les événements de la trace dans un espace bidimensionnel avec le temps sur l'axe vertical et un groupement variable sur l'axe horizontal. Chaque événement est représenté par un rectangle dans lequel se trouve son nom et les informations qu'il contient. Selon le niveau de zoom, les informations de l'événement sont plus ou moins détaillées. À une échelle très grande, même le nom de l'événement n'est pas affiché. Chaque rectangle possède également une couleur qui représente le module qui a émis cet événement. Un agrandi de cette vue est présenté à la figure 2.1. On y voit des événements

classés horizontalement selon le module qui a émis l'événement en question et verticalement selon l'ordre chronologique. À ce niveau de zoom, le nom des événements est visible mais seulement une partie du contenu est affichée.

La vue de statistiques

Dans la vue de statistiques, les événements sont représentés par des barres verticales. Les événements sont groupés selon leur type. Les types d'événements sont à leur tour groupés selon une catégorie englobant plusieurs types. Le résultat final est une liste de types d'événements sous forme d'arborescence avec chaque événement individuel représenté comme une barre verticale à côté de l'étiquette de son type. Les barres reprennent les mêmes couleurs que les rectangles de la vue de flot des événements. Un exemple de cette vue est présenté à la figure 2.2. On y voit les événements individuels classés selon leur type. Les sous-types sont également visibles grâce à l'utilisation d'une arborescence des types et sous-types. Lorsqu'un groupement contient trop d'événements pour l'espace horizontal disponible, les événements supplémentaires ne sont pas visibles.

Certains types d'événements peuvent être regroupés en paires, par exemple un événement indiquant l'entrée d'une fonction et un autre indiquant le retour de cette même fonction. Pour ces paires d'événements, il existe un mode de temps écoulé. Dans ce mode, il est possible de représenter le temps écoulé entre deux événements pairés en modifiant la hauteur de chaque barre. Dans ce mode de représentation, des barres plus hautes indiquent un intervalle plus grand entre les deux événements.

La vue de contexte de séquence

La troisième vue présente le contexte entourant un certain type d'événements. Ce contexte peut représenter deux choses : la séquence des événements menant à un type d'événement ou alors la séquence d'événements qui suit un type d'événement. Ces séquences sont représentées sous la forme d'un graphe dirigé avec les noeuds représentant un certain type d'événements. Dans l'éventualité où un type d'événements se répète dans la séquence, un cycle est formé dans le graphe. Chaque arête du graphe a un poids égal au nombre de fois que la séquence passe par ces noeuds du graphe. La séquence d'événements la plus probable est représentée par des arêtes plus foncées. Afin de limiter la taille du graphe, les séquences les plus rares sont initialement cachées mais peuvent être étendues au besoin. Un agrandi de cette vue est présenté à la figure 2.3. Dans cette vue, la séquence cyclique dominante de trois événements est représentée par l'utilisation des flèches en gras. Les séquences d'événements alternatives

Figure 2.1 La vue de flot des événements de Zinsight

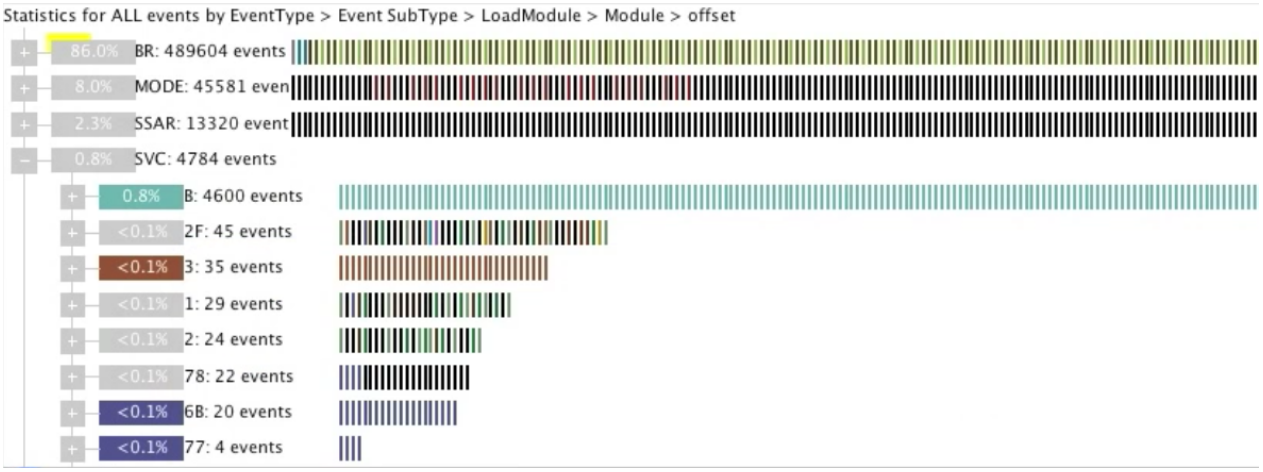

Figure 2.2 La vue de statistiques de Zinsight

rencontrées sont représentées au centre. Les séquences peu probables sont initialement cachées et sont représentées par des rectangles non identifiés.

Toutes ces vues permettent la sélection d'événements ou de types d'événements. Les événements sélectionnés sont représentés par un rectangle de surbrillance autour d'eux. La sélection dans une vue entraîne les mêmes événements à être sélectionnés dans les autres vues. Cette synchronisation entre les vues permet ainsi de facilement se déplacer d'une vue à l'autre.

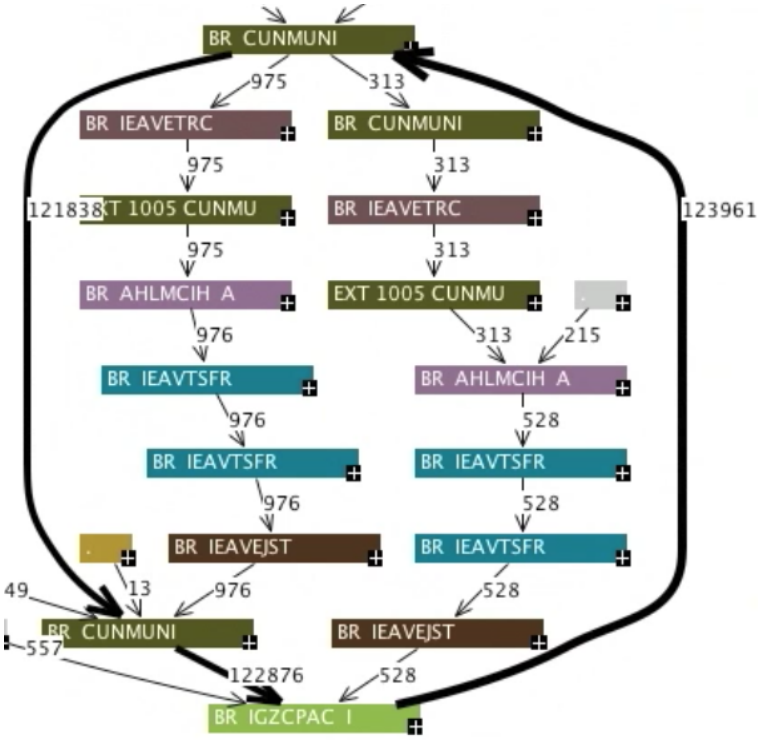

Figure 2.3 La vue de contexte de séquences de Zinsight

2.3.2 L'outil TMF

Le *Trace Monitoring Framework* (TMF) est un outil d'affichage de trace basé sur la plateforme Eclipse [31]. Cet outil possède trois vues principales : une liste des événements, une vue en histogramme, et une vue de flot de contrôle. Grâce à ces vues, il est possible de naviguer à l'intérieur de la trace et de faire un agrandissement sur une zone en particulier. Un aperçu des vues principales est présenté à la figure 2.4.

La liste des événements présente simplement les événements contenus dans une trace dans leur ordre chronologique. Les informations détaillées de chaque événement sont affichées

dans cette vue : l'estampille de temps, le nom et type de l'événement ainsi que l'information additionnelle contenue dans les champs de celui-ci.

La vue en histogramme permet de visualiser l'ensemble de la trace en présentant la densité des événements en fonction du temps. Sur l'axe horizontal, on retrouve l'intervalle de temps représentant la durée de la trace. Le nombre d'événements est présenté sur l'axe vertical. Cette vue permet de repérer les endroits de la trace qui renferment plus ou moins d'événements. À partir de cet histogramme, il est possible de naviguer à un endroit plus précis de la trace. Un second histogramme permet d'avoir l'information plus détaillée sur la zone présentement affichée.

La vue de flot de contrôle permet d'afficher de l'information sur le déroulement de la trace sous forme d'activité des processus. Ainsi, chacun des processus contenus dans la trace est affiché dans une liste le long de l'axe vertical et les états correspondants sont affichés chronologiquement à l'aide de rectangles le long de l'axe horizontal.

L'état des processus est généré à partir d'une machine à états qui transforme des événements-clés en changement d'état pour un ou plusieurs processus. Ces états permettent notamment d'identifier si un processus est en exécution sur un processeur ou encore s'il est en train d'exécuter un appel système.

Le système de stockage des états

Afin d'éviter d'avoir à recréer l'état des processus lorsque l'utilisateur déplace la vue courante, un système de stockage des états a été créé [32]. Ce système permet de récupérer l'état courant à n'importe quel moment en une seule opération. Pour ce faire, chaque état est représenté comme un intervalle contenant un identifiant, un temps de début et de fin ainsi qu'une valeur d'état. Ces intervalles sont stockés sur disque dans une structure en arbre. Chaque noeud de cet arbre correspond à un intervalle de temps de la trace. Chaque noeud représente un sous-intervalle de l'intervalle du noeud parent. Ce sous-intervalle est exclusif par rapport aux sous-intervalles des autres noeuds frères. Ainsi, lorsqu'un requête est faite pour un temps précis, une seule branche de l'arbre doit être parcourue pour récupérer l'état à ce moment.

Grâce à ce système de stockage des états, il est possible de générer la vue de flot de contrôle en effectuant une requête pour chaque pixel horizontal de la vue. Ainsi, le temps de génération de la vue demeure faible, peu importe le nombres d'événements contenus dans la zone de la trace à afficher.

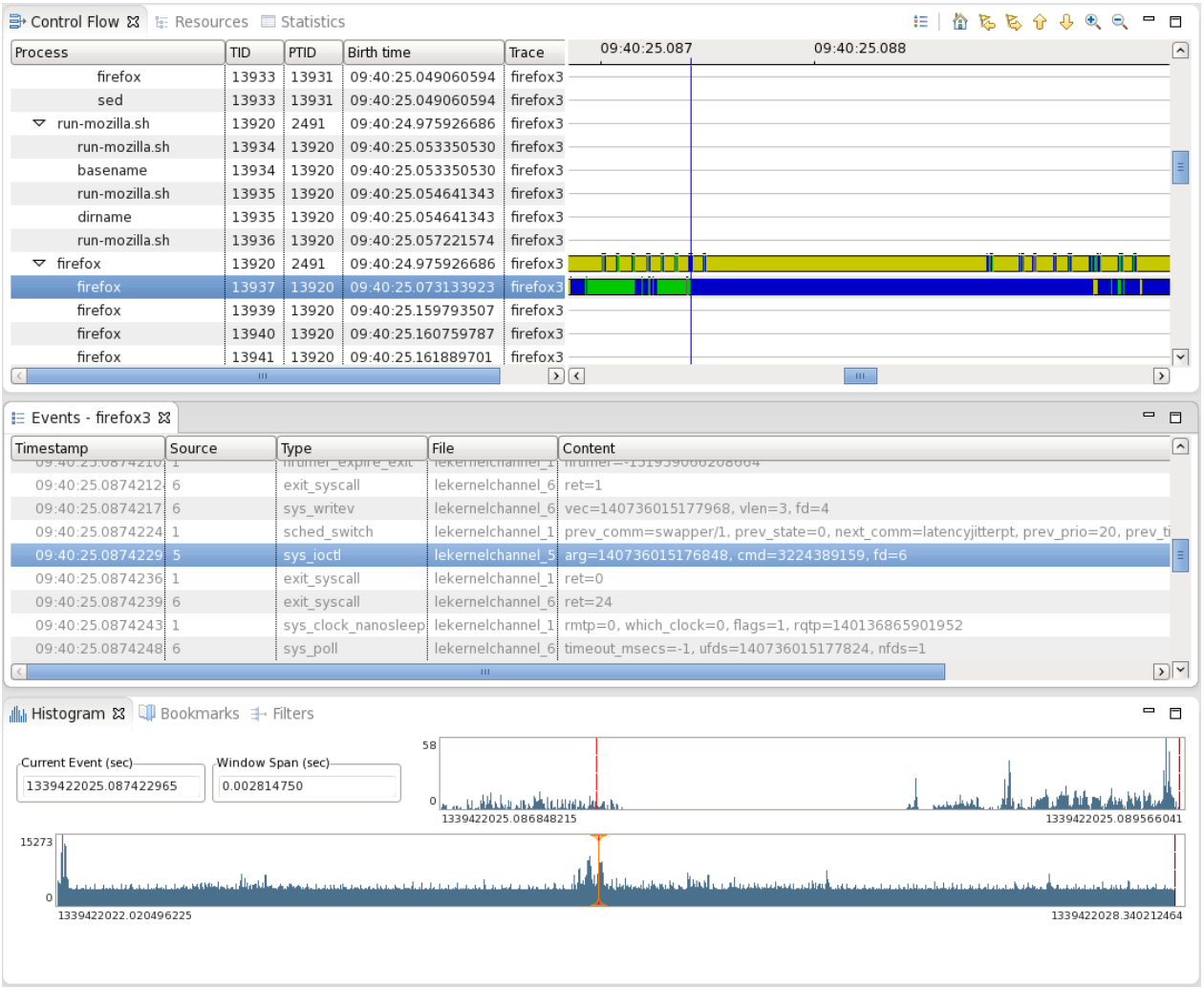

Figure 2.4 La vue principale de TMF

2.3.3 L'outil TuningFork

L'outil TuningFork est un système de traitement et d'affichage de traces [33]. Son architecture ressemble à un pipeline à plusieurs étapes. Les différentes traces, pouvant provenir de différentes sources, sont d'abord fusionnées en un seul flux d'événements. Ce flux unique est par la suite filtré pour obtenir des flux spécialisés selon une métrique particulière. Des opérations mathématiques peuvent par la suite être appliquées sur ces flux, comme la dérivation ou la différence, afin d'obtenir les flux finaux qui seront affichés.

TuningFork utilise une bibliothèque de dessin afin de générer ses différentes figures. Cet outil est ainsi en mesure d'utiliser la plupart des figures standards comme les graphes à courbes, les histogrammes ou encore les graphes en secteurs. Outre ces figures, il est possible de créer des vues spécialisées et de les connecter au pipeline de traitement de flux.

La vue d'oscilloscope

La vue d'oscilloscope permet de visualiser le contenu fréquentiel d'un flux d'information. Cette vue permet l'affichage d'intervalles le long d'une ligne de temps qui est repliée sur elle même, comme les lignes d'un livre. Les intervalles sont dessinés chronologiquement de gauche à droite sur une ligne à l'aide de rectangles. Lorsque la ligne arrive à sa fin, les intervalle sont dessinés alors sur la ligne suivante. En choisissant judicieusement la durée en temps d'une ligne, il est possible de synchroniser chaque ligne selon une période voulue. En sachant la période du patron qu'on cherche à observer, il est possible de configuer la largeur de la vue pour contenir exactement un patron par ligne. Ainsi, chaque ligne contient une période d'intérêt. Un exemple de cette vue est présenté à la figure 2.5.

La division de la trace en périodes permet de visualiser des patrons périodiques connus. La quantité de périodes visibles demeure limitée par le nombre de lignes qu'il est possible d'afficher à l'écran. La solution proposée par TuningFork consiste à continuer le dessin des intervalles sur les lignes existantes, tout comme un oscilloscope analogique. Là où plus d'intervalles sont dessinés, la couleur résultante est plus fonçée. Réciproquement, là où peu d'intervalles sont dessinés, la couleur est plus pâle. Il est ainsi possible d'observer la stabilité d'un patron périodique sur l'ensemble d'une trace. Un patron stable sera indiqué par une bande foncée dessinée sur toutes les lignes de la vue. Un patron plus instable aura une bande de couleur plus faible et plus étendue horizontalement.

Figure 2.5 La vue d'oscilloscope de TuningFork

2.3.4 La suite d'outils Vampir

La suite d'outils Vampir a été créée pour mieux comprendre les intéractions qui existent dans les systèmes parallèles et distribués [34]. Cette suite d'outils se divise en deux parties : un composant d'instrumentation et un composant d'analyse.

Le composant d'instrumentation est appelé VampirTrace. L'instrumentation peut être effectuée de quatre façons. Premièrement, le compilateur peut être utilisé pour insérer des points de trace aux entrées et aux sorties de fonctions. Afin de réduire le surcoût de tracer toutes les fonctions, il est possible de désactiver l'instrumentation des courtes fonctions qui sont appelées souvent. La deuxième technique utilise un préprocesseur pour ajouter directement les points de trace dans le code source. La troisième technique instrumente les librairies de communication telles que la librairie Message Passing Interface (MPI). Une librairie intermédiaire est alors utilisée pour intercepter les appels avant de les rediriger vers la librairie originale. Finalement, il est également possible d'instrumenter manuellement le code source à l'aide d'une interface de traçage fournie par VampirTrace.

Les traces ainsi récupérées peuvent alors être analysées. À cause de l'aspect distribué des systèmes instrumentés, une grande quantité d'information peut être générée simultanément. Afin de traiter toute cette information, un serveur dédié est utilisé pour effectuer l'analyse.

Les résultats de cette analyse peuvent alors être transférés vers une machine moins puissante pour visualisation.

La ligne de temps globale

La ligne de temps globale permet d'afficher les interactions entre les différents processus qui participent à l'échange d'informations dans une application distribuée. Les processus sont placés sur l'axe vertical et on retrouve sur l'axe horizontal l'état de chaque processus en fonction du temps. Des flèches sont dessinées lors d'appel à la librairie de communication pour montrer le sens des échanges. Une exemple de cette vue est présenté à la figure 2.6.

La ligne de temps par processus

Un processus individuel peut être analysé plus en détails en utilisant la ligne de temps par processus. Dans cette vue, on retrouve l'évolution de la profondeur de la pile d'appels en fonction du temps. Des compteurs peuvent également être affichés en même temps, présentant par exemple l'utilisation en mémoire du processus en fonction du temps. Un exemple de cette vue est présenté à la figure 2.7, représentant la profondeur de la pile d'appels d'un processus en fonction du temps.

Les vues de statistiques

En plus des vues temporelles, des statistiques peuvent être affichées pour observer le comportement global d'une application. Ainsi, il est possible de présenter le temps total passé dans une fonction particulière pour un seul processus ou encore l'application entière. Finalement, une dernière vue permet de visualiser le nombre d'échanges entre chaque paire de processus. Une matrice des processus permet d'afficher la quantité d'information échangée entre chaque paire de processus. Des couleurs sont utilisées pour représenter la quantité d'informations échangées. Dans le cas où deux processus n'ont jamais échangé d'information, la case correspondante sera grise. Pour les processus qui ont fait des échanges, la couleur varie du bleu (peu d'échanges) au rouge (beaucoup d'échanges).

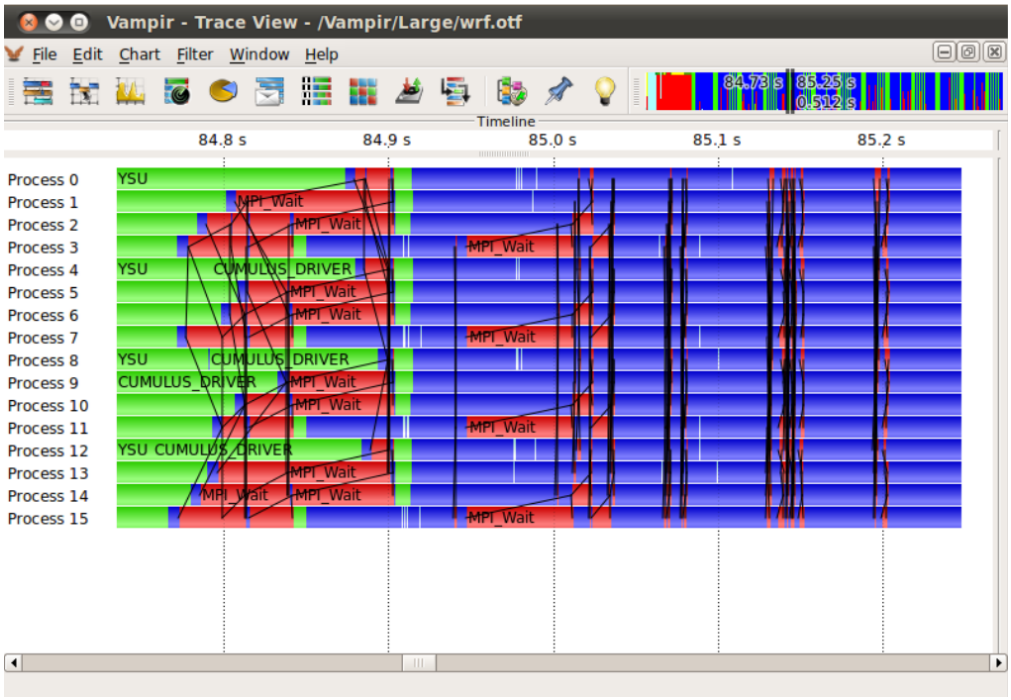

Figure 2.6 La ligne de temps globale de Vampir

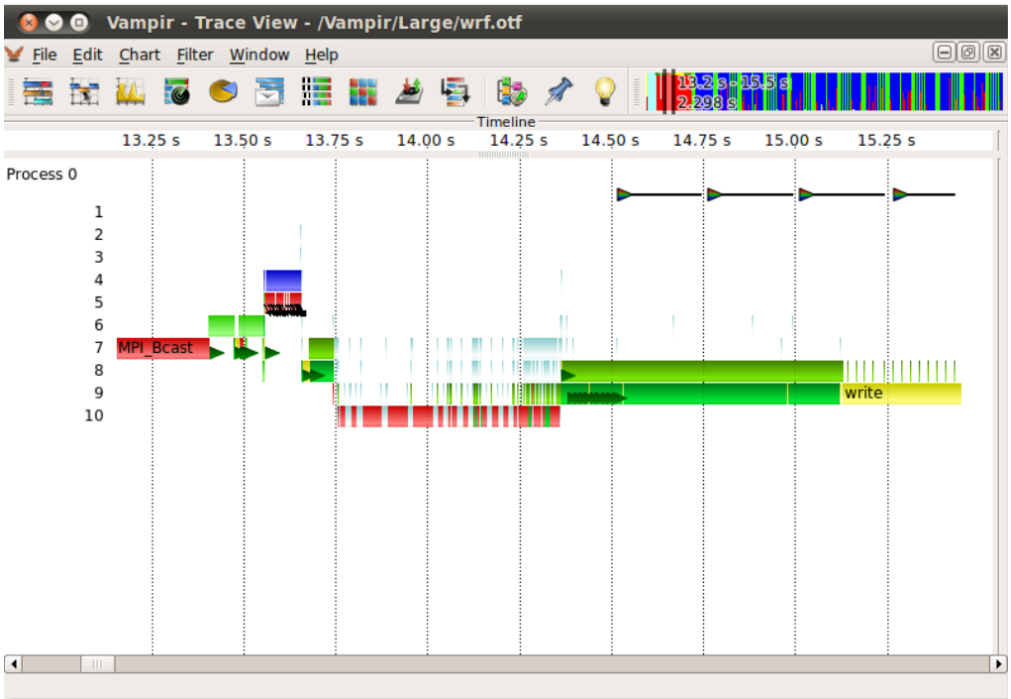

Figure 2.7 La ligne de temps par processus de Vampir

2.4 Conclusion de la revue de littérature

Bien que récent, le domaine du traçage offre déjà plusieurs outils différents qui répondent à des besoins particuliers. La détection de patrons, quant à elle, est un domaine beaucoup plus vaste dont les algorithmes pourraient être utiles pour découvrir des patrons d'intérêt à l'intérieur de traces volumineuses. Les outils de visualisation semblent aussi être des incontournables de l'analyse de trace. De par leur interactivité, ils permettent à un utilisateur de naviguer à travers une trace, récupérer les détails d'exécution ou encore présenter des statistiques pour obtenir un portrait plus global.

Un outil permettant d'analyser des traces de systèmes temps réel pourrait venir complémenter l'offre d'outils existants. Il devient donc intéressant d'identifier les informations de traçage qui permettront de générer des analyses sur mesure.

CHAPITRE 3

MÉTHODOLOGIE

Le développement d'un outil d'analyse commence tout d'abord par circonscrire l'environnement dans lequel il doit évoluer. Nous décrivons dans cette partie les environnements matériel et logiciel avec lesquels nous travaillerons, en plus de présenter les problèmes qui seront considérés pour l'analyse. Finalement, les programmes qui serviront à générer la charge temps réel sur le système sont décrits.

3.1 Environnement de travail

Une station de travail standard a été utilisée pour réaliser les diverses expériences. Les configurations matérielle et logicielle de cette machine sont décrites ci-bas.

3.1.1 Matériel

La même machine a été utilisée pour à la fois récupérer les traces ainsi que pour effectuer l'analyse. Les spécifications de cette machine sont décrites au tableau 3.1. Bien que le processeur utilisé n'ait pas été conçu afin de fournir les garanties nécessaire aux applications temps réel à contraintes dures, il est suffisant dans le cas où nous cherchons à observer des charges de travail précises et non à obtenir les meilleures latences possibles. Certaines précautions ont quand même été prises afin d'éliminer des sources de latence potentielles. En particulier, les technologies Hyperthread et Turbo Boost ont été désactivées. La technologie Hyperthread peut ajouter de la latence à un processus en permettant à plusieurs processus de partager un même coeur physique. La technologie Turbo Boost, quant à elle, permet à la fréquence d'horloge du processus d'être dynamiquement réduite afin de diminuer la consommation en courant du processeur. Outre la désactivation de ces deux technologies, le fonctionnement du matériel n'a pas été davantage ajusté.

Tableau 3.1 Spécifications techniques de la machine de travail

Carte mère	Intel DX58SO
Mémoire vive	2 X 3 Go KINGSTON DDR3 1067 MHz
Processeur	Intel Core i7 930 avec quatre coeurs à 2.8 GHz

3.1.2 Logiciel

Le système d'exploitation utilisé est Ubuntu 12.04. Afin d'obtenir de meilleures performances temps réel, un noyau Linux 3.2 personnalisé a été compilé avec le correctif `PREEMPT_RT`. Les traces ont été récupérées avec le traceur LTTng version 2.3.1.

L'outil `cpuset` [35] a été utilisé afin d'isoler un coeur du processeur sur lequel les applications temps réel sont exécutées. Dans notre cas, le coeur `1` a été réservé pour les applications temps réel et tous les autres processus ont été migrés vers les coeurs `0`, `2` et `3`. L'isolation ainsi effectuée n'est pas complète. Les processus du système d'exploitation qui sont liés à ce coeur ne sont pas migrés, l'objectif étant de fournir un environnement stable et connu pour les applications temps réel. Le script permettant d'isoler un coeur et de migrer les processus vers les autres coeurs est présenté ici :

```
# cset set --cpu=1 --set=rt
# cset set --cpu=0,2,3 --set=rest
# cset proc --move -k --fromset=root --toset=rest
```

3.2 Phénomènes d'intérêt

3.2.1 La stabilité de la latence

Les applications temps réel ont besoin d'un certain déterminisme afin d'assurer qu'elles puissent s'exécuter à l'intérieur du délai limite. La stabilité de la latence est donc un important indicateur de la qualité générale du système. Une application qui exhibe peu de variabilité dans sa latence est une application qui a de bonnes caractéristiques temps réel. Inversement, une latence qui varie beaucoup peut être un indice d'interactions non prévues avec le système ou d'autres applications.

3.2.2 Inversion de priorité

L'inversion de priorité est un problème bien connu dans le domaine des systèmes temps réel. Afin de prioriser certaines tâches plus urgentes, on attribue typiquement des priorités aux tâches et celles-ci indiquent à l'ordonnanceur quelles tâches exécuter en premier. L'inversion de priorité survient lorsqu'une tâche à haute priorité est bloquée par une tâche à basse priorité et que cette tâche moins prioritaire ne peut pas libérer le verrou parce qu'une tâche de priorité moyenne accapare le processeur. Même si cette tâche à priorité moyenne ne possède aucune interaction avec les deux autres, elle retarde l'exécution de la tâche à haute priorité.

Ce genre d'attente non bornée est à éviter dans les applications temps réel. Un mécanisme qui permet d'éliminer les inversions de priorité est l'héritage de priorité. Selon ce mécanisme,

un processus à basse priorité qui bloque un processus de haute priorité verra sa priorité augmentée temporairement. Ainsi, les processus de priorité intermédiare ne pourront pas s'exécuter et préempter le processus à basse priorité, permettant à ce dernier de libérer son verrou plus rapidement.

3.3 Charges de travail

Afin de tester les algorithmes, deux charges de travail différentes ont été utilisées. Ces charges de travail devaient être assez simples pour pouvoir être facilement reproductibles mais assez complexes pour permettre d'y introduire des problèmes typiques des applications temps réel. La première charge de travail permet de générer des tâches périodiques tandis que la seconde génère plutôt des tâches sporadiques.

3.3.1 L'outil cyclictest

L'outil cyclictest permet de vérifier le comportement temps réel d'un système en observant la latence maximale observée pendant un long lapse de temps [36]. Cet outil fonctionne en lançant un ou plusieurs fils d'exécution en priorité temps réel. Chaque fil exécute une simple boucle qui mesure la différence de temps écoulé entre le moment de réveil attendu et le moment de réveil réel. La différence entre ces deux moments correspond à la latence calculée par cyclictest. Les statistiques de cette latence sont ensuite mises à jour avant que le fil ne se remette en attente pour son prochain réveil.

Voici un exemple d'utilisation de cyclictest sur une durée d'une minute. Sur chaque ligne du résultat, on trouve le numéro de chaque fil d'exécution, sa priorité, sa période, le nombre de cycles exécutés ainsi que les latences minimale, courante, moyenne et maximale observées.

```
# cyclictest -t10 -p99 -n -q -D 1m
T: 0 (11786) P:99 I:1000 C:  60000 Min:     1 Act:     2 Avg:    1 Max:     24
T: 1 (11787) P:98 I:1500 C:  40000 Min:     1 Act:     2 Avg:    1 Max:     22
T: 2 (11788) P:97 I:2000 C:  30000 Min:     1 Act:     2 Avg:    1 Max:     23
T: 3 (11789) P:96 I:2500 C:  24000 Min:     1 Act:     2 Avg:    2 Max:     23
T: 4 (11790) P:95 I:3000 C:  20000 Min:     1 Act:    10 Avg:    2 Max:     15
T: 5 (11791) P:94 I:3500 C:  17143 Min:     1 Act:     2 Avg:    1 Max:     10
T: 6 (11792) P:93 I:4000 C:  15000 Min:     1 Act:     2 Avg:    1 Max:      8
T: 7 (11793) P:92 I:4500 C:  13334 Min:     1 Act:     2 Avg:    1 Max:     10
T: 8 (11794) P:91 I:5000 C:  12000 Min:     1 Act:     2 Avg:    1 Max:     21
T: 9 (11795) P:90 I:5500 C:  10910 Min:     1 Act:     2 Avg:    1 Max:     14
```

3.3.2 Application producteur-consommateur

Nous avons créé une application de type producteur-consommateur afin de générer des tâches sporadiques en plus de vérifier les mécanismes d'héritage de priorité. L'application est constituée de trois tâches temps réel. Le producteur possède une priorité faible qui a comme seule tâche de remplir un tampon avec des valeurs aléatoires. Le consommateur, lui, possède une priorité moyenne et se charge de lire les valeurs émises par le producteur. Finalement, la troisième tâche est périodique et possède la priorité la plus élevée. Son rôle est de perturber les deux autres tâches en forçant l'ordonnanceur à les préempter.

Le tampon

Le producteur et le consommateur s'échangent les valeurs à l'aide d'un tampon circulaire en mémoire partagée. La synchronisation de l'écriture et de la lecture est effectuée à l'aide d'une paire de sémaphores. La première sémaphore s'assure qu'il y a de l'espace dans le tampon pour que le producteur puisse écrire. Lorsque le consommateur lit une valeur, il l'indique en incrémentant ce sémaphore. Le producteur quant à lui incrémente le sémaphore lorsqu'il rajoute une valeur au tampon. Le second sémaphore a un rôle corollaire, il s'assure que le tampon n'est pas vide et que le consomateur puisse lire. Il est incrémenté après que le producteur ait placé une valeur dans le tampon et décrémenté avant que le consommateur n'effectue une lecture.

3.4 Aperçu de l'approche proposée

Au chapitre 4, une approche est présentée permettant de modéliser l'exécution d'une application temps réel afin de découvrir les problèmes de stabilité de la latence et d'inversion de priorité décrits à la section 3.2 en plus d'observer d'autres comportements d'intérêt. Les charges de travail présentées à la section 3.3 sont utilisées afin de mettre en évidence ces comportements. Grâce aux traces du système et à des vues spécialisées, les sources de latence sont découvertes et expliquées.

CHAPITRE 4

ARTICLE 1 : REAL-TIME LINUX ANALYSIS USING LOW-IMPACT TRACER

Authors

François Rajotte
École Polytechnique de Montréal
francois.rajotte@polymtl.ca

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

Submitted to Hindawi Advances in Computer Engineering, March 20th, 2014

4.1 Abstract

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

4.2 Introduction

Real-time applications distinguish themselves from their non-realtime counterparts by their strict timing constraints. The correct operation of a real-time system requires that it responds to stimuli in a bounded time. Real-time systems are often separated in two categories: hard and soft real-time. Hard real-time requires the response time to be bounded

and never exceeded. In soft real-time systems, an exceeded response time is undesirable but does not incur the complete failure of the system.

The real-time capabilities of the Linux kernel have been improved thanks to the work done by the `PREEMPT_RT` patch contributors. Many tools have been developed to help demonstrate the real-time capabilites and limits of Linux systems. Previous work has also demonstrated the good real-time behavior of the LTTng tracer [11]. LTTng provides both kernel and userspace instrumentation. Because of the demonstrated low impact of LTTng on real-time applications, we have chosen to use it to gather the traces required for the analysis.

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

The low overhead needed for the instrumentation means that the events are recorded with as little preprocessing as possible. In order to extract more advanced information from the events, it is possible to apply a post-processing step on a recovered trace. Using the semantics of the events, it is possible to extract metrics such as CPU or memory usage over time [37]. Our contribution consists in an analysis framework to extract additional debugging information and metrics from a trace recorded using the LTTng tracer on a Linux system running real-time applications, without the need for manual instrumentation.

4.3 Related work

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

4.3.1 Existing Algorithmic Techniques

Some techniques use knowledge of the execution of a task to compute metrics and statistics. Santos and Wellings calculate blocking time experienced by a task to help identify errors in the worst-case execution time assumptions. [38] This algorithm uses knowledge of the tasks' base and active priorities to identify periods of priority inversion. Modifications in the operating system were required to acquire all the information necessary for the algorithm. Terrasa and Bernat use finite state machines to extract metrics at the task and global level. [29] Simple automatons can generate meta-events to feed into larger automatons enabling the computation of more complex metrics. This approach defines four minimal events

required to build the automatons. These events describe when a task becomes ready, when it is finished as well as its base priority. The final event describes context switches. Of these four events, only the context switch event is directly retrievable from a Linux kernel trace.

Data mining techniques are also useful to find periodic patterns and anomalies in a trace. The concept of episode is introduced to describe temporal relationships between events [19]. An algorithm is provided to find frequent episodes. The frequency of episodes is determined by splitting the trace into windows of a fixed size and measuring the number of windows that contain the episode. These frequent episodes can then be used to infer rules about the presence or absence of events in a trace. Other techniques focus on finding periodic patterns. Some of these techniques require the definition of windows or specific events to split the trace into individual worksets [39] while others can find patterns without this need [26]. Common to all these methods is the incremental approach to building the patterns. It is therefore necessary to read the trace multiple times or preprocess the trace in a different format.

4.3.2 Existing Visualization Techniques

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

Zinsight provides three views to present trace data in different formats [30]. The first one places the events in rectangles in a two-dimensional plane with time on the vertical axis and a variable grouping scheme on the horizontal axis. The second view groups events by type and provides timing information on paired events such as function entry and exit. The third view calculates sequences of events leading up or following an event type of interest, presenting this information in a directed graph. This view allows a developer to see common event sequences and abnormal ones. These views have the advantage of letting users display information in many different ways, but they still require advanced knowledge of the trace events to find sequences of interest.

TuningFork is a framework developed specifically to help debug complex real-time systems [33]. It provides filters and aggregation functions to generate data for views. Among its generic views is the Oscilloscope view. This view allows the visualization of high frequency data by separating the trace in strips using a predefined time interval and stacking them. This view allows the observation of periodic behaviour but it requires the knowledge of the period of the task and is of limited use on tasks exhibiting a varying period, such as sporadic tasks.

4.3.3 TMF

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

4.4 Proposed model

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

In schedulability analysis [40], a periodic task is defined using a period, a relative deadline and a worst case execution time. A sporadic task is similar to a periodic task but has a variable period. It is rather defined using a minimum arrival time, specifying the minimum time between two jobs.

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

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

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

can wake up the thread. Killable sleep is a specialization of uninterruptible sleep that also allows the thread to wakeup when a fatal signal is received.

4.4.1 Modeled task states

The thread states contained within the Linux kernel represent the concerns of the operating system and do not translate directly to the realities of the application or real-time task. A recurring real-time task will follow a pattern of two major phases. It is either executing to complete before a deadline, or waiting until its next execution. Because other real-time tasks are executing at the same time, the execution phase will generally be broken up by periods of waiting. This waiting can happen because a higher priority task must execute first, or because a necessary resource is currently held by another task.

A real-time task will therefore follow a series of waiting and execution states. Our approach models the different states that a task can be in during its execution. Most importantly, it distinguishes between the different reasons for waiting. For that purpose, we have modeled four different states describing different types of waiting. A fifth state is used to describe the running state. These states and the possible transitions between them are shown in figure 4.1.

WAITING represents the duration between the end of the previous job and the start of the following job.

READY represents the duration between a task receiving the signal to wake up (its *arrival*) and the actual start of the job.

BLOCKED is reached when a task is blocked from entering a critical section.

PREEMPTED happens when a task is preemptively stopped from running because of a higher priority task entering the running state on the same processor.

RUNNING is the state in which the task executes the job.

4.4.2 Trace events

In order to extract these states from the trace, we have identified the kernel events that allow us to define the necessary state transitions. When tracing a real-time system, it is important to disturb the system as little as possible. As such, we have chosen those events because they are the minimal set that allows to describe the transitions of our model.

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

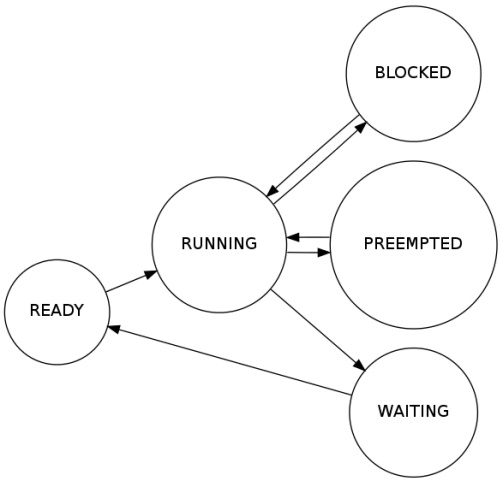

Figure 4.1 The states of the model and the possible transitions between them.

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

4.4.3 State transitions

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

The running-to-blocked and running-to-waiting transitions are trickier because in both cases the thread will be in the `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, or `TASK_KILLABLE` state. Using the `prev_state` field is not sufficient in this case because the kernel uses the same values to describe different realities at the task level. The ambiguity arises if the task uses mutexes with priority inheritance to delimit its critical sections. According to the priority inheritance protocol, in case of contention, the priority of the offending thread will be boosted to the priority of the highest priority blocked thread. We can therefore use the `prev_prio` and `next_prio` fields to distinguish the blocked and waiting states. If the next thread to run has a lower priority, we can be certain that the previous thread has transitioned to the waiting state. If the priority of the next process is equal or higher, the previous thread has transitioned to the blocked state. Figure 4.2 presents the graph of transitions with the required events and their fields.

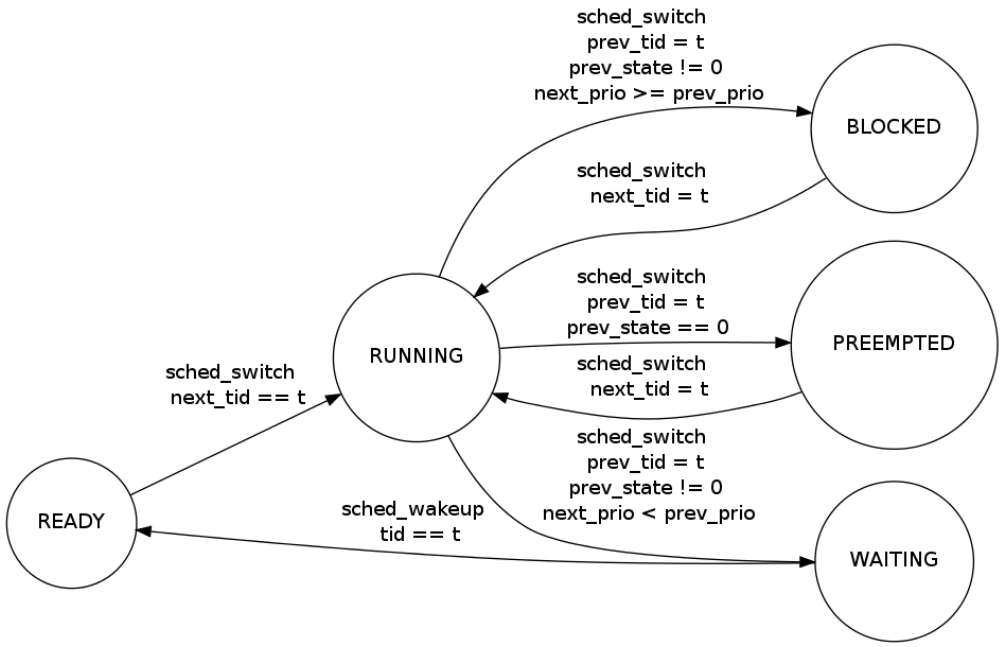

Figure 4.2 The state transitions are defined using specific events and their fields

This distinction can be made if all threads follow the first-in first-out scheduling (`SCHED_FIFO`) and have different base priorities. In the case of a higher priority thread becoming ready, it will immediately preempt the current running thread. The current task will therefore always enter the *preempted* state at that point. In the case of a lower priority thread becoming ready, it will not have the chance to run until all other higher priority tasks voluntarily enter the *waiting* state. The only ambiguous transition is when a thread of the same priority replaces the current one. Since we have as a restriction that all threads have different base priorities, the only way another thread would have the same priority is if its priority was boosted. The boosted priority requires that the current thread be blocked from acquiring a mutex still held by the offending thread.

4.4.4 Statistics extraction

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

value for a job is kept. Table 4.1 shows an example of common statistics and the transitions used to compute them.

4.5 Performance analysis

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

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

For this test, we used cyclictest running ten threads at varying periods and priorities while tracing the kernel. Larger traces were achieved by running cyclictest for a longer period of time. The analysis was run on an Intel Core7 processor running at 2.8 GHz with 6 GB of RAM. The analysis time is presented in figure 4.3. The time spent only reading the trace is also presented to better show the actual time spent generating the model.

We observe a linear progression of the time spent generating the model compared to the size of the trace. This is expected since evaluating a state change of the model requires verifying conditions on a finite number of fields for each event. Most of the time is actually spent reading and parsing the trace file. This cost is unavoidable but is not a major issue when we take into account the fact that other analyses can be run at the same time on the same trace. The cost of reading the trace is therefore amortized over all the analyses running on the same data.

Table 4.1 State transitions that start and end the accumulation of a given statistics

statistics	transition begin	transition end
latency	waiting-to-ready	running-to-waiting
running time	*-to-running	running-to-*
blocking time	*-to-blocked	blocked-to-*
inter-arrival time	waiting-to-ready	waiting-to-ready
wakeup time	waiting-to-ready	ready-to-running

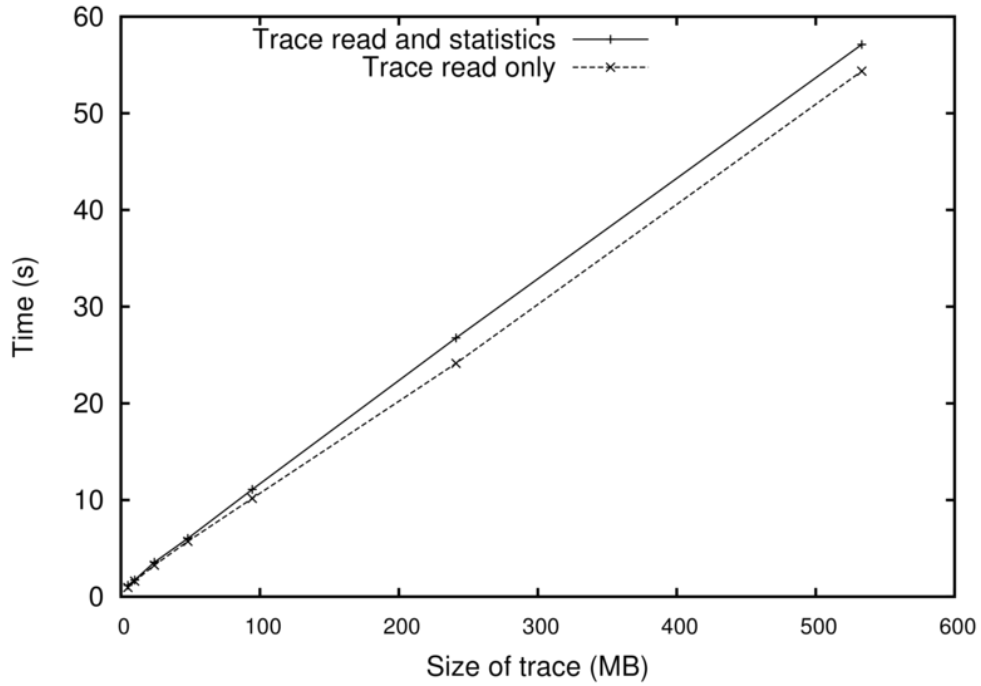

Figure 4.3 Time spent calculating our model for traces of varying sizes

4.6 View

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

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

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

4.7 Test cases

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

4.7.1 Basic concepts

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

Once this run of cyclictest is traced, the analysis is performed. The gathered statistics provide a good overview of the whole run. Figure 4.4 presents the histogram of the latency observed for the fifth highest priority task. The maximum latency observed is 14 us but most latencies are between 2 and 3 us. Other peaks are observed at 7 and 10 us.

Statistics alone cannot explain the observed behavior, but they provide clues for potential problems. In this case, we would like to understand the cause of the observed latency peaks. Although still acceptable, the peaks could be signs of a deeper problem. The Control Flow View of TMF presents the trace in a chronological fashion that allows zooming in on a time range to observe the interactions between threads. Such a view is presented in Figure 4.5. In this view, threads of higher priority are at the top. Although most jobs are executed without delay, sometimes a higher-priority job will preempt the execution of a lower-priority one. This preemption increases the latency of lower priority jobs. We also observe that longer delays can happen when multiple higher-priority jobs need to execute in a short interval.

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

The resulting view is shown in figure 4.6. Using this view, we can observe that the longer latencies are not randomly distributed but follow a pattern. Every other job, a higher priority preempts the current job. Every sixth job, another job also preempts it, producing an even

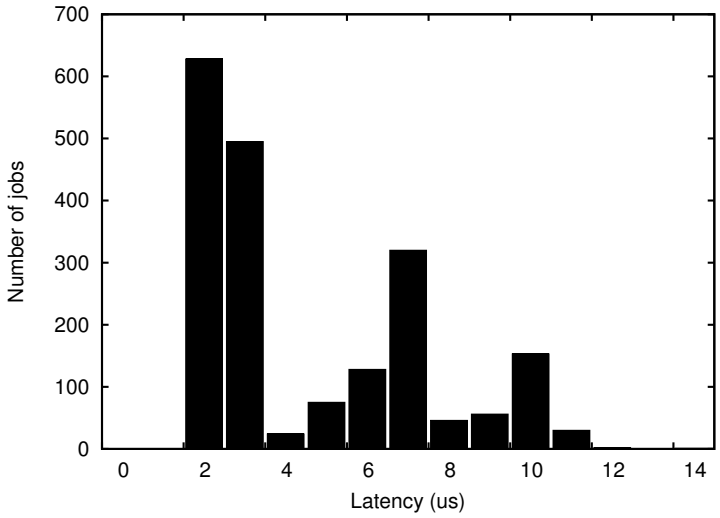

Figure 4.4 Histogram of the latency of a cyclictest thread running at medium priority

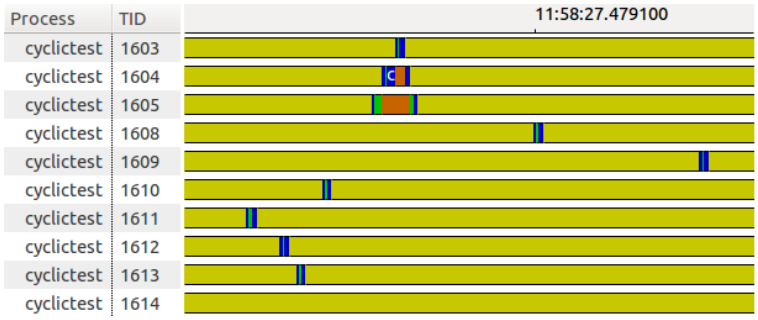

Figure 4.5 A zoomed-in Control Flow View of TMF showing the preemption of threads

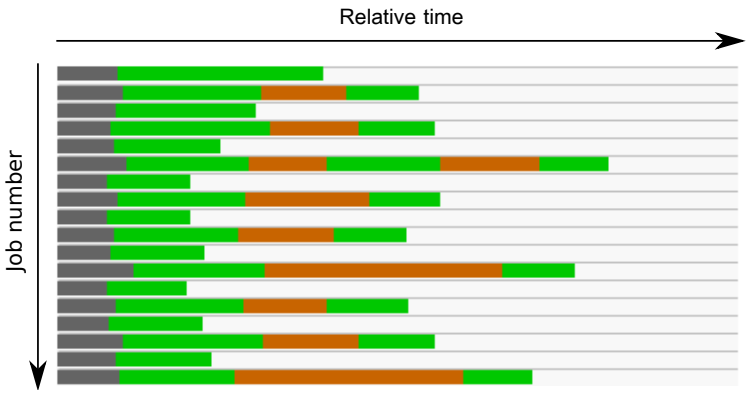

Figure 4.6 Our view showing individual jobs stacked and synchronized on their arrival time

larger delay. This is caused by the arrival times of different tasks that happen too close to one another.

4.7.2 Abnormal delay

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

Since we want to find a missed deadline, we use our view while sorting by longest latency first (figure 4.7). The worst offending job will be at the top of the view. We can further explore the source of the problem by examining the surrounding area of the trace. Since our view is synchronized with the Control Flow View, simply clicking on a job will take the Control Flow View to the same location in the trace. This view is shown in figure 4.8.

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

4.7.3 Sporadic tasks

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

We have attributed higher priority to the consumer task than the producer task to keep the overall latency of the application as low as possible. The data is transferred from the producer to the consumer using a shared buffer in memory. To prevent concurrent access, this buffer is synchronized using semaphores. A third task is also running at the same time,

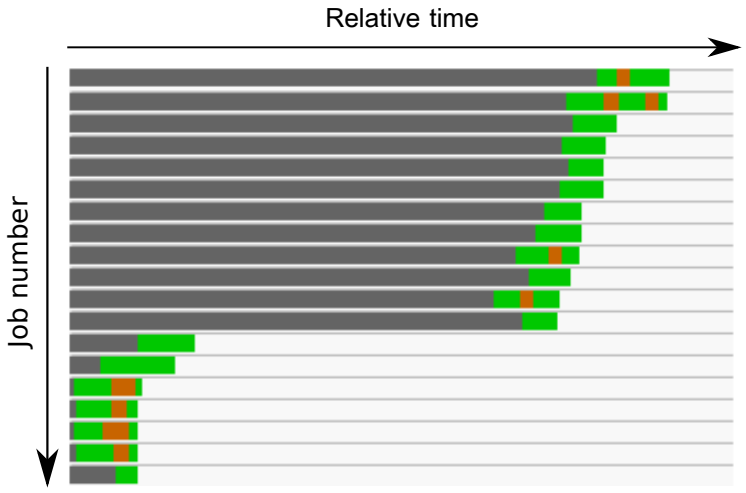

Figure 4.7 Our custom view showing individual jobs, sorted by longest latency

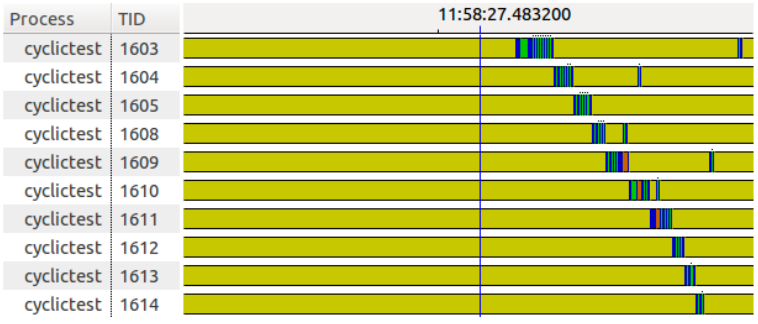

Figure 4.8 The delayed threads as seen in the Control Flow View of TMF

with the highest priority. This task's purpose is to disturb the two other tasks of interest and create jitter.

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

We can try to verify this behaviour using the Control Flow View of TMF. In figure 4.9, we can see the tasks executing one after the other as expected. However, it is not clear when each of the expected phase is active. There appears to be additional scheduling activity that was not predicted by the previous analysis. In figure 4.10, we use our algorithm to split the consumer task in its individual phases.

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

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

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

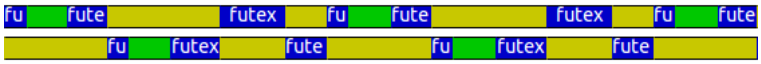

Figure 4.9 The producer (top) and consumer (bottom) threads as seen in TMF's Control Flow View

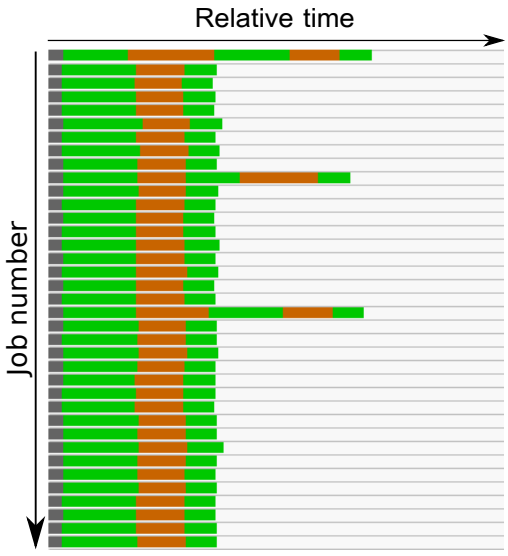

Figure 4.10 The individual jobs of the consumer thread, ordered chronologically

the consumer wakes up yet again; this time, to wait on the availability of data on the shared buffer.

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

Using this real-time application as example, we were able to observe some inner workings of the Linux kernel that are not obvious when programming at the userspace level. Our task-splitting algorithm can improve the comprehension of a trace by extracting important states and displaying them in a way that helps the user discover interesting patterns that are not obvious in a strictly linear chronological view.

4.8 Conclusion

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

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

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

CHAPITRE 5

RÉSULTATS COMPLÉMENTAIRES

5.1 Statistiques continues

Au chapitre 4, nous avons vu comment il était possible de récupérer des statistiques à partir de machines à états. Ces statistiques étaient associées à un travail particulier. Dans certains cas, il n'est pas possible d'associer une statistique à un élément ponctuel. Par exemple, le temps de calcul utilisé par une tâche évolue continuellement pendant qu'elle s'exécute. Il n'est pas approprié d'associer une seule valeur à tout cet intervalle, car la valeur évolue en fonction d'un état et non d'un événement.

5.1.1 Approche actuelle

Les statistiques de base représentées dans TMF correspondent à un simple compte du nombre d'événements d'un certain type pendant une période donnée. Afin de récupérer ce compte rapidement, une stratégie basée sur le système d'états est utilisée. Cette stratégie consiste à conserver le compte total d'un type d'événement dans un état. Lors de la lecture de la trace, ce compte est incrémenté à chaque événement lu. Une représentation de cette prodédure est illustrée à la figure 5.1 où chaque flèche représente un événement. Afin d'obtenir le compte exact dans un intervalle donné, une requête est effectuée au système d'états au temps de fin et une autre au temps de début. La différence entre le compte de fin et le compte au début permet de retrouver le nombre d'événements survenus pendant cet intervalle.

5.1.2 Calcul du temps d'exécution en continu

Un principe similaire peut être utilisé pour récupérer rapidement le temps d'exécution d'un processus dans un intervalle quelconque. Il suffit de récupérer le temps cumulatif d'exécution à la fin de l'intervalle et le soustraire au temps cumulatif au début de l'intervalle. Pour

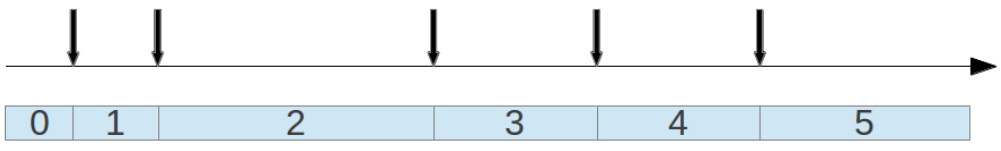

Figure 5.1 Représentation du décompte d'un type d'événement en fonction du temps

stocker le temps cumulatif, il est encore possible d'utiliser le système d'états. Un état est utilisé pour représenter le cumulatif du temps d'exécution d'un processus. Ce temps est mis à jour lorsqu'un processus cesse de s'exécuter. Le décompte est alors incrémenté du temps que le processus a passé en exécution. Un exemple de cette représentation est présenté à la figure 5.2, dans laquelle les rectangles supérieurs représentent les périodes d'exécution avec le temps écoulé dans cette période, et les rectangles du bas le temps d'exécution cumulatif stocké.

Cette représentation donne des résultats imprécis lorsqu'un temps cumulatif est demandé pour un moment où le processus est encore en exécution. Dans ce cas, le temps cumulatif retourné sera le temps tel que mis à jour lors de la fin de la période d'exécution précédente. La période d'exécution courante n'est donc pas prise en compte et cela entraîne une imprécision.

5.1.3 Interpolation du temps de calcul

Afin de récupérer le temps cumulatif exact à un instant donné, il faut pouvoir ajouter le temps d'exécution courant si le processus est en exécution. Pour récupérer cette information, nous pouvons utiliser le système d'états qui est déjà calculé par TMF. Dans ce système d'états, on retrouve déjà l'état d'exécution de chaque processus à n'importe quel moment. Une requête à ce système d'états permet de savoir facilement si le processus d'intérêt est en cours d'exécution. Si ce n'est pas le cas, alors il n'est pas nécessaire de faire une interpolation, le temps cumulatif d'exécution est exact. Cependant, si le processus est en cours d'exécution, il faudra interpoler le temps manquant. À la figure 5.3, on présente une requête de temps d'exécution à l'intérieur d'une période délimitée par des barres verticales.

Pour faire cette interpolation, il faut connaître le temps de début d'exécution du processus courant. Cette information est également disponible dans le système d'états de TMF. La même requête qui a permis de savoir si le processus courant était en exécution peut retourner aussi tous les champs de l'intervalle, y compris son temps de début. Le temps de début de cet intervalle correspond au temps de début de l'exécution du processus. Comme étape finale, il suffit de soustraire le temps de la requête au temps de début de l'intervalle pour obtenir le

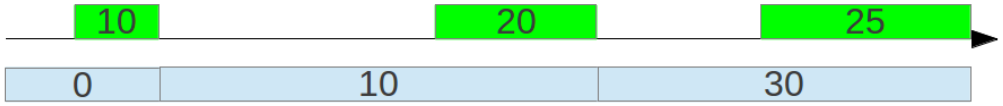

Figure 5.2 Représentation du temps d'exécution cumulatif d'un processus en fonction du temps

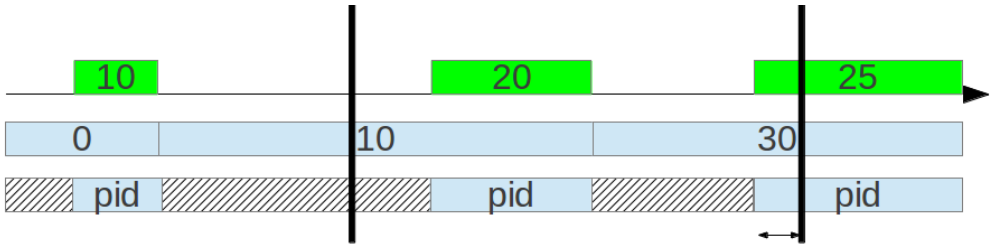

Figure 5.3 Représentation d'une requête de temps d'exécution d'un processus (haut) en utilisant le temps cumulatif (milieu) et une interpolation en utilisant le système d'états déjà existant de TMF (bas)

temps d'exécution courant du processus. Cette différence est présentée par une double flèche horizontale à la figure 5.3.

5.1.4 Création d'un nouveau type d'intervalle

Le système d'états utilisé par TMF permet de conserver des états sous forme d'intervalles. Ces intervalles possèdent cinq attributs principaux. Le détail des champs des intervalles est présenté dans le tableau 5.1. Grâce au champ type, il est possible d'indiquer comment interpréter le champ valeur. La taille du champ valeur est cependant limitée à 4 octets. Pour stocker des valeurs plus grandes, comme des chaînes de caractères, il est possible d'inscrire dans le champ valeur un décalage qui permet de référencer un autre endroit dans le fichier, qui lui contient l'information recherchée.

Dans le cas qui nous intéresse, nous voulons stocker des statistiques qui peuvent aisément dépasser la capacité d'un entier de quatre octets. La précision des estampilles de temps des événements dans une trace produite par LTTng est de l'ordre de la nanoseconde. Si nous prenons comme statistique le temps d'exécution d'une tâche, précis à la nanoseconde, le compte

Tableau 5.1 Liste des champs contenus dans un intervalle du système d'états de TMF ainsi que leur taille

Nom du champ	Taille en octets
Temps de début	8
Temps de fin	8
Identifiant	4
Type de la valeur	1
Valeur	4

total peut facilement dépasser les quelque milliards (quelques secondes d'exécution totale) et dépasser la capacité d'un entier de quatre octets.

Afin de pallier à cette limitation, un type d'intervalles pouvant contenir des entiers de 8 octets a été créé. Ce type d'intervalles utilise le même principe utilisé pour stocker les chaînes de caractères. Au lieu d'écrire directement la valeur de l'entier dans l'intervalle, une valeur de décalage y est inscrite. En allant à la position ci-inscrite, l'entier de 8 octets est retrouvé.

Cette solution permet de stocker des intervalles contenant des valeurs de 8 octets sans briser la compatibilité avec le format original. Une solution alternative aurait été de redéfinir les champs d'un intervalle afin de contenir des valeurs de 8 octets par défaut et ainsi augmenter l'espace nécessaire pour chaque intervalle par 4 octets au lieu de 8. Cette solution n'a pas été retenue parce que le format du fichier n'aurait pas été compatible avec les versions précédentes. De plus, dans le cas où les entiers de 4 octets sont suffisants, l'espace supplémentaire disponible aurait été gaspillé. Puisque le système d'états a été conçu à la base pour le stockage d'entiers de 4 octets, la plupart des cas d'utilisation existants ne requiert pas cet espace supplémentaire. Ainsi, il est préférable de conserver la compatibilité dans les cas existants en échange d'un léger surcoût en espace disque lorsqu'une valeur de 8 octets doit être stockée.

5.1.5 Efficacité en espace de stockage

La taille du système d'états résultant du calcul des temps d'exécution peut être estimé simplement en dénombrant le nombre d'événements `sched_switch` contenu dans une trace. En effet, chaque événement `sched_switch` indique qu'un processus cesse de s'exécuter au profit d'un autre. Ainsi, pour chaque événement `sched_switch`, un nouveau cumulatif est calculé, entraînant l'ajout d'un nouvel état dans le système d'états. Or, chaque état est représenté par un intervalle dont la taille est toujours fixe, soit 33 octets (25 octets pour un intervalle de base et l'espace pour l'entier de 8 octets).

Au tableau 5.2, la taille des différents systèmes d'état sont présentés pour une trace contenant 1 million d'événements `sched_switch`. On remarque que la taille du système d'états pour les temps d'exécution est en effet environ 33 octets pour chaque événement `sched_switch`. La taille de ce système d'états est également beaucoup plus petite que celle des deux autres, indiquant qu'il ne sera pas problématique d'ajouter cette analyse à la fonctionnalité de base de TMF.

Tableau 5.2 Comparaison des systèmes d'états pour une trace de 286 Mo contenant 11 millions d'événements dont 1 million d'événements `sched_switch`

Système d'états	Taille du fichier en Mo
État des processus	578.5
Statistiques du nombre d'événements	572.3
Statistiques des temps d'exécution	34.7

5.2 Vue de l'utilisation du processeur

Grâce à ce nouveau système d'états, il est possible de populer une vue qui affiche l'utilisation du processeur en fonction du temps, et ce à n'importe quelle échelle de précision souhaitée. Pour chaque pixel horizontal de cette vue, une requête est effectuée au système d'états au temps correspondant aux frontières gauche et droite du pixel. La différence d'utilisation cumulative obtenue entre ces requêtes correspond au temps d'utilisation à l'intérieur des frontières du pixel. Cette même procédure est répétée pour chaque pixel horizontal de la vue afin de générer le portrait complet de l'utilisation du processeur. Il est important de noter que le coût pour construire cette vue dépend essentiellement du nombre de pixels et non de la taille de la trace. Le portrait global pour une trace de très grande taille (e.g. 200 Go) peut donc être calculé très efficacement. Un prototype d'une telle vue est présenté à la figure 5.4. Cette vue, ainsi que les autres outils et résultats présentés précédemment, constituent l'ensemble des contributions de cette partie du projet.

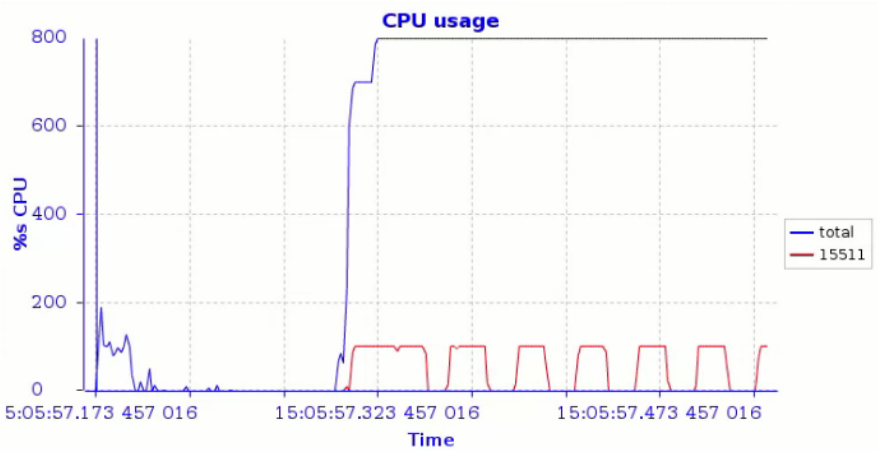

Figure 5.4 Vue de l'utilisation du processeur en fonction du temps montrant à la fois l'utilisation totale du processeur et la contribution d'un processus

CHAPITRE 6

DISCUSSION GÉNÉRALE

Dans ce chapitre, nous allons revenir sur les résultats obtenus aux chapitres 4 et 5. Nous allons entre autres mettre en évidence les contributions face aux différentes techniques présentées au chapitre 2.

6.1 Retour sur les résultats

Notre contribution principale consiste en un modèle qui permet d'extraire le comportement en phases d'une application temps réel. Une caractéristique importante de ce modèle est qu'il permet de transformer les informations de processus contenues dans le système d'exploitation en concepts de plus haut niveau utilisés dans la conception d'applications temps réel. L'objectif principal de cette recherche était d'élaborer un outil pour analyser un système temps réel grâce aux informations de traçage. Afin d'avoir une grande portée, cet outil se devait de s'intégrer aux outils d'analyse existants. Au chapitre 2, nous avons vu comment des patrons peuvent être extraits d'une séquence à l'aide d'algorithmes spécialisés. D'autres outils laissent plutôt à l'utilisateur la chance d'explorer et de découvrir lui-même les patrons d'intérêt. Notre approche combine ces deux grandes idées. Premièrement, une lecture de la trace permet d'extraire un modèle. Grâce à l'utilisation d'une machine à état, ce modèle peut être généré rapidement et ne nécessite qu'une seule lecture. Deuxièmement, une vue a été développée afin de permettre à un utilisateur d'explorer le modèle et de mettre en évidence des patrons d'intérêt. Finalement, une intégration avec les autres outils d'analyse permet à l'utilisateur d'explorer à sa guise le reste de la trace.

6.1.1 Choix des événements tracés

Une des préoccupation lors de la conception du modèle était que l'information nécessaire à sa génération soit facilement accessible grâce aux outils de traçage. Les événements déjà contenus dans le noyau Linux ont été utilisés car ils ne nécessitent aucune modification du côté de l'application, en plus de fournir de l'information qui n'est pas disponible du côté utilisateur. Une autre préoccupation lors du traçage d'une application temps réel est de limiter l'impact négatif sur le déterminisme du système. Afin de réduire l'impact du traçage, notre modèle a été conçu de façon à nécessiter l'utilisation de seulement deux événements noyau :

`sched_wakeup` et `sched_switch`. Ces deux événements permettent à eux seuls d'observer les interactions complexes qui peuvent exister entre plusieurs tâches.

6.1.2 Contraintes du modèle

Un des problèmes rencontrés lors de l'élaboration du modèle est la distinction entre une période de blocage due à une demande de réservation de ressource et une période d'attente entre deux exécutions d'une tâche temps réel. À l'intérieur du système d'exploitation, ces deux types de blocage sont représentés à l'aide du même état. Afin de lever cette ambiguïté, certaines contraintes ont été émises afin de restreindre les possibilités. Ces contraintes ont été choisies de façon à pouvoir modéliser les états du modèle sans ajouter de points de trace supplémentaires. L'ajout de points de trace est indésirable car nous voulons limiter l'impact de l'instrumentation le plus possible. Ces restrictions permettent donc de contraindre le modèle et limiter les situations ambiguës, lorsque cela est possible.

L'idée de base provient de l'observation que les tâches temps réel utilisent des verrous qui implémentent l'héritage de priorité afin d'éviter le problème d'inversion de priorité. Les inversions de priorité peuvent retarder l'exécution d'un processus de haute priorité parce qu'une ressource dont il a besoin est réservée par un processus de faible priorité, qui ne s'exécute pas parce qu'un processus de priorité moyenne accapare le processeur. L'héritage de priorité permet au processus de faible priorité de s'exécuter comme s'il possédait la priorité du procesus de haute priorité et ainsi de libérer plus rapidement la ressource.

Notre première contrainte consiste donc à ce que les ressources partagées entre les tâches temps réel soient protégées à l'aide de primitives de synchronisation implémentant l'héritage de priorité. Ainsi, nous pouvons déterminer la raison qui a entraîné un processus à entrer en attente. Dans le cas d'un blocage qui est dû à la réservation d'une ressource, le processus qui possède la ressource verra sa priorité augmenter, de sorte qu'à l'intérieur de l'événement `sched_switch` annonçant le blocage, le prochain processus à s'exécuter aura une priorité égale à celle du processus bloqué. Si le prochain processus à s'exécuter avait une priorité inférieure, c'est que l'héritage de priorité n'a pas eu lieu et donc que l'attente n'est pas due à la réservation d'une ressource.

Notre deuxième contrainte consiste à ce que les tâches temps réel aient des priorités de base différentes. Si plusieurs tâches partageaient la même priorité, il ne serait pas possible de distinguer un blocage dû à une ressource verouillée et un blocage dû à une attente volontaire.

Lorsqu'il n'est pas possible pour une application de respecter ces contraintes, il faut alors ajouter des points de trace supplémentaires. Afin de limiter l'impact de cette instrumentation, il est possible d'instrumenter uniquement le moment où une tâche entre en attente à la fin de son exécution. De cette façon, un seul événement additionnel est nécessaire par exécution

de la tâche. Les périodes de blocage sont alors identifiées par l'absence de cet événement additionnel.

6.1.3 Systèmes multi-processeurs

Dans le cas de systèmes multi-processeurs, certaines modifications doivent être apportées à notre modèle. Nous avons émis précedemment l'hypothèse selon laquelle un processus qui est bloqué transfère sa priorité au processus fautif qui prendra donc sa place comme processus en exécution. Dans le cas d'un système à plusieurs processeurs, le processus ne s'exécutera pas nécessairement sur le même processeur. L'information contenue dans l'événement `sched_switch` ne concerne que le processeur courant. Cet événement n'est donc plus suffisant pour savoir si la priorité d'un processus a été augmentée. Heureusement, un autre événement permet de savoir précisément quand la priorité d'un processus est augmentée à cause de l'héritage de priorité : l'événement `sched_pi_setprio`. Dans un sytème multi-processeurs, il faudra donc activer cet événement lors du traçage. Le modèle devra également être modifié afin de permettre la conservation de l'information sur les processus qui ont eu leur priorité augmentée. Notons que cette information peut être stockée dans une liste et que dès qu'il est déterminé qu'un processus est bloqué à cause d'un accès à une ressource, la liste peut être effacée. Lorsqu'il s'exécutera à nouveau, ce sera nécessairement parce que la ressource sera disponible et que la priorité des processus fautifs ne sera plus augmentée.

6.1.4 Cas de test

Deux cas de test ont été présentés à la section 4.7. Ces deux cas ont été choisis de façon à représenter deux grands types de tâches temps réel : les tâches périodiques et les tâches sporadiques.

Dans le premier cas, l'application `cyclictest` a été utilisée afin de démontrer l'utilité de notre modèle pour extraire des statistiques et aider à comprendre les particularités d'une application connue de la communauté. Le comportement de `cyclictest` est très simple. Ses tâches sont indépendantes les unes des autres et ne partagent pas de ressources. Il s'agit donc d'un bon exemple afin d'introduire les concepts de base du modèle. Malgré sa simplicité, notre modèle a permis de découvrir des patrons particuliers dans l'histogramme de latence et de comprendre leur origine à l'aide de l'outil de visualisation associé.

Dans le deuxième cas, une application sur mesure a été développée afin de présenter le cas de tâches qui échangent de l'information et qui ne sont donc pas déclenchées périodiquement mais selon la disponibilité de données à consommer. Ce cas a permis de mettre en évidence l'avantage du traçage noyau comparativement à un traçage strictement dans l'espace uti-

lisateur. Grâce au modèle, il a été possible d'observer une période de blocage récurrente à l'intérieur du système d'exploitation. L'explication de phénomènes inattendus à l'intérieur du noyau n'est pas une tâche facile pour le développeur d'une application. Cependant, nous avons été en mesure de corréler cette période de blocage inattendue au correctif `PREEMPT_RT` utilisé pendant nos tests. Ceci démontre que l'information récupérée par notre modèle peut aussi être utile à un développeur noyau pour vérifier l'impact d'un correctif ou encore diagnostiquer un phénomène inattendu.

6.2 Limitations

Certaines suppositions ont été émises lors de l'élaboration du modèle sur la façon dont une tâche temps réel est implémentée. Particulièrement, il a été supposé qu'une seule tâche temps réel est exécutée sur un fil d'exécution. Cependant, il est possible qu'un même fil d'exécution soit en charge de plusieurs tâches simultanément. Dans un tel cas, les événements du noyau ne permettent pas de distinguer laquelle des tâches est en exécution. Un avantage important de notre modèle est qu'il est capable de séparer l'exécution d'un processus en exécutions individuelles d'une tâche récurrente. Si plusieurs tâches sont traitées par un même processus, les statistiques de toutes ces tâches sont fusionnées en une seule statistique globale. Une solution à ce problème serait de définir des événements manuellement que le modèle pourrait utiliser pour identifier chaque tâche de façon unique.

La vue présentée à la section 4.6 permet de comparer chaque exécution d'une tâche en les superposant les unes au-dessus des autres. Il est également possible de trier ces exécutions en fonction des métriques récupérées pour chacune et ainsi présenter l'information de différentes façons. Afin de pouvoir trier et afficher rapidement chaque exécution, les événements représentant chaque exécution sont conservés en mémoire. La vue est donc limitée, quant au nombre d'exécutions qu'elle peut afficher, par la mémoire disponible sur la machine d'analyse.

CHAPITRE 7

CONCLUSION ET RECOMMANDATIONS

7.1 Synthèse des travaux

Dans ce mémoire, nous avons abordé la problématique de l'analyse d'un système temps réel grâce aux informations de traçage. Les problèmes qui surviennent dans les systèmes temps réel sont traditionnellement difficiles à observer car il n'est pas possible de conserver les contraintes temporelles avec une instrumentation trop lourde. Le traçage permet ainsi de récupérer de l'information sur l'exécution d'une application temps réel. Le traçage n'est pas une panacée en soi. Les traces ainsi récoltées peuvent être volumineuses et il devient difficile d'extraire l'information souhaitée.

Notre objectif principal consistait à identifier les concepts propres aux applications temps réel qui aideraient à la compréhension et à l'analyse du système. Pour cela, nous avons utilisé deux cas de test qui représentent deux grandes familles de tâches temps réel, soient les tâches périodiques et les tâches sporadiques. À partir des ces deux types de tâches, nous avons identifié le comportement d'intérêt qui peut être extrait d'une trace. Nous nous sommes attardés à l'aspect récurrent des tâches temps réel, qu'elles soient périodiques ou non. En particulier, nous voulions observer les sources de latence dans le système. Ces latences sont causées par deux phénomènes observables : la préemption par une tâche de haute priorité ou bien un blocage à l'entrée d'une section critique.

Cette analyse initiale a permis de définir un modèle grâce auquel les tâches peuvent être modélisées. Le modèle a été défini à l'aide d'une machine à états de telle sorte qu'il soit rapide à générer et ne nécéssite qu'une seule lecture de la trace. Les événements nécessaires à la définition de ce modèle ont été identifiés afin de fournir l'information nécessaire tout en évitant de surcharger l'application de points de trace inutiles. Un des avantages de ce modèle est qu'il permet de diviser une tâche temps réel en ses exécutions constitutives. Grâce à cette segmentation, il est possible de récolter des statistiques individuelles sur chaque exécution et de générer des histogrammes.

En plus de ces statistiques, une vue a été développée afin de présenter les exécutions les unes par dessus les autres, de façon à facilement les comparer. Des problèmes de latence ont été présentés et expliqués en utilisant cette vue. Premièrement, il a été démontré que cette vue permet d'observer le comportement régulier d'une application. Puis, la vue a été utilisée pour diagnostiquer un problème précis et trouver la source du problème.

Enfin, une amélioration à la vue des statistiques de TMF a été décrite qui permet l'affichage de statistiques qui évoluent de façon continue dans le temps. Cette amélioration permet de calculer rapidement une statistique continue à l'intérieur d'un intervalle quelconque sans erreur d'échantillonage. Grâce à cela, il est possible de récupérer le temps d'exécution d'un processus à n'importe quel intervalle de la trace en deux requêtes au système d'états.

7.2 Améliorations futures

Les améliorations futures des nos travaux se divisent en deux domaines : la généralisation du modèle et la diversification des analyses.

Dans le cadre de ces travaux, nous avons présenté une approche qui permet de modéliser une application temps réel à partir des informations contenues dans une trace. Ce modèle permet de décrire une application générique sous certaines conditions. Il peut être nécessaire dans certains cas particuliers de devoir rajouter des points de trace dans l'application afin d'observer un comportement non décrit pas notre modèle. Dans ce cas, il serait pertinent de permettre la redéfinition du modèle selon les événements disponibles dans la trace. Il pourrait ainsi être possible de définir ses propres états et ses propres statistiques. D'une même façon, il serait possible de définir ses propres vues.

Au delà des statistiques et des outils de visualisation, il pourrait être intéressant d'utiliser un modèle afin d'effectuer la vérification d'un système temps réel. Le modèle représenterait alors les spécifications du système. En traçant le système, il serait possible de déterminer si les exigences de la spécification sont respectées ou non. Une intégration avec les outils de développement permettrait de relier directement ce modèle avec les modèles utilisés lors de la conception du système. Dans ce mode de vérification, il ne serait plus nécessaire de stocker la trace sur disque. Il serait alors possible d'effectuer l'analyse en temps réel et de simplement enregistrer les moments qui correspondent à des situations d'intérêt.

RÉFÉRENCES

[1] S. Rostedt. (2010, Mar) Using the trace_event() macro (part 1). [en ligne]. Disponible sur : http://lwn.net/Articles/379903/

[2] S. Goswami. (2014, Apr) An introduction to kprobes. [en ligne]. Disponible sur : https://lwn.net/Articles/132196/

[3] (2014, Mar) Debugging with gdb. [en ligne]. Disponible sur : https://sourceware.org/gdb/onlinedocs/gdb/index.html

[4] B. P. Miller et A. R. Bernat, "Anywhere, any time binary instrumentation," September 2011.

[5] F. C. Eigler et R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium*, 2006, pp. 261–268.

[6] J. Corbet. (2012, May) Uprobes in 3.5. [en ligne]. Disponible sur : https://lwn.net/Articles/499190/

[7] S. Rostedt, "Finding origins of latencies using ftrace," *Proc. RT Linux WS*, 2009.

[8] J. Edge. (2009, Jul) Perfcounters added to the mainline. [en ligne]. Disponible sur : http://lwn.net/Articles/339361/

[9] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, et T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[10] M. Desnoyers et M. R. Dagenais, "The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006, 2006, pp. 209–224.

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

[12] M. Desnoyers. (2013, Dec) Common trace format (ctf) specification (v1.8.2). [en ligne]. Disponible sur : http://git.efficios.com/?p=ctf.git;a=blob;f=common-trace-format-specification.txt

[13] Rapita Systems Ltd. (2014) Rapita verification suite. [en ligne]. Disponible sur : http://www.rapitasystems.com/products/rvs

[14] J. Han, H. Cheng, D. Xin, et X. Yan, "Frequent pattern mining : current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[15] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[16] J. Han, J. Pei, et Y. Yin, "Mining frequent patterns without candidate generation," in *ACM SIGMOD Record*, vol. 29, no. 2, 2000, pp. 1–12.

[17] R. Agrawal et R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, 1995, pp. 3–14.

[18] J. Pei, H. Pinto, Q. Chen, J. Han, B. Mortazavi-Asl, U. Dayal, et M.-C. Hsu, "Prefixspan : Mining sequential patterns efficiently by prefix-projected pattern growth," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2001, pp. 0215–0215.

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

[20] Z. Li, Z. Chen, S. M. Srinivasan, et Y. Zhou, "C-miner : Mining block correlations in storage systems." in *FAST*, 2004, pp. 173–186.

[21] C. LaRosa, L. Xiong, et K. Mandelberg, "Frequent pattern mining for kernel trace data," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 880–885.

[22] S. Laxman et P. S. Sastry, "A survey of temporal data mining," *Sadhana*, vol. 31, no. 2, pp. 173–198, 2006.

[23] B. Ozden, S. Ramaswamy, et A. Silberschatz, "Cyclic association rules," in *Proceedings of the 14th International Conference on Data Engineering*, 1998, pp. 412–421.

[24] C. Berberidis, W. G. Aref, M. Atallah, I. Vlahavas, et A. K. Elmagarmid, "Multiple and partial periodicity mining in time series databases," in *ECAI*, vol. 2, 2002, pp. 370–374.

[25] H. Cao, D. W. Cheung, et N. Mamoulis, "Discovering partial periodic patterns in discrete data sequences," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2004, pp. 653–658.

[26] S. Ma et J. L. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *Proceedings of the 17th International Conference on Data Engineering*. IEEE, 2001, pp. 205–214.

[27] H. Waly, "Automated fault identification : Kernel trace analysis," Mémoire de maîtrise, Université Laval, 2011.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

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

[30] W. De Pauw et S. Heisig, “Zinsight : a visual and analytic environment for exploring large event traces,” in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 143–152.

[31] (2014) Linux tools project - lttng integration. [en ligne]. Disponible sur : http://www.eclipse.org/linuxtools/projectPages/lttng/

[32] A. Montplaisir-Gonçalves, “Stockage sur disque pour accès rapide d’attributs avec intervalles de temps,” Mémoire de maîtrise, École Polytechnique de Montréal, 2012.

[33] D. F. Bacon, P. Cheng, D. Frampton, et D. Grove, “Tuningfork : Visualization, analysis and debugging of complex real-time systems,” *IBM Research, RC24162*, 2007.

[34] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, et W. E. Nagel, “The vampir performance analysis tool-set,” in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[35] (2013, Dec) Cpuset management utility. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/Cpuset_Management_Utility

[36] (2013, Oct) Cyclictest. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/Cyclictest

[37] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, et M. Desnoyers, “Recovering system metrics from kernel trace,” *OLS (Ottawa Linux Symposium)*, pp. 109–116, 2011.

[38] 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.

[39] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, et M. Santana, “Debugging embedded multimedia application traces through periodic pattern mining,” in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT ’12. New York, NY, USA : ACM, 2012, pp. 13–22. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/2380356.2380366

[40] B. Sprunt, L. Sha, et J. Lehoczky, “Scheduling sporadic and aperiodic events in a hard real-time system,” DTIC Document, Rapport technique, 1989.