



Titre: Runtime Verification of Real-Time Applications Using Trace Data and
Title: Model Requirements

Auteur: Raphaël Beamonte
Author:

Date: 2016

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Beamonte, R. (2016). Runtime Verification of Real-Time Applications Using Trace
Citation: Data and Model Requirements [Thèse de doctorat, École Polytechnique de
Montréal]. PolyPublie. <https://publications.polymtl.ca/2363/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/2363/>
PolyPublie URL:

**Directeurs de
recherche:** Michel Dagenais
Advisors:

Programme: Génie informatique
Program:

UNIVERSITÉ DE MONTRÉAL

RUNTIME VERIFICATION OF REAL-TIME APPLICATIONS USING TRACE DATA
AND MODEL REQUIREMENTS

RAPHAËL BEAMONTE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

RUNTIME VERIFICATION OF REAL-TIME APPLICATIONS USING TRACE DATA
AND MODEL REQUIREMENTS

présentée par: BEAMONTE Raphaël

en vue de l'obtention du diplôme de: Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de:

Mme BOUCHENEB Hanifa, Doctorat, présidente

M. DAGENAIS Michel, Ph. D., membre et directeur de recherche

M. GUIBAULT François, Ph. D., membre

M. KHENDEK Ferhat, Ph. D., membre externe

“ *Anyone who stops learning is old, whether at twenty or eighty. Anyone who keeps learning stays young. The greatest thing in life is to keep your mind young.* ”

Henry Ford

*À ma mère et mon père, pour l'éducation,
l'amour et le support que j'ai toujours reçus.*

ACKNOWLEDGMENTS

I wish firstly to thank Michel Dagenais, my research advisor, for the opportunity to work in a field that I am passionate about as well as for the support he has given me throughout my Ph.D., and the confidence he granted me from day one and has constantly renewed. His deep interest in the search for knowledge and understanding was a vector to always go further.

I would also like to acknowledge the financial support for my research project by OPAL-RT, CAE, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ) for the first year, and Ericsson, EfficiOS, and the NSERC for the following years.

I also thank all my past and present colleagues from the DORSAL laboratory for the countless discussions that have contributed in one way or another in this project or in maintaining my sanity. In particular, I would like to emphasize the collaborations and exchanges with Julien Desfossez, Matthew “Zoidberg” Khouzam, Yannick Brosseau, Geneviève Bastien and Suchakrapani Datt Sharma. Special thanks also to the multiple generations of colleagues of the “small room” M-3216, for the always fun but still serious ambience, and particularly to Mathieu Côté who jump started me on the state system, and Loïc Prieur-Drevon with whom I had the chance to work on related research and articles.

I would also like to thank my friends from my former *Association des Étudiants des Cycles Supérieurs de Polytechnique* (AÉCSP) teams, with whom we have lived beautiful moments as well as went through bad ones, but always kept smiling and moving forward. In addition, thanks to Samuel Kadoury for having entrusted me with teaching to hundreds of students during my Ph.D., whom also contributed to my growing up as a person. These moments “outside research” allowed me to take a step back and be more serene to face my project.

Finally, I would like to thank those of my family, relatives and friends, who have always been confident in my skills and inspired me to constantly push myself to reach my dreams, while supporting me at the same time.

RÉSUMÉ

Surveiller les systèmes multi-cœurs est difficile en raison des processus s'exécutant en parallèle, et pouvant interférer les uns avec les autres lorsqu'il s'agit d'accéder aux ressources du système, ou simplement d'avoir du temps processeur. Un tel système peut avoir à suivre des principes temps réel, ajoutant des contraintes de temps qui invalident les résultats dès qu'une date limite est manquée. Sur ce genre de système, des données précises n'auront ainsi de valeur que si elles peuvent être produites en respectant un délai donné. Le traçage peut fournir une grande quantité d'informations d'exécution de haute précision, à la fois sur le système et les applications. Il est ainsi souvent l'outil le plus précis et fiable pour étudier et analyser des systèmes ainsi contraints. Cependant, les utilisateurs doivent disposer d'une grande expertise du système afin de comprendre les événements du noyau du système et leur signification. De plus, il peut être très long d'analyser et étudier manuellement de volumineuses traces d'exécution de haute précision.

Dans cette thèse, nous proposons les méthodes et algorithmes permettant d'automatiquement détecter et identifier l'origine de comportements inattendus dans des applications, et ce à l'aide de traces de leurs exécutions et de modèles des exigences. Nous décrivons la structure interne des modèles, la méthodologie pour suivre l'exécution d'une application à travers ses événements de l'espace utilisateur, et les structures de données nécessaires pour vérifier les contraintes. Nous détaillons ensuite le processus utilisé pour détecter et finalement comprendre la source du comportement non désiré. Nous proposons aussi une approche pour construire automatiquement les modèles pour des applications temps réel courantes.

L'hypothèse servant de point de départ pour ce travail est que les traces d'exécution du système et de l'application à analyser peuvent être utilisées pour automatiquement suivre l'exécution de cette application, y détecter les anomalies et trouver leurs sources.

Les résultats de ce travail sont les concepts, méthodologies et structures de données utilisés pour suivre et contraindre des applications, ainsi que les méthodes et algorithmes qui permettent de détecter et identifier les comportements inattendus dans ces applications. Ces derniers ont été testés sur de réelles applications temps réel, et ont permis avec succès de détecter et identifier l'origine des irrégularités à leur exécution. De plus, nous avons pu automatiquement, et de façon précise, construire des modèles pour ces applications. Cette dernière étape rend l'utilisation des méthodes de traçage beaucoup plus accessible aux utilisateurs non-experts.

Le résultat final est que la détection automatique et la localisation automatique de la source des comportements inattendus dans une application est une option viable et fonctionnelle, qui accélère et simplifie le travail des utilisateurs pour analyser les exécutions de leurs applications.

ABSTRACT

Monitoring multi-core systems is hard because of the concurrently running processes that can contend with each other to access resources of the system or CPU time. Such a system may have to follow real-time principles, adding time constraints that invalidate results as soon as a deadline is missed. This means that accurate data will only be valuable if it can be produced in a timely fashion. Tracing is often the most accurate and reliable tool to study and analyze those systems, as it can provide a lot of high precision runtime information about both the system and applications. Nevertheless, a deep level of expertise of the system is required in order for the users to understand the kernel events and their meaning. Moreover, it can be time consuming to manually analyze and study voluminous high precision execution traces.

In this thesis, we propose methods and algorithms to automatically detect and identify the origin of unwanted behaviors in applications, by using traces of their execution and models of the requirements. We describe the internal structure of the models, the methodology to follow an application runtime through its userspace events, and the data structures needed to verify constraints. We then detail the process followed to detect and finally understand the root cause of the unwanted behavior. We also propose an approach to automatically build the models for common real-time applications.

The hypothesis serving as starting point for this work is that execution traces of both the system and the application to analyze can be used to automatically follow this application's execution, detect its anomalies and find their root causes.

The results of this work are the concepts, methodologies and data structures used to follow and constrain applications, as well as the methods and algorithms allowing to detect and identify unexpected behaviors in those applications. These have been applied on actual real-time applications and succeeded in detecting and identifying the root causes of the irregularities in their runtime. Moreover, we were able to automatically and accurately build models for those applications, making it even easier for non-expert users to take advantage of tracing methods.

The final result is that automatically detecting and pinpointing the origin of unwanted behaviors, in an application, is a valid and interesting option, making it faster and easier for users to analyze executions of their applications.

CONDENSÉ EN FRANÇAIS

Les développeurs ont besoin d'outils d'analyse système pour faire leur travail correctement. Les débogueurs permettent d'arrêter l'exécution de l'application afin d'identifier les causes probables des problèmes, et souvent de se déplacer pas à pas pour identifier le moment exact où le problème est apparu. S'ils sont pratiques, le fait d'arrêter et de pauser une application pendant son exécution ne fonctionne pas dans certaines situations, comme dans les cas d'applications temps réel où les contraintes de temps sont aussi importantes que la validité des résultats. Les traceurs fournissent beaucoup d'information sur ce qu'il s'est passé dans le système à un moment donné, et ce qui a conduit à cette série d'événements. Les différents traceurs existant ont tous leurs propres caractéristiques, poids et niveau de précision. Parmi eux, LTng permet d'instrumenter le code du noyau et de l'espace utilisateur avec des horloges coordonnées, de collecter un grand nombre de données sur le système à l'aide des compteurs de performance et des contextes ajoutés, et de fonctionner dans un contexte temps réel avec une latence ajoutée de moins de 10 μ s. Le faible impact de ce traceur, et la quantité d'information qu'il est capable de fournir, le rendent très intéressant pour analyser des systèmes multi-cœurs et temps réel.

Cependant, les outils d'analyse de trace, même graphiques, ne sont pas simples à employer pour les utilisateurs non familiers. Les traces d'exécution peuvent contenir des millions d'événements, et l'utilisateur a le choix entre une analyse manuelle ou l'exécution d'analyses permettant de mettre en valeur des éléments spécifiques dans la trace. Le public visé par ces outils d'analyse est principalement les développeurs, capables de comprendre comment le système d'exploitation fonctionne, et qui pourront ainsi utiliser les informations fournies par la trace afin de comprendre le comportement de leur application. Il est donc toujours nécessaire d'avoir une connaissance avancée et suffisante pour identifier les problèmes d'exécution de l'application avant d'analyser leur cause probable.

L'objectif de cette thèse est ainsi de fournir un moyen d'automatiquement détecter et identifier la cause de comportements non souhaitables pour une application donnée, à travers les traces noyau et espace utilisateur de son exécution, en utilisant des contraintes spécifiées dans un langage de modélisation simple.

Cette automatisation se découpe en trois étapes majeures : la définition et l'organisation du système qui va nous permettre de détecter les problèmes, l'analyse de l'origine de ces problèmes une fois détectés, et enfin la déduction de modèles à partir de traces.

La représentation des modèles se base sur quatre éléments connectés : les états, qui représentent les différents états de l'application, les transitions, qui représentent les mouvements d'un état vers lui-même ou un autre, les variables, qui sont utilisées pour récupérer et stocker les valeurs des métriques vérifiées dans les contraintes, et les contraintes, qui sont utilisées pour exprimer les attentes pour l'exécution de l'application.

Le système d'états est un arbre d'attributs qui maintient une modélisation de l'état du système d'exploitation et des applications. Il est basé sur l'arbre d'attributs d'états de Trace Compass, pour lequel la base de données d'historique d'états contient les métriques auxquelles on souhaite pouvoir accéder plus tard. Les valeurs pour ces métriques sont ainsi calculées et stockées lors de la lecture initiale de la trace noyau, ce qui permet d'y accéder facilement par la suite, via des requêtes à la base de données d'historique d'états.

Trois catégories de variables sont définies : les variables indépendantes de l'état courant du système, les compteurs et les minuteurs. Les variables sont catégorisées en fonction du nombre de requêtes qu'il est nécessaire de faire à la base de données d'historique d'états pour calculer leur valeur à un temps donné. Il faut ainsi respectivement 0, 1 et 2 requêtes pour calculer la valeur d'une variable indépendante de l'état du système, d'un compteur et d'un minuteur. Chaque variable aura un type qui sera dans l'une de ces catégories. Par exemple, les variables de type « date limite » sont indépendantes, alors que les variables pour compter le nombre de préemptions sont des compteurs.

Une contrainte est associée à une transition. Elle est donc vérifiée lorsque cette transition est réalisée. Étant donné que nous travaillons avec des traces d'exécution, une transition sera réalisée même si certaines de ses contraintes sont invalides. Une contrainte peut être incertaine si les données sont manquantes pour calculer la valeur de la variable à laquelle elle s'applique. Une transition sera valide si toutes ses contraintes sont valides, incertaine si au moins une de ses contraintes est incertaine et aucune n'est invalide, ou invalide si au moins une de ses contraintes est invalide.

Les états et transitions sont utilisés pour suivre les instances de l'application. Une instance correspond à une exécution de l'application avec un *tid* donné. À chaque étape de l'instance, c'est-à-dire lorsqu'une transition est utilisée pour se déplacer à l'état suivant, l'instance sera vérifiée. On évaluera ainsi le statut de l'étape de l'instance comme étant valide, incertain ou invalide, et ce dernier sera passé à l'état général de l'instance.

L'étape d'analyse est basée sur la comparaison de ce qu'il s'est passé dans les cas où une contrainte était valide, à ce qu'il s'est passé lorsque la contrainte était invalide, en identifiant les points communs aux situations invalides. Dans le contexte d'une trace, cela correspond à identifier les événements ou listes d'événements qui peuvent être significatifs, et les comparer

aux évènements ou listes d'évènements qui étaient attendus, afin de mettre en avant les discordances.

Avec notre structure, une fois qu'une trace a été analysée, nous obtenons une liste d'instances de l'application qui suivent le modèle défini. Ces instances contiennent le détail de leurs étapes, ce qui correspond à une liste ordonnée des états du modèle rencontrés, ainsi que les évènements qui ont enclenché chaque transition. Une étape de l'instance contient elle-même les informations sur toutes les contraintes qui ont été vérifiées à cette étape, ainsi que l'étape à laquelle la variable utilisée dans la contrainte a été initialisée pour la dernière fois. Il est donc possible d'identifier l'ensemble des étapes d'instances similaires pour lesquelles une contrainte était concernée, et de séparer les cas où cette contrainte était valide de ceux où elle était invalide. Les cas où la validation était incertaine sont mis de côté, étant donné que nous ne pouvons être sûrs de quel serait le statut de l'étape d'instance en question. Les étapes d'instance étant considérées par contrainte, il est tout à fait possible qu'une instance qui était invalide pour une contrainte soit considérée comme valide pour une autre.

Pour chaque contrainte pour laquelle au moins un cas était invalide, on lancera donc le processus d'analyse. Cependant, comme beaucoup d'instances peuvent être dans les traces, toutes les analyser prendrait du temps. Nous utilisons donc de l'échantillonnage avec sélection aléatoire de l'échantillon pour limiter le nombre d'instances à analyser. Cet échantillonnage sera appliqué d'un côté pour les instances valides, et de l'autre pour les instances invalides. Pour les instances sélectionnées, il est nécessaire d'identifier les éléments clés, ou éléments d'intérêt, qui donneront les informations nécessaires pour déterminer l'origine du problème. Ces éléments diffèrent selon le type de la variable utilisée pour la contrainte. Les compteurs et les minuteurs utiliseront ainsi les données sauvegardées dans le système d'états. Ceci permet de directement faire une recherche pour les dates auxquelles la valeur de la variable a changé au cours de l'instance étudiée. Une fois ces estampilles de temps obtenues, nous pouvons directement lire depuis la trace les évènements qui ont provoqué ces changements de valeur, et en extraire les informations souhaitées. Cette information pourrait par exemple être le nom de l'appel système pour un compteur d'appels système, ou le statut du processus reporté pour un minuteur d'usage du processeur. En ce qui concerne les variables indépendantes, les éléments clés sont tous ceux qui pourraient avoir une influence sur la valeur de cette variable. Par exemple, dans le cas d'une variable de date limite, nous avons créé un attribut dans le système d'états qui suit l'état du processus tout au long de la trace. Ainsi, chaque changement d'état du processus pendant la période est extrait comme étant un élément d'intérêt.

Les éléments d'intérêt ainsi extraits seront par la suite stockés comme des durées d'élément. Chaque durée d'élément contient l'élément ainsi que l'incrément causé par l'apparition de

cet élément à ce moment. Toutes les durées trouvées pendant une période de temps sont par la suite agrégées dans un ensemble de durées d'élément. Cet ensemble contient ainsi un certain nombre de durées qui peuvent ou non concerner un même élément. Un ensemble peut être vide, ce qui signifie qu'aucun élément n'a été trouvé pendant la période analysée. Pour une liste d'instances, les ensembles de durées d'élément similaires sont ensuite réunis dans un ensemble d'intervalles d'élément. On considère deux ensembles de durées comme étant similaires s'ils partagent la même configuration de clés, c'est-à-dire le même nombre de durées concernant les mêmes éléments. Lorsqu'on agrège des ensembles de durées dans des ensembles d'intervalles, on considère simplement leur contenu et non leur ordre d'apparition sur la période. Ainsi, lorsqu'on crée un ensemble d'intervalles à partir d'un ensemble de durées, on obtient un nombre d'intervalles avec des valeurs minimum et maximum identiques, correspondant aux différentes durées de l'ensemble d'origine. Lorsqu'on agrège un ensemble de durées dans un ensemble d'intervalles existant, les durées de l'ensemble de durées seront appariées avec les intervalles de l'ensemble d'intervalles dans le but de garder les intervalles résultats aussi petits que possible. Si les ensembles de durée utilisés pour former un ensemble d'intervalles sont vides, alors ce dernier sera vide aussi.

L'étape d'analyse termine par l'attribution des responsabilités aux éléments extraits pour identifier clairement les éléments qui ont le plus à voir avec la violation de la contrainte. Deux algorithmes sont utilisés selon le type et la valeur de la contrainte utilisée. Si la contrainte est absolue, autrement dit si n'importe quel changement à la valeur de la variable pendant la période analysée est prohibé, on choisira l'algorithme d'analyse partielle pour calculer directement les responsabilités pour tous les éléments qui ont mené à un changement de valeur. On assignera donc les responsabilités minimum et maximum pour chaque élément comme étant leur pourcentage d'implication dans respectivement le minimum total et le maximum total de l'ensemble d'intervalle. On calculera ensuite la responsabilité d'un élément comme étant la moyenne de ces deux valeurs. Dans le cas où la contrainte n'est pas absolue, c'est-à-dire que des changements à la valeur de la variable sont utilisés, l'algorithme d'analyse complète est nécessaire afin de réaliser la comparaison des ensembles car on ne peut pas considérer directement qu'un élément fait partie du problème. Ainsi, on identifie les différents cas valides et invalides représentés par des ensembles d'intervalles d'élément. Pour chacun des cas invalides, la distance à chaque cas valide est calculée, et un poids de proximité leur est associé. Ce sont les cas valides aux poids les plus élevés qui seront ensuite utilisés afin de les soustraire à l'intervalle invalide traité. Cette soustraction se fait selon l'opérateur utilisé pour la contrainte, et retourne un ensemble différentiel d'intervalles local. Si plusieurs ensembles différentiels locaux sont obtenus, ils sont agrégés selon un processus appelé « inter-union » qui fait une intersection sur les éléments contenus, et une union sur

les valeurs des intervalles. Ce processus permet d'éliminer des éléments qui pourraient être considérés comme problématiques dans un cas, mais qui ne le sont pas dans un autre, tout en élargissant les intervalles si nécessaire. Finalement, on calcule la responsabilité locale d'un élément par rapport à son pourcentage d'implication dans l'intervalle différentiel local. Si plusieurs instances invalides aux configurations de clés différentes ont été découvertes, ces responsabilités locales seront alors réunies pour former la responsabilité globale de chaque élément concerné. Le diagnostic peut enfin être posé en utilisant les responsabilités calculées. Certains cas cependant nécessitent des analyses plus précises. Lorsque l'un de ces cas a une responsabilité élevée, les analyses correspondantes seront automatiquement déclenchées. Les résultats de ces analyses passeront ensuite au travers des mêmes algorithmes afin d'affiner le résultat présenté à l'utilisateur.

La dernière partie de l'automatisation du processus consiste à automatiser la génération du modèle à partir de la trace. Afin de déterminer premièrement le déroulement de l'application, c'est-à-dire ses états et transitions, il faut analyser la trace de l'espace utilisateur qui contient les événements générés par l'application. Cependant, les événements dans la trace de l'espace utilisateur peuvent concerner de multiples applications et processus. Les modèles utilisés pour l'analyse suivent des processus, il est donc important d'identifier les différents déroulements selon les processus impliqués. Pour cela, la première phase consiste à organiser les événements de la trace par fil d'exécution, tout en conservant leur ordre d'apparition.

Il est possible que plusieurs fils d'exécution partagent le même déroulement d'exécution. Ainsi, en utilisant un algorithme de plus longue sous-séquence commune sur les séquences d'événements des processus, on peut les regrouper en un nombre limité de déroulements communs. Comme les événements peuvent contenir des différences au niveau de leur contexte que l'on pourrait vouloir éliminer, comme par exemple une information sur le numéro de processus pour une application multi-processus, on a défini un taux minimum de regroupement qui, lorsqu'il n'est pas atteint, réduit les contraintes d'identité pour associer les événements. Ce comportement n'est effectif que si le taux de regroupement idéal – lorsque l'on ne considère que les noms des événements et pas leur contenu – atteint le taux minimum de regroupement.

Les différents déroulements d'exécution identifiés peuvent eux-mêmes contenir des répétitions internes dues à une boucle par exemple. Dans ce cas, conserver ces répétitions ne permet pas de refléter un modèle précis de l'exécution de l'application. Ce problème s'apparente à celui de la plus longue sous-chaîne répétée, auquel nous ajoutons une condition de non-chevauchement. Chaque déroulement est traité séparément étant donné que le problème est distinct pour chaque cas. On construit alors un arbre des suffixes, duquel on extrait un tableau des suffixes, que l'on utilise enfin pour trouver la plus longue sous-chaîne répétée sans

chevauchement. Cette chaîne peut encore contenir des répétitions qui pourraient représenter les multiples exécutions successives d'une boucle. Ainsi, si la même séquence d'évènements est répétée encore et encore dans la chaîne en question, nous la réduisons à une seule occurrence.

La machine à états peut enfin être construite en suivant les différents déroulements d'exécution déterminés. Pour chaque déroulement, si on considère une séquence de n évènements $e_0 \dots e_{n-1}$, on aura une machine avec n états $s_0 \dots s_{n-1}$. Dans ce modèle, on se déplacera de l'état s_i à l'état s_{i+1} lorsqu'on lira l'évènement e_{i+1} pour $0 \leq i < n-1$. Pour finaliser la boucle générale du modèle, étant donné qu'il est impossible de savoir où le modèle débute vraiment, puisque la trace peut avoir commencé après le début de l'application, on se déplacera de l'état s_{n-1} à l'état s_0 lorsqu'on lira l'évènement e_0 . On créera finalement des transitions initiales en direction de tous les états disponibles en utilisant les évènements correspondants.

Chaque modèle individuel sera ensuite contraint. On ajoutera pour cela une variable de chaque type, ce qui peut être limité par l'utilisateur, sur chaque état du modèle individuel. Sur chaque transition, une contrainte sera placée pour chaque variable de ce même modèle. On utilisera ici un type de contrainte particulier : des contraintes adaptatives, appelées ainsi puisqu'elles adaptent leur opérateur et la valeur de comparaison aux données lues dans la trace. Elles ne sont pas très coûteuses en temps processeur puisque, comme des contraintes de base, elles sont traitées en partie au fur et à mesure de la lecture de la trace. La différence est que le traitement fait à cette étape est simplement d'enregistrer, à chaque fois que la contrainte est rencontrée, la valeur de la variable à contraindre. Une fois la lecture de la trace terminée, ces listes de valeurs sont traitées par différents algorithmes dans une étape de re-validation pour compléter la contrainte de ses parties manquantes, que ce soit l'opérateur, la valeur, ou les deux, et lui donner un statut. Le statut de la contrainte adaptative sera par la suite propagé à l'étape d'instance puis à l'instance pour mettre à jour leurs statuts. On terminera par nettoyer le modèle des contraintes et variables inutiles. En effet, il est possible qu'à la fin de la lecture de la trace, certaines contraintes n'aient pas pu déterminer de valeur ou d'opérateur à utiliser. Dans ce cas, cette contrainte n'est pas utile, et elle sera retirée du modèle. Toute variable qui ne sera plus contrainte sera elle aussi supprimée. Le modèle résultant sera finalement présenté à l'utilisateur pour qu'il le vérifie, l'améliore et le valide. L'analyse pourra alors être exécutée sur ce modèle.

Tous ces algorithmes ont été testés sur des cas d'études concrets et réels. Pour l'ensemble de ces cas, nous avons pu détecter les problèmes des applications dans les traces d'exécution, analyser de façon précise avec succès l'origine de ces problèmes, et générer automatiquement les modèles de ces applications qui nous ont permis d'obtenir les mêmes résultats d'analyse que nos modèles créés manuellement. Le temps pris pour le suivi du modèle dans la trace

et la détection des contraintes se met à l'échelle de façon linéaire par rapport au nombre d'évènements de l'espace utilisateur, avec un facteur plus élevé correspondant au nombre de requêtes à faire dans la base de données d'historique d'état. Par ailleurs, remplir cette base de données est linéairement proportionnel au nombre d'évènements dans la trace noyau. Enfin, cette durée est linéairement proportionnelle au nombre d'états dans le modèle, et au nombre de contraintes à vérifier. Le nombre de transitions entre deux états n'influe cependant pas sur le temps d'analyse. La mise à l'échelle de l'analyse des origines des problèmes a été testée en fonction du nombre d'instances, du nombre de changements de valeur pour une variable pendant une instance invalide, de la position de l'instance invalide dans la trace, et ce pour les analyses partielles et complètes, et tout ou une partie des évènements noyau activés. Ces tests ont montré que l'analyse peut être exécutée en un temps proportionnel à tous ces éléments, excepté la position de l'instance invalide qui n'a globalement pas d'impact. Finalement, la construction automatique du modèle augmenté par des contraintes peut se faire en un temps proportionnel au carré du nombre d'états à déterminer, au carré du nombre d'instances du déroulement d'exécution par fil d'exécution, au nombre de fils d'exécution exécutant une instance de l'application, et au nombre de déroulements d'exécution différents. Le temps pris pour traiter les contraintes adaptatives est directement lié au nombre d'instances de l'application dans la trace, tandis que le temps de nettoyage du modèle est relié à la taille du modèle généré.

L'ensemble de nos algorithmes peuvent s'exécuter dans un temps raisonnable pour conserver l'expérience utilisateur, tout en fournissant des éléments de réponse détaillés, et en simplifiant l'accessibilité au traçage pour des utilisateurs avertis mais non-experts. Ces approches permettront donc d'élargir la communauté du traçage, de même que de simplifier et accélérer l'analyse des traces.

En conclusion de cette recherche, nous pouvons affirmer qu'il est possible d'utiliser les traces d'exécution d'une application pour automatiquement détecter et identifier avec précision et rapidité la cause de comportements non souhaitables pour une application donnée, et ce en utilisant les traces noyau et espace utilisateur de son exécution ainsi qu'un modèle contraint des attentes, ce dernier pouvant aussi être généré automatiquement.

Notre approche est orientée pour les systèmes temps réel mais peut très bien s'appliquer sur d'autres types de systèmes. Des types de variable pourraient être intégrés pour profiter des informations des traces noyau que nous n'utilisons pas encore. De plus, un nouveau niveau d'analyse pourrait être ajouté afin de conseiller l'utilisateur sur les actions à prendre selon les résultats qui lui sont présentés, comme par exemple augmenter la priorité de son processus si ce dernier est préempté. Finalement, toutes nos analyses s'exécutent *a posteriori*. Il serait

intéressant de profiter du mode enregistreur de vol de LTTng afin de fournir une détection « en direct » des problèmes que l'on pourrait rencontrer. Il serait ainsi possible, pendant que l'application s'exécute, de construire le modèle, démarrer la détection, prendre un instantané de la trace en cas de contrainte invalide, et analyser l'origine du problème dans la trace ainsi obtenue.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
CONDENSÉ EN FRANÇAIS	viii
TABLE OF CONTENTS	xvi
LIST OF TABLES	xxi
LIST OF FIGURES	xxiii
LIST OF ALGORITHMS	xxxvi
LIST OF SIGNS AND ABBREVIATIONS	xxxvii
CHAPTER 1 INTRODUCTION	1
1.1 Concepts	1
1.2 Problem	4
1.3 Hypothesis	4
1.4 Research objectives	5
1.5 Claim for originality	6
1.6 Outline	6
CHAPTER 2 LITERATURE REVIEW	8
2.1 Software tracing and profiling tools for real-time applications	8
2.1.1 Kernel tracers	8

2.1.2	Userspace tracers	23
2.2	Common analysis methods and tools for real-time applications	25
2.2.1	Model checking oriented analysis tools	25
2.2.2	Data extraction tools	30
2.3	Constraints representation and modeling formats	37
2.3.1	UML	37
2.3.2	State Chart XML	39
2.4	Pattern recognition and model inference	39
2.5	Literature review conclusion	41
CHAPTER 3	METHODOLOGY	43
3.1	Research steps	43
3.1.1	Analysis system definition and organization	43
3.1.2	Detection and analysis of the problems	44
3.1.3	Automated model inference	44
3.2	Overview of the articles	45
CHAPTER 4	ARTICLE 1: DETECTION OF COMMON PROBLEMS IN REAL-TIME AND MULTICORE SYSTEMS USING MODEL-BASED CONSTRAINTS	46
4.1	Abstract	46
4.2	Introduction	47
4.3	Related work	48
4.3.1	Existing software userspace tracers	48
4.3.2	Model-checking analysis and data extraction tools for traces	50
4.4	Using model-based constraints to detect unwanted behaviors	52
4.4.1	General representation	52
4.4.2	Specific constraints on metrics	57
4.5	Case studies	62
4.5.1	Occasional missing of deadlines	62

4.5.2	Priority inversion	64
4.5.3	Unefficient synchronization method	67
4.5.4	Wait-blocked processes on multiprocessor activity	68
4.5.5	Wait-blocked processes while using external resources	70
4.6	Analysis results and time	72
4.6.1	Constraints validation	72
4.6.2	Running time	73
4.6.3	Scalability	78
4.7	Conclusion and Future Work	82
4.8	Disclosure	83
4.9	Competing Interests	83
4.10	Acknowledgments	83
CHAPTER 5 ARTICLE 2: MODEL-BASED CONSTRAINTS OVER EXECUTION TRACES TO ANALYZE MULTI-CORE AND REAL-TIME SYSTEMS		
5.1	Abstract	84
5.2	Introduction	85
5.3	Related work	86
5.3.1	Software tracers for both kernel and user-space	86
5.3.2	Automatic data extraction tools and model-checking for traces	87
5.3.3	Statistics algorithms for relation analysis	89
5.4	Using model-based constraints	89
5.4.1	Internal structure	90
5.4.2	Model representation	90
5.5	Algorithms for data organization and extraction	92
5.5.1	Data organization	93
5.5.2	Data extraction	94
5.6	Case studies	108
5.6.1	Too low priority	108

5.6.2	In-kernel wake lock priority inversion	112
5.6.3	Bad userspace code	114
5.6.4	Frequency scaling	118
5.6.5	Preempted waker	119
5.7	Running time and scalability	120
5.7.1	Running time	122
5.7.2	Scalability	126
5.8	Conclusion and Future Work	131
5.9	Acknowledgments	132
CHAPTER 6	ARTICLE 3: MODEL-BASED CONSTRAINTS INFERENCE FROM RUNTIME TRACES OF COMMON MULTICORE AND REAL-TIME APPLICATIONS	133
6.1	Abstract	133
6.2	Introduction	134
6.3	Related work	135
6.3.1	Software tracers to simultaneously trace user-space and kernel	135
6.3.2	Automatic analysis of traces with model-based constraints and data extraction	137
6.3.3	Model generation and pattern detection	138
6.4	Model-based constraints	140
6.5	Generation of the model workflow	142
6.5.1	Organization of the trace events	142
6.5.2	Identification of the common workflows	142
6.5.3	Removal of the unneeded repetitions	146
6.5.4	Build and clean the model	146
6.6	Adaptive constraints	149
6.6.1	Adaptive comparison value	150
6.6.2	Adaptive comparison operator	152

6.7	User interface	156
6.8	Case studies	156
6.8.1	JACK2	156
6.8.2	cyclictest	159
6.8.3	In-kernel wakelock application	165
6.9	Running time and scalability	167
6.9.1	Running time	170
6.9.2	Scalability	173
6.10	Conclusion and Future Work	178
6.11	Acknowledgments	179
CHAPTER 7	COMPLEMENTARY RESULTS	180
7.1	JACK2	180
7.2	cyclictest	184
7.3	In-kernel wakelock application	187
CHAPTER 8	GENERAL DISCUSSION	194
8.1	Objectives achievement	194
8.2	Contributions	196
8.3	Limitations	197
CHAPTER 9	CONCLUSION AND RECOMMENDATIONS	198
9.1	Synthesis of work	198
9.2	Recommendations	198
REFERENCES	200

LIST OF TABLES

Table 4.1	Number of events and sizes of the traces used to benchmark our analysis	72
Table 4.2	Average (<i>Avg.</i>) and standard deviation (<i>Std. dev.</i>) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace <i>tk-preempt (1)</i>	76
Table 4.3	Average (<i>Avg.</i>) and standard deviation (<i>Std. dev.</i>) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace <i>tk-preempt (2)</i>	77
Table 4.4	Average (<i>Avg.</i>) and standard deviation (<i>Std. dev.</i>) of the time taken in seconds to build the state system during the first run, versus to verify if it exists in the subsequent runs, computed using 100 runs of the analysis of the traces	78
Table 5.1	Computed distance sets and values between the invalid duration set represented in Figure 5.9 and the valid interval set in Figure 5.6 for different constraints on a target value x	104
Table 5.2	Real-time priorities of the processes in the setup used to generate the trace for the in-kernel wake lock priority inversion. An empty priority means that the process was not started.	114
Table 5.3	Number of events and sizes of the traces used to measure our analysis running time	121
Table 5.4	Results of the instances duration analysis for the traces of the cases studied, which computes information about the similar instances steps in the valid and invalid instances	122
Table 5.5	Statistical comparison between the use of sampling or not for the process state analysis, in terms of number of instances analyzed, analysis execution duration and result (percentage of responsibility for the highest responsible state for added time); the highest responsible state was, in all cases, identical within 0.1% whether sampling was used or not; statistics were computed with 100 runs of the analysis for each situation.	123

Table 5.6	Statistics on the execution duration of the different parts of our analysis process, computed with 100 runs of the analysis for each case studied in 5.6	125
Table 6.1	Organization per thread of the events read in the simplified trace example presented in Figure 6.2	144
Table 6.2	Example of the process used to regroup similar sequences to identify the different workflows by solving the longest common subsequence problem on the sequences of events. This example uses the sequences of events provided in Figure 6.3	145
Table 6.3	Real-time priorities of the processes in the setup used to generate the trace for the in-kernel wakelock application. An empty priority means that the process was not started.	166
Table 6.4	Number of events and sizes of the traces used to measure our analysis running time	170
Table 6.5	Statistics on the execution duration of the different parts of our model building process, computed with 100 runs of the analysis for each case studied in 6.8	171
Table 6.6	Statistics on the execution duration of the different parts of our model building process for <code>cyclictest</code> , while using the whole trace or only a randomly selected 15 ms chunk of it, computed with 100 runs of the analysis	172

LIST OF FIGURES

Figure 2.1	DTrace architecture [22]	14
Figure 2.2	Processing steps and components of SystemTap [24]	17
Figure 2.3	Ktap architecture [27]	19
Figure 2.4	General architecture of LTTng [32]	21
Figure 2.5	RCU flow to handle resources [33]	22
Figure 2.6	Simplified diagram of the trace recording of LTTng-UST	24
Figure 2.7	The Tango system [44]	26
Figure 2.8	Trace analysis methodology used in [46]	28
Figure 2.9	Overall architecture of the tracing work flow of KOJAK [50]	30
Figure 2.10	Performance analysis workflow of Scalasca [57]	32
Figure 2.11	CUBE3 window used to analyze Scalasca results [58]	33
Figure 2.12	Conceptual overview of SETAF’s workflow [60]	35
Figure 2.13	Perspective of Trace Compass [62]	36
Figure 4.1	State machine representation that can be used to check metrics using traces	54
Figure 4.2	State machine representation with late verification of constraints and a transitional state	55
(a)	The verifications will use initializations that only appear at state “S”	55
(b)	The verifications will use both initializations declared at different states	55
Figure 4.3	State machine representation with multiple next states for state “S”	56
Figure 4.4	State machine representation using a loop, to go over the initializations when reading an event of type “event (1)”	56
Figure 4.5	State machine representation of a constraint validating whether we spent at most 2 ms between the states “S” and “S+1”	57
Figure 4.6	State machine representation of a constraint validating that our process has not been preempted between the states “S” and “S+1” . . .	58

Figure 4.7	State machine representation of a constraint validating whether our process used at most 1 % of the CPU time between states “S” and “S+1”	59
Figure 4.8	State machine representation of constraints validating whether the process spent at most 10 % of the time period between states “S” and “S+1” waiting for a CPU, and at most 15 % of this same period being blocked	60
Figure 4.9	State machine representation of a constraint validating that our process has not done any system call between states “S” and “S+1” . .	61
Figure 4.10	Preemption of a process by another of higher priority (−2 vs −21, the lower the value, the higher the priority)	63
(a)	Screenshot of Trace Compass showing the preemption	63
(b)	Trace event “sched_switch” happening to do the preemption	63
Figure 4.11	State machine representation of tk-preempt ’s work using constraints to check if our process spent at most 45 ms working, used 100 % of the CPU time and was not preempted during its critical real-time task	64
Figure 4.12	Unexpected kernel work while tracing an userspace-only application	66
(a)	Screenshot of Trace Compass showing the period between two “npt:loop” events in the application	66
(b)	Kernel events traced between the two “npt:loop” events showing kernel tasks running while the application is waiting to continue its work, causing latency	66
Figure 4.13	State machine representation of npt ’s loop using constraints to check if our process used 100 % of the CPU time between each loop iteration	67
Figure 4.14	Screenshot of Trace Compass showing the barrier at which threads are waiting before unmapping operations, after their calls to munmap [92]	69
Figure 4.15	Views showing wait situations while using external resources; these views do not exist in the Trace Compass mainline version yet [94] .	71
(a)	Unified CPU-GPU view showing the process unscheduled from its CPU causing wait on the GPU side	71
(b)	Unified CPU-GPU view showing the process waiting for the GPU while the GPU is still working on another task	71

Figure 4.16	Results of the analysis using the model-based constraints on userspace and kernel traces	74
(a)	Result shown following the analysis of both kernel and userspace traces when the constraints are satisfied	74
(b)	Result shown following the analysis of both kernel and userspace traces when the constraints are not satisfied	74
(c)	Result following the analysis of the userspace trace only to simulate a case where we would not have any kernel trace, thus making the state system unavailable for the analysis	74
Figure 4.17	Examples of invalid sections as reported by our tool for other cases discussed in section 4.5	75
(a)	Invalid section for the case presented in 4.5.2 when verifying the model represented in Figure 4.13, during which four (4) system calls were issued	75
(b)	Invalid section for the case presented in 4.5.3 when verifying a deadline constraint of less than one (1) second for a task length, which lasted nearly five (5) seconds in this case	75
(c)	Invalid section for the case presented in 4.5.4 when verifying a wait blocked constraint of less than ten percent (10%) for a task, which spent more than fifteen percent (15%) of its time being blocked in this case	75
Figure 4.18	Time (in s) to build the instances and check their constraints as a function of the number of userspace events. Lines represent linear regressions of the data.	79
Figure 4.19	Time (in s) to build the state system as a function of the number of kernel events. The line represents a linear regression of the data.	80
Figure 4.20	Time (in ms) to build the instances as a function of the number of successive states in the model	81
Figure 4.21	Time (in ms) to build the instances as a function of the number of transitions between two states	81
Figure 4.22	Time (in s) to build the instances and check their constraints, as a function of the number and categories of constraints between two states	82

Figure 5.1	State machine representation of tk-preempt 's real-time process using constraints to check that our process was not preempted during its critical real-time task	91
(a)	Graphical representation of the period of JACK2	91
(b)	XML representation of the critical real-time task of tk-preempt using State Chart XML	91
Figure 5.2	Example of a trace containing 4 instances, following a model specifying that the instance execution duration should be of 13 units of time; 3 of the instances are valid, and 1 is invalid with a duration of 15 units of time.	93
Figure 5.3	When merging duration sets, only the content is considered and not the order, which means that all the <i>keymaps</i> presented here are considered identical	96
Figure 5.4	Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the valid instances which are in equal number for each element, but with different order or duration.	97
Figure 5.5	The three duration sets extracted from the three valid instances presented in Figure 5.4; the durations contained in the set are shown in the simplified form of the number of durations for a given element and the sum of these durations	98
(a)	The duration set of the first valid instance contains two durations for the element RUNNING that can be summed to 6 units of time, one duration for the element SYSCALL for 3 units of time, one duration for the element BLOCKED for 1 units of time and one duration for the element PREEMPTED for 3 units of time	98
(b)	The duration set of the second valid instance contains two durations for the element RUNNING that can be summed to 6 units of time, one duration for the element SYSCALL for 4 units of time, one duration for the element BLOCKED for 1 units of time and one duration for the element PREEMPTED for 2 units of time	98

(c)	The duration set of the third valid instance contains two durations for the element RUNNING that can be summed to 7 units of time, one duration for the element SYSCALL for 3 units of time, one duration for the element BLOCKED for 1 units of time and one duration for the element PREEMPTED for 2 units of time	98
Figure 5.6	Interval set resulting from the merge of the three duration sets in Figure 5.5; it contains two intervals for the element RUNNING that are both between 1 and 5 units of time, one interval for the element SYSCALL between 3 and 4 units of time, one interval for the element BLOCKED for 1 unit of time and one interval for the element PREEMPTED for 2 to 3 units of time	99
Figure 5.7	Distance calculation step by step, showing the distance for intermediate sets	100
Figure 5.8	Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the invalid instance, that differ in number or in duration when compared to valid instances.	103
Figure 5.9	The duration set extracted from the invalid instance presented in Figure 5.8; it contains two durations for the element RUNNING that can be summed to 4 units of time, one duration for the element SYSCALL for 4 units of time, one duration for the element PREEMPTED for 2 units of time and two duration for the element IRQ that can be summed to 5 units of time	103
Figure 5.10	ALSA driver being configured by the JACK2 daemon process after it was started with a low real-time priority of 1 and pinned on CPU 0 [106]	108
Figure 5.11	Algorithm intermediate results for the analysis of JACK2	110
(a)	Set of intervals of time spent in the different process states for the invalid instances of JACK2	110
(b)	Set of intervals of time considered to be in excess in the invalid interval set compared to the valid interval sets with a weight of at least 50 %	110
(c)	Weighting of the valid interval sets against the invalid one in Figure 5.11(a), showing the number of occurrences of each interval set, its distance to the invalid interval set and its computed weight . . .	110

Figure 5.12	Sections of the analysis report generated by our analysis for JACK2	111
(a)	State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process waits to be woken up	111
(b)	Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process is preempted	111
(c)	CPUTop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 0, where JACK2 was running, during the period of time for which JACK2 was preempted	111
Figure 5.13	Schema of a priority inheritance while waiting for a resource: the lowprio1 process takes the lock of the resource and is preempted by a higher priority process highprio1 , but can finish its task and free the lock when inheriting temporarily the priority of process highprio0	112
Figure 5.14	Schema of an in-kernel wake lock priority inversion: the lowprio1 process takes the lock of the resource and cannot free it for the higher-priority highprio0 process because lowprio1 is currently preempted by the highprio1 process, which has a higher priority than highprio0	113
Figure 5.15	Sections of the analysis report generated by our analysis for the in-kernel wake lock priority inversion	115
(a)	State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on an open system call	115
(b)	Priority inheritance analysis triggered by the state analysis, comparing the average time where the priority inheritance was active in invalid instances to the maximum time in valid instances	115
(c)	Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process waits after the lowprio1 process, which is preempted	115
(d)	CPUTop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 1, where lowprio1 was running, during the period of time for which lowprio1 was preempted	115

Figure 5.16	Graphical representation of the period within our userspace application	116
Figure 5.17	Sections of the analysis report generated by our analysis for the bad userspace code	117
(a)	State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly running	117
(b)	State machine state analysis triggered by the state analysis, showing the responsibility of each state of the model for the added time in the invalid instances: in this case, the process spends most of its time in the “step3” state	117
Figure 5.18	Frequency scaling analysis, showing the CPU frequencies for valid and invalid instances, and comparing their average frequencies in order to identify if there is a probability of a frequency scaling problem . . .	119
Figure 5.19	Graphical representation of the period of <code>cyclictest</code>	119
Figure 5.20	Sections of the analysis report generated by our analysis for <code>cyclictest</code>	121
(a)	State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on the <code>rt_sigtimedwait</code> system call	121
(b)	Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process mostly waits after the <code>ktimersoftd/3</code> process, which is preempted	121
(c)	CPUTop analysis triggered by the critical path analysis, showing the processes that were running on CPU 3, during the period of time for which <code>ktimersoftd/3</code> was preempted	121
Figure 5.21	Execution duration of the initial variable analysis according to the number of instances in the trace, when only one of these instances is invalid; the original number of instances is 2; each data point is the average of 20 runs	127
Figure 5.22	Number of instances to analyze according to the number of valid or invalid instances in the trace, using (5.1) with $p = 50\%$, $z = 1.96$ and $e = 5\%$	128

Figure 5.23	Execution duration of the initial variable analysis according to the number of variable value changes during an instance; the number of variable value changes for a valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs	129
Figure 5.24	Execution duration of the initial variable analysis according to the position of the invalid instance among the 8192 instances in the trace; the number of variable value changes for an valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs	130
Figure 6.1	XML representation of the model using State Chart XML	141
Figure 6.2	Simplified example of a trace with events generated by four different processes, where the letters between quotes represent events that should be considered different because of their name or content . . .	143
Figure 6.3	Sequences of events per thread, for all the threads in the simplified trace example presented in Figure 6.2	143
Figure 6.4	Unconstrained workflow model built from the sequence of events [A B C]. This model links the events in the order in which they appear and allows for a loop.	147
Figure 6.5	Unconstrained workflow model built from all the sequence of events found in the example of trace shown in Figure 6.2. This model allows to enter any state identified from the trace, and to follow the related workflow.	148
Figure 6.6	User interface to configure the model generation, or choose an existing model to use for the detection	157
Figure 6.7	ALSA driver being configured by the JACK2 daemon process after it was started with a low real-time priority of 1 and pinned on CPU 0 [106]	158
Figure 6.8	Part of the content of JACK2's userspace trace used for the case study	158
Figure 6.9	Models to check the valid execution of JACK2 and detect unwanted behaviors	160
(a)	Model manually built to correspond to the period of JACK2	160

	(b)	Model automatically generated from a runtime trace of JACK2 , using all the variables types	160
Figure 6.10		Results of the detection on JACK2 's trace	161
	(a)	Results of the detection on JACK2 's trace using the model presented in 6.9(a), showing only the invalid cases	161
	(b)	Results of the detection on JACK2 's trace using the model presented in 6.9(b), showing only the invalid cases	161
Figure 6.11		Part of the content of cyclictest 's userspace trace used for the case study	162
Figure 6.12		Models to check the valid execution of cyclictest and detect unwanted behaviors	163
	(a)	Model manually built to correspond to the period of cyclictest . .	163
	(b)	Model automatically generated from a runtime trace of cyclictest , using only deadline variables	163
Figure 6.13		Results of the detection on cyclictest 's trace	164
	(a)	Results of the detection on cyclictest 's trace using the model presented in 6.12(a), showing only the invalid cases	164
	(b)	Results of the detection on cyclictest 's trace using the model presented in 6.12(b), showing only the invalid cases	164
Figure 6.14		Schema of a priority inheritance while waiting for a resource: the lowprio1 process takes the lock of the resource and is preempted by a higher priority process highprio1 , but can finish its task and free the lock when inheriting temporarily the priority of process highprio1	165
Figure 6.15		Schema of an in-kernel wake lock priority inversion: the lowprio1 process takes the lock of the resource and cannot free it for the higher-priority highprio0 process, because lowprio1 is currently preempted by the highprio1 process, which has a higher priority than highprio0	166
Figure 6.16		Models to check the valid execution of the in-kernel wakelock application and detect unwanted behaviors	168
	(a)	Model manually built to correspond to the period of the in-kernel wakelock application	168

	(b)	Model automatically generated from a runtime trace of the in-kernel wakelock application, using only deadline variables	168
Figure 6.17		Results of the detection on the wakelock application's trace	169
	(a)	Results of the detection on the wakelock application's trace using the model presented in 6.16(a), showing only the invalid cases	169
	(b)	Results of the detection on the wakelock application's trace using the model presented in 6.16(b), showing only the invalid cases of the "m0/*" part of the model	169
Figure 6.18		Execution duration of the different steps of our approach according to the number of states of the model to determine; all the runs use 1 instance per thread, 1 thread, and 1 workflow; each data point is the average of 20 runs	174
Figure 6.19		Execution duration of the different steps of our approach according to the number of instances of the model to determine; all the run use 4 states per instance, 1 thread, and 1 workflow; each data point is the average of 20 runs	175
Figure 6.20		Execution duration of the different steps of our approach according to the number of threads running each instance of the model to build; all the runs use 4 states per instance, 1 instance per thread, and 1 workflow; each data point is the average of 20 runs	176
Figure 6.21		Execution duration of the different steps of our approach according to the number of different model workflows to build; all the runs use 4 states per instance, 1 instance per thread, and 2048 threads; each data point is the average of 20 runs	177
Figure 7.1		Sections of the analysis report generated for JACK2, for the violation of the constraint on "deadline/d0.1" when entering state "m0/state0"	181
	(a)	Results of the state analysis	181
	(b)	Results of the critical path analysis, triggered by the state analysis	181
	(c)	Results of the CPUPop analysis, triggered by the critical path analysis	181
Figure 7.2		Sections of the analysis report generated for JACK2, for the violation of the constraint on "deadline/d0.1" when entering state "m0/state1"	182
	(a)	Results of the state analysis	182

	(b)	Results of the critical path analysis, triggered by the state analysis .	182
	(c)	Results of the CPUtop analysis, triggered by the critical path analysis	182
Figure 7.3		Sections of the analysis report generated for JACK2 , for the violation of the constraint on “deadline/d0.0” when entering state “m0/state0”	183
	(a)	Results of the state analysis	183
	(b)	Results of the critical path analysis, triggered by the state analysis .	183
	(c)	Results of the CPUtop analysis, triggered by the critical path analysis	183
Figure 7.4		State analysis generated for JACK2 , for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”	183
Figure 7.5		Sections of the analysis report generated for cyclictest , for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”	184
	(a)	Results of the state analysis	184
	(b)	Results of the critical path analysis, triggered by the state analysis .	184
	(c)	Results of the CPUtop analysis, triggered by the critical path analysis	184
Figure 7.6		Sections of the analysis report generated for cyclictest , for the violation of the constraint on “deadline/d0.0” when entering state “m0/state0”	185
	(a)	Results of the state analysis	185
	(b)	Results of the critical path analysis, triggered by the state analysis .	185
	(c)	Results of the CPUtop analysis, triggered by the critical path analysis	185
Figure 7.7		Sections of the analysis report generated for cyclictest , for the violation of the constraint on “deadline/d0.1” when entering state “m0/state1”	186
	(a)	Results of the state analysis	186
	(b)	Results of the critical path analysis, triggered by the state analysis .	186
	(c)	Results of the CPUtop analysis, triggered by the critical path analysis	186
Figure 7.8		State analysis generated for cyclictest , for the violation of the constraint on “deadline/d0.1” when entering state “m0/state0”	187

Figure 7.9	Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”	188
(a)	Results of the state analysis	188
(b)	Results of the priority inheritance analysis, triggered by the state analysis	188
(c)	Results of the critical path analysis, triggered by the state analysis .	188
(d)	Results of the CPUtop analysis, triggered by the critical path analysis	188
Figure 7.10	Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state2”	189
(a)	Results of the state analysis	189
(b)	Results of the priority inheritance analysis, triggered by the state analysis	189
(c)	Results of the critical path analysis, triggered by the state analysis .	189
(d)	Results of the CPUtop analysis, triggered by the critical path analysis	189
Figure 7.11	Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.0” when entering state “m1/state3”	190
(a)	Results of the state analysis	190
(b)	Results of the critical path analysis, triggered by the state analysis .	190
(c)	Results of the CPUtop analysis, triggered by the critical path analysis	190
Figure 7.12	State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.1” when entering state “m1/state3”	191
Figure 7.13	State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.2” when entering state “m1/state3”	191
Figure 7.14	State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.0” when entering state “m1/state2”	192

Figure 7.15	State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.1” when entering state “m1/state2”	192
Figure 7.16	State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d2.0” when entering state “m2/state1”	193

LIST OF ALGORITHMS

Algorithm 5.1	Distance calculation between an <i>interval</i> and a <i>duration</i> , taking into account the direction of the constraint C , the result being a numerical value	101
Algorithm 5.2	Distance calculation between an interval set <i>intSet</i> and a duration set <i>durSet</i> , taking into account the direction of the constraint C , the result being a duration set epresenting the distance per element . . .	102
Algorithm 5.3	Substraction of a valid interval set <i>validIntSet</i> from an invalid one <i>invalidIntSet</i> , taking into account the direction of the constraint C , the result being a differential interval set	106
Algorithm 6.1	Segmentation algorithm for a list of <i>data</i> , using a given <i>threshold</i> that can be expressed in absolute or relative terms. The result is a list of segmented values from the original data.	151
Algorithm 6.2	Algorithm used to determine the best operator and value to use for the constraint with a given list of runtime values V , a set of segments of these values s , and the index of the biggest segment <i>idxB</i>	154

LIST OF SIGNS AND ABBREVIATIONS

ANSI	American National Standards Institute
CCXML	Call Control eXtensible Markup Language
CPU	Central Processing Unit
CTF	Common Trace Format
FIFO	First In First Out
HRT	Hard Real-Time
I/O	Input/Output
IRQ	Interrupt ReQuest
LCS	Longest Common Sequence
LTtng	Linux Trace Toolkit next generation
MPI	Message Passing Interface
PLC	Programmable Logic Controller
PMU	Performance Monitoring Unit
QoS	Quality of Service
RCU	Read-Copy Update
RISC	Reduced Instruction Set Computing
RTOS	Real-Time Operating System
SCXML	State Chart eXtensible Markup Language
SHT	State History Tree
SMP	Symmetric MultiProcessor
SoftIRQ	Software Interrupt ReQuest
SRT	Soft Real-Time
UML	Unified Modeling Language
URCU	Userspace Read-Copy Update
UST	UserSpace Tracer
VM	Virtual Machine
XML	eXtensible Markup Language

CHAPTER 1 INTRODUCTION

System analysis tools are necessary for developers to do their work properly. Debuggers allow to stop the runtime of an application to identify a probable cause of problems, and often to move step by step to identify the exact moment when the problem occurs. However, for time sensitive applications, step by step execution is not possible. Tracers are then suitable and provide much information on what happened in the system at a specific time, and what leads to that series of events. The different existing tracers each have their own characteristics, weight and level of precision. It is thus necessary to choose wisely the tracer to use, depending on what we need to trace and what information we need to have in the trace. For instance, some tracers only allow to trace kernel events, while others also provide userspace tracing. Userspace tracing allows to understand the correlation between the application behavior and the kernel events happening in the background.

However, current tracing analysis tools, even with graphical user interfaces, are not easy to use for unfamiliar users, as discussed in Chapter 2. The public mostly targeted by such tools are developers, able to understand how the operating system works, and that will thus be able to use the tracing information to properly understand the application's behavior. An advanced understanding is currently still necessary to pinpoint where a problem appeared before analyzing its probable causes. It would therefore be interesting to target an informed but not expert population by simplifying the analysis of the trace. This represents a bigger audience, and can thus improve the industrial and academic tracing usage. Hence, given such observations on the current state of the analysis, our general objective is focused on improving and automating the analysis methods for a less sophisticated audience.

1.1 Concepts

A real-time system is a computer system taking into account time constraints [1]. These constraints are then as important as the accuracy of the results obtained, and a result will be invalidated if it is not received on time. We say that a real-time system must meet logical determinism – accuracy of the result – and temporal determinism – result obtained in the allotted time. There are two types of real-time systems depending on the importance of the time constraints:

- The soft real-time systems (SRT), where the events processed out of time or simply lost do not have catastrophic consequences for the smooth running of the system. For

example, multimedia systems – including streaming – for which it is possible to lose a few seconds of audio or video without jeopardizing the general operation of the system;

- The hard real-time systems (HRT), which correspond to systems that have to respond to stresses or events, internal or external, within a maximum time, thus requiring high temporal determinism. For example, the control systems for nuclear power plants or embedded aerospace systems are in this category.

A real-time application is one for which the compliance to time constraints during processing is as important as the outcome of processing. It is generally based on two deterministic criteria: logic, i.e. the same data processed must give the same result, and time, i.e. a given task must absolutely be done in a timely manner, or meet the deadline. These two determinisms create a number of situations in which a real-time application or system doesn't work and needs to be analyzed to identify the origin of the problems. This analysis cannot be performed in isolation as, doing so, we would remove the external factors that could be the ones responsible for the deadline misses.

Real-time systems can be found in control, location or security devices for instance. Multiple architectures exist for those systems, such as dedicated circuits, Real-Time Operating System (RTOS) like QNX [2], and more generic systems such as Linux PREEMPT_RT [3] which aims at finding a balance between flexibility, features and reliability.

The studied applications and systems are working with the official Linux kernel, or the PREEMPT_RT patched one. The latter mainly provides changes to the former for better scheduling. It aims at providing all the official kernel's features while guaranteeing reaction delays, i.e. the delay between a stimulation and the linked reaction. The main difference is that the PREEMPT_RT patched Linux kernel is fully preemptible, and manages the high priority routines – as the interruption managers – as normal processes to which we can assign priorities.

Hardware interruptions are signals sent by Input/Output peripheral devices to a processor. They aim to inform the operating system of multiple devices status (data to be read, data written, etc). Each device gets its own unique interruption id. When the operating system receives an interruption, it stops the current running process to execute the interruption manager. Software interruptions can also be emitted directly from the processor itself, in case of an exceptional situation or simply by reaching an interruption instruction. The number of different hardware interruptions is limited by the number of Interrupt ReQuest (IRQ) lines in the processor, while there can be hundreds of software interruptions, as there is no hardware limitation for Software Interrupt ReQuest (SoftIRQ).

On Linux kernels, a number of performance analysis tools are provided. Chapter 2 details the different analysis tools that allow to get both system and userspace information. We can distinguish five different tool categories: the debuggers that stop the application execution to take actions or identify the current state, the profilers which perform sampling to profile the system activity, the aggregators that run analyzes when specific events appears, the benchmark tools that measure system performance according to synthetic tests, and the tracers that save sequences of events in execution traces.

A step by step approach using a debugger would not work to analyze real-time applications, as the slowing factor would be too high, leading to potential changes in the application behavior. Indeed, stopping the application execution to analyze the current state would not stop the internal timers, and provoke deadline misses for instance. Debuggers are thus not a suitable solution for such analysis.

Profilers are also rarely used for sporadic performance analysis, as they are built to provide a global resource usage state, making it very unlikely to identify problems appearing only once or twice every thousands of application's execution. However, their measurement mechanism being active only for a fraction of the time, their impact on the application execution stays low.

Aggregators perform runtime analysis in the critical path of the system. This means that as soon as an event relevant for the analysis appears, they take control to perform their computation, and give control back. Thus, the analysis result is known almost immediately, but the drawback is the high overhead added by the analysis being in the critical path, particularly in concurrent systems. This is explained by the fact that such systems often aim at being generic, and thus lock every structure that could potentially be shared. Moreover, as the analysis is done while the analyzed system runs, if any data is missing the only solution would be to replicate the experiment and hope for the event that we try to explain to appear again.

The benchmark tools are a good solution to test and verify system characteristics. However, these are synthetic tests that usually do not represent the real runtime situations. Moreover, they cannot be used to check and validate the performance of a production system.

A tracer's objective is to output a coherent list of events in time, based on system activity. When working on performance analysis, we work mainly with tracers built for low system disturbance. This allows to perform sophisticated analysis, as a lot of information is obtained without slowing down the system too much. Tracing is thus often the more reliable and accurate solution for the analysis of real-time systems and applications. However, such

system's drawback is precisely the quantity of information, which requires a considerable time and expert human intervention to analyze and identify the problems.

Modeling is used to represent the workflow of an application. In formal verification, this is used to check that the given workflow is feasible and verifies a number of conditions defined on the model. Applying modeling to runtime analysis means to model the information that can be retrieved from the application while it runs. This is similar as modeling Message Passing Interface (MPI) communications for instance, where communication logs are analyzed through a given model to verify that the MPI interactions appeared in the right order from the right processes. Chapter 2 gives an overview of different approaches used to model and analyze applications and systems using models.

1.2 Problem

Modeling allows technical and non-technical users to specify how an application should work and the logical and time constraints to satisfy. Using models and traces could thus allow us to compare the specifications we want to satisfy and the real behavior. Current models allow to specify basic and complex interactions between the nodes of a model, but not all the annotations that could be useful for specifying runtime constraints, such as a minimum or maximum Central Processing Unit (CPU) usage, number of system calls, or number of preemptions. Such constraints could however be really useful for an operating systems analysis approach to modeling.

Using this kind of extended model with tracing data could then allow to identify if the traced application complies to the model constraints and, if this is not the case, identify what are the constraints that are not satisfied. Once a violation has been identified, statistical analysis and comparisons could improve the identification of the origin of the identified infringement.

1.3 Hypothesis

The hypothesis serving as starting point for this research is that it is possible to use traces of the operating system and the running applications, and more specifically a trace containing events to track a workflow of a specific application, to detect anomalies in the aforesaid application and identify the root cause of those anomalies. This involves following the state of the operating system, as well as the state of the application, all along the application runtime in the trace.

1.4 Research objectives

A few years ago, tracing tools and techniques were few and far between. The kernel tracing software was almost exclusively reserved to expert users with an in-depth knowledge of the operating system. The advantages of tracing instead of debugging are numerous, particularly for real-time systems where using breakpoints is not really possible, leading to the more widespread use of tracers for software development and system analysis by non-expert users.

There is now a wealth of available tracers, as seen in Chapter 2, but even if the user interfaces are becoming more user-friendly, tracing users still need to have a deep knowledge of the operating system to be able to extract the data from the generated traces. Automating trace analysis is an approach that could simplify application or system analysis. Even if some implementations already provide automatic pattern matching, and constraint and modeling verification, such systems are currently very specific to a programming language or require complex user interventions to either prepare the analysis or view the results.

Our objective is therefore to provide a way to automatically detect and identify the causes of unwanted behaviors of a given application, through its runtime kernel and userspace traces, using constraints specified in a simple modeling language. Such modeling and constraints may be applied at different levels. For a real-time application, for instance, such levels could be, but are not limited to, the resources usage, and the latency and deadlines. The user thus would not need to intervene during the analysis process. This process would be fully automatic, using the execution traces. The first step of this process is to automate the detection of unwanted behaviors using a model constrained with both kernel and userspace data, providing a precise indication of the unsatisfied constraint or constraints and the location of those infringements in the model. The second step is to provide further analysis to identify the root cause of the constraints violations when detected following the first step. The last step of the automation is to provide a way to automatically build simple linear models, usually sufficient to model the critical sections for common real-time applications, directly from the trace.

The specific objectives arising from the general objective are as follows:

- Propose a methodology allowing to use userspace traces of a real-time application, and its workflow model, to validate its runtime execution and latency constraints;
- Develop the algorithms and data structures to use kernel traces to add other levels of constraints such as resources usage;

- Develop algorithms to pinpoint the origin of constraints violations from its userspace and kernel traces and its workflow model;
- Provide the algorithms and methodology to automate the model building step for common real-time applications by using userspace and kernel traces.

The availability of the tools built on the results of this research would help to improve application development and anomaly detection, as well as enlarge the tracing community by facilitating the access to trace analysis. Most of this currently needs to be done by hand, which requires both time and expertise to do.

1.5 Claim for originality

The literature review presented in Chapter 2 helped us identify the gaps in the tracing and analysis area. We have seen that, even if different ways have been studied to provide automatic analysis of systems and applications using checkers or traces, the results of those researches are still not easily applicable for non-expert users. Our objective is to fill these gaps by providing a user-friendly analysis workflow, and thus allowing non-expert users to use tracing for the development of their software or the analysis of their applications.

1.6 Outline

The state of the art is presented in Chapter 2. It focuses on the software tracing tools for real-time applications, analysis methods for these systems and different prior work related to our research.

The methodology behind this research is presented in Chapter 3. It explains the different steps followed for our approaches, and details the requirements for each of those. It also depicts an overview of the three articles.

The core of the thesis consists in those three articles. The process to detect common problems in real-time and multi-core systems is presented in the article “Detection of Common Problems in Real-Time and Multicore Systems Using Model-Based Constraints” [4] in Chapter 4. We present the algorithms and structures used to provide those detections over execution traces and models. Our approach is proposed and tested for multiple common real-time and multi-core cases. This article has been published in *Scientific Programming*.

The root cause analysis after detecting a constraint violation is presented in the article “Model-based constraints over execution traces to analyze multi-core and real-time systems” [5] in Chapter 5. We detail the process, algorithms and statistics used to provide

a thorough analysis in a proper time. Our analysis is presented over different common real-time and multi-core cases. This article has been submitted to *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*.

The automatic generation of model-based constraints is presented in the article “Model-based constraints inference from runtime traces of common multicore and real-time applications” [6] in Chapter 6. We explain the process used to determine the workflow of an application directly from the userspace and kernel trace, as well as how we augment this workflow with model-based constraints. The model inference process is presented using multiple real-time and multi-core cases. This article has been submitted to *ACM Transactions on Software Engineering and Methodology*.

Chapter 7 presents additional results that were not included in the articles.

Chapter 8 discusses the results obtained and highlights the links between the results obtained in the individual articles.

Finally, we conclude the thesis in Chapter 9 by recalling the progress realized, the main research contribution, and proposing future work leveraging this research.

CHAPTER 2 LITERATURE REVIEW

This chapter presents the state of the art related to software tracing tools for real-time applications, analysis methods for these systems and how to automate the analysis.

2.1 Software tracing and profiling tools for real-time applications

Tracing is a computer systems performance analysis technique that allows to retrieve information about the operating system or an application during its execution, with minimal impact on that execution. Tracing differs from debugging in that its goal is not to stop the operations of the analyzed program, to inspect the state at any given time, but rather to let it run in order to retrospectively analyze the course of its execution. Furthermore, tracing provides an overview of the interactions between components of an application and those of the operating system on which it was running. Common divisions of tracing are made according to its functional aspect (static or dynamic instrumentation) or its intended use (kernel or userspace tracing), all of which will be addressed in this section, within the specific scope of this research.

In this section, we will focus on the various tracing tools available to trace the operating system kernel and on the subset that also have a component for tracing user space applications.

2.1.1 Kernel tracers

To trace the kernel, two different approaches are possible: dynamic and static instrumentation.

Dynamic instrumentation in the Linux kernel is made with `Kprobes` [7]. It allows to dynamically insert, at the desired place, a probe of the same type as a breakpoint, and the code to process it automatically and resume the execution which was suspended. This type of instrumentation, on the bright side, allows not to touch the kernel source code, and therefore not require recompilation. However, the cost of executing a dynamically inserted probe is larger than for a static probe and may alter more significantly the application timing, as it normally relies on a soft interrupt mechanism. This change in behavior and the added overhead make this kind of instrumentation unreliable for tracing a real time system.

Static instrumentation is, for its part, carried out with tracepoints statically added in the source code of the kernel. Kernel recompilation is required when adding such instrumentation points. It allows a more consistent behavior across a traced execution and an untraced one. The macro `TRACE_EVENT()` is used to create this type of tracepoints [8, 9, 10]. It was created based on the need to find a model suitable to operate in different situations for different user needs. Being flexible and easy to use, it is now the standard method for creating tracepoints in the Linux kernel. Once a tracepoint is created with this macro, any tracer supporting that format can use it. Tracers such as `perf`, `ftrace`, `SystemTap` or `LTTng` are able to use these tracepoints. Given the sensitivity to runtime conditions and latency in the context of real-time systems, tracers supporting static instrumentation will be preferred in the context of our research.

Feather-Trace

Feather-Trace [11] is a tracer using very lightweight static events. It is available to trace on Intel x86 architectures. Designed to trace real-time systems and applications, it is a low-overhead tracer: inactive tracepoints only cause the execution of one statement – an unconditional jump – while active tracepoints will cause the execution of two instructions – an unconditional jump and a system call. **Feather-Trace** is able to work in a multiprocessor environment and uses wait-free and multiprocessor-safe FIFO buffers. It is easily portable since this tracer does not require using the synchronization primitives of the operating system or any modification to the interrupt handler.

However, **Feather-Trace** does not support variable event sizes, since its memory reservation mechanism is based on table indexes, which limits the content of the events and their multiplicity, thus not allowing the user to add system context information that could be useful for a system-oriented analysis. **Feather-Trace** uses the system call `gettimeofday()` as time source for the traces, which only allows microsecond accuracy. It also does not include a trace writing mechanism. Finally, **Feather-Trace** only uses its own defined events and does not provide a way to use the standard `TRACE_EVENT()` macro to use system events. As a matter of fact, when **Feather-Trace** was used as a case study to analyze the locks in the Linux kernel, a patch was applied to insert its own tracepoints to allow the analysis [12]. Getting useful information on the system behavior while tracing with **Feather-Trace** would therefore be much more difficult to maintain along the kernel development.

Paradyn

Paradyn [13] is a tracer performing dynamic instrumentation by modifying the binaries to insert the calls to the tracepoints. Although it uses dynamic instrumentation that can be inserted at runtime, **Paradyn** is using a patch-based instrumentation to rewrite the binary (or even instrument the currently executing code) to impose only low-overhead latency. This technique was used to monitor and analyze the execution of malicious code.

While the tracing tool provides an extensive application programming interface (API) to change the executables, called **Dyninst**, it does not include a trace buffers management system, a way to define different types of events nor a trace writing mechanism. **Paradyn** does not allow to use the Linux kernel static tracepoints, nor to add system context information. Moreover, no assurance can be given about the tracing conditions on multi-core systems. Finally, the overhead imposed by **Paradyn** is proportional to the number of instrumented locations.

perf

The **perf** [14] tracer is one of the built-in Linux kernel tracers. It was originally designed to allow easy access to the performance counters located in the processors. Its use was later extended to interface with the `TRACE_EVENT()` macro and therefore access the tracepoints of the Linux kernel. It is now possible to use it to generate statistics on the number of times a tracepoint is executed, for example, or to analyze the number of events that a processor has recorded during a period. A tool, also called **perf**, is available in the kernel source to control the tracer.

The statistics generated by **perf**, using its hardware and software performance counters, are illustrated by a simplistic execution of the tool with the provided command line execution:

```
$ perf stat hackbench 10 >/dev/null
```

Performance counter stats for 'hackbench 10':

1568,083128 task-clock	#	3,352 CPUs utilized	
34 316 context-switches	#	0,022 M/sec	
3 503 CPU-migrations	#	0,002 M/sec	
29 148 page-faults	#	0,019 M/sec	
4 133 963 671 cycles	#	2,636 GHz	[87,27%]
1 756 375 029 stalled-cycles-frontend	#	42,49% frontend cycles idle	[88,52%]


```

    718 044 218 stalled-cycles-backend # 17,37% backend cycles idle [66,34%]
4 267 624 983 instructions           # 1,03 insns per cycle
                                     # 0,41 stalled cycles per insn
                                     [83,08%]
1 190 870 838 branches              # 759,444 M/sec [86,29%]
    9 433 482 branch-misses         # 0,79% of all branches [88,72%]

0,467785294 seconds time elapsed

```

This tracer is based on sampling. Whenever the limit set by the tracer is reached (e.g., 100000 cycles were elapsed), an interrupt is generated for the information about the current instruction and data to be recorded. Because of the low overhead associated with sampling, one could consider that **perf** would be interesting to trace real-time systems. However, sampling sacrifices accuracy and relies on much fewer samples than tracing, but each is more costly and invasive to collect because of the interrupt involved.

Perf can also function as a conventional tracer but has not been optimized for this use. Moreover, its multi-core scalability is limited and it is thus less interesting than **ftrace** [15].

ftrace

The **ftrace** [16] tracer is a set of different tracers built into the Linux kernel. It was originally developed for the real-time Linux kernel but has been integrated into the standard Linux kernel since version 2.6.27. This tracer, called **ftrace** for *Function Tracer*, was created in order to follow the relative cost of the functions called in the kernel and thus determine the bottleneck for a defined period of time. It has since evolved and now includes more comprehensive analysis modules such as the latency analysis or scheduling analysis [17]. The **ftrace** tracer has the advantage of not requiring an external tool to control or read the generated data, being managed through the pseudo-filesystem **debugfs**. **Ftrace** works through the activation and deactivation of its different tracers. To compare **ftrace** to the other tracers investigated in this research, we concentrate on the dynamic and event tracers of **ftrace**.

The dynamic tracer of **ftrace** [18, 19, 20] allows the dynamic insertion of tracepoints directly in the kernel through **Kprobe**. The file `/sys/kernel/debug/tracing/kprobe_events` is used to define tracepoints. Two types of tracepoints exist: the “**probes**”, that can be inserted anywhere in the kernel, according to the limitations of **Kprobe**, and the

“return probes”, that are inserted at the function exits. Once a tracepoint is defined, **ftrace** creates a folder named after the probe name in the folder `/sys/kernel/debug/tracing/kprobes`. This new folder contains four files to configure and manage the probe:

- The **enabled** file is used to activate or deactivate the tracepoint by setting the content of this file respectively to 1 or 0. Note that it is not possible to activate or deactivate a group of tracepoints at the same time, thus needing to use the **enabled** file of each concerned probe.
- The **format** file describes the format used to display the data from this tracepoint.
- The **filter** file allows to define filter rules for the concerned event. The filters are used as conditions for tracepoints. However, these filters are fairly limited in the conditions that they can express. Indeed, they can't use the code's variables using symbols and can only use logical operators. Moreover, they are used after the data has been collected. Filters thus cannot prevent the tracer from collecting unnecessary data in the case where a condition is not satisfied.
- The **id** file allows to show the tracepoint identifier.

The **ftrace** tracer cannot read the kernel debug information. It is thus impossible to define a tracepoint by specifying the line number of a source file or get the value of a variable by specifying its name. **Ftrace** is thus unable to evaluate and trace expressions using variables. All the recorded values are stored in hexadecimal format in the trace. The dynamic tracer of **ftrace** therefore requires its users to have a deep understanding of the kernel source code. Debugging tools are typically used to identify the variables locations in memory and the addresses of source code lines.

The event tracer of **ftrace** [21] allows to connect to the static tracepoints of the kernel. Just like **perf**, **ftrace** uses the macro `TRACE_EVENT()`. As for the dynamic tracer, the configuration and view of the trace is done using text files. For each tracepoint, there are four files, similar to those used for the dynamic tracepoints. Unlike the dynamic tracer, however, **ftrace** allows to activate or deactivate groups of static tracepoints at the same time. The available tracepoints are listed in the `/sys/kernel/debug/tracing/available_events` file. At compilation time, **ftrace** allocates for each static tracepoint a static memory space that will be used to save tracepoint details. This memory space will then be accessed to list the available tracepoints.

The event tracer also uses filters instead of conditions. These filters cannot use any variable accessible at the tracepoint position, since `ftrace` doesn't have access to the register values when the tracepoint is reached, and cannot read the kernel debugging information. Therefore, only the variables that were transmitted as parameter of the tracepoint function can be used by filters, and only with logical or comparison operators. The filters are, here again, applied after the data has been collected, thus potentially making the tracer collect unnecessary data. The execution time of an `ftrace` handler with an associated filter is almost always the same, regardless of the filter's outcome. The event tracer of `ftrace` only allows to collect the data as defined in the `TRACE_EVENT()` macro, using the `TP_printk` macro. No further data can be collected.

The following is an example of a trace generated with `ftrace` while all the static “`sched`”-type system tracepoints were activated. These tracepoints are the ones located in the scheduler.

```

trace-cmd-23807 [004] 287757.950139: sched_stat_runtime:
    comm=trace-cmd pid=23807 runtime=96455 [ns]
    vruntime=4402378808457 [ns]
trace-cmd-23807 [004] 287757.950143: sched_switch:
    trace-cmd:23807 [120] S ==> swapper/4:0 [120]
    ls-23808 [007] 287757.950165: sched_wakeup:
migration/7:42 [0] success=1 CPU:007
    ls-23808 [007] 287757.950166: sched_stat_runtime:
    comm=trace-cmd pid=23808 runtime=88101 [ns]
    vruntime=4439088182548 [ns]
    ls-23808 [007] 287757.950167: sched_switch:
    trace-cmd:23808 [120] R ==> migration/7:42 [0]
migration/7-42 [007] 287757.950168: sched_migrate_task:
    comm=trace-cmd pid=23808 prio=120 orig_cpu=7 dest_cpu=4
<idle>-0 [004] 287757.950170: sched_switch:
    swapper/4:0 [120] R ==> trace-cmd:23808 [120]

```

Each line in the trace represents an encountered tracepoint. A line is composed of a timestamp, to indicate the time when the event occurred, and the name of that event. The remaining information available in the line is the data collected according to the definition of the `TP_printk` macro.

`Ftrace` is a very flexible and configurable tracer. However, it is limited to kernel tracing.

DTrace

DTrace [22] was originally developed by Sun Microsystems with the aim of using it for kernel tracing in its Solaris platform, and was first implemented in Solaris 10. It was soon after ported to different operating systems such as MacOS, QNX or FreeBSD for instance. Like for other tracing techniques, DTrace is based on the concept of instrumentation. The user writes probe handlers using D, a C-like script language. The DTrace architecture uses special kernel modules, called “providers”, that instrument the kernel or applications and create “probes” allowing to gather data. Any provider can publish probes to which the DTrace framework can connect to gather data.

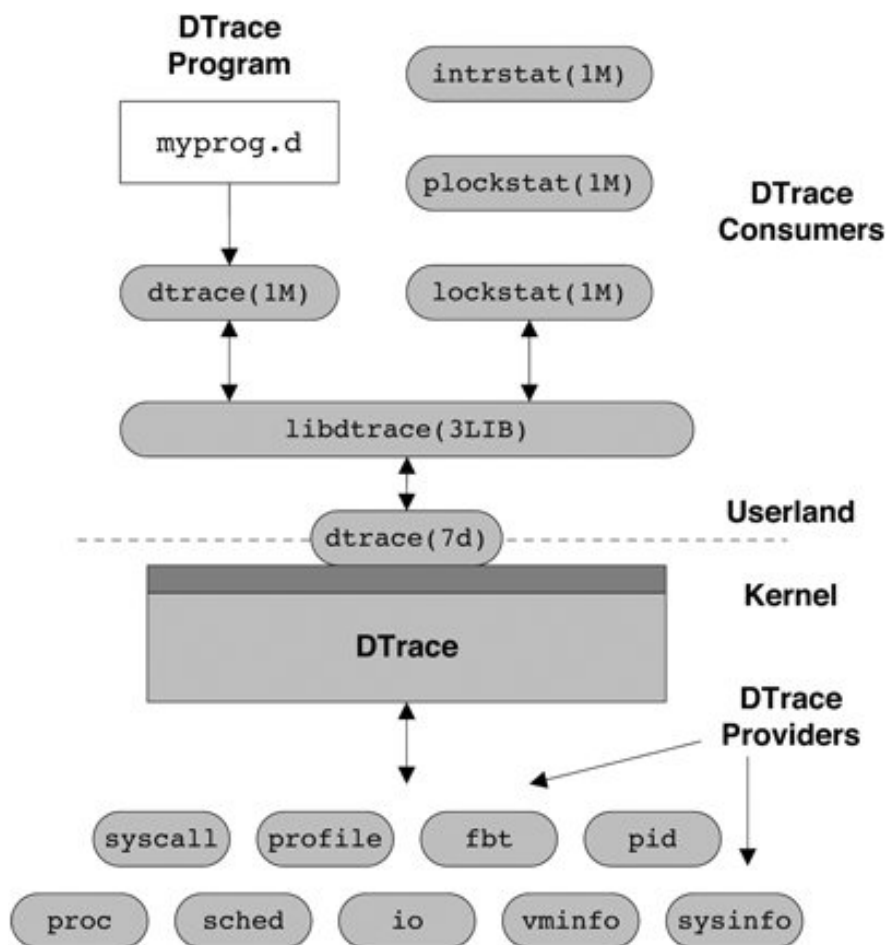


Figure 2.1 DTrace architecture [22]

The `libdtrace` library allows to interface the user-level trace consumers with the kernel driver `dtrace(7d)`. The general architecture of DTrace is shown in Figure 2.1. That library contains the D compiler that converts the D source to a machine independent format used for the virtual machine of DTrace, located in the kernel of the operating system. That indepen-

dent format is called **DIF** for *D Intermediate Format*, and represents a RISC-like instruction set understandable by **DTrace**'s register based virtual machine. When put together, multiple **DIF** objects are called **DOF** for *DTrace Object Format*. A **DOF** is a format to encode **DTrace** programs and contains details about the probe, the string and the variable tables. At instrumentation time, this **DOF** is injected into the kernel.

Following this instrumentation, each time the probe is encountered as the system is running, the **DTrace** framework will be called by the providers, and will then be able to carry out the tracing directives by using the `dtrace_probe()` function. A dynamic aspect of **DTrace** is that the tracepoints can be activated only when needed, and deactivated as soon as the data was collected [23].

Even if the tracing approach used by **DTrace** is very comprehensive, it is mostly interrupt-driven and thus adds delays to the overall process. Such delays are most often unacceptable for real-time use.

SystemTap

SystemTap [24] is a monitoring system for Linux. It is primarily aimed at the community of system administrators and provides scripts to interface with static instrumentation provided by the macro `TRACE_EVENT()`, or automatically insert dynamic instrumentation.

Just as for **DTrace**, the instrumentation and trace code in **SystemTap** is written in a special scripting language, with a syntax similar to **C**. The script is converted in **C** and then compiled to produce a kernel module that will communicate with **SystemTap** for tracing [25]. The script language supports all of **ANSI C** operations but only works with integers and strings as data types. One **SystemTap** script can be used to declare multiple instrumentation points.

An instrumentation point declaration is composed of a first part to target the event that we want to link to the instrumentation point, and a second part specifying the code to execute when the tracepoint is reached. Using the “**if**” instruction, these tracepoints can be conditional, as shown in the following example of a **SystemTap** script:

```
probe kernel.function("vfs_read")
{
    dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
    if (dev_nr > 5)
        printf("%x\n", dev_nr)
}
```

In this script, the instrumentation point is inserted at the entry of the `vfs_read` kernel function. We then extract the information we need from the `file` parameter of the function and copy it in a temporary variable, here called `dev_nr`. This variable is then used in the conditional statement, and in the `printf` statement if the condition is satisfied. A large number of events are available to be associated with instrumentation points in **SystemTap** scripts, including the following few examples listed below:

```
/* Function entry */
probe kernel.function("vfs_read").call

/* Function exit */
probe kernel.function("vfs_write").return

/* specific location in kernel code */
probe kernel.statement("*/fs/read_write.c:42")

/* Specific address in binary */
probe kernel.statement(0xc00424242)
```

SystemTap also supports a *guru mode* in which safety features of the script language, such as data and code memory reference protection, are removed. When the tool is in *guru mode*, the translator accepts C code to be enclosed in the script language. This C code will then be kept as is, in sequence, in the generated C code, without analysis. This mode thus allows to overcome the limitations of the script language.

Dynamic kernel tracepoints are made available in **SystemTap** by using **Kprobes**. They can be conditional by the use of an “if” statement, as shown in the previous example. **SystemTap** uses the **DWARF** debugging information, generated during the kernel compilation, to identify the addresses of the instrumentation points, as well as to resolve the references to the kernel variables that are used in the instrumentation scripts. **SystemTap** then uses the registers passed to the handler of the **Kprobe** to extract the values from these variables. The dynamic tracepoints can use all the variables that are available from the instrumentation point addresses. Figure 2.2 gives a detailed view of how **SystemTap** works.

It is also possible to connect **SystemTap** to the static kernel tracepoints that have been defined using the `TRACE_EVENT()` macro. A static tracepoint can be activated in a **SystemTap** script using the following `probe`:

```
probe kernel.trace("event_name")
```

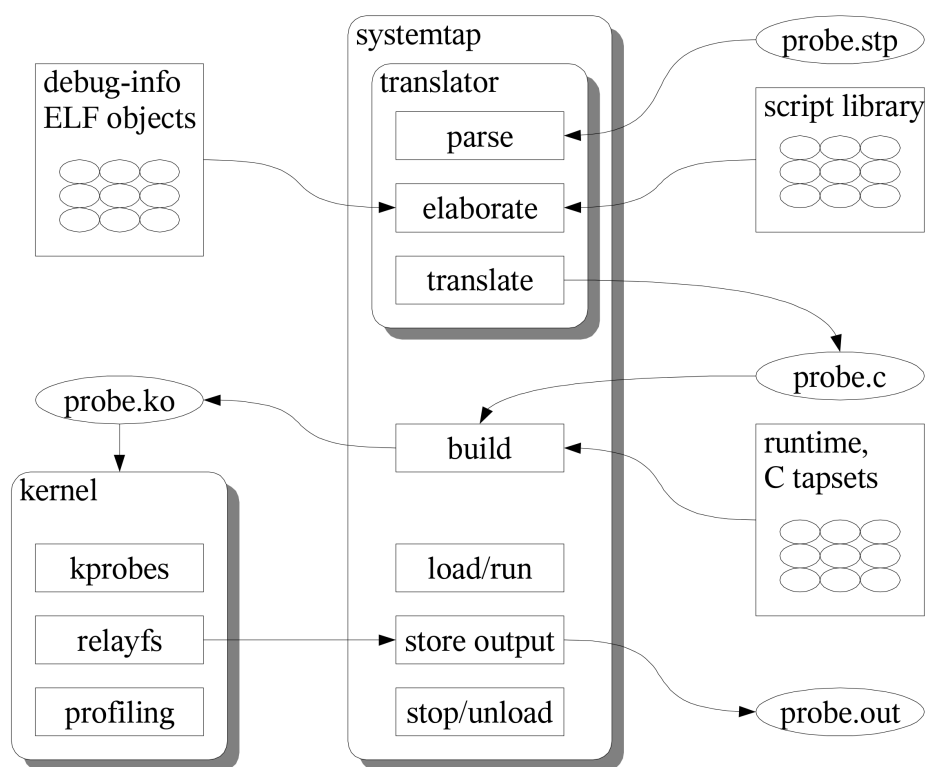


Figure 2.2 Processing steps and components of SystemTap [24]

In this example, the `event_name` is the name of the event as given when the `TRACE_EVENT()` macro has been called. As other trace probe types seen previously, these tracepoints can be conditional using the “if” statement. Unlike dynamic tracepoints though, static tracepoints can only use the variables received as parameters at the instrumentation point. This limitation is due to the fact that static tracepoint handlers do not receive any copy of the registers when the event is encountered, thus making it impossible to get the values of other variables.

As we have seen, given the format of the instrumentation points, the data analysis is provided in the instrumentation itself and the results can be displayed in the console at regular intervals, or written directly to a file. The analysis is therefore performed on the fly, and to the extent of our knowledge, there is no facility to efficiently serialize the raw events for later analysis. The absence of an efficient compact format would become a limitation in the case of large traces, and thus is a clear limitation of **SystemTap**. However, using its built-in data structures, it offers the possibility to calculate custom statistics during event capture.

Ktap

Ktap [26, 27, 28, 29] is a very lightweight tracing tool for the kernel using different design principles from Linux mainstream dynamic tracing.

Ktap uses an approach similar to **SystemTap** and **DTrace** as it uses a script-based instrumentation. The script-writing uses a format called **Ktap** format. That format uses a C-like language based on Lua, lacking arrays and pointers, and provides a simple function definition mechanism. The foreign function interface (FFI) of **Ktap** also allows to define and call C kernel functions from a **Ktap** script. The following gives an example of a C-function calling script:

```
ffi.cdef[[
    int printk(char *fmt, ...);
]]
ffi.C.printk("Called from ktap FFI\n")
```

Ktap uses a bytecode-based Virtual Machine (VM) embedded into the kernel and compiles its tracing scripts in bytecode for that VM, which will then act as an interpreter. The VM then sends the data received to the tracing ringbuffer which is based on the `ftrace` ringbuffer implementation. The VM approach, quite different from the compiled module approach used by **SystemTap**, simplifies **Ktap** usage in embedded systems. These systems often do not have a full compiler toolchain installed and, even if there is one, compiling and linking a module can be very slow and resource hungry, thus making it undesirable for embedded systems.

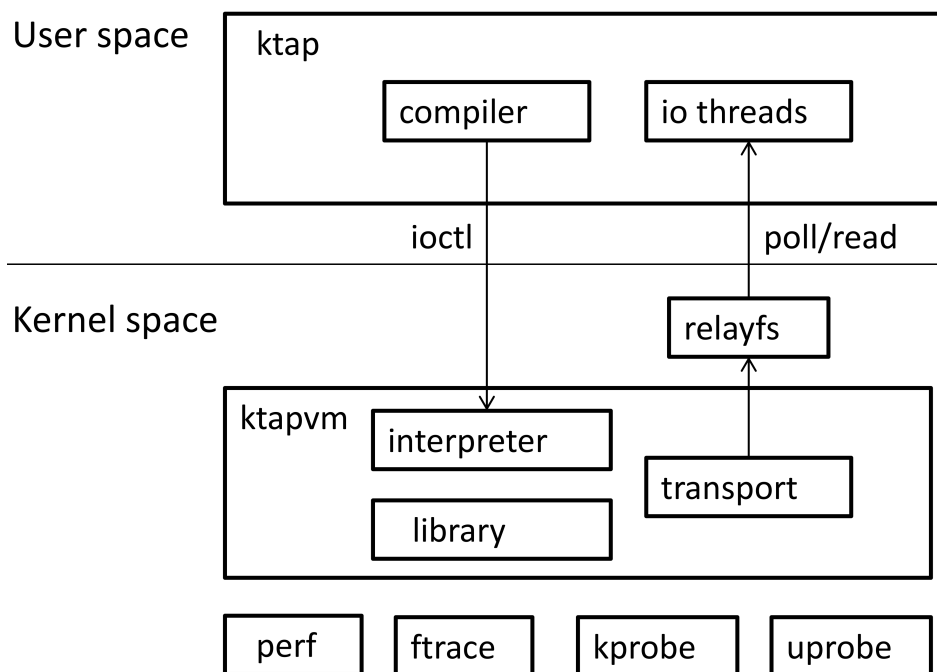


Figure 2.3 Ktap architecture [27]

Figure 2.3 gives an overview of the architecture of `Ktap`. In userspace, the `ktap` process uses its compiler to convert the scripts to bytecode, then feeding the `ktapvm` process in kernel space. The VM then loads the bytecode and executes it. Following the instructions in the scripts, the library's built-in functions are used by the VM to interact with the ringbuffer. The `ktapvm` source code is around 5 000 lines of code and the `ktap` userspace compiler around 4 000. With 40 bytecodes defined, the `Ktap` VM is small and efficiently implemented.

`Ktap` does not contribute any specific new tracing building block but instead leverages current tracing building blocks in the Linux kernel such as tracepoints, kprobes, uprobes, function tracing provided by `ftrace` and the Performance Monitoring Unit (PMU). `Ktap` aims to combine all those tracing approaches through a unified script tracing interface. `Ktap` furthermore allows to write scripts for these different building blocks. The following shows a way to trace the `malloc` function entry in the `libc` library:

```
trace probe:/lib/libc.so.6:malloc {
    print("entry:", execname, argstr)
}
```

`Ktap` is a young project and will probably mature. However, using the tracing blocks available in the kernel, it unfortunately doesn't provide a good way to trace userspace events without

overhead. The section 2.1.2 will detail why the **uprobes** approach for userspace tracing is restrictive for systems needing low-overhead.

LTTng

Linux Trace Toolkit next generation (LTTng) is a tracer that was created with special emphasis on low system disturbance [30, 31]. Until version 2.0, LTTng was a set of patches to be applied to the Linux kernel source to add instrumentation that did not exist in the standard kernel. Since version 2.0, the core portion of the tracer exists in the form of modules loaded during the tracing tools initialization. The modules work with different static instrumentation points in the kernel using the `TRACE_EVENT()` macro. The modules also let you enable tracepoints using **Kprobes**, trace functions and use performance counters values.

When tracing, events are consumed by an external process, the consumer, that writes traces to disk or forwards them using the network. Figure 2.4 shows the general architecture of LTTng where we can see that the consumer daemon is acting as a central point in the trace data flow. Circular buffers are allocated per processor to achieve good scalability without requiring any waiting. In addition, the control variables for the circular buffers are updated by atomic operations instead of locks. Important variables are moreover protected using RCU data structures. Figure 2.5 shows how RCU manages multiple readers trying to access a resource.

Unlike **Feather-Trace**, it is possible for LTTng to support arbitrary types of events by using the *Common Trace Format* (CTF) [34]. The **babeltrace** tool allows to view and convert traces stored in this format to human-readable text output.

A trace read by **babeltrace** will display as follows:

```
[02:47:05.128959473] (+0.000003560) station11-64 exit_syscall:
    { cpu_id = 0 }, { ret = 0 }
[02:47:05.128962750] (+0.000003277) station11-64 sys_readv:
    { cpu_id = 0 }, { fd = 12, vec = 0x7FFF57CB7AF0, vlen = 1 }
[02:47:05.128965968] (+0.000003218) station11-64 exit_syscall:
    { cpu_id = 0 }, { ret = 141 }
[02:47:05.128976987] (+0.000011019) station11-64 kmem_cache_alloc:
    { cpu_id = 2 }, { call_site = 0xFFFFFFFF811098A6,
    ptr = 0xFFFFF8801A2BE0B80, bytes_req = 208, bytes_alloc = 256,
    gfp_flags = 32976 }
[02:47:05.129004111] (+0.000027124) station11-64 sys_poll:
```

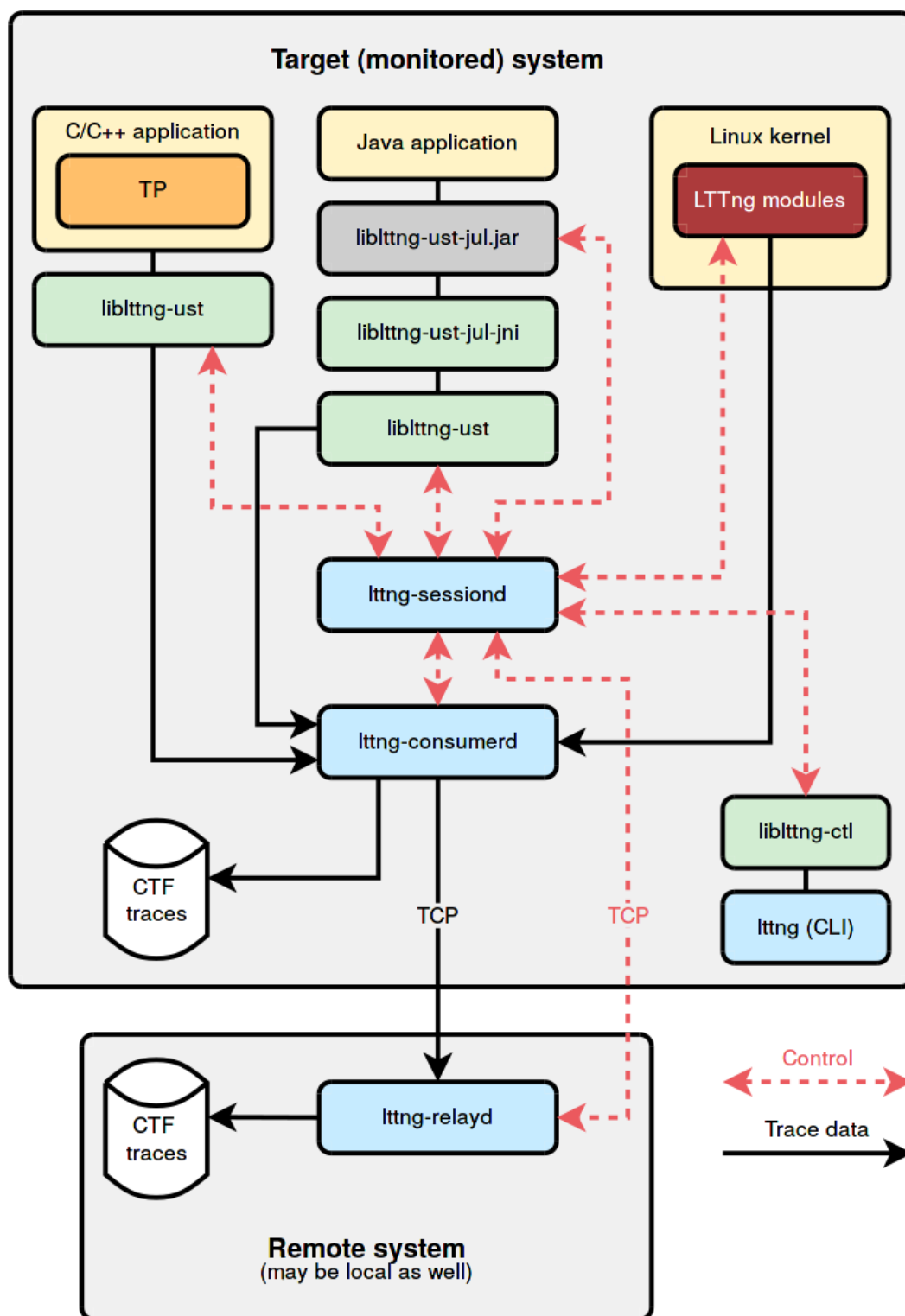


Figure 2.4 General architecture of LTTng [32]

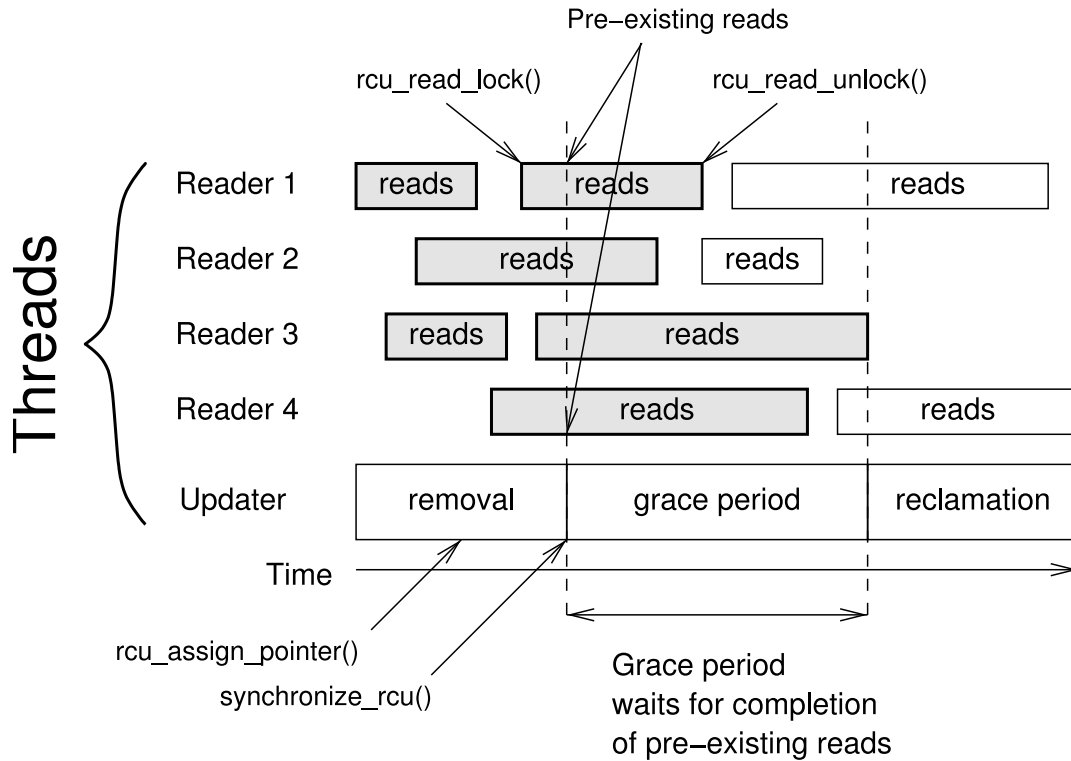


Figure 2.5 RCU flow to handle resources [33]

```

{ cpu_id = 0 }, { ufds = 0x7FC1D70B7E70, nfd = 3,
  timeout_msecs = 227 }
[02:47:05.129007224] (+0.000003113) station11-64 hrtimer_init:
  { cpu_id = 0 }, { hrtimer = 18446612139423701672, clockid = 1,
    mode = 0 }
[02:47:05.129008246] (+0.000001022) station11-64 hrtimer_start:
  { cpu_id = 0 }, { hrtimer = 18446612139423701672,
    function = 18446744071579214861, expires = 849575927150,
    softexpires = 849575700151 }
[02:47:05.129009489] (+0.000001243) station11-64 rcu_utilization:
  { cpu_id = 0 }, { s = "Start context switch" }
[02:47:05.129010026] (+0.000000537) station11-64 rcu_utilization:
  { cpu_id = 0 }, { s = "End context switch" }

```

In this output format, we can see the date of the event, the time difference from the previous event, the name of the machine on which the trace was recorded, the type of event (“`exit_syscall`” for the first line), the processor on which the event was executed and finally all the other variables associated with the type of the traced event. We can also see many

other information fields, such as the number of branches or cache faults, as long as we asked to record them while tracing.

2.1.2 Userspace tracers

Userspace tracing is based on the same principles as operating system kernel tracing. In other words, the tracing is based on taking events in an application without interrupting its execution, while limiting the impact on the latter. Many implementations of userspace tracing tools are based on blocking system calls and string formatting (with `printf` or `fprintf` for instance), or achieve thread consistency by blocking the shared resources against concurrent writing. The logging system `Poco::Logger` is a sample tool implemented in this way [35]. This class of tracers is slow and inadequate for use in the context of multiprocessors or real-time applications and systems.

Two other types of userspace tracers can be identified: those requiring the cooperation of the kernel and those operating entirely in userspace. For this study, we are primarily interested in tracers able to correlate kernel and userspace traces on Linux. Although **Feather-Trace** satisfies these criteria, the limitations we have noted in the previous subsection are too important to use it for our research.

SystemTap with uprobes

SystemTap can be used to trace userspace applications in addition to the kernel. Before the Linux 3.8 kernel, it was necessary to add a patch to the kernel, called `utrace` [36], to allow userspace tracing. This patch is a set of functions that are not integrated into the kernel code, but are intended to replace transparently to the user all the features offered by the `ptrace` subsystem, while adding features to trace the userspace. However, since the Linux 3.8 kernel, a new component appeared. This component, called `uprobes` [37, 38], works in the same way as `Kprobes` but for userspace. It is now only necessary to enable this option in the kernel configuration to be able to trace userspace applications with **SystemTap**, as with other tracers identified in the previous section that use `uprobes`.

However, tracing the userspace with `uprobes` imposes a major constraint: like for **Feather-Trace**, a system call is needed for each tracepoint encountered. Such behavior implies a major impact on the performance of the traced application, especially when dealing with a real-time application. Indeed, entering the kernel of the operating system through a system call means returning control to it. In that case, if internal kernel tasks are waiting, the kernel will try to execute them before returning control to the application. This problem applies to both

dynamic and static tracepoints with tracers using **uprobes**. Indeed, the static tracepoints use the same method for tracing the application as the dynamic tracepoints. The only difference in using a static tracepoint is that it makes a system call instead of generating an interrupt, which may be slightly more efficient, and it is simpler to implement than dynamic instrumentation.

LTTng-UST

UserSpace Tracer (UST) is the user space component of the LTTng tracer [39]. It provides macros for adding statically compiled tracepoints to a program. It works in the same way as the kernel tracer, including the protection of important trace configuration variables carried out here with a version of RCU specially adapted to userspace, called URCU. URCU avoids the exchange of cache lines between viewers that can occur with traditional read and write locking systems [33].

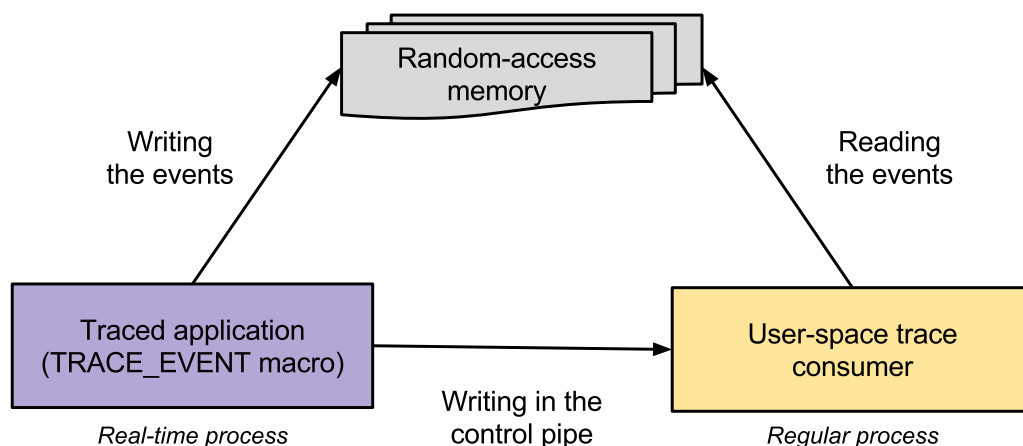


Figure 2.6 Simplified diagram of the trace recording of LTTng-UST

LTTng-UST avoids going back to the kernel since the traced application writes directly in shared memory, which is then read by the traces consumer, a separate process from the real-time application. Figure 2.4 shows the place of UST in LTTng. To simplify the userspace data acquisition, Figure 2.6 shows a simplified diagram of generating a trace record using this tool. We identify the two processes involved in tracing: the application and the consumer. The traces consumer used here is specific to userspace, it is therefore a different process from the kernel traces consumer. The application writes directly events in memory and therefore has no need to go through the operating system kernel. Once the buffer in which it writes is full, it uses a non-blocking control tube to wake the consumer, and then switches to the

next available buffer to continue recording the events. The consumer will then empty the buffer located in shared memory, with little or no impact on the application. The traced application and the consumer are two distinct processes that require little communication with each other. The application may therefore be a process with real time priority, while the consumer remains a regular process. Moreover, as time values for the kernel and userspace are using the same time source, traced events for these two separate system levels can be correlated to the nanosecond.

Some benchmarks [40, 41] compared **SystemTap** using `utrace` and **LTTng-UST** and showed that **UST** was 289 times faster than **SystemTap** in some instances for userspace tracing (in flight recorder mode). This huge difference was identified as being due to the buffering of **UST** while **SystemTap** uses system calls for each event.

Finally, our previous work on **LTTng-UST** [42, 43] enables tracing userspace applications with a very low overhead, allowing to trace real-time applications with latencies under 10 microseconds. These benchmarks also confirm the latency efficiency of **LTTng** over **SystemTap** and **uprobes** in general.

2.2 Common analysis methods and tools for real-time applications

Obtaining data is only the first step. It is not useful in itself unless it can be analyzed properly. Different analysis exist to partially or completely automate systems and applications checking, and design verification. This includes trace analysis, which is our focus.

In this section, we will concentrate on the analysis tools that use traces to simplify the analysis of real-time applications for non-advanced users.

2.2.1 Model checking oriented analysis tools

Tango

Tango [44, 45] is an automatic generator of backtracking trace analysis tools for specifications written in the **Estelle** formal description language. It can be used for single-process specifications only. **Tango** is based on a modified **Estelle-to-C++** compiler, and generates tools that are specific to the given model. They check the validity of any execution trace against these specifications using a number of checking options.

Figure 2.7 gives an overview of the **Tango** system. In this diagram, we can see the path along which the original **Estelle** specification is converted by **Tango** to **C++** source code, before being compiled by **g++**, using the appropriate libraries, to an executable trace analyzer (*TA*

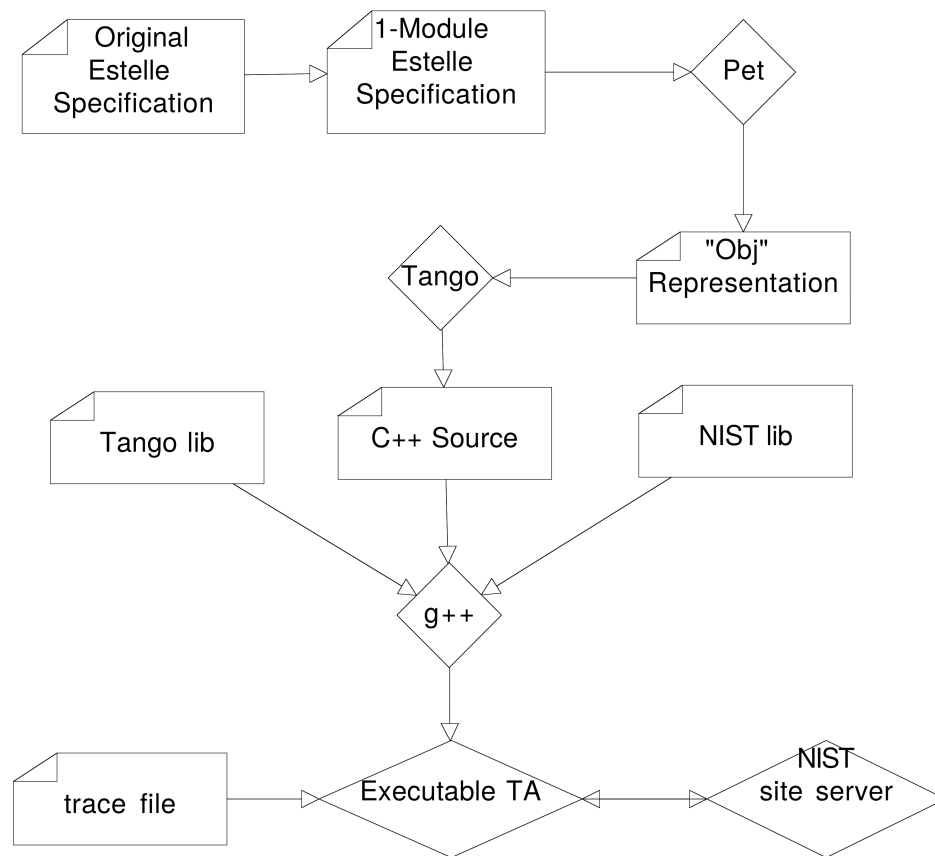


Figure 2.7 The Tango system [44]

in the diagram). This executable trace analyzer will then check that the specifications are satisfied in the given trace file.

However, the executable trace analyzer is not a standalone executable and needs to be run under the NIST X Windows Dingo Site Server. **Tango** authors created a shell script to run a local server, but the user still needs to run the executable through a Unix shell. The results of the trace analysis will be stored in a log file containing (debug mode) or not all the transitions attempted by the trace analyzer module, then the confirmation (“*all outputs verified*”) or invalidation (“*trace is invalid*”) of the trace according to the analyzer specifications.

From the start (generation of the **C++** source from the **Estelle** specifications, using an intermediate object-oriented static representation state) to the end (view of the results of the trace check), **Tango** needs a lot of actions from the user.

Tango’s approach to trace analysis is interesting but limited. Indeed, **Tango** aims to validate specifications of a protocol, and thus only needs to know what is happening in the application, and not in the system. It is therefore not possible to specify constraints based on the system’s state, such as the CPU load and memory use.

Logic of Constraints checker

In the same idea as **Tango**, [46, 47] present algorithms to automatically generate trace checkers. In their approach, they also use formulas written in a formal quantitative constraint language, and use them in correlation with traces to analyze the traced simulation for functional and performance constraint violations. The language used to specify the properties to be verified is the Logic of Constraints [48] formalism, which follows a logic that is suitable for specifying constraints at the abstract system level, where the execution coordination is of the highest importance.

Figure 2.8 shows the methodology for verification used by the proposed system. The elements needed at the beginning are the Logic of Constraints formula and the simulation trace format. The definition file used for the Logic of Constraints formulas to validate, and the trace format, look like the following example taken from [46]:

```
[LOC: rate]
  formula: t(Display[i + 1] - t(Display[i])) == 10
  annotation: event value t
  trace: "%s : %d at time %f"
[LOC: latency]
  formula: t(Display[i]) - t(Stimuli[i]) <= 25
```

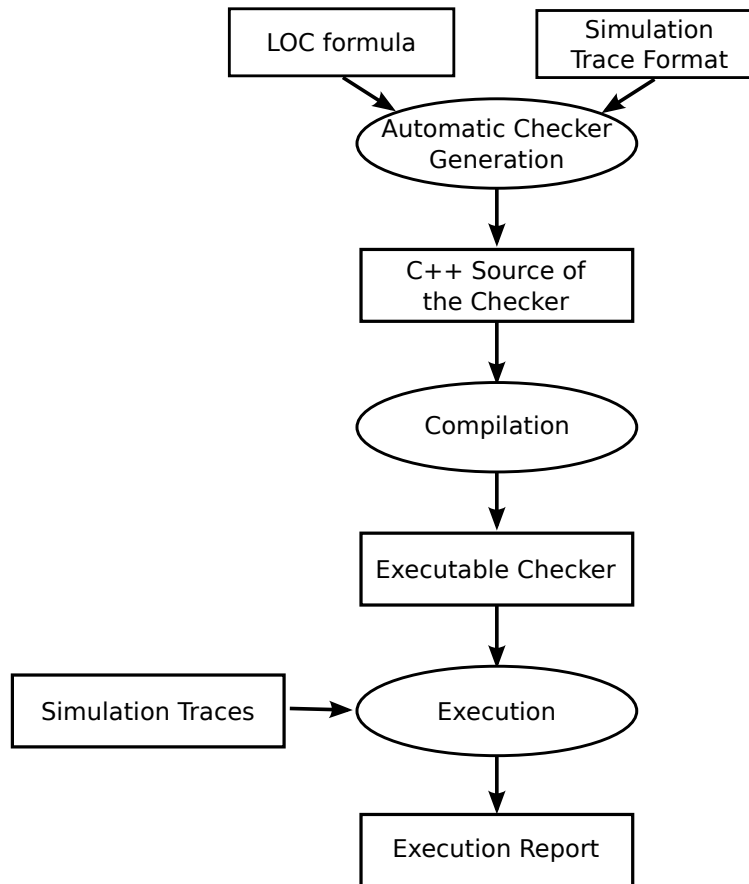


Figure 2.8 Trace analysis methodology used in [46]

```

annotation: event value t
trace: "%s : %d at time %f"

```

In the definition file format, each Logic of Constraints formula is preceded by a label and followed by the format needed to extract the event names and their annotations from the traces. In the given example, the labels of the two constraint formulas are **rate** and **latency**. In both cases, the trace format expected is the same: a string which ends in a “:”, followed by a space and an integer, the string pattern “at time” surrounded by two spaces and finally a float. The annotation definition allows to match these values to variables in the formula. In our case, the string is stored in the **event** variable, representing the event name, which can be discarded if it is not used in the formula. The integer is used as the **value** of the event instance, not used in our case, and the float is used as the **t** variable which is needed in both formulas.

The automatic checker generator parses the definition file to generate the C++ source of the checker, that will then be compiled to generate an executable checker. This executable checker will then take in simulation traces and analyze them to produce an evaluation report specifying any constraint violation, including the value of index **i** which violates the constraint, as shown in the following example of error report taken from [46]:

```

username@host$ checker latency.trace
Reading from trace file "latency.trace" ...
Formula t(Display[i]) - t(Stimuli[i]) <= 25 is violated
    at trace line# 278: Display: -6 at time 87
where i = 23
t(Display[i]) = 87
t(Stimuli[i]) = 60

```

The trace analysis methodology and tools proposed here only need the user intervention for starting the executable checker after its compilation. The results analysis is also user-friendlier than the approach followed by **Tango**, as we can see in the previous example: the status (in this case the violation) is clear, and enough trace information are given to the user to understand where there was a violation. However, this trace checker only provides information about the satisfaction or not of a given constraint, and does not push the analysis to tell the user the reason of a given infringement.

2.2.2 Data extraction tools

KOJAK

The KOJAK (Kit for Objective Judgment & Automatic Knowledge-based detection of bottlenecks) project, started in 1998, introduced a new performance-analysis environment designed to identify numerous performance problems on typical parallel computers with Symmetric MultiProcessor (SMP) nodes [49, 50, 51]. KOJAK has been developed to analyze parallelism-related performance problems that come from inefficient usage of the parallel programming interfaces MPI and OpenMP, or hybrid applications. KOJAK uses execution traces to provide automatic or manual analysis. To identify the performance problems, KOJAK uses a number of execution patterns that are then recognized in the event traces.

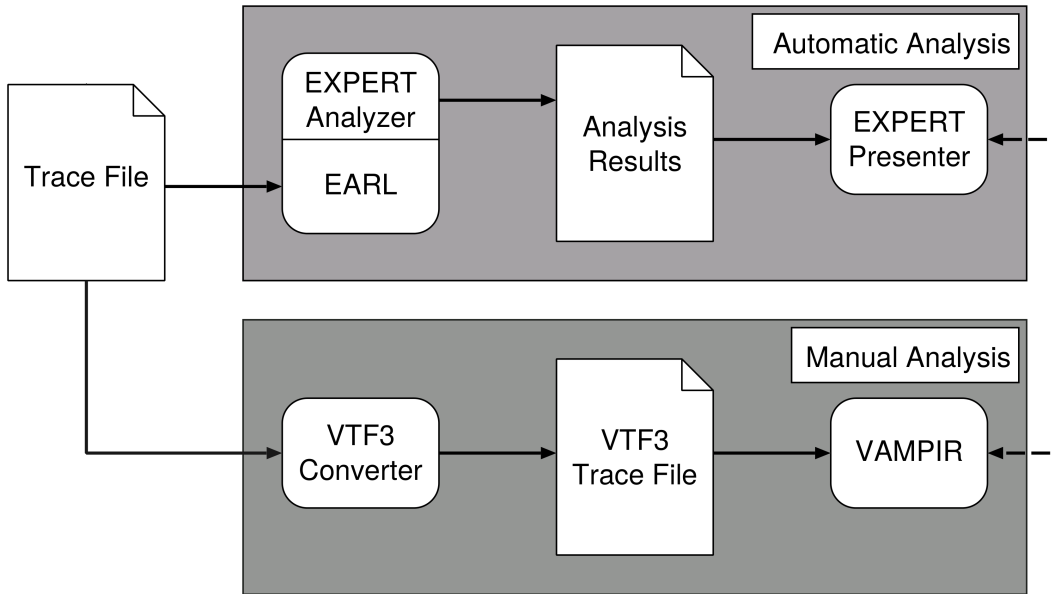


Figure 2.9 Overall architecture of the tracing work flow of KOJAK [50]

Figure 2.9 gives an overview of the analysis part of the KOJAK performance-analysis environment. Two analysis approaches are possible, either automatic or manual. The traces analyzed have been generated by an automatic instrumentation of parallel programs, provided in the environment to instrument the user functions at compilation time (using a compiler-supplied profiling interface) or later at binary-level. The traces fed to the automatic analyzer are not used as raw traces but accessed through an abstraction level providing random access to single events. The events are identified using their relative position and are delivered as a list of key-value pairs representing event attributes such as their time and location. These events are used to match the execution patterns. The detected patterns are classified by type and

quantified in severity. The results are presented in a single view with three dimensions: the class of performance behavior, the call path and the thread of execution. Each dimension is represented as a hierarchy to help the user to understand the behavior using different levels of detail.

An approach has been proposed in [52] to use **KOJAK** in correlation with a profiling tool, **PeekPerf**, and a timeline analysis tool used primarily to visualize and analyze a parallel events trace file, **Paraver** [53]. The experiment, with final results given in [54], aimed to prove the capability of **KOJAK** to interface with multiple analysis tools, providing data and automatically searching patterns in the resulting traces to identify the sources of wait states in parallel applications.

Scalasca

The **Scalasca** project is the successor of **KOJAK** and was started in 2006 [55, 56, 57]. Faster than **KOJAK** and scalable, **Scalasca** aims to simplify the identification of bottlenecks using execution traces. Even if it is mainly targeting large-scale systems, **Scalasca** is also suited for small and medium-scale high-performance computing platforms.

Scalasca offers analysis using both aggregated statistical runtime summaries and event traces. The summary report provides an overview of the performance behavior of an application by showing which process, in which call-path, consumes time and how much. By aggregating the data for the whole execution, the runtime summary is mostly independent of the execution duration. **Scalasca** uses that approach to measure the wall-clock time, the number of visits to a call-path, the message counts, the transferred bytes and other hardware counters. On the other side, the event traces target a deep study of the concurrent behavior of the program. The traces give more information about interprocess communication, particularly those happening during communication or synchronization operations. They also allow to analyze how parallel activities influence the performance of each other. The used events are mainly the runtime events that are critical for communication or computation, such as entering and leaving functions or sending and receiving point-to-point messages as well as the participation in collective communication.

Figure 2.10 gives an overview of the performance analysis workflow of **Scalasca**. The application can be instrumented to generate the summary report, with aggregate performance metrics, or to generate event traces. The first option provides an overview of the performance behavior and use the report to optimize the measurement configuration for later trace generation. In tracing mode, each process will generate a tracing file containing records for all its local (in process context) events. **Scalasca** analyzes the traces at the end of the execution

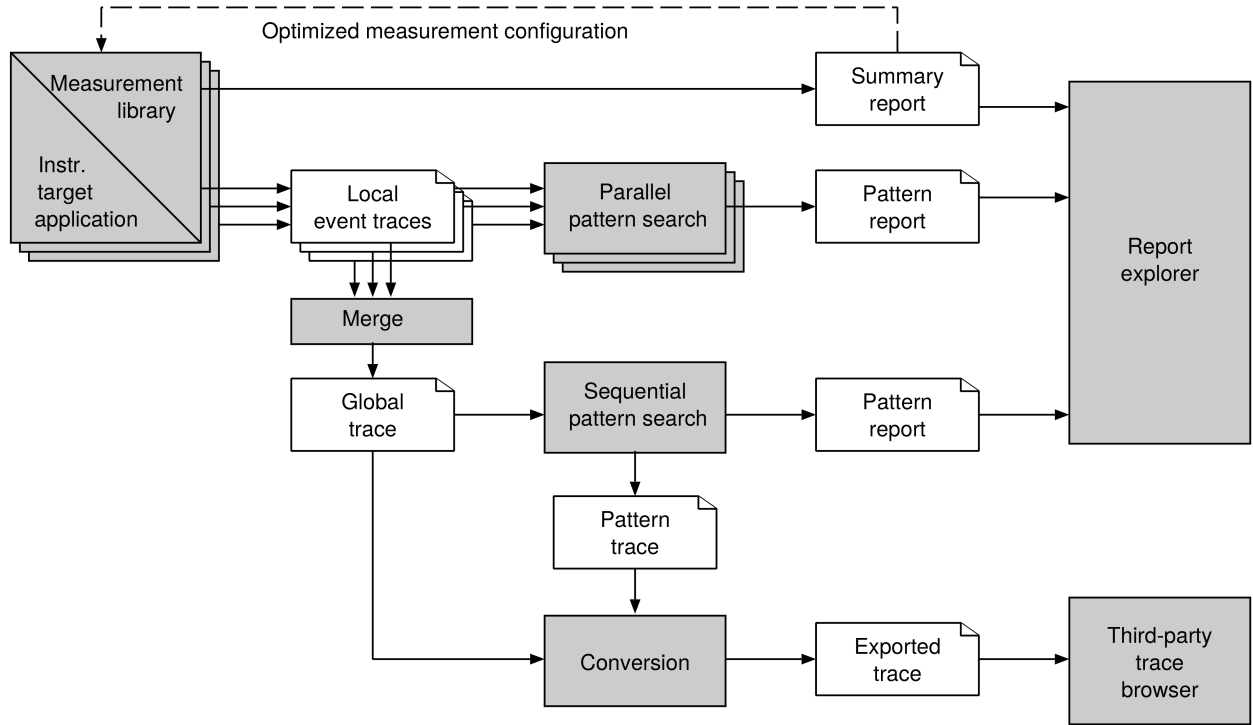


Figure 2.10 Performance analysis workflow of *Scalasca* [57]

of the program and searches for specific patterns that can indicate wait states and related performance properties. Like *KOJAK*, it then classifies findings by category of the detected instances and quantify their significance. *Scalasca* then produces a pattern-analysis report containing performance metrics for every function call-path and system resource.

The produced report can be explored using a graphical report explorer as shown in Figure 2.11. The “Metric tree” pane allows the user to select a metric problem to analyze. The “Call tree” pane then shows “where” in the program the problem appeared. The “System tree” pane finally shows “which process” is concerned.

Scalasca has the ability to identify wait states even for a very large processor count, and supports the parallel programming interfaces *MPI*, *OpenMP* and hybrid applications that are widely used in scalable high-performance computing platforms.

Scalasca was also extended to provide a way to analyze metacomputing applications [59], thus allowing to extend the trace analysis to multiple systems interacting with each other.

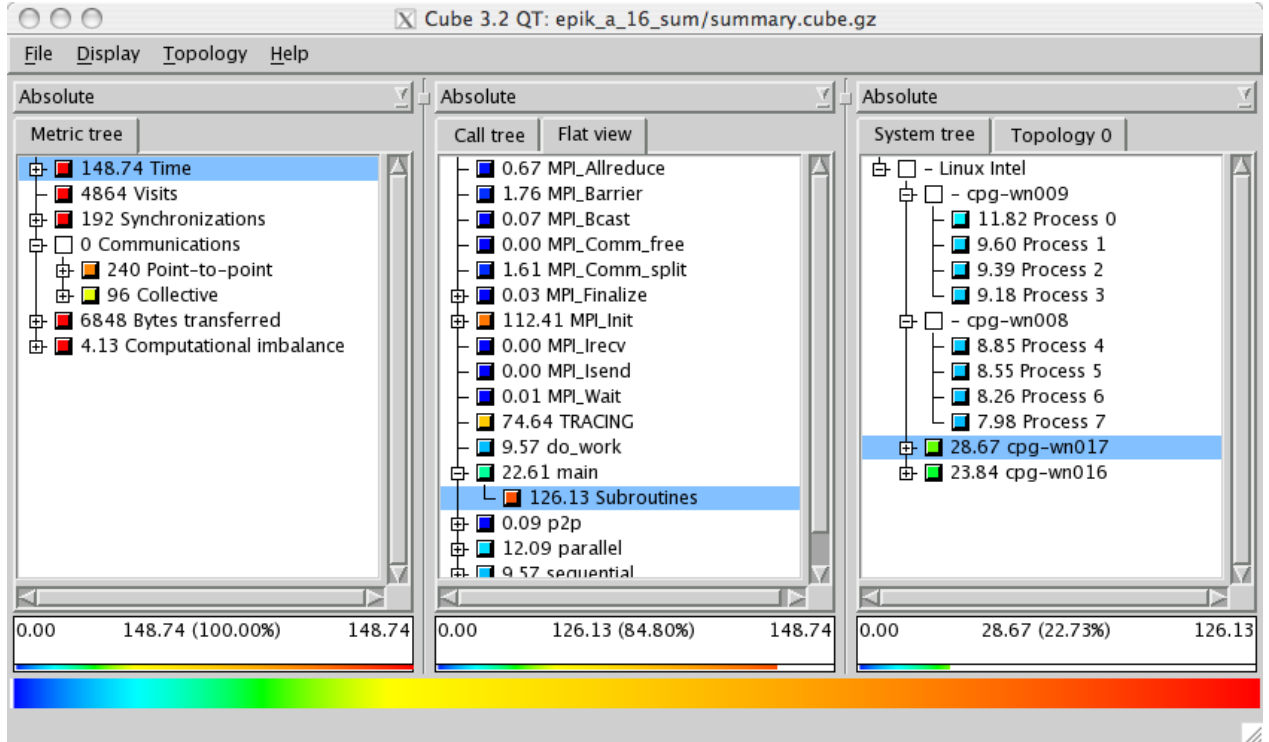


Figure 2.11 CUBE3 window used to analyze Scalasca results [58]

SETAF

The System Execution Trace Adaptation Framework (SETAF) is presented in [60]. SETAF is a framework adapting the system execution traces and dataflow models to have the required properties for analysis and validation of the Quality of Service (QoS).

SETAF bases its work on a tool: Understanding Non-functional Intensions via Testing and Experimentation (UNITE). UNITE describes a method to use system execution traces in order to validate the distributed system QoS properties [61]. UNITE is able to perform such analysis in a runtime complexity of $O(n)$, independently of the target system execution trace. UNITE needs first to extract the interesting information from log messages using a log format. The following shows an example of log messages that can be used by UNITE:

```
Config: sent event 8 at 120345678
Planner: sent event 9 at 120456789
Planner: received event 8 at 120567893
Effector: received event 9 at 120678934
```

We would then need the following log formats taken from [60] to extract data from the log messages:

```

LF1: {STRING cmpid} sent event {INT eventid} at {INT sent}
LF2: {STRING cmpid} received event {INT eventid} at {INT recv}
Relation:
    LF1.cmpid = LF2.cmpid
    LF1.eventid = LF2.eventid

```

Each log format (LF1 and LF2 in this example) contains static (“**sent event**”, “**at**” and “**received event**”) and variable parts. The variable parts are used for extracting the metrics from its corresponding log message in the trace. For instance, in the LF1 log format, there are three variables: **cmpid**, a string representing the origin of the log message, **eventid**, an integer representing the event identifier, and **sent**, an integer representing the sending time of the event. The last lines of the example of log formats define a dataflow model capturing relationships between variables across log messages. Users then can define expressions to validate a given QoS property using the variables defined in the log formats.

SETAF has been created to overcome the shortcomings of UNITE such as the correlation of log formats that have non-unique instances or hidden relations. SETAF’s approach is to adapt the dataflow model with user-defined external adapters. This approach thus allows to write adaptation specifications according to the system domain, without modifying the existing source code of the distributed system, but needs specification writers to know about the dataflow model’s limitations.

Before using SETAF, the users must first analyze manually the system traces. This analysis allows to identify the adaptation pattern, identifying what properties need to be added to the dataflow model in order for UNITE to correctly analyze the execution traces. Each of the adaptation patterns contains the variables, the data points, the relations and the adaptation code. The data points are used by UNITE to create a valid execution flow and to create new relations. The variables are private data points only used in the adaptation pattern. The relations allow to insert new causality relations between log formats in the dataflow model. The adaptation code is the logic behind the adaptation pattern, giving information on how to update the dataflow model.

Figure 2.12 gives a conceptual overview of the workflow of SETAF. SETAF converts specifications into C++ source code which is then compiled into an external UNITE module. This module is then loaded by UNITE which uses it to adapt its corresponding traces to allow QoS analysis. SETAF therefore acts as an overlay used by UNITE to transform the traces and provides the missing information to improve the analysis.

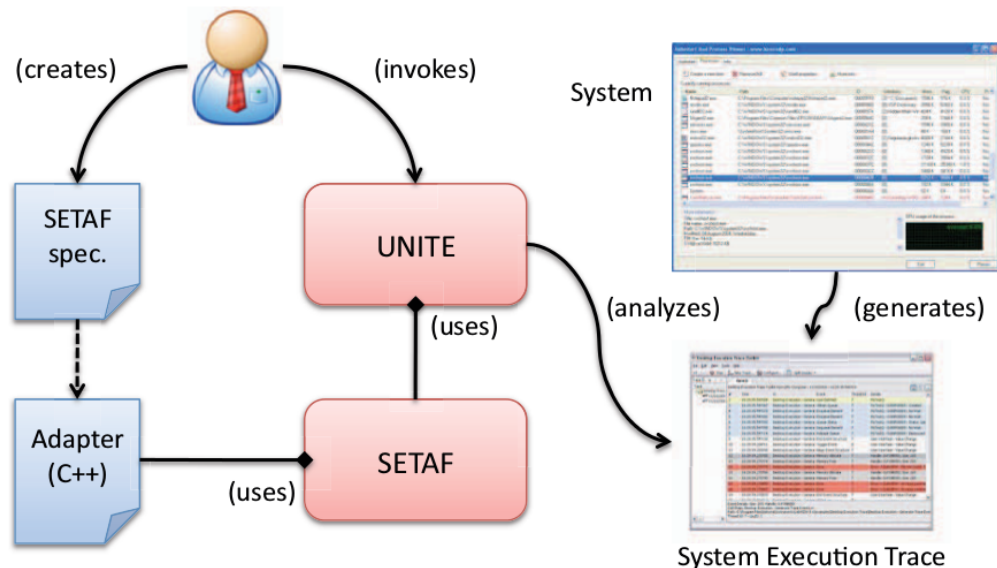


Figure 2.12 Conceptual overview of SETAF's workflow [60]

Trace Compass

Trace Compass [62] comes from the LTTng component of the Eclipse Linux Tools project. The aim of the LTTng component was to provide a graphical interface in Eclipse for the LTTng tracing tools.

Trace Compass, previously named Tracing and Monitoring Framework (TMF), was developed as a generic framework for reading, parsing and analyzing traces, independently of their type, using LTTng as the reference implementation. Over time, support for other trace formats and types was added. In the current version of Trace Compass, LTTng traces are only one of many trace types supported. Other types include, but are not limited to, all the traces using the Common Trace Format (CTF), GDB traces for debugging, traces using the Best Trace Format (BTF), and traces using the libpcap format.

Figure 2.13 gives an overview of the perspective of Trace Compass to visualize and analyze traces. The “Project Explorer” on the left lets the user choose between the traces to visualize, while the different windows on the right are different “views” for the trace, allowing to zoom on specific information from the analyzed trace. For instance, the “Control Flow” view at the top of the Figure shows on the left a hierarchy of the processes that spawned events during the trace. That view shows on the right the different status, represented by different colors, of those processes along a timeline for the whole trace duration. The “Resources” view, second view from the top of the Figure, gives an overview of the status of the different resources of the system like the CPUs and the IRQs they received. The different status are

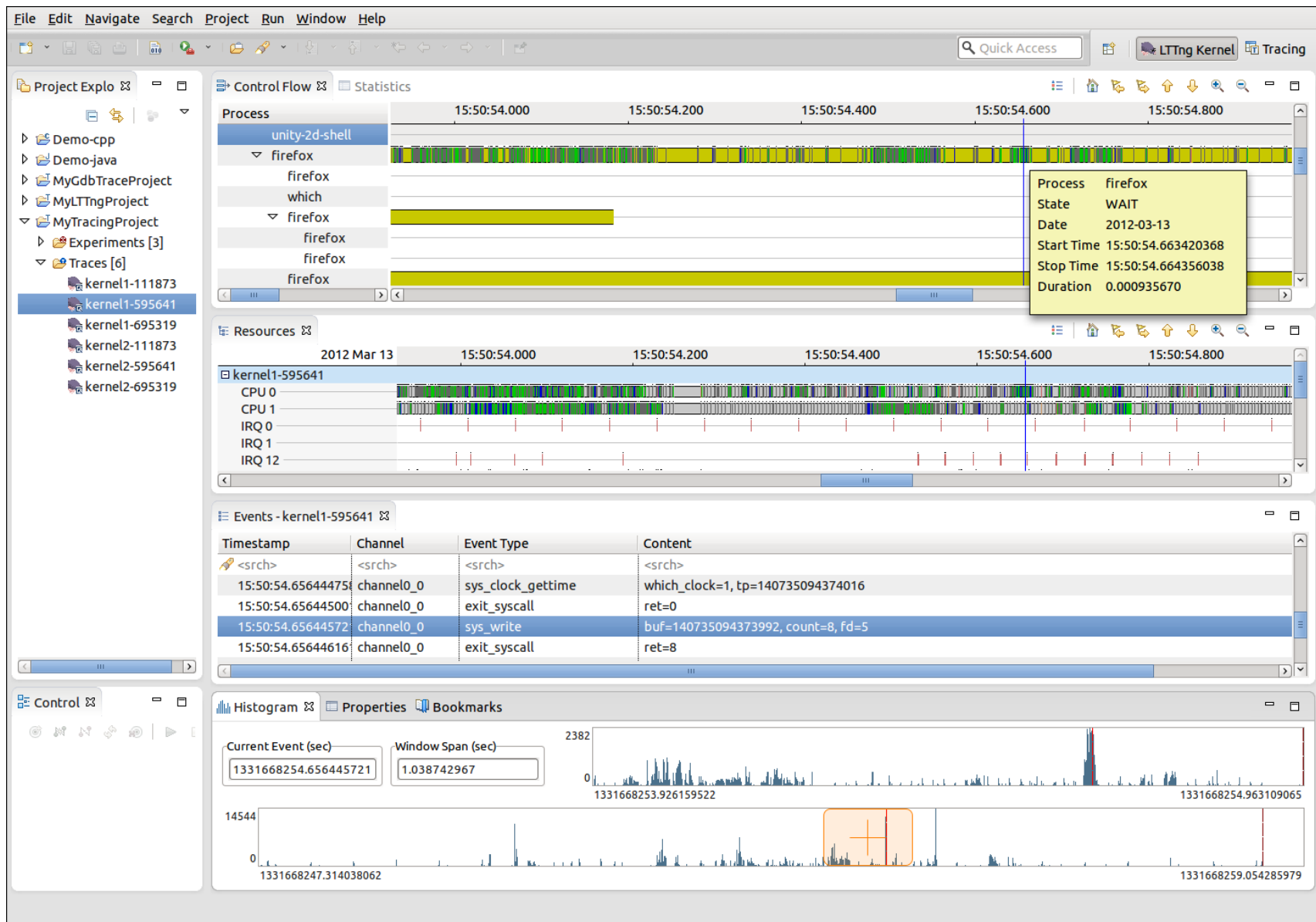


Figure 2.13 Perspective of Trace Compass [62]

represented here again by different colors along the timeline. The third view from the top of the Figure, the “Events” view, shows a list of events in the trace in four columns representing the time at which the event occurred, the channel in which it was registered (depending on the CPU on which it was running), the event type and the content of the event. Filters can be used against this list to find specific events. The last view shown in the Figure is the “Histogram” view. That view shows a histogram of the number of the events, which the user can browse to reach a specific place with more (using the text inputs to give a precise time) or less (using the mouse to browse directly from the histogram) precision.

As part of previous work, **Trace Compass** also offers a view designed to provide a first level of automatic pattern matching [63]. This view first splits the tasks according to the individual jobs they contain, using the state system of **Trace Compass**. Such a view can then be used to identify performance issues related to one precise job in a task, for instance a peak of latency occurring when doing a particular operation.

Another view, called the “Critical Flow” view, is of particular interest for our research. This view, based on the analysis of the system-level critical path of applications, presented in [64], aims to provide an automatic analysis tool to recover segments of execution affecting the waiting time of a given computation. The critical path analysis is only using kernel traces to provide an application independent analysis.

2.3 Constraints representation and modeling formats

To model an application and its constraints, several modeling languages and tools exists. Most of those languages and tools, though, don’t have any way to represent real-time constraints, let alone system-specific constraints such as the CPU usage and number of system calls for instance. Moreover, as we aim at providing tools for simplifying the user intervention, we search user-friendly approaches to constraints representation.

2.3.1 UML

Unified Modeling Language (UML) [65] is a modeling language that aims to provide tools for the analysis, design and implementation of software-based systems. UML combines multiple modeling concepts which are organized in what we call language units. Each language unit consists in a set of modeling concepts, tightly-coupled, that allows users to represent aspects of the system under study following a particular formalism or paradigm. UML is an Object Management Group (OMG) standard and, since 2000, an approved ISO standard [66]. Considering the general use of UML in computer systems modeling, and since this language

doesn't provide a representation for real-time constraints, we will present in this section the different real-time constraints representation approaches that are used over UML.

UML-MAST

The UML Modeling and Analysis Suite for Real-Time Applications, or MAST, is a model for representing the temporal and logical elements of real-time applications [67, 68]. This model allows a rich description of the system, and comes with its own suite of tools to analyze any system represented using it. The MAST representation includes fixed priority systems and is designed as an extensible open model to accommodate other kinds of systems.

MAST uses the class diagram representation of UML to define time counters and time constraints. Although MAST is an interesting approach, the use of class diagrams doesn't seem appropriate to represent a system with states and transitions.

UML-SPTP and UML-MARTE

The UML Profile for Schedulability, Performance and Time Specification (SPTP) [69] provides the concepts needed to annotate the UML models with time and performance characteristics. SPTP was created following the observation that while UML was used in a large number of time-critical and resource-critical systems, it was lacking in some key areas that are of particular interest and concern to real-time system designers and developers. For instance, the lack of quantifiable notion of time and resources was a great obstacle to its broader use in the real-time and embedded domain. To address such problems, SPTP was created and adopted as an OMG standard in 2003. In 2009, SPTP was replaced by the UML Profile for MARTE, (Modeling And Analysis Of Real-Time Embedded Systems [70]).

The ideas behind SPTP and MARTE are the same in that we add annotations to diagrams to specify the time and resource constraints. These evolutions of UML mostly use a sequence diagram to specify the time constraints, which represents more naturally the application's life cycle than a class diagram, as used by MAST.

UPPAAL time constraints

UPPAAL is a toolbox for the modeling, simulation and verification of real-time systems. UPPAAL is appropriate for systems which can be represented as a set of non-deterministic processes (with a finite control structure and real-valued clocks), and that communicate through shared variables or channels [71, 72].

For modeling purposes, UPPAAL uses state machine diagrams like those of UML, but with improved constraints and conditions. Those constraints and conditions can be linked to other state machine diagrams using the shared variables or channels provided in the description-language of UPPAAL. That description language serves as a modeling language to describe the system behavior as networks of automata, extended with clock and data variables.

The state machine representation of UPPAAL is interesting as it is very visual, and allows users to easily understand the system workflow.

2.3.2 State Chart XML

The State Chart XML (SCXML) [73] is a specification that provides a generic state-machine representation in eXtensible Markup Language (XML). State Chart eXtensible Markup Language (SCXML) was built as a general-purpose, event-based, state machine language. It combines the concepts of both Call Control eXtensible Markup Language (CCXML) and Harel State Tables.

CCXML is an event-based state machine language. It was designed to support call control features in Voice Applications, including specifically, but not limited to, VoiceXML [74]. The specification for CCXML 1.0 defines a state machine and event handling syntax, as well as a set of standardized call control elements.

Harel State Tables [75] are a state machine notation that offer a clean semantics for sophisticated constructs, such as parallel states. They were included in the UML specification since its version 2.3, but have been defined as a graphical specification language. Hence, they do not have an XML representation.

The goal of SCXML is thus to combine Harel State Tables semantics with an XML syntax that is a logical extension of the state and event notation of CCXML. SCXML therefore provides all the facilities to specify and represent states, variables assignments, transitions and constraints.

2.4 Pattern recognition and model inference

Multiple approaches exist to detect and recognize patterns in traces, logs or communications between processes. In this section, we will review the existing methods, algorithms and tools that provide a way for pattern recognition and model inference in traces.

CSight [76] was developped to help debug and understand concurrent systems. For that matter, it infers concise and accurate models of their behavior by using the logs of their

executions. These models are provided in the form of communicating finite state machines, and can be used to understand complex behaviors or detect anomalies. **CSight** requires the logged events to be augmented by vector timestamps, but the authors provide a tool that automatically adds those vector timestamps to system logs. However, **CSight** still requires the user to specify how to parse and process the log before searching for the patterns.

A pattern language, in the form of a scripting language, is presented in [77]. It was designed to be used to analyze Linux kernel traces, and provide a solution for system administrators and normal users to detect some problems or even just understand the system behavior by abstracting it. Yet, this does not allow to identify problems, as it first requires to define a pattern to be found in the trace. Therefore, this approach only allows to find occurrences of a known problem for which a pattern was already specified.

A data mining approaches is presented in [78] to discover periodic behaviors in multimedia applications. A pattern mining approach is used to discover automatically all the periodic patterns that occur in the execution trace of a multimedia application. To do so, they use the gaps between events of the same type. Once multiple patterns have been found, they check their conflicts by looking at the perturbations in the periodicity of one, and correlating it to the activity of the others. However, this method splits the trace into individual work-sets, which requires to specify windows or specific events as splitting points. Other similar approaches can find patterns without this need [79]. Yet, real-time tasks are not always periodic, and both of those approaches would not work for sporadic tasks.

[80] presents another data mining approach to find frequent episodes in a sequence of events. An episode is defined as a collection of events occurring relatively close to each other in a given partial order. Episodes thus describe temporal relationships between events. To compute the frequency of the episodes, the trace is first split into windows of a fixed size, and the number of windows containing the aforesaid episode is counted. Yet, splitting the trace in windows can hide patterns that cannot be split. Furthermore, this approach is incremental and thus requires the trace to be read several times.

[81] proposes an algorithm to extract communication patterns from MPI communication traces automatically. The detected communication patterns can reflect performance bottlenecks or highlight special program characteristics, and thus do not intend to represent the full program workflow. The algorithm uses suffix trees to finds locally repeating sequences, and iteratively grows them into global patterns. However, this approach aims at detecting flaws in the sequence of communications, and it is therefore not possible to augment these patterns with system and application constraints. It is thus not possible to use this approach to detect problems with causes external to the MPI processes.

To understand inter-process communications, the extraction of communication patterns has also been studied in [82]. Two algorithms are proposed to recognize the repeating patterns in MPI communication traces, and to search if a given communication pattern occurs in a trace. Both algorithms use the n-gram extraction technique, which is a natural language processing approach. They can work on the trace as it is generated, and do not require complex data structures. However, the patterns have to be repeated in the trace in order to be detected. Also, as previously, metrics cannot be set on the resulting detected pattern.

[83] presents a method to build a high-level model of the behavior of a Programmable Logic Controller (PLC) program component, as it is observed during a program execution. It uses a deterministic record and replay technique to build a model augmented with the timing information and the transition and Input/Output (I/O) behavior of the component. This model can then be used to check other executions of the same program, or similar programs, for compliance. Yet, this approach only considers the behavior of the PLC component, and not of the system and other applications.

[84] defines a framework to extract temporal properties from runtime traces of real-time systems. The approach is based on event-recognition finite state machines, representing the different metrics to be extracted from the system, such as the computation time or the response time of tasks. However, the different state changes highly depend on the scheduling policies, as well as the method used to avoid priority inversions. This method thus only computes valid results for one specific combination. Furthermore, this does not identify clearly the problematic executions, but rather only provides some metrics.

2.5 Literature review conclusion

As there is renewed interest in tracing with multi-core systems, we can see that it is already widely developed in terms of available tracing and analysis tools. Unfortunately, trace analysis tools accessible for non-specialist users are not yet available. Such tools would need a minimal and simple intervention from the user to specify its system constraints, and thus verify that those constraints are satisfied during execution, by tracing the application and the operating system.

Many analysis methods and processes exist for traces and systems, but using automated trace analysis is already less widespread. Our survey of common analysis methods for execution traces in the area of real-time systems has shown two types of analysis: the model checking oriented analysis tools and the data extraction tools. The model checking tools work by generating a checker tool using specifications that will then be used to check execution

traces. The data extraction tools use a more data-oriented approach, extracting patterns and information from the execution traces. The model checking tools are mostly specific to an application type or programming language, and sometimes use integrated tracers that are limited in terms of instrumentation and collected data, and do not provide low-overhead tracing. **LTng** has been presented and compared to its competitors in terms of both instrumentation (kernel and userspace tracing with correlated clocks, dynamic and static instrumentations, etc.), collected data (use of performance counters, variable context, etc.) and real-time performance (less than 10 microseconds of latency added by active tracepoints). The low impact of this tracer and its functionalities lead us to think that we could make an original contribution that would benefit from these, while adapting and improving the existing approaches based on less performing or accurate tracers.

When it comes to constraints representation and modeling formats, even if UML is a safe choice because of its flexibility and standardization, we have seen that many different approaches are proposed to fill the void of the real-time and resources constraints. Each of those approaches uses a different type of diagram to reach the desired representation. A sequence diagram would be interesting if we were to represent process switches and interactions, for instance to check the validity of a protocol. However, when tracing, each event gives an information on the state of the traced system or application, and does not mean a change in the active process or operation. An event primarily means that its tracepoint has been reached, and each event can give a different information. State machines therefore gives us the advantage of representing directly our events on a stateful model in an intelligible way. Similarly, previous work on automatic trace analysis using pattern matching, although limited to kernel traces, have represented systems and constraints with state machines [85]. Moreover, SCXML provides a simple and generic way to transcribe state machines with full specifications.

Finally, we identified another gap in model inference and pattern recognition, as existing approaches do not take fully advantage of system and application metrics. Moreover, most of these approaches do not distinguish valid and invalid executions while searching for patterns in a trace. The trace used to infer the model is considered as representing only valid executions of the program. It means that when inferring the model, the user must be sure that all the traced instances of its application are valid in order to build an accurate model.

CHAPTER 3 METHODOLOGY

In this chapter, we present the methodology organizing this research through the different steps, before depicting an overview of the three articles.

3.1 Research steps

This section presents the different steps around which we organized our research to reach our objectives.

3.1.1 Analysis system definition and organization

Constraints definition. The first task for our research was to identify the type of constraints that will be used and the best way to define the model to represent them. The constraint types must be useful for real-time applications and systems. The model needs to be upgradeable to include new types of constraints if needed. Based on our literature review, the representation of the model should be based on state machines annotated with specific constraints. The files used to store and load those state machines would be using the SCXML semantics.

Pattern identification. The next step was pattern identification. Having a model representation of an application and a generated trace, we needed to correlate the events of the trace with our model, to then be able to verify the given constraints. UST tracepoints were to be used here for following the transitions between the states of the model. UST offers flexibility for defining events, thus allowing to instrument many applications with different events and workflows, therefore easily linking the trace to the model representation of that application.

Constraints categorization. The constraints needed to be categorized. We needed to separate those that can be verified by using only UST traces (i.e. needing only information about the application internals, such as deadline constraints for instance), and those that need a complementary kernel trace (i.e. needing information about the system state during the execution of the application, such as CPU usage for instance). We also needed to determine an efficient way to store and retrieve the data, to verify the constraints that needed complementary kernel trace information. Our constraints verification methodology needed

to include an uncertainty level, depending on the accessible trace types and content, and the categories of constraints that need to be verified, according to our model. In this way, constraint verifications were to be partial, if not all the data required for the trace analysis was available.

3.1.2 Detection and analysis of the problems

Constraints verification and problem detection. The constraints will then be verified using the available data. One trace could contain multiple instances of the same task (a task represents one run of the model representation from start to end), thus allowing to verify those multiple instances at the same time. Depending on the verification results, we could classify the task instances in three classes, depending on how they satisfy the constraints: valid, invalid, uncertain. The valid status represents the instances that satisfied the constraints and for which we had complete information available to match with the model. The invalid status represents the instances that did not satisfy one or more of the constraints. Finally, the uncertain status represents the instances that satisfy the constraints, for which we had enough information to match with the model, but still unmatched with other constraints due to a lack of data. Each instance can be split into “instance steps”, that correspond to a change from one state to itself or another in the instance. Instance steps will also be classified, as well as each constraint that was verified during the transition.

Problem root cause identification. Using the statuses determined for the instances, instance steps and constraints, we can easily, for each constraint, define a list of valid and invalid cases. Using the instance steps per constraint allow for a wider range of valid and invalid cases, as an instance can be valid but contain valid and invalid instances steps. This is also true for instance steps and constraints. This means that, for each constraint, we aim at listing the valid and invalid instances to be compared. This comparison extracts the discrepancies between valid and invalid cases in order to either pinpoint directly the root cause of the detected problem, or start another analysis which will help us do so.

3.1.3 Automated model inference

Construction of the state machine. In order to infer models, we need first to find the patterns in the execution traces. A trace can contain multiple instances of a task, and it is important to split these tasks to generate an appropriate model. Only the UST traces are needed for that identification, as we are only working on constructing the state machine representation of the workflow of the application. We outlined previously that the state of the

model is defined by the UST tracepoints of the application. Once the different instances of the application are split, we can use pattern comparisons between the instances to understand what are the events that appear each time in the trace. These events, and their flow, will therefore be used to construct the state machine.

Inference of the constraints. The state machine constructed in the previous step will only allow to verify that the general workflow of the application is followed when tracing it. Depending on the available information, the generated state machine can be improved to perform a more thorough verification. Using kernel traces, constraints can then be added on the expected CPU usage, number of preemptions, etc. Real-time constraints can also be added, such as a maximum latency between two states. All of those constraints can be automatically identified for the worst working case available, and the user can select, among those, which ones to apply to the model for verifying its application.

3.2 Overview of the articles

The article presented in Chapter 4 treats the analysis system definition and organization, as well as the constraints verification and problem detection. Thereafter, the article in Chapter 5 builds on those concepts and on the detection process to provide problem root cause identification. Finally, the article in Chapter 6 focuses on the automated model inference process. Chapter 7 completes those articles by discussing case studies with automated model inference.

CHAPTER 4 ARTICLE 1: DETECTION OF COMMON PROBLEMS IN REAL-TIME AND MULTICORE SYSTEMS USING MODEL-BASED CONSTRAINTS

Authors

Raphaël Beamonte
Polytechnique Montréal
raphael.beamonte@polymtl.ca

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

Published into Scientific Programming, March 16th, 2016

Reference R. Beamonte and M. R. Dagenais, “Detection of Common Problems in Real-Time and Multicore Systems Using Model-Based Constraints,” *Scientific Programming*, vol. 2016, Mar. 2016, article ID 9792462, acceptance rate: 9%.

4.1 Abstract

Multi-core systems are complex in that multiple processes are running concurrently and can interfere with each other. Real-time systems add on top of that time constraints, making results invalid as soon as a deadline has been missed. Tracing is often the most reliable and accurate tool available to study and understand those systems. However, tracing requires that users understand the kernel events and their meaning. It is therefore not very accessible. Using modeling to generate source code or represent applications’ workflow is handy for developers, and has emerged as part of the model-driven development methodology.

In this paper, we propose a new approach to system analysis using model-based constraints, on top of userspace and kernel traces. We introduce the constraints representation, and how traces can be used to follow the application’s workflow and check the constraints we set on the model. We then present a number of common problems that we encountered in real-time and multi-core systems, and describe how our model-based constraints could have helped to save time by automatically identifying the unwanted behavior.

Keywords Linux, Real-time systems, Multi-core systems, Performance analysis, Tracing, Modeling

4.2 Introduction

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

Modeling allows technical and non-technical users to define the workflow of an application and the logical and quantitative constraints to satisfy. Modeling is also often used in the real-time community to do formal verification [71]. Models and traces could thus be used together to define specifications to satisfy and to check these against the real behavior of our application. This real behavior, reported in traces, would moreover take into account the influences of other running applications, the system resources and the kernel tasks. Using kernel traces information, we could therefore extend the set of internal constraints, to add system-wide constraints to satisfy, such as a minimum or maximum CPU usage or a limit on the number of system calls our application can do.

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

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

We present the related work in section 4.3. We explain our approach on using model-based constraints and present some specific constraints in section 4.4. We then detail some common real-time and multi-core application problems, as part of case studies to evaluate our proposed approach, in section 4.5. Results, computation time and scalability for our approach are shown in section 4.6. Future work and the conclusion are in section 4.7.

4.3 Related work

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

4.3.1 Existing software userspace tracers

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

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

Feather-Trace [11] uses very lightweight static events. It was mainly designed to trace real-time systems and applications, and is thus a low-overhead tracer. **Feather-Trace**'s Inactive tracepoints only cause the execution of one statement while active ones executes two. It uses multiprocessor-safe and wait-free FIFO buffers and achieve buffer concurrency safety using atomic operations. This tracer achieves low-overhead by using its own event definitions of a fixed size. The memory mechanism for these events is based on indexed tables. However, this design choice limits overhead but makes **Feather-Trace** unable to add system context information to the events, for instance. In its current form, the tracer also cannot use the standard `TRACE_EVENT()` macro to access system events and, even with improvements, would not be able to take advantage of the different event sizes and the information it provides. Also, **Feather-Trace** does not include a writing mechanism for storing the traces on permanent storage. Finally, the timestamp source used is the `gettimeofday()` system call, limited to microsecond precision.

Paradyn uses dynamic instrumentation by inserting calls to tracepoints directly in the binary executables [86]. Although the instrumentation can be done at runtime [87], **Paradyn** uses a patch-based instrumentation to rewrite the binary, only imposing a low-overhead latency [13]. This method has been used to monitor and analyze the execution of malicious code. This tracer however offers limited functionality. It cannot switch to another buffer when the buffer is full nor can it store the tracing data to disk while tracing. Furthermore,

it cannot support the definition of different event types. It thus isn't possible to use the Linux kernel static tracepoints defined by the standard `TRACE_EVENT()` macro, nor to add system context information. Also, no assurance can be given on the tracing condition for multi-core systems. In addition, **Paradyn** imposes an overhead proportional to the number of instrumented locations.

Perf [14] is one of the built-in Linux kernel tracers that was designed to access the performance counters in the processors. Its use was however later extended to interface with the `TRACE_EVENT()` macro, and thus access the Linux kernel tracepoints. Yet, **perf** is mostly oriented towards sampling. It is possible to use **perf** as a regular tracer but it has not been optimized for this use. If sampling does allow low-overhead, making it interesting for real-time systems, it does so by sacrificing accuracy. Furthermore, the collection process is based on an interrupt, which is both costly and invasive. Finally, **perf**'s multi-core scalability is limited [15].

The *Function Tracer*, or **ftrace**, is a set of different tracers built into the Linux kernel [16]. It was created in order to follow the relative costs of the functions called in the kernel to determine the bottlenecks. It has since evolved to include more comprehensive analysis modules such as latency or scheduling analysis [17]. **Ftrace** is directly managed through the `debugfs` pseudo-filesystem, and works through the activation and deactivation of its tracers. It can connect to the static tracepoints in the kernel through its `event tracer` using the `TRACE_EVENT()` macro [21]. It collects only data defined in this macro using the `TP_printk` macro, to save analysis time on the tracer side. This behavior, however, comes with the drawback of not being able to add system context information to trace events. Finally, **ftrace** can also connect to user-space applications using UProbes, since Linux kernel 3.5. This instrumentation is using interruptions though, which adds unacceptable overhead for most real-time and high performance applications and systems.

SystemTap [24] is a monitoring system for Linux primarily aimed at the community of system administrators. It can instrument dynamically the kernel using KProbes [88] or interface with static instrumentation provided by the `TRACE_EVENT()` macro. It can also be used to instrument user-space applications using UProbes, since Linux kernel 3.8. The instrumentation is done in both cases using a special scripting language that is compiled to a kernel module. The data analysis is directly bundled inside the instrumentation and the results can be printed at regular interval on the console. As far as we know, the analysis being done in-flight, there are no efficient builtin facilities to write events to stable storage. Moreover, user-space probes as well as kernel probes, even if they have been statically compiled in precise places, incur an interrupt to work. If this interrupt is avoidable on the kernel side by

using only the static instrumentation provided by `TRACE_EVENT()`, this is not possible on the userspace side. Interrupts add overhead that can be problematic for real-time tracing.

LTTng-UST provides macros for adding statically compiled tracepoints to programs. Produced events are consumed using an external process that writes events to disk. Unlike **Feather-Trace**, **LTTng-UST** uses the Common Trace Format, allowing the use of arbitrary event types [34]. The architecture of this tracer is designed to deliver high performance. It allocates per-CPU ring-buffers to achieve scalability and wait-free properties for event producers. Moreover, control variables for the ring-buffer are updated using atomic operations instead of locking. Also, read-copy update (RCU) data structures are used to protect important tracing variables. This avoids cache-line exchanges between readers that occur with traditional read-write lock schemes [89, 33]. A similar architecture is available for tracing at the kernel level. Moreover, kernel and userspace timestamps use the same clock source, allowing events to be correlated across layers at the nanosecond scale. This correlation is really useful to understand the behavior of an application. Finally, previous work demonstrated **LTTng**'s ability for high performance tracing of real-time applications [90]. **LTTng** is therefore the best candidate to trace real-time and multi-core systems while correlating userspace and kernel activities.

4.3.2 Model-checking analysis and data extraction tools for traces

In this section, we present different approaches used for model-checking analysis on traces. We also review interesting tools aiming at extracting data from traces.

Tango [44, 45] is an automatic generator of backtracking trace analysis tools. It works using specifications written in the **Estelle** formal description language. It is based on a modified **Estelle-to-C++** compiler. **Tango** generates tools that are specific to a given model, and that allow to check the validity of any execution trace against the specifications, using a number of checking options. However, **Tango** can only be used for single-process specifications and needs a NIST X Windows Dingo Site Server to do its analysis. Moreover, it was mainly designed to validate protocol specifications, and therefore does not provide a way to specify constraints based on the system's state.

Other algorithms to automatically generate trace checkers are presented in [91]. These algorithms follow the same idea as **Tango** as they use formulas written in a formal quantitative constraint language, Logic of Constraints, in correlation with traces. They can thus analyze a traced simulation for functional and performance constraint violations. The specifications file is converted to **C++** source, which is then compiled to generate an executable checker. Using simulation traces, the executable will produce an evaluation report mentioning any

constraint violation. However, this tool uses text-format traces and is thus very sensitive to any change in the trace format.

Scalasca [55] aims to simplify the identification of bottlenecks using execution traces. It offers analysis using both aggregated statistical runtime summaries and event traces. The summary report gives an overview of which process, in which call-path, consumes time and how much. The event traces are used for a deep study of the concurrent behavior of programs. **Scalasca** analyzes the traces at the end of the execution to identify wait states and related performance properties. It then produces a pattern-analysis report with performance metrics for every function call-path and system resource. If it allows to extract interesting metrics from the runtime of an application, **Scalasca** does not allow to provide our own specifications.

SETAF [60] is a framework to adapt the system execution traces and dataflow models to have the required properties for analysis and validation of the QoS. **SETAF** works using **UNITE**, which describes a method to use system execution traces in order to validate the distributed system QoS properties [61]. **SETAF** acts as an overlay used by **UNITE** to transform the traces and provide the missing information. To do so, it requires the user to first manually analyze the execution trace to identify what properties need to be added to the dataflow model, and thus provide the correct adaptation pattern. This adaptation pattern will allow to add information leading to the creation of a valid execution flow, and new causality relations between log formats in **UNITE**, but requires the user to have deep understanding of the trace format and **UNITE** requirements.

Trace Compass [62] is a graphical interface in **Eclipse** for the **LTTng** tracing tools. It supports multiple types of trace formats and provides different views showing specific analysis of the traces. Amongst these views, **Trace Compass** provides analysis for real-time applications [63] and an analysis of the system-level critical path of applications [64]. The later aims to recover segments of execution affecting the waiting time of a given computation. Finally, **Trace Compass** also allows the creation of state system attribute trees and to store metrics throughout time in the State History Tree database. This database provides efficient queries to the modeled state of the traced system for any given point in time.

To our knowledge, model analysis is not yet exploiting all the available information. By combining model analysis and trace analysis tools, the gap of unused information can be reduced. This would allow the specified behavior of the system to be verified through its execution trace, during or after running our application. Previous work has also been done on automatic kernel trace analysis using pattern matching, through state machines [85]. This work shows that trace events could be used to follow the workflow on an application, and thus link the states of a state machine to the states of a running application.

4.4 Using model-based constraints to detect unwanted behaviors

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

In this section, we will present our approach, which uses finite state machine models and constraints over kernel and userspace traces to detect unwanted behaviors in programs. These models will require instrumented applications to delimit the constraints application. We will first detail the general representation, and then propose some model-based constraints that could be applied to existing applications.

4.4.1 General representation

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

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

Internal structure

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

The variables are used to get and store the values of the metrics we need to verify. There are three main categories of variables: the state system free variables (not based on the state system such as those used to store timestamps or values available directly from the received events), the counter variables (or counters, such as those used to store the number of system

calls throughout time) and the timer variables (or timers, such as those used to store the time spent running a process). The variables are categorized depending on the number of calls needed to get their value from our state system.

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

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

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

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

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

transition, thus making it uncertain. Finally, a transition will be valid if and only if all of its linked constraints are valid.

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

Models

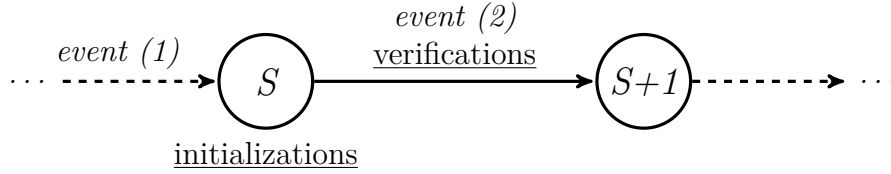


Figure 4.1 State machine representation that can be used to check metrics using traces

Figure 4.1 shows a representation of a section of a state machine for an application where we would like to verify some metrics. The states in the Figure are named “S” and “S+1” respectively for the start and end states of the zone we want to check.

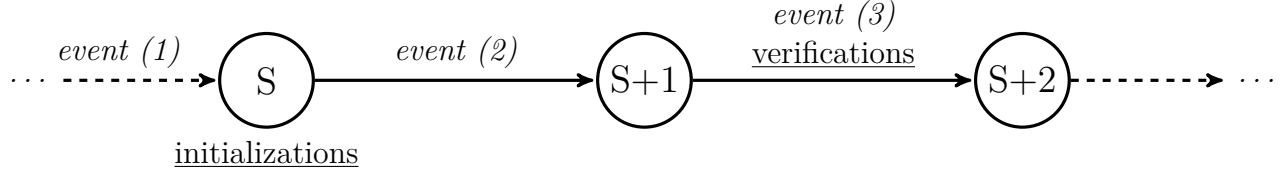
The “event (1)” string represents the event that would be used to enter state S of the state machine, and the “event (2)” string would be the one used to move from state S to state $S+1$.

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

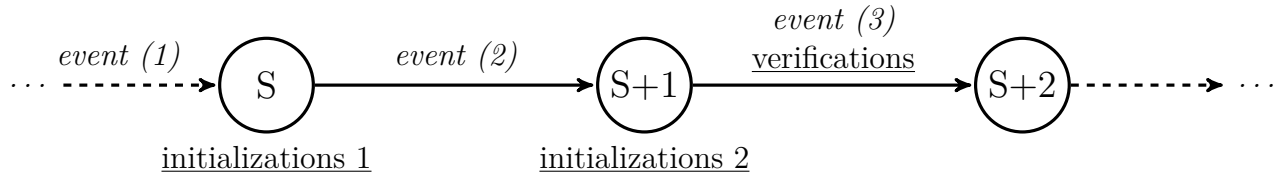
Finally, the “verifications” string represents the list of constraints we would verify when passing from state “S” to state “S+1”, i.e. when reading an event of type “event (2)”.

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

This also allows to initialize variables at one state, but to only check them at a later state of our state machine, as shown in Figure 4.2(a). The period of the constraint would then only be larger than if we used a more recent initialization. This also allows to check multiple constraints at one point, while the initializations appeared at different states of the application workflow, as shown in Figure 4.2(b). Having larger check periods would not add up to the



(a) The verifications will use initializations that only appear at state “S”



(b) The verifications will use both initializations declared at different states

Figure 4.2 State machine representation with late verification of constraints and a transitional state

verification time, since constraints validations are done in a constant number of operations for a given category of constraint, as explained in 4.4.1.

It is also possible for a state to have two (or more) next states. It would still be possible to validate the related constraints. In such cases, different events would be used for each of the possibilities, as shown in Figure 4.3. When an event is reached while in state “S”, we would automatically know if that event was of the type “event (2)” or “event (3)”. We thus would be able to move to the right state, and thus to read and execute the related verifications, if any.

Finally, in our approach, an event can be used at each junction of the model, but only once per junction. This removes any uncertainty about the flow to follow, in order to verify the constraints. Using this and the possibility to have multiple exits per node, we could for instance allow to execute the initializations each time we encounter an event of type “event (1)”, to only verify metrics between the last event of type “event (1)” and the first of type “event (2)”. Figure 4.4 shows a representation of this example. If we want to implement this specific example, we would not define any constraint in “verifications 1”.

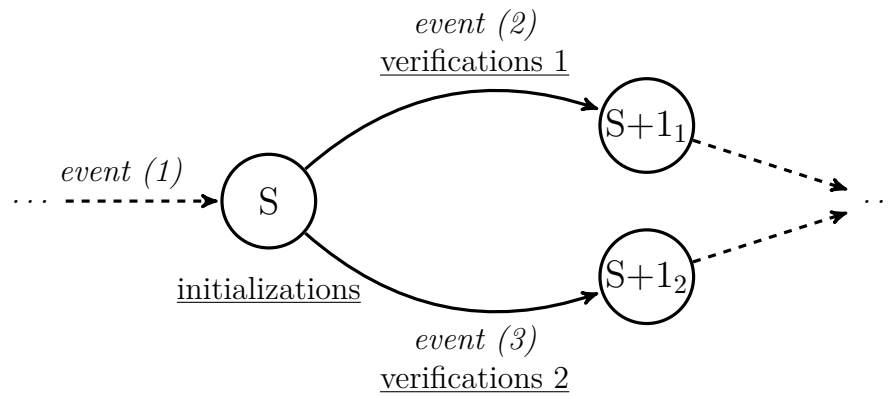


Figure 4.3 State machine representation with multiple next states for state “S”

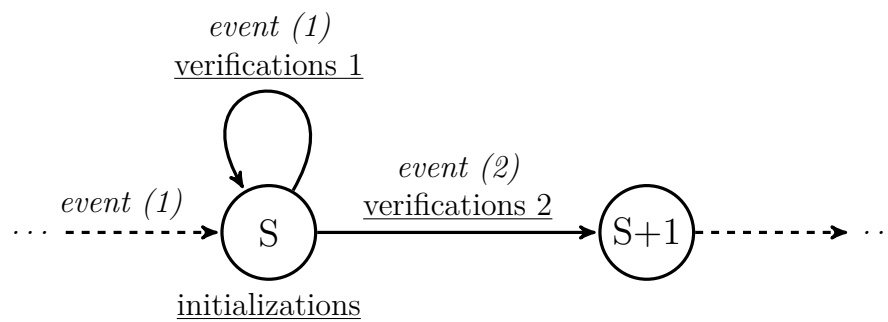


Figure 4.4 State machine representation using a loop, to go over the initializations when reading an event of type “event (1)”

4.4.2 Specific constraints on metrics

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

Deadline constraint

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

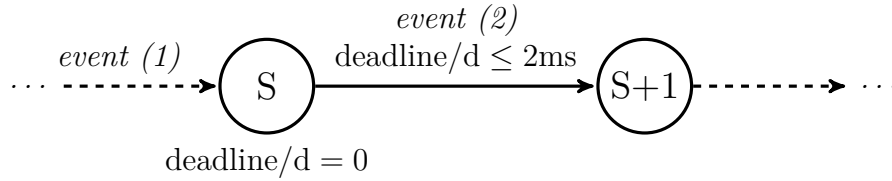


Figure 4.5 State machine representation of a constraint validating whether we spent at most 2 ms between the states “S” and “S+1”

Figure 4.5 gives a model representation of a constraint that could be used in that case. State “S” is the state of the application when starting the time-related task. State “S+1” is the state of the application when it finishes that task. The events of type “event (1)” and “event (2)” are the events generated by tracepoints in the application when switching to those states.

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

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

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

Preemption constraint

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

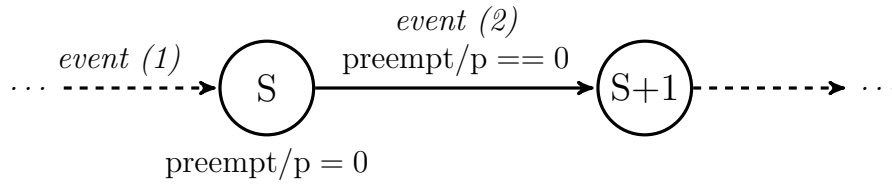


Figure 4.6 State machine representation of a constraint validating that our process has not been preempted between the states “S” and “S+1”

Figure 4.6 gives a model representation of a constraint that can be used to limit the number of preemptions that the process suffers during the given period, delimited by the “S” and “S+1” states.

When entering the “S” state, a preemption counter has to be initialized to know how many preemptions the process has experienced when reaching the “S+1” state. In the Figure, we initialize a preemption counter variable “p” using the string “preempt/p = 0”. When entering the “S+1” state, the preemption counter variable is checked to validate that we didn’t have any preemption, using the string “preempt/p == 0”. We could also have allowed at most one preemption for instance, using the string “preempt/p ≤ 1”.

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

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

The preemption constraint is using a counter variable.

Resource usage constraint

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

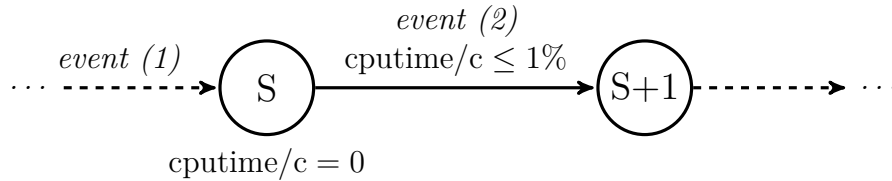


Figure 4.7 State machine representation of a constraint validating whether our process used at most 1 % of the CPU time between states “S” and “S+1”

Figure 4.7 gives a model representation of a constraint that could be used in that later case, and could be easily changed to be used for the former.

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

Programmatically, we use the events of type “event (1)” and “event (2)” to delimit the time period during which we look at the CPU usage. Using the kernel traces for the same time

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

The resource usage constraint is using a timer variable.

Wait status constraint

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

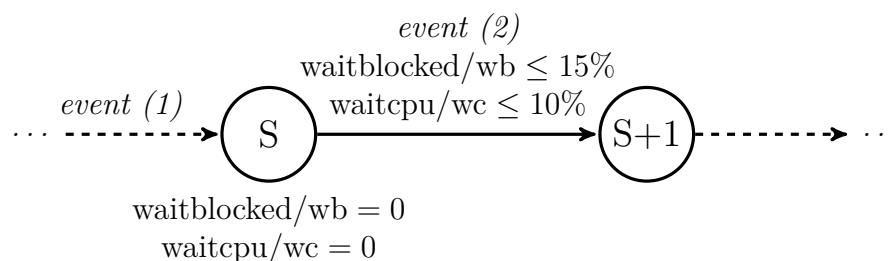


Figure 4.8 State machine representation of constraints validating whether the process spent at most 10 % of the time period between states “S” and “S+1” waiting for a CPU, and at most 15 % of this same period being blocked

Figure 4.8 shows a representation of both “wait-for-cpu” and “blocked” status constraints being used on a model.

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

Programmatically, the events of type “event (1)” and “event (2)” would allow to delimit the working time period. We would then look at the kernel events in that period to check the status of our process and compute the time percentage spent in the status we want to check, in the same way that we computed this information for the CPU usage constraint, but using the new state of the unscheduled process to know if it is now waiting for CPU or blocked.

This information is directly computed in our state system; we can thus access it easily for the given interval and verify our constraint.

The wait status constraint is using a timer variable.

System calls constraint

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

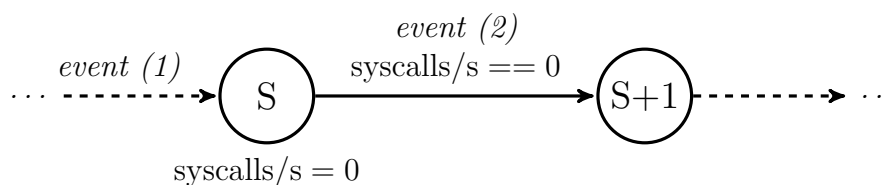


Figure 4.9 State machine representation of a constraint validating that our process has not done any system call between states “S” and “S+1”

Figure 4.9 gives a model representation of a constraint that can be used to limit the number of system calls issued by the process during the given period.

When entering the “S” state, a system calls counter variable is initialized. In the Figure, the system calls counter variable “s” is initialized using the string “syscalls/s = 0”. When entering the “S+1” state, we check this counter to validate that we didn’t have any system call since the “S” state. We use the string “syscalls/s == 0” to do so.

Programmatically, we count the number of kernel events whose name starts by “syscall_entry”, using the events of type “event (1)” and “event (2)” to limit the search of these events. For each event encountered that matches our search, we can increment our system calls counter.

We can then compute the difference for that counter and use that difference to check against the requirement.

The system calls constraint is using a counter variable.

4.5 Case studies

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

4.5.1 Occasional missing of deadlines

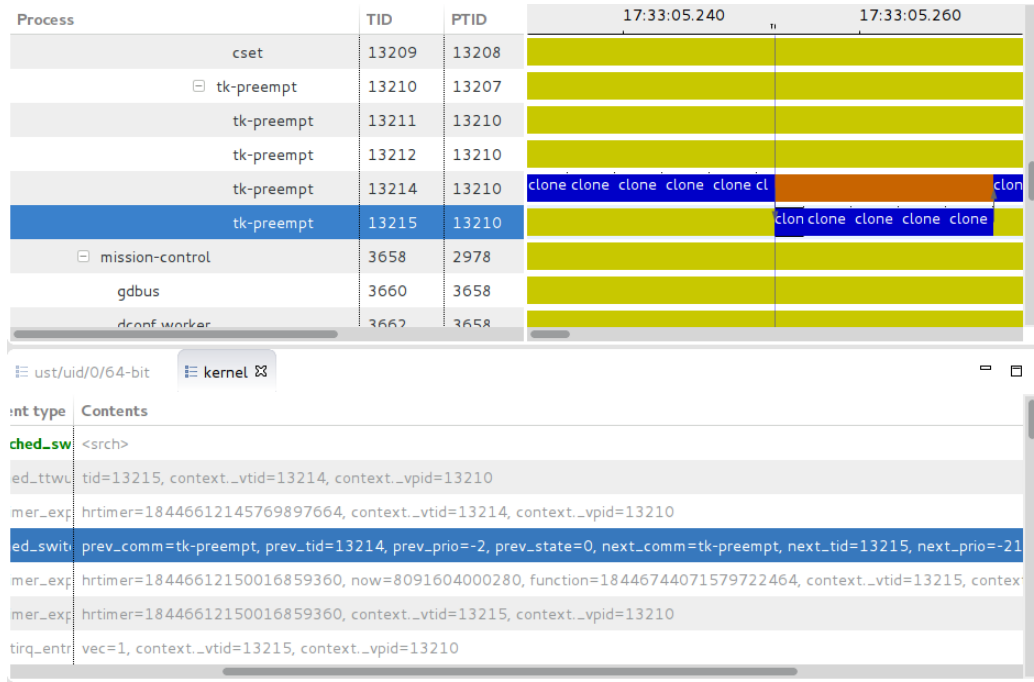
Problem summary

In real-time systems, we have to comply with the given deadlines for a task. That task can happen multiple times in a short period of time. In some cases that we encountered, a task happening up to 1 000 times per second was missing its deadline one or two times per second. That task being a hard real-time one, those missed deadlines were not acceptable.

Trace analysis approach

Kernel and userspace traces were used to identify the task execution and see what happened on the kernel side. These traces lead to see that for each task that did not reach its deadline in time, another process of higher priority was scheduled instead. That process was not scheduled the rest of the time, letting the other tasks – having the same system priority as the ones missing their deadlines – reach their deadlines.

Figure 4.10(a) shows a visualization of such situation in Trace Compass. We can see that the thread of TID 13 214 is running uninterrupted while the thread of TID 13 215 is in wait-blocked status (yellow on the Figure). As soon as the thread 13 215 exits its wait-blocked status, it is scheduled on the CPU, instead of thread 13 214. The later is then in wait-for-cpu status until the former finishes its task and returns in wait-blocked status. As shown in Figure 4.10(b), that preemption was caused because the priority of the thread of TID 13 214 was only of -2 (field “prev_prio”) while the one of the thread of TID 13 215 was of -21 (field “next_prio”). In this case, we need to read the priorities in reverse, meaning that thread 13 215 had a higher priority and thus preempted the other while it was running.



(a) Screenshot of Trace Compass showing the preemption

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

(b) Trace event “sched_switch” happening to do the preemption

Figure 4.10 Preemption of a process by another of higher priority (−2 vs −21, the lower the value, the higher the priority)

Using model-based constraints

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

Depending on what we expect our application to do, we could also take advantage of other constraints like a preemption constraint or a CPU usage one to get more information as to why we don't follow the expected workflow. These constraints would, however, need kernel traces to be verified. Figure 4.11 shows the state machine representation of our example using all three mentioned constraints.

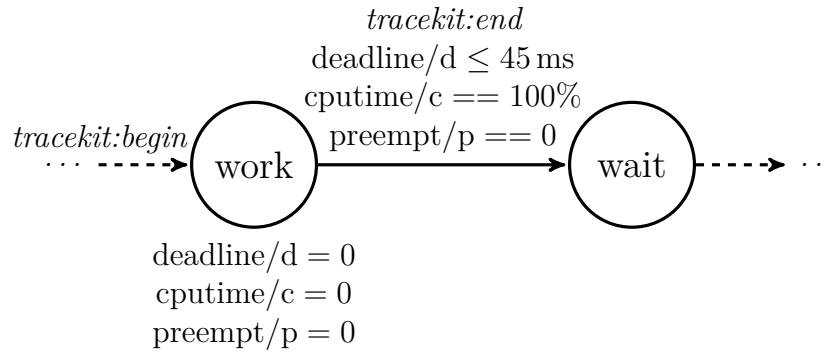


Figure 4.11 State machine representation of **tk-preempt**'s work using constraints to check if our process spent at most 45 ms working, used 100 % of the CPU time and was not preempted during its critical real-time task

4.5.2 Priority inversion

Problem summary

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

In previous work [90], we wanted to allow tracing such applications. We thus created a minimal **UST**-traced application doing only loops and calculating their duration, and saw that a delay was unfortunately added when tracing. Our application, even while being the highest priority one in the system, was unscheduled at some point while being traced.

Trace analysis approach

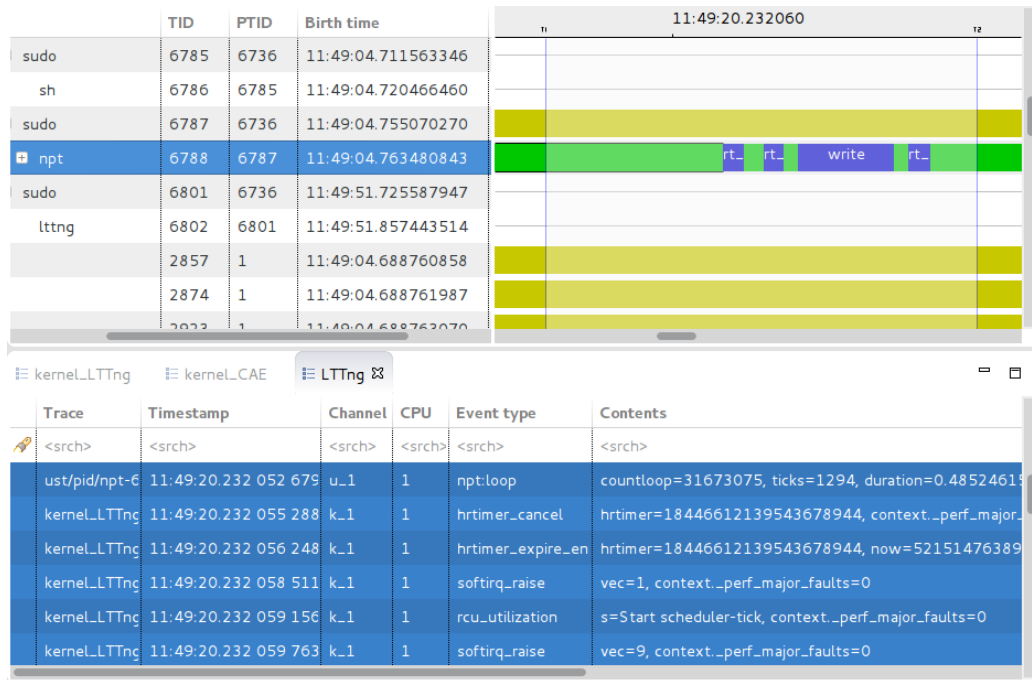
Using kernel traces, it is possible to see the different processes being scheduled and compare their priority.

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

Figure 4.12 shows the high duration between the two “loop” events of the `npt` application. We can see on Figure 4.12(a) the presence of system calls in between the two events. Looking further at the trace, we can see a number of system calls running on the scheduled CPU, as listed in Figure 4.12(b). We can also see that “npt:loop” events are only 0.48 μ s apart when there is no system call.

Using model-based constraints

With our model approach, we can use a state identifying that our application is in a loop and, for each event read that informs us we’re doing another iteration of the loop (“npt:loop” in our example), a constraint would be validated. This constraint could be a CPU usage constraint for instance, ensuring that our application had at least a high share of its CPU. We could otherwise use a preemption constraint, to limit the number of times that our application has been preempted during that iteration of the loop. Finally, if we consider that our application should only be working in userspace during the given period of time, a syscall constraint can be used. Figure 4.13 shows the state machine representation of our example using both a CPU usage and a syscall constraints.



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

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

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

Figure 4.12 Unexpected kernel work while tracing an userspace-only application

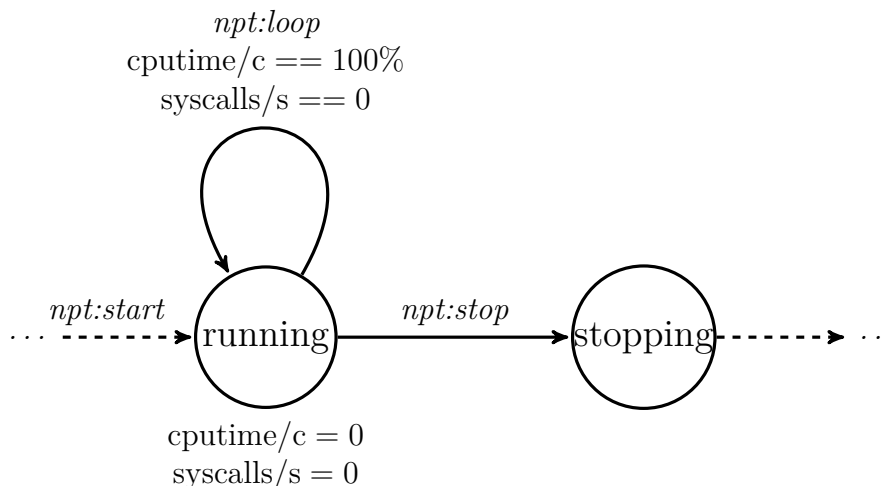


Figure 4.13 State machine representation of `npt`'s loop using constraints to check if our process used 100 % of the CPU time between each loop iteration

4.5.3 Unefficient synchronization method

Problem summary

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

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

Trace analysis approach

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

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

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

Using model-based constraints

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

4.5.4 Wait-blocked processes on multiprocessor activity

Problem summary

Processes sometimes expect high performance for multi-threaded tasks on a multi-core system. In these cases, cache accesses and synchronization are usually optimized to achieve a good scalability. However, it may happen that some part of the task misses these opti-

mizations and does not scale well, causing regressions when using parallel cores. For high scalability multi-threaded processes, this behavior should be avoided.

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

Trace analysis approach

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

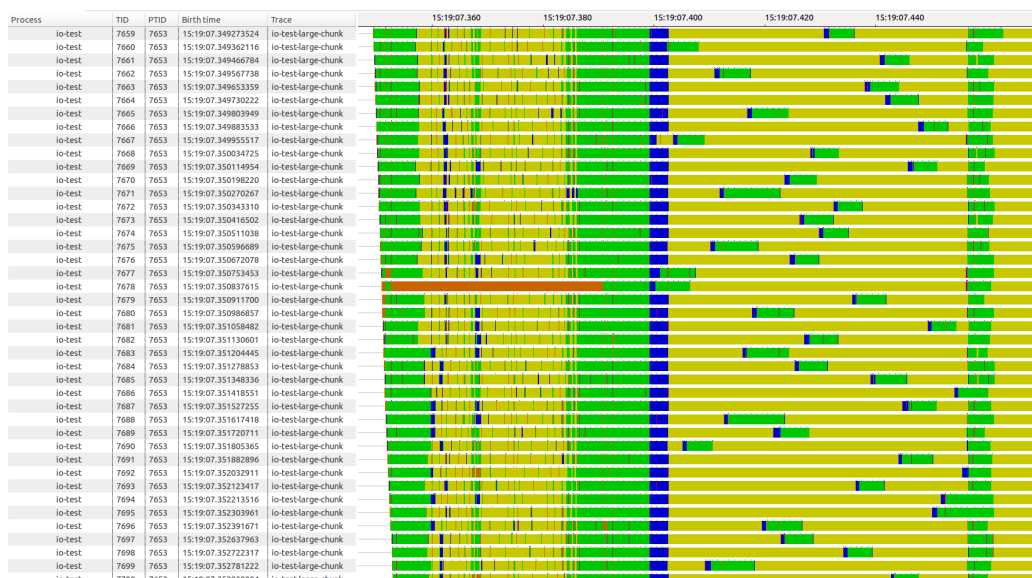


Figure 4.14 Screenshot of Trace Compass showing the barrier at which threads are waiting before unmapping operations, after their calls to `munmap` [92]

This behavior is normally associated with the use of a barrier, as seen in Figure 4.14. We were able to find in the Linux kernel source code that Linux uses a global semaphore protecting the `mm_struct` data structure. We thus found a solution to circumvent the problem for our application, using a unique thread for memory mapping. The scalability of the `mmap` system call was also analyzed in [93].

Using model-based constraints

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

4.5.5 Wait-blocked processes while using external resources

Problem summary

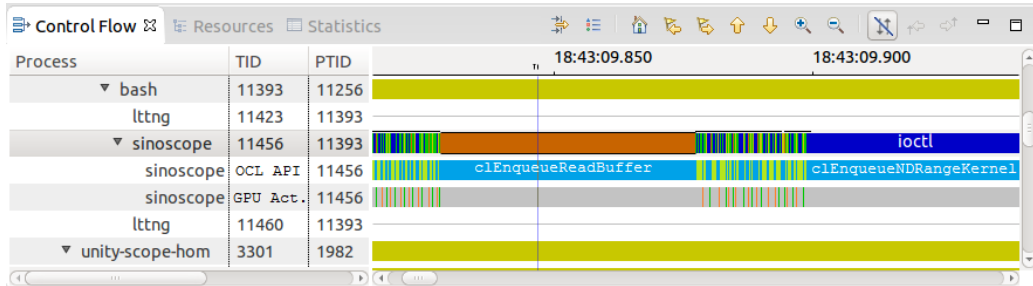
External resources are necessary in some cases to perform specific tasks. For instance, for some highly parallel computing tasks, GPUs are becoming more and more interesting, as compared to multi-core CPUs. In such cases, computation or data rendering depends on another processing unit, different than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

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

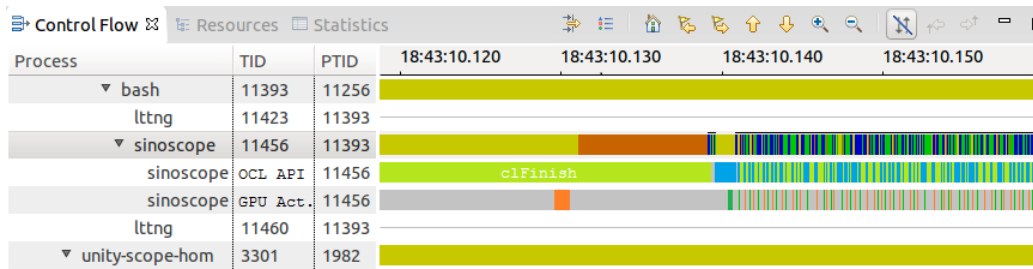
Trace analysis approach

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

In some situations, the latency was induced by CPU preemption: the GPU had finished its work but unable to get back to the process, currently unscheduled or already busy, as shown in Figure 4.15(a). In some other cases, as we can see in Figure 4.15(b), it was linked to GPU sharing: the process was in wait-blocked status, waiting for the GPU to get back to it, while the GPU was not working on that task, being already busy on another one. As a solution for such situations, the process itself and its GPU tasks can be optimized to improve their use of the available resources. On the CPU side, it could be using a higher priority for the process to prevent preemption. On the GPU side, it could be a better division of the tasks.



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



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

Figure 4.15 Views showing wait situations while using external resources; these views do not exist in the Trace Compass mainline version yet [94]

Using model-based constraints

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

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

4.6 Analysis results and time

Using the case presented in 4.5.1, the associated detection constraints and the model represented in Figure 4.11 to automatically identify its unwanted behavior, we ran our analysis and benchmarked the time it took to analyze different traces detailed in Table 4.1. This case has the ability to provide interesting results, as we can use for it the three different categories of variables available in our approach.

Table 4.1 Number of events and sizes of the traces used to benchmark our analysis

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

4.6.1 Constraints validation

Figure 4.16 gives an overview of the different outputs of our analysis, depending on the constraints satisfaction and available data. These results are just a section of the full report containing all the instances of the application, according to our model and their analysis

results. For these examples, we used the trace 1. For all those results, the first part shows the entry in the work state when receiving a “tracekit:begin” event, and the entry in the wait state when receiving the “tracekit:end” event. We also have the timestamp of the read event, and we can see the list of initialized variables and constraints, if any.

Results presented in Figure 4.16(a) and Figure 4.16(b) were computed using both the userspace and kernel traces. In Figure 4.16(a), we can see that the requirements were satisfied, and thus each constraint is in the valid status. In Figure 4.16(b), none of the requirements were met, thus setting all constraints to the invalid status. We can see in that latter case that the computed value is shown in the report to understand why the requirements were not met. The results presented in Figure 4.16(c) were computed using only the userspace trace. We can see that given only the userspace trace, the analysis was not able to verify if all the constraints were satisfied and thus set the CPU usage and preempt constraints to uncertain status, while the deadline constraint has been analyzed and is, in this case, invalid as the time spent in the work state was 45.4054ms while the maximum was 45 ms.

Figure 4.17 shows some invalid sections, as reported by our tool for other cases presented in section 4.5. Figure 4.17(a) shows a section in which, even though the application discussed in 4.5.2 was still scheduled and should have been running only in userspace, four system calls were executed, making the section last for more than 49 μ s. Figure 4.17(b) shows that MongoDB, discussed in 4.5.3, took much more time than expected for running a command. Finally, Figure 4.17(c) shows that the application discussed in 4.5.4 spends an unexpected amount of time in the wait-blocked status.

4.6.2 Running time

Switching on and off the different constraints put in the model represented in Figure 4.11, we benchmarked the running time of our analysis. Our test system consist of an Intel® Core™ i7-4810MQ CPU at 2.8 GHz, with 16 GiB of DDR3 RAM at 1600 MHz. Table 4.2 shows the results for trace 1 while Table 4.3 shows the results for trace 2.

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

```

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

```

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

```

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

```

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

```

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

```

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

Figure 4.16 Results of the analysis using the model-based constraints on userspace and kernel traces


```

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

```

(a) Invalid section for the case presented in 4.5.2 when verifying the model represented in Figure 4.13, during which four (4) system calls were issued

```

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

```

(b) Invalid section for the case presented in 4.5.3 when verifying a deadline constraint of less than one (1) second for a task length, which lasted nearly five (5) seconds in this case

```

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

```

(c) Invalid section for the case presented in 4.5.4 when verifying a wait blocked constraint of less than ten percent (10%) for a task, which spent more than fifteen percent (15%) of its time being blocked in this case

Figure 4.17 Examples of invalid sections as reported by our tool for other cases discussed in section 4.5

Table 4.2 Average (*Avg.*) and standard deviation (*Std. dev.*) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace *tk-preempt (1)*

Constraints	Time (in s)		
	State system build	Model build & constraint verif.	Total
Deadline			
<i>Avg.</i>	3.8500	0.343 52	4.1935
<i>Std. dev.</i>	0.230 75	0.024 853	0.235 32
CPU usage			
<i>Avg.</i>	3.8592	0.729 52	4.5888
<i>Std. dev.</i>	0.204 32	0.047 765	0.210 79
Preemption			
<i>Avg.</i>	3.8271	0.461 62	4.2887
<i>Std. dev.</i>	0.177 92	0.043 790	0.188 40
All three			
<i>Avg.</i>	3.8609	1.1251	4.9860
<i>Std. dev.</i>	0.223 94	0.042 746	0.224 84
None			
<i>Avg.</i>	3.8312	0.157 89	3.9891
<i>Std. dev.</i>	0.179 02	0.017 863	0.179 54

Table 4.3 Average (*Avg.*) and standard deviation (*Std. dev.*) of the time taken in seconds by a run of the model-based constraints analysis, computed using 100 runs of the analysis of the trace *tk-preempt (2)*

Constraints	Time (in s)		
	State system build	Model build & constraint verif.	Total
Deadline			
<i>Avg.</i>	29.663	6.9751	36.638
<i>Std. dev.</i>	0.842 78	1.1635	1.5931
CPU usage			
<i>Avg.</i>	29.599	41.223	70.821
<i>Std. dev.</i>	1.1128	1.0844	1.7550
Preemption			
<i>Avg.</i>	29.462	22.220	51.682
<i>Std. dev.</i>	1.0688	0.599 45	1.2692
All three			
<i>Avg.</i>	29.766	58.855	88.620
<i>Std. dev.</i>	1.1275	1.2559	1.8220
None			
<i>Avg.</i>	30.049	5.2234	35.272
<i>Std. dev.</i>	0.980 00	0.919 95	1.3683

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

Table 4.4 Average (*Avg.*) and standard deviation (*Std. dev.*) of the time taken in seconds to build the state system during the first run, versus to verify if it exists in the subsequent runs, computed using 100 runs of the analysis of the traces

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

For each trace, however, we can see in Tables 4.2 and 4.3 that the state system build time is always the same, independently of the active constraints. It thus only depends on the kernel trace size. This is because the state system is computed to acquire all the metrics necessary to set constraints at once, no matter which ones are actually used. While this behavior is costly for the first run, the state history tree database built is saved in stable storage to allow fast access for the following runs, as shown in Table 4.4.

4.6.3 Scalability

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

In order to measure the scalability relative to trace length, we generated a number of traces containing events needed to follow the model presented in Figure 4.11. Each data point presented in Figures 4.18 and 4.19 is the average elapsed execution time over twenty runs of our algorithm.

Figure 4.18 presents the results using the different userspace traces and our different categories of constraints. We can see in the Figure that for all categories of constraints used,

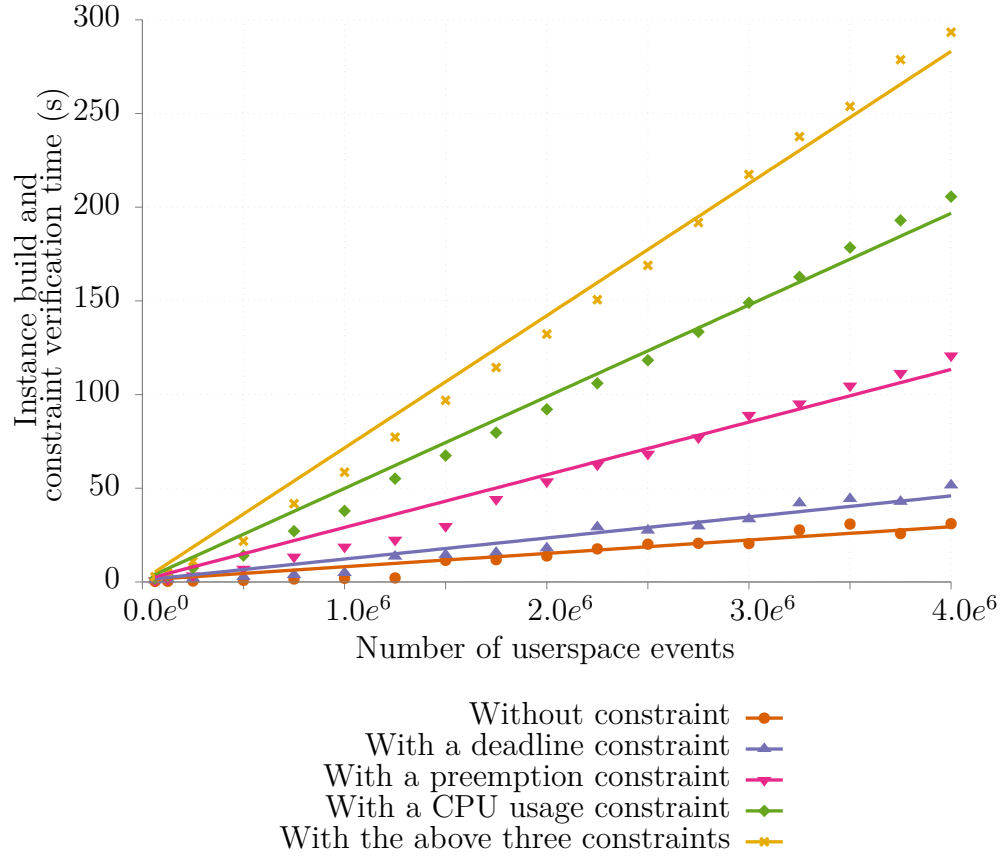


Figure 4.18 Time (in s) to build the instances and check their constraints as a function of the number of userspace events. Lines represent linear regressions of the data.

the complexity of the approach is proportional to the number of userspace events. We also observe that it takes about twice as much time to use timer variables as compared to counter variables. Both those variables are more expensive than using a state system free variable.

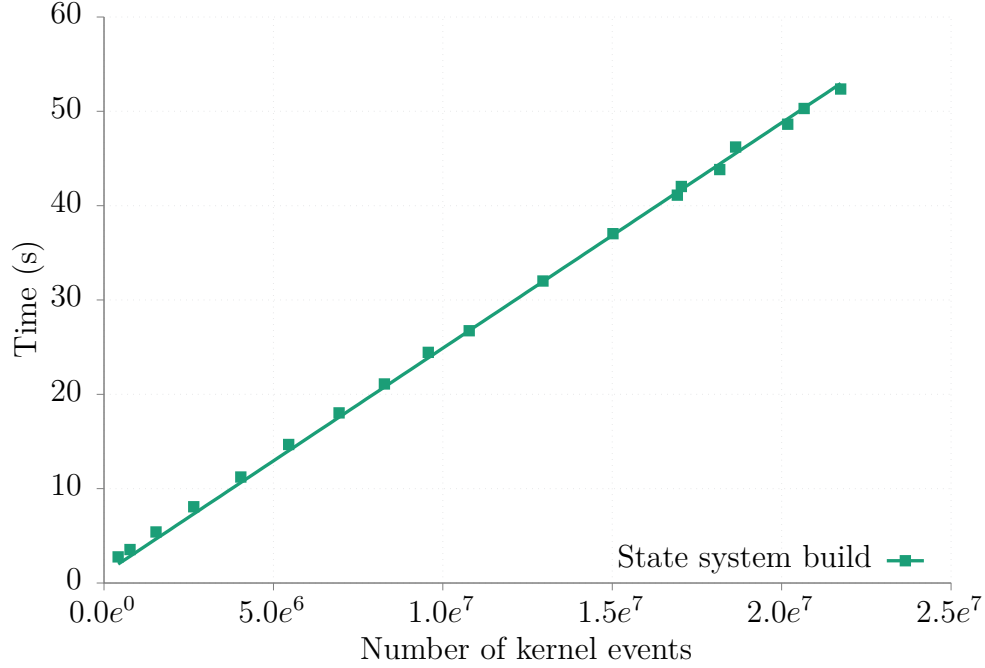


Figure 4.19 Time (in s) to build the state system as a function of the number of kernel events. The line represents a linear regression of the data.

Figure 4.19 shows the results using the different kernel traces. It shows that the time it takes to build the state system is proportional to the number of kernel events in the trace.

Also, the linear regressions in both Figures 4.18 and 4.19 show that the proportionality follows a linear pattern for both kernel and userspace traces.

To analyze the model scalability, we used the trace *modelbench* (3) that contains 100 calls to 1024 different tracepoints. Each data point presented in Figures 4.20, 4.21 and 4.22 is the average elapsed execution time over twenty runs of our algorithm.

We used this trace to first consider a model without constraint, and with only one transition per state, to analyze the scalability according to the number of successive states in the model, as shown in Figure 4.20. We see that the time it takes to build the instances is proportional to the number of states involved.

We then analyzed a situation in which the number of states was fixed, but the number of transitions from one state to the other was variable. Figure 4.21 shows the results for this case that does not support any constraint, where the transitions are each based on a different

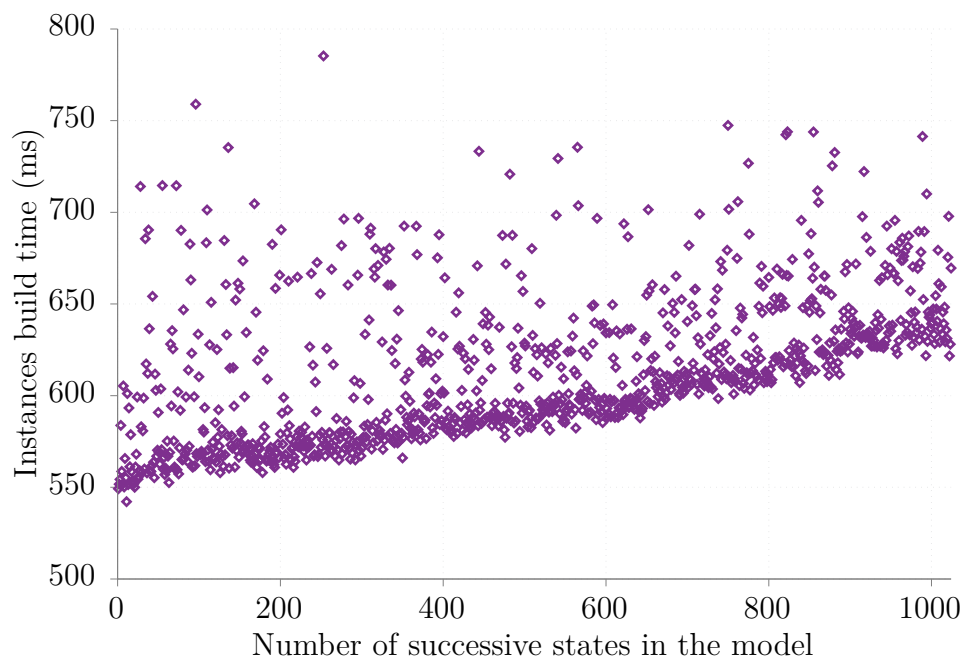


Figure 4.20 Time (in ms) to build the instances as a function of the number of successive states in the model

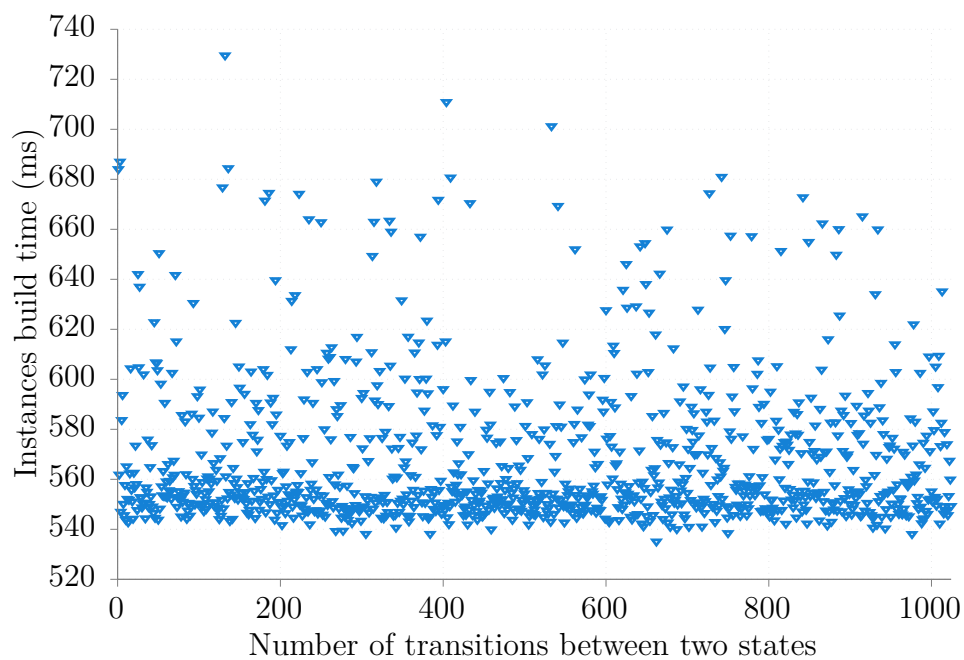


Figure 4.21 Time (in ms) to build the instances as a function of the number of transitions between two states

trace event. These results illustrate that the number of transitions between two events does not impact the time it takes to build the instances.

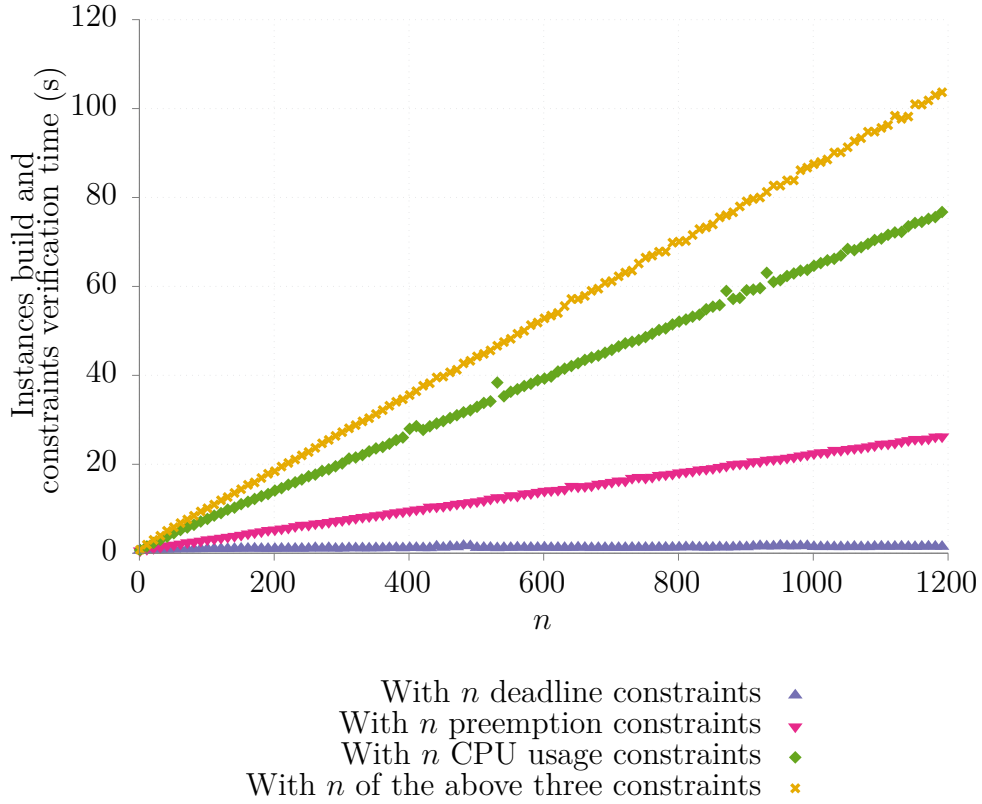


Figure 4.22 Time (in s) to build the instances and check their constraints, as a function of the number and categories of constraints between two states

Finally, we studied the constraints scalability by fixing the number of states and transitions, and by varying the number of constraints on that transition. We repeated that test for the three different categories of constraints, and for a case using one constraint of each category. Figure 4.22 shows the results of those tests. We can observe that in each case the time is linearly proportional to the number of constraints involved.

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

4.7 Conclusion and Future Work

We have presented our approach for application modeling, using model-based constraints, and kernel and userspace traces, to automatically detect unwanted behavior in real-time and

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

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

4.8 Disclosure

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

4.9 Competing Interests

The authors declare that they have no competing interests.

4.10 Acknowledgments

The authors are grateful to Mathieu Côté, David Couturier, François Doray, Francis Giraldeau and Fabien Reumont-Locke for the cases studied in this paper. This research is supported by OPAL-RT, CAE, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ).

CHAPTER 5 ARTICLE 2: MODEL-BASED CONSTRAINTS OVER EXECUTION TRACES TO ANALYZE MULTI-CORE AND REAL-TIME SYSTEMS

Authors

Raphaël Beamonte
Polytechnique Montréal
raphael.beamonte@polymtl.ca

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

Submitted to ACM Transactions on Modeling and Performance Evaluation of Computing Systems, June 14th, 2016

Reference R. Beamonte and M. R. Dagenais, “Model-based constraints over execution traces to analyze multi-core and real-time systems,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, to appear: submitted.

5.1 Abstract

As part of the model-driven development methodology, modeling is used to represent the application workflow or automatically generate source code. This methodology is convenient for developers, particularly to create or improve real-time applications embedded in complex mechanical systems.

Multi-core systems are difficult to monitor because of the multiple concurrently running processes that can interfere with each other. When such a system has to be real-time compliant, timing constraints add to the system complexity, invalidating results as soon as a deadline is missed. In such circumstances, tracing is usually the most accurate and reliable tool available to study and understand the behavior of those applications. However, the interpretation of voluminous detailed execution traces requires from its users a deep understanding for the operating system and application behavior, and a long time to dig through the millions of trace events.

In this paper, we present the use of model-based constraints on top of user-space and kernel traces to provide weighted analysis results. We describe our model-based constraints system and how constraints are validated or invalidated. We then introduce how to extract, organize,

compare and weight data from the trace. We then explain how we assign responsibilities to unwanted behaviors. Our algorithms have been applied to multiple input traces showing common problems for multi-core real-time systems, while varying the number of trace segment of interest instances, duration of the instances, position of the instances as well as the number of events. The experimental results show that our algorithm can quickly identify many different types of problems, while its runtime stays in the order of a few tens of seconds, even for traces with millions of events, thus helping to save time when analyzing thousands of trace events for complex systems.

Keywords Linux, Real-time systems, Performance analysis, Tracing, Modeling, Automatic analysis

5.2 Introduction

To quickly diagnose problems, in complex time-sensitive applications, developers usually require system analysis tools. Tracers are particularly interesting, as they do not freeze the application, while still providing a lot of information on its runtime, and the status of the system throughout the traced period. Every tracer has characteristics such as its overhead, intrusiveness, precision level and whether it can trace the kernel, user-space, or both. Yet, all of those tracers require significant and tedious human intervention to analyze the detailed and extensive information contained in traces. Moreover, sufficient expertise to understand those events is also necessary to take advantage of such information.

The workflow of an application, and the quantitative constraints to satisfy, can easily be defined by technical as well as non-technical users through modeling. Modeling is also widely used in the real-time community for formal verification [71]. Using models and traces together allows to automatically check specifications against the real application behavior. These traces take into account the influence of other running applications on an application's workflow. However, uncovering specification violations just provides a general idea of the non-compliance area, and the detailed trace information may be used to further analyze what happened when a constraint is invalid. The automation of this further analysis is the focus of the present work.

This paper describes a new approach to use kernel and user-space traces as well as model-based constraints to automatically identify the origin of an unwanted behavior in applications by means of comparison. It also demonstrates this approach on common problems encountered in real-time and multi-core applications. The analysis results save time by pointing directly to the system activities being the most probable cause, such as the most likely pro-

cess responsible for preemption, the state in which the application is staying too long, or the system call that should not have happened. Further analysis can then be started automatically to achieve a more precise diagnostic. Our main contribution is to provide an automated analysis of the probable origins of problems. Those are discovered thanks to system-side events such as process preemption, system calls and scheduling.

We present the related work in section 5.3. The model-based constraints approach is presented in section 5.4. We then explain our algorithms for data extraction, organization and comparison, as well as how we assign responsibilities in section 5.5. Results, computation time and scalability for our approach are shown in section 5.7. Suggestions for future work and the conclusion are in section 5.8.

5.3 Related work

This section presents the related work in the two main areas relevant for this paper, software tracers for both kernel and user-space, and tools for the analysis of traces using model-based constraints.

5.3.1 Software tracers for both kernel and user-space

In order to extend the checking and analysis of the specifications of an application, we first need to acquire trace data at both the application and operating system levels. It is also necessary to prioritize the low disturbance and high precision of the tracer used to generate those traces. In this section, we thus present the characteristics of currently available software tracers, that provide both user-space and kernel tracing capabilities, with the ability to interface with the Linux kernel `TRACE_EVENT()` macro.

Perf [14] is a built-in Linux kernel tracer. It was at first designed to access the processors' performance counters, but its use has later been extended to interface with the `TRACE_EVENT()` macro and the Linux kernel tracepoints. However, the main orientation of **perf** is for sampling and, even if it is possible to use it as a regular tracer, it has not been optimized for this. Sampling is interesting for high performance systems as it allows for low-overhead, but it loses in accuracy as compared to regular tracing. Besides, an interrupt is at the origin of the collection process, making it invasive and costly. Finally, **perf**'s scalability for multi-core is limited [15].

The *Function Tracer* (**ftrace**) is a set of built-in Linux kernel tracers [16]. Its primary aim was to determine the bottlenecks in the kernel by monitoring the relative cost of the called functions. Since then, it has evolved to include other analysis modules, such as schedul-

ing or latency analysis [17]. The `debugfs` pseudo-filesystem allows to manage `ftrace` and to activate and deactivate its tracers. The *event tracer* of `ftrace` allows the use of the `TRACE_EVENT()` macro [21]. To save analysis time on the tracer side, `ftrace` only collects data defined in this macro using the `TP_printk` macro. However, this removes the ability to add system context information to trace events. Since Linux kernel 3.5, `ftrace` can also use UProbes to connect to user-space applications. Still, UProbes uses interruptions for its instrumentation, adding unacceptable overhead for high performance and real-time applications.

SystemTap [24] is a Linux monitoring system that was created for system administrators. It can use both static and dynamic instrumentation with `TRACE_EVENT()` or KProbes respectively [88]. Since Linux kernel 3.8, it can also instrument user-space applications using UProbes. In either case, the instrumentation is written in a special scripting language, then compiled to a kernel module. Some data processing is done in-flight, inside the instrumentation, and the results can be printed at regular interval. There are no special facilities included to write events to stable storage. Moreover, KProbes as well as UProbes, incur an interrupt for each tracepoint. Like `ftrace`, UProbes being the only option on the user-space side, the interrupt overhead cannot be avoided, which is problematic for real-time and high performance applications.

LTTng-UST is the user-space component of **LTTng**. It provides macros to add statically compiled tracepoints to programs. An external process consumes the produced events and writes them to disk. **LTTng-UST** allows the use of arbitrary event types, thanks to the Common Trace Format [34]. **LTTng-UST** was designed for high performance, scalability and wait-free properties for event producers, by using per-CPU ring-buffers. Moreover, atomic operations are preferred to locking for the update of the ring-buffer control variables. Read-copy update (RCU) data structures are used to protect important tracing variables, thus avoiding the cache-line exchanges between readers that happen when using read-write lock schemes [89, 33]. The kernel level uses an equivalent architecture. Also, kernel and user-space tracers use the same clock for timestamps, allowing an easy correlation of events across layers at the nanosecond scale. Finally, **LTTng** can trace real-time applications with low latency [43]. **LTTng** is therefore the best candidate to trace real-time and multi-core applications at both the kernel and user-space levels.

5.3.2 Automatic data extraction tools and model-checking for traces

In this section, we present different tools and methodologies used for data extraction and model-checking for traces.

Tango [44, 45] allows to automatically generate backtracking trace analysis tools. Using formal specifications, it will generate tools that are specific to a given model. The tools will then allow to verify that any execution trace complies to the specifications. **Tango** was mainly designed to validate protocol specifications and does not allow to specify constraints based on the system's state. This limitation also prevents from further analysis of a specification invalidation using other resources.

The Logic of Constraints checker [91] also allows to generate an executable checker from formulas written in a formal quantitative constraint language. An evaluation report is generated by running this executable on simulation traces. This report will inform of any constraint violation. This tool is, however, using text-format traces, making it very sensitive to any change in the trace format. Moreover, no further analysis than the constraint verification is offered.

Scalasca [55] provides a way to identify bottlenecks using execution traces. It allows to analyze aggregated statistical runtime summaries to identify which process consumes CPU time and how much. An analysis of event traces is also available, and used for a deeper study of the concurrency of programs. The traces are used to identify wait states and linked performance properties. **Scalasca** then generates a pattern-analysis report containing performance analysis metrics for each function and system resource. Yet, **Scalasca** does not allow to provide our own specifications.

SETAF [60] is a framework that allows to add properties for the analysis and validation of the QoS in system execution traces and dataflow models. In its workflow, **SETAF** requires the user to manually analyze the execution trace to identify what are the missing properties, and thus provide the related adaptation patterns. This pattern will then be used to add those properties and create the necessary causality relations. By this workflow, **SETAF** requires the user to have a deep understanding of the trace format. Also, no further analysis than the specifications invalidation is provided.

Trace Compass [62] is an **Eclipse** graphical interface for **LTTng**. It provides multiple views to show specific analysis of the traces. Some of these views are of particular interest for our research, such as the analysis for real-time applications [63] and the analysis of the system-level critical path of applications [64]. The later allows to recover segments of execution that affect the waiting time of a given computation. **Trace Compass** also support different trace formats. Finally, it provides methods to create state system attribute trees, and to store metrics that vary along the time axis in the State History Tree database. This database allows to query efficiently the modeled state, for any given point in time.

To our knowledge, existing model analysis tools do not exploit all the available information and limit their analysis to the detection of invalidated constraints. Using model analysis and trace analysis tools, and by applying statistical methods and algorithms, we take advantage of this unused information to provide more insight to the user. This not only provides further details on the invalidated constraints, but also possible reasons explaining those invalidations.

5.3.3 Statistics algorithms for relation analysis

In order to take advantage of the available data, we based our work on common methods and algorithms used to compare and associate data. Those are reviewed in this section.

The edit distance is a method to quantify how similar or dissimilar two entities are to one another, such as strings [95] or trees [96]. To do so, it counts the minimum number of operations required to transform one entity into the other, amongst a defined limited set of operations. Multiple implementations exist that differ only in the set of operations. For instance, the Levenshtein distance allows the removal or insertion of a single element, as well as the substitution of one element for another.

The *association rule learning* is a data mining method that aims to find interesting relations between variables in large databases. These relations are identified using different measures of interestingness [97]. Association rules were first introduced to discover consumers regularities in supermarkets purchases [98]. Such application would then allow to identify that when a consumer buys items A and B, there is a high chance that he will buy an item C, and thus help in making decisions for products placement for instance. There are multiple algorithms implementing association rule learning using different approaches. One of the most used is the Apriori algorithm, using a breadth-first search to identify the frequent individual items, before extending them to larger sets, until these item sets do not appear often enough anymore [99].

5.4 Using model-based constraints

The developers usually have specific expectations when designing high performance applications. They know in which order the different operations should be performed, and have estimates for the values of different metrics and their evolution during the application runtime. With these expected values, a developer can check that the application is behaving as planned, and thus adjust accordingly the length of the debugging and tuning period.

Model-based constraints have been used in the past to specify and validate the workflow of an application or system using different metrics [100]. This has been extended with success

in previous work to include constraints based on more sophisticated metrics, extracted from operating system tracing [4]. This section provides an overview of such constraints, which is the basis for our work on automatic problem identification analysis.

5.4.1 Internal structure

The representation used is based on four different elements: the states, representing the multiple states of the application, the transitions, representing the movement from one state to itself or to another state, the variables, used to get and store the value of the metrics we aim to verify, and the constraints, used to express the specification of what we expect for the application execution.

The modeled state of the operating system and applications is maintained in an attribute tree called the state system. The state system is based on the state attribute tree of **Trace Compass**, for which the state history database is built to contain the different metrics needed for access later during the analysis. The values for those metrics are stored during the first reading of the kernel trace and can be accessed later by querying the state history database.

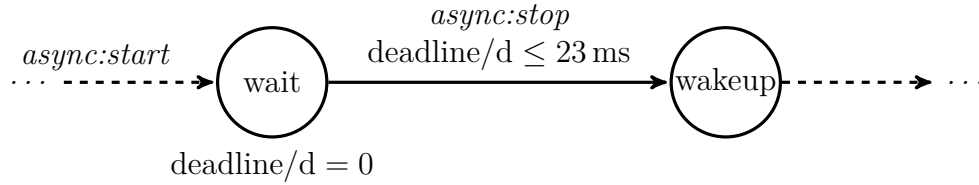
We distinguish three main categories of variables used to verify constraints: variables not dependent on the current state (state system free) , the counter variables – or counters – and the timer variables – or timers. The variables are categorized as such depending on the number of queries needed in the state system to compute their next value. It is necessary to make respectively 0, 1 and 2 queries to the state history database to compute the value of a state system free variable, counter variable and timer variables at a given time.

Each constraint is linked to a transition and is being checked when this transition happens. The constraint will then either be valid, invalid, or uncertain if data is missing to check that constraint. The linked transition will then be valid if all its constraints are valid, invalid if at least one constraint is invalid, or uncertain if at least one constraint is uncertain and none are invalid.

The succession of states and transitions is finally used to follow the multiple instances of the application that will be checked at each step, i.e. when we use a transition to move to a next state.

5.4.2 Model representation

Figure 5.1(a) shows the representation of a part of a state machine, for an application where we would want to verify some metrics. The two states of the application in the Figure are



(a) Graphical representation of the period of JACK2

```

<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0">
  <initial>
    <transition event="async:start" target="wait"/>
  </initial>

  <state id="wait">
    <onentry>
      <assign location="deadline/d" expr="0"/>
    </onentry>

    <transition event="async:stop" target="wakeup" cond="deadline/d &
      ⇨ lt;= 23ms"/>
  </state>

  <state id="wakeup">
  </state>
</scxml>

```

(b) XML representation of the critical real-time task of tk-preempt using State Chart XML

Figure 5.1 State machine representation of tk-preempt's real-time process using constraints to check that our process was not preempted during its critical real-time task

called “wait” and “wakeup”. *wait* represents the state that starts the period to check, and *wakeup* represents the state that ends that period. To follow the workflow of the application, user-space events are used: “async:start” is the event received that will signal us to enter state *wait*, while “async:stop” will signal to move from state *wait* to enter state *wakeup*.

The initializations are all the different variables initializations that will appear at a given state, to allow the future verification of our metrics. A variable is initialized by being set to 0. That initialization will create the given variable of the given type to be checked later in a constraint. For a variable *d* of type *deadline*, the initialization would look like the following: “deadline/d = 0”, as shown in the Figure.

The verifications are all the constraints that will be checked when following that transition. A constraint for a variable *d* of type *deadline*, that needs to be less or equal to a value *23ms*, would look like the following: “deadline/d \leq 23 ms”, as shown in the Figure.

Figure 5.1(b) shows a State Chart XML representation of the same section, but considering only that section as entry and exit points. Such files can automatically be used by the analysis application to build the state machine and its constraints.

5.5 Algorithms for data organization and extraction

Once the instances of matched patterns of the application have been built from the traces, using the process described in 5.4, where the different constraints were verified simultaneously, we end up with information about constraints validation, uncertainty or invalidation for each step. Even if that information allows us to pinpoint the location (tracepoint location and time) of a given problem, for instance a missed deadline or a preemption, it does not necessarily reveal the origin of that problem. To know the origin of the problem, someone would thus have a given location and time point to look at (tracepoints), and would also know the kind of unexpected behavior (constraint violation) to look for, but the subsequent investigation work would still have to be done by hand. Moreover, understanding the problem origin may require a good understanding of the application and of the operating system internal behavior.

As we work towards automating the developer’s work to save time and allow more informed decisions, this section presents our approach to organize the data, resulting from that previous analysis, to perform a more thorough analysis and extract more informative results. We first present the general data organization, before explaining the extraction process and the algorithms used.

5.5.1 Data organization

Data analysis is achieved by looking at what happened when encountering unexpected situations, and comparing it with what normally happens, identifying what is common in those unexpected cases. In the context of traces, this means identifying events or lists of events that can be significant, and in some cases comparing them with events or lists of events that we expected to appear, in order to highlight the discrepancies.

When a trace has been analyzed using the model-based constraints checking described earlier, we get a list of instances of the application, according to the workflow defined in the model used. These instances come along with the list of all the states encountered, and the events that brought us there; these are called instances steps. These steps contain themselves the list of all the constraints that have been verified when arriving there, the validation status for each of those constraints, as well as the step at which the verified variables were last initialized.

In order to obtain our list of significant elements, we first need to separate the validated cases from the invalidated ones, for each constraint. The cases for which the validation is considered uncertain are not considered, as we cannot be sure of the actual outcome for the concerned period. Our aim is to build, for each constraint that has been invalid at least once, two lists containing respectively all the invalid and valid cases for this constraint. We thus browse the constraints, for each instance step of each instance, and store the constraints validation information in either the valid or invalid list. This means that an instance that has been invalid for one constraint can still be used as a valid instance for another constraint. Once the data is organized in this way, we run a variable-specific analysis, corresponding to the type and category of the variable used in the constraint, for all the constraints having at least one invalid occurrence.

This analysis will receive both the valid and invalid lists of instances, that can then be used for the identification of significant elements to compare.

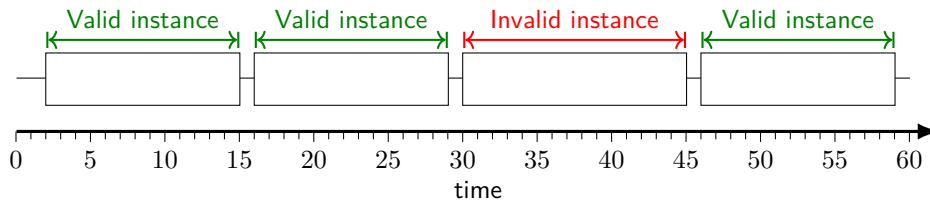


Figure 5.2 Example of a trace containing 4 instances, following a model specifying that the instance execution duration should be of 13 units of time; 3 of the instances are valid, and 1 is invalid with a duration of 15 units of time.

Figure 5.2 gives an example for which we would end up with 3 instances in the list of valid instances, and 1 instance in the list of invalid ones.

5.5.2 Data extraction

The data extraction process follows multiple stages for each variable that has to be analyzed. These stages are the instances selection, the extraction of elements of interest from these instances, and the assignation of responsibilities.

The work done in some of these stages depends on the algorithm chosen to do the analysis. Two algorithms are available, the partial analysis algorithm and the full analysis algorithm. The algorithm choice is based on the kind and value of the constraint to analyze. If our constraint is absolute, meaning that any change to the variable value during the analyzed period of time is prohibited, we will chose the partial analysis to directly compute responsibilities for each element leading to a variable value change. However, if changes to the variable value are authorized, a full analysis must be done in order to compare what is happening during valid and invalid instances, extract the discrepancies and then compute responsibilities. For instance, when analyzing a counter, if the constraint is not to have any increment to this counter, any increment to that value is responsible for the constraint violation. However, if we allow two increments to that counter but in some situations we end up with three increments, it is not clear which increment is responsible for the violation. Different computations are therefore needed for these different cases, hence the two analysis processes.

Selection of the instances

A large number of instances can be found in a trace, depending on the trace size and the model duration. Finding the source of violated constraints for a large number of instances can thus take a time linearly proportional to the number of instances. This can therefore impact the scalability of our analysis.

However, when the instances are regrouped as valid or invalid for a constraint, they will most likely follow a finite number of different patterns according to the elements of interest that we will next extract. This is because, for a real-time application, the tasks that have to be done are highly likely to be similar. This means that instead of checking all of the valid and invalid instances, we could determine a sample big enough in each case to consider that the different patterns are represented. This sample would then allow us to perform a much faster analysis with minimal precision loss.

A correct sample size for a given list of instances can be determined by using the following formula, which takes into account the finite population correction [101]:

$$S = \left\lceil \frac{\frac{z^2 \times p(1-p)}{e^2}}{1 + \left(\frac{z^2 \times p(1-p)}{e^2 N} \right)} \right\rceil \quad (5.1)$$

Where S is the computed sample size, N is the population size, e the margin of error, p the initial probability and z the z-score corresponding to the expected confidence level. In our case, the population size is the number of instances in the list in which we will select the sample. We set the initial probability as 50 %, which is the worst case scenario, leading to the biggest sample size. Finally, the margin of error and confidence level follow the industry standard of respectively 5 % and 95 %, which corresponds to a z-score of 1.96.

Once we determined the sample size, we need to determine which instances will compose the tested sample. In order for our results to be statistically correct, we need for the instances to each have an equal chance to be part of the sample, which means random sampling. Among the methods for random sample selection, we chose to use *reservoir sampling*, which allows the equality of chances for each member of the population to be in the selected sample. Reservoir sampling is originally derived from the Fisher-Yates shuffle [102] and exists in multiple implementations of different complexity [103, 104]. The one we use in our analysis is the efficient algorithm for sequential random sampling [105], which allows for fastest sample selection on big data sets.

Extraction of elements of interest

Once the instances to analyze are selected, we need to identify the key elements that will provide us with the source of the invalid constraint. We call these the elements of interest, or elements. They differ depending on the kind of variable used for the constraint.

Counter and timer variables directly follow values stored in our state system. This allows us to directly request the timestamps of the variable's value changes along the duration of the instance under consideration. Once these timestamps are extracted, we can directly read from the trace the events that triggered those value changes, and extract the data needed. For instance, the extracted data could be the name of the system call for a system calls counter, or the status of the process being unscheduled for a CPU usage timer.

The extraction process for state system free variables depends on all that can influence that variable. For instance, in the case of the deadline variable, we created a state system attribute

that follows the state of the process throughout the trace. Then, each change of the process state during the analyzed period is extracted as being of interest.

Independently of the constraint, elements are then stored as element durations. Element durations contain the element, and the exact value increment caused by the appearance of that element at that moment, for the variable used by the constraint. The element is thus the duration key, while the increment is the duration value. All the durations found during a period of time are then merged in an element duration set. Element duration sets thus contain a number of durations that may or not be about the same element. A duration set can be empty, which means that during the analyzed period, no element was found.

$$[a, b, c] = [b, c, a] = [c, a, b] = [c, b, a] = [a, c, b] = [b, a, c]$$

Figure 5.3 When merging duration sets, only the content is considered and not the order, which means that all the *keymaps* presented here are considered identical

For a given list of instances, similar element duration sets are then merged into element interval sets. We consider that two element duration sets are similar if they share the same *keymap*, i.e. the same number of durations about the same elements. This means that when merging duration sets into interval sets, we only use their content without considering the order of appearance. For instance, two duration sets containing only one occurrence of each *a*, *b* and *c*, no matter the order, will be merged in the same interval set, as shown in Figure 5.3. The interval sets will therefore share the same *keymap* as the duration sets that were used to compose them. It will moreover have minimum and maximum durations (for timers and state system free variables) or minimum and maximum values (for counters).

Returning to the example shown in Figure 5.2, we could consider a small real-time application which shares its time between computation, requesting a resource via a system call, and being preempted by a higher priority task. In such case, Figure 5.4 shows the content of the different valid instances and Figure 5.5 the different duration sets extracted from these instances. We can see that even if the order and duration for each element is different from an instance to another, we still have two occurrences of element **RUNNING** and one occurrence of each element **SYSCALL**, **BLOCKED** and **PREEMPTED**. The duration sets' *keymap* of each valid instance thus being the same, the duration sets of theses instances will be merged into one unique valid interval set.

When creating an interval set from a duration set, we obtain an interval set containing a number of intervals with identical minimum and maximum values. When merging a duration set into an interval set, the element durations of the duration sets will be matched with the

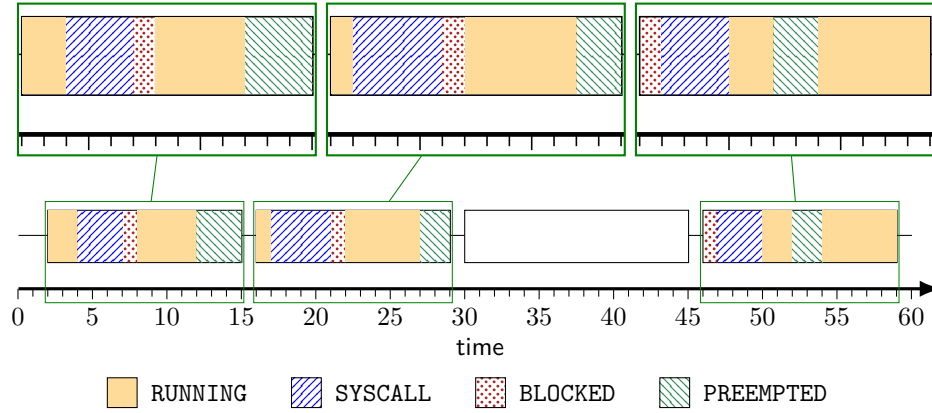


Figure 5.4 Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the valid instances which are in equal number for each element, but with different order or duration.

element intervals of the interval set in order to keep the resulting intervals as short as possible. For instance, if for an element **RUNNING** we have two intervals $[1, 2]$ and $[4, 5]$ and two durations 6 and 2 to merge into, the 2 will be merged in the interval $[1, 2]$ and the 6 in the interval $[4, 5]$. Doing so, the second interval will only become larger by 1, while the first interval will stay untouched.

Considering this whole process, we can deduce that if the duration set used to create an interval set was empty, the interval set will also be empty.

Figure 5.6 represents the valid interval set of our example, shown in the simplified form of the number of intervals about a given element, and the minimum and maximum values of the smaller interval containing all of the intervals for the element. We can observe that its *keymap* is the same as the duration sets' one, and that the different durations have been merged into the smallest intervals.

Assignation of responsibilities

The final stage of the analysis is to assign responsibilities to the elements extracted in order to identify clearly which element or elements have the most to do with the constraint violation.

When using the partial analysis algorithm, each and every element extracted is partly responsible, which means that all these have to be used to assign responsibilities. If we consider I to be the extracted invalid interval set and $I_{j,vmin}$ and $I_{j,vmax}$ respectively the minimum value and maximum value of the interval for the element j , then the minimum responsibility

```
DurationSet: [
  [RUNNING, 2, 6], // durations 2 and 4
  [SYSCALL, 1, 3],
  [BLOCKED, 1, 1],
  [PREEMPTED, 1, 3],
]
```

(a) The duration set of the first valid instance contains two durations for the element **RUNNING** that can be summed to 6 units of time, one duration for the element **SYSCALL** for 3 units of time, one duration for the element **BLOCKED** for 1 units of time and one duration for the element **PREEMPTED** for 3 units of time

```
DurationSet: [
  [RUNNING, 2, 6], // durations 1 and 5
  [SYSCALL, 1, 4],
  [BLOCKED, 1, 1],
  [PREEMPTED, 1, 2],
]
```

(b) The duration set of the second valid instance contains two durations for the element **RUNNING** that can be summed to 6 units of time, one duration for the element **SYSCALL** for 4 units of time, one duration for the element **BLOCKED** for 1 units of time and one duration for the element **PREEMPTED** for 2 units of time

```
DurationSet: [
  [RUNNING, 2, 7], // durations 2 and 5
  [SYSCALL, 1, 3],
  [BLOCKED, 1, 1],
  [PREEMPTED, 1, 2],
]
```

(c) The duration set of the third valid instance contains two durations for the element **RUNNING** that can be summed to 7 units of time, one duration for the element **SYSCALL** for 3 units of time, one duration for the element **BLOCKED** for 1 units of time and one duration for the element **PREEMPTED** for 2 units of time

Figure 5.5 The three duration sets extracted from the three valid instances presented in Figure 5.4; the durations contained in the set are shown in the simplified form of the number of durations for a given element and the sum of these durations


```
IntervalSet: [
  [RUNNING, 2, [1, 5]], // [1, 2] and [4, 5]
  [SYSCALL, 1, [3, 4]],
  [BLOCKED, 1, [1, 1]],
  [PREEMPTED, 1, [2, 3]],
]
```

Figure 5.6 Interval set resulting from the merge of the three duration sets in Figure 5.5; it contains two intervals for the element **RUNNING** that are both between 1 and 5 units of time, one interval for the element **SYSCALL** between 3 and 4 units of time, one interval for the element **BLOCKED** for 1 unit of time and one interval for the element **PREEMPTED** for 2 to 3 units of time

$r_{e,vmin}$ and maximum responsibility $r_{e,vmax}$ of the element e in the constraint violation can be formulated this way:

$$r_{e,vmin} = \begin{cases} \frac{I_{e,vmin}}{\sum_{j \in I} I_{j,vmin}} & \text{if } \sum_{j \in I} I_{j,vmin} \neq 0 \\ r_{e,vmax} & \text{if } \sum_{j \in I} I_{j,vmax} \neq 0 \\ \frac{1}{\#I} & \text{else} \end{cases} \quad (5.2)$$

$$r_{e,vmax} = \begin{cases} \frac{I_{e,vmax}}{\sum_{j \in I} I_{j,vmax}} & \text{if } \sum_{j \in I} I_{j,vmax} \neq 0 \\ r_{e,vmin} & \text{if } \sum_{j \in I} I_{j,vmin} \neq 0 \\ \frac{1}{\#I} & \text{else} \end{cases} \quad (5.3)$$

Which makes it possible to compute the mean responsibility r_e for element e , which is the one that will be shown to the user, as follows:

$$r_e = \frac{r_{e,vmin} + r_{e,vmax}}{2} \quad (5.4)$$

However, when using the full analysis algorithm, some of the extracted elements are expected to be there. The situation is thus more complex to analyze, since we cannot directly consider each element as incorrect, and responsible for the constraint violation. We thus need to identify the elements that are not there in valid instances, but appear in invalid ones.

To do that, we first select the valid instances to analyze and perform the extraction of elements of interest. As explained in 5.5.2, this stage leaves us with some element interval sets, each with different *keymaps*, called thereafter valid element interval sets. These are the baseline that will be used for comparison purposes.

We do the same thing with the selected invalid instances, from which we obtain a number of invalid interval sets. For each of these interval sets, we need to identify the closest valid interval sets in order to get the most accurate comparison. This is done by calculating, for each pair of valid and invalid interval sets, a weight that takes into account two distinct metrics: the distance between the interval sets in consideration and the number of occurrences of the valid one. We consider that the valid interval set closest to the invalid one has the highest probability of being the desired behavior, instead of the invalid behavior that lead to the violation. Furthermore, the probability increases with the number of occurrences of that valid interval set.

Distance between interval sets The distance between the current invalid interval set and a given valid interval set i is noted d_i and represents the differences in the invalid interval set that separate it from the valid one. The distance d_i is computed as being the average of the distances from the duration sets that compose the invalid interval set to the valid interval set.

$[a, a, b, c]$	to $[a, a, b]$	$(d = 1)$	remove c
	to $[a, a]$	$(d = 2)$	remove b
	to $[a]$	$(d = 3)$	remove a
	to $[\]$	$(d = 4)$	remove a
	to $[e]$	$(d = 5)$	insert e

Figure 5.7 Distance calculation step by step, showing the distance for intermediate sets

The distance calculation is largely based on an edit distance that only takes into account additions and deletions, as the order is not considered. An example of such approach, and intermediate distances when moving from a set of non unique elements to another, is given in Figure 5.7. We however consider the directional component of the analyzed constraint: when an element appears in the valid interval set but is not in the invalid one, if the constraint was for the value to be less than a given limit, this element will not add to the distance. Respectively, if the constraint was for the value to be higher than a limit, only the elements appearing in the valid interval set and not in the invalid duration set will add to the distance.

Finally, when the constraint is for the value to be equal or different than a given target, all the discrepancies have to add to the computed distance.

Algorithm 5.1 Distance calculation between an *interval* and a *duration*, taking into account the direction of the constraint C , the result being a numerical value

```

1 function distance(duration, interval,  $C$ )
2    $d \leftarrow \text{duration.value}()$ 
3    $i_{\min} \leftarrow \text{interval.minValue}()$ 
4    $i_{\max} \leftarrow \text{interval.maxValue}()$ 
5   if  $d \geq i_{\min}$  and  $d \leq i_{\max}$  then
6     return 0
7   if  $C$  in  $[<, \leq]$  then
8     return  $\max(d - i_{\max}, 0)$ 
9   else if  $C$  in  $[=, \neq]$  then
10    return  $\max(i_{\min} - d, 0)$ 
11  return  $\min(|i_{\min} - d|, |d - i_{\max}|)$ 

```

Algorithm 5.1 gives the pseudocode of the algorithm used to compute the distance between a single element duration and a single element interval. We can see that the directional component is taken into account while computing the distance.

Algorithm 5.2 uses Algorithm 5.1 in order to compute the full distance between a distance set and an interval set. The `closestPermutation` function called in Algorithm 5.2 returns the list of durations and interval in an order that allows for the shortest cumulated distance. This calculation can easily end up being of combinatorial complexity, when trying to select the x elements that will generate the smallest distance among y available elements. As we consider a statistical approach, another sample is selected randomly here, using the same process as used for the selection of instances. This allows to keep the algorithm scalable. The size of the returned distance duration set is considered as being the distance value.

Following on our example, we can see the content of the invalid instance in Figure 5.8. This instance contains occurrences of the element `IRQ` that was not existent in valid instances, but does not contain any occurrence of element `BLOCKED` that was in valid instances. Moreover, we can see in the extracted duration set presented in Figure 5.9 that the number and values of the durations for the common elements are also different between this invalid instance and the valid ones seen in Figure 5.5.

Table 5.1 shows the different computed distances for different constraints that could be encountered in the form of both the distance duration set and the resulting distance value. The case “ $> x$ or $\geq x$ ” is shown for the purpose of highlighting the consideration of the

Algorithm 5.2 Distance calculation between an interval set *intSet* and a duration set *durSet*, taking into account the direction of the constraint *C*, the result being a duration set representing the distance per element

```

1 function distanceSet(durSet, intSet, C)
2   distSet  $\leftarrow$  new duration set
3   forall element in durSet do
4     D  $\leftarrow$  durSet.getDurations(element)
5     I  $\leftarrow$  intSet.getIntervals(element)
6     if isEmpty(I) then
7       if C in [=,  $\neq$ , <,  $\leq$ ] then
8          $\lfloor$  distSet.addAll(D)
9       continue
10    D, I  $\leftarrow$  closestPermutation(D, I)
11    min  $\leftarrow$  minSize(D, I)
12    forall i in 0..min do
13      dist  $\leftarrow$  distance(D, I, C)
14      if dist > 0 then
15         $\lfloor$  distSet.add(element, dist)
16    if C in [=,  $\neq$ , <,  $\leq$ ] then
17      s  $\leftarrow$  size(D)
18       $\lfloor$  distSet.addRange(D, min..s)
19    if C in [=,  $\neq$ , >,  $\geq$ ] then
20      s  $\leftarrow$  size(I)
21       $\lfloor$  distSet.addRange(I, min..s)
22  return distSet

```

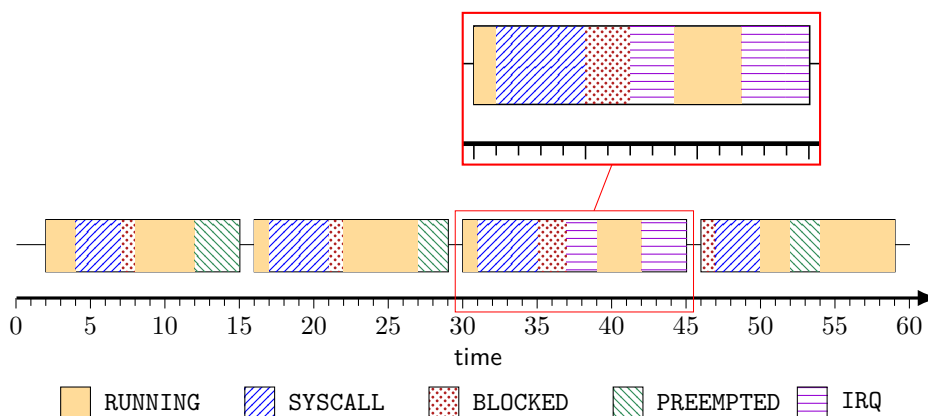


Figure 5.8 Example of a trace containing 4 instances following a model specifying that the instance execution duration should be of 13 units of time; the scope is highlighting the elements of the invalid instance, that differ in number or in duration when compared to valid instances.

```

DurationSet: [
  [RUNNING, 2, 4], // durations 1 and 3
  [SYSCALL, 1, 4],
  [PREEMPTED, 1, 2],
  [IRQ, 2, 5], // durations 2 and 3
]

```

Figure 5.9 The duration set extracted from the invalid instance presented in Figure 5.8; it contains two durations for the element **RUNNING** that can be summed to 4 units of time, one duration for the element **SYSCALL** for 4 units of time, one duration for the element **PREEMPTED** for 2 units of time and two duration for the element **IRQ** that can be summed to 5 units of time

Table 5.1 Computed distance sets and values between the invalid duration set represented in Figure 5.9 and the valid interval set in Figure 5.6 for different constraints on a target value x

Constraint	Distance	
	Duration set	Value
$= x$ or $\neq x$	<pre> DurationSet: [[RUNNING, 1, 1], [BLOCKED, 1, 1], [PREEMPTED, 1, 2], [IRQ, 2, 5], // 2 and 3] </pre>	5
$< x$ or $\leq x$	<pre> DurationSet: [[BLOCKED, 1, 1], [IRQ, 2, 5], // 2 and 3] </pre>	3
$> x$ or $\geq x$	<pre> DurationSet: [[RUNNING, 1, 1], [PREEMPTED, 1, 2],] </pre>	2

directional element, as we would not end up verifying such constraint while having valid instances not validating the constraint themselves. We can see for the different cases that the distance value does not take into account the value of each contained element duration. The distance value is actually the number of durations in the distance duration set, which means that two different distance duration sets could lead to the same distance value. The different computed distance duration sets also show that our algorithms' output adapt correctly to the directional element; only the relevant elements add to the distance.

Weight calculation Once we know the distance between our interval sets, we compute a weight that aims to reflect the chances for the valid interval set to be the one that should have happened instead of our invalid one. We call W_{ri} the relative weight of the valid interval set i for our currently analyzed invalid interval set. To put emphasis on the distance, over the number of occurrences, we chose to use the following formula:

$$W_{ri} = \frac{O_i}{\sum_{d_j \leq d_i} O_j} \times \frac{d_i}{\max(1, s)} + \frac{s - d_i}{\max(1, s)} \quad (5.5)$$

Where O_i is the number of occurrences of the valid interval set i , d_i the distance between this interval set and the invalid one, $\sum_{d_j \leq d_i} O_j$ the sum of occurrences of all the valid interval

sets that have a distance lower or equal to d_i , and s the size – or number of intervals – of the invalid interval set.

However, this formula does not take into account the uncertainty of the results, that grows with the number of different valid interval sets available. We defined the maximum uncertainty as the confidence factor F_C :

$$F_C = 0.1 \quad (5.6)$$

This factor thus represents the maximum penalty that will be subtracted from our weight results. The penalty P is defined by the following formula:

$$P = F_C \times \left(1 - \frac{1}{N_{valid}}\right) \quad (5.7)$$

Where N_{valid} is the number of different valid interval sets that have been found during the analysis. The penalty P will thus be null when there is only one valid interval set found, and tend to F_C while the number of valid interval sets grows. By subtracting that penalty from our relative weight W_{ri} , we calculate the weight W_i :

$$W_i = W_{ri} - P \quad (5.8)$$

If a valid interval set ends up with a weight W_i under 0, this set will simply be dropped. The growing penalty therefore allows to automatically exclude interval sets with a relative weight W_{ri} under the value of P , i.e. with probabilities too low given the number of valid interval sets.

Differential interval set After weighting the valid interval sets against an invalid one, the next step is to identify the discrepancies. For this purpose, we will compute a differential interval set for each valid interval set with a weight greater or equal to 50%. If no such valid interval set is found, the valid interval set with the highest weight will be the only one considered. The local differential interval set is computed by subtracting the valid interval set from the invalid one. This is done by comparing the intervals in both sets for each element, and computing new intervals that embody the difference in terms of number of intervals, and minimum and maximum value, of all the intervals. Algorithm 5.3 gives the pseudocode of the **subtract** function used for this end. We can see that the resulting differential interval set takes into account the directional component of the constraint.

Algorithm 5.3 Subtraction of a valid interval set *validIntSet* from an invalid one *invalidIntSet*, taking into account the direction of the constraint *C*, the result being a differential interval set

```

1 function subtract(invalidIntSet, validIntSet, C)
2   DIntSet  $\leftarrow$  new differential interval set
3   forall element in invalidIntSet do
4     I  $\leftarrow$  invalidIntSet.getIntervals(element)
5     V  $\leftarrow$  validIntSet.getIntervals(element)
6     NI  $\leftarrow$  size(I)
7     minI  $\leftarrow$  sumMinValueOfAll(I)
8     maxI  $\leftarrow$  sumMaxValueOfAll(I)
9     if isEmpty(V) then
10      if C in [=,  $\neq$ , <,  $\leq$ ] then
11        DIntSet.addInterval(NI, minI, maxI)
12      continue
13     ND  $\leftarrow$  NI - size(V)
14     minD  $\leftarrow$  minI - sumMinValueOfAll(V)
15     maxD  $\leftarrow$  maxI - sumMaxValueOfAll(V)
16     if C in [<,  $\leq$ ] then
17       ND  $\leftarrow$  max(0, ND)
18       minD  $\leftarrow$  max(0, minD)
19       maxD  $\leftarrow$  max(0, maxD)
20     else if C in [>,  $\geq$ ] then
21       ND  $\leftarrow$  max(0, -ND)
22       minD  $\leftarrow$  max(0, -minD)
23       maxD  $\leftarrow$  max(0, -maxD)
24     if ND  $\neq$  0 or minD  $\neq$  0 or maxD  $\neq$  0 then
25       minD, maxD  $\leftarrow$  minMax(minD, maxD)
26       DIntSet.addInterval(ND, minD, maxD)
27   if C in [=,  $\neq$ , >,  $\geq$ ] then
28     forall element in validIntSet do
29       if element in DIntSet then
30         continue
31       V  $\leftarrow$  validIntSet.getIntervals(element)
32       NV  $\leftarrow$  - size(V)
33       minV  $\leftarrow$  - sumMinValueOfAll(V)
34       maxV  $\leftarrow$  - sumMaxValueOfAll(V)
35       DIntSet.addInterval(NV, minV, maxV)
36   return DIntSet

```

When there is more than one computed differential interval set, i.e. when we have more than one valid interval set with a weight greater or equal to 50 %, we merge them in order to get only one differential interval set that represents the smallest discrepancies between the invalid interval set and the closest valid interval sets. The merge process is called **interUnion**, as it does both an intersection on the contained elements, and an union on the interval values. The intersection part of the process allows to eliminate elements that could be considered discrepancies while looking at one valid interval set, but are actually present in another one. The union process allows to widen the values of the interval to the smallest interval containing all the differential intervals for the given element.

Finally, we can use the equation (5.4) to identify the responsibility r_e of each element e in our differential interval set. If multiple invalid instances sets were identified, i.e. invalid instances sets with different *keymaps*, the computed responsibilities for each element in each invalid case are merged to form the global responsibilities. For instance, consider that we have two invalid cases. The computed responsibilities for the first case are of 80 % for element a , 15 % for element b and 5 % for element c . However, the responsibilities for the second case are of 50 % for element a , 40 % for element b and 10 % for element d . Therefore, the final merged responsibilities will be of 65 % for element a , 27.5 % for element b , 2.5 % for element c and 5 % for element d .

Diagnostic

For counters and timers, the data extraction process will commonly give the user enough information to understand what happened in the system. However, when working with state system free variables, the origin of the problem is usually not as clear. If we take as example a deadline constraint, once the analysis has been done on the process' state, the underlying problem, when the process is spending too much time being blocked on a system call, or even running, is not necessarily straightforward.

In such cases, other specific analysis can be run, that in turn can run other analysis in order to pinpoint as precisely as possible the origin of the problem. In order to consider only the elements with the highest probabilities for further analysis, an element will be considered only if its responsibility is greater or equal to the minimum responsibility for consideration R_{min} as expressed in (5.9).

$$R_{min} = \max_{e \in r} r_e - \sigma(\forall_{e \in r} r_e) \quad (5.9)$$

Among those analysis, we can count the critical path analysis that allows to retrieve the critical path of the process in the system during a period of time, for which the same data extraction and comparison processes were done. Section 5.6 presents multiple analysis cases and the different analysis that were automatically triggered in order to give a thorough report to the user.

5.6 Case studies

This section presents different case studies of common real-time problems on which we evaluated our automated analysis approach to pinpoint the origin of the problem encountered.

5.6.1 Too low priority

Situation

JACK2 is the C++ version of the JACK Audio Connection Kit sound server. This application is real-time and can run with a periodic workload. When started, JACK2 configures the ALSA driver to sample at a configurable frequency and to accumulate a specific number of frames before raising an interrupt. That whole period can thus be identified and instrumented. Tracepoints were added around the working period of the application, allowing to identify in the trace when JACK2 started to wait to be awakened, and when it actually woke up.

JACK2 already detects when its period is not met, which is called an *xrun* and generates a log message when this situation happens. This report is useful as it provides a way to validate the results we get from analyzing a trace of the running application.

```
$ taskset -c 0 jackd -P 1 -v -d alsa -H
...
creating alsa driver ... hw:0|hw:0|1024|2|48000|0|0|hwmon|swmeter|-|32bit
configuring for 48000Hz, period = 1024 frames (21.3 ms), buffer = 2
    ↪ periods
...
```

Figure 5.10 ALSA driver being configured by the JACK2 daemon process after it was started with a low real-time priority of 1 and pinned on CPU 0 [106]

Taking advantage of the way JACK2 works, we simulated a priority problem as was done in [106]. We first started the JACK2 daemon with a low real-time latency priority, and we pinned it on a given CPU, as shown in Figure 5.10. We can see that the ALSA driver is being configured to sample at 48 kHz and accumulate 1024 frames. This period has an expected

runtime of 21.3 ms. We then ran a CPU-intensive task on CPU 0, with a higher priority, in order to disrupt the period of JACK2. Once that task stopped, JACK2 reported an *xrun*, meaning the application was indeed preempted for a sufficient time to miss its period.

Using the tracepoints reached before and after the period, and the duration of that period, we can easily build an analysis model for that case, as presented in Figure 5.1. The deadline constraint is for a period having a duration of less than 23 ms to take into account the waking time of the process. We then used that model to perform an analysis of a trace during which JACK2 reported an *xrun* of “at least” 353.963 ms.

Automatic analysis

Figure 5.11 presents details of the work performed to identify the source of the problem using a process state analysis. Figure 5.11(a) shows an invalid interval set, which represents the intervals of time taken by different process states during the analyzed invalid instances. In our case, this set contains the data of only one instance, hence the interval having the same duration for both minimum and maximum. Figure 5.11(c) shows the weighted results between the invalid interval set presented in Figure 5.11(a) and the different valid interval sets computed using valid instances. The results show both the distance and occurrences of each valid interval set that were used to compute its weight against our invalid one. Figure 5.11(b) finally shows the computed difference after the weighting step: each valid interval set with a weight greater than or equal to 50 % is subtracted from the invalid one, generating an interval set of the states in excess in the invalid interval set.

The intervals of time added by each process state are then computed as a percentage of the total time added by all of the process states in the difference interval set. Figure 5.12(a) thus shows these results. We can see that the prominent state is *WAKING*, meaning that our application spent 96.65 % of its time waiting to be woken up. A critical path analysis is then triggered by those results. The analysis process is the same, but follows the critical path of the process instead of its own states. Figure 5.12(b) shows the differential results between the critical path of the invalid instances of JACK2 and the valid ones. This analysis shows that the JACK2 daemon process was preempted for 96.51 % of the time. A CPUTop analysis is finally triggered on the period for which JACK2 was identified as preempted, and on the CPU on which this process was running. Figure 5.12(c) shows the results of the CPUTop analysis, on which we can see that process *cpuburn* was the one running on the process while JACK2 was preempted. We can also see in the results that the priority of the *cpuburn* process during that period of time was of -61 (or a real-time priority of 60), while the priority of our process was of -2 (or a real-time priority of 1).

```
IntervalSet(1): [
  [BLOCKED on poll, 3, [4.1705323E7, 4.1705323E7]]
  [RUNNING, 14, [238189.0, 238189.0]]
  [SYSCALL futex, 1, [10094.0, 10094.0]]
  [Interrupted by IRQ29 (snd_hda_intel), 1, [977.0, 977.0]]
  [WAKING, 3, [5.85536417E8, 5.85536417E8]]
  [SYSCALL poll, 6, [58995.0, 58995.0]]
  [SYSCALL ioctl, 5, [103256.0, 103256.0]]
  [SYSCALL write, 5, [31727.0, 31727.0]]
]
```

(a) Set of intervals of time spent in the different process states for the invalid instances of JACK2



```
Diff: [
  [RUNNING, 11, [126560.0, 206248.0]]
  [BLOCKED on poll, 1, [1.8902285E7, 2.1001798E7]]
  [WAKING, 1, [5.85450974E8, 5.8552551E8]]
  [SYSCALL futex, 1, [10094.0, 10094.0]]
  [SYSCALL ioctl, 5, [103256.0, 103256.0]]
  [SYSCALL poll, 2, [0.0, 24049.0]]
  [SYSCALL write, 5, [31727.0, 31727.0]]
  [Interrupted by IRQ29 (snd_hda_intel), 1, [977.0, 977.0]]
]
```

(b) Set of intervals of time considered to be in excess in the invalid interval set compared to the valid interval sets with a weight of at least 50 %

Valid list	Distance	Occurrences	Weight
IntervalSet(105): [[BLOCKED on poll, 2, [2.0703525E7, 2.2803038E7]] [RUNNING, 3, [31941.0, 111629.0]] [WAKING, 2, [10907.0, 85443.0]] [SYSCALL poll, 4, [34946.0, 90288.0]]]	30	105	93.33% <div></div>
IntervalSet(2): [[BLOCKED on poll, 2, [2.0767886E7, 2.17529E7]] [RUNNING, 3, [61252.0, 87261.0]] [WAKING, 2, [15856.0, 23237.0]] [SYSCALL poll, 5, [43815.0, 76711.0]] [Interrupted by HRTIMER, 1, [5848.0, 15496.0]]]	31	2	13.28% <div></div>
IntervalSet(4): [[BLOCKED on poll, 2, [2.0757138E7, 2.1794678E7]] [RUNNING, 4, [46599.0, 115511.0]] [WAKING, 2, [13054.0, 24173.0]] [SYSCALL poll, 4, [32867.0, 61412.0]] [Interrupted by HRTIMER, 1, [7908.0, 12598.0]]]	33	4	9.62% <div></div>



(c) Weighting of the valid interval sets against the invalid one in Figure 5.11(a), showing the number of occurrences of each interval set, its distance to the invalid interval set and its computed weight

Figure 5.11 Algorithm intermediate results for the analysis of JACK2

State	Responsibility for added time
WAKING	96.65% 
BLOCKED on poll	3.29% 
RUNNING	0.03%
SYSCALL ioctl	0.02%
SYSCALL write	0.01%
SYSCALL poll	0.00%
SYSCALL futex	0.00%
Interrupted by IRQ29 (snd_hda_intel)	0.00%


Minimum responsibility for a case to be considered: 64.83%

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process waits to be woken up

Critical path state	Responsibility for added time
jackd PREEMPTED	96.51% 
swapper/0 RUNNING	1.88%
swapper/0 PREEMPTED	1.55% 
jackd RUNNING	0.05%
irq/29-snd_hda_ PREEMPTED	0.01%
irq/29-snd_hda_ RUNNING	0.00%

Minimum responsibility for a case to be considered: 60.79%

(b) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process is preempted

Analyzed CPUs: 0			
Analyzed timerange: [11:54:37.711 896 155, 11:54:38.319 519 477]			
Per-TID Usage	Process	Migrations	Priorities
96.34% 	cpuburn (11371)	0	[-61]
0.29%	bash (11375)	1	[20]
0.13%	bash (11376)	2	[20]
0.11%	bash (11377)	0	[20]
0.10%	/usr/bin/x-term (3154)	2	[]

Priorities of the PREEMPTED process during that interval: -2

(c) CPUPtop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 0, where JACK2 was running, during the period of time for which JACK2 was preempted

Figure 5.12 Sections of the analysis report generated by our analysis for JACK2

needed to release the lock. That is what we can see in Figure 5.13. This process with the raised priority will keep it until it releases the lock on the resource, then its old priority is restored. In the case shown in the Figure, this means that **lowprio1** will again be preempted by **highprio1**, but this time without holding the resource. With the lock released, **highprio0** will be able to do its work with the resource.

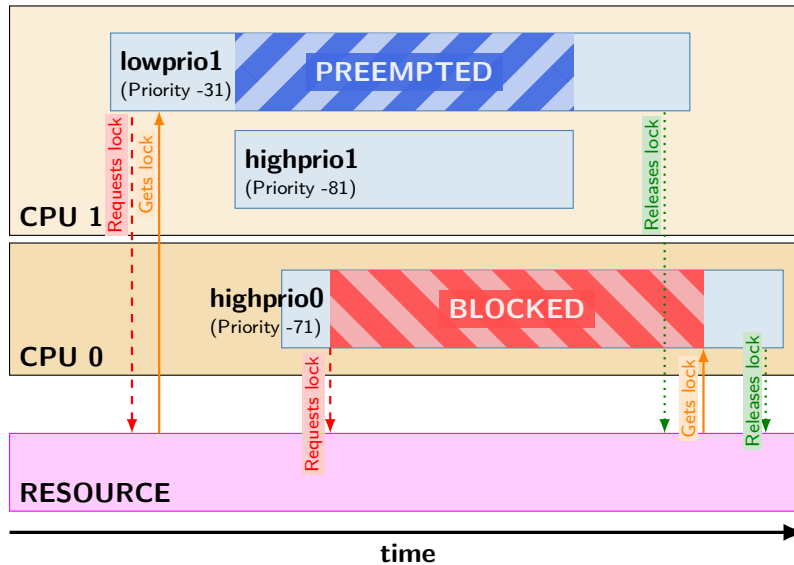


Figure 5.14 Schema of an in-kernel wake lock priority inversion: the **lowprio1** process takes the lock of the resource and cannot free it for the higher-priority **highprio0** process because **lowprio1** is currently preempted by the **highprio1** process, which has a higher priority than **highprio0**

Figure 5.14 shows the schematization of an in-kernel wake lock priority inversion situation. The difference with Figure 5.13 is the priority of the **highprio0** process, which is now “70” instead of “90”, and thus lower than the priority of **highprio1**. Therefore, in this case, the process preempting the low priority one has a higher priority than the high priority process waiting for the resource. We thus end up in a situation where a low priority process prevents a high priority process from running by holding the lock on a needed resource.

A kernel module and small application have been created to simulate this problem. The kernel module creates a file that can be opened only once at a time using a **rtmutex**. The application starts the multiple threads presented in Figures 5.14 and 5.13. We ran that application with different setups in order to trace both working and non-working scenarios and analyze the generated traces automatically. These setups are presented in Table 5.2.

Table 5.2 Real-time priorities of the processes in the setup used to generate the trace for the in-kernel wake lock priority inversion. An empty priority means that the process was not started.

Process	Real-time priority			
	Setup 1	Setup 2	Setup 3	Setup 4
lowprio1		30	30	30
highprio1			80	80
highprio0	70	70	70	90

Automatic analysis

The results of the automatic analysis are presented in Figure 5.15.


Figure 5.15(a) shows the results of the state analysis of the application simulating an in-kernel wake lock priority inversion. We can see in those results that, for the invalid instances, the process spends most of its excess time being blocked on an **open** system call. This thus triggers both the priority inheritance analysis and the critical path analysis, for which results are respectively presented in Figure 5.15(b) and Figure 5.15(c). The former shows that, in invalid instances, the priority inheritance stays active for an abnormal amount of time compared to valid instances, resulting in a verdict of very high probability of a priority inversion. The latter shows that the added time in invalid instances was spent waiting for the **lowprio1** process which was preempted. The CPUTop analysis is then triggered by these last results. As we can see in Figure 5.15(d), the CPUTop analysis shows that the **lowprio1** process was preempted by the **highprio1** process, which has a greater priority.

These analysis results thus allow to directly identify the in-kernel wake lock priority inversion problem, and its origin.

5.6.3 Bad userspace code

Situation

When working on high performance applications, all the algorithms are important, as a poorly written source code can induce latency. In some cases, however, a portion of code that should not be run frequently is thus not optimized. In such a situation, the cause of a missed deadline is not external to our task, preventing us from analyzing what is happening

State	Responsibility for added time
BLOCKED on open	100.00% 
RUNNING	0.00%
SYSCALL open	0.00%
SYSCALL gettid	0.00%


Minimum responsibility for a case to be considered: 56.70%

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on an `open` system call

```
Duration of active 'sched_pi_setprio'...
... in valid instances (maximum): 170.909us
... in invalid instances (average): 329.534us

Active 92.81% more time in invalid instances than in valid instances.
Verdict: Very high probability of a priority inversion
```


(b) Priority inheritance analysis triggered by the state analysis, comparing the average time where the priority inheritance was active in invalid instances to the maximum time in valid instances

Critical path state	Responsibility for added time
lowprio1 PREEMPTED	100.00% 

Minimum responsibility for a case to be considered: 100.00%

(c) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process waits after the `lowprio1` process, which is preempted

Analyzed CPUs: 1
Analyzed timerange: [21:40:35.867 825 012, 21:40:36.033 045 371]

Per-TID Usage	Process	Migrations	Priorities
99.18% 	highprio1 (26408)	0	[-100, -81]
0.82%	lowprio1 (26407)	0	[-71]
0.00%	ltnng-consumerd (26369)	0	[20]
0.00%	ltnng-consumerd (26370)	0	[20]
0.00%	Cache2 I/O (3011)	0	[20]

Priorities of the PREEMPTED process during that interval: -71

(d) CPUPtop analysis triggered by the critical path analysis, showing the processes that were running on the CPU 1, where `lowprio1` was running, during the period of time for which `lowprio1` was preempted

Figure 5.15 Sections of the analysis report generated by our analysis for the in-kernel wake lock priority inversion

outside of the studied task. However, it is still possible to give an helpful insight into what is happening in the application, thanks to the model.

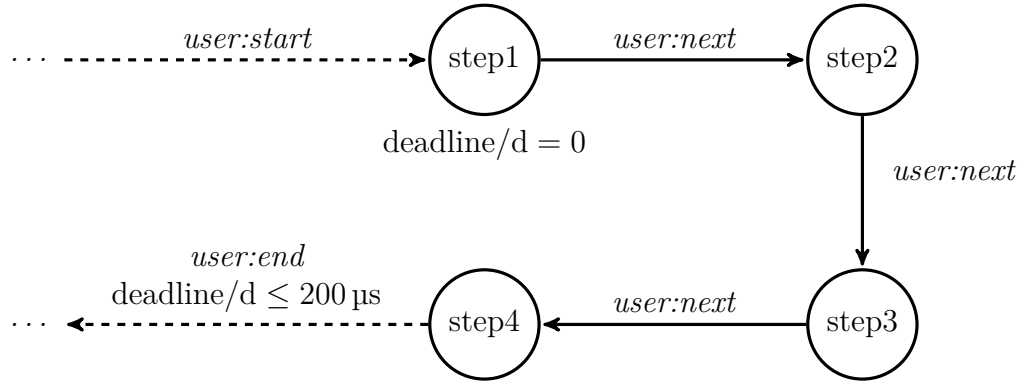



Figure 5.16 Graphical representation of the period within our userspace application

Figure 5.16 shows the model of an application that has been built to have such a problem. In this application, we know that the global time taken between the entry in state “step1” and the exit of step “step4” should be less than 200 μ s. In the application, in between each state of the model, we do a busy loop. Though, using a random number generator, we double the number of loop iterations executed by the application while in the “step3” state, once in a while, making the application miss its deadline. Using multiple runs of that application, we were able to generate a trace with working and non-working instances to analyze with our automatic analysis. This closely resembles an actual problem encountered in an industrial real-time application.


Automatic analysis

The results of the automatic analysis are presented in Figure 5.17.

Figure 5.17(a) presents the results of the state analysis of our userspace application where invalid instances are due to a busy loop. We can see that the state analysis highlights the running state of the application. This means that the excess time spent in invalid instances was while the application was running. The state analysis results thus trigger a state machine state analysis, as shown in Figure 5.17(b). This analysis compares the time spent in each machine state for valid and invalid instances. Such analysis allows to pinpoint the place in the source code where the excess time is spent for invalid instances. In this case, we can see that the application was spending most of its excess time in the “step3” state.

State	Responsibility for added time
RUNNING	99.89% 
Interrupted by HRTIMER	0.04%
Interrupted by SOFTIRQ_TIMER	0.03%
Interrupted by SOFTIRQ_RCU	0.02%
SYSCALL write	0.02%
Interrupted by SOFTIRQ_SCHED	0.00%
SYSCALL gettid	0.00%
Minimum responsibility for a case to be considered: 64.94%	

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly running

Model state	Responsibility for added time
step3	99.92% 
step1	0.05%
step4	0.03%
step2	0.01%

(b) State machine state analysis triggered by the state analysis, showing the responsibility of each state of the model for the added time in the invalid instances: in this case, the process spends most of its time in the “step3” state

Figure 5.17 Sections of the analysis report generated by our analysis for the bad userspace code

These analysis results thus allow to recognize that the problem in this case was linked to the way the application was written, and to get a fix on where in the code to look for correcting the problem.

5.6.4 Frequency scaling

Situation

With the advent of real-time systems, general purpose computers are often able to run soft real-time applications. However, the frequency of processors is usually configured to scale with the work intensity, to optimise power consumption.

The frequency of a running application can also change when there is a CPU migration and the two processors do not have the same frequency. Some embedded architectures rely on this same principle, such as the **big.LITTLE** architecture, which takes advantage of using simultaneously a very low-power processor (*LITTLE*) as the main microprocessor, and several higher-power processors (*big*) activated when there is a need for high computing power.

These situations can thus lead to instances of the modeled application taking more or less time throughout the trace, even without any internal or external interference apart from the CPU frequency change. In such cases, analysis results can be altered from what we would expect, and might be useless.

Automatic analysis

While working on the different case studies, we encountered this problem. We thus worked on a frequency scaling analysis to include at the beginning of the automatic generated report.

Figure 5.18 shows the results of such analysis, in a case where there was a frequency scaling problem while a deadline constraint was verified. We can see, in the analysis, the two histograms showing the different frequencies while valid and invalid instances were running. An average frequency is then computed for each situation, and those averages are finally compared. In this case, the average CPU frequency was 22.81 % higher in valid instances than in invalid instances, leading to a verdict of probability of a frequency scaling problem.

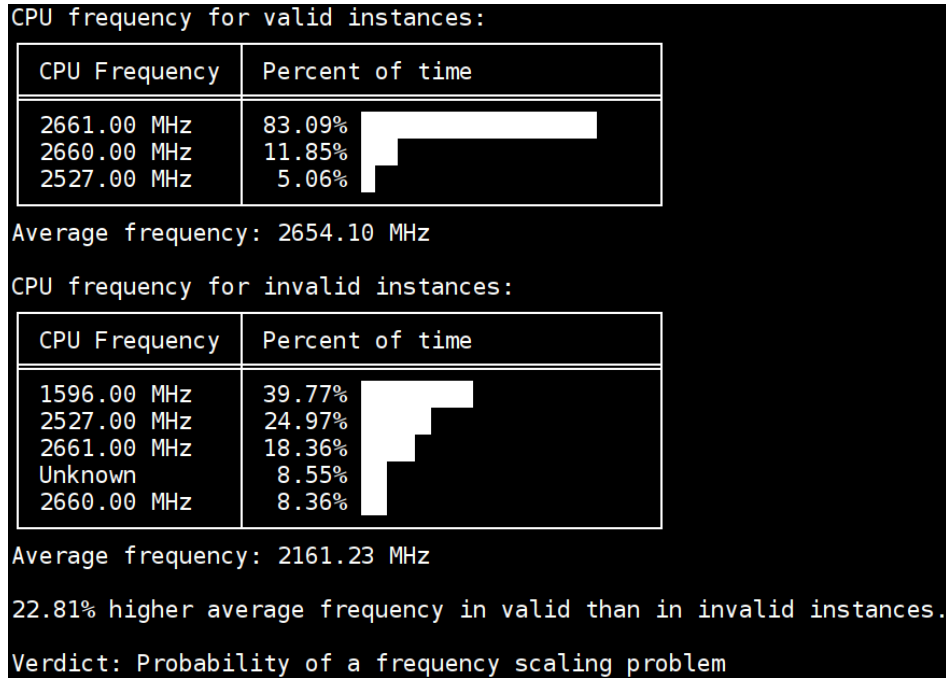


Figure 5.18 Frequency scaling analysis, showing the CPU frequencies for valid and invalid instances, and comparing their average frequencies in order to identify if there is a probability of a frequency scaling problem

5.6.5 Preempted waker

Situation

The `cyclictest` tool allows to verify the software real-time performance of a system. It does so by executing a periodic task with multiple processes. Each process runs on a different CPU, and each task can be set with a different period. The aim of the task is to be woken up during the given period. The performance is then evaluated by measuring the discrepancy between the desired period and the real wake up time.

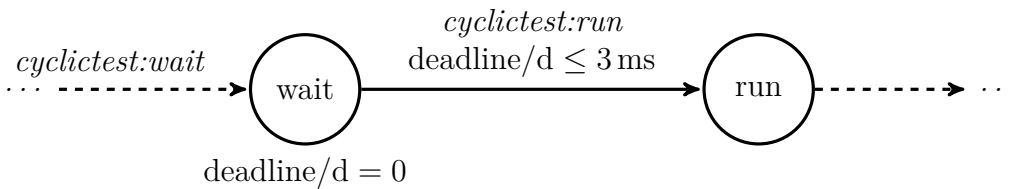


Figure 5.19 Graphical representation of the period of `cyclictest`

We instrumented `cyclictest` to have tracepoints around its periodic task, which leads to the model representation in Figure 5.19. We then ran it with a task on each CPU, a real-

time priority of 99, and the `SCHED_FIFO` scheduler. The experience was done on an NVIDIA Jetson TK1 while tracing in snapshot mode with `LTTng`. The snapshot mode allows to write the trace buffers to disk as soon as a specific event, high latency in our case, is detected by the application.

At some point, the tool reported an abnormal latency of 7 ms, and the tracer saved the snapshot of the trace buffers to analyze. Without any insight on the origin of the problem, we used our automatic analysis to understand what happened.

Automatic analysis


The results of the automatic analysis of the `cyclictest` run are presented in Figure 5.20.

Figure 5.20(a) presents the results of the state analysis for our `cyclictest` execution. We can see that the `cyclictest` instance that missed the deadline was blocked on a `rt_sigtimedwait` system call. These results trigger a critical path analysis that is shown in Figure 5.20(b). This analysis puts forward that in invalid instances, the `cyclictest` process spent time waiting to be woken up by the `ktimersoftd/3` process, which was preempted. The CPUtop analysis is finally triggered by the results of the critical path analysis. As we can see in Figure 5.20(c), while the `ktimersoftd/3` process was preempted, the `irq/154-hpd` process was running on the CPU. The preempted process had a real-time priority of 1, while the interrupting process had a real-time priority of 50.


The `irq/154-hpd` process corresponds to the interrupt handler for the hot plug detect feature. This feature is a communication mechanism between two devices allowing to be aware when one is connected to or disconnected from the other [107]. This mechanism is supported by the HDMI protocol. Knowing that, we were able to identify that the latency was caused when we plugged an HDMI screen to the NVIDIA Jetson TK1 while trying to see the current `cyclictest` results.

5.7 Running time and scalability


This section presents the timing results of our analysis as well as its scalability according to different factors. All the tests presented were executed on a test system that consists of an Intel® Core™ i7-4790 CPU at 3.6 GHz, with 32 GiB of DDR3 RAM at 1600 MHz. Hyperthreading was disabled, as well as the idling and turbo modes of the CPU.

State	Responsibility for added time
BLOCKED on rt_sigtimedwait	99.95% 
RUNNING	0.03%
SYSCALL getcpu	0.01%
SYSCALL clock_gettime	0.01%
Minimum responsibility for a case to be considered: 56.68%	

(a) State analysis showing the computed responsibility of process states for the added time in the invalid instances: in this case, the process is mostly blocked on the `rt_sigtimedwait` system call

Critical path state	Responsibility for added time
ktimersoftd/3 PREEMPTED	99.68% 
ktimersoftd/3 RUNNING	0.24%
cyclicttest PREEMPTED	0.08%
Minimum responsibility for a case to be considered: 52.77%	

(b) Critical path analysis triggered by the state analysis, showing the responsibility of steps of the critical path of the application for the added time in the invalid instances: in this case, the process mostly waits after the `ktimersoftd/3` process, which is preempted

Analyzed CPUs: 3			
Analyzed timerange: [15:55:53.528 529 561, 15:55:53.536 012 105]			
Per-TID Usage	Process	Migrations	Priorities
99.73% 	irq/154-hpd (162)	0	[-51]
0.26%	irq/25-54200000 (163)	0	[-51]
0.01%	ktimersoftd/3 (32)	0	[-2]
0.00%	irq/22-Tegra PC (160)	0	[-51]
0.00%	irq/388-eth0 (611)	0	[-51]
Priorities of the PREEMPTED process during that interval: -2			

(c) CPUTop analysis triggered by the critical path analysis, showing the processes that were running on CPU 3, during the period of time for which `ktimersoftd/3` was preempted

Figure 5.20 Sections of the analysis report generated by our analysis for `cyclicttest`

Table 5.3 Number of events and sizes of the traces used to measure our analysis running time

Case	Number of events		Size (MiB)
	UST	Kernel	
jackd	321	419 164	3.144 25.09
wakelock	42	194 997	1.605 11.92
userspace	29	89 840	0.6353 5.456
cyclicttest	41 677	208 489	12.90 21.63

5.7.1 Running time

To present the running time of our analysis, we use the different cases presented in 5.6. Table 5.3 presents the information about the traces that were used to analyze our different cases. We can see that all those traces are different in terms of duration and number of events.

Table 5.4 Results of the instances duration analysis for the traces of the cases studied, which computes information about the similar instances steps in the valid and invalid instances

		Instances Steps Execution Duration (in ms)			
Instances	Count	Minimum	Average	Maximum	St. Dev.
jackd					
<i>Valid</i>	112	20.857	21.324	22.062	0.427
<i>Invalid</i>	1	627.685	627.685	627.685	0.000
wakelock					
<i>Valid</i>	3	0.025	113.541	170.952	80.270
<i>Invalid</i>	1	329.575	329.575	329.575	0.000
userspace					
<i>Valid</i>	2	0.070	0.074	0.078	0.004
<i>Invalid</i>	1	21.463	21.463	21.463	0.000
cyclictest					
<i>Valid</i>	20 834	0.830	1.734	2.686	0.559
<i>Invalid</i>	1	7.562	7.562	7.562	0.000

Table 5.4 reports the results of the instance duration analysis performed when working on a deadline constraint. It allows to identify the variation in the duration of the instances of the application, as well as to have an idea on the number of valid and invalid instances in the trace that will be analyzed.

Table 5.5 shows the process state analysis execution duration and results with and without sampling. Only the **JACK2** and **cyclictest** cases are presented here as they are the only ones of the four cases taking advantage of sampling, as the sample size starts to be smaller than the list size when reaching a cardinality of 21. We can thus observe that the number of analyzed instances is very different when sampling is used. While looking at both Tables 5.4 and 5.5, we can see that the number of analyzed instances for **cyclictest** is different than the total number of instances found in the trace. This is justified by the fact that **cyclictest**

Table 5.5 Statistical comparison between the use of sampling or not for the process state analysis, in terms of number of instances analyzed, analysis execution duration and result (percentage of responsibility for the highest responsible state for added time); the highest responsible state was, in all cases, identical within 0.1% whether sampling was used or not; statistics were computed with 100 runs of the analysis for each situation.

Case	# instances analyzed	Process state analysis			
		Execution duration (ms)		Most responsible (%)	
jackd					
<i>Without sampling</i>	113	23 930	$\pm 1\,552$	96.65	$\pm 0.$
<i>With sampling</i>	88	15 310	$\pm 1\,038$	96.65	$\pm 0.002\,222$
<i>Difference</i>	-25	-8 615		-0.000 590 7	
cyclctest					
<i>Without sampling</i>	4 191	103 100	$\pm 4\,609$	99.99	$\pm 0.$
<i>With sampling</i>	379	7 125	± 432.1	99.91	$\pm 0.063\,78$
<i>Difference</i>	-3 812	-95 990		-0.077 95	

Note: The number of analyzed instances can be different than the total number of instances, even when sampling is disabled, as instances steps are dropped when data is missing.

was traced in snapshot mode. In this mode, when buffers are full, the oldest data is dropped in favor of the new one. This means that data can be missing to analyze instances, hence the analysis dropping these.

We can also see in Table 5.5 that while the analysis execution duration is drastically reduced, the results are highly similar, still leading to the right conclusions. Using sampling in our analysis can thus be considered as an approach to retain, as it allows to process a much reduced but sufficient number of instances while keeping a good precision.

Table 5.6 presents the benchmark of the different steps of the analysis, per step of the analysis and per case analyzed.

We can see for instance that the *Instances duration* analysis, which goes through all the instances in the trace, is taking much more time for the `cyclictest` case than for the three others. This is easily explained by the number of instances in this case versus the others. This is also the situation for the *CPU Frequency* and the *Priority inheritance* analysis.

The *Process state* analysis is the initial analysis performed with our algorithms. We can observe that this analysis is taking more time for the `jackd` case than for `cyclictest`, despite the large number of instances. This is justified by the fact that `jackd` instances, even valid ones, are taking much more time than `cyclictest` ones, as we can see in Table 5.4. The main influence of the instance duration when performing a *Process state* analysis is that when instances take more time, more state changes will happen, leading to more data to analyze.

The *Critical path* analysis as well suffers partly of the instance duration, but is less influenced by the number of instances as the closest instances that are used for comparison have been identified during the *Process state* analysis. This means that this analysis is mostly impacted by the duration of the invalid instances.

The *CPUTop* analysis is an external python analysis part of the `l1tng-analyses` python scripts. Its duration is mostly limited by the period of time identified for further analysis by the *Critical path* analysis. It however takes time as it has to read the trace from the beginning until it reaches the period of time to be analyzed. This is thus highly dependent on the number of events and the position of the period to analyze in the trace.

This last statement can be extended to all the steps of the analysis, as the more events there is in the trace, the more events there is to read to reach the ones that are of importance to the analysis.

Table 5.6 Statistics on the execution duration of the different parts of our analysis process, computed with 100 runs of the analysis for each case studied in 5.6

Analysis	Analysis execution duration (in ms)			
	Minimum	Average	Maximum	St. Dev.
Instances duration				
<i>jackd</i>	0.833 6	0.947 7	1.228	0.067 26
<i>wakelock</i>	0.613 2	1.374	8.734	1.497
<i>userspace</i>	0.628 9	1.609	6.917	1.311
<i>cyclictest</i>	24.59	25.51	35.18	1.099
CPU Frequency				
<i>jackd</i>	21.32	24.67	29.10	1.372
<i>wakelock</i>	4.966	11.90	34.50	6.727
<i>userspace</i>	5.107	11.99	37.03	5.944
<i>cyclictest</i>	14 810	15 850	17 130	431.8
Process state				
<i>jackd</i>	12 740	15 310	18 380	1 038
<i>wakelock</i>	264.6	402.1	591.0	82.76
<i>userspace</i>	185.2	250.7	358.3	29.10
<i>cyclictest</i>	6 068	7 125	8 389	432.1
State machine state				
<i>userspace</i>	0.484 7	0.680 0	3.799	0.399 8
Priority inheritance				
<i>wakelock</i>	0.690 9	0.990 7	4.629	0.591 7
<i>cyclictest</i>	8 360	8 725	8 960	135.0
Critical path				
<i>jackd</i>	1 092	1 289	1 632	101.7
<i>wakelock</i>	758.5	990.8	1 163	96.49
<i>cyclictest</i>	155.1	520.8	656.8	107.1
CPUTop				
<i>jackd</i>	4 919	5 020	5 322	59.38
<i>wakelock</i>	2 500	2 558	2 667	29.97
<i>cyclictest</i>	7 980	8 201	8 398	84.29

5.7.2 Scalability

In order to test and verify the scalability of our approach according the different points raised in 5.7.1, we created a simple program performing a parametrable number of *chmod* system calls. We ran and traced the execution of that program while varying different parameters such as the number of instances or the number of system calls performed for a valid or an invalid instance. In each case, only one invalid instance was created to limit the variation of the experience parameters. We then ran our analysis on these multiple traces in order to generate the graphs presented in Figures 5.21 and 5.23. These graphs reflect the evolution of the analysis duration for full and partial analysis, with more or less events enabled while tracing the application.

Figure 5.21 shows the evolution of our analysis duration according to the number of instances in the trace.

We can see in the graph that, for a partial analysis, the analysis time is relatively constant. This can be explained by the fact that, when performing a partial analysis, only the invalid instances are considered, and only one invalid instance was created when running the program.

The full analysis is following a different pattern where the duration of the analysis is proportional to the number of instances in the trace. That can be explained by the number of instances to analyze in order to extract the right information, which is increasing with the number of instances available in the trace. This tendency is shown in Figure 5.22, which represents the size of the sample to analyze according to the number of instances in the traces used for these tests. We can see however that, even if the sample size stabilizes, the full analysis duration continues to increase with the number of instances to analyze. This is also explained by the number of events to read in order to reach the instances to analyze.

Figure 5.23 shows the evolution of our analysis duration according to the number of variable changes during an instance. The traces used contained only two instances, one of which was invalid. We can see that the more variable value changes we have for an instance, the more time the analysis will take. This is easily explained by the fact that for each variable value change, an event has to be extracted from the trace to perform the analysis. Moreover, the higher the number of system calls generated by the invalid instance, the longer the instance will take, and the higher the number of external events in the trace will be, also adding to the analysis time when searching for the specific event that provoked a change to the variable value.

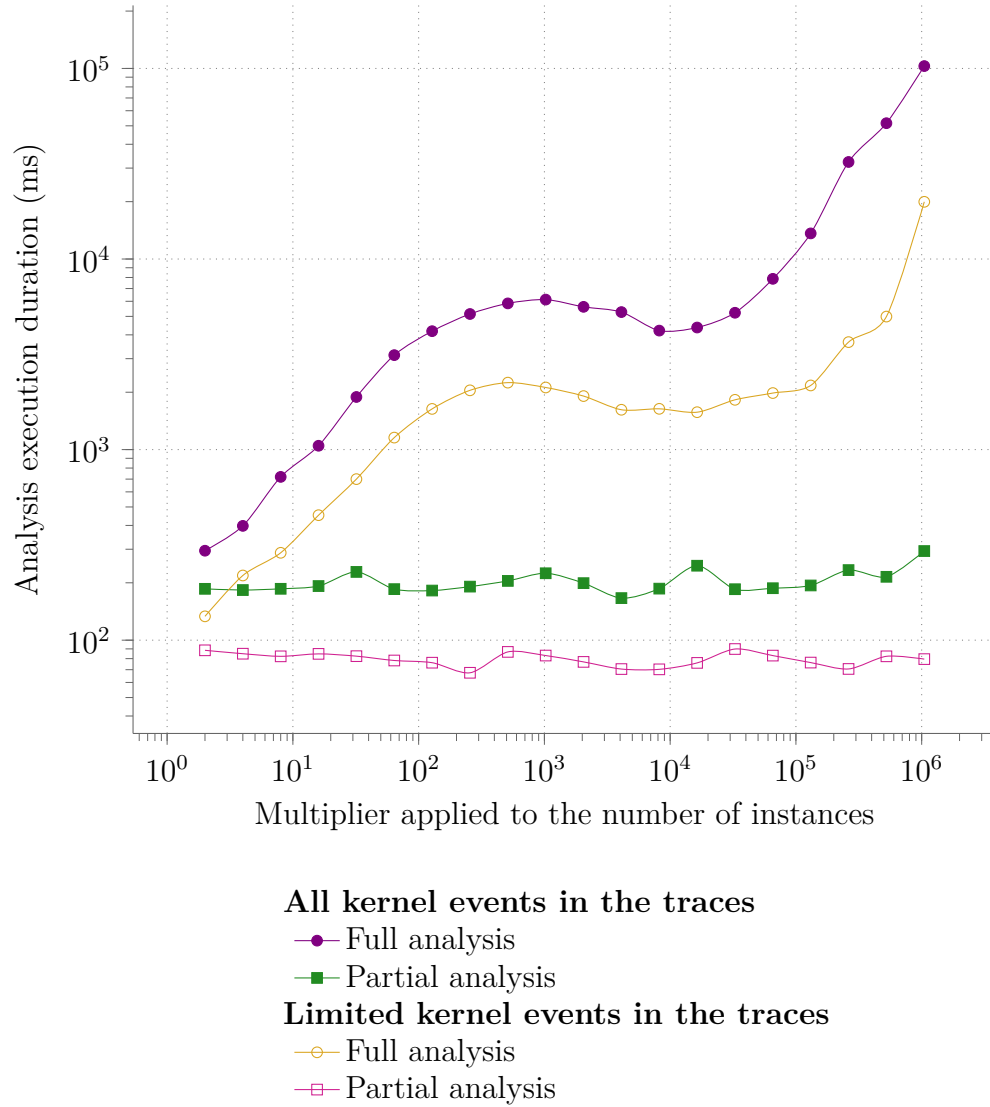


Figure 5.21 Execution duration of the initial variable analysis according to the number of instances in the trace, when only one of these instances is invalid; the original number of instances is 2; each data point is the average of 20 runs

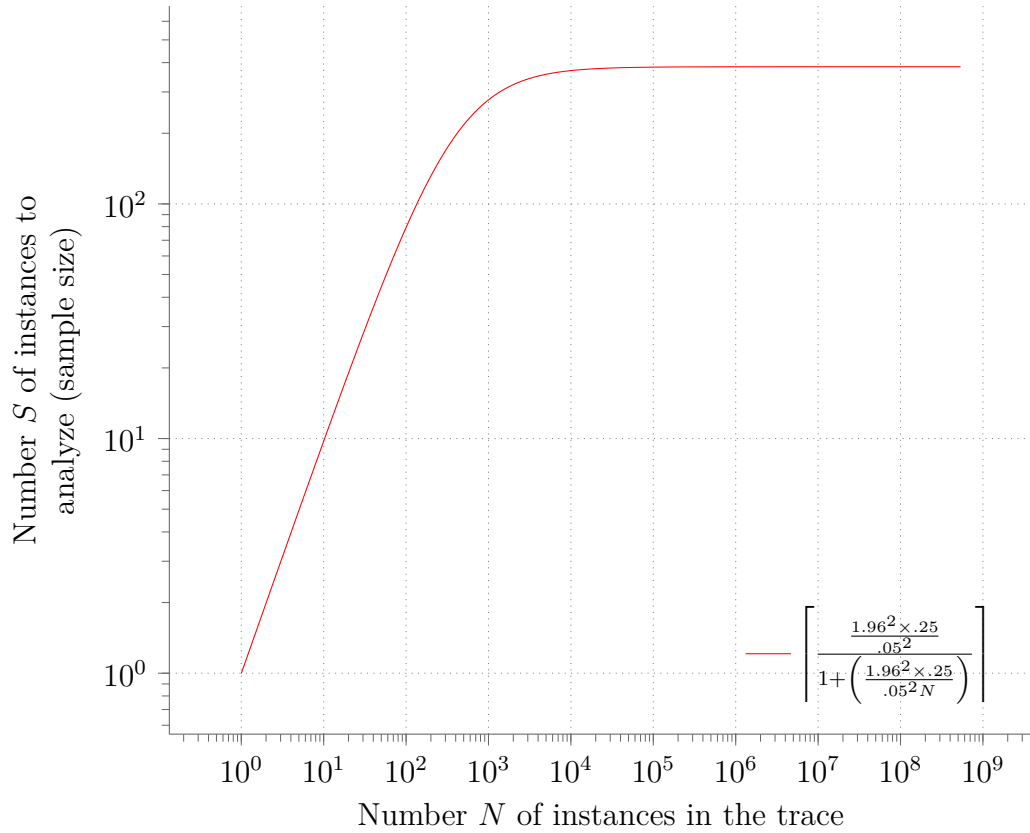


Figure 5.22 Number of instances to analyze according to the number of valid or invalid instances in the trace, using (5.1) with $p = 50\%$, $z = 1.96$ and $e = 5\%$

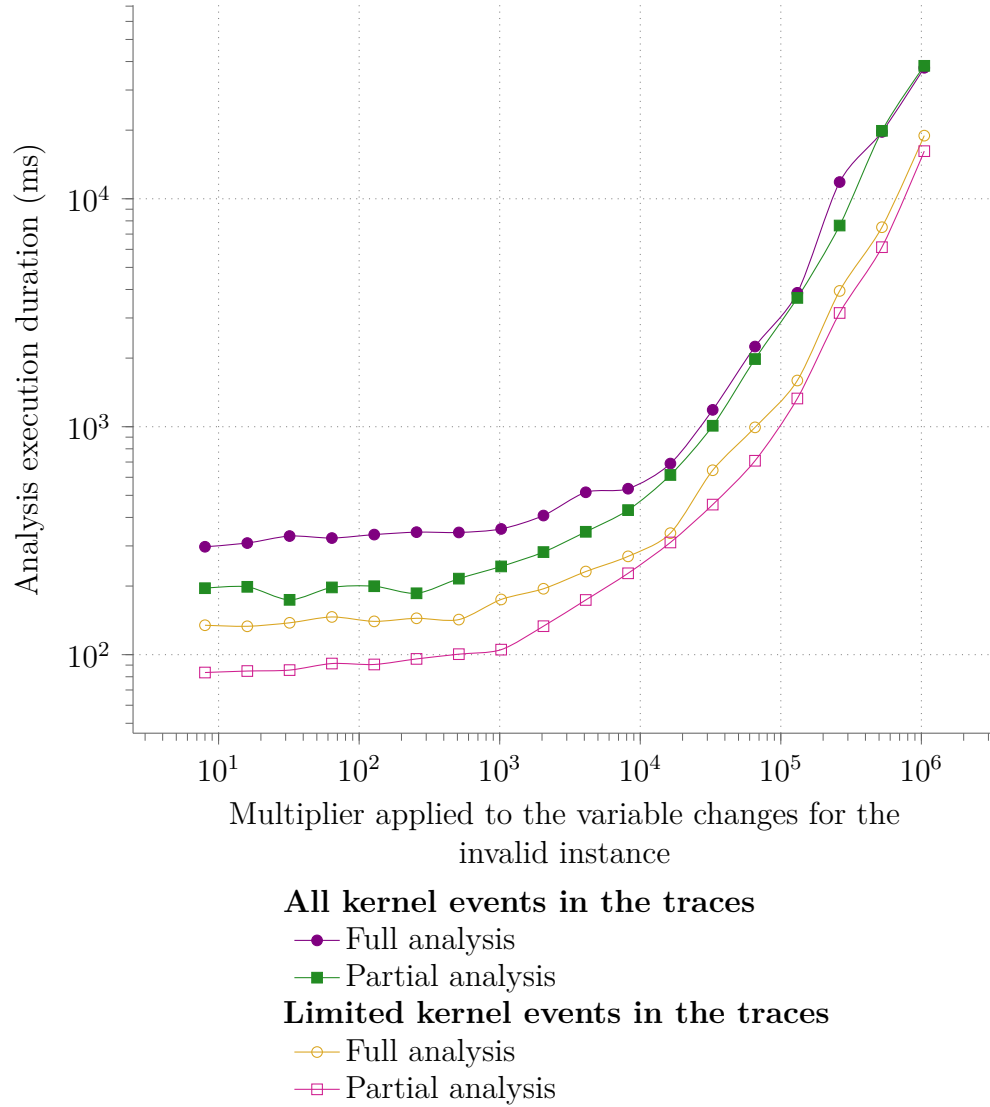
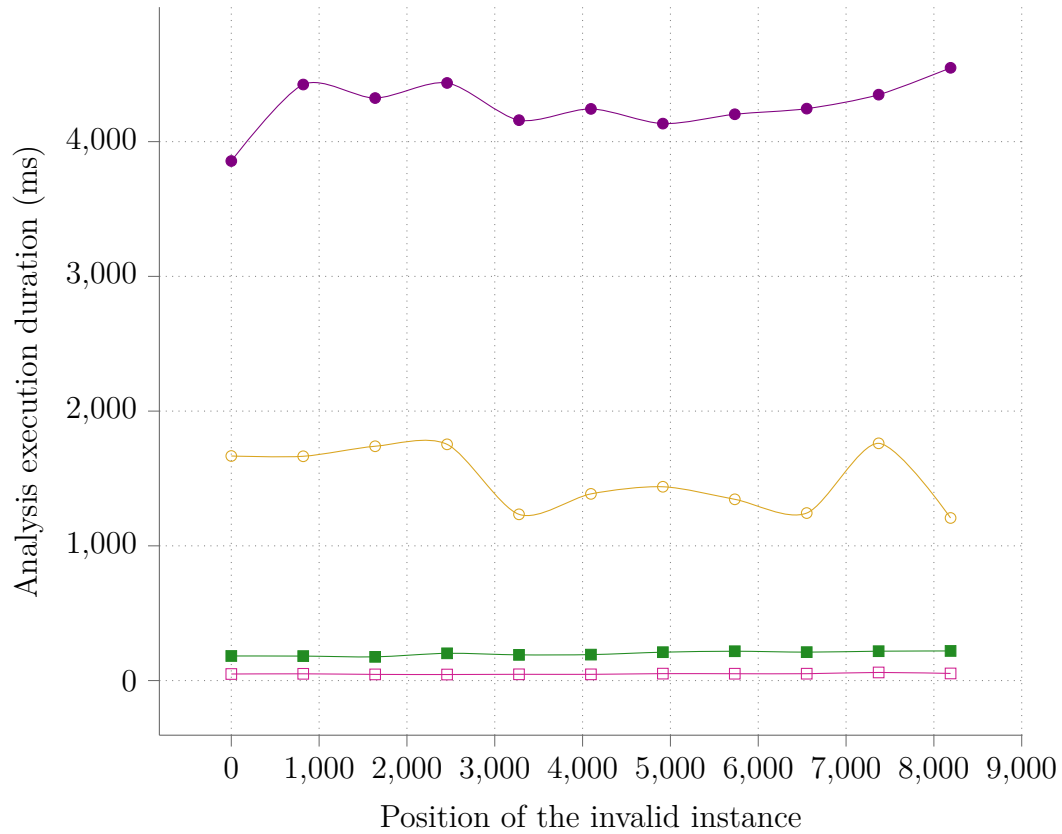


Figure 5.23 Execution duration of the initial variable analysis according to the number of variable value changes during an instance; the number of variable value changes for a valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs



All kernel events in the traces

- Full analysis
- Partial analysis

Limited kernel events in the traces

- Full analysis
- Partial analysis

Figure 5.24 Execution duration of the initial variable analysis according to the position of the invalid instance among the 8192 instances in the trace; the number of variable value changes for a valid instance is 5 for the traces leading to a full analysis, and 0 for the partial analysis; each data point is the average of 20 runs

Figure 5.24 shows the influence of the position of the instances on the analysis duration. We can see that for the partial analysis, the analysis duration is always the same and does not depend on the position of the invalid instance in the trace. This is due to the fact that only the invalid instance will be analyzed. Therefore, once we reach the position of the instance in the trace, we will not have to go back in time in the trace for another one. However, this is not the case for the full analysis, where we also have to analyze some of the valid instances. This is why we can see variations in the duration of the full analysis.

Finally, we can see in Figures 5.21, 5.23, and 5.24 that if the number of events in the trace does not impact the global tendency of the analysis duration, it still impacts its value. We can thus observe that when there are fewer events in the trace, it takes less time to perform the analysis.

All those tests confirm that our analysis can be executed in time proportional to the number of events in the trace, the number of instances of the application, and the number of variable value changes during an instance. However, the position of the instances in the trace has no direct impact on the duration of the analysis.

5.8 Conclusion and Future Work

We proposed a new approach to use model-based constraints and kernel and user-space traces to automatically analyze the origin of unexpected behaviors in real-time and multi-core applications. We presented a brief overview of the model representation used to follow the workflow of the application in the trace, part of previous work. We then detailed the organization and extraction of the interesting data for a given invalidated constraint. This extraction is followed by an assignation of responsibility that provides the user with a list of potential causes for the constraint violation. Depending on the type of cause identified, further analysis can be automatically started to obtain more precise information for the user. We applied our new approach to common real-time and multi-core problems, and provided examples of analysis reports that helped us to pinpoint the cause of encountered problems. We finally characterized the execution time as well as the scalability of our implementation, as a function of the number of instances in the trace, the duration of the instances, the position of the instance and the number of events in the trace.

This approach is novel and an important step to automate performance analysis and problem detection, in order to save time and make tracing more accessible to people without a deep system knowledge. We intend to pursue our work in order to propose an approach to automatically set values to the constraints when analyzing a trace.

5.9 Acknowledgments

This research is supported by Ericsson, EfficiOS, and the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant CRD-468687-14.

CHAPTER 6 ARTICLE 3: MODEL-BASED CONSTRAINTS INFERENCE FROM RUNTIME TRACES OF COMMON MULTICORE AND REAL-TIME APPLICATIONS

Authors

Raphaël Beamonte
Polytechnique Montréal
raphael.beamonte@polymtl.ca

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

Submitted to ACM Transactions on Software Engineering and Methodology, September 26th, 2016

Reference R. Beamonte and M. R. Dagenais, “Model-based constraints inference from runtime traces of common multicore and real-time applications,” *ACM Transactions on Software Engineering and Methodology*, to appear: submitted.

6.1 Abstract

Analyzing the runtime of a real-time application is particularly difficult with interferences from concurrently running processes. Low-overhead tracing is usually the most reliable tool to understand and check the behavior of such applications. In previous work, the automatic detection of common real-time problems was proposed, using models and constraints over the behavior of the application and operating system. Such a model automates system verification, alleviating the need for a thorough and deep understanding of all the system internals, and reduces drastically the time needed to find root causes for problems. Nevertheless, coming up with a model is not trivial.

In this paper, we present a way to automate building a model of the process with constraints, based on user-space and kernel execution traces. Recurrent event sequences are used to build an approximate model of the behavior, and typical timings are used for setting up tentative constraints. The resulting model can then be refined as needed through user intervention. Our algorithms and their scalability have been tested and the experimental results show that our approach allows to build a model to automatically detect common problems in applications, with a relatively modest analysis cost.

Keywords Linux, Real-time systems, Performance analysis, Tracing, Modeling, Automatic model generation

6.2 Introduction

Investigating issues in multicore, parallel and real-time applications is a complicated process. In order to help developers, different system analysis tools exist, such as tracers, profilers and debuggers. Tracers are of particular interest for such applications since they do not stop their workflow, but still provide a significant amount of information on their runtime, as well as the system status during the traced time interval. Each tracer has its specific characteristics, such as its overhead, its precision level, and its ability to trace the kernel, the user-space or even both at the same time. However, the generated traces usually contain considerable and detailed information, needing significant human intervention for its analysis. Furthermore, to understand such information, a sufficient knowledge is necessary.

Modeling allows both technical and non-technical users to easily define the workflow of an application, as well as the quantitative constraints that it must satisfy. Moreover, it is commonly used for formal verification in the real-time community [71]. By using models along with tracing, it is possible to check specifications against the actual runtime execution of the application. This validation is similar to the runtime validation of a protocol. The aim is to check that when each event appears, the specified constraints are met. Using the execution traces and a model of the application behavior, we can verify the application execution, checking the generated trace events, while considering the influence of other running applications. However, in order to perform such analysis, it is necessary to first build at least a simple model of the application behavior with the right specifications. When working on real-time applications, the model may be something as simple as a basic real-time loop with a beginning and end, with a deadline, followed by a wait until the next iteration. The present work focuses on automating the construction process of such models by characterizing the data extracted from the trace.

This paper therefore describes a new approach to use kernel and user-space traces to build a behavioral model and define its constraints. Such a model can then be used to automatically detect unwanted behavior in applications and run further analysis. We then demonstrate this approach using traces of common problems encountered in real-time and multi-core applications. The automated model generation saves time by building an initial model of the traced application and setting constraints following user-defined settings. The process to detect unwanted behavior in the application, presented in previous work [4], can then use the generated model in order to identify and analyze problems in the application's workflow.

Our main contribution is to automatically generate an approximate model and associated constraints for an application. These are identified using both the user-space events, to understand the workflow of the application, and system-side events to identify some of the constraints to be used.

The related work is presented in section 6.3. A short reminder of the model-based constraints approach is in section 6.4. Section 6.5 explains how we identify the workflow of the application and build it as a model with constraints. We then explain our algorithms to compute the operators and values for constraints using the trace content in section 6.6. Section 6.7 gives an overview of the user interface. We evaluated our proposed approach on a set of common real-time cases in section 6.8. Results, computation time and scalability are shown in section 6.9. Suggestions for future work and the conclusion are in section 6.10.

6.3 Related work

This section presents related work in the areas relevant for this paper, software tracers for Linux that can trace both the user-space and the kernel, tools for trace analysis using model-based constraints and data extraction, and finally different model generation and pattern detection methods.

6.3.1 Software tracers to simultaneously trace user-space and kernel

Trace data at both the application and operating system levels are necessary to generate the specifications that will be used for thorough checking and analysis of aforesaid application. Moreover, as we want the specifications to be as precise as possible, we must choose a tracer with low disturbance and high precision to generate the traces. In this section, we present the available software tracers that provide simultaneous tracing of user-space and kernel, and that can interface with the Linux kernel `TRACE_EVENT()` macro.

Perf [14] is one of the tracers built into the Linux kernel. Its main purpose was to access the processors' performance counter, but it was later extended to use the `TRACE_EVENT()` macro and the Linux kernel tracepoints. Yet, even if **perf** can be used as a regular tracer, it has been optimized for sampling. Sampling allows for low-overhead, which makes it interesting in the context of high performance systems. However, it loses in accuracy as compared to regular tracing. Moreover, the collection process is invasive and costly as it is started by an interrupt. Lastly, **perf**'s scalability is limited for multi-core [15].

Another set of built-in Linux kernel tracers are available under the name of *Function Tracer* (**ftrace**) [16]. Its original purpose was to monitor the relative cost of the functions called

in the kernel to identify its bottlenecks. However, it now includes other analysis modules such as for the latency or scheduling analysis [17]. The **ftrace**'s tracers are configured, activated and deactivated through the **debugfs** pseudo-filesystem. **Ftrace** can interface to the `TRACE_EVENT()` macro with its *event tracer* [21]. Analysis time is saved on the tracer side by collecting only the data defined in **ftrace**'s macro through the `TP_printk` macro. However, this makes it impossible to enrich trace events with system context information. UProbes can also be used to connect to user-space applications since Linux kernel 3.5. Yet, to instrument applications, UProbes uses interruptions. This adds potentially unacceptable overhead for real-time and high performance applications.

SystemTap [24] is a monitoring system for Linux targeted for system administrators. To trace the kernel, **SystemTap** can use the `TRACE_EVENT()` macro or KProbes for respectively static or dynamic instrumentation [88]. UProbes can also be used to trace the user-space since Linux kernel 3.8. A special scripting language is used to write the instrumentation, which is then compiled into a kernel module. The scripting language allows to do some data processing in-flight and to print the results at regular interval. Still, there are no facilities to write events to stable storage. Also, both KProbes and UProbes use an interrupt for each tracepoint. As for **ftrace**, it is not possible to avoid this overhead since the only option to trace the user-space is UProbes. This is thus an important issue for high performance and real-time applications.

LTTng-UST is the component of **LTTng** to trace user-space. It allows to add statically compiled tracepoints to applications through the use of macros. The events generated are consumed by an external process that writes them to stable storage. It uses the Common Trace Format [34] and can define arbitrary event types. **LTTng-UST** uses per-CPU ring-buffers and offers high-performance, scalability and wait-free operation. Furthermore, the ring-buffer control variables are updated through atomic operations instead of locking. To avoid cache-line bouncing that happens between readers when using read-write lock schemes, **LTTng-UST** uses read-copy update (RCU) to protect its data structures [89, 33]. A similar architecture is used by the kernel component of the tracer. Moreover, events across layers can easily be correlated at the nanosecond scale, as both user-space and kernel tracers use the same clock for timestamps. Finally, previous work has shown that **LTTng** allows for low latency tracing of real-time applications [43]. Hence, **LTTng** is the best choice to trace real-time and multi-core applications at both user-space and kernel layers.

6.3.2 Automatic analysis of traces with model-based constraints and data extraction

In this section, we present multiple tools and methods that can be used to extract data from traces or perform model-checking over those traces.

Tango [44, 45] automatically generates backtracking trace analysis tools. It uses formal specifications to generate model-specific tools. These tools can then be used to verify that any execution trace follows the aforesaid specifications. However, **Tango** was designed to validate specifications and there is thus no facilities to specify constraints based on the state of the system. It is moreover not possible to automatically build the specifications from the trace.

Similar to **Tango**, the Logic of Constraints (LOC) checker [91] generates an executable checker. The formulas used to generate such a checker are written in a formal quantitative constraint language. Running the executable checker on simulation traces will generate a report informing of any constraint violation. Still, the LOC checker is using text-format traces, which makes it highly sensitive to any trace format change. There are no facilities to build the formulas directly from the simulation traces.

Scalasca [55] uses execution traces to identify bottlenecks. It provides aggregated statistical runtime summaries that can be analyzed to identify the CPU time consumption of system processes. It also provides an event trace analysis that can be used to study the program concurrency. Wait states and related performance properties are extracted from the traces, which allows **Scalasca** to generate a pattern-analysis report. This report contains performance analysis metrics for each function and system resource. Still, it is not possible to provide our own specifications and to adapt them to other contexts.

SETAF [60] is a framework to add properties to the system execution traces and dataflow models for the analysis and validation of the Quality of Service (QoS). **SETAF** needs the user to provide the adaptation patterns to be used on the execution trace. This is done by manually analyzing the trace to identify the missing properties. These patterns will then be used to add the related properties and create the linked causality relations. It thus means that, in its workflow, **SETAF** requires its users to have a deep understanding of the trace format. Moreover, as **SETAF** needs an important user intervention, it is far from automating the model generation.

Trace Compass [62] is originally a trace viewer for **LTng**, but supports multiple trace formats. Multiple views are available to show the results of specific analysis of the trace. **Trace Compass** provides methods to create state system attribute trees. We can then store

metrics that vary along the time axis in the state history tree database. This database is optimized for query efficiency for any requested point in time. It is thus interesting to store and request metrics to check later. Previous work built upon **Trace Compass** provides a way to automatically detect common real-time and multi-core problems using model-based constraints [4]. This approach used both application and system states to allow users to specify thoroughly the expected behavior. However, there is currently no way to automatically build the model-based constraints used for the verification.

To our knowledge, existing model analysis tools do not allow for an automatic generation of the specifications to be used for the verification, while using all the available information. Yet, it was shown in previous work that taking advantage of all the information (i.e., at user and kernel level) provides a more thorough runtime model checking. This work can be improved by using these resources to automatically build a model and a first set of specifications that could then be approved or improved upon by the user.

6.3.3 Model generation and pattern detection

Multiple approaches exist to detect patterns in traces or logs. For instance, **CSight** [76] aims at determining the models of concurrent systems by using the logs of their behavior, in order to help debug and understand such systems. However, this approach still requires the user to specify how to parse and process the log before searching for the patterns.

[77] presents a pattern language that can be used to detect security attacks. Once a pattern is defined, the trace is processed until it is found. Yet, this approach does not allow to identify problems, but rather to find occurrences of a known problem for which a pattern was specified.

A method to identify periodic patterns is presented in [78]. The patterns are identified using the gap between events of the same type. Once different patterns are found, they check the conflicts between them by looking at the perturbation in the periodicity of one, and correlating it to the activity of the others. Yet, this method requires to specify windows or specific events to split the trace into individual worksets, while other similar approaches can find patterns without this need [79]. However, real-time tasks are not always periodic, and both of these approaches do not work for sporadic tasks.

Another method using data mining techniques is presented in [80] to find frequent episodes, which describe temporal relationships between events. The frequency of those episodes is computed by splitting the trace into windows of a fixed size, and computing the number of windows that contain the aforesaid episode. However, splitting the trace in windows can hide

patterns that cannot be split. Moreover, this method uses an incremental approach to build the patterns, making it necessary to read the trace multiple times.

[81] proposes an approach to detect patterns in MPI traces by using suffix trees. The detected communication patterns reflect potential performance bottlenecks, or can highlight special program characteristics. They thus do not intend to represent the full program workflow. Their process is incremental, as they first search for the repeating occurrences of identical events, before building global patterns with the results of the first step. However, it is not possible to set system and application constraints over the patterns found, and thus to use it to detect problems in the execution.

The detection of communication patterns, to understand inter-process communications, has also been studied in [82]. In this case, two algorithms are proposed to do two different parts of the work: recognizing repeated patterns and searching if a given pattern occurs in a MPI trace. Yet, the obtained pattern has to be repeated in the trace to be detected. Also, once again, metrics are missing on the resulting detected pattern.

A model synthesis method is proposed in [83] for PLC programs. It takes into account the reactive behavior of the program, as well as its timing and I/O behaviors, and build an augmented transition model with timing and I/O information. However, this approach only takes into account the behavior of the program and not of the system and other applications.

Useful system metrics can be extracted from kernel traces. The method presented in [84] defines a way to do it. In this approach, multiple state machines represent the different metrics of the application and system, such as the usage rate of system resources, the CPU running time or the elapsed time. However, the different state changes highly depend on the scheduling policies as well as the method used to avoid priority inversions. This method will thus only compute valid results for one specific combination. Moreover, this does not pinpoint the problematic executions, but only provide some metrics.

We can see that existing work to build models from execution traces do not provide metrics that could be used to check the validity of execution instances of the application. Moreover, most of these approaches do not distinguish valid and invalid executions, since all the data read in the trace is considered as valid. Machine learning approaches in the domain of pattern recognition are interesting, but they should be used in correlation with algorithms to detect outliers in order to allow traces containing invalid instances to be used to build a model.

An example of such an outlier detection algorithm is Grubb's test [108]. However, it is too strict to use in a real-time situation where even the smallest variation can lead to an instance invalidity. Other approaches based on clustering can be used. Yet, as we work on one-

dimensional data sets, most clustering algorithms are not appropriate, since one-dimensional data can easily be sorted and thus organized faster. We thus looked at segmentation algorithms. Jenks Natural Breaks Optimization [109] is a solution used in cartography to color maps. It works by seeking to minimize each class deviation from the class average, while maximizing each class deviation from the average of other classes. However, this method requires to specify a number of classes to be determined. Another interesting algorithm for one-dimensional data is the Kernel Density Estimation (KDE) [110]. KDE allows to make inference about the density of the population at any point of its finite data sample. By using the local minima in the density, we could infer the interesting places where to split the data set. However, KDE requires multiple passes over the data to be built and find the local minima at which to perform the split.

6.4 Model-based constraints

Specifying and validating the workflow of an application or system using different metrics can be done using model-based constraints [100]. This process has been extended with success in previous work to take advantage of more sophisticated metrics, extracted from operating system tracing [4]. These metrics can thus be used in constraints to check the correct workflow of an application. The present work uses the same structure and terminology, as the automatically generated models will be used with this process for the detection of unwanted behavior.

The model representation is based on four connected elements: the states, that represent the states of the application, the transitions, representing the movements from one state to itself or another, the variables, that are used to get and store the values of the metrics that are verified in the constraints, and the constraints, that are used to express the expectations for the execution of the application.

The state system is an attribute tree that maintains the modeled state of the operating system and applications. It is based on **Trace Compass**'s state attribute tree, for which the state history database contains the metrics needed for later access. The values for those metrics are computed and stored during the initial reading pass of the kernel trace, but queries to the state history database allow for easy access later.

Three main categories of variables are defined: variables not dependent on the current state of the system (state system free), counters and timers. Variables are categorized according to the needed number of queries to compute their value at a given timestamp. It is thus necessary to query the state history database 0, 1 and 2 times to compute the value of

respectively a state system free variable, a counter and a timer. Each variable will have a type that will be in one of those categories. For instance, the variables of type deadline are in the state system free category, while variables used to count the number of preemptions are in the counter category.

A constraint is associated to a transition. It is checked when this transition happens. As we work with a runtime trace, a transition will happen even if some of its constraints are invalid. A constraint can be uncertain if data is missing to compute the value of the variable on which it applies. A transition will be valid if all of its constraints are valid, uncertain if at least one constraint is uncertain and none are invalid, or invalid if at least one constraint is invalid.

States and transitions are used to follow the instances of the application, which corresponds to a run of that application with a given *tid*. At each instance step, i.e. when a transition is used to move to the next state, the instance will be checked, computing a valid, uncertain or invalid status for the instance step, which will pass on to the instance status.

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0">
  <initial>
    <transition event="event (1)" target="S"/>
  </initial>

  <state id="S">
    <onentry>
      <assign location="type1/var1" expr="0"/>
      <assign location="type2/var2" expr="0"/>
      ...
    </onentry>

    <transition event="event (2)" target="S+1" cond="type1/var1 == 0;
      ↳ type2/var2 > 2; ..."/>
  </state>

  <state id="S+1">
  </state>
</scxml>
```

Figure 6.1 XML representation of the model using State Chart XML

State Chart XML (SCXML) is used to store and load the model-based constraints into the application. Figure 6.1 shows the SCXML representation of a model. We can see the symbols “type1/var1” and “type2/var2” representing two different variables initializations, and “type1/var1 == 0” and “type2/var2 > 2” the two corresponding verifications.

6.5 Generation of the model workflow

Checking the validity of the runs of an application using its trace is similar to protocol or MPI communications verification: we do not know the exact path followed internally by the code of the application; however, the events generated by the application give enough information to understand the evolution of its state. It is by modeling those events, and the inherent application states, that we can model the workflow of the application as seen in the trace. This section explains the process used to generate this high-level approximate model automatically, using the runtime trace, and thus simplifying the user intervention to a simple check and correct.

The process can be summarized as the following steps: reading and organizing the trace events in which the workflow will be identified, identifying and grouping the common workflows in those organized trace events, removing unneeded repetitions in the workflows, building the appropriate model, and finally allowing the user to check and correct the model. These steps will be described in that order.

6.5.1 Organization of the trace events

In order to determine the model workflow of an application from its trace events, we need to understand which events appear in which order. If a full application trace is already accurate in terms of order of appearance of the events, it contains events from multiple applications. Among those multiple applications, we can also find multiple executions of the same application. These multiple executions can follow the exact same workflow (i.e. two threads doing parallel work), or complete each other to perform a general action (i.e. a thread that formats the data while another manages the user interface). The first step of our algorithm is thus to organize the trace events per thread ID. The result is a set of ordered event sequences, each corresponding to the events related to a specific thread.

Figure 6.2 shows the simplified example of a trace with events from four different processes. This example is taken from one of our case studies that will be presented in section 6.8, for which we reduced the events to single letters for readability. These events are thus organized per thread, as shown in Table 6.1. This organization allows to finally see the thread workflows as shown in Figure 6.3.

6.5.2 Identification of the common workflows

When the trace events have been organized per thread, we want to identify the different workflows by regrouping the similar ones. This process is the same as solving the longest

```

{ tid = 1, "A" }
{ tid = 1, "B" }
{ tid = 1, "J" }
{ tid = 1, "C" }
{ tid = 2, "D" }
{ tid = 2, "E" }
{ tid = 3, "H" }
{ tid = 4, "A" }
{ tid = 4, "K" }
{ tid = 2, "F" }
{ tid = 4, "B" }
{ tid = 4, "C" }
{ tid = 3, "I" }
{ tid = 2, "G" }
{ tid = 3, "H" }
{ tid = 3, "I" }

```

Figure 6.2 Simplified example of a trace with events generated by four different processes, where the letters between quotes represent events that should be considered different because of their name or content

```

tid 1 ← [ A B J C ]
tid 2 ← [ D E F G ]
tid 3 ← [ H I H I ]
tid 4 ← [ A K B C ]

```

Figure 6.3 Sequences of events per thread, for all the threads in the simplified trace example presented in Figure 6.2

Table 6.1 Organization per thread of the events read in the simplified trace example presented in Figure 6.2

Event	Thread			
	1	2	3	4
{ tid = 1, "A" }	A			
{ tid = 1, "B" }	B			
{ tid = 1, "J" }	J			
{ tid = 1, "C" }	C			
{ tid = 2, "D" }		D		
{ tid = 2, "E" }		E		
{ tid = 3, "H" }			H	
{ tid = 4, "A" }				A
{ tid = 4, "K" }				K
{ tid = 2, "F" }		F		
{ tid = 4, "B" }				B
{ tid = 4, "C" }				C
{ tid = 3, "I" }			I	
{ tid = 2, "G" }		G		
{ tid = 3, "H" }			H	
{ tid = 3, "I" }			I	

common subsequence (LCS) problem multiple times, as we need to find, for each two sequences of events that are sharing at least one event, the longest common subsequence of events. The aim is to identify, among all the available sequences, the ones that are sharing the most similarities, and to extract those similarities to be the common workflow. Each treated sequence will, in turn, be merged into the most alike common workflow, while removing differences. This process can thus be done in a time linearly proportional to the number of elements in the sequences.

Table 6.2 Example of the process used to regroup similar sequences to identify the different workflows by solving the longest common subsequence problem on the sequences of events. This example uses the sequences of events provided in Figure 6.3

Sequence	Workflows	LCS	Result
[A B J C]	\emptyset		Add [A B J C]
[D E F G]	[A B J C]	\emptyset	Add [D E F G]
[H I H I]	[A B J C]	\emptyset	
	[D E F G]	\emptyset	Add [H I H I]
[A K B C]	[A B J C]	[A B C]	
	[D E F G]	\emptyset	
	[H I H I]	\emptyset	Merge into [A B C]

Table 6.2 shows the process on our example. We can see that, among the different sequences we got in the previous step, two were identical except for one event each. This process removed those events to match the two sequences together, and merge them into a common workflow [A B C]. At the end of this process, our example thus ends up with three different workflows which are [A B C], [D E F G] and [H I H I].

In some cases, beyond our simple example, the events in the trace could contain information that would make them distinct while they should be considered identical. If we take for instance the case of an application that runs a parallel task doing the same work in multiple threads, and for which events store the thread number. Each thread will have its own sequence of events, but all the events in those sequences will also contain a number that will differ from one thread to another, and thus from one workflow to another. Such a situation would prevent the LCS solver to work properly. We thus defined a minimum grouping rate G_{\min} that, when not met, reduces the event matching requirements. When strict matching is used, both the name and the content of the events have to be identical. With flexible matching, the name of the events still has to be the same, but only a fraction of the content

has to be identical. This fraction is defined by computing the maximum number of entries in the content of an event of the sequence, and reducing by one the number of elements that have to match until we reach the minimum grouping rate. This can be avoided when the ideal grouping rate of the sequences – i.e. when considering only event names – does not reach the minimum grouping rate.

6.5.3 Removal of the unneeded repetitions

When reaching this step of the model building process, the different sequences represent different workflows found in the trace. Some of those workflows can be based on repetitions of the same sequence of events, for instance if the traced application is based on a loop. In such cases, keeping that workflow does not provide an accurate model of what should be checked. Moreover, this lack of accuracy will impact the constraints that will be defined in the next step. At this step, we thus aim to remove unneeded repetitions, as well as unneeded events that would appear in between those repetitions.

This problem is similar to the longest repeated substring problem, to which we add a non-overlapping condition. Each workflow will be treated separately, as the problem has to be solved distinctly for each case. We first build a suffix tree using the sequence of events of the workflow, which can be done in linear time [111]. Once the tree is built, we can extract the ordered suffix array and use it to find the longest repeated sequence which does not overlap, as explained in [112]. The extracted substring, at this step, represents the longest most repeated sequence of events that can be found in the workflow.

However, this longest most repeated sequence of events can itself contain repetitions of events, which would represent the multiple times the loop would have ran. At this point, no event should be excluded from the workflow. Given that, if the workflow contains only the same sequence of events repeated over and over, we reduce it to just one occurrence of that sequence.

Following our example, the workflows [A B C] and [D E F G] would remain intact, while [H I H I] would become [H I] by the end of this step.

6.5.4 Build and clean the model

The last step of the model generation is to actually build the model using the identified workflows. For each workflow, if we consider a sequence of n events $e_0 \dots e_{n-1}$, we will have a model with n states $s_0 \dots s_{n-1}$. In this model, state s_i transitions to s_{i+1} when triggered by the event e_{i+1} for $0 \leq i < n - 1$. To finalize the general loop of the model, state s_{n-1} will

transition to s_0 when triggered by the event e_0 . Finally, initial transitions will be created to all available states, using the corresponding events, i.e. event e_i will move the model to state s_i when outside an instance of the model, for $0 \leq i \leq n - 1$. This schema is used as it is difficult from the trace to know what is exactly the start of a workflow. Moreover, it is possible that the tracing started after the application, and therefore that the first events in the trace would not be the first ones in the workflow. Allowing a loop in the states built, as well as to enter the workflow from any state, thus permits to manage such cases properly.

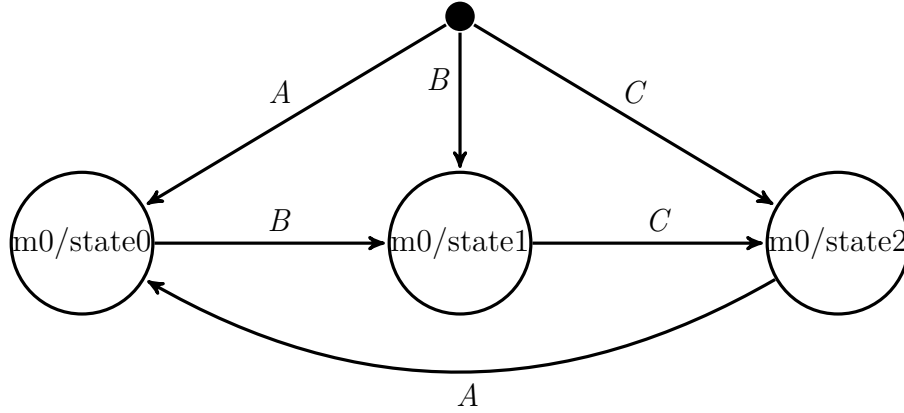


Figure 6.4 Unconstrained workflow model built from the sequence of events [A B C]. This model links the events in the order in which they appear and allows for a loop.

Figure 6.4 gives the example of the unconstrained model built from the [A B C] workflow. Such a model is built for each of the workflows identified in the trace, and those are merged to a general workflow that applies to the trace read. The general workflow of our example is as shown in Figure 6.5.

However, an unconstrained model is not that useful for problem detection. Therefore, we add to each created state the initialization of a new variable of each type. For each secluded workflow, each transition from one state to another will be constrained by a constraint on each of the variables. For instance, if each of the n states of a model have k variables, each transition will have $n \times k$ constraints. This can thus be done in time linearly proportional to the number of states and types of variables. The type of variables that will be used can be limited by the user, allowing for a simpler model, and thus a faster building. But specifying the variables on which a constraint applies is only one of the three needed elements: the operator used for comparison and the value to compare to are still missing. We thus use a special kind of constraint, called “adaptive” constraints as they adapt to the traces on which the model is used for analysis. These constraints are described in section 6.6.

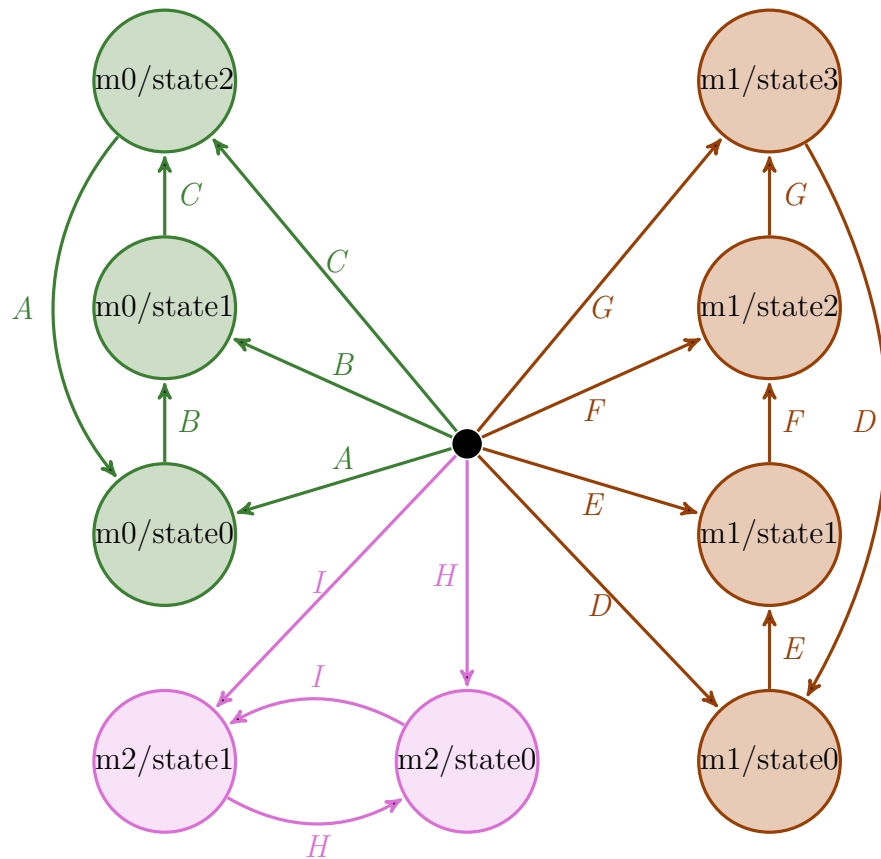


Figure 6.5 Unconstrained workflow model built from all the sequence of events found in the example of trace shown in Figure 6.2. This model allows to enter any state identified from the trace, and to follow the related workflow.

While the trace will be read again for detection using this generated model, the adaptive constraints will receive data when relevant. Once the trace has been fully read, the useful adaptive constraints will be able to determine both an operator and a value to use for comparison. The adaptive constraints left undetermined will be cleaned out of the model, as well as the variables for which no constraint is left in the model.

With this process, extra constraints, i.e. constraints on transitions that should not be constrained by the related variables, are to be expected in the final model. This is why the cleaned-up model is presented to the user to be checked and verified for accuracy. Once the user confirms the model, the detection process will start. If the user did not change anything in the model, there will be no need to read the trace again before displaying the detection and analysis results, as these were computed while completing the adaptive constraints.

6.6 Adaptive constraints

Constraints are defined by three elements: the two operands, and the operator. When working with our representation, one of the operands is the variable on which the constraint applies. The other operand is a value that will be used for comparison by using the operator to determine whether or not the variable complies with the expectations when it is encountered in the model workflow. Depending on the kind of variable, the variable value when checking the constraint can be obtained directly from the event that lead to the trace (for a state system free variable) or from the built state system (for counters and timers). When working to automatically generate an application workflow from an execution trace, having the user manually set each constraint value and operator defies the purpose.

Adaptive constraints are a kind of constraint that adapts to the traces on which the model is used for analysis. It means that, given a model of the application workflow that we can follow, along with a set of constraints for which some have unknown comparison values or operators, we can use the trace data to compute the best values and operators to use for those constraints.

In order to save processing time, the adaptive constraints, as usual constraints, are processed while reading the traces. However, in our previous work we used to set the status for the constraints while reaching them during the analysis. It is not possible to do the same with adaptive constraints. Indeed, the value that we will use for comparison will not be known until the traces have been fully read. Instead, we store the runtime values per-constraints. This means that two adaptive constraints A_1 and A_2 will each have their own list of values

V_1 and V_2 once we finish reading the trace. These lists of values will then be used to define the operator and/or the value needed to complete the related constraint.

At the end of the detection process, the adaptive constraints of the model enter the revalidation step. This step consists in using the computed operator and/or value to assign a status to the now completed constraint, and thus to its instance step, and to its instance. At the end of this step, the adaptive constraint status will either be valid, if the runtime step validated the constraint, invalid, if it did not, or uncertain, if we were unable to choose an operator or a value to complete the constraint.

Considering the list of runtime values gathered while doing the detection step of the analysis, we will first present the process used to assign comparison values when the operator is known, then the process used to assign comparison operators with or without a known comparison value.

6.6.1 Adaptive comparison value

To determine the comparison value that will be used for a constraint, we first need a non-empty list of runtime values. Indeed, if the list of values with which we would need to work is empty, it means that the constraint for which we want to compute a comparison value has never been reached while reading the trace, making the constraint useless for the analysis of this trace. The second condition is for the operator not to be “not equal”: it would not be possible to determine to which value the runtime values should not be equal to, as the possibilities for such value would be endless. If those conditions are not met, the computed value would stay undetermined, insuring the uncertain status for the constraint.

The user can ask to consider all the runtime values found in the trace as valid. This situation can be encountered when the trace fed only contains valid runs of the application. It is then used to complete the constraints of the models in order to check other runs for which the user is not sure of the validity. In such a case, a simple analysis of the values in the list for this constraint allows to determine the comparison value to use. When using less, less or equal, greater or greater or equal operators, the value can be straightforwardly computed from the maximum or minimum values in the list. On the other hand, with the equal operator, we can determine the value if and only if the list of values contain one unique value V_0 , as all values in the list must be considered as validating the constraint.

(6.1) summarizes the computed comparison value v for the different constraint operators when all values in the list of runtime values V are considered as valid.

$$v = \begin{cases} \max V + 1 & \text{if operator is } < \\ \max V & \text{if } \leq \\ \min V - 1 & \text{if } > \\ \min V & \text{if } \geq \\ V_0 & \text{if } = \text{ and } V = [V_0 V_0 \dots V_0] \\ \text{unknown} & \text{else} \end{cases} \quad (6.1)$$

Algorithm 6.1 Segmentation algorithm for a list of *data*, using a given *threshold* that can be expressed in absolute or relative terms. The result is a list of segmented values from the original data.

```

1 function segmentation(V, threshold, usePercent)
2   s ← new list
3   if #V = 0 then return s
4   s.add(new list)
5   next ← V[0]
6   forall i in 1..#V do
7     prev ← next
8     next ← V[i]
9     s[#s - 1].add(prev)
10    diff ← next - prev
11    if usePercent then
12      diff ←  $\frac{\text{diff}}{\text{prev}}$ 
13    if diff ≥ threshold then
14      s.add(new list)
15  s[#s - 1].add(next)
16  return s

```

A more thorough analysis, with a less reliable outcome, can be required when the traces contain both valid and invalid instances of the application. Two different situations persist at that step: whether the constraint uses an equal operator or not. In the case of an equal operator, the computed value will be the value that is most present in the list of runtime values. In the other case, we use segmentation to split our runtime values according to their proximity to each other. As seen in section 6.3, current methods requires to specify a number of segments or heavy computation to determine the actual segments. As segmentation has to be performed on a number of data sets, we chose to segment our data using a simpler algorithm that will create a new segment each time a threshold is reached in the distance between two successive values of a sorted list. Algorithm 6.1 gives the pseudocode of such

algorithm. However, the segmentation cannot be done directly on the list of values without knowing the right threshold for these data, that we will call *split threshold*. We determine such threshold by using the segmentation algorithm on the ordered set of adjacent differences between our sorted values, with a significant difference relative threshold (in percentage) T_{\gg} . T_{\gg} chosen to mark a sufficient discrepancy between adjacent difference values in order to identify outliers.

If we note t_{split} the computed split threshold, s the list of segments after the segmentation process, and s_i the list of values in the i^{th} segment, the value of t_{split} is determined by (6.2).

$$t_{\text{split}} = \begin{cases} \frac{\max s_{\#s-2} + \min s_{\#s-1}}{2} & \text{if } \#s > 1 \\ \overline{s_0} & \text{else} \end{cases} \quad (6.2)$$

A second call to the segmentation algorithm, on our sorted list of values, with t_{split} as threshold, will give us the segments with which we will decide on the value to use. At this point, the segment of interest for us will be the one with the highest number of values. This biggest segment will then be used according to the operator defined for the constraint, and its maximum and minimum will be used to determine the computed value.

Considering that the values in the list of runtime values V can be valid or invalid, (6.3) summarizes the computed comparison value v for the different constraint operators, with B the biggest segment.

$$v = \begin{cases} \max B + 1 & \text{if operator is } < \\ \max B & \text{if } \leq \\ \min B - 1 & \text{if } > \\ \min B & \text{if } \geq \\ \text{Mo } V & \text{if } = \\ \text{unknown} & \text{else} \end{cases} \quad (6.3)$$

6.6.2 Adaptive comparison operator

When it comes to determining the comparison operator to use for the constraint, we also have to consider whether or not all the runtime values are valid – as specified by the user – and whether or not we have a comparison value or we need to determine it too.

With a known comparison value

If the comparison value is defined in the constraint but only the operator is to be determined, we can use that value along the runtime values to determine the operator that should be used. In the case where all the runtime values are valid, we can compare them to the comparison value. In this situation, the constraint operator o could be determined according to (6.4), where V is the list of runtime values and v the comparison value.

$$o = \begin{cases} = & \text{if } V = [V_0 V_0 \dots V_0] \text{ and } v = V_0 \\ \neq & \text{if } V = [V_0 V_0 \dots V_0] \text{ and } v \neq V_0 \\ < & \text{if } v > \max V \\ \leq & \text{if } v = \max V \\ > & \text{if } v < \min V \\ \geq & \text{if } v = \min V \\ \text{unknown} & \text{else} \end{cases} \quad (6.4)$$

However, when it is needed to differentiate valid and invalid values among the runtime data, (6.5) can be used instead.

$$o = \begin{cases} = & \text{if } v = \text{Mo } V \\ \geq & \text{if } \# \{x \in V | x < v\} < \# \{x \in V | x > v\} \\ \leq & \text{else} \end{cases} \quad (6.5)$$

With an unknown comparison value

If the comparison value for the constraint is not defined either, we need to both define this value and the operator to use. If the user specified for the runtime values to be all considered as valid, (6.6) will be used, with V the list of runtime values and v and o respectively the computed value and operator.

$$o, v = \begin{cases} =, V_0 & \text{if } V = [V_0 V_0 \dots V_0] \\ \leq, \max V & \text{else} \end{cases} \quad (6.6)$$

In such a situation, we consider only the equal and less or equal situations as they are the most common operators to be used for high performance and real-time applications. Indeed, in such systems, we usually set thresholds not to be exceeded.

Algorithm 6.2 Algorithm used to determine the best operator and value to use for the constraint with a given list of runtime values V , a set of segments of these values s , and the index of the biggest segment $idxB$

```

1 function computeOpAndVal( $V, s, idxB$ )
2   while true do
3      $n_{\text{before}} \leftarrow \# \{x \in V | x < \min s[idxB]\}$ 
4      $n_{\text{after}} \leftarrow \# \{x \in V | x > \max s[idxB]\}$ 
5     if  $n_{\text{before}} = 0$  and  $n_{\text{after}} = 0$  then
6       return unknown
7     if  $\frac{\#s[idxB]}{\#V} \geq T$  then
8        $b \leftarrow n_{\text{before}} = 0$  or  $n_{\text{before}} > n_{\text{after}}$ 
9        $a \leftarrow n_{\text{after}} = 0$  or  $n_{\text{after}} > n_{\text{before}}$ 
10    else
11       $b \leftarrow n_{\text{after}} > 0$  and  $\frac{n_{\text{before}}}{n_{\text{after}}} \geq (1 + T_{\gg})$ 
12       $a \leftarrow n_{\text{before}} > 0$  and  $\frac{n_{\text{after}}}{n_{\text{before}}} \geq (1 + T_{\gg})$ 
13    if  $b$  then return  $\leq, \max s[idxB]$ 
14    if  $a$  then return  $\geq, \min s[idxB]$ 
15    if  $idxB = 0$  then
16       $closest \leftarrow 1$ 
17       $space_{\text{out}} \leftarrow \min s[closest] - \max s[idxB]$ 
18       $space_{\text{in}} \leftarrow \max s[idxB] - \min s[idxB]$ 
19      if  $space_{\text{out}} > 3 \times space_{\text{in}}$  then
20        return  $\leq, \max s[idxB]$ 
21    else if  $idxB = \#s - 1$  then
22       $closest \leftarrow idxB - 1$ 
23       $space_{\text{out}} \leftarrow \min s[idxB] - \max s[closest]$ 
24       $space_{\text{in}} \leftarrow \max s[idxB] - \min s[idxB]$ 
25      if  $space_{\text{out}} > 3 \times space_{\text{in}}$  then
26        return  $\geq, \min s[idxB]$ 
27    else
28       $space_{\text{before}} \leftarrow \min s[idxB] - \max s[idxB - 1]$ 
29       $space_{\text{after}} \leftarrow \min s[idxB + 1] - \max s[idxB]$ 
30      if  $\frac{\min space_{\text{before}}, space_{\text{after}}}{\max space_{\text{before}}, space_{\text{after}}} \geq 90\%$  then
31        if  $n_{\text{before}} \geq n_{\text{after}}$  then
32           $closest \leftarrow idxB - 1$ 
33        else
34           $closest \leftarrow idxB + 1$ 
35      else if  $space_{\text{before}} < space_{\text{after}}$  then
36         $closest \leftarrow idxB - 1$ 
37      else
38         $closest \leftarrow idxB + 1$ 
39     $idxB = s.\text{mergeSegments}(idxB, closest)$ 

```

Still, when considering that both valid and invalid values can be in the gathered data, we do not have any indication for the value and operator to use. We thus chose first to consider that if a value is consistently present among the runtime values, this is most probably the value that we expected to read. This is represented by the threshold $T_{=}$, that must be set to be a sufficiently high representation of a unique value, while being possible to reach. In that case, using an equal operator and the most present value in the list of runtime values as respectively the comparison operator and value for the constraint seems appropriate. Nonetheless, when such a condition is not met, we require the use of segmentation. The segmentation is performed as shown in Algorithm 6.1, with the same process explained in section 6.6.1 to determine the split threshold. The operator and value are then determined using the process described in Algorithm 6.2. This process consists in merging the biggest segment with its closest neighbor until we reach a situation where:

1. There is not any other segment than the biggest, in which case we cannot determine which direction should the operator take – as there could be invalid values – and thus return an unknown result;
2. The biggest segment represents more than T of the runtime values, and there is no values before the biggest segment or the number of values before this segment is greater than the number of values after it, in which case we return an operator less or equal than the maximum value in the biggest segment;
3. The biggest segment represents more than T of the runtime values, and there is no value after the biggest segment, or the number of values after this segment is greater than the number of values before it, in which case we return an operator greater or equal than the minimum value in the biggest segment;
4. The number of values before the biggest segment represents more than T_{\gg} more values than the number of values after this segment, in which case we return an operator less or equal than the maximum value in the biggest segment;
5. The number of values after the biggest segment represents more than T_{\gg} more values than the number of values before this segment, in which case we return an operator greater or equal than the minimum value in the biggest segment.

The threshold T is expected to be set lower than $T_{=}$, as the latter is used for a type of operator that is much more restrictive than the former. Indeed, setting an equal constraint forces the values to be exactly the one used in the constraint, while the other operators allow a wider range of valid values.

Determining the operator and value to use can thus be summarized by (6.7), where V is the list of runtime values.

$$o, v = \begin{cases} =, \text{Mo } V & \text{if } \frac{\#\{x \in V | x = \text{Mo } V\}}{\#V} \geq T_{=} \\ \text{Algorithm 6.2} & \text{else} \end{cases} \quad (6.7)$$

6.7 User interface

To improve the user experience, a simple user interface allows to define the needed parameters for the model generation, as well as to specify a model when such generation is not needed. Figure 6.6 shows this interface. We can use it to choose the type of variables to be used for the model generation, as well as to limit the time interval in the trace in which to process relevant events on which to build the workflow. Two checkboxes allow the user to specify whether or not to consider all the data in the trace as valid, and whether or not it wants to check the model before running the detection process.

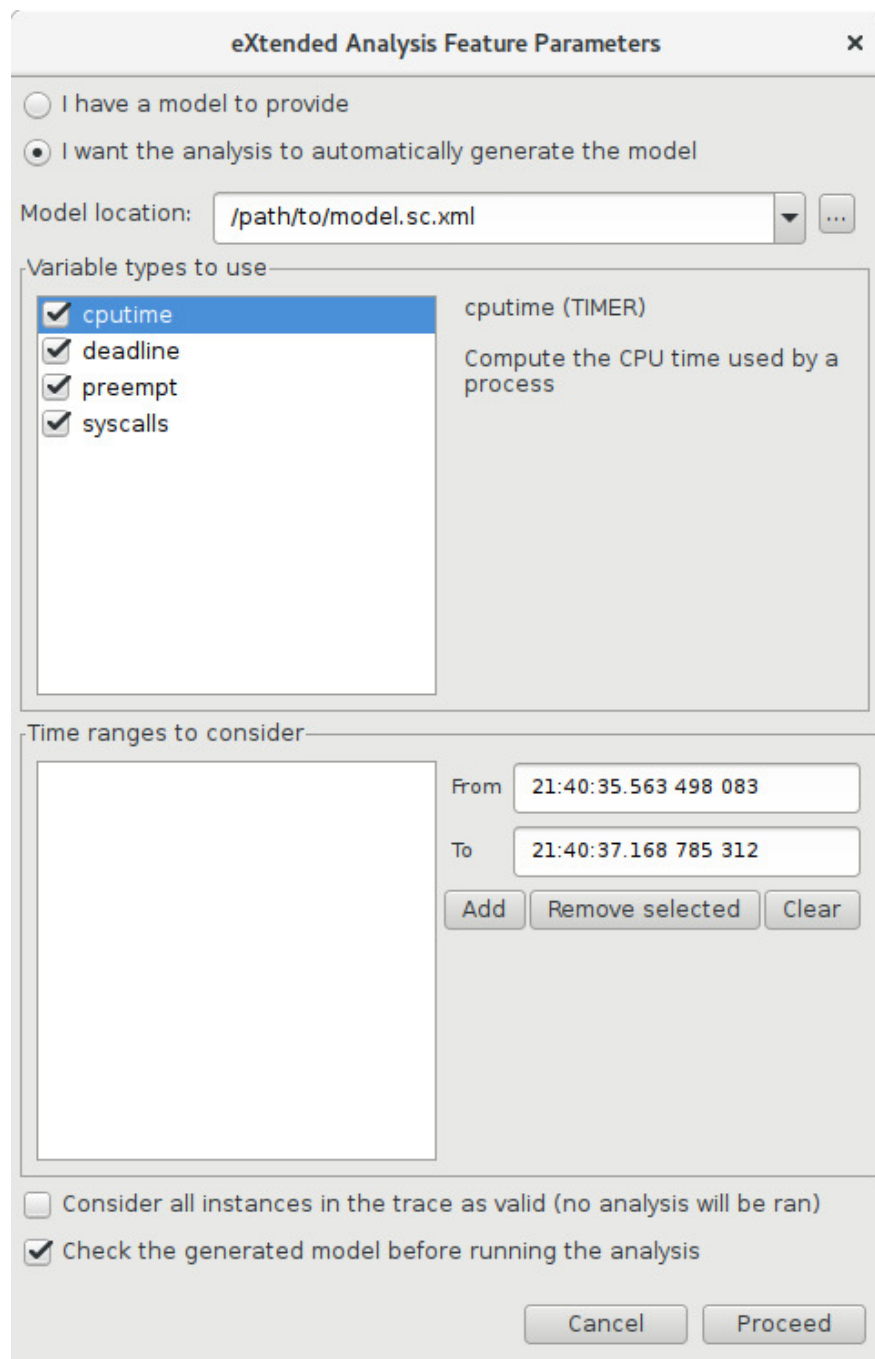
6.8 Case studies

This section presents different case studies of real-time applications on which we evaluated our automated model generation approach. These applications were traced while running correctly as well as while a problem was occurring, in order to see if our algorithms were able to generate a model that would properly detect those problems. We also took advantage of our knowledge of those applications to compare the generated models to models we built manually in order to detect these situations.

6.8.1 JACK2

JACK2 is a real-time application that can run with a periodic workload. As soon as it starts, the ALSA driver is being reconfigured to sample at a given frequency and accumulate a number of frames before raising an interrupt. It is therefore possible to instrument that period by adding tracepoints around it. This will identify in the trace when JACK2 starts its sampling period by waiting to be awakened, and when it is actually woken up.

We generated a trace using this instrumentation, which contains an instance of the application missing its deadline. Figure 6.7 shows the JACK2 daemon being started and reconfiguring the ALSA driver. We can identify that the sample frequency is 48kHz. The number of



The dialog box is titled "eXtended Analysis Feature Parameters" and has a close button (X) in the top right corner. It contains two radio buttons for model selection: "I have a model to provide" (unselected) and "I want the analysis to automatically generate the model" (selected). Below the radio buttons is a text field for "Model location:" containing the path "/path/to/model.sc.xml" and a browse button (...). The "Variable types to use" section features a list box with four checked items: "cputime", "deadline", "preempt", and "syscalls". To the right of the list box, the description "cputime (TIMER) Compute the CPU time used by a process" is displayed. The "Time ranges to consider" section includes a large empty list box on the left and a right-hand area with "From" and "To" time input fields (showing "21:40:35.563 498 083" and "21:40:37.168 785 312" respectively) and three buttons: "Add", "Remove selected", and "Clear". At the bottom, there are two checkboxes: "Consider all instances in the trace as valid (no analysis will be ran)" (unchecked) and "Check the generated model before running the analysis" (checked). Finally, "Cancel" and "Proceed" buttons are located at the bottom right.

eXtended Analysis Feature Parameters

☐ I have a model to provide

☒ I want the analysis to automatically generate the model

Model location: ...

Variable types to use

<input checked="" type="checkbox"/> cputime	cputime (TIMER)
<input checked="" type="checkbox"/> deadline	Compute the CPU time used by a process
<input checked="" type="checkbox"/> preempt	
<input checked="" type="checkbox"/> syscalls	

Time ranges to consider

	From <input type="text" value="21:40:35.563 498 083"/>
	To <input type="text" value="21:40:37.168 785 312"/>
	<input type="button" value="Add"/> <input type="button" value="Remove selected"/> <input type="button" value="Clear"/>

☐ Consider all instances in the trace as valid (no analysis will be ran)

☒ Check the generated model before running the analysis

Figure 6.6 User interface to configure the model generation, or choose an existing model to use for the detection

```

$ taskset -c 0 jackd -P 1 -v -d alsa -H
...
creating alsa driver ... hw:0|hw:0|1024|2|48000|0|0|hwmon|swmeter|-|32bit
configuring for 48000Hz, period = 1024 frames (21.3 ms), buffer = 2
    ↪ periods
...

```

Figure 6.7 ALSA driver being configured by the JACK2 daemon process after it was started with a low real-time priority of 1 and pinned on CPU 0 [106]

```

...
[11:54:37.670218483] (+0.000006748) duck lttng_ust_tracef:event: { cpu_id
    ↪ = 0 }, { vtid = 11366 }, { _msg_length = 11, msg = "async_start" }
[11:54:37.691894701] (+0.021676218) duck lttng_ust_tracef:event: { cpu_id
    ↪ = 0 }, { vtid = 11366 }, { _msg_length = 10, msg = "async_stop" }
[11:54:37.691900793] (+0.000006092) duck lttng_ust_tracef:event: { cpu_id
    ↪ = 0 }, { vtid = 11366 }, { _msg_length = 11, msg = "async_start" }
[11:54:38.297457783] (+0.605556990) duck lttng_ust_tracef:event: { cpu_id
    ↪ = 0 }, { vtid = 11366 }, { _msg_length = 18, msg = "xrun 584.244
    ↪ msecs" }
[11:54:38.319585808] (+0.022128025) duck lttng_ust_tracef:event: { cpu_id
    ↪ = 0 }, { vtid = 11366 }, { _msg_length = 10, msg = "async_stop" }
...

```

Figure 6.8 Part of the content of JACK2's userspace trace used for the case study

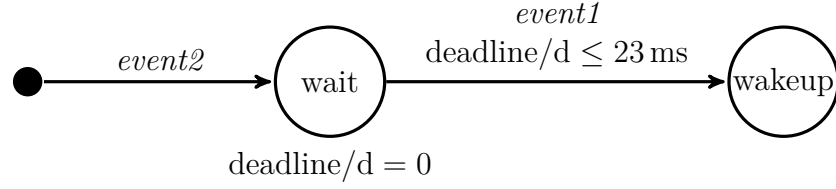
frames accumulated during each period will be 1024, which leads to an expected runtime of 21.3ms per period. The deadline miss was provoked by running the JACK2 daemon with a low real-time priority, and pinning it on a single CPU. We then ran a CPU-intensive task on the same CPU, for a duration longer than JACK2's period, and with a higher priority. This latter then disrupts the JACK2 process, causing a missed deadline. The generated trace thus contains a period during which JACK2 reported an *xrun* of “at least” 353.963ms. We can see the events corresponding to a normal as well as the disrupted period in Figure 6.8.

Figure 6.9(a) represents the model that we built manually to follow JACK2's period. The deadline set on that model is for a duration of less than 23ms, as it needs to consider the waking time of the process. When we compare this manually built model to the automatically generated one presented in Figure 6.9(b), we can see that the “wait” state of Figure 6.9(a) corresponds to the “m0/state1” state of the Figure 6.9(b), while the “wakeup” state corresponds to the “m0/state0” state. On the constraints side, as we did not limit the type of constraint to use for the generation, a lot of constraints appear on the model but are not useful for the detection of the deadline miss. Only deadline constraints are interesting at this point, and we can see that there are more constraints than what we had in the manually built model. This is expected as the automatic process does not know what the user wants. If we look at the values of the constraint, we can see that the constraint on the variable “deadline/d0.1” in the transition from “m0/state1” to “m0/state0” corresponds to the one we had in our manually built model. We can see that this constraint considers valid a value of less than about 22.062ms, which is very close to the manually set 23ms.

When looking at the results, we can see that the *xrun* is clearly detected when using the manually built model, as seen in Figure 6.10(a). Using the automatically generated model, the results presented in Figure 6.10(b) show that, among the 7 constraints invalidated, 3 are about that *xrun*. When removing the non-deadline constraints, 3 of the 4 invalid constraints are about the *xrun*. This shows that the automatic generation of the model worked properly for JACK2, as we did not expect a perfect match of our manually built model. A close approximation, allowing to detect automatically unwanted behavior, is already a considerable achievement.

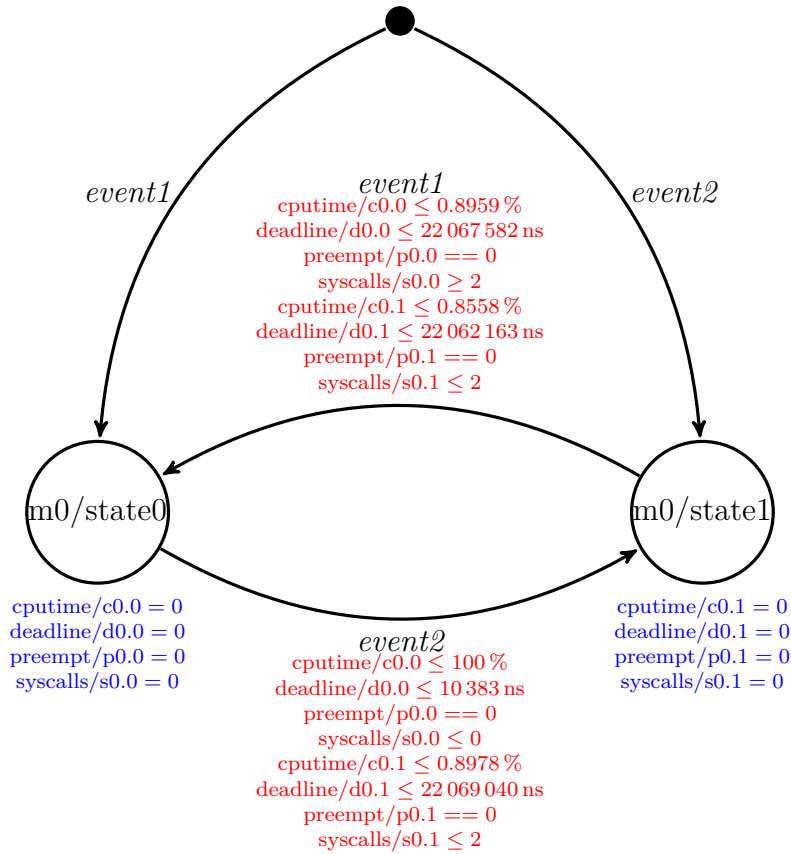
6.8.2 cyclicttest

Cyclicttest is a tool that checks the real-time performance of a system by executing a periodic task with multiple processes. The aim of the task is to be woken up during the given period. The performance is then evaluated by measuring the distance between the desired period duration and the real duration after wake up.



`event1 = lttnng_ust_tracef:event[msg=async_stop, _msg_length=10]`
`event2 = lttnng_ust_tracef:event[msg=async_start, _msg_length=11]`

(a) Model manually built to correspond to the period of JACK2



`event1 = lttnng_ust_tracef:event[msg=async_stop, _msg_length=10]`
`event2 = lttnng_ust_tracef:event[msg=async_start, _msg_length=11]`

(b) Model automatically generated from a runtime trace of JACK2, using all the variables types

Figure 6.9 Models to check the valid execution of JACK2 and detect unwanted behaviors

```

Received lttng_ust_tracef:event[msg=async_stop, _msg_length=10] at
  ↳ 11:54:38.319 585 808
    Entering state: wakeup
    Constraints:
      - deadline/d <= 23ms [INVALID] value: 627.6850ms

```

(a) Results of the detection on JACK2's trace using the model presented in 6.9(a), showing only the invalid cases

```

Received lttng_ust_tracef:event[msg=async_start, _msg_length=11] at
  ↳ 11:54:36.689 252 577
    Entering state: m0/state1
    Constraints:
      - syscalls/s0.0 <= 0 [INVALID] value: 1.0
Received lttng_ust_tracef:event[msg=async_start, _msg_length=11] at
  ↳ 11:54:36.732 033 184
    Entering state: m0/state1
    Constraints:
      - deadline/d0.0 <= 10383.0000ns [INVALID] value: 38983.0000ns
Received lttng_ust_tracef:event[msg=async_stop, _msg_length=10] at
  ↳ 11:54:38.319 585 808
    Entering state: m0/state0
    Constraints:
      - deadline/d0.0 <= 22067582.0000ns [INVALID] value:
        ↳ 627691107.0000ns
      - deadline/d0.1 <= 22062163.0000ns [INVALID] value:
        ↳ 627685015.0000ns
      - syscalls/s0.1 <= 2 [INVALID] value: 13.0
Received lttng_ust_tracef:event[msg=async_start, _msg_length=11] at
  ↳ 11:54:38.319 590 857
    Entering state: m0/state1
    Constraints:
      - deadline/d0.1 <= 22069040.0000ns [INVALID] value:
        ↳ 627690064.0000ns
      - syscalls/s0.1 <= 2 [INVALID] value: 13.0
Received lttng_ust_tracef:event[msg=async_stop, _msg_length=10] at
  ↳ 11:54:39.705 960 368
    Entering state: m0/state0
    Constraints:
      - syscalls/s0.0 >= 2 [INVALID] value: 1.0

```

(b) Results of the detection on JACK2's trace using the model presented in 6.9(b), showing only the invalid cases

Figure 6.10 Results of the detection on JACK2's trace

```

...
[15:55:53.528520854] (+0.000019250) tegra cyclicttest:wait: { cpu_id = 3 },
    ↪ { vtid = 1887 }, { id = 3 }
...
[15:55:53.535000736] (+0.000014750) tegra cyclicttest:wait: { cpu_id = 0 },
    ↪ { vtid = 1884 }, { id = 0 }
[15:55:53.535942648] (+0.000941912) tegra cyclicttest:run: { cpu_id = 0 },
    ↪ { vtid = 1884 }, { id = 0 }
[15:55:53.535982731] (+0.000040083) tegra cyclicttest:wait: { cpu_id = 0 },
    ↪ { vtid = 1884 }, { id = 0 }
[15:55:53.536082481] (+0.000099750) tegra cyclicttest:run: { cpu_id = 3 },
    ↪ { vtid = 1887 }, { id = 3 }
[15:55:53.536093064] (+0.000010583) tegra cyclicttest:outlier: { cpu_id = 3
    ↪ }, { vtid = 1887 }, { id = 3 }
...

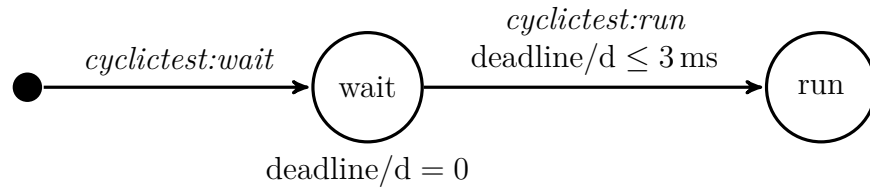
```

Figure 6.11 Part of the content of `cyclicttest`'s userspace trace used for the case study

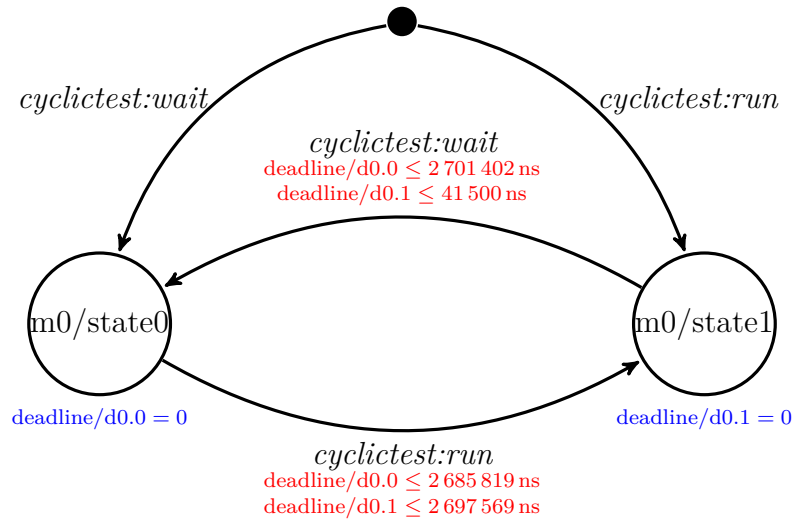
We instrumented `cyclicttest` to generate events before and after its periodic task, in order to be able to verify that it stays within a given period. We then generated a trace by running a task on each CPU, with a real-time priority of 99 and the `SCHED_FIFO` scheduler. The trace was saved using the snapshot mode of `LTTng`, allowing to write the trace buffers to disk as soon as a high latency was detected by the application. Figure 6.11 shows some events of the trace, where we can see, among others, a valid and an invalid periods, the latter causing the generation of a “`cyclicttest:outlier`” event.

Figure 6.12(a) shows a model that was manually built to automatically detect such outlier, by putting a deadline constraint of 3 ms. Our algorithms lead to the generation of the model presented in Figure 6.12(b). We can see that the “wait” state in our manually built model corresponds to the “m0/state0” state, while the “run” matches the “m0/state1” one. On the constraints side, the constraint on variable “deadline/d0.0” coincide with the one on “deadline/d” of our manually built model. We can see as well that, where the value we set was of 3 ms, the automatically set value is of 2.685 ms. The models are thus really close, except for the extra constraints.

When comparing the detection results, we can see that in both the results for the manually built model shown in Figure 6.13(a) and those for the automatically generated one shown in Figure 6.13(b) the outlier is detected. The latter shows more invalid constraints, but we can see with their values that they are linked to the same outlier. This therefore shows that the automated model generation worked properly in this case too.



(a) Model manually built to correspond to the period of `cyclicttest`



(b) Model automatically generated from a runtime trace of `cyclicttest`, using only deadline variables

Figure 6.12 Models to check the valid execution of `cyclicttest` and detect unwanted behaviors

```

Received cyclicttest:run at 15:55:53.536 082 481
  Entering state: run
  Constraints:
    - deadline/d <= 3ms [INVALID] value: 7.5616ms

```

(a) Results of the detection on `cyclicttest`'s trace using the model presented in 6.12(a), showing only the invalid cases

```

Received cyclicttest:run at 15:55:53.536 082 481
  Entering state: m0/state1
  Constraints:
    - deadline/d0.0 <= 2685819.0000ns [INVALID] value: 7561627.0000ns
    - deadline/d0.1 <= 2697569.0000ns [INVALID] value: 7580877.0000ns
Received cyclicttest:wait at 15:55:53.536 208 563
  Entering state: m0/state0
  Constraints:
    - deadline/d0.0 <= 2701402.0000ns [INVALID] value: 7687709.0000ns
    - deadline/d0.1 <= 41500.0000ns [INVALID] value: 126082.0000ns

```

(b) Results of the detection on `cyclicttest`'s trace using the model presented in 6.12(b), showing only the invalid cases

Figure 6.13 Results of the detection on `cyclicttest`'s trace

6.8.3 In-kernel wakelock application

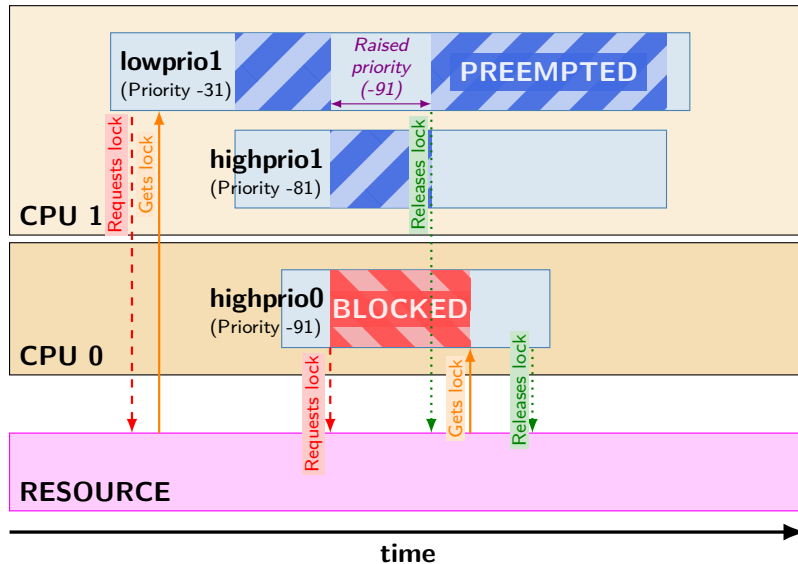


Figure 6.14 Schema of a priority inheritance while waiting for a resource: the **lowprio1** process takes the lock of the resource and is preempted by a higher priority process **highprio1**, but can finish its task and free the lock when inheriting temporarily the priority of process **highprio0**

One of the most common problems in real-time systems are priority inversions. They happen when a process prevents another process with a higher priority to run. They usually involve a shared resource with a limited access. We built an application to emulate such a situation through a Linux kernel module. Figure 6.14 shows the setup of that application in a case where there is no priority inversion. We can see two processes (**lowprio1** and **highprio0**) requiring the same resource while being pinned on different CPUs. The low priority process gets the resource first, and thus lets the high priority process waiting for it. If the low priority process is preempted by a higher priority one (**highprio1**), it cannot free the resource, letting the high priority process hung. The Linux kernel uses priority inheritance in such situations: the low priority process's priority is raised to the priority of the high priority process that waits for the resource, for the time needed to release the lock.

In some cases though, the priority of the high priority process that waits for the resource is not high enough to help the low priority process to take back control of the CPU. This is what is shown in Figure 6.15. In this case, we end up in a situation where a low priority process holds a lock needed by a higher priority one, and cannot release it as it is preempted.

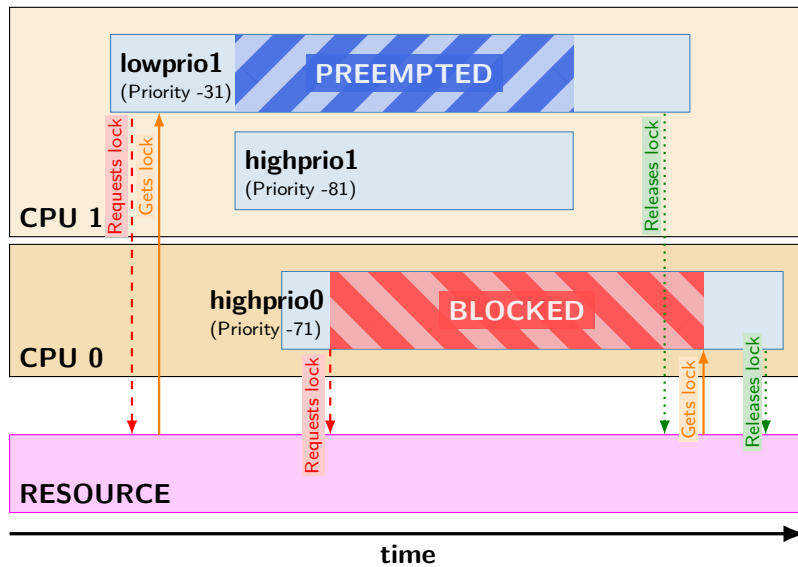


Figure 6.15 Schema of an in-kernel wake lock priority inversion: the **lowprio1** process takes the lock of the resource and cannot free it for the higher-priority **highprio0** process, because **lowprio1** is currently preempted by the **highprio1** process, which has a higher priority than **highprio0**

Table 6.3 Real-time priorities of the processes in the setup used to generate the trace for the in-kernel wakelock application. An empty priority means that the process was not started.

Process	Real-time priority			
	Setup 1	Setup 2	Setup 3	Setup 4
lowprio1		30	30	30
highprio1			80	80
highprio0	70	70	70	90

We traced our application while being run in different setups, in order to trace working and non-working scenarios. Table 6.3 shows these setups, as well as the priorities used for each process in each scenario. Each scenario is present once in the trace.

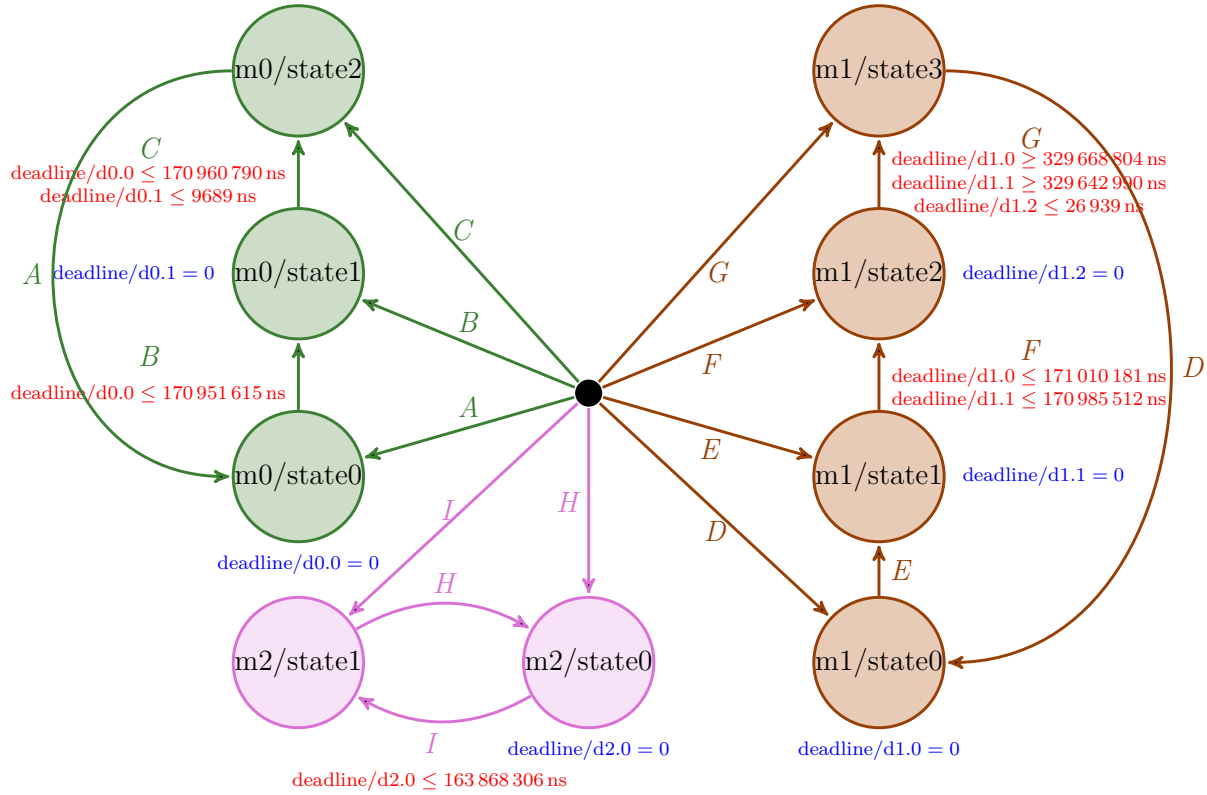
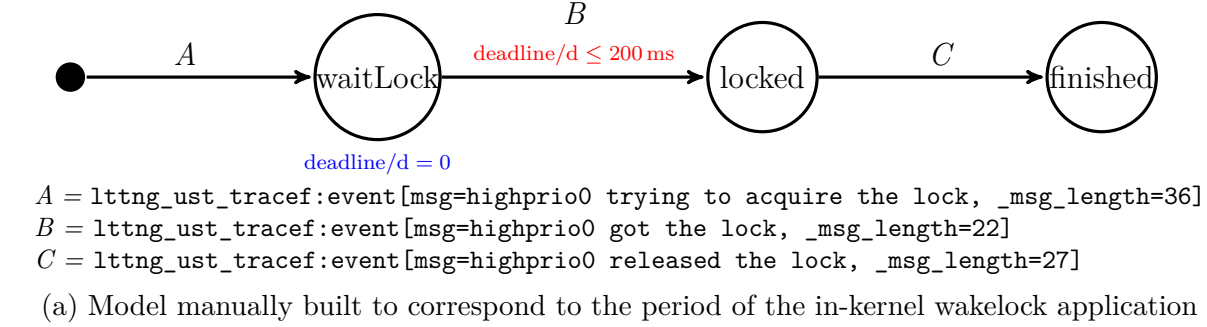
To detect the unwanted behaviors in the application, we built a model over the `highprio0` process, which is the one preempted by a lower priority one. We set a deadline for its execution at 3 ms, as shown in Figure 6.16(a). When using our algorithms, the model presented in Figure 6.16(b) was generated over the trace. We can see that the latter contains more states, which is expected since multiple processes are running in parallel and generate events. These states are organized in three different sets, each corresponding to a specific process. Therefore, the `highprio0` process is represented by the “m0/*” part of the model, the `lowprio1` process by the “m1/*” part, and the `highprio1` process by the “m2/*” part. If we look specifically at the “m0/*” part of the model, in order to compare it with our manually built model, we can see that the “m0/state0” state matches with the “waitLock” one, “m0/state1” with “locked”, and “m0/state2” with “finished”. Moreover, the constraint on variable “deadline/d0.0” in between states “m0/state0” and “m0/state1” corresponds to the constraint on variable “deadline/d” in the manually built model. Its value is also very close, with 170.951 ms where the manually set one was 200 ms.

Figure 6.17(a) presents the detection results using our manually built model. We can see that the priority inversion case is detected as the deadline is missed. The detection results using the automatically generated model are shown in Figure 6.17(b). The priority inversion case is also detected, thanks to the constraints on variable “deadline/d0.0”, at two places in the model. Those two detections are however highly linked, and detect the same deadline miss in between states “m0/state0” and “m0/state1”. These results thus show that the automatic generation of model-based constraints worked properly in this case, with a trace that correlates the work of multiple processes.

6.9 Running time and scalability

This section presents the timing results of our algorithms as well as their scalability according to multiple factors. All the tests presented were executed on a test system that consists of an Intel® Core™ i7-4790 CPU at 3.6 GHz, with 32 GiB of DDR3 RAM at 1600 MHz. Hyperthreading was disabled, as well as the idling and turbo modes of the CPU.

In those tests, the algorithms are parametrized with the values in (6.8).



(b) Model automatically generated from a runtime trace of the in-kernel wakelock application, using only deadline variables

Figure 6.16 Models to check the valid execution of the in-kernel wakelock application and detect unwanted behaviors

```

Received lttnng_ust_tracef:event[msg=highprio0 got the lock, _msg_length
↪ =22] at 21:40:36.197 389 304
  Entering state: locked
  Constraints:
    - deadline/d <= 200ms [INVALID] value: 329.5748ms

```

(a) Results of the detection on the wakelock application's trace using the model presented in 6.16(a), showing only the invalid cases

```

Received lttnng_ust_tracef:event[msg=highprio0 got the lock, _msg_length
↪ =22] at 21:40:36.197 389 304
  Entering state: m0/state1
  Constraints:
    - deadline/d0.0 <= 170951615.0000ns [INVALID] value: 329574797.0000ns
Received lttnng_ust_tracef:event[msg=highprio0 released the lock,
↪ _msg_length=27] at 21:40:36.197 398 993
  Entering state: m0/state2
  Constraints:
    - deadline/d0.0 <= 170960790.0000ns [INVALID] value: 329584486.0000ns

```

(b) Results of the detection on the wakelock application's trace using the model presented in 6.16(b), showing only the invalid cases of the “m0/*” part of the model

Figure 6.17 Results of the detection on the wakelock application's trace

$$\begin{aligned}
G_{\min} &= 50 \% \\
T_{\gg} &= 40 \% \\
T_{=} &= 80 \% \\
T &= 75 \%
\end{aligned}
\tag{6.8}$$

6.9.1 Running time

Table 6.4 Number of events and sizes of the traces used to measure our analysis running time

Case	Number of events			Size (MiB)
	UST	Kernel		
jackd	321	419 164	3.144	25.09
cyclictest	41 677	208 489	12.90	21.63
wakelock	42	194 997	1.605	11.92

We use the different cases discussed in section 6.8 to present the running time of our analysis on real situations. The traces used for each of those cases are presented in Table 6.4. We have seen previously that all the cases are different, but we can also see that their traces are different in terms of duration and number of events, on both the kernel and userspace sides.

Table 6.5 presents the benchmark results of the different steps involved in the model building process, i.e. the model building step that determines the workflow and generates a full model with adaptive constraints, the treatment of the adaptive constraints after reading the trace, and the clean up of the model once adaptive constraints have been completed.

We can observe that the processing of adaptive constraints does not take much time in each case, and that the clean up part is not consuming much time either. The latter is easily explained by the size of the generated models, and the fact that cleaning up the model is done using only data stored in memory.

We can also see in each case that the model building step is the longest. For **cyclictest**, this step takes around 7 min to complete. This can be explained by the number of userspace events in the trace, as well as by their distribution. Indeed, there are only 4 different threads running, with each generating around 10 000 events, for a workflow represented by only 2 events.

Table 6.5 Statistics on the execution duration of the different parts of our model building process, computed with 100 runs of the analysis for each case studied in 6.8

Step	Execution duration (in ms)		
	JACK2	cyclictest	wakelock
Model building			
<i>Minimum</i>	29.71	378 800	20.72
<i>Maximum</i>	65.53	444 000	29.06
<i>Average</i>	35.44	423 900	22.81
<i>St. Dev.</i>	4.712	14 440	1.783
Adaptive constraints			
<i>Minimum</i>	9.165	130.9	8.030
<i>Maximum</i>	11.05	174.8	9.336
<i>Average</i>	10.31	139.0	8.531
<i>St. Dev.</i>	0.341 1	7.942	0.279 4
Clean up			
<i>Minimum</i>	0.454 0	0.541 9	1.517
<i>Maximum</i>	0.966 9	0.704 7	1.784
<i>Average</i>	0.683 9	0.646 1	1.637
<i>St. Dev.</i>	0.137 2	0.030 88	0.052 82

Table 6.6 Statistics on the execution duration of the different parts of our model building process for `cyclictest`, while using the whole trace or only a randomly selected 15 ms chunk of it, computed with 100 runs of the analysis

Step	Execution duration (in ms)	
	Full trace	15 ms range
Model building		
<i>Minimum</i>	378 800	208.5
<i>Maximum</i>	444 000	258.6
<i>Average</i>	423 900	227.7
<i>St. Dev.</i>	14 440	11.33
Adaptive constraints		
<i>Minimum</i>	130.9	79.10
<i>Maximum</i>	174.8	138.1
<i>Average</i>	139.0	117.0
<i>St. Dev.</i>	7.942	18.87
Clean up		
<i>Minimum</i>	0.541 9	0.556 7
<i>Maximum</i>	0.704 7	0.760 8
<i>Average</i>	0.646 1	0.678 2
<i>St. Dev.</i>	0.030 88	0.035 88

Table 6.6 compares the execution time to run our algorithms over the full `cyclictest` trace, or only over a randomly selected 15 ms interval in the trace. We can see by those results that the model building duration can be heavily shortened when limiting the number of events to analyze. The treatment of adaptive constraints still takes around the same time, which is explained by the fact that the full trace is still analyzed to specify constraints validity at each instance step. The clean up is also executed on the same final model, which justifies the absence of change in its execution duration.

6.9.2 Scalability

To test and verify the scalability of our approach, we created a simple program to generate traces with userspace events in the form of simple model workflows. This program takes four different parameters: the number of states to generate for each workflow, the number of different workflows to emulate, the number of instances of workflows per thread, and the number of threads to be run. We ran this program while varying each of these parameters one by one, and traced its execution. We then ran our algorithms on the generated traces with one state system free variable, two counters and one timer activated. While running our algorithms, we benchmarked the execution duration to determine and build the model, to process the adaptive constraints after the first detection pass, and the clean up of the model unuseful constraints and variables. In each case, the generated models were as expected from the application execution.

Figure 6.18 shows the evolution of the execution duration according to the number of states in the model workflow. We can see that with more events, the execution duration is longer. This is explained by the fact that we build a suffix tree to find the longest non-repeated substring, which build time is directly proportional to the number of states. Also, more states means more variables and constraints to set, hence the increasing execution time not only to build the model but also to process adaptive constraints and clean up.

Figure 6.19 shows the execution duration according to the number of instances of the same workflow ran on the same thread. We can see that the time to build the model is also proportional to the number of instances of the same model found in the trace. In these traces, all of the instances of the model are generated by the same thread, putting them end-to-end. This thus can also be explained by the suffix tree, which needs to be generated for the whole sequence, in order to identify the researched workflow. The time to process adaptive constraints also increases, which is related directly to the number of existing instances of the model that have to be updated as valid, invalid or uncertain. Finally, the clean up time can

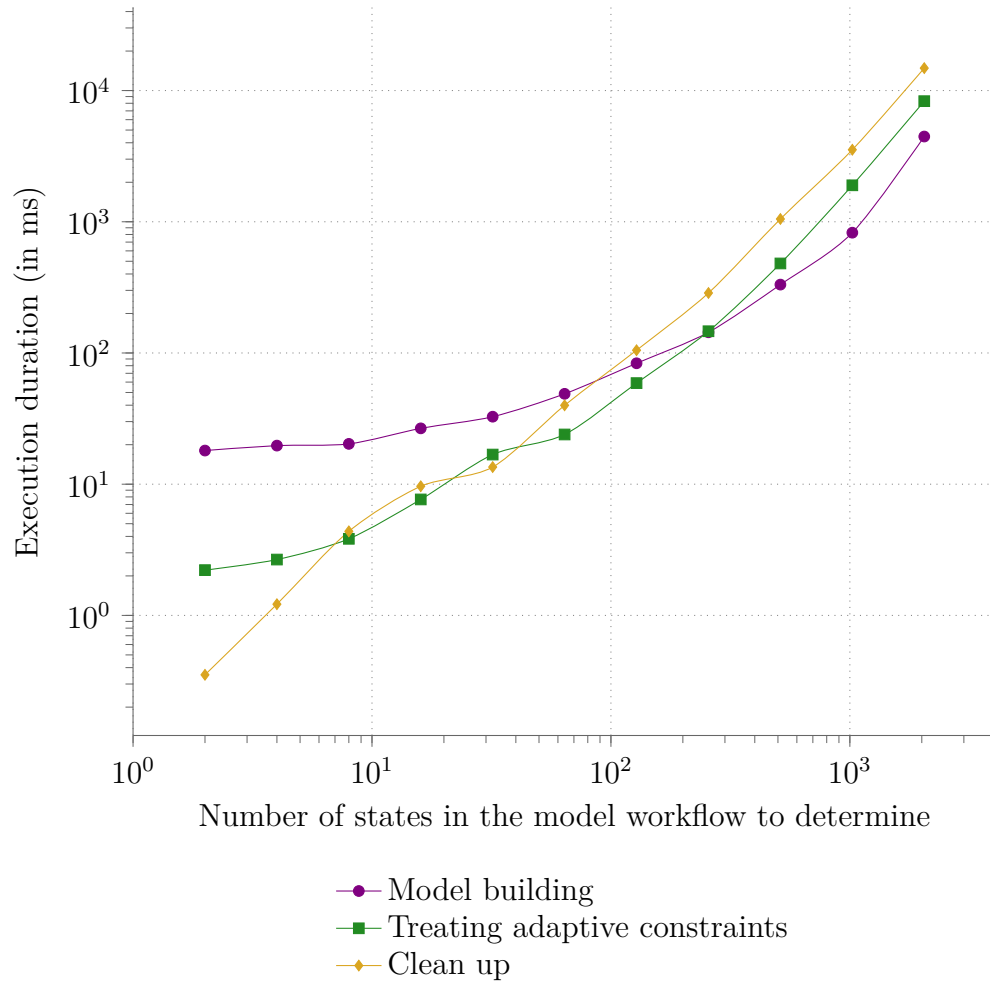


Figure 6.18 Execution duration of the different steps of our approach according to the number of states of the model to determine; all the runs use 1 instance per thread, 1 thread, and 1 workflow; each data point is the average of 20 runs

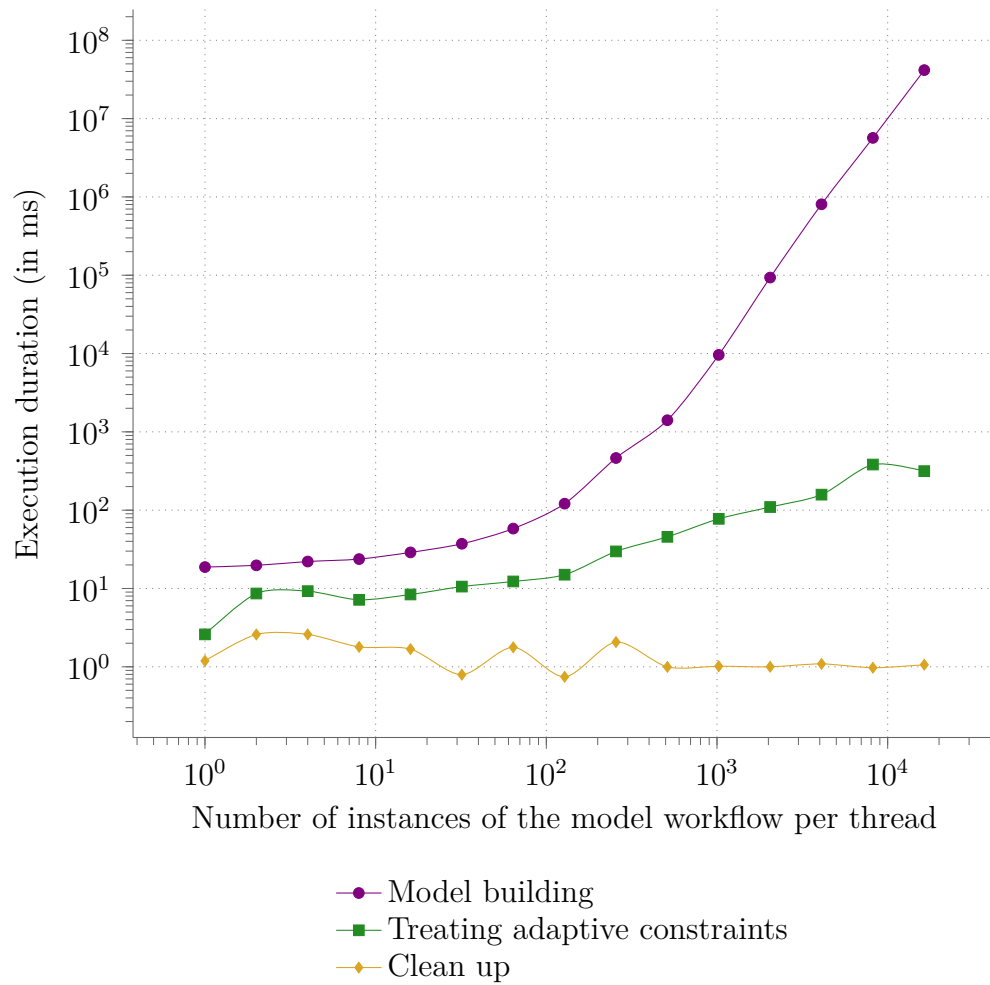


Figure 6.19 Execution duration of the different steps of our approach according to the number of instances of the model to determine; all the run use 4 states per instance, 1 thread, and 1 workflow; each data point is the average of 20 runs

be seen as constant, despite the increasing number of instances, which is expected as the model that has to be cleaned up is the same no matter the number of instances in the trace.

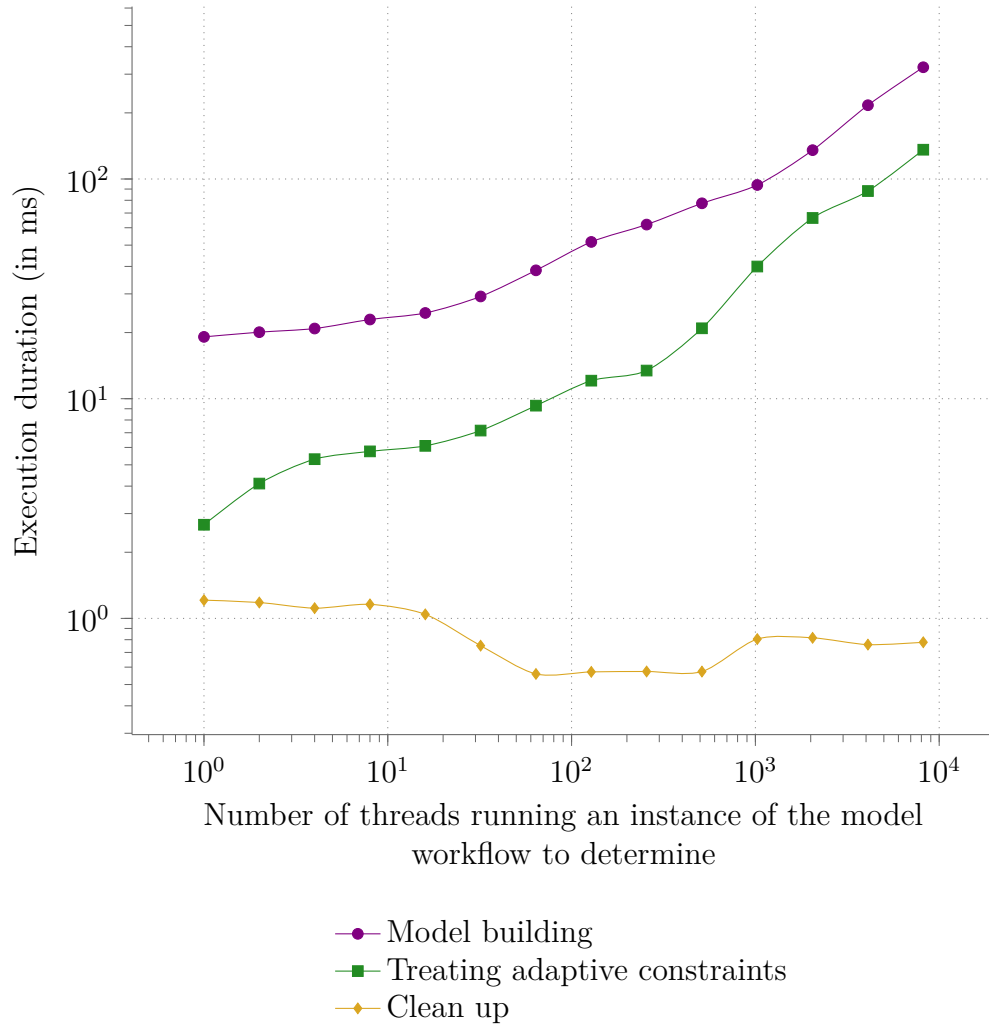


Figure 6.20 Execution duration of the different steps of our approach according to the number of threads running each instance of the model to build; all the runs use 4 states per instance, 1 instance per thread, and 1 workflow; each data point is the average of 20 runs

Figure 6.20 shows the influence of the number of threads running an instance of the workflow to build. We can see that the model building execution time increases with the number of threads, which is explained by the increasing number of workflows to merge together. The time to process the adaptive constraints also increases, as there is an increasing number of instances (one instance per thread). Once again, the clean up time is constant as the final generated model is the same no matter the number of threads.

Figure 6.21 shows the relation between the execution time and the number of different workflows to identify in the trace. We can see that the model build time is proportional to the

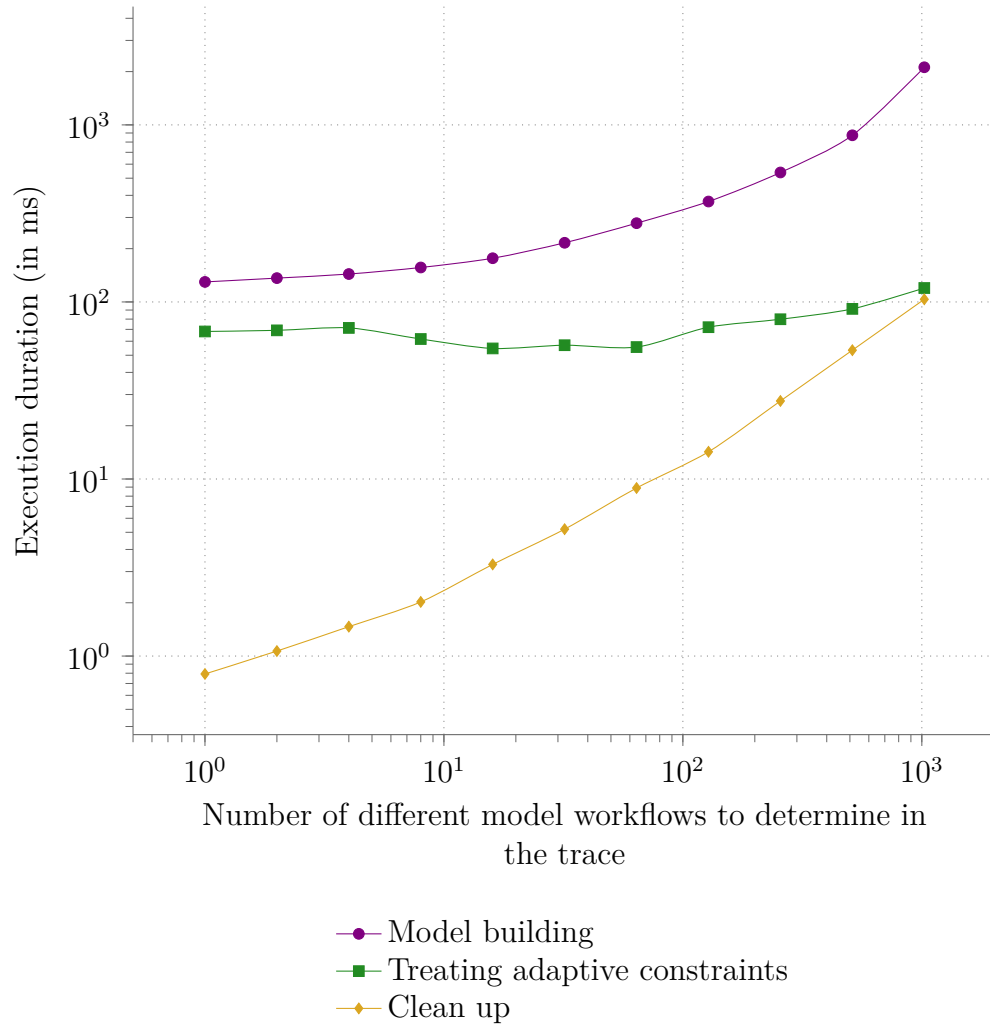


Figure 6.21 Execution duration of the different steps of our approach according to the number of different model workflows to build; all the runs use 4 states per instance, 1 instance per thread, and 2048 threads; each data point is the average of 20 runs

number of different workflows to be determined using the trace. The processing of adaptive constraints is fairly constant in this case, as the number of threads and instances per thread is constant, no matter the number of different workflows. The clean up time increases however with the number of workflows, which is expected as the more workflows, the bigger the generated model is.

All those tests confirm that our algorithms can be executed in time proportional to the square of the number of states to be determined, the square of the number of instances of the model workflow per thread, the number of threads running an instance and the number of different model workflows. However, the time taken to process the adaptive constraints is directly linked to the number of instances of the workflow in the trace, while the clean up time is related to the size of the generated model.

6.10 Conclusion and Future Work

We proposed a new approach to automatically build models of the workflow of applications, augmented with model-based constraints. These models are built using kernel and userspace traces of the application execution, but can be built whether or not invalid instances are present in the trace. We presented a brief reminder of the model representation that was introduced in previous work. We then explained how we determine the applications workflows in the trace, and how the model is built. We presented the concept of adaptive constraints that can have an adaptive comparison value and/or operator, explained how we proceed to compute the missing comparison elements, and how constraints are set. We finally characterized the execution time as well as the scalability of our implementation, as a function of the number of states in the workflow, the number of instances, the number of threads and the number of different workflows.

This approach is an important step to automate performance analysis and problem detection, as the user intervention is further reduced to the check and correction of the generated model. It is remarkable that meaningful models and constraints could be obtained for real use cases simply by identifying repeated patterns and their associated timing in traces. This is explained in large part by the fact that many real-time applications have a fairly simple main processing loop. This thus allows to save time and make tracing more accessible to people without a deep system knowledge. For more complex modeling requirements, which we have not encountered in our study of actual real-time systems, the possibility to modify, enhance and thus correct the automatically built model remains. As future work, it would be interesting to extend the proposed approach by allowing for on-the-fly model building and detection, which would be compatible with the flight-recorder mode of LTTng.

6.11 Acknowledgments

This research is supported by Ericsson, EfficiOS, and the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant CRD-468687-14.

CHAPTER 7 COMPLEMENTARY RESULTS

Chapter 5 presents the process used to analyze the root cause of a detected problem. Chapter 6 introduces our automated model building approach, and shows that the generated models allow to detect the unexpected behaviors. However, we did not present the analysis results that would be shown to the user using both approaches at the same time. In this chapter, we thus will present those results according to the different cases studied, and compare them to the results presented in Chapter 5 with manually built models.

7.1 JACK2

We ran our analysis to generate a model for **JACK2**, as was done in 6.8.1. However, we only asked for deadline variables in order to reduce the verbosity of the results, as we aim at comparing the results of that run to those presented in 5.6.1. The model built automatically is thus identical to the one presented in Figure 6.9(b), except for the constraints and variables being only the “deadline/”-prefixed ones.



Figure 7.1 shows the results of the analysis triggered by the infringement of the “deadline/d0.1” constraint when entering state “m0/state0”. This constraint corresponds to the one set when building manually the model. If we compare this analysis report to the one presented in Figure 5.12, we can see that all the steps taken by our analysis are the same.

Figures 7.2 and 7.3 show other parts of the generated analysis report for other constraints of the model. We can see that those constraint violations were actually linked to the same problem.

Only Figure 7.4 shows different results for the state analysis. This report is generated because, at one point, it took longer to move from state “m0/state0” to “m0/state1”. The constraint on that transition is a constraint exactly outside of the period we aimed at checking, as the period it checks represents the time during which **JACK2** runs before starting to wait for the next period. By looking further in the trace, to understand why at one point that period took longer, we identified that it was because of the deadline miss and the **JACK2** error reporting process (*xrun* identification and log).

Most of the results obtained were thus about the same problem, detected with our manually built model. Moreover, we have determined that the only result, that was slightly different from the others, was still related to the same execution problem. Therefore, in the case


General computed difference between invalid and valid instances:

State	Responsibility for added time
WAKING	96.66% 
BLOCKED on poll	3.29% 
RUNNING	0.03%
SYSCALL ioctl	0.02%
SYSCALL write	0.01%
SYSCALL poll	0.00%
SYSCALL futex	0.00%
Interrupted by IRQ29 (snd_hda_intel)	0.00%

Minimum responsibility for a case to be considered: 64.83%

(a) Results of the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
jackd PREEMPTED	96.51% 
swapper/2 RUNNING	1.88%
swapper/2 PREEMPTED	1.55%
jackd RUNNING	0.05%
irq/29-snd_hda_ PREEMPTED	0.01%
irq/29-snd_hda_ RUNNING	0.00%

Minimum responsibility for a case to be considered: 60.79%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 0
Analyzed timerange: [17:54:37.711 896 155, 17:54:38.319 519 477]



Per-TID Usage	Process	Migrations	Priorities
96.34% 	cpuburn (11371)	0	[-61]
0.29%	bash (11375)	1	[20]
0.13%	bash (11376)	2	[20]
0.11%	bash (11377)	0	[20]
0.10%	/usr/bin/x-term (3154)	2	[]

Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUtop analysis, triggered by the critical path analysis

Figure 7.1 Sections of the analysis report generated for JACK2, for the violation of the constraint on “deadline/d0.1” when entering state “m0/state0”



General computed difference between invalid and valid instances:

State	Responsibility for added time
WAKING	96.66% 
BLOCKED on poll	3.29% 
RUNNING	0.03%
SYSCALL ioctl	0.02%
SYSCALL write	0.01%
SYSCALL poll	0.00%
SYSCALL futex	0.00%
Interrupted by IRQ29 (snd_hda_intel)	0.00%

Minimum responsibility for a case to be considered: 64.83%

(a) Results of the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
jackd PREEMPTED	96.64% 
swapper/2 RUNNING	1.82%
swapper/2 PREEMPTED	1.49% 
jackd RUNNING	0.05%
irq/29-snd_hda RUNNING	0.01%
irq/29-snd_hda_ PREEMPTED	0.00%

Minimum responsibility for a case to be considered: 60.87%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 0
Analyzed timerange: [17:54:37.711 896 155, 17:54:38.319 519 477]



Per-TID Usage	Process	Migrations	Priorities
96.34% 	cpuburn (11371)	0	[-61]
0.29%	bash (11375)	1	[20]
0.13%	bash (11376)	2	[20]
0.11%	bash (11377)	0	[20]
0.10%	/usr/bin/x-term (3154)	2	[]

Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUPtop analysis, triggered by the critical path analysis

Figure 7.2 Sections of the analysis report generated for JACK2, for the violation of the constraint on “deadline/d0.1” when entering state “m0/state1”



General computed difference between invalid and valid instances:

State	Responsibility for added time
WAKING	96.65% 
BLOCKED on poll	3.30% 
RUNNING	0.03%
SYSCALL ioctl	0.02%
SYSCALL write	0.01%
SYSCALL poll	0.00%
SYSCALL futex	0.00%
Interrupted by IRQ29 (snd_hda_intel)	0.00%

Minimum responsibility for a case to be considered: 64.83%

(a) Results of the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
jackd PREEMPTED	96.51% 
swapper/2 RUNNING	1.88% 
swapper/2 PREEMPTED	1.55%
jackd RUNNING	0.05%
irq/29-snd_hda_ PREEMPTED	0.01%
irq/29-snd_hda_ RUNNING	0.00%

Minimum responsibility for a case to be considered: 60.79%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 0
Analyzed timerange: [17:54:37.711 896 155, 17:54:38.319 519 477]



Per-TID Usage	Process	Migrations	Priorities
96.34% 	cpuburn (11371)	0	[-61]
0.29%	bash (11375)	1	[20]
0.13%	bash (11376)	2	[20]
0.11%	bash (11377)	0	[20]
0.10%	/usr/bin/x-term (3154)	2	[]

Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUtop analysis, triggered by the critical path analysis

Figure 7.3 Sections of the analysis report generated for JACK2, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state0”

General computed difference between invalid and valid instances:

State	Responsibility for added time
RUNNING	74.60% 
Interrupted by HRTIMER	25.40% 

Minimum responsibility for a case to be considered: 64.60%


Figure 7.4 State analysis generated for JACK2, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”

of JACK2, our automated analysis, using an automatically built model, still gives accurate results to the user on the origin of the unexpected behavior.

7.2 cyclicttest

In order to compare the analysis results for `cyclicttest` with an automatically built model, or the manually built one, we ran our analysis to generate a model as was done in 6.8.2. The generated model was thus the same as shown in Figure 6.12(b). Those results can then be compared to those obtained in 5.6.5.


General computed difference between invalid and valid instances:

State	Responsibility for added time
BLOCKED on rt_sigtimedwait	99.85% 
SYSCALL rt_sigtimedwait	0.10%
RUNNING	0.03%
SYSCALL getcpu	0.01%
SYSCALL clock_gettime	0.01%

Minimum responsibility for a case to be considered: 59.93%

(a) Results of the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
ktimersoftd/3 PREEMPTED	99.69% 
ktimersoftd/3 RUNNING	0.24%
cyclicttest PREEMPTED	0.04%
cyclicttest RUNNING	0.03%

Minimum responsibility for a case to be considered: 56.57%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 3
Analyzed timerange: [21:55:53.528 529 561, 21:55:53.536 012 105]

Per-TID Usage	Process	Migrations	Priorities
99.73% 	irq/154-hpd (162)	0	[-51]
0.26%	irq/25-54200000 (163)	0	[-51]
0.01%	ktimersoftd/3 (32)	0	[-2]
0.00%	irq/22-Tegra PC (160)	0	[-51]
0.00%	irq/388-eth0 (611)	0	[-51]

Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUtop analysis, triggered by the critical path analysis

Figure 7.5 Sections of the analysis report generated for `cyclicttest`, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”

The sections of the analysis report relating to the constraint set on variable “deadline/d0.0” when entering state “m0/state1” are shown in Figure 7.5. This constraint matches with the one we defined in the manually built model. When comparing those results to those presented in Figure 5.20, we can see that the analysis followed the same steps with the same conclusions.

General computed difference between invalid and valid instances:

State	Responsibility for added time
BLOCKED on rt_sigtimedwait	98.07%
RUNNING	0.85%
BLOCKED on clock_gettime	0.39%
SYSCALL write	0.32%
SYSCALL futex	0.22%
SYSCALL rt_sigtimedwait	0.10%
SYSCALL clock_gettime	0.03%
SYSCALL getcpu	0.02%
SYSCALL timer_getoverrun	0.01%

Minimum responsibility for a case to be considered: 67.32%

(a) Results of the state analysis

General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
ktimersoftd/3 PREEMPTED	98.12%
cyclicttest RUNNING	1.42%
ktimersoftd/3 RUNNING	0.23%
cyclicttest PREEMPTED	0.23%

Minimum responsibility for a case to be considered: 55.90%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 3
Analyzed timerange: [21:55:53.528 529 561, 21:55:53.536 012 105]

Per-TID Usage	Process	Migrations	Priorities
99.73%	irq/154-hpd (162)	0	[-51]
0.26%	irq/25-54200000 (163)	0	[-51]
0.01%	ktimersoftd/3 (32)	0	[-2]
0.00%	irq/22-Tegra PC (160)	0	[-51]
0.00%	irq/388-eth0 (611)	0	[-51]


Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUPtop analysis, triggered by the critical path analysis

Figure 7.6 Sections of the analysis report generated for `cyclicttest`, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state0”

Figures 7.6 and 7.7 present other sections of the analysis report, related to two other constraints set in the automatically built model. We can observe that, even if these are different constraints, the analysis took the same path as in Figure 7.5 to determine the origin of the


General computed difference between invalid and valid instances:

State	Responsibility for added time
BLOCKED on rt_sigtimedwait	99.78% 
SYSCALL rt_sigtimedwait	0.10%
RUNNING	0.08%
SYSCALL clock_gettime	0.02%
SYSCALL getcpu	0.01%
SYSCALL timer_getoverrun	0.00%

Minimum responsibility for a case to be considered: 62.61%

(a) Results of the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
ktimersoftd/3 PREEMPTED	99.74% 
ktimersoftd/3 RUNNING	0.24%
cyclicttest PREEMPTED	0.02%

Minimum responsibility for a case to be considered: 52.78%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 3
Analyzed timerange: [21:55:53.528 529 561, 21:55:53.536 012 105]

Per-TID Usage	Process	Migrations	Priorities
99.73% 	irq/154-hpd (162)	0	[-51]
0.26%	irq/25-54200000 (163)	0	[-51]
0.01%	ktimersoftd/3 (32)	0	[-2]
0.00%	irq/22-Tegra PC (160)	0	[-51]
0.00%	irq/388-eth0 (611)	0	[-51]

Priorities of the PREEMPTED process during that interval: -2

(c) Results of the CPUPop analysis, triggered by the critical path analysis

Figure 7.7 Sections of the analysis report generated for `cyclicttest`, for the violation of the constraint on “deadline/d0.1” when entering state “m0/state1”

problem, and ended up with the same conclusion. Hence, those other constraints also allowed to detect the same execution problem, and pinpoint its root cause.

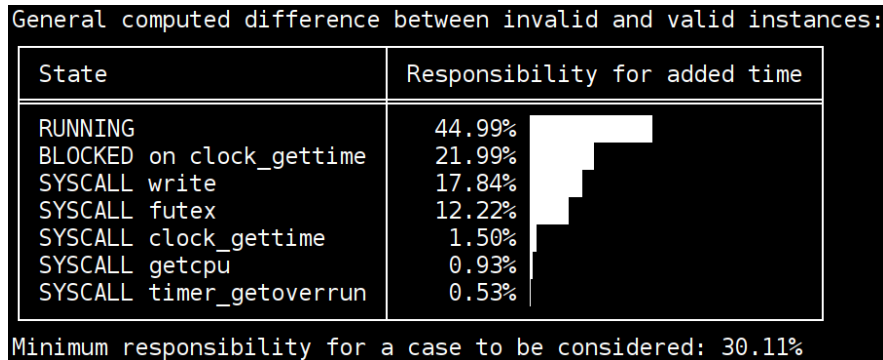


Figure 7.8 State analysis generated for `cyclicttest`, for the violation of the constraint on “deadline/d0.1” when entering state “m0/state0”

Figure 7.8 shows the state analysis related to the constraint on “deadline/d0.1” when entering state “m0/state0”. The period checked by this constraint is exactly outside of the execution period checked in our manually built model. We can see in the results of the state analysis that, during the unique invalid instance for that period, `cyclicttest` was running longer than usual, and doing a number of system calls, including some to know the current time, current CPU and write. When analyzing the trace at the time of this infringement, we saw that it happened directly following the principal constraint infringement. This was actually related to the generation of a “cyclicttest:outlier” trace event. This event was only to be generated when an outlier was identified during a cycle, which was the case here. Hence, this constraint infringement is also linked to the principal problem encountered during `cyclicttest` execution.


As for `JACK2`, we have seen that all the violations of the constraints that were set on our automatically built model were related to the same problem. Most of the results presented to the user pinpointed directly the root cause of the problem, while another identified the error reporting process of `cyclicttest`, following a cycle longer than expected. Therefore, these results show that the model automatically built by our analysis was able to help identify and understand the unexpected behavior appearing in this case.

7.3 In-kernel wakelock application

We followed the same process as previously for the in-kernel wakelock application. We used our analysis to automatically build the model of the application execution as was done

in 6.8.3. Using this model, identical to the one presented in Figure 6.16(b), the analysis provided us with an analysis report that we can compare to the results obtained in 5.6.2.

General computed difference between invalid and valid instances:

State	Responsibility for added time
BLOCKED on open	100.00% 
SYSCALL open	0.00%

Minimum responsibility for a case to be considered: 90.00%


(a) Results of the state analysis

```
Duration of active 'sched_pi_setprio'...
... in valid instances (maximum): 170.909us
... in invalid instances (average): 329.534us

Active 92.81% more time in invalid instances than in valid instances.
Verdict: Very high probability of a priority inversion
```

(b) Results of the priority inheritance analysis, triggered by the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
lowprio1 PREEMPTED	100.00% 

Minimum responsibility for a case to be considered: 100.00%

(c) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 1
Analyzed timerange: [03:40:35.867 825 012, 03:40:36.033 045 371]

Per-TID Usage	Process	Migrations	Priorities
99.18% 	highprio1 (26408)	0	[-100, -81]
0.82%	lowprio1 (26407)	0	[-71]
0.00%	lttng-consumerd (26369)	0	[20]
0.00%	lttng-consumerd (26370)	0	[20]
0.00%	Cache2 I/O (3011)	0	[20]


Priorities of the PREEMPTED process during that interval: -71

(d) Results of the CPUtop analysis, triggered by the critical path analysis

Figure 7.9 Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state1”

Figure 7.9 presents the sections of the analysis report related to the constraint set on “deadline/d0.0” when entering state “m0/state1”. This constraint corresponds to the one we set in the manually built model, and is located on model 0 (“m0”), which matches with the `highprio0` process. If we compare these results to those presented in Figure 5.15, we can see that the same steps and conclusions are made by the analysis.

General computed difference between invalid and valid instances:

State	Responsibility for added time
BLOCKED on open	100.00% 
SYSCALL write	0.00%
SYSCALL open	0.00%

Minimum responsibility for a case to be considered: 52.86%

(a) Results of the state analysis


```
Duration of active 'sched_pi_setprio'...
... in valid instances (maximum): 170.909us
... in invalid instances (average): 329.534us

Active 92.81% more time in invalid instances than in valid instances.

Verdict: Very high probability of a priority inversion
```

(b) Results of the priority inheritance analysis, triggered by the state analysis


General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
lowprio1 PREEMPTED	100.00% 

Minimum responsibility for a case to be considered: 100.00%

(c) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 1
Analyzed timerange: [03:40:35.867 825 012, 03:40:36.033 045 371]

Per-TID Usage	Process	Migrations	Priorities
99.18% 	highprio1 (26408)	0	[-100, -81]
0.82%	lowprio1 (26407)	0	[-71]
0.00%	lttng-consumerd (26369)	0	[20]
0.00%	lttng-consumerd (26370)	0	[20]
0.00%	Cache2 I/O (3011)	0	[20]



Priorities of the PREEMPTED process during that interval: -71

(d) Results of the CPUTop analysis, triggered by the critical path analysis

Figure 7.10 Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d0.0” when entering state “m0/state2”

Figure 7.10 shows the report for another constraint still applying on model 0. We can see that the steps and conclusions are also related to the same problem, and pinpoint the same origin: the fact that the `lowprio1` process was preempted by the `highprio1` process, and that the raised priority did not suffice to take back the CPU.


General computed difference between invalid and valid instances:

State	Responsibility for added time
Interrupted by HRTIMER SYSCALL open	51.72%  48.28% 

Minimum responsibility for a case to be considered: 48.28%

(a) Results of the state analysis



General computed difference between invalid and valid instances:

Critical path state	Responsibility for added time
lowprio1 PREEMPTED	100.00% 

Minimum responsibility for a case to be considered: 100.00%

(b) Results of the critical path analysis, triggered by the state analysis

Analyzed CPUs: 1
Analyzed timerange: [03:40:35.867 796 094, 03:40:36.033 045 371]

Per-TID Usage	Process	Migrations	Priorities
99.18% 	highprio1 (26408)	0	[-100, -81]
0.82% 	lowprio1 (26407)	0	[-100, -71, -31]
0.00%	lttng-consumerd (26369)	0	[20]
0.00%	lttng-consumerd (26370)	0	[20]
0.00%	Cache2 I/O (3011)	0	[20]

Priorities of the PREEMPTED process during that interval: -71, -31

(c) Results of the CPUPop analysis, triggered by the critical path analysis

Figure 7.11 Sections of the analysis report generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.0” when entering state “m1/state3”

Figures 7.11 and 7.12 present the parts of the analysis report that concern the constraints on “deadline/d1.0” and “deadline/d1.1” when entering state “m1/state3”. These constraints are related to the model 1, which matches with the `lowprio1` process. These constraints say that it should take at least 329 668 804 ns and 329 642 990 ns to move respectively from state “m1/state0” and “m1/state1”, where the variables are initialized, to “m1/state3”. However, they are invalid once in the trace, because of one of the setups used to generate the trace, as presented in Table 6.3: at some point, only `lowprio1` and `highprio0` are started, which means `highprio1` is not preempting `lowprio1`. The constraint is then set using this instance as anomaly, and this is what the analysis report is showing for that infringement.

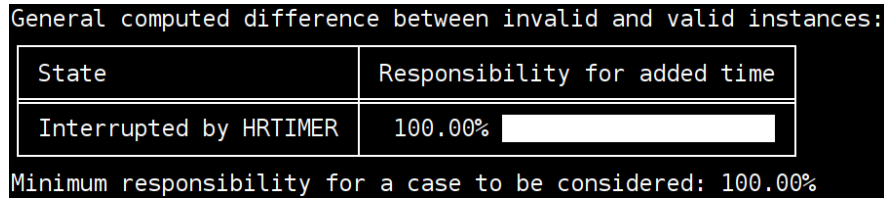


Figure 7.12 State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.1” when entering state “m1/state3”

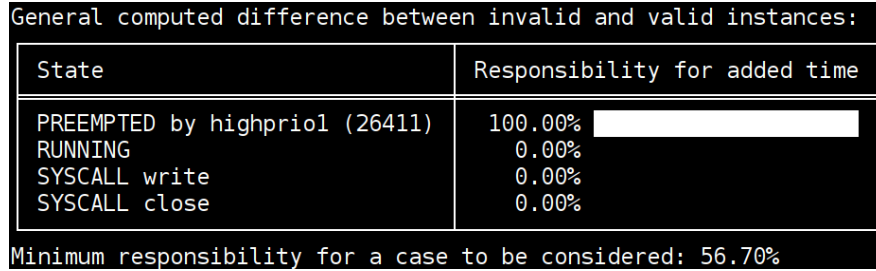



Figure 7.13 State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.2” when entering state “m1/state3”

The constraint on “deadline/d1.2” when entering state “m1/state3” is also invalid once during the trace. Figure 7.13 shows the state analysis for that constraint. We can see that between states “m1/state2” and “m1/state3”, at one point, the application took too much time. We can also identify that this discrepancy was caused by the fact that the application was preempted by `highprio1` for the invalid case. However, by looking further at the trace, the instance identified as invalid for that constraint is actually the last setup during which `lowprio1` inherits `highprio0`’s priority of 90. The fact that this was considered as an irregularity by the automatic model building process can easily be explained. Indeed, the event triggering the transition to “m1/state2” happens just before `lowprio1` releases the lock, while the one triggering the transition to “m1/state3” appears just after. In the case where `highprio1` was not running, once `lowprio1` released the lock, it could continue unstopped its execution, thus generating the event triggering the transition to “m1/state3”. In the next case, where `highprio0`’s priority was not sufficient, `lowprio1` was only able to restore its activity, and thus release the lock, once `highprio1` had already finished its execution. The two events could then be generated without any problem. Yet, in the last case, `lowprio1` can resume its activity thanks to the raised priority. This priority raise is temporary, though, and `lowprio1` needs to restore back its priority once it releases the locked resource wanted by the higher priority process. It means that, as soon as the resource is released, `lowprio1` is once again preempted by `highprio1` until this latter finishes its execution. The event triggering

the transition to “m1/state2” was thus generated before this preemption, while the other will only appear once `lowprio1` gets back the CPU. This is thus what the analysis report shows, explaining that in the invalid case, `lowprio1` was preempted by `highprio1`.


General computed difference between invalid and valid instances:

State	Responsibility for added time
PREEMPTED by highprio1 (26408)	99.99% 
Interrupted by SOFTIRQ_TIMER	0.01%
Interrupted by HRTIMER	0.00%
SYSCALL open	0.00%

Minimum responsibility for a case to be considered: 56.69%

Figure 7.14 State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.0” when entering state “m1/state2”

General computed difference between invalid and valid instances:

State	Responsibility for added time
PREEMPTED by highprio1 (26408)	99.99% 
Interrupted by SOFTIRQ_TIMER	0.01%
Interrupted by HRTIMER	0.00%


Minimum responsibility for a case to be considered: 52.86%

Figure 7.15 State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d1.1” when entering state “m1/state2”

Figures 7.14 and 7.15 show the parts of the analysis report that relate to the constraint on variables “deadline/d1.0” and “deadline/d1.1” when entering state “m1/state2”. These constraints specify a duration limit, respectively between the states “m1/state0” and “m1/state1”, to the state “m1/state2”. As the comparison value is really close for both, we can see that the period really constrained is between states “m1/state1” and “m1/state2”, which corresponds to the period where `lowprio1` got the lock, until it ends its busy loop. Those constraints are invalid once, and both at the same place, which is for the setup during which `highprio0`’s priority is not sufficient. We can see in the results that the problem is that `lowprio1` is being preempted by `highprio1`, which is the main cause of the constraints violations. This is actually the problem that we detected while constraining the `highprio0` process.

The last constraint for which we had a report was on “deadline/d2.0” when entering state “m2/state1”. This constraint is thus in model 2, which corresponds to the `highprio1` process. We can see in Figure 7.16 that when the constraint was invalid, it was because `highprio1` was preempted by `lowprio1` and thus took more time to run its busy loop. The instance

General computed difference between invalid and valid instances:

State	Responsibility for added time
PREEMPTED by lowprio1 (26410)	99.57% 
RUNNING	0.42%
Interrupted by SOFTIRQ_RCU	0.01%
Interrupted by HRTIMER	0.00%
SYSCALL write	0.00%
Interrupted by SOFTIRQ_TIMER	0.00%

Minimum responsibility for a case to be considered: 62.49%

Figure 7.16 State analysis generated for the in-kernel wakelock application, for the violation of the constraint on “deadline/d2.0” when entering state “m2/state1”

considered as invalid is actually the last case, where `lowprio1`’s priority was raised to the priority of `highprio0`, higher than `highprio1`’s in this setup.

As we have seen with that analysis of the reports generated for each constraint, a lot of those still allow to detect and pinpoint the origin of the actual problem, that the manually built model helped to understand. However, some other constraints are noise in the results, but can easily be explained by the content of the trace fed to the analysis. Finally, the aim was to be able to automatically build models to detect irregular behaviors, and these results show that the expectations are also reached for this case study.

CHAPTER 8 GENERAL DISCUSSION

This chapter recalls the elements identified as research objectives and explains, by construction, how we were able to meet them. This is followed by a discussion about our contributions and their impact to demonstrate the scientific and industrial impacts of this research. Finally, we will detail the limitations of the presented solutions.

8.1 Objectives achievement

In this section, we recall the specific research objectives to assess the extent to which they were achieved.

Validate the runtime execution and latency constraints. Using a state machine representation of the application workflow, we were able to follow its runtime execution. The different states of the state machine represent intermediary states in the application, located between the tracepoints. The transitions of the model, along with the initial transition entering the first states, are triggered when receiving specific events leading to this transition. Multiple deadline variables can be set on each state, and checked at each transition. When setting a deadline variable, it will be initialized using the timestamp of the event that triggered the transition, with which we entered this state. When checking a constraint on a deadline variable, we will compare the expected value to the difference between the value of that variable and the timestamp of the event that triggered this transition. This methodology thus allows to efficiently validate the runtime execution and latency constraints of a real-time application by using userspace traces of its execution. This process is efficient as, once the model is known, it can be done entirely while reading the trace, ensuring a linear scalability to the number of events in the trace.

Use kernel traces to add other levels of constraints. The traces of what happened on the kernel side, while an application was running, can contain a lot of information about the system state and the execution of other applications. Extracting those informations is really interesting, as it can provide a way to put constraints that relate to the system in the models to be checked. In order to take advantage of this data, we built a state system by using the state attribute tree of **Trace Compass**. Our state system stores multiple informations on the different processes of the system and the system itself, through time, for the duration of the trace. This means that, when searching for the value of a given attribute at a given

timestamp, a request to the state system returns directly that value, without needing to read the trace again to compute it. The kernel trace thus only needs to be read once to build the state system, which is important for scalability. We then were able to characterize the kind of values that we would need to store, and to limit them to two categories: the counters and the timers. The counters exist to count, being incremented once each time specific events are reached. A unique request to the state system then allows to get the value of the counter at a given timestamp. The timers can measure the time spent. They are incremented from an amount of time at the end of the incrementing period. Two requests are then needed to get the value of the timer at a given timestamp, as we need to verify that the counter was not being incremented at the timestamp. Indeed, if the timer was being incremented but its value was not yet updated, we need to account for the current increment. With counters, we were able to store the number of preemptions and system calls for instance. With timers, we were able to store the CPU usage time, blocked time and preemption time of processes. It was then possible to add constraints to the model using variables of those types, and thus to take advantage of system information to verify the runtime execution of applications, thanks to kernel traces.

Pinpoint the origin of constraints violations. After reading the trace and matching the application execution to the model to verify, constraints can be valid or invalid. In order to understand what happened when a constraint was invalid, we used the data acquired from the cases where the same constraint was valid as point of comparison. As a runtime application can take multiple paths while still meeting its requirements, we developed an algorithm that allowed to match the invalid instances with the closest valid ones. These matches then allowed us to extract the differences between valid and invalid cases, and to use probabilities to understand what was the most likely cause of invalidity. Depending on the type of results, other analysis can be run to pinpoint even further the origin of the constraint violation. For instance, when a process is identified as being blocked in invalid situations, a critical path analysis is triggered, using the same comparison algorithm to identify the differences between valid and invalid cases. This approach requires further readings of the kernel trace, as it performs a deeper analysis than just the detection. However, as this part of the analysis is only done on randomly selected samples of valid and invalid instances, the scalability can be maintained. The results returned by that analysis process are clear and particularly accurate, allowing to clearly identify the origins of the constraint violations.

Automate the model building step. Most common real-time applications follow simple internal models. To further enhance tracing accessibility, we were able to provide the algo-

rithms and methodology to automatically build the general workflow of such applications, and augment it with the different available types of constraints. Our algorithms were built using the longest common sequence algorithm, as well as suffix trees to efficiently identify repeated sequences. Adaptive constraints, a new kind of constraint, were developed to adapt to the data read in the trace while limiting the impact on the scalability. Indeed, while usual constraints get a state of validity while encountered in the model, adaptive constraints will store the variable value. Once the trace has been fully read, adaptive constraints will compute their missing information, i.e. operator and/or value. This means that we do not need to read the trace again, but only to resolve the adaptive constraints. Moreover, an efficient segmentation approach is used to determine the right operator and value to use, while considering that some read values can be invalid. Our algorithms and methodology thus allow to infer a model augmented with at least the needed specifications, and thus to start the detection process without a user-provided model. When more complex modeling is required, which we did not encounter in our study of actual real-time applications, our approach still allows the user to modify, improve and correct the automatically built model.

8.2 Contributions

This section presents the scientific and industrial impacts of our research through our contributions.

Trace Compass. The analysis developed during this research was built as part of **Trace Compass**, and the aim is for it, along the related algorithms, to be fully integrated in the mainline version of the application. Moreover, while working on our algorithms, several contributions were made to **Trace Compass** in order to either fix bugs or improve it. Indeed, our use case of some **Trace Compass** internal APIs had not been seen previously, leading to unexpected behaviors that we needed to fix, or missing resources that we needed to access. All of these changes have been included in the mainline version of the application.

Real-time applications use cases. In order to develop algorithms that worked for common real-time situations, we needed to first survey those situations, then find ways to reproduce them, and finally generate traces of their execution. Our work thus provides descriptions on how to reproduce the surveyed problems. The source code developed for the needs of some of those cases are also accessible with open source licenses.

8.3 Limitations

The main limitation of the proposed approaches, to detect and analyze the common real-time problems, resides in the scalability of the data structure used to store and query the system state at a given timestamp. Indeed, even if state system free variables do not need to query the state system in order to be verified, the detailed analysis of the constraint violations causes requires kernel information. This means that the state system scalability is really important to insure a fast response time. Even if the State History Tree (SHT) performance have been proven [113], it still has scalability problems with large numbers of threads in the same trace. However, SHT is currently being improved to optimize building and query times [114].

Another limitation is related to the automated model generation, both in terms of the algorithms used and possibilities. Indeed, if the chosen approach allows to identify the simple internal loop of a real-time application, it does not provide a way to identify directly more complex models. However, in our study of actual real-time systems, we did not encounter applications that needed more complex modeling to represent their critical internal workflow, which is the part that we are interested in constraining and verifying.

CHAPTER 9 CONCLUSION AND RECOMMENDATIONS

9.1 Synthesis of work

This research improved the state of the art in the domain of real-time systems problem detection and analysis, with a focus on automated problem analysis. Our main achievement is the development of algorithms to detect and understand complex problems, causing abnormal behaviors, while keeping an analysis runtime of a few tens of seconds. Our contributions include:

- A methodology that allows to use userspace traces of real-time applications, and their workflow model, to validate their runtime execution and latency constraints;
- the algorithms and data structures to use kernel traces to add other levels of constraints, such as resources usage;
- the algorithms to pinpoint the origin of constraint violations from userspace and kernel traces and their workflow model;
- the algorithms and methodology to automate the model building step for common real-time applications by using userspace and kernel traces.

In conclusion, the proposed approach, that takes advantage of both userspace and kernel tracing, allows to detect easily and accurately unexpected behaviors in common real-time applications, in real execution situations. It also allows for a fast and accurate analysis of the root cause of such detected behaviors, providing the user with the different elements to look at. Moreover, we have shown that it is possible to build automatically the models required to run such analysis for actual real-time applications. The user thus only needs to provide the execution traces, when not familiar with the application workflow, making our approach even more accessible for non-expert users.

9.2 Recommendations

Future possible improvements of our work are divided into three areas: improvements of the detection, improvements of the analysis, and on-the-fly analysis.

For detection, we determined a number of constraints on system resources, but we did not define variables using all the resources available in the trace. Indeed, our work was primarily

to define a methodology to use those resources, with a focus on real-time situations. However, our methodology could be extended for other cases, and take advantage of more of the resources available in the kernel trace. Such resources, like the performance counters, could be useful when wanting to constrain the application execution, and allow to detect a higher number of unwanted behaviors.

On the analysis side, while we are now able to pinpoint automatically and accurately the root cause of detected problems, the user still needs to understand what changes could be done to the system or application parameters to fix these problems. It could be interesting to provide another analysis level which, using the analysis results, would advise the user on this aspect.

Finally, all of our analysis are done *a posteriori*. This means that our analysis can only be executed on traces that we already finished acquiring. This is directly related to the fact that we need to build the state system before being able to check our constraints. Using the LTTng tracer flight recorder mode, it would be interesting to provide an on-the-fly analysis. Indeed, this would allow for a direct detection of the problems. Moreover, when the underlying cause of such problem is identified, actions could be taken to adjust the system and application configurations.

REFERENCES

- [1] P. A. Laplante, *Real-Time Systems Design and Analysis*. IEEE, 1993.
- [2] D. Hildebrand, “An Architectural Overview of QNX.” in *USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992, pp. 113–126.
- [3] L. Fu and R. Schwebel. (2016, May) Real-time linux wiki. RT PREEMPT HOWTO. [Online]. Available: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO
- [4] R. Beamonte and M. R. Dagenais, “Detection of Common Problems in Real-Time and Multicore Systems Using Model-Based Constraints,” *Scientific Programming*, vol. 2016, Mar. 2016, article ID 9792462, acceptance rate: 9%.
- [5] —, “Model-based constraints over execution traces to analyze multi-core and real-time systems,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, to appear: submitted.
- [6] —, “Model-based constraints inference from runtime traces of common multicore and real-time applications,” *ACM Transactions on Software Engineering and Methodology*, to appear: submitted.
- [7] S. Goswami. (2005, Apr.) An introduction to KProbes. [Online]. Available: <http://lwn.net/Articles/132196/>
- [8] S. Rostedt. (2010, Mar.) Using the TRACE_EVENT() macro (Part 1). [Online]. Available: <http://lwn.net/Articles/379903/>
- [9] —. (2010, Mar.) Using the TRACE_EVENT() macro (Part 2). [Online]. Available: <http://lwn.net/Articles/381064/>
- [10] —. (2010, Apr.) Using the TRACE_EVENT() macro (Part 3). [Online]. Available: <http://lwn.net/Articles/383362/>
- [11] B. Brandenburg and J. Anderson, “Feather-trace: A light-weight event tracing toolkit,” in *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Jul. 2007, pp. 61–70.
- [12] B. Branderburg. (2007, Jul.) Complete OSPERT’07 modification: `ospert_complete.patch` (against Linux 2.6.20). [Online]. Available: http://www.cs.unc.edu/~bbb/feathertrace/files/ospert_complete.patch

- [13] A. R. Bernat and B. P. Miller, “Anywhere, any-time binary instrumentation,” in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, ser. PASTE ’11. New York, NY, USA: ACM, Sep. 2011, pp. 9–16. [Online]. Available: <http://doi.acm.org/10.1145/2024569.2024572>
- [14] J. Edge. (2009, Jul.) Perfcounters added to the mainline. [Online]. Available: <http://lwn.net/Articles/339361/>
- [15] M. Desnoyers. (2010, Aug.) A new unified Lockless Ring Buffer library for efficient kernel tracing. [Online]. Available: <http://www.efficios.com/pub/linuxcon2010-tracingsummit/presentation-linuxcon-2010-tracing-mini-summit.pdf>
- [16] J. Edge. (2009, Mar.) A look at ftrace. [Online]. Available: <http://lwn.net/Articles/322666/>
- [17] S. Rostedt. (2008) ftrace - Function Tracer. [Online]. Available: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>
- [18] J. Corbet. (2009, Jul.) Dynamic probes with ftrace. [Online]. Available: <http://lwn.net/Articles/343766/>
- [19] M. Hiramatsu. (2009, Aug.) Kprobe-based Event Tracing. [Online]. Available: <https://www.kernel.org/doc/Documentation/trace/kprobetrace.txt>
- [20] ——. (2010, Apr.) Dynamic Event Tracing in Linux Kernel. [Online]. Available: https://events.linuxfoundation.org/slides/lfcs2010_hiramatsu.pdf
- [21] S. Rostedt. (2010) Ftrace: Now and Then. [Online]. Available: <http://people.redhat.com/srostedt/trace-cmd-linuxcon-2010.odp>
- [22] R. McDougall, J. Mauro, and B. Gregg, *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*, ser. Oracle Solaris Series. Sun Microsystems Press/Prentice Hall, 2007.
- [23] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic Instrumentation of Production Systems,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247415.1247417>
- [24] F. C. Eigler, “Problem Solving With Systemtap,” in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, Jul. 2006, pp. 261–268.

- [25] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen, “Architecture of systemtap: a Linux trace/probe tool,” Jul. 2005. [Online]. Available: <https://sourceware.org/systemtap/archpaper.pdf>
- [26] J. Corbet. (2013, May) Ktap – yet another kernel tracer. [Online]. Available: <http://lwn.net/Articles/551314/>
- [27] J. Zhangwei. (2013, May) Ktap - A New Scripting Dynamic Tracing Tool For Linux. Presented at the LinuxCon Japan. [Online]. Available: https://events.linuxfoundation.org/sites/events/files/lcjpcojp13_zhangwei.pdf
- [28] ——. (2013, Jul.) ktap 0.2 released. [Online]. Available: <http://lwn.net/Articles/561568/>
- [29] ——. (2013, Aug.) ktap: enable scripting for Linux tracing subsystem. [Online]. Available: <http://lwn.net/Articles/562643/>
- [30] M. Desnoyers and M. R. Dagenais, “The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux,” in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, Jul. 2006, pp. 209–224.
- [31] M. Desnoyers, “Low-impact operating system tracing,” Ph.D. dissertation, École Polytechnique de Montréal, Québec, Canada, Dec. 2009.
- [32] LTTng. Documentation - LTTng. [Online]. Available: <http://lttng.org/docs/>
- [33] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, “User-Level Implementations of Read-Copy Update,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, Feb. 2012.
- [34] M. Desnoyers. (2011) Common Trace Format (CTF) Specifications. [Online]. Available: http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md
- [35] Applied Informatics Software Engineering GmbH. (2012) POCO C++ Libraries. [Online]. Available: <http://pocoproject.org/>
- [36] J. Corbet. (2007, Mar.) Introducing utrace. [Online]. Available: <http://lwn.net/Articles/224772/>
- [37] J. Keniston and S. Dronamraju. (2010, Apr.) Uprobes: User-Space Probes. [Online]. Available: http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf

- [38] J. Corbet. (2011, Mar.) Uprobes: 11th time is the charm? [Online]. Available: <http://lwn.net/Articles/433568/>
- [39] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, “Combined Tracing of the Kernel and Applications with LTTng,” in *Proceedings of the 2009 Linux Symposium*, Jul. 2009. [Online]. Available: <http://www.dorsal.polymtl.ca/sites/www.dorsal.polymtl.ca/files/publications/fournier-combined-tracing-ols2009.pdf>
- [40] J. Desfossez. (2011, Feb.) LTTng-UST vs SystemTap userspace tracing benchmarks. [Online]. Available: <https://sourceware.org/ml/systemtap/2011-q1/msg00244.html>
- [41] M. Desnoyers. (2011, Feb.) Re: LTTng-UST vs SystemTap userspace tracing benchmarks. [Online]. Available: <https://sourceware.org/ml/systemtap/2011-q1/msg00247.html>
- [42] R. Beamonte, F. Giraldeau, and M. Dagenais, “High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems,” in *Proceedings of the 14th Real-Time Linux Workshop*. OSADL, Oct. 2012.
- [43] R. Beamonte and M. R. Dagenais, “Linux Low-Latency Tracing for Multicore Hard Real-Time Systems,” *Advances in Computer Engineering*, vol. 2015, Aug. 2015, article ID 261094.
- [44] S. A. Ezust, “Tango: The Trace ANalysis GeneratOr,” Master’s thesis, McGill University, Québec, Canada, Apr. 1995.
- [45] S. A. Ezust and G. V. Bochmann, “An Automatic Trace Analysis Tool Generator for Estelle Specifications,” in *Proceedings of ACM SIGCOMM 95 Conference*, 1995, pp. 25–4.
- [46] X. Chen, H. Harry, F. Balarin, and Y. Watanabe, “Automatic Trace Analysis for Logic of Constraints,” in *Proceedings of the 40th annual Design Automation Conference*, ser. DAC ’03, ACM. New York, NY, USA: ACM, Jun. 2003, pp. 460–465. [Online]. Available: http://alumni.cs.ucr.edu/~xichen/pub/dac03_loc.pdf
- [47] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, “Automatic Generation of Simulation Monitors from Quantitative Constraint Formula,” in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE ’03. Washington, DC, USA: IEEE Computer Society, Mar. 2003, pp. 11174–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=789083.1022906>

- [48] F. Balarin, J. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A. Sangiovanni-Vincentelli, “Constraints specification at higher levels of abstraction,” in *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, 2001, pp. 129–133.
- [49] F. Wolf and B. Mohr, “Automatic performance analysis of hybrid MPI/OpenMP applications,” in *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, Feb. 2003, pp. 13–22.
- [50] —, “Hardware-Counter Based Automatic Performance Analysis of Parallel Programs,” in *Proceedings of the Minisymposium 'Performance Analysis', Conference on Parallel Computing (PARCO)*, Sep. 2003.
- [51] B. J. N. Wylie and B. Mohr, “KOJAK hardware counter performance analysis of parallel programs,” in *ScicomP 2005*, Edinburgh, Scotland, May 2005. [Online]. Available: <http://www.spscicomp.org/ScicomP11/Presentations/User/wylie.pdf>
- [52] J. Giménez, H.-F. Wen, F. Voigtländer, J. Labarta, D. Klepacki, and B. Mohr, “Scalable Performance Analysis combining Profile and Trace Tools,” Department of Computer Architecture, Universitat Politècnica de Catalunya, Tech. Rep., 2010.
- [53] V. Pillet, J. Labarta, T. Cortes, and S. Girona, “PARAVER: A Tool to Visualize and Analyze Parallel Code,” Department of Computer Architecture, Universitat Politècnica de Catalunya, Tech. Rep., 1995.
- [54] J. Giménez, J. Labarta, F. X. Pegenaute, H.-F. Wen, D. Klepacki, I.-H. Chung, G. Cong, F. Voigtländer, and B. Mohr, “Guided Performance Analysis Combining Profile and Trace Tools,” in *Euro-Par 2010 Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, M. R. Guarracino, F. Vivien, J. L. Träff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Martino, and M. Alexander, Eds. Springer Berlin Heidelberg, 2011, vol. 6586, pp. 513–521. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21878-1_63
- [55] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, “The Scalasca Performance Toolset Architecture,” *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 702–719, Apr. 2010.
- [56] B. Mohr, “Automatic Trace Analysis with Scalasca.” Jülich Supercomputing Center, 2012. [Online]. Available: <http://www.vi-hps.org/upload/projects/hopsa/hopsa-nov12-scalasca.pdf>

- [57] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Furlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, “Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications,” in *Proc. of the 2nd Parallel Tools Workshop, Stuttgart, Germany*. Springer, Jul. 2008, pp. 157–167.
- [58] Scientific Computing Center, Aristotle University of Thessaloniki. WebHome. [Online]. Available: <http://wiki.grid.auth.gr/>
- [59] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, and B. Mohr, “Automatic Trace-Based Performance Analysis of Metacomputing Applications,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, Mar. 2007, pp. 1–10.
- [60] T. M. Peiris and J. H. Hill, “Adapting System Execution Traces for Validation of Distributed System QoS Properties,” in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*. IEEE, 2012, pp. 162–171.
- [61] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, “Unit Testing Non-functional Concerns of Component-based Distributed Systems,” in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, Apr. 2009, pp. 406–415.
- [62] Trace Compass. [Online]. Available: <https://projects.eclipse.org/proposals/trace-compass>
- [63] F. Rajotte and M. R. Dagenais, “Real-Time Linux Analysis Using Low-Impact Tracer,” *Advances in Computer Engineering*, vol. 2014, Jun. 2014, article ID 173976.
- [64] F. Giraldeau, “Analyse de performance de systèmes distribués et hétérogènes à l’aide de traçage noyau,” Ph.D. dissertation, École Polytechnique de Montréal, Québec, Canada, Jul. 2015.
- [65] Object Management Group (OMG). (2009, Feb.) OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.2. [Online]. Available: <http://www.omg.org/spec/UML/2.2/Superstructure/PDF>
- [66] ISO, “Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure,” International Organization for Standardization, Geneva, CH, Standard, Apr. 2012.

- [67] M. González Harbour, J. J. Gutiérrez, J. C. Palencia, and J. M. Drake, “MAST: Modeling and Analysis Suite for Real-Time Applications,” in *Proceedings of 13th Euromicro Conference on Real-Time Systems*, I. C. S. Press, Ed., Delft, The Netherlands, Jun. 2001, pp. 125–134.
- [68] J. L. Medina Pasaje, M. G. Harbour, and J. M. Drake, “MAST Real-Time View: a graphic UML tool for modeling object-oriented real-time systems,” in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec. 2001, pp. 245–256.
- [69] Object Management Group (OMG). (2005, Jan.) UMLTM Profile for Schedulability, Performance, and Time Specification Version 1.1. [Online]. Available: <http://www.omg.org/spec/SPTP/1.1/PDF>
- [70] ——. (2011, Jun.) UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1. [Online]. Available: <http://www.omg.org/spec/MARTE/1.1/PDF>
- [71] L. Aceto, A. Bergueno, and K. G. Larsen, “Model Checking via Reachability Testing for Timed Automata,” in *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS 1384, B. Steffen, Ed., Gulbenkian Foundation, Lisbon, Portugal, Mar. 1998, pp. 263–280.
- [72] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a Nutshell,” *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.
- [73] W3C. (2015, Sep.) State Chart XML (SCXML): State Machine Notation for Control Abstraction. [Online]. Available: <https://www.w3.org/TR/scxml/>
- [74] ——. (2011, Jul.) Voice Browser Call Control: CCXML Version 1.0. [Online]. Available: <https://www.w3.org/TR/ccxml/>
- [75] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The StateMate Approach*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1998.
- [76] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, “Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479.

- [77] H. Waly, “Automated fault identification: Kernel trace analysis,” Master’s thesis, Université Laval, Québec, Canada, 2011.
- [78] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana, “Debugging Embedded Multimedia Application Traces Through Periodic Pattern Mining,” in *Proceedings of the Tenth ACM International Conference on Embedded Software*. New York, NY, USA: ACM, 2012, pp. 13–22.
- [79] S. Ma and J. L. Hellerstein, “Mining partially periodic event patterns with unknown periods,” in *Proceedings of the 17th International Conference on Data Engineering*, 2001, pp. 205–214.
- [80] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, “Discovery of Frequent Episodes in Event Sequences,” *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.
- [81] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. d. Supinski, and D. J. Quinlan, “Detecting Patterns in MPI Communication Traces,” in *37th International Conference on Parallel Processing*, Sep. 2008, pp. 230–237.
- [82] L. Alawneh and A. Hamou-Lhadj, “Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication,” in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, Mar. 2011, pp. 211–220.
- [83] H. Prähofer, R. Schatz, and A. Grimmer, “Behavioral model synthesis of PLC programs from execution traces,” in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, pp. 1–5.
- [84] A. Terrasa and G. Bernat, “Extracting Temporal Properties from Real-Time Systems by Automatic Tracing Analysis,” in *Real-Time and Embedded Computing Systems and Applications*, ser. Lecture Notes in Computer Science, J. Chen and S. Hong, Eds. Berlin, Heidelberg: Springer, 2004, vol. 2968, ch. 29, pp. 466–485.
- [85] G. Matni and M. Dagenais, “Automata-based approach for kernel trace analysis,” in *Electrical and Computer Engineering, 2009. CCECE '09. Canadian Conference on*, May 2009, pp. 970–973.
- [86] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance

- Measurement Tool,” *IEEE Computer*, vol. 28, no. 11, pp. 37–46, Nov. 1995, special issue on performance evaluation tools for parallel and distributed computer systems.
- [87] R. Wismüller, M. Bubak, W. Funika, and B. Baliś, “A Performance Analysis Tool for Interactive Applications on the Grid,” *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 305–316, Aug. 2004.
 - [88] R. Krishnakumar, “Kernel Korner: Kprobes-a Kernel Debugger,” *Linux Journal*, vol. 2005, no. 133, pp. 11–, May 2005.
 - [89] P. E. McKenney and J. Walpole, “Introducing Technology into the Linux Kernel: A Case Study,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 4–17, Jul. 2008.
 - [90] R. Beamonte, “Traçage de systèmes Linux multi-coeurs en temps réel,” Master’s thesis, École Polytechnique de Montréal, Québec, Canada, Aug. 2013.
 - [91] X. Chen, H. Harry, F. Balarin, and Y. Watanabe, “Logic of constraints: a quantitative performance and functional constraint formalism,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1243–1255, Aug. 2004.
 - [92] F. Reumont-Locke. (2015, Jan.) On the surprising behaviour of memory operations at high thread counts. [Online]. Available: https://medium.com/@The_Zorg/on-the-surprising-behaviour-of-memory-operations-at-high-thread-counts-f0ce630d9240
 - [93] R. M. Yoo, A. Romano, and C. Kozyrakis, “Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system,” in *IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 198–207.
 - [94] D. Couturier and M. R. Dagenais, “LTTng CLUST: A system-wide unified CPU and GPU tracing tool for OpenCL applications,” *Advances in Software Engineering*, vol. 2015, Jul. 2015, article ID 940628.
 - [95] E. S. Ristad and P. N. Yianilos, “Learning string-edit distance,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, May 1998.
 - [96] P. Bille, “A survey on tree edit distance and related problems,” *Theoretical Computer Science*, vol. 337, no. 1–3, pp. 217–239, Jun. 2005.

- [97] G. Piatetsky-Shapiro, "Discovery, analysis and presentation of strong rules," in *Knowledge Discovery in Databases*, G. Piatetsky-Shapiro and W. J. Frawley, Eds. Cambridge, MA: AAAI/MIT Press, 1991, pp. 229–248.
- [98] R. Agrawal, T. Imieliński, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *SIGMOD Record*, vol. 22, no. 2, pp. 207–216, Jun. 1993.
- [99] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of 20th International Conference on Very Large Data Bases*, ser. VLDB '94, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.
- [100] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 80–86, Jan. 1985.
- [101] K. P. Suresh and S. Chandrashekara, "Sample size estimation and power analysis for clinical research studies," *Journal of Human Reproductive Sciences*, vol. 5, no. 1, pp. 7–13, Jan. 2012.
- [102] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*, 3rd ed. London: Oliver and Boyd, 1949.
- [103] J. S. Vitter, "Random Sampling with a Reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, Mar. 1985.
- [104] K.-H. Li, "Reservoir-sampling Algorithms of Time Complexity $O(N(1 + \log(N/N)))$," *ACM Transactions on Mathematical Software*, vol. 20, no. 4, pp. 481–493, Dec. 1994.
- [105] J. S. Vitter, "An Efficient Algorithm for Sequential Random Sampling," *ACM Transactions on Mathematical Software*, vol. 13, no. 1, pp. 58–67, Mar. 1987.
- [106] J. Desfossez. (2016, Jan.) Monitoring real-time latencies. [Online]. Available: <https://ltnng.org/blog/2016/01/06/monitoring-realtime-latencies/>
- [107] National Instruments. (2015, Aug.) Test Engineer's Guide to HDMI 1.4. [Online]. Available: <http://www.ni.com/white-paper/12680/en/>
- [108] F. E. Grubbs, "Sample criteria for testing outlying observations," *Annals of Mathematical Statistics*, vol. 21, pp. 27–58, 1950.

- [109] G. F. Jenks, “The data model concept in statistical mapping,” in *International Yearbook of Cartography*, K. Frenzel, Ed., vol. 7, ICA. USA: Rand McNally & Co, 1967, pp. 186+.
- [110] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, ser. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1986.
- [111] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995. [Online]. Available: <http://dx.doi.org/10.1007/BF01206331>
- [112] J. E. Gray, II. (2008) Longest Repeated Substring (153). [Online]. Available: <http://rubyquiz.com/quiz153.html>
- [113] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, “State History Tree: An Incremental Disk-Based Data Structure for Very Large Interval Data,” in *2013 International Conference on Social Computing*, Sep. 2013, pp. 716–724.
- [114] L. Prieur-Drevon, R. Beamonte, N. Ezzati Jivan, and M. R. Dagenais, “Enhanced State History Tree (eSHT) : a Stateful Data Structure for Analysis of Highly Parallel System Traces,” in *Proceedings of the IEEE 5th International Congress on Big Data*. IEEE, Jun. 2016.