UNIVERSITÉ DE MONTRÉAL

MESURE ET ANALYSE DE LATENCES DANS LES SYSTÈMES PARALLÈLES EN TEMPS RÉEL

JULIEN DESFOSSEZ
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)
AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

MESURE ET ANALYSE DE LATENCES DANS LES SYSTÈMES PARALLÈLES EN TEMPS RÉEL

présentée par : DESFOSSEZ Julien
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doctorat, président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
Mme BELLAÏCHE Martine, Ph. D., membre
M. KHENDEK Ferhat, Ph. D., membre externe

DÉDICACE

À Claude-Marie et notre fils Eliott,
pour l'Amour quotidien et l'émerveillement perpétuel.

REMERCIEMENTS

Je remercie tout d'abord mon professeur Michel Dagenais pour m'avoir donné l'occation de travailler dans ce domaine qui me passionne, et d'avoir toujours su me guider dans une bonne direction tout en respectant le flôt imprévisible de la recherche. Je tiens à remercier Mathieu Desnoyers qui a agi comme un mentor et avec qui les questions même les plus techniques ont toujours trouvé réponse. Je remercie également Benoît des Ligneris qui est une source d'inspiration dans ma vie et qui m'a fait rencontrer mon professeur pour me permettre de continuer mon chemin. Je remercie également tous mes collègues présents et passés du laboratoire DORSAL pour les innombrables discussions qui ont contribué d'une manière ou d'une autre à ce projet. En particulier, je tiens à souligner les collaborations et échanges très riches avec Francis Giraldeau, David Goulet, Geneviève Bastien, Yannick Brosseau, Suchakrapani Sharma, Matthew Khouzam, Raphaël Beamonte, Mohamad Gebai et Naser Ezzati.

Bien sûr, je tiens à remercier ma famille, mes racines, qui m'ont toujours inspiré, suivi, encouragé, et permis d'aller vers mes rêves, toujours porté par leur amour sans frontières.

RÉSUMÉ

Avec les infrastructures de type infonuagiques qui augmentent de plus en plus et les services qui exploitent les avantages de la parallélisation, on arrive à un point où les analyses de problèmes de performance sont de plus en plus complexes. En particulier, avec la parallélisation, un problème de latence sur un des composants peut ralentir une requête complète. Avec la multiplication du nombre de serveurs responsables d'une seule requête, la quantité de tests et de combinaisons à valider pour trouver une source de latence peut augmenter de manière exponentielle. Les problèmes à analyser ne sont pas nouveaux, ils sont similaires à ceux étudiés dans les systèmes temps-réel. La problématique cependant se situe au niveau de la détection automatisée en temps réel des problèmes dans des conditions réelles d'exploitation, et la mise à l'échelle de la collecte de données de contexte permettant la résolution.

Dans cette thèse, nous proposons le `latency-tracker` comme solution efficace pour la mesure et l'analyse en temps réel de latences, et de le combiner avec le traceur `LTTng` pour la collecte et l'extraction de traces localement et sur le réseau. L'objectif principal est de rendre ces analyses complexes assez efficaces et non-intrusives pour fonctionner sur des machines de production, que ce soit sur des serveurs ou des appareils embarqués dédiés aux applications temps réel. Cette approche de la détection et de la compréhension des problèmes de latence dans l'ordre des dizaines de micro-secondes au niveau du noyau Linux est nouvelle et il n'existe pas d'équivalent à l'heure actuelle.

En mesurant l'impact de tous les composants ajoutés dans le chemin critique des applications de manière individuelle, nous démontrons qu'il est possible d'utiliser cette approche dans des environnements très exigeants. Les mesures se concentrent au niveau de la consommation des ressources, jusqu'à l'effet sur les lignes de cache, mais également sur la mise à l'échelle sur des applications concurrentes et distribuées.

La contribution principale de cette recherche se situe au niveau de l'ensemble des algorithmes développés permettant de mesurer précisément les latences avec un impact minimal, et de collecter assez d'informations de contexte pour en expliquer les causes. Ce faible impact permet l'application de ces méthodes dans des situations réelles où il était jusqu'à présent impossible de faire ce type de mesures sans modifier les conditions d'exécution. La spécialisation et l'optimisation des techniques actuelles d'agrégation, et la combinaison avec le domaine du traçage, donne ainsi naissance au domaine du traçage à état.

ABSTRACT

Today's server infrastructures are more and more organized around the cloud and virtualization technologies, and the services that run on these infrastructures tend to heavily use parallelisation to scale up to the demand. With this type of distributed systems, the performance analyses are becoming increasingly complex. Indeed, the work required to answer a single request can be divided among multiple servers, and a problem with any of the nodes can slow down the whole request. Finding the exact source of an abnormal latency in this kind of configuration can be really difficult and requires a lot of time. The problems we encounter are not new, they are similar to the ones faced by real-time systems. The biggest issue is to automatically detect in real-time these problems in production, and to have a scalable way to collect the context information required to understand and solve the problems.

In this thesis, we propose the `latency-tracker` as a solution to efficiently measure and analyse latency problems in real-time, and to combine it with the `LTTng` tracer to gather and extract traces locally and on the network. The main objective is to make these complex analyses efficient enough to run on production machines, either servers in data-centers, or embedded platforms dedicated to real-time tasks. This approach to detect and explain latency issues in the order of tens of microseconds in the Linux kernel is new and there is no equivalent solution today.

By individually measuring the impact of all the components added in the critical path of the applications, we demonstrate that it is possible to use this approach in very demanding environments. We measure the impact on the usage of resources, down to the impact on cache lines, but we also study the scalability of our approach on highly concurrent and distributed applications.

The main contribution of this research is the set of algorithms developed to accurately measure latencies with a minimal impact, and to collect and extract enough context informations to understand the latency causes. This low impact enables the use of these methodologies in production, under real loads, which would be impossible with the existing tools today without risking to modify the execution conditions. We specialize and optimize the current techniques related to event agregation, and combine it with tracing to create the new domain of stateful tracing.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Cette thèse aborde la mesure de latence et l'extraction de traces concurrentes sur le réseau en temps réel sur des systèmes parallèles. La notion de temps réel est utilisée ici pour indiquer que les mesures et analyses sont exécutées dans le chemin critique des applications, mais également parce que les analyses s'appliquent aux systèmes temps réel garanti où le temps de réponse est borné.

Au niveau du traçage en réseau, on étudie ici l'extraction et l'analyse de traces de systèmes parallèles pendant que la capture est active, contrairement à l'analyse de trace qui est habituellement faite *a posteriori*. Notre objectif général est de développer une méthode permettant d'analyser efficacement les latences de systèmes fonctionnant sous Linux tout en collectant le plus d'informations de contexte pertinentes pour les analyses avancées. Le travail vise autant les systèmes embarqués que les serveurs dans les centres de données.

1.1 Définitions et concepts de base

Avec les progrès de la micro-électronique, on réussit à fabriquer maintenant des appareils de plus en plus compacts et puissants. Ces appareils sophistiqués sont utilisés pour de nombreuses applications de la vie courante en interaction directe avec les usagers (services en ligne) ou avec l'environnement (commande en temps réel de moteurs, d'appareils ou même de voitures). Etant donné la sophistication de ces systèmes, il peut être difficile d'en analyser le comportement en isolation. Ainsi, pour les problèmes complexes, il faut tenter de réaliser le diagnostique en situation réelle en pleine charge, sans nécessairement connaître à l'avance quand et comment le problème va apparaître.

Dans de tels cas, une approche pas à pas avec un débogueur est impossible puisque le facteur de ralentissement est trop important et le comportement de l'application risque d'être modifié. Le traçage qui écrit des événements dans un tampon au fil de l'éxécution, sans beaucoup la ralentir, est souvent la seule solution possible. Néanmoins, dans les systèmes critiques avec de nombreux processeurs en parallèle et une grosse charge, il n'est parfois même pas possible de tout tracer sans avoir un impact sérieux sur le système. Il faut alors trouver des moyens pour sélectionner les informations à tracer et conserver. Par exemple, selon le problème détecté, on peut choisir le sous-système, ou l'intervalle de temps, pour lequel un traçage détaillé sera effectué.

Une fois que l'on a un mécanisme en place pour collecter et éventuellement extraire l'infor-

mation de contexte pertinente, il convient de détecter que le problème est survenu et réagir immédiatement pour s'assurer que toute l'information nécessaire à la compréhension et résolution du problème soit sauvegardée de manière permanente. Pour ce faire, il est important de considérer l'environnement d'opération et les contraintes d'exploitation. En effet, sur les serveurs dans les centres de données, un surcoût causé par la supervision n'a pas le même impact que sur un système temps réel embarqué.

Un système temps réel est un système qui doit fournir une réponse correcte en un temps borné sous peine de conséquences graves ou d'échec complet Laplante (1993). On distingue souvent deux catégories de systèmes temps réel : *Soft Real-Time* où l'on considère que la performance dégradée due à un non-respect de certains délais de réponse n'est pas fatale, et les systèmes *Hard Real-Time* qui au contraire ne tolèrent aucune erreur de respect des délais de réponse.

Les systèmes temps réel sont donc principalement associés aux environnements nécessitant un contrôle sur le délai de réaction. On retrouve ces systèmes dans les dispositifs de contrôle, de positionnement, de sécurité, etc. Au niveau de l'architecture de ces systèmes, on retrouve des circuits dédiés à base de micro-contrôleurs, des systèmes d'exploitation de type RTOS (*Real-Time Operating System*) tels que QNX Hildebrand (1992) et des systèmes plus génériques tels que Linux PREEMPT_RT Fu and Schwebel qui cherchent à trouver un équilibre entre fonctionnalités, flexibilité et fiabilité. Pour ces systèmes, le délai de réaction est le temps écoulé entre une stimulation et la réaction à celle-ci. Dans les systèmes informatiques, les sources de stimulation communiquent sous forme d'interruptions.

Les interruptions matérielles (IRQ) sont des signaux envoyés par des périphériques d'entrée-sortie vers un processeur, ils servent à informer le système d'exploitation de diverses conditions des périphériques (données prêtes à être lues, données écrites, etc). Chaque périphérique se fait assigner un identifiant d'interruption unique. Lorsqu'une interruption est reçue par le système d'exploitation, celui-ci interrompt l'exécution courante pour exécuter le gestionnaire d'interruption à la place. Toutefois, certaines sections du code sont critiques et nécessitent un accès exclusif sans interruption au processeur. Pour ces cas, il est possible de masquer les interruptions pendant le temps de l'exécution de ces routines et les réactiver une fois la section critique terminée. Il est aussi possible de simplement désactiver la préemption, donc une autre tâche (même de plus haute priorité) ne peut pas interrompre le fil d'exécution courant. Par contre, les interruptions sont quand même traitées.

Le délai de réaction d'un système informatique peut être étudié à différents niveaux. Dans un contexte de système temps réel, on parle souvent de temps de réponse pour exprimer le délai entre une interruption et le moment où le système commence le traitement lié à cette inter-

ruption. Le masquage des interruptions est donc un facteur important des temps de réponse trop élevés. La désactivation de la préemption a un impact au niveau de l'ordonnanceur car il empêche une tâche de plus haute priorité de prendre le contrôle du processeur. Il s'agit de deux types de latence ayant un impact sur le temps de réponse, si l'on considère seulement la réactivité du système comme le temps avant lequel le noyau commence à répondre aux stimulations.

Notre étude porte sur le noyau Linux officiel et le noyau Linux PREEMPT_RT. Ce dernier est pour l'instant un ensemble de modifications apportées au noyau officiel pour ajouter des mécanismes permettant un ordonnancement mieux contrôlé et un meilleur contrôle sur les délais de réaction. L'objectif de ce noyau est de fournir toutes les fonctionnalités qu'on attend de Linux avec en plus un moyen de donner des garanties sur les délais de réaction. La différence majeure réside dans le fait que ce noyau est entièrement préemptible, c'est-à-dire que les routines extrêmement prioritaires, telles que les gestionnaires d'interruptions, sont dans ce noyau traitées par des processus ordinaires. Il est ainsi possible de leur assigner des priorités.

Le noyau Linux est déjà équipé de nombreux mécanismes pour les mesures de performance et de latence. Le Chapitre 2 présente les différents traceurs et systèmes pour mesurer les délais de réaction déjà disponibles. On retrouve quatre grandes familles d'outils :

1. les traceurs, qui enregistrent des séquences d'événement dans des fichiers de trace ;
2. les agrégateurs, qui font des analyses quand des événements se produisent ;
3. les profileurs, qui font de la scrutation (*sampling*) pour essayer de dresser un profil de l'activité du système ;
4. les outils de mesures actifs (*benchmark*), qui visent à mesurer les performances d'un système basé sur des tests synthétiques.

Les traceurs ont comme objectif de sortir une liste d'événements cohérente dans le temps basée sur l'activité d'un système. Dans le contexte des mesures de performance, on travaille principalement avec des traceurs conçus pour avoir le moins de perturbation possible sur le système mesuré. Il s'agit d'un type d'outil permettant de réaliser des analyses très poussées, car beaucoup d'information s'y retrouve. Comme les analyses sont réalisée *a posteriori*, l'utilisateur peut passer tout le temps nécessaire à la compréhension du problème sans nécessairement avoir besoin de le reproduire. L'inconvénient de tels systèmes est justement la quantité d'information générée qui nécessite une partie de la bande passante (mémoire, réseau, disque) du système pour l'extraction, mais aussi du stockage et potentiellement un temps d'analyse considérable qui nécessite un expert du domaine ou des outils avancés. Au niveau de l'extraction de traces, la plupart des traceurs permettent l'extraction des tampons

circulaires de traçage vers le disque ou vers le réseau, mais l'analyse n'est souvent possible qu'une fois la capture terminée.

Les agrégateurs réalisent leurs analyses dans le chemin critique du système, c'est-à-dire que dès qu'un événement pertinent pour l'analyse se produit, ils prennent le contrôle, réalisent leurs calculs et rendent le contrôle. L'avantage de ces systèmes est que le résultat de l'analyse est connu presqu'immédiatement. Les analyses se présentent souvent sous forme de script compilé et injecté dans le noyau. Les scripts d'analyses communs sont des mesures de latence entre deux points dans le noyau (début et fin des appels systèmes de lecture/écriture par exemple), ou des mesures de débit (bande passante réseau d'un seul processus par exemple). L'inconvénient de ces systèmes est que le surcoût imposé par l'analyse dans le chemin critique est souvent très élevé, particulièrement dans les cas de concurrence. Les détails sont présentés individuellement dans le Chapitre 2, mais globalement les différents systèmes existants cherchent à être génériques et fiables, donc ils utilisent des verrous pour tous les accès aux structures potentiellement partagées. Un autre aspect à considérer lors de l'utilisation de tels systèmes, est que la source de données est uniquement le système actif, donc s'il manque de l'information, il est nécessaire de reproduire l'expérience et espérer que le phénomène que l'on cherche à expliquer réapparaît.

Les profileurs quant à eux sont rarement utilisés dans les problèmes de performances sporadiques. En effet, ils sont conçus pour obtenir un état global de l'utilisation des ressources, donc les problèmes rares ont peu de chance d'être détectés par ces systèmes. Par contre, étant donné que ces mécanismes de mesure sont actifs seulement une fraction du temps, leur impact a tendance à être plus faible et donc préférable dans des cas de forte charge.

Au niveau des outils de mesures actifs, ils sont une bonne solution pour tester les caractéristiques d'un système, mais il s'agit de tests synthétiques pas nécessairement représentatifs de la réalité, et ils ne peuvent pas être utilisés pour valider le bon fonctionnement d'une machine en production.

Pour mesurer les latences de réactivité d'un système en production, la seule méthode qui existe actuellement se situe dans le traceur `ftrace` et se concentre uniquement sur les latences de désactivation des interruptions et de la préemption. Cependant, les problèmes de latence anormale dans un système peuvent survenir à d'autres niveaux et, pour ces cas, la seule alternative actuelle est de se tourner vers des approches plus intrusives.

1.2 Éléments de la problématique

La capture de traces systèmes par les traceurs conçus pour la performance est souvent réalisée de manière concurrente sans synchronisation entre les processeurs. Cela permet d'enregistrer un flux de trace par processeur avec le moins d'impact possible. Pour réaliser des analyses par la suite, on utilise des algorithmes de tri de type fusion (*merge sort*) pour produire des traces cohérentes au niveau temporel. Les traceurs de ce type enregistrent les événements dans des tampons circulaires par processeur divisés en sous-tampons (*sub-buffer*). Lorsqu'un sous-tampon est plein, il est extrait vers des fichiers de trace ou vers le réseau. Cette méthode est utilisée pour limiter la quantité d'échanges entre contextes d'exécution et ainsi amortir le coût de l'extraction de trace. À l'inverse, un traceur comme `strace` affiche les événements dès qu'ils se produisent, ceci a pour avantage de donner une réponse rapide à l'utilisateur, mais l'impact est tel qu'il n'est pas recommandé d'utiliser ce traceur pour analyser les problèmes de concurrence.

Pour analyser les traces systèmes concurrentes pendant qu'elles sont produites, il convient donc de trouver l'équilibre entre surcoût et délai avant la disponibilité des données, tout en gardant la même garantie que tous les événements qui sont rendus disponibles aux analyseurs sont cohérents dans le temps et qu'aucun événement du passé n'arrivera dans le désordre. Pour respecter cette garantie, il est nécessaire de connaître l'activité de tous les flux de trace possibles. Or, certains flux peuvent être inactifs, donc du point de vue de l'analyseur en attente de donnée, il n'est pas possible de déterminer si un flux vide est inactif ou simplement plus lent à se remplir. Un délai d'attente arbitraire ne réglerait qu'une partie du problème et risquerait de corrompre l'analyse. Il convient donc de suivre l'activité de tous les flux et informer l'analyseur si un flux est réellement inactif. La problématique ici est de faire ce suivi tout en respectant les contraintes énergétiques de la plateforme.

Cette partie de la solution nous apporte une vue détaillée de l'activité générale d'un système. On peut y trouver les événements d'ordonnancement, les interruptions, les appels systèmes, les signaux, les événements de systèmes de fichiers, les périphériques d'entrée/sortie, etc. Cependant, cette approche requiert un travail important en terme d'analyse *a posteriori* et utiliser ce mécanisme pour faire de la supervision en continu peut nécessiter beaucoup de ressources. Or, dans un système en production, on s'attend à ce que la supervision utilise seulement un faible pourcentage des capacités de traitement de l'infrastructure. C'est pourquoi la plupart des systèmes de supervision se contentent de vérifier à intervalle régulier des compteurs d'utilisation et produire des statistiques. Une liste de tels outils est présentée dans le Chapitre 2. Cette approche est suffisante pour détecter la majorité des problèmes d'utilisation de ressources. Par contre, elle ne permet pas de détecter ni de comprendre les causes

de problèmes sporadiques de latences anormales.

Il convient donc de trouver une solution alliant faible surcoût pour le cas normal et vue détaillée lorsqu'une situation anormale se produit. Pour parvenir à ce résultat, il est nécessaire de détecter les solutions anormales et d'avoir un système efficace pour la capture d'informations de contexte en tout temps. Le mode `snapshot` de `LTTng` est un excellent mécanisme pour l'enregistrement de traces en continu dans une mémoire temporaire sans aucune extraction. La quantité d'information stockée dans cette mémoire est configurable. En fonction de la mémoire disponible, les événements activés et la fréquence de génération des événements, il est possible d'ajuster la longueur de la fenêtre temporelle permettant de savoir ce qu'il s'est passé quelques temps avant un événement intéressant. Le défi réside donc au niveau de la détection pour pouvoir déclencher l'extraction des tampons en mémoire vers le disque et réaliser l'analyse sur une trace relativement courte ciblée autour d'un problème connu.

Les cas de latences anormales requièrent de l'instrumentation au début et à la fin d'opérations potentiellement longues. Pour des cas de production, on va par exemple surveiller les appels systèmes, les opérations sur les périphériques, les opérations de l'ordonnanceur, ainsi que les cas de longues préemptions. Pour réaliser cette supervision en continu, il est nécessaire d'insérer du code aux endroits spécifiques où ces opérations commencent et terminent. Le surcoût relié à garder l'état entre plusieurs opérations est alors critique car il s'agît d'opérations réalisées à haute fréquence. Un algorithme capable d'assurer un suivi efficace de ces opérations tout en limitant l'impact sur le système permet alors de mesurer des latences entre deux points arbitraires.

Pour le cas plus spécifique du délai de réaction, que ce soit sur les serveurs ou sur des systèmes embarqués temps réel, la chaîne de traitement entre une source de réveil et la fin du traitement des données peut être longue. Les sources de réveil sont nombreuses et peuvent être partagées entre plusieurs destinations, et toutes les sources de réveil ne mènent pas nécessairement à un travail de la part de l'application cible. De plus, il n'y a pas d'identifiant unique permettant de mettre en lien le réveil et la fin du traitement. Il n'est donc pas possible pour ces cas là de simplement instrumenter le point de départ et le point d'arrivée et calculer la différence de temps entre les deux. Pour ces cas, le défi réside au niveau de la création de la machine à état en temps réel et du suivi de toutes les branches suivant une source de réveil jusqu'à être capable de déterminer si la branche courante est pertinente ou non. Tout cela arrive dans un contexte où le surcoût en termes de calcul et d'utilisation de la mémoire doit être connu et borné pour respecter les contraintes temps réel fortes.

1.3 Objectifs de recherche

L'objectif général de cette recherche est de fournir des algorithmes et des méthodologies qui permettent aux administrateurs systèmes et développeurs de comprendre les problèmes de latences anormales sporadiques lorsqu'ils se produisent dans les environnements de production.

Les objectifs particuliers qui découlent de l'objectif général sont :

— Développer un algorithme permettant de traiter des traces en cours de production, respectant l'équilibre entre cohérence absolue des traces parallèles, latence d'accès aux données et contrôle du surcoût ;
— Développer les algorithmes et structures de données permettant l'agrégation d'événements fréquents dans le chemin critique ;
— Déterminer et mesurer la durée du chemin critique entre une source de réveil et la fin du traitement d'une application cible dans un environnement temps réel en production.

La disponibilité d'outils basés sur les résultats de cette recherche améliorerait la détection et la résolution de cas de latences anormales. Ce processus est en ce moment manuel et nécessite un moyen de reproduire exactement le même comportement, ce qui n'est pas nécessairement facile ou rapide à obtenir en laboratoire.

1.4 Plan du mémoire

La revue de la littérature est présentée au Chapitre 2. On y présente les travaux antérieurs relatifs à notre sujet de recherche. Les trois articles scientifiques issus de la recherche suivent successivement.

L'article «Efficient Network Live Trace Streaming and Processing» au Chapitre 3 présente la recherche sur l'analyse de traces concurrentes en direct. Cet article présente un algorithme permettant de garantir que les traces parallèles reçues sont toujours cohérentes au niveau temporel, et offre un contrôle à l'utilisateur sur la latence de réception des traces en fonction du surcoût tolérable pour le système étudié. Cet article a été soumis au journal *Operating System Review* de *l'Association for Computing Machinery*.

Le second article, «Runtime latency detection and analysis», au Chapitre 4, porte sur l'étude de structures de données et d'algorithmes permettant de réaliser des mesures de latence dans le chemin critique du noyau Linux. On y aborde le sujet du traçage d'événements contextuels plutôt que ponctuels et la réaction possible lorsqu'une latence mesurée dépasse un seuil prédéterminé. L'article est principalement orienté sur l'aspect performance de l'approche

afin que celle-ci soit convenable dans le plus de situations possibles. Cet article a été accepté dans le journal *Software : Practice and Experience* de *Wiley Online Library*.

Le troisième article, «Real-Time Linux Kernel Response Time Measurement», au Chapitre 5, porte sur le suivi des délais de réaction dans les environnements temps réel. L'étude porte sur le suivi des états, entre toutes les sources de réveil et la fin du traitement dans les applications cibles, de manière automatisée, et assistée par l'espace utilisateur lorsqu'il n'est pas possible de faire autrement. L'article se base sur le précédent et l'enrichit pour que les états générés ne soient plus de simples tuples début/fin avec un identifiant permettant le lien entre les deux, mais des séries arbitrairement longues d'étapes. Le cas à l'étude est le lien entre une interruption et la fin du traitement lié à cette interruption dans un programme en espace utilisateur utilisant des files de traitement asynchrones. Cet article a été soumis dans le journal *Real-Time Systems* de *Springer*.

D'autres travaux ayant été réalisés dans le cadre de cette recherche mais qui n'ont pas été intégrés à des articles sont présentés dans le Chapitre 6.

La thèse se termine avec une discussion générale au Chapitre 7 et la conclusion au Chapitre 8.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans le contexte d'intérêt et selon les spécifications établies, cette section décrit l'état de l'art dans le domaine pertinent. Ceci permet de comprendre ce qui est présentement disponible et les possibilités de contributions originales afin d'atteindre nos objectifs.

La revue de littérature a été réalisée en utilisant les moteurs de recherche mis à la disposition par l'École Polytechnique de Montréal. Les articles proviennent principalement des revues de l'IEEE, ACM et Usenix. Google Scholar a été utilisé pour compléter les recherches.

La revue commence par présenter les traceurs disponibles sous Linux, ensuite nous présentons les travaux en lien avec la supervision et les mesures de performance dans les environnements virtualisés et d'infonuagique. Enfin, nous présentons les travaux liés aux analyses de latence dans les serveurs et les systèmes temps réel.

2.1 Traçage dans le noyau Linux

Sous Linux, on retrouve différents systèmes dans la catégorie des traceurs, cependant on peut distinguer trois catégories d'outils à ce niveau : les traceurs, les profileurs et les agrégateurs. Les traceurs enregistrent une trace d'exécution de manière chronologique, les profileurs mesurent l'évolution de certaines métriques à intervalle régulier, et les agrégateurs collectent des informations à différents points dans un programme et produisent un rapport à la fin de l'analyse. Nous présentons ici les acteurs majeurs dans le noyau Linux dans ces différentes catégories.

2.1.1 Traceurs

Le traceur `ftrace` Edge (2009a) a débuté comme une initiative sur le noyau Linux temps réel mais depuis a été ramené dans le noyau Linux officiel. L'objectif initial était d'avoir un mécanisme automatisé pour connaître l'ordre des fonctions exécutées dans le noyau afin de connaître le chemin critique pendant une certaine période de temps. En plus de tracer toutes les fonctions du noyau, il peut être configuré pour simplement utiliser la macro `TRACE_EVENT` et ainsi suivre les points de trace statiques du noyau. Il est aussi équipé de traceurs spécifiques pour des besoins de temps réel :

- `irqsoff` : trace les régions qui désactivent les interruptions et sauvegarde la trace de la plus longue période de désactivation ;
- `preemptoff` : similaire à `irqsoff` mais pour les régions qui désactivent la préemption ;

— `wakeup` : trace et enregistre le plus long délai avant que la tâche la plus prioritaire ne s'exécute, une fois qu'elle a été réveillée.

Un avantage de `ftrace` est le fait qu'il est entièrement intégré au noyau et aucun outil externe n'est nécessaire pour le contrôler ou même lire les données. Tout le contrôle se passe dans le pseudo système de fichiers `debugfs` et les fichiers de trace de base sont extraits au format texte ou binaire selon le besoin. Étant donné le volume de données généré, il est également possible de sortir la trace dans un format binaire et utiliser l'outil graphique `KernelShark` pour consulter plus en détail les informations recueillies. Ce traceur est très performant et bien adapté pour analyser les problèmes de concurrence et de latence. Par contre, il vise comme public les développeurs noyau qui cherchent à résoudre un problème localement. Il n'est pas conçu avec des fonctionnalités pour l'utiliser comme système de supervision, local ou en réseau en entreprise. Certains mécanismes existent pour le rendre utilisable en réseau ou avec de la virtualisation, mais ce ne sont pas des mécanismes robustes qui pourraient convenir dans de plus gros déploiements (synchronisation à base de `sleep`, beaucoup de configuration manuelle à faire au cas par cas, peu d'options pour la gestion de la mémoire, etc).

Le traceur `LTTng` Desnoyers and Dagenais (2006), commencé en 2006, est le premier traceur a avoir pris en considération l'importance de l'impact sur le système Desnoyers (2009a). Il s'agit du successeur de LTT Dagenais et al. (2005) créé en 1999. Son développement a toujours progressé à l'extérieur des sources du noyau officiel. `LTTng` est un ensemble d'outils en espace noyau et utilisateur permettant de générer, contrôler et analyser des traces. Avant la version 2.0, `LTTng` était un ensemble de modifications à appliquer sur le noyau officiel et apportait beaucoup d'instrumentation personnalisée. À partir de la version 2.0, tout le coeur du traceur fonctionne sous la forme de modules. Le traceur noyau peut s'interfacer avec différents mécanismes d'instrumentation :

— `TRACE_EVENT` : déployés par les développeurs du noyau Linux ;
— `kprobes` Rostedt (2005) : pour l'instrumentation dynamique, un événement est généré quand la sonde est exécutée. Les sondes `kprobes` peuvent être insérées presque partout dans le noyau, seuls certains emplacement sont interdits car trop risqués ;
— `kretprobes` : pour l'instrumentation dynamique de fonctions arbitraires. Un événement est généré au début et à la fin des fonctions.

Les événements sont activés dans des canaux, chaque canal possède son propre tampon circulaire divisé en sous-tampons. Chaque processeur a son propre ensemble de sous-tampons dédié. Chaque canal peut être configuré individuellement en termes d'espace mémoire dédié et nombre de sous-tampons disponibles par processeur. Il est aussi possible d'associer une liste de compteurs de performance (`Perf PMU counters`) à aller lire à chaque fois qu'un

événement est généré dans un canal. Ces compteurs permettent d'aller chercher des informations contextuelles sur l'état du système par processeur (par exemple : le nombre de fautes de pages, le nombre de fautes de cache L1, le compteur de cycles, etc).

Les traces générées dans la version 2.x sont écrites dans le format `Common Trace Format` Desnoyers (2011). LTTng permet également de tracer des applications en restant dans l'espace utilisateur et les corréler ensuite avec les traces noyau. Le traceur en espace utilisateur fonctionne de la même manière que le traceur noyau (tampon circulaire par canal par processeur). L'avantage principal pour les applications est que l'accès au noyau n'est nécessaire que pour extraire les paquets de trace pleins. Cette approche est très performante car le surcoût est amorti par le nombre d'événements contenu dans un paquet.

Au niveau des analyses, un riche écosystème exite autour de LTTng et du format CTF. Babeltrace est l'outil par défaut pour convertir des traces du format CTF à une représentation intermédiaire (utilisable avec la librairie C et Pyton) ou au format texte. Trace Compass [tra] est l'interface graphique de référence pour explorer des traces CTF. De nombreuses vues et analyses sont incluses pour présenter les données de traces selon différents points de vues (par processus, par ressource, dans le temps, en fréquence, etc). Des analyses plus avancées telles que l'analyse de chemin critique Giraldeau and Dagenais (2015) sont incluses également pour représenter graphiquement les chaînes de réveil. Des analyses sont également possibles sous forme de scripts automatisables avec le projet LTTng-analyses [ltt] qui se base sur la libraire Python de Babeltrace. Ces outils sont organisés pour calculer des métriques systèmes à partir de traces noyau sous forme de statistiques (minimum, maximum, moyenne, écart type), de distribution en fréquence (pour les latences), ou de listes d'occurences triées sous forme temporelle ou par délai.

`perf` Edge (2009b) est un hybride entre traceur et profileur. Il est intégré au noyau Linux et il a été conçu à l'origine pour fournir une interface unifiée d'accès vers les compteurs de performance présents dans les processeurs, aller lire ces compteurs de manière périodiques, et surtout enregistrer l'activité courante quand un de ces compteurs dépasse une certaine valeur. Il a remplacé `oprofile` comme outil standard de profilage. Après être devenu un profileur, il a également évolué pour s'interfacer avec la macro `TRACE_EVENT` et fournir le même type de données que les traceurs. Ainsi, il est possible de l'utiliser pour produire rapidement des statistiques sur le nombre de fois qu'un point d'instrumentation a été exécuté, ou le nombre d'événements enregistrés sur un processeur pendant une période donnée. Le traceur se contrôle par l'outil du même nom situé dans le répertoire `tools/` des sources du noyau. Perf peut également convertir ses traces générées au format CTF pour analyse avec les outils supportant ce format.

Plus récemment (2014), `sysdig` sys (2014) est apparu comme un nouvel outil de traçage conçu autour de problèmes concrets et fréquents d'exploitation. Celui-ci s'interface avec certains `TRACE_EVENT` spécifiques : les événements d'ordonnanceur et les appels systèmes. Avec ces informations, des scripts (`chisels`) sont fournis pour extraire des métriques telles que l'activité des fichiers par utilisateur, l'utilisation des ressources réseau et disque, la liste des connexions réseau par processus ou globalement, etc. Donc les analyses réalisées sont similaires à ce qui est faisable avec les autres traceurs, mais il s'agit ici de l'objectif principal. La trace peut être lue en direct, mais elle peut également être sauvegardée sur le disque pour une analyse différée. L'intérêt est que les outils sont simples à utiliser et déjà adaptés pour la plupart des cas d'utilisations. Par contre, ce traceur n'est pas adapté pour les problèmes plus précis d'ordonnancement ou de temps réel qui nécessitent un plus grand nombre de sources d'informations pour avoir un profil complet de l'utilisation des ressources.

2.1.2 Agrégateurs

`Kprobes` Rostedt (2005) fournit un point d'entrée pour exécuter du code arbitraire dans le noyau, utiliser cette technologie requiert une expertise en développement noyau, ce qui n'est pas forcément le cas de tous les utilisateurs voulant plus d'information sur leur système d'exploitation. C'est pourquoi le traceur `SystemTap` sys a été conçu. Celui-ci vise la communauté des administrateurs système en leur fournissant un ensemble de scripts d'instrumentation qui s'interfacent automatiquement avec l'instrumentation statique fournie par `TRACE_EVENT`, mais également dynamiquement avec des points plus précis grâce à `kprobes`. Une limitation de `SystemTap`, dans un contexte où des traces volumineuses sont générées, est qu'il ne possède pas de format de trace compact ; les traces sont exportées sous une forme textuelle et cela impose des appels à une fonction équivalente à `printf` à chaque fois que des données doivent être exportées. De ce fait, ce traceur impose une charge sur le système et ne permet pas une analyse *a posteriori* détaillée. Par contre, les structures de données intégrées lui donnent la possibilité de calculer des statistiques personnalisées pendant la capture, ce qui est le principal cas d'utilisation de ce traceur. Une autre limitation en terme de mise à l'échelle, est l'accès aux structures de données partagées (tables de hâchage), entre plusieurs coeurs d'exécution, qui implique toujours un verrou.

Plus récemment, `eBPF` Corbet (2014) a été inclus dans le noyau. BPF était à l'origine une machine virtuelle dédiée à filtrer des paquets réseau (utilisé par exemple par `tcpdump`). Cependant, avec `eBPF`, il est maintenant accessible dans tous les sous-systèmes du noyau et le langage a été étendu pour permettre plus d'actions. Avec `eBPF`, il est possible de s'attacher à des points d'instrumentation du noyau avec `kprobes` et `TRACE_EVENT` et extraire des

informations pour calculer des statistiques. Les informations peuvent ensuite être rendues accessibles en espace utilisateur pour extraire les données et les présenter à l'utilisateur. Ce cas d'utilisation est similaire à `SystemTap`, la grande différence étant que le code exécuté est dans une machine virtuelle plutôt que dans du code C généré. Cette différence améliore la sécurité, car l'exécution du code `eBPF` n'a pas d'impact sur le fonctionnement du noyau Linux, il est isolé, alors qu'il est simple de causer des erreurs graves dans le noyau Linux avec des programmes `SystemTap`.

2.2 Supervision dans les environnements d'infonuagique

Depuis les dernières années, on constate un fort intérêt pour la recherche dans la gestion et la supervision dans les environnements d'infonuagique. La plupart des articles présentés dans cette section datent de 2011 et plus. On retrouve également des articles sur des technologies plus anciennes qui ont été portées aux nouvelles infrastructures.

ML-ADSD Su et al. (2002) est un système pour répartir les tests à faire sur les grappes de calcul de manière hiérarchique. Le contrôle est fait sous forme d'une architecture multi-maîtres afin d'éviter les pannes des serveurs de test. Bien que cette architecture soit intéressante pour le côté redondant, son implantation requiert une réorganisation du réseau, ce qui peut freiner son adoption. De plus, les tests ont été faits seulement sur 32 noeuds à superviser, ce qui est peu pour valider les arguments d'économie de bande passante. Enfin, le système assume qu'il n'y aura pas de bris du lien réseau, seulement des pannes de machines, ce qui n'est pas très réaliste.

NetLogger Gunter and Tierney (2003),Tierney et al. (1998) est un système de diagnostic en temps réel des problèmes de performance dans les environnements distribués. Cet outil est intéressant car il permet un traçage de bout-en-bout de la chaîne et il possède un système graphique de visualisation de métriques. Le diagnostic est intéressant car on est en mesure de visualiser l'endroit du ralentissement dans toute l'architecture. Ce logiciel a été conçu en 1998 mais a été maintenu jusqu'en 2009. Par contre le traceur n'est pas assez performant pour l'information que nous cherchons à obtenir. En effet, l'impact sur le système est de 0.2 à 0.5ms par événement. Aussi, le format de trace est sous forme de texte et l'instrumentation est très spécifique à l'application concernée.

Dans Taufer and Stricker (2003), les auteurs font de l'échantillonnage à intervalle régulier du pseudo système de fichier */proc* pour lire des compteurs de performance déjà en place et d'autres ajoutés spécifiquement pour leurs besoins. Ils se basent sur la collection des métriques au niveau d'un intergiciel (*middleware*) entre leur application de calcul scientifique et le noyau

pour attribuer les changements d'états des compteurs aux opérations dans leur application. Afin de limiter la charge sur le système, ils adaptent leur fréquence d'interrogation du système à la charge courante. De plus, ils utilisent le protocole UDP pour le transfert des données, et leur système à un modèle d'interpolation pour estimer les données manquantes. Ce système est intéressant et répond à leur besoin dans le domaine du calcul scientifique. Par contre l'ajout manuel de l'instrumentation dans le noyau n'est pas très détaillée et les appels aux fonctions dans */proc* sont souvent protégés par des verrous qui peuvent nuire à la performance du système.

Katsaros et al. Katsaros et al. (2012) présentent un système de supervision qui s'ajuste également en fonction de la charge. Les métriques sont choisies par les développeurs et l'objectif est d'avoir un impact minimal sur le système supervisé. L'article est intéressant car il justifie la pertinence de s'adapter au niveau de charge et discute des techniques utilisées pour y arriver. Le système est mis en place dans un environnement réel de rendu vidéo, afin de garder les meilleures performances possible, il respecte les couches logiques et évite les échanges coûteux. Par contre, il s'agit d'un traçage *ad-hoc* pour leur application et les échanges sont réalisés par un service échangeant des données au format XML d'assez haut niveau.

Cecchet et al. (2009) présentent un article très intéressant car concret qui cherche à résoudre un problème en production. Il s'agit d'une application d'échange d'actions en bourse où la latence pour faire une requête ne doit pas dépasser 10ms. Dans un tel environnement, l'instrumentation est jugée trop intrusive et ils ont donc développé un système pour limiter la quantité d'instrumentation nécessaire, mais quand même rester en mesure d'identifier les problèmes de performance. Par exemple, plutôt que de mesurer le temps pris par une requête, ils calculent le nombre moyen de requêtes passant par un noeud et, en fonction de leur modèle, ils sont en mesure de déterminer si les seuils sont respectés. Ce système est une optimisation intéressante et montre encore une fois le besoin de lire des métriques avec un impact minimal sur le système. Toutefois, ce concept est applicable uniquement dans les environnements très contrôlés où l'application est modélisable et les données à chercher sont connues.

Après les protocoles Netflow Claise (2004) et IPFIX (RFC 5101 Claise (2008) et RFC 5102 Quittek et al. (2008)), l'industrie cherche à définir un nouveau standard pour la supervision d'application dans les grands parcs informatique. Il s'agit de AppFlow. Ce protocol et format est compatible avec NetFlow et IPFIX, ce qui le rend très intéressant pour les environnements où ces protocoles sont déjà en place. Il s'agit d'un protocole se voulant léger et conçu pour l'infonuagique. Dans Khargharia et al. (2010), les auteurs présentent l'utilisation de ce protocole pour reconfigurer automatiquement un parc informatique en fonction du besoin et ainsi optimiser la consommation énergétique. AppFlow est conçu pour des considérations

d'assez haut niveau, en effet la plupart des gabarits fournis dans la spécification indiquent des métriques qui donnent des informations globales sur l'état du parc. Le protocole est également trop lourd pour transporter efficacement des données de performance à haut volume. Par contre, on le retient pour un niveau plus haut d'abstraction car plusieurs applications déjà en utilisation dans les parcs informatique sont compatibles avec ce format.

Le système IMA4SSP Kertesz et al. (2012) (pour *Integrated Monitoring Approach for Seamless Service Provisioning*) propose un système de supervision afin de déterminer le meilleur emplacement pour déployer un service. Le système mesure les performances d'application WSDL, définit une interface pour réaliser des tests à distance et gère le choix du meilleur fournisseur pour placer un service en fonction d'un ensemble de critères. Ce système est assez éloigné de notre sujet, mais on note toutefois le processus de sélection du meilleur fournisseur qui pourrait être réutilisé une fois que notre infrastructure de base sera en place.

Pour poursuivre dans la même idée, Gohad et al. (2012) proposent des algorithmes pour optimiser le placement des machines virtuelles (VM) chez différents fournisseurs en fonction des termes de service, des coûts et des performances des machines proposées. Les démarches de calcul du meilleur fournisseur et du choix de placer une VM sont intéressantes, mais le modèle semble uniquement théorique et les contraintes telles que la charge imposée sur le système supervisé et l'impact des intéractions entre les VMs semblent ignorés.

Afin de choisir le fournisseur d'infonuagique à privilégier, les auteurs de Yigitbasi et al. (2009) ont conçu un outil pour réaliser des tests à distance et ainsi mesurer avec précision l'efficacité des différentes implémentations en fonction du besoin (environnement, conditions d'accès réseau, ordonnancement, etc). L'article est intéressant dans le concept mais on aurait aimé y trouver les mesures prises et comment elles sont collectées. Apparemment, certaines données proviennent de */proc*, le système profiterait sans doute de métriques plus précises dans les cas où les données de haut niveau ne sont pas suffisantes pour départager deux fournisseurs.

Dans Heward et al. (2010), nous avons une confirmation que le surcoût introduit par les systèmes de supervision sont un enjeu majeur à régler car ils ont un impact sur les activités normales. Dans cet article, les auteurs se concentrent uniquement sur les serveurs web et plus particulièrement les architectures de type SOA, mais les conclusions sont clairement vraies pour d'autres environnements d'informatique distribués.

Dans Fu et al. (2013), les auteurs présentent une méthode pour regrouper les noeuds entre eux de manière hiérarchique dans des applications distribuées afin de faire des groupes de métriques et ainsi limiter la quantité de messages nécessaires pour la supervision. Malheureusement, l'article manque de réalisme, les résultats sont simulés et les calculs restent très théoriques, on a pas de détail sur le type d'instrumentation ni sur le niveau de détail fourni

par le système. De plus, l'objectif principal ici est de mesurer la disponibilité du réseau, ce qui n'est pas exactement le même objectif que nous, mais on retient les algorithmes présentés possibles pour le transfert des données en amont.

Le document de Tierney et al. (2002) est une spécification contenant les métriques à collecter dans les environnements de grille de calcul. Il liste les propriétés des différentes métriques, les méthodes de mesure et les pré-requis pour la supervision de grille. On ne nous montre pas de détail, mais plutôt des suggestions sur comment réaliser les tests. Ce document montre l'importance de ce besoin. Par contre il s'agit encore une fois d'un système à haut niveau. Les protocoles recommandés sont basés sur XML ou SOAP, ce qui nous montre qu'ils ne cherchent pas nécessairement à récolter le niveau de détail qui nous intéresse. Étant donné que leur environnement est standard et dédié à une tâche, ils cherchent plutôt à avoir une idée globale du fonctionnement de la grille, sans spécialement chercher à rentrer dans les détails ou optimiser certains composants.

Dhingra et al. (2012) nous présentent un système distribué de supervision spécialisé dans l'infonuagique. L'architecture est constituée d'agents à tous les niveaux (VM, hyperviseur, machine physique) pour réaliser une collecte hiérarchique des métriques. Ils proposent une interface avec les clients pour assurer une transparence des données de comptabilité. L'approche est intéressante car elle réutilise beaucoup d'outils différents déjà existants, avec chacun avec une tâche : bwm-ng pour le réseau, oprofile pour l'utilisation processeur dans une machine physique pour une VM, XenMon tool pour l'utilisation de CPU virtuel. Ils ne cherchent pas à obtenir des données sur le bas niveau. Aussi, l'impact des mesures sur le système global n'est pas pris en compte. En effet, des outils comme oprofile ont un impact à ne pas sous-estimer qui le rend bien souvent inutilisable dans des conditions de production.

Dans Chandrasekar et al. (2012), les auteurs cherchent également à valider le respect des termes de service par le fournisseur d'infrastructure. Le système est découpé en deux parties : une partie pour le fournisseur et une autre pour le client. Le système se base sur XenMon, présenté par Gupta et al. (2005), pour extraire les données de performance, et le client a la possibilité de valider les données extraites par le fournisseur afin d'établir un niveau de confiance avec celui-ci. L'algorithme s'assure de calculer ce niveau de confiance en fonction des données échangées dans un modèle basé sur les chaînes de Markov. Ce système ne prend pas en compte la granularité de informations de performance pertinentes et assume que le fournisseur va installer un logiciel intrusif dans son hyperviseur, dans le but que les clients puissent valider son niveau de confiance.

Lindner et al. (2011) proposent une vue détaillée de tout l'éco-système de l'infonuagique. Il présente les aspects communs à la gestion de services hébergés en ligne dans des environ-

nements qui ne sont pas gérés par l'utilisateur. Ce qui est intéressant pour notre étude est l'aspect supervision qui rentre ici dans le cadre des modèles de facturation et dans les performances globales des services. L'article ne fournit pas de détails techniques, mais l'approche en largeur fournit une bonne vision globale de tous les paramètres à prendre en compte lorsqu'on étudie ces environnements.

Pour l'aspect sécurité, Muñoz et al. (2012) présentent une architecture pour la supervision des événements de sécurité et l'application de mesures correctives dans un environnement d'infonuagique. Cette architecture est découpée en plusieurs couches et est dotée d'un langage pour exprimer des règles ainsi qu'une machine à états finis pour l'évaluation du comportement. Le problème avec cette implémentation est qu'elle se concentre sur les événements de haut niveau (par exemple : le nombre de fois qu'un utilisateur s'est trompé de mot de passe) et ne fournit pas d'information quant à l'impact en performance lié à l'utilisation des différents composants. Enfin, on ne voit pas clairement pourquoi ce système est fait pour l'infonuagique plutôt que n'importe quel autre environnement.

Kang et al. (2010) présentent la conception du logiciel RTM (*Real-Time Monitor*) qui sert à superviser le comportement des applications pendant leur exécution, analyser les données collectées et optimiser les ressources dans un environnment d'infonuagique. Il s'agit d'une instrumentation des bibliothèques ainsi que de la lecture des compteurs de performance du système d'exploitation. L'application vise en particulier les processeurs multi-coeurs tels que le Tilera TILE64 et modifie les bibliothèques `libpthread` et `MPI` pour ajouter de l'instrumentation. Le processus d'instrumentation consiste à surcharger les bibliothèques en les faisant calculer le temps passé dans l'appel de chaque fonction jugée importante et écrire ce résultat sur un tuyau (*pipe*) de communication. Il est clair ici que les considérations de performance n'ont pas été prises en compte car une telle pratique risque d'avoir des conséquences désastreuses sur les performances de l'application, particulièrement sur des applications parallèles.

2.3 Supervision spécialisée dans la virtualisation

Étant donné le lien fort qui existe entre les infrastructure d'infonuagique, la virtualisation et les mesures de performance, cette revue de littérature s'intéresse maintenant à l'état de l'art et aux dernières avancées en matière de collecte d'information de performance dans les hyperviseurs et les machines virtuelles (VM).

Pour commencer, ComMon Xiang et al. (2012) est un logiciel permettant d'étudier et de mesurer l'activité d'une VM sans avoir besoin d'intéragir avec celle-ci. Pour ce faire, il utilise le détecteur d'intrusions réseau Snort pour caractériser le comportement réseau, et force

l'hyperviseur à reprendre le contrôle à chaque appel système. Ainsi, l'hyperviseur, et donc la machine physique, connaît l'activité de la VM. Cependant, cette pratique a un coût très élevé, car tous les appels systèmes qui pourraient être gérés à l'intérieur de la VM sont maintenant interceptés par la machine physique, simplement pour faire des statistiques.

Dans l'article Khandual (2012), les auteurs utilisent le traceur Perf Edge (2009b) pour superviser des machines virtuelles KVM. Ils nous présentent des cas d'utilisation détaillés sur les statistiques que Perf peut extraire depuis les VM et depuis la machine physique. Les auteurs discutent également de l'intégration qui est en train d'être faite pour permettre l'accès à une vue des compteurs de performance matériels depuis les VM, ce qui est pour l'instant impossible car cela donne trop d'information. Cet article est un résumé de ce qui se fait et ce qui est en train d'être développé et il est illustré par des exemples. Cela semble être jusqu'à présent une des meilleures référence en ce qui concerne la collecte de données de performance dans une VM et dans l'hyperviseur, car il s'agit de l'approche intégrée au noyau Linux et relativement standardisée.

Dans la même veine, l'article Nikolaev and Back (2011) présente l'outil Perfctr-Xen qui sert à fournir un accès direct aux compteurs de performance matériels dans l'environnement Xen à travers l'interface PAPI Mucci et al. (1999). Toutefois, peu de détails sont donnés quant à la performance et surtout à savoir si les données exportées sont les données brutes ou virtualisées (ne prenant en compte que les données liées à la VM).

La supervision de VM attire beaucoup les chercheurs en sécurité, car cela leur permet d'étudier le comportement d'un logiciel sans être détecté. Dans le cas de Yang et al. (2012a), les chercheurs ont développé un système pour détecter les connections réseau invisibles depuis le système d'exploitation. En effet, certains *rootkits* dans le noyau (Linux ou Windows) s'arrangent pour masquer leur activité réseau et ainsi être le plus discrets possible. Pour détecter ces logiciels malveillants, le système proposé ici intercepte chaque connexion réseau avant qu'elle ne soit établie, vérifie dans la VM si cette connexion est visible ; si oui il la laisse passer, sinon il rapporte une alerte et ferme la connexion. Bien que la technique soit intéressante, elle est néanmoins coûteuse en terme de performance réseau,

Si l'on s'intéresse maintenant aux mesures de performance, le logiciel VSA présenté dans Shao et al. (2012) présente le lien entre l'utilisation de processeurs virtuels et physiques ainsi que l'analyse des intéractions entre plusieurs processeurs virtuels appartenant à la même machine virtuelle. Ce logiciel se base sur XenTrace pour la capture des traces. Une fois les traces enregistrées, il les utilise pour déterminer les raisons de blocage, les problèmes de préemption, etc. La technique est intéressante et l'analyse des blocages pourrait permettre de trouver des optimisations. Cependant, pour l'appliquer dans notre cas, il faudrait que ce

système soit utilisable en production, ce qui n'est pas le cas avec cette technique.

Lattice Clayman et al. (2010) est un système de supervision conçu pour la virtualisation. Les tests respectent les couches logiques et il est prévu pour la mise à l'échelle en offrant la possibilité d'activer des sources de données au besoin. Par contre, ces données semblent être de haut niveau (utilisation du processeur, de la mémoire, etc), aucune information n'est donnée sur la performance du système en production et son impact sur le système supervisé. On ne sait pas non plus s'il serait possible facilement d'ajouter d'autres métriques à tester.

Pour terminer ce tour d'horizon, voyons XMM Kim et al. (2012). Il s'agit d'un système de supervision des ressources de l'hyperviseur Xen qui fonctionne de manière peu intrusive. Le système est une extension à XenTrace pour produire des événements lors d'hypercalls, extraire les informations d'ordonnancement, les interruptions, la création et la destruction de domaines (VM) et les informations de pagination mémoire. Avec ces données, les auteurs sont en mesure de déterminer les ressources qu'une VM utilise et ainsi choisir la meilleure stratégie d'ordonnancement. Ils fournissent également une interface graphique permettant de bien visualiser les données. Toutefois, aucune mention n'est faite de l'impact en performance d'un tel système.

2.4 Supervision spécifique pour des applications à haute performance

Les grands acteurs de l'infonuagique ont déjà adopté depuis longtemps des solutions qui répondent à leur besoin en terme de supervision. Pour confirmer qu'il n'existe pas à l'heure actuelle de solution universelle répondant au besoin, il suffit de regarder ce qu'ils utilisent et se rendre compte que chacun a développé sa propre solution.

Par exemple, Twitter a développé Zipkin [zip]. Il s'agit d'un système d'instrumentation de leur infrastructure basé sur l'échantillonnage et le traçage de bout-en-bout. Étant donné le volume de données et la quantité de requêtes que l'infrastructure de Twitter doit gérer, ils instrumentent seulement une certaine proportion des requêtes qui leur arrivent. Par contre, ces requêtes sont instrumentées sur toute la chaîne d'appel. Ainsi, avec le grand nombre, ils sont en mesure de déterminer les endroits de ralentissement et détecter les problèmes de performance quand ils se présentent. Zipkin a été publié sous license APLv2. Il s'agit d'une technique très intéressante car elle trouve un compromis entre quantité de données et précision des données. Par contre, elle est limitée aux composants de l'API de Twitter ayant été instrumentés. Les données de plus bas niveau ne sont pas prises en compte par ce système.

Google de son côté a développé Dapper Sigelman et al. (2010), ce système est très intéressant car c'est un des rares traceurs distribués pour les infrastructures à grande échelle. Il s'agit

de l'outil de référence chez Google pour diagnostiquer les problèmes de performance. Il est déployé sur la plupart des serveurs de production. Il s'agit d'une composition entre échantillonnage et traçage des appels. Toute l'infrastructure est prévue avec les considérations de performance et un grand volume de données à traiter. Ce système semble être une très bonne solution et avoir une grande valeur chez Google. L'instrumentation quant à elle est spécifique aux applications de Google. Ces métriques ne semblent pas intégrées avec les données plus bas niveau du système et encore une fois se basent sur l'échantillonnage, étant donné la quantité de requêtes et l'impact d'ajouter de l'instrumentation à ce niveau.

En plus de ce système, Google a développé et publié sous license GPLv2 son système d'alerte Rocksteady [roc]. Ce système hiérarchique collecte les données de performances générées par des agents sur chaque serveur, fait des calculs sur ces données, valide des seuils, envoie l'information à un moteur de graphiques et génère des alertes en cas de problème. Les données sont acheminées par le système d'envoi de message RabbitMQ, les graphiques sont gérés par Graphite et les alertes par Nagios qui est en mode de réception uniquement (donc il ne fait pas de test sur les serveurs, il attend les résultats). Pour le traitement des données, ils utilisent un moteur de traitement des événements EPL (*Event Processing Language*) qui permet de faire des calculs avancés à la réception de chaque événement. Ce système est très intéressant et a été testé dans les phases d'analyse de notre projet. Il s'avère puissant, mais la communication est clairement trop coûteuse pour transporter des événements de trace niveau noyau.

Chez Amazon, il existe une interface pour collecter des métriques dans les VMs des clients [ama]. A priori, les métriques présentes ici sont de haut niveau (utilisation du processeur, disque, réseau), ce qui leur sert à la facturation. Les clients peuvent également utiliser leur interface de programmation pour envoyer des métriques personnalisées, mais peu de détails sont disponibles à ce sujet.

2.5 Mesure de latences et systèmes temps réel

En dehors des mesures de l'utilisation des ressources, un des points où le traçage est très intéressant se trouve au niveau des mesures de latence. En effet, l'utilisation se mesure souvent avec des moyennes, qui sont assez facilement calculables avec les compteurs systèmes. Cependant, pour être précises, les mesures de latence doivent se faire au niveau des appels individuels. Les systèmes temps réel ont comme objectif de garantir un délai maximal de réaction à un événement, donc c'est dans cette famille de systèmes que l'on trouve le plus d'information pertinentes à ce sujet. Toutefois, les mesures de latence sont d'un intérêt grandissant pour les opérateurs de serveur, particulièrement dans les infrastructures à grande échelle. Par exemple, une requête de recherche sur Google peut communiquer avec plusieurs

milliers de serveurs Luu (2016) en parallèle avant de retourner les réponses à l'utilisateur. Les serveurs vont chacuns faire un traitement de 1 à 2 ms. On retrouve également ce type d'architecture hautement paralléle et distribuée chez Facebook Rothschild (2009) et dans la grande majorité des services en ligne à haut débit.

Dans de tels systèmes, une latence trop élevée sur n'importe quel noeud a un impact sur la requête au complet. Donc, le risque qu'une requête utilisateur soit traitée dans le pire temps de réponse d'un serveur est proportionnel au nombre de sous-requêtes. Il est ainsi essentiel de contrôler très précisément le délai de réaction de tous les maillons de la chaîne et surtout de mesurer le pire cas à chaque niveau plutôt que de se limiter à la moyenne globale.

Le manque d'analyses précises dans ce domaine est source de grande incertitude, ce qui conduit les opérateurs à limiter le nombre de requêtes à envoyer à un serveur en parallèle, afin de rester dans une moyenne de temps de réponse acceptable Reiss et al. (2012). La conséquence de ce manque de contrôle sur les ressources et l'augmentation constante du nombre d'utilisateurs est que les fournisseurs de service ajoutent du matériel pour être sûrs de répondre au besoin. Cela a des conséquences sur les besoins énergétiques. Afin de limiter ces dépenses énergétiques, de la recherche est effectuée sur l'optimisation du placement des services. Par exemple, dans Lo et al. (2014), les auteurs démontrent qu'il est possible de faire cohabiter des tâches de calcul non-interactives (telles que de l'indexation) avec un service nécessitant un faible délai de réponse.

La mesure du délai de réponse est un problème complexe, car les délais sont en général très faibles. En conséquence, n'importe quel ajout pour la supervision devient perceptible. Les environnements temps réel sont souvent équipés de systèmes de mesure de latence. Ftrace Edge (2009a) a été conçu à l'origine pour ces systèmes, en particulier les traceurs `preemptoff` et `irqsoff` Rostedt (2009). Ces traceurs sont dédiés pour déterminer les cas de désactivation de préemption et masquage d'interruption les plus longs. Avec ces analyses, il est possible de déterminer l'origine de ces actions qui risquent d'empêcher le système de réagir à un événement dans un temps acceptable. Une autre approche pour essayer de détecter les cas de latence anormales est d'utiliser un test périodique (dont on connait à l'avance la période de réveil) et comparer la différence entre le moment prévu de réveil et le véritable moment de réveil. L'outil `cyclictest` Gleixner est le programme de référence pour ce type de mesure, il fait partie des tests standards réalisés sur les systèmes temps réel pour tester leur latence de réaction aux événements. Il s'intègre avec les traceurs `Ftrace` expliqués précédemment pour essayer de corréler des latences anormales avec des cas de désactivation de la préemption ou des interruptions trop longues. Cet outil est par contre uniquement un outil de test de performance actif, il ne peut pas être utilisé pour valider le bon fonctionnement pendant que

le système est en production.

Pour fonctionner correctement, `cyclictest` doit connaître à l'avance l'instant où l'interruption de réveil survient, donc ce test valide seulement les charges de travail périodiques (donc prévisibles). Dans les cas où la source de réveil est une interruption apériodique, avec des interruptions provenant d'un périphérique matériel par exemple, il ne peut pas être utilisé. Dans Rybaniec and Wieczorekb (2012), les auteurs utilisent un port d'un micro-contrôleur pour émettre une interruption et attendent de recevoir une réponse sur un autre port. Avec cette approche complètement externe, ils mesurent ainsi le délai de réponse de la chaîne complète. Cette approche a l'avantage d'être proche d'un cas d'utilisation réel et il serait possible d'y ajouter le support du traçage noyau pour analyser les cas problématiques. Toutefois, ceci ne permet pas non plus la validation dans des conditions de production.

Comme on peut le voir, les mesures de la chaîne complète de latence sont difficiles à réaliser dans des cas réels car il est nécessaire d'être en contrôle de la source d'interruption, ce qui n'est pas souvent le cas en production. La plupart des analyses de latence se limitent donc souvent à un endroit spécifique, qui a le potentiel de ralentir le traitement d'une réponse. Les traceurs `preemptoff` et `irqsoff` sont un exemple qui sont les plus proches possibles de la source d'interruption. Plus loin dans la chaine, on retrouve également des scripts `SystemTap` et `eBPF` qui peuvent analyser les latences spécifiques au niveau du système d'exploitation (ordonnanceur, temps passé inactif, bloquage dans des appels systèmes, etc), mais rien qui fasse le lien entre les différentes étapes de traitement des interruptions jusqu'à l'espace utilisateur.

2.6 Synthèse

D'après les résultats de cette revue de littérature, on se rend compte qu'il existe un intérêt grandissant pour les architectures de supervision dans les infrastructures de type infonuagique. En effet, étant donné que le modèle est basé sur la facturation à l'utilisation, ces infrastructures sont essentielles pour les opérateurs, mais également pour les utilisateurs qui souhaitent vérifier le respect des engagements de leur fournisseur et limiter leurs coûts.

Toutefois, il apparaît clairement que les infrastructures présentées se limitent souvent à valider des métriques de haut niveau et que la plupart des algorithmes et techniques présentés n'ont pas été testés dans des environnements de production de taille suffisante pour valider leur applicabilité.

Étant donné le surcoût que représente la prise d'un grand volume d'informations détaillées en production, l'approche que l'on retrouve dans des environnements réels est, la plupart du temps, basée autour de la collecte d'échantillons. Sur un très grand volume, cette technique

semble la meilleure pour obtenir le profil du comportement d'une application avec le moins d'impact en performance. Par contre, cette technique se limite à l'identification des points "chauds" dans l'application, mais le reste de l'analyse doit se faire hors-ligne dans des conditions de laboratoire et il peut arriver que le problème ne se reproduise pas facilement. Par exemple, on va détecter des latences trop élevées ou une utilisation abusive d'une ressource en production, mais les causes seront à déterminer manuellement par la suite, puis les correctifs devront être testés en production pour valider les hypothèses.

Lorsque l'on s'intéresse plus spécifiquement aux mesures de latences et aux délais de réaction, que ce soit dans les systèmes temps réel ou dans des environnement de serveurs, on se rend compte qu'il n'existe pas de méthode de supervision fonctionnelle en production permettant de valider le respect du délai complet de réaction à des événements apériodiques. Les approches basées sur de l'échantillonnage ne sont pas intéressantes ici, car on cherche à détecter les cas rares et anormaux. Il est donc nécessaire d'avoir le maximum d'information possible. Les analyses avec des moyennes risquent de masquer les cas extrêmes. Et les approches basées sur le traçage complet et analyse *a posteriori* sont coûteuses (extraction à haut volume, stockage, traitement) pour des cas très peu fréquents.

Nous sommes donc ici à la recherche d'une solution permettant d'extraire des données très précises sur le fonctionnement du système d'exploitation avec un contrôle sur le volume de données générées ainsi que sur le compromis entre le délai de réception des événements et le surcoût sur la machine tracée et sur le réseau. Lorsque l'on s'intéresse plus spécifiquement aux problèmes de latence, nous sommes à la recherche d'une méthode assez efficace pour fonctionner en tout temps sur le système et qui mesure la chaîne complète entre le moment où l'interruption a été reçue par le noyau jusqu'à la fin du traitement. Pour ces deux aspects, il est nécessaire que les approches soient génériques et applicables dans des environnements très variés et hétérogènes (matériel embarqué jusqu'aux serveurs de type infonuagique). Pour la source d'information, nous savons que nous pouvons nous baser sur l'instrumentation fournie par le noyau Linux (`TRACE_EVENT`).

CHAPITRE 3 ARTICLE 1 : EFFICIENT NETWORK LIVE OPERATING SYSTEM TRACE STREAMING AND PROCESSING

Authors

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

Mathieu Desnoyers
ÉfficiOS Inc.
mathieu.desnoyers@efficios.com

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

Submitted to

Operating System Review

Keywords

Tracing, Streaming, Concurrency, Live Processing

3.1 Abstract

High performance kernel and user-space tracers rely on minimising the amount of synchronization performed on the fast path to limit their impact on the traced system. These optimisations are designed for the use-case where the tracing is stopped before the viewer processes the trace. However, tracers are becoming an interesting source of data for monitoring, so we need to find a way to keep the low overhead and add the possibility to process the trace in near-realtime.

The major challenges for live trace reading are that the events are buffered and each processor generates its own stream of data at its own rhythm. So in order to process the trace and

present the events in a time-consistent manner, we need to add some synchronization without compromising on performance.

In this paper, we present an algorithm to solve the problem of reading multiple independant buffered streams over the network with minimal overhead added, no packet drop for slower streams, and a user-configurable upper-bound for event production-consumption latency. We also provide measurements in different environments.

3.2 Introduction

In computing, tracing is the action of recording information about the execution of a system or an application without stopping it. One intent is to record the behaviour of the application with the minimal amount of disturbance, for *a posteriori* analyzes, it can also be used to extract more information from a system regardless of the added cost.

More and more static trace points are included in user-space applications and in the Linux kernel which makes tracing an interesting source of data for gathering a wide range of metrics about a running system. Usually targeted for developers, tracing is also becoming a valuable resource for system administrators who want more fine-grained or more accurate metrics.

Moreover, the expansion of large-scale Linux server deployments and the ever-increasing number of processor cores per server add new requirements in terms of scaling of the monitoring solutions which can be solved by high-performance tracers such as LTTng Desnoyers and Dagenais (2006), Perf Edge (2009b) and Ftrace Edge (2009a).

Among these tracers, LTTng is the only one that already has support all of the requirements : multiple concurrent sessions (different set of events and parameters), unified kernel and user-space traces, and efficient network streaming protocol, which makes it an ideal candidate for an adaptation into a live system monitoring solution.

Throughout this paper, we refer to live tracing as the action of reading a trace while recording it. This is opposed to offline tracing where the trace is stopped before being processed.

In terms of requirements, we are aiming to provide the trace data at a speed that can be configured by the user with a tradeoff in terms of intrusiveness : the quicker the user wants the data, the more intrusive the tracer is and the more we risk to loose events in the ring-buffer. The main use-case that motivates this modification is the long-lasting low-impact server monitoring sessions. In this use-case, the minimal intrusiveness is the main requirement, we can wait for the data for up to one minute and still provide interesting results. Since this is not the only use-case for live tracing, the latency for data availability is user-configurable

and we provide measurements for different rates. We are not aiming for real-time event generation like strace does, we can always tolerate a delay between the generation and the availability of the events. The high-level architecture of the live streaming is illustrated in Figure 3.1. We can see two different use-cases : the monitoring server that collects metrics, and the administrator station that needs to analyse the trace in near real-time to identify some problem.

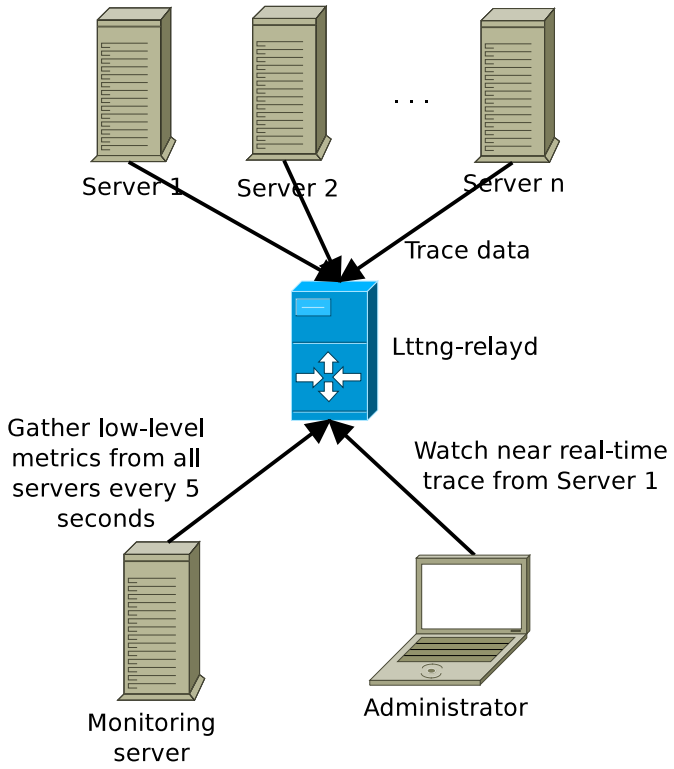

Figure 3.1 LTTng live use-cases

Once we can process live traces, we can merge the traces from multiple hosts and run low-level distributed analyzes in near realtime. By default, the traces from mulitple hosts are merged based on the realtime clock (NTP Mills et al. (2010)), but for more accurate results we can apply advanced trace synchronization algorithms such as the one described in Jabbarifar et al. (2012).

In this paper, we detail our solution to address the synchronization of multiple independant buffered streams with the smallest possible impact on the target system. In Section 3.3, we list the related work relevant to this problem. In Section 3.4, we introduce LTTng, an efficient tracer, and what are the challenges to add the live tracing capability to this tracer. In 3.5, we address the three main problems. Then, in Section 3.6 we measure the impact on the system,

and the latency between an event generation and its availability in various conditions. In Section 3.7, we summarize the results and propose avenues for future work.

3.3 Related Work

The best known protocol for synchronized network streams is RTSP Schulzrinne et al. (1998). This protocol is designed to control audio and video streaming, it does not handle data transport. The transport and the synchronization is performed by RTP Schulzrinne et al. (2003) with a "synchronization source" (SSRC). This protocol can handle multiple streams from multiple sources and synchronize them to deliver a seamless integration of various sources. The sources are meant to be audio or video streams, but could eventually be any kind of data stream. This protocol works by computing the difference between a local time-stamp (RTP Timestamp) and a shared time-reference (NTP by default) and readjusting periodically the relative position of the streams.

When dealing with multiple sources, since the time references are different, a RTP mixer is usually required to aggregate the various streams, perform the synchronization and send the result to the client. This mixer is also responsible for detecting inactive streams and handling the timeouts.

Although this protocol is mature and well-designed, the use-cases from the multi-media streaming and the trace streaming differ slightly. In multi-media, the most important parameter is the interactive capability. This protocol is designed for audio and video conferences where the users can tolerate losing data, as long as the conversations remain coherent. Also, they don't wait for inactive streams, if for some reason a stream does not arrive quickly enough, it is discarded and eventually removed from the list when the timeout is reached. When the protocol is used to stream data without interactivity, the streams are buffered until they can be delivered to the client without any interruption. This behaviour is suitable for streams that have predictable bandwidth requirements. With tracing, the network bandwidth varies a lot depending on the trace volume which can be null or hundreds of thousands of events per second for just one machine, so computing the ideal buffer size is almost impossible for this use-case.

In trace streaming, we are not working in an interactive environment, and even though we want to receive our traces as fast as possible, the most important factors are the completeness of the traces and the time-ordering accuracy. We aim to receive the data in the order of seconds, not milliseconds as in multi-media. For that reason, we cannot use a multi-media streaming protocol such as RTP, we could eventually use RTSP for the control of the tracing

sessions (creation, start, stop, routing, etc.), but this question is out-of-scope of the present study.

The idea of using periodical beacons to synchronize multiple streams is not new. In the patent Davenport et al. (1997), we can see that one source is identified as the time reference and they use special packets (transport system packets) to send "program clock references" at a fixed time interval. This is similar to the design of the RTP/RTSP approach and is targeted at multi-media streaming. Just like RTP/RTSP, the methods described here only address partially the problems we are solving.

New standards and algorithms also achieve synchronization across clients. This is particularly useful for IPTV in order to make sure clients are viewing the same streams at the same time. In order to do so, the standard ETSI TISPAN IPTV described in Stokking et al. (2010) uses a "Media Synchronization Application Server" that collects statistics and computes the delays to introduce in the clients. This information is reported using SIP and might eventually be used with RTP as well. So, even for client-based synchronisation, we rely on buffering and applying delays over reference timeframes sent periodically.

Another area where we need to have synchronized streams is the Data Stream Management Systems (DSMS) such as Arasu et al. (2004). This kind of system is designed to apply queries over independant data streams. It is used for financial analysis (stock trading), sensor networks and various monitoring situations. The interesting part related to our work is the fact that the streams are processed when they arrive and the results of the queries may be partial or require synchronization with other streams. Thus before returning the result of a query, some systems perform synchronization and events reordering. For example, in Dragos et al. (2012), the authors implemented a way to perform fault-tolerant synchronized queries with the use of a binary tree and timeouts. Once again, the problem with this kind of approach is that it can silently discard events that arrive too late and the timeout delay is static. The trace users can tolerate that the tracer discards events on overload situations, but they have to know exactly when and how much events were discarded. Moreover, the method can be costly in terms of memory and CPU usage for bursty streams. A method to measure the overhead and the latency introduced by the synchronization for this kind of system is presented in Gorawski and Chrószcz (2012). This paper is interesting because it clearly shows that the synchronization methods currently in use are really costly and difficult to predict. The major difference with our system is that we are working in a packet based environment so we receive sets of events instead of one event at a time, but we still need to perform a synchronization between the different streams of packets.

3.4 Design Challenges

3.4.1 Independant streams

In order to collect events, software tracers add code to be executed at specific sites in instrumented programs. This code is responsible to extract the data when tracing is active. Because the tracer shares the same physical resources as the traced application, the design of a tracer depends on the target application and the acceptable performance impact. For basic debugging of a single-threaded program, the developer could choose to simply output the relevant information into a log file, but if the target is the kernel or a parallel application, we don't want the tracer to introduce much latency or to change the behaviour of the application. To tackle this issue and still keep an accurate temporal representation of the events, high-efficiency tracers such as LTTng Desnoyers and Dagenais (2006) rely on a monotonic clock synchronized between cores and allocate a trace stream per CPU. That way, they can store in a trace stream all the events generated by a specific CPU sequentially and avoid any locking on the critical path.

When a system has multiple CPUs, the events are reordered globally in post-processing by relying on the monotonic clock guarantees. This method diminishes significantly the impact of the tracer on the system as the number of core grows, it works fine when reading the trace after it has been stopped, but it introduces a significant challenge when trying to read the trace while it is being recorded. Indeed, the trace viewer must be sure to have all the data up to a certain point in time before displaying an event. If it receives an event that was recorded before an already displayed event, we can only assume that the state representation of the trace is invalid and the analyzes that rely on the data become inaccurate. The easiest method would be to discard the events coming from the past, but this is not an acceptable solution because it would be silent and would add another point of failure in the chain. The tracer can already discard events in case of overload when it cannot extract the tracing data as fast as it is created, but this situation is detected while it is happening and can be fixed by configuration or hardware upgrade.

3.4.2 Buffering

Another characteristic of high-efficiency tracers is the buffering of trace events. If the process that extracts the trace from the tracer (the *consumer*) writes to disk or sends over the network every time an event is generated, it will have a significant impact on the traced system and the tracer risks losing events since it is too busy doing I/O operations (it might also induce a feedback loop if we are tracing I/O operations). To avoid these problems, buffering is used.

At the beginning of a trace session, the tracer allocates a ring-buffer based on the user's preferences and the number of CPUs. This ring-buffer is divided into multiple sub-buffers to allow the tracer to continue writing while a process reads some trace data already recorded without requiring any locking mechanism between the two processes.

In LTTng, the events are enabled inside channels, the user can enable an arbitrary amount of channels and for each channel it can specify the number and the size of each sub-buffer allocated per stream. For each channel, the ring-buffer allocates a stream per CPU. During the tracing session, once a sub-buffer is full, the tracer wakes up a process responsible to collect these sub-buffers and to write them to disk. During this operation, the tracer doesn't stop, it requests a new sub-buffer from the ring-buffer and continues to write the events. This mechanism allows the communication between the tracer (running inside the traced application or the kernel) to be concise and thus limits the perturbation on the system. Once again, this is an interesting way to record traces that will be read once the tracing is stopped, but depending on the size of the sub-buffers and the rate at which events are generated, it can complexify the task of reading the trace as it is being written : we don't want to wait too long for a sub-buffer switch, but in the meantime, we don't want to force the tracer to switch sub-buffer too often because it increases the impact of the tracer on the system and the load on the I/O subsystem.

3.4.3 CTF and metadata

One important aspect of LTTng is that it uses the CTF format Desnoyers (2011) to represent the traces. One particularity about this format, is that it is self-described. Indeed, a metadata file, generated by the tracer, describes the layout of the data written in the trace files. It is unique for each trace and is absolutely required to read the trace. This metadata is stored in a separate file in the trace output folder. New metadata is generated at the beginning of a session and when each tracepoint configured to be activated becomes available (when the application starts or when the kernel module containing it gets loaded). The metadata is also buffered in the tracer and only flushed to disk (or network) when the buffer containing it is full. The only real difference between a metadata stream and a data stream from the tracer's point of view, is that there is only one stream of metadata, and the tracer will never loose any data targeted to this stream, it will block instead to prevent the trace from becoming unreadable. This implies some synchronization when tracing starts for a session, and when an application starts or a module gets loaded, but it is done only once. In order to read the trace while is it being generated, we need to have all the metadata required to read the data we are receiving. This step can be quite challenging because new events could be enabled

during a tracing session. Moreover, since the metadata size is almost never an exact multiple of the sub-buffer size and there is no periodical flush on this stream by default, we usually have metadata not yet written to disk that remains in memory until the trace is stopped, which forces a flush of the current metadata sub-buffer. So we need to find an efficient way to extract the metadata while recording the trace to guarantee the viewer that it will never be able to read data for which it does not have the metadata.

3.4.4 Challenges for live reading

These three characteristics are the main challenges that we need to address in order to provide a way to read the traces as they are generated. The intent here is to limit the overhead introduced by this functionnality on the system being traced while providing an upper-bound in time for an event to be available to a trace viewer.

So we have four possible situations we need to take into account :

1. All CPUs are active and produce enough data to fill the sub-buffers quickly enough to prevent the viewer to wait too long for any stream ;
2. Some of the CPUs are producing very little data compared to other CPUs or no data at all ;
3. All the CPUs are producing very little data ;
4. No CPU is producing any data.

Regarding the delay between an event generation and its availability to the trace viewer, it depends on the use-case and the acceptable overhead on the system. The constraints are really different in a monitoring situation, on production servers where the user expects statistics every five minutes, and does not want any additional load on its servers, compared to a development environment where the user wants to see all the data as soon as it is produced and does not care as much about the additional load introduced by the tracing.

3.4.5 LTTng live architecture

Before digging into the challenges of live reading, let's first introduce the high-level overview of how the live reading works from the trace producer to the trace reader. Figure 3.2 illustrates the flow directions, all TCP connections imply that the processes can either run on the same host or remotely, the Unix sockets are local to the same host.

In LTTng, the session daemon (*lttng-sessiond*) is responsible to create and manage the sessions. There is one session daemon started as root if the user wants to trace the kernel, and one session daemon for each user interested in user-space tracing.

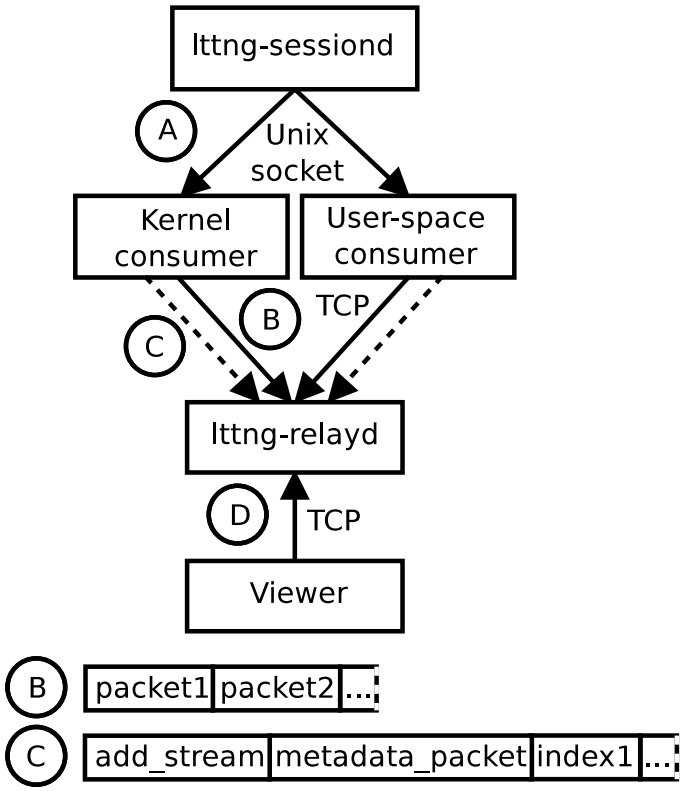

Figure 3.2 LTTng Live Architecture

When a session is created, the session daemon communicates with the tracers to create the channels and streams. These streams are sent to a trace consumer *lttng-consumerd* (A in the diagram). The consumer is responsible for extracting the data from the tracer and writing it to disk or over the network. There is one consumer for each domain (kernel, user-space 32-bit, user-space 64-bit).

If a trace is configured to be sent over the network (which is mandatory in live tracing), a relay (*lttng-relayd*) must be started on the receiving host to accept the trace packets and write them to disk. The relay is a multi-threaded application that both receives trace data from the consumer (B for the data connection and C for the control connection) and handles the live reading communication with the viewer (D). It is in this daemon that most of the optimisations discussed in this paper are introduced, because the biggest challenge is to minimize the inter-dependency and locking between the data receiving threads and the live-reading threads. The relay is designed to handle multiple concurrent sessions from multiple hosts.

The trace format specification is open so there are readers implemented in various platforms and languages, for different requirements. The live-reading protocol must also accomodate

these viewers, so the protocol is implemented over a TCP connection, even if the tracer and reader are on the same host. For our measurements, we use the Babeltrace text-only client. For now, a live session can only be processed by one viewer at a time.

3.5 Live reading

3.5.1 Synchronisation between independant buffered streams

In LTTng, a trace is divided into multiple independant streams (the buffers). Each stream S is divided into packets P (the sub-buffers) that contain a sequence of events E that are generated by a single CPU in the order they are produced. Thus a stream is a list of packets ordered sequentially based on the time-stamp they are created.

$$S_i = \{P_1, P_2, ... | P_i < P_j\} \tag{3.1}$$

A packet is a list of events ordered sequentially based on the time-stamp they are generated.

$$P_i = \{E_1, E_2, ... | E_i \leq E_j\} \tag{3.2}$$

In a tracing session, we can create multiple channels in which we can enable different events. Each channel C_x has one stream S_y per CPU, so that

$$\{S_0, S_1, S_2, ..., S_n - 1\} \in C \text{ where } n = |CPU| \tag{3.3}$$

So for any kernel tracing session, we have

$$|kernel\_streams| = |channels| * |CPU| + 1 \tag{3.4}$$

For user-space tracing sessions, we can have two kinds of configurations : *per-uid* buffers or *per-pid* buffers. In the *per-uid* buffer configuration, all the instrumented processes running with the same UID write into the same set of buffers. With *per-pid* buffers, each instrumented application requires a new set of buffers. The default configuration is *per-uid* buffers, so in a session with kernel and user-space tracing enabled, with only one user running both 32-bit and 64-bit instrumented applications, we have

$$\begin{aligned} |streams| = (&|kernel\_channels| + \\ &2 * |userspace\_channels|) * |CPU| + 3 \end{aligned} \tag{3.5}$$

To represent the events in the order they are produced on the system, we need to check the time-stamp of each event not yet displayed in every stream and select the one with the lowest time-stamp.

$$\begin{aligned} displayed\_event = head(S_x) \text{ where } T(head(S_x)) \\ \leq T(head(S_n)) \end{aligned} \tag{3.6}$$

In order to do that, we use a heap that contains the head of all the streams to handle the merge sort based on the event time-stamps. Every time we process an event, we remove the top of the heap. This implies that we know the state of all the streams. In the case where two events have the same time-stamp, we rely on the stream name (based on the CPU number) to enforce the second-order sorting and make the analyzes reproducible.

When we are processing a live trace, the relation in Equation 3.6 might stall for an unknown and potentially infinite amount of time depending on the stream activity. At any point in time, the condition to continue processing events is expressed in Equation 3.7 and is the condition we are optimizing in this paper.

$$\begin{aligned} last\_displayable\_event = E_x \text{ where } T(E_x) \\ \leq end\_time - stamp(S_n) \end{aligned} \tag{3.7}$$

From the specification of CTF Desnoyers (2011), a packet is "a sequence of physically contiguous events within an event stream". In LTTng, each sub-buffer contains exactly one packet. At the end of a packet, the viewer opens the next packet in the event stream, reads its header and continues the merge-sort event by event. In a situation where the trace is still being recorded, we cannot allow the viewer to read the trace files by itself because it is not aware of the state of the trace. For efficiency reason, we don't send stream data event by event to the viewer. Instead, every time the viewer needs a new event packet, it requests it from the *relay*. If the packet is available, the viewer receives it immediately, otherwise it retries until it is ready.

Since each trace stream is buffered and each channel can have a different buffer size, event size and event generation rate, we have no way to know precisely what is the delay before a packet is ready to be processed by the viewer. So, waiting an arbitrary amount of time does not give us the guarantee that we will have the data we are waiting for.

For the situation where all the streams are active and produce enough data to not stall the viewer (situation 1 in section 3.4.4), the synchronization between streams is transparent for the viewer : just like when the viewer reads the trace from disk, every time it needs to switch

packet, the next one is available immediately. In this situation, the live trace viewing works just like with an offline trace.

When some streams are not producing as much data as others (situation 2), we start introducing delays in the trace viewer. To help minimize the impact of this problem, LTTng already has a "switch-timer" parameter that can be configured on a per-channel basis, it forces the tracer to flush a sub-buffer at a certain rate even if it is not yet full. The problem with this timer is that it is configured on a per-domain basis, so the user would have to configure exactly the same for each domain and start it at the same time. However, the bigger problem is that this timer does not do anything if a stream is empty, so the viewer is not informed that the stream is empty.

The proposed approach to solve this problem is to introduce a session-wide timer : the "live-timer". This timer is user-configurable depending on the maximum delay the user can wait before obtaining the data. For now, the timer is configured when creating the session and cannot be changed after that. Contrarily to the per-channel switch-timer, this timer is the maximum amount of time the user can wait for a stream to be informed about its state. If the stream is active and produces data faster than the timer threshold, the timer is not useful, but if the stream does not produce enough data to fill a sub-buffer before the threshold, then the timer handler checks if there is any data to flush. The intent here is to minimize the wake ups caused by the sub-buffer flushes. This optimisation is especially important for kernel tracing because, in this mode, only the CPU that owns the stream can flush it, but any CPU can read the position counters. So we require only one active CPU to handle the timer. For user-space tracing, any CPU can read and flush any stream so the problem is easier to solve.

To implement this solution, we use the snapshot API already available in the tracer. This API allows the trace collector to check the positions of the reader and writer in the ring-buffer without forcing a packet switch. That way, when the timer fires up, the trace collector checks the positions in the ring-buffer for the concerned stream and determines if the position of the writer moved since the last check. If it moved, it means that some data was produced for this stream and it needs to be sent to the trace viewer. Otherwise, it means the stream is inactive and we just inform the viewer that it can continue processing the other streams up to the current time-stamp without forcing an empty packet switch. This "beacon" guarantees the trace viewer that it won't receive data from the past for this stream and we don't introduce unnecessary load on inactive streams.

The idea of disabling the timer when the tracer is completely idle (no stream produce any data) has been explored, but was discarded because the use-case for this solution is to eventually be able to merge the traces from multiple hosts in real-time. If we disabled the timer on

a host that does not produce data, it would prevent the viewer from processing the streams of other active hosts or we would need to coordinate all the hosts which could become costly.

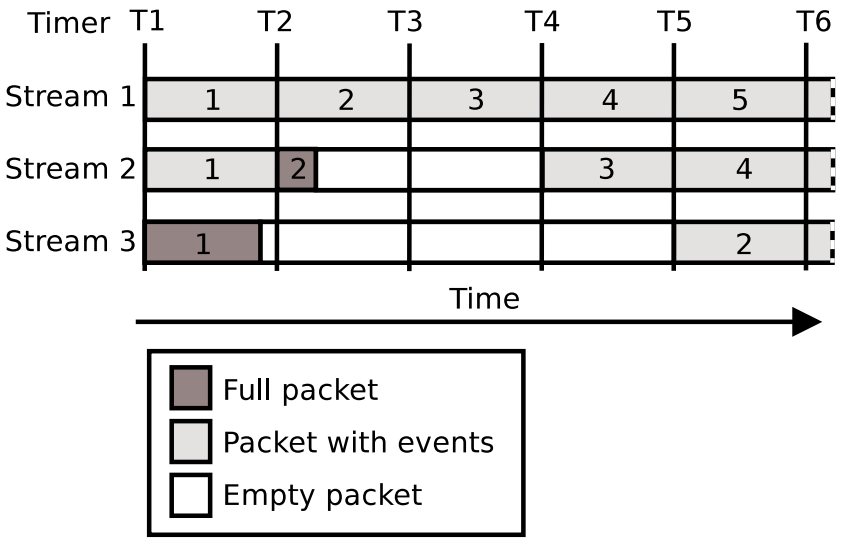

Figure 3.3 Stream temporal consistency example

Figure 3.3 is an example of the possible state of three concurrent streams. When the reader receives the packet 1 of stream 3, it has to wait up to T2 (timer) to receive the packet 1 of stream 1 and 2. At T2, it is also informed that there is no data for stream 3 between the end of packet 1 and T2, so it can read completely the packet 1 of all the streams. Then, if the reader does not request data before T5, then at T5, it can ask for all the data packets that have been produced and won't notice the gap that happened in stream 2 but will receive a beacon for stream 3. Once all this data has been received, the viewer will have to wait up to T6 to continue processing the trace.

3.5.2 Creation of streams during the session

Although the tracer prevents the creation of additional channels after the session has started, new streams can be added while the session is active. A common use-case for this scenario is the per-pid user-space tracing. In this mode, a set of streams is created every time a process containing events of interest is created. These streams are dedicated to the process and are destroyed when the process exits. This situation is also true, although less common, in per-uid user-space tracing. If a user-space session has been created by root with per-uid buffers, a set of streams is created for each user who starts a process that contains enabled trace events. Finally, a new stream can be created if a new CPU appears online (*CPU-hotplug*) while a session is active. Since there is one stream per-channel per-cpu, when a new CPU

appears online, we have to create a new stream in each channel.

If a viewer is reading the trace from an active session and new streams are added, it needs to know about these streams before any events are generated.

The key to make sure we can send the new streams to the viewer before it reads events from other streams is to centralize the creation of live streams on the relay. The relay listens over the network, so it can receive packets from many different hosts and sessions. When a new stream is added to a session, the consumer sends the stream information to the relay (session ID, path, parameters) and the relay replies with a unique network stream identifier. This identifier is used every time the consumer sends a data packet belonging to this stream. That way, the relay can look up the stream information and write the packet into the right trace file.

Since the relay is a central point for the creation of new streams and for the live reading, we redesigned the handling of streams inside the relay to allow performing per-trace look ups in a RCU hash table. We also added an atomic compare and exchange on a flag in a data structure shared between the streaming side and the viewer-side to detect the stream creation from the viewer-side without requiring taking any lock.

3.5.3 Buffered packets

Since the trace data is contained in trace packets of variable size, the trace readers usually index the whole trace by packets in order to permit a quick navigation in the trace streams, while processing the trace. An index contains data about each packet such as the time-stamp at which it was created, the time-stamp at which it got filled, the size of the packet, and the offset inside the trace file where the packet is located. In the trace reader, the indexes are stored in an array per stream. When the reader finishes reading a packet, it fetches the next index from the array, seeks in the trace file at the offset stored in the index and reads the trace from there. The index also allows to quickly read data from anywhere in the trace, instead of looking for a specific time-stamp in the whole trace, we can use the time-stamps begin and end of the packets from the index array to identify the packet that contain the event we are looking for.

Although the index is optional when reading an offline trace, for live tracing it is mandatory because it allows the whole infrastructure to rely on buffered packets without requiring the intermediate processes to read the trace. Since any stream can produce data at its own pace, we need to know if we have data to send to the viewer whenever it asks for the next packet. The consumer does not read inside the data packets, it does not have the knowledge of how

to process a trace or even the notion of packets, it just extracts opaque data packets from the tracer. Moreover, a packet does not need to be the size of a sub-buffer to be sent. As detailed in section 3.5, a timer can force a buffer switch which triggers an extraction from the consumer. So the packets can have a different size depending on various conditions.

For the same efficiency reason as the buffered packets in the tracer, the viewer reads whole packets from the trace, instead of one event at a time. Since, by design, the packet content is also opaque to the relay, it needs to rely on additional information produced by the tracer : the index. The index is received from the tracer by the consumer every time a packet is ready to be extracted. As soon as the packet has been sent to the relay on the data connection, the index corresponding to this packet is sent on the control connection. When the relay has received the packet and its corresponding index, it knows that a new packet is ready to be sent to the viewer.

When the viewer needs a new packet for a stream, it asks the relay for a new index. If the relay has received an index it never sent, it sends the index, otherwise it informs the viewer to come back later. When the index is received by the viewer, it requests the data packet corresponding to the index : an offset in the trace file and a size. The relay extracts the requested data from the trace file and sends it to the viewer.

With this algorithm, the relay can store all the data to disk as fast as possible without any complex processing and locking between its threads, and the viewer can process the trace at its own rhythm without impacting the memory usage of the relay. The relay also acts as the synchronization point between the tracer and the viewer. Also, the viewer can disconnect at any point in time and restart processing the trace from the start or anywhere in the trace.

In Algorithm 1, we present the pseudo-code in the relay that sends the index to the viewer. Most of the complexity of this algorithm comes from the on-disk ring buffer covered in section 3.5.4

3.5.4 On-disk ring-buffer

The downside of caching the data at the relay level is that there is no way to delete the data from the relay. So for long-lasting sessions, it could end up storing hundreds of gigabits of trace data even after the viewer has received it. A solution introduced in a previous version of LTTng for offline traces was to create an on-disk ring buffer. The user can specify the maximum size of each stream file and the number of files to keep for each stream and the consumer (or the relay) rotates on these files. For live-reading, the same mechanism is required to avoid filling up disks but still allow the reader to catch-up with older data in case it does

```
viewer_get_next_index()
if no new index then
    return
/* Check stream status and handle the trace file rotation */
lock viewer_stream_rotation_lock
if stream still active then
    if trace file overwritten then
        switch to next index file (trace file rotation)
    if reading the active trace file then
        if last index already sent then
            unlock viewer_stream_rotation_lock
            return inactive beacon
        if all indexes already sent then
            unlock viewer_stream_rotation_lock
            return return later
else if trace file closed in writing and last index sent  then
    return stream_hung_up
else set stream closed in writing
unlock viewer_stream_rotation_lock
check_new_metadata()
check_new_streams()
/* Actually read the index */
lock viewer_stream_rotation_lock
lock overwrite_lock
check if we got overwritten (trace file rotation)
if overwritten then
    unlock overwrite_lock
    unlock viewer_stream_rotation_lock
    return try again
read the index
unlock overwrite_lock
unlock viewer_stream_rotation_lock return index
```

Algorithm 1 Find the next packet index

not start reading at the same time the trace starts.

For offline traces, the mechanism is the basic ring-buffer algorithm, but as soon as a client is reading the trace, the problem becomes more complex. The reader and the writer can share the same trace file as long as the reader follows the writer, but the reader must never prevent the writer from writing because it would slow down the trace extraction and increase the number of discarded events. So, if the writer opens a trace file currently opened by the reader, it informs it to abort its reading and skip to the next trace file. This is one of the few cases where the data writing thread and live-reading thread of the relay lock each other (as illustrated in Algorithm 1). When a reading aborts, the viewer is informed to retry and it then receives data from the next available packet. Currently, the viewer does not know how many events or packets were skipped because of the trace file rotation.

3.5.5 Metadata availability guarantee

As explained in Section 3.4.3, the metadata generated by the tracer is absolutely required to read a CTF trace.

In a LTTng session, there can be multiple running traces at the same time and each trace requires its own metadata. For example, if the kernel and user-space tracers are both active and there are 32 and 64-bit applications running, the session is composed of at least three traces, one for each domain. The number of streams can be computed with the Equation 3.5. When a client attaches to a live LTTng session, it first receives all the streams, and for each trace it receives a metadata stream. Before asking for data packets from the relay, the client must first ask for all the metadata available. Once this process is done, it can ask for indexes and then data packets, and start reading the trace.

When the client is processing a live trace, if new events are enabled and new metadata is generated, the relay informs the viewer that it needs to ask for new metadata before processing new packets. As long as the viewer has not asked for the metadata, the relay prevents the viewer from accessing new data packets.

The relay does not have any notification channel to the viewer, the viewer always asks the relay for new information (metadata, indexes, packets), so between two requests from the client, the relay cannot intervene. In order to offer the guarantee that the client is never missing any metadata, we rely on the fact that metadata is extracted in priority by the consumer, so before any data packet with new event types becomes available to the relay, we already know that new metadata was generated and we can inform the viewer when it asks for new index or data packet. Once again, for this synchronization, we don't need

any lock between the streaming and live threads in the relay, but we make sure that the metadata streams have absolute priority over any other streams. This modification could lead to starvation if the metadata stream was flooding the connection, but the metadata is usually less than 8kB, so we can use this solution without locking out the consumer too long (the default packet size for kernel being 256kB).

Figure 3.4 illustrates what happens when new metadata is added and the client tries to get a data packet. This ensures that the client is always in a position where it is safe to read the trace.

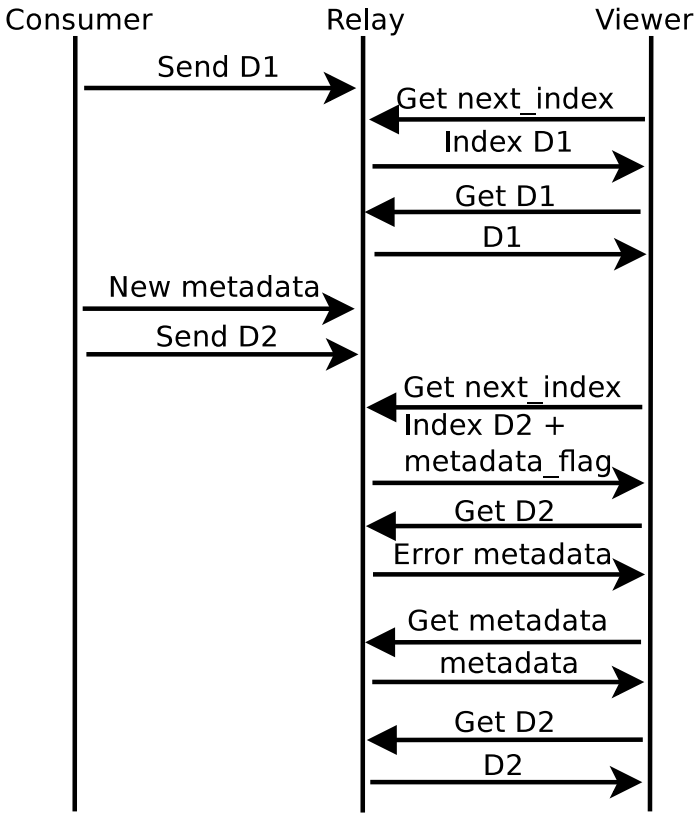

Figure 3.4 Metadata consistency protocol

3.6 Measurements

3.6.1 Latency between event production and reading

The objective of the live-reading protocol is to allow the user to process the events closely after they have been produced. The exact delay between the production and the availability of an event to the viewer is determined by multiple factors including the live-timer value, the

event rate, the network condition, the system load, the size of the tracing buffers, and the processing speed of the viewer. As explained before, we are not aiming for a near-realtime approach where all events are available as soon as they are produced (like in *strace*), but we expect to receive the data on the relay at the rate defined by the live-timer or faster. After that, the client has to ask for new packets for each stream, so we have additional delays here.

This test was performed on an Intel Core i7 920 with 6GB of RAM with a test program that outputs one event every 250μs. We used the mainline version 2.4.1 of LTTng and Babeltrace 1.2. The only custom modification introduced for the purpose of this test is the computation of the time difference between the event generation and the actual output.

In this test, we doubled the period of the live timer until we reached the point where the tracer always filled buffers before the timer fired, and we measured the delay between the event production and the reading time (on the same machine).

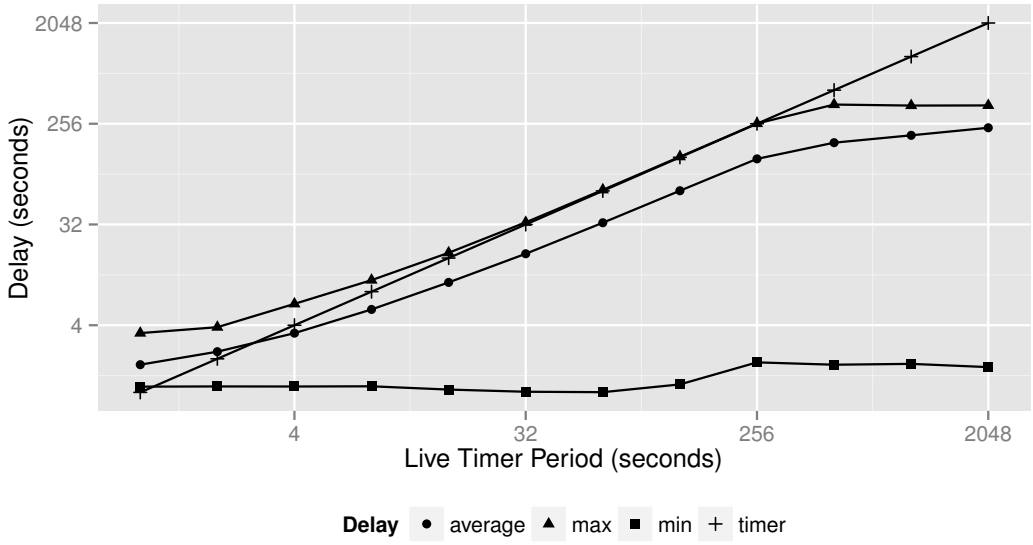

Figure 3.5 Latency between event production and reading

In this graph, the "Timer" line represents the maximum delay the user would want between an event produced and the time it is processed in the viewer. This time does not take into account the time taken by the viewer to ask for new packets. Thus as we expect, the actual maximum latency is higher than this limit for half the test (up to 64 seconds), but the average is below this limit starting at a timer of 4 seconds. The minimum reading time is always close to zero because the processing time of the events is extremely fast (more than 430000 events/s on this machine) and the viewer processes an event packet much faster than its rate of production.

3.6.2 Live-timer performance impact

The main overhead added to LTTng comes from the new live-timer. In this measurement, we compare the performance impact of LTTng on a system with different timer values and study how the consumer behaves when we increase the number of streams (CPUs) with the same tracing load. Since it needs to iterate over all the streams every time the timer fires, we need to confirm that the optimisations put in place to handle the empty streams are scaling.

For this test, we generate 4000 events per second until we have generated 150000 events. The test is run on a 64 cores machine, we start with only one core enabled, run the test 20 times, then enable 2 cores and continue doubling the number of cores until we have enabled the 64 cores. The baseline in this test is the "No timer" test, where we send the trace to a relay, but no live timer is enabled, so the packets are only extracted when they are full.

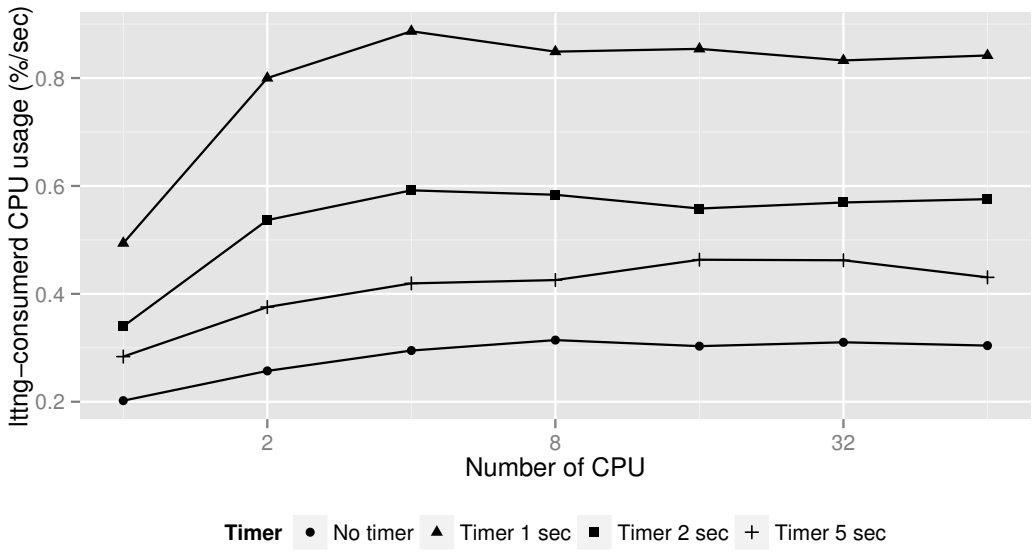

Figure 3.6 LTTng Live Scaling

The Figure 3.6 represents the average total CPU usage of lttng-consumerd from the 20 runs for each CPU configuration. For most of the tests, the coefficient of variation is below 5%. Only for the configuration with 5 seconds of live timer and 16 or 32 cores we see a coefficient of variation of 40%.

From the results, we can see that the impact of the live timer is proportional to its period and that increasing the number of cores with the same number of events generated does not add more load on the system even though the consumer has to iterate over each stream every time the timer fires. Between 1 and 4 cores, we see that the load increases, and after that

it stays linear, this variation is due to the scheduling, the program is mostly scheduled on 4 different cores, regardless on the number of additional cores available.

3.7 Conclusion and Future Work

We presented a new way to synchronize multiple independent buffered streams over the network. This new method is based on the compromise between interactivity (user-defined maximum delay), reliability (no packet loss) and minimal overhead on the traced system (with the buffering). We also enforce that all metadata is received before any data packet. Finally, we provide a way to record long lasting sessions while controlling the disk space required with on-disk buffering. This set of new techniques, all implemented in LTTng, allow the user to process a trace while it is being recorded with a minimal overhead on the traced system compared to normal tracing.

As of now, the algorithm to synchronize multiple streams has no way to disable the live timer when the system is not producing any traces. Adding this feature would result in less energy consumption. Also, having the possibility to change dynamically the live timer period would give more flexibility depending on the workload. Adding these improvements requires to take into account the fact that a user may want to have multiple tracing sessions from multiple hosts streaming at the same time.

Another possible improvement is the feature to inform the viewer about the number of discarded events that happened because of the on-disk trace file rotation (see section 3.5.4 for details).

Finally, since this work is aimed towards large-scale infrastructures, the relay infrastructure should become more scalable and reliable. We should be able to spawn new relay servers and use them with existing live tracing sessions. These improvements will require an iteration over the current protocol to make it more local stateless.

3.8 Acknowledgments

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to David Goulet and the Efficios team for the help in the implementation and testing, and to Naser Ezzati for the reviews.

CHAPITRE 4 ARTICLE 2 : RUNTIME LATENCY DETECTION AND ANALYSIS

Authors

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

Mathieu Desnoyers
ÉfficiOS Inc.
mathieu.desnoyers@efficios.com

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

Accepted in

Software : Practice and Experience

Keywords

Latency, Tracing, Cloud Computing, Real-Time

4.1 Abstract

Detecting latency-related problems in production environments is usually done at the application level with custom instrumentation. This is enough to detect high latencies in instrumented applications but does not provide all the information required to understand the source of the latency, and is dependent on manually deployed instrumentation. The abnormal latencies usually start in the operating system kernel due to contention on physical resources or locks. Hence, finding the root cause of a latency may require a kernel trace. This trace can easily represent hundreds of thousands events per second. In this paper, we propose and evaluate a methodology and efficient algorithms and concurrent data structures to detect and

analyze latency problems that occur at the kernel level. We introduce a new kernel-based approach that enables developers and administrators to efficiently track latency problems in production, and trigger actions when abnormal conditions are detected. The result of this study is a working scalable latency tracker and an efficient approach to perform stateful tracing in production.

4.2 Introduction

Monitoring resources usage is a mandatory practice in production environments. The tools commonly deployed extract metrics from servers to graph resources usage over time, and eventually generate alerts. The resources monitored include processor, memory, network and disk usage. The load average is also a commonly used metric to detect if a server is saturated. When the servers are in a controlled state, with predictable resources usage, this monitoring is enough to detect the major problems.

The latency-specific monitoring is usually performed in production-critical applications. For example, a web application can compute the time difference between when a request is received and served to provide metrics and alerts based on this value. This process implies that the application developers add this instrumentation in their code and that the monitoring tools interface with this information source. Adding instrumentration in an application adds some cost, so it is usually done at high level, or is only enabled a fraction of the time (for example in Twitter Zipkin zip), to provide an overview and eventually a breakdown of where the time is mostly spent. This allows sampling the latencies, without impacting too much the production, and it can orient the search for the latency root causes in high usage services.

However, there are cases where a latency is caused by an uninstrumented application, by a resource contention, by a scheduling problem or by an external dependency (for example : interrupt handlers and virtualization). Covering all of these cases with custom instrumentation increases significantly the amount of instrumentation, and software maintenance, required at each deployment and update.

To circumvent this problem, some system deployments include a phase of micro-benchmarking, where the operators verify that the machine satisfies the requirements before starting the production on it. These benchmarks, however, cannot be run again during production, so if the conditions change during production, they can only rely on the existing monitoring information to detect the change and react. In a virtualized environment, such as cloud computing, the contention on the physical resources can change because of other virtual machines spawned on the same physical server.

Finally, in a production environment, the most interesting latencies are the highest ones, the outliers, the ones that are rare but cause serious problems for the end-user applications, and can reveal serious unhandled problems. The probability of detecting these problems with a sampling-based approach is low and they are completely invisible to average-based monitoring.

When trying to understand this class of sporadic low-level high latency problems, which are difficult to reproduce outside of production conditions, system administrators face a challenge in their problem-solving methodology. They can usually identify what is the problematic subsystem by running some of the usual diagnostic tools on the server Gregg (2013). After that, they are either bound to do a trial and error test on solution ideas (update the kernel, drivers, other software, etc), or try to capture more information on the faulty subsystem, and hope for the problem to reappear, even though they changed the experiment conditions.

At this stage, tracing is the class of tools used. It can be at the process-level (*strace*, *perf* Edge (2009b)), at the network layer (*tcpdump* Jacobson et al. (1989)), and the kernel-level (*ftrace* Edge (2009a), *perf*, *LTTng* Desnoyers and Dagenais (2006), *SystemTap* sys), or completely *ad-hoc*, added just for the occasion in the target application (with *printf*(), *logger*, etc). All of these tools have their own advantages and drawbacks, but one common factor for all these tools is the additional work on the target system caused by the instrumentation and the trace extraction. This may be enough to change the resources usage, the scheduling, the locking, and change completely the experiment conditions. When all of this is in place, the problem has to be detected again while tracing. Depending on how long it takes for the problem to reappear, the traces can be huge and require an expert and advanced analysis tools to understand them.

In this paper, we are looking for the most efficient way to detect latency outliers and extract background information to understand the root cause. This study serves as the basis for developping the new *latency_tracker*. This module aims at measuring and executing actions upon detecting high latencies, while providing these guaranties :

— Provide a flexible framework for instrumenting various parts of the kernel.
— Work in every kernel execution context (including *NMI* and *MCE* handlers).
— Scale on concurrent workloads with a small and predictable overhead.
— Work in production environment as background monitoring.

The result is a scalable approach to generic runtime latency detection, and an efficient algorithm to perform stateful tracing.

We start by surveying the existing related work in Section 4.3, we then present the proposed

architecture in Section 4.4, we present some analyses implemented with this new mechanism in 4.5 and we measure the overhead in Section 4.6. Finally, we discuss the results and ideas for future work in Section 4.7.

4.3 Related Work

4.3.1 Latency analysis

With the number of large online services always increasing, a lot of research is already in progress to eliminate as much latency as possible throughout the software and hardware stack. The kernel itself is known to be a major factor in the latency of network applications. The intent here is to serve as many concurrent clients as possible with the smallest number of servers. Applications like Chronos Kapoor et al. (2012) remove the kernel and the network stack from the critical path of network applications, and reduce the latency of certain applications by a factor of twenty. A complete breakdown of the latency of all the components involved in a network communication is presented in Larsen et al. (2009) and it clearly shows that most of the time is spent in the network stack.

Identifying latencies in specific subsystems is useful to fix one particular bottleneck but, when operating real services, the load usually relies on the performance of multiple subsystems, and it can be difficult to get an overview of where the time is spent. In Leverich and Kozyrakis (2014), Leverich et al. classify the latencies of real-world co-located applications in three categories : queueing delay, scheduling delay and thread load imbalance. In addition, they explain most of the delays and give insights on the parameters to take into account, when trying to optimize the usage of servers while still keeping a fast response time. In the case studied, they managed to co-locate computing services (such as analytics) on *memcached* cache servers in an architecture inspired from the Facebook use-case.

Under-usage of computing resources is a common problem. In this 29-days trace of a Google cluster Reiss et al. (2012), we see that the main factor for under-usage is a lack of accurate estimate of resources usage. In more interactive applications, we see that engineers and cloud load balancers tend to leave a lot of headroom, because of the unknown high-impact latencies that might occur. For example, in a study on server utilization in public clouds (Liu (2011)), we see that the estimated server utilization on Amazon EC2 is below 20%, because there is no reliable way to guarantee a certain quality of service, in terms of responsiveness, due to the lack of accurate measurements and understanding of these conditions.

Maximising the usage of servers, while maintaining a low latency, is also of interest for energy efficiency research. For example, in Lo et al. (2014), researchers at Google identified patterns

of usage, estimated real world Service Level Objectives and built dynamic controllers to dispatch the work, while meeting the users expectations in terms of latency. This research confirms the need for more low-level characterization of usage and workloads.

We can see that a lot of research is currently targeting the optimization of the high-impact delays in the production chain, but we did not find any paper or project working on actually detecting and explaining the latency outliers in real world production environments.

4.3.2 Latency measurements

In order to measure the time spent in serving a request in production, we need an efficient request tracking mechanism. SystemTap sys is an efficient tool for dynamically hooking analyses in the critical path of a subsystem in the kernel. The language allows to quickly create a custom analysis probe. It is often used for statistics because it is designed to avoid perturbating the system too much ; a lot of work is done in the generated code to account for the time spent in each probe and return if it is higher than a specified threshold. We experimented with SystemTap for our use-case and identified that its generic design makes it difficult to scale analyses for concurrent applications. Since we are tracking the latency between entry and exit events that can occur randomly on any processor, the safe approach used in SystemTap, to require exclusive locks to access elements in associative arrays, imposes a high impact on our workload. Moreover, the SystemTap scripting environment is self-contained, which makes it difficult to emit tracepoints and be called directly from an external module or the kernel itself. The results from our experiments in scaling are available in Section 4.6.

To measure the entries and exits of various subsystems in the kernel, we could use the tracepoint infrastructure, with one of the available kernel tracers (perf, ftrace, LTTng), and compute the latency metrics offline. However, the amount of data to extract can be in the order of megabytes/sec, which is very high for only identifying outliers.

The other methods to measure the performance of a system rely on polling the *proc* filesystem. It usually consists of usage metrics that show the average load of the system. Most of the common production monitoring tools such as Ganglia Massie et al. (2004), SMILE Uthayopas et al. (1998), or Parmon Buyya (2000) rely on local agents or the SNMP protocol to read these counters.

Ftrace has latency analyses targetted specifically for real-time systems Rostedt (2009). They are hardcoded in the kernel and provide an interesting view of what caused the interrupts or the preemption to be disabled for some time, with a trace leading to this event. This kind of analysis is exactly what we are trying to achieve, but for a more general purpose than hard

realtime systems. Moreover, we need more flexibility in terms of reaction to an interesting event.

The other known realtime analyses are micro-benchmarks (Calandrino et al. (2006), Betz et al. (2009)) aimed at certifying that a hardware architecture satisfies the response-time needs for a certain application. Tools like *hackbench* and *cyclictest*, part of the $rt-test$ tools suite, are often used for this purpose. They cannot be run with production data, since they are benchmarking tools.

4.4 Architecture

In this paper, we propose a generic approach to track latency events in production environments. Our main goal is to provide an efficient way for developers and advanced users to measure the delay between two or more related events, and act on this measurement from the critical path of the kernel. Thus, in this section, we are trying to find the best way to keep track of thousands of concurrent events and provide enough flexibility and background information to allow the users to understand the source of the latency.

This work aims to find the appropriate balance between generating too much data for offline post-processing and intruding too much on the critical path.

The high-level architecture of our work is presented in Figure 4.1.

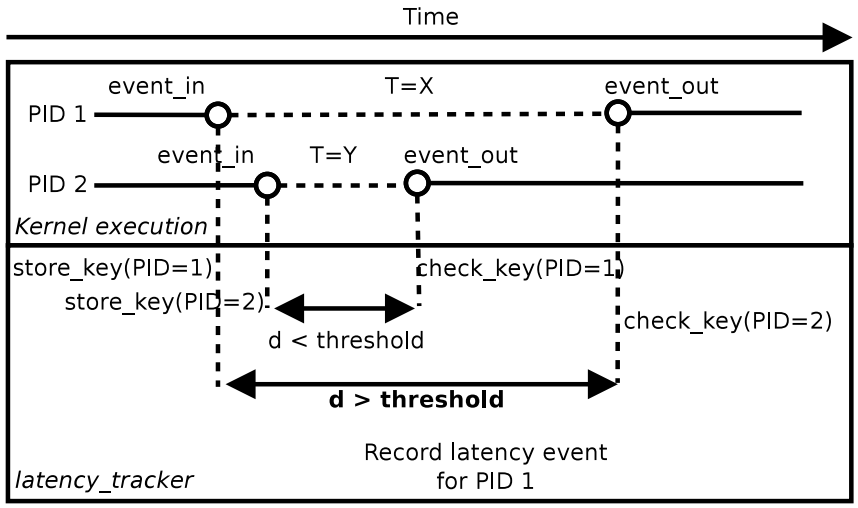

Figure 4.1 Latency tracker architecture

As shown in Figure 4.1, our approach consists of collecting data in the critical path (*Kernel execution*) and storing timing information to detect high latencies (*latency_tracker*). When

a latency is detected, we record an event in a tracing ring-buffer and trigger the extraction of this memory ring-buffer and copy to disk.

In the following sections, we first detail the collection and extraction of data and then focus on the algorithms and the design decisions that consitute the core of the tracking mechanism.

4.4.1 In-memory data collection

In order to collect the background history efficiently, we use the LTTng tracer configured in "flight-recorder" mode 4.2. However, the overall solution is not dependant on any specific tracer. LTTng since version 2.3 has the ability to record the trace only in memory, in a ring-buffer. This process completely removes the cost of the I/O operations required to extract the trace and send it to disk or to the network. When an external factor detects that an interesting condition happened (explicit user action or a coredump handler for now), we can trigger a "snapshot" which takes all the content of the ring-buffer and copies it into trace files (local or remote). Then, the user can run analyses (manual or automatic) on a trace that contains only a relatively small window of data, around the interesting event. The analyses can range from simple statistics, to more advanced critical path analysis.

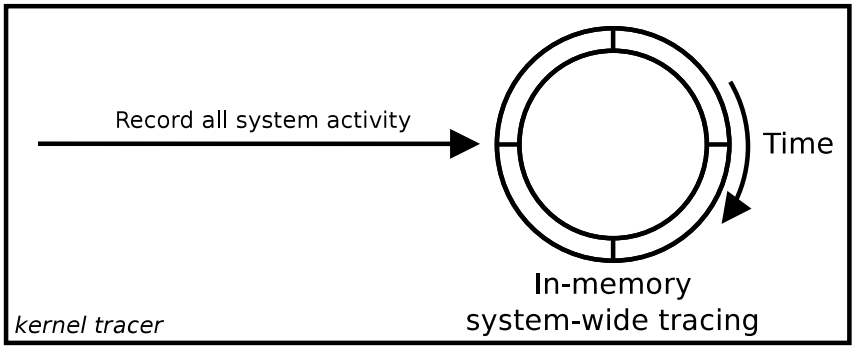

Figure 4.2 LTTng in-memory ring-buffer

The total buffer size per-channel is calculated in Equation 4.1.

$$buf\_size = num\_subbuf * subbuf\_size \tag{4.1}$$

The relationship between the number of events in a snapshot and the configuration of the tracer can be approximated by Equation 4.2.

$$avg\_nb\_events = buf\_size/avg\_event\_size \tag{4.2}$$

Therefore, the period covered by a snapshot is approximated by Equation 4.3.

$$snapshot\_period = avg\_nb\_events/event\_rate \tag{4.3}$$

For example, if we enable only the *sched_switch* on an idle 8-core server, with the default configuration of 4 sub-buffers of 256kB per-cpu, the snapshots contain around 15000 events, covering a period of 30 seconds (around 60 bytes/event). On the other hand, with the exact same configuration, but with a high scheduling load on all cores, the snapshots have the same amount of events but only cover a period of 5 seconds.

For the latency tracking analysis, we use this feature to keep a relatively short history of trace data around the interesting events. The parameters such as the sub-buffer size and enabled events are configured depending on the conditions and the problems we are trying to solve.

4.4.2 Callbacks : data extraction

The callbacks are the actions called when the delay between two events is higher than a predefined threshold, and when the timeout is reached. The callback is a function provided by the user of the module. For the case of high delay, the function is called during the exit event, so it is in the critical path of the tracked system. For that reason, the action executed here need to be as fast as possible, to avoid starting a feedback loop of latencies. The advantage of calling it from this site is that the user gains full control over the execution, and can extract accurate information about the current state of the process or the system.

For example, in the use-case we defined earlier, we want the callback to record an event in an active tracing session to mark the exact time where the high latency was detected, and then extract the trace buffers that contain this event and as much history as possible. Emitting an event in the context of the target application is fast enough (328-338 ns/event Desnoyers (2009b)) to be run directly in the callback, since it is usually multiple orders of magnitude less than the latency threshold. However, collecting the snapshot is a more intrusive operation that requires communicating between two user-space daemons over unix sockets, so we need to delegate another task to perform this operation in the background, and give back the control to the calling thread as soon as possible.

To handle this problem, we created a special *procfs* file. This file is owned by the tracking module. When a user-space process reads this file, it is put in a wake-up queue. When we detect a high latency, we wake up all the processes in the queue and make the read operation return. When the process returns from the *read* or *poll*, it knows that an important event happened and can handle it in user-space, without blocking the kernel. Since the wake-up can

happen in the context of the scheduler (for example in the wakeup-latency analysis), we have to create an *irq_work*, and proceed from the IRQ context, to avoid calling *sched_wakeup* during a *sched_switch* since it results in a deadlock.

The system is most likely loaded when the callback is activated. Therefore, there is a risk that the user-space program will not be scheduled soon enough to extract sufficient background information in the snapshot, before it is overwritten by the tracer. For this reason, it is important to set a real-time priority for this task if its result is critical.

Finally, to avoid feedback loops, we implemented a simple rate limiter, set to one snapshot per second.

4.4.3 Latency tracker

Once we have a way to collect efficiently a short history around an interesting event, we need to automatically detect the interesting conditions and trigger the recording of the snapshot. Performing the state tracking efficiently with a scalable algorithm and low intrusiveness is the most important original contribution of our work.

Design principles

The objective is to trigger an action whenever a high latency is detected inside the kernel. The action is provided by the user and can be, for example, to extract the tracing buffers, generate alerts, compute advanced statistics, record a stack trace, etc. The cost of this additional tracking must be bounded and low enough to run in production.

In order to do that, we developed a kernel module (lat) that exposes new functions to the kernel. These functions are designed to be called from anywhere in the kernel : hardcoded in the source code, or from a tracepoint handler, a *kprobe* callback, a $netfilter$ hook, etc. Moreover, these functions can be called from any context including system calls, timers and interrupt handlers.

The module uses a common *key* to track the time difference between two punctual events (the entry and the exit) and take actions depending on the delay between the two events. The key depends on the context and can be in any form (string, structure, integer, etc) as long as it is the same in the two events.

In Algorithm 2, we present the basic usage of the tracker. In this example, we use the static instrumentation already existing in the Linux kernel and connect two probes. These probes only extract the parameters *sector* and *dev* and use these as a key to track the request. The

latency tracker module then handles storing and matching the key. If the delay between the two events is higher than *threshold*, then the callback function *cb* is executed.

Algorithm 2 Block latency tracker example

function INIT
  $tracker \leftarrow latency\_tracker\_create()$
  $tracepoint\_probe\_reg(\text{"}block\_rq\_issue\text{"})$
  $tracepoint\_probe\_reg(\text{"}block\_rq\_complete\text{"})$
function PROBE_RQ_ISSUE(e)
  $key.d \leftarrow e.dev$
  $key.s \leftarrow e.sector$
  $t \leftarrow theshold$
  $latency\_tracker\_event\_in(tracker, key, t, cb)$
function PROBE_RQ_COMPLETE(e)
  $key.d \leftarrow e.dev$
  $key.s \leftarrow e.sector$
  $latency\_tracker\_event\_out(tracker, key)$

In addition, the module provides a *timeout* parameter in order to call the callback function if the exit event has not been called before a user-defined expiry time. This function allows to perform off-CPU profiling and some focused sampling which is particularly useful to detect high latencies while they are happening (before the exit event arrives). It is for example a good place to take a stack trace of the blocked process and sample external counters. The callback function receives a parameter to let it know if it was called for a timeout or a normal high latency. Therefore, the user has the control regarding the operation to take, depending on the case. Also, the key is not deleted until the exit event happens, so the user callback gets the control two times in total when a timeout has been reached.

Moreover, at any point during the lifetime of a pending event, it is possible to query the latency_tracker to see if a key is currently active. This opens up the possibility of creating *stateful tracing events*. For example, a probe can be hooked on any kernel function with a *kprobe*, when the callback is called, the probe can check if the current process has a pending latency event and extract additional information specifically related to this condition, such as the parameters passed to the function. Before this active state tracking, we had to extract the data every time the callback was hit, and process the result after the fact, which resulted in additional noise, overhead and processing time.

Memory allocation

To allow the calls to work in all of these different states, we need to make sure our module does not trigger any page fault, which could lead to deadlocks in certain situations. In order to do that, we allocate the memory required to store the keys in the hash table when the module is loaded, and ensure that it is ready to be used. The user evaluates the maximum number of keys that should be used concurrently and all the memory is allocated by the module before starting the work. The free memory is organized in a simply linked list.

In some production cases, the user does not know how many concurrent keys can co-exist and we don't want to miss interesting events because of misconfiguration. To solve this problem, if the impact is tolerable, the free list can be dynamically resized, outside of the critical path. In order to do so, we set a special flag when allocating the element stored in the middle of the list. When we start using this element, we set a flag in the tracker. Periodically, a timer handler (also used for garbage collection) checks if this flag is set and, if it is, starts a *workqueue* to resize the list (up to a maximum size defined by the user). This process is detailed in Algorithm 3.

We have to use a timer to start the *workqueue* process, even though the *workqueue* is itself an independant task. Indeed, when queueing new work, the Linux kernel informs the scheduler that the task needs to run using a *sched_wakeup*, which is a function we are interested in when tracking scheduling wakeup latencies. Even though the code is reentrant, we want to limit the impact on the server, in the critical path of waking up a task.

Since our workload can be executed in parallel, we need to protect the access to the free list. We experimented with two linked-list implementations built-in the Linux kernel and three locking strategies (along with the various hash tables algorithms detailed in Section 4.4.3).

We first experimented with a basic linked-list protected by an IRQ-safe *spinlock*, this experiment helped us create the first prototype of the module and identify the problematic concurrency situations. We then looked at the strategy used by SystemTap to protect their associative arrays and ported this mechanism to protect the free list and the hash table. SystemTap uses a *rwlock* and loops on a $write_trylock$ every 10 micro-seconds, until it obtains the lock. We noticed a performance similar to *spinlock*, but the *rwlock* creates less pressure on the CPU than the *spinlock*. This is especially interesting in case of *hyperthreading* cores since they share some resources. The down-side is the risk of starvation.

Since the free list is a significant contention point, we experimented with the lock-free linked-list of the Linux kernel (protected by RCU) and improved significantly the overall performance (all results in Section 4.6).

Algorithm 3 Freelist allocation and resize

```
function FREELIST_INIT(tracker, size)
    list ← tracker.list
    for i = 0; i < size; i++ do
        e ← alloc(latency_tracker_event)
        if i == size/2 then
            set_resize_flag(e)
        list_add(list, e)
    tracker.size ← size
function FREELIST_GET_NEW(tracker)
    if list_empty(tracker.list) then
        return NULL
    e ← list_first(tracker.list)
    if has_resize_flag(e) then
        tracker.need_resize ← True
    return e
function TRACKER_TIMER_HANDLER(tracker)
    if tracker.need_resize then
        queue_work(tracker.resize_work)
function TRACKER_RESIZE_WORK(tracker)
    size ← tracker.size
    size ← min(size * 2, tracker.max_size)
    freelist_init(tracker, size)
```

Hash table

We are looking to compare *keys* of arbitrary data type at entry and exit sites, to track the state of any operation that can be expressed as a request. Hence, the core of the tracking mechanism is a generic hash table inside the kernel. To provide maximum performance to users, the *hashing* and *matching* functions can be customized, to perform best depending on the type of key used. Because of the possibility of preemption, thread migration, interrupt handling, etc, the table is shared among the processors. Our intent is to find the best locking strategy to limit the overhead added along the critical path. One important aspect of this workload is that the hash table usage is typically symmetric for each key : one insertion, and one lookup immediately followed by a removal. However, this process can happen in parallel with thousands of active keys, so we cannot just use a list of active keys. The lookups need to happen in $\mathcal{O}(\log n)$ or better. In addition, the user must be able to handle the case of duplicated entries.

The Linux kernel already provides multiple choices for locking strategies and hash table implementations. Since our use-case is the worst-case for a hash table, and can be called from any context, we studied closely the algorithms of every option available, both in terms of speed and scalability.

The default hash table in the Linux kernel requires a lock (IRQ-safe in our case) for each operation. We identified that a stategy based on *write_trylock* (from *rwlock*) is more efficient for this use-case than a *spinlock* on hyperthreaded CPU cores. However, the impact is still high for parallel workloads (details in Section 4.6).

Since 3.17, the relativistic hashtable (*rhashtable* Triplett et al. (2011)) is included in the mainline Linux kernel. This hashtable has the advantage of being resizable, which is an important feature for the future of our work, but also lock-free for the read-side. Thanks to *RCU*, we only need to lock the table when inserting new elements, which is half of our workload. Moreover, according to the benchmarks provided by the authors, the performance seems better than the default hash table. However, after experimentation, we identified a high overhead starting at 16 concurrent processors. We did not use the resize capability of this table, because the resize operation is triggered in a context where the associated memory allocation can be problematic for our operating conditions.

The last hash table investigated comes from the userspace-RCU library Desnoyers et al. (2012), that we ported to the kernel. This hashtable is completely lock-free and helped us achieve our best scalability results. This hash table can also be resized, and the resize operation can be performed in a separate execution context, another advantage favoring this

structure.

Locking

Here is a summary of the various linked-list, hash tables and locking algorihms that we compared for this study, to identify the best combination for our use case. The Id is used as a legend for the performance graph in Section 4.6.

TABLE 4.1 Locking and data structure tested

Id	Free list	Hash table	Locking
A	default list	default HT	spinlock
B	default list	default HT	try lock
C	lockless list	default HT	spinlock
D	lockless list	default HT	try lock
E	default list	rhashtable	spinlock
F	default list	rhashtable	try lock
G	lockless list	rhashtable	spinlock
H	lockless list	rhashtable	try lock
I	default list	urcuht	spinlock
J	default list	urcuht	try lock
K	lockless list	urcuht	N/A

4.5 Use-cases implemented

4.5.1 Off-CPU profiling

Based on this architecture, we created a kernel module to track the time a process spends not running, and record its kernel stack when it returns on a CPU. That way, we know at runtime all the processes that are blocked and we can then confirm if it is a normal delay or not. It helps identify locks imbalance, resources shortage and other concurrency issues. In addition to the stack of the off-cpu process, we also extract the stack of the process doing the wake-up (only if the target process is interesting). That way, we also know why the wake-up occurred.

Here is the result with a threshold set at 5 seconds. In this output, we see that the *swapper* process on CPU 3 received a timer interrupt and woke up the $qemu - system - x86$ thread (that was waiting on $futex$ for 10 seconds).

```
offcpu_sched_wakeup:
```

```
  waker_comm=swapper/3 (0),
  wakee_comm=qemu-system-x86 (7726),
  wakee_offcpu_delay=10000018451,
  waker_stack=
    ttwu_do_wakeup+0xb2/0xc0
    ttwu_do_activate.constprop.74+0x5d/0x70
    try_to_wake_up+0x1d2/0x2c0
    wake_up_process+0x23/0x40
    hrtimer_wakeup+0x22/0x40
    __run_hrtimer+0x77/0x1d0
    hrtimer_interrupt+0xef/0x230
    local_apic_timer_interrupt+0x37/0x60
    smp_apic_timer_interrupt+0x45/0x60
    apic_timer_interrupt+0x6d/0x80

offcpu_sched_switch:
  comm=qemu-system-x86,
  pid=7726,
  delay=10000140896,
  stack=
    schedule+0x29/0x70
    futex_wait_queue_me+0xdd/0x140
    futex_wait+0x182/0x290
    do_futex+0xde/0x760
    SyS_futex+0x71/0x150
    system_call_fastpath+0x1a/0x1f
```

For this example, we recorded the activity with the tracker for 16 seconds on an idle desktop. During this period, we got 133 off-cpu notifications while *ftrace* generated 51405 *sched_switch* and 26158 *sched_wakeup* events (12MB of text). We thus ended up with much more relevant data and a lot less to parse.

4.5.2 Scheduler wake up latency

For real-time systems, it is important to have a boundary on the delay between when the scheduler decides a process should be running and when it actually runs. We used our new architecture to compute the time difference between a *sched_wakeup* and a *sched_switch* on the same PID. We used the callback to record a LTTng snapshot when the delay was above a threshold of 5 milliseconds. We let this analysis run for 24 hours on an active web and mail server and identified interesting latency patterns that would have been really hard to understand without this new method. Thanks to the small trace generated when a latency

occurs, these new results led us to devise new analyses tools that will focus on explaining these latencies, as part of future work.

4.5.3 System calls latency

Another useful module implemented with this new tracking mechanism is a system call latency tracker. We created a module that hooks on system call entries and exits and computes the time difference between the two events. If the delay is above a threshold, we generate an event and record a LTTng snapshot. In addition, we used the feature to check the state of an event. Thus, at each scheduling change (*sched_switch*), we check if the next process to be scheduled has been blocked in a system call for more than the threshold. If true, we extract its kernel stack. This allows us to see exactly where a system call is blocked inside the kernel, every time this process gets some CPU time.

4.6 Measurements

In order to evaluate the performance of our approach, and decide if it is suitable for usage in production, we measured the impact of the latency tracker in terms of CPU usage with the scheduling latency module. Thereafter, we measured the overhead in terms of disk I/O with the block latency module, and finally, we evaluated the overall impact of the tracker, and various configurations of LTTng, on a stress test of MySQL which combines all the resources.

4.6.1 CPU overhead

In this experiment, we evaluate the impact of the hash table, and the linked list implementations, and of locking on a highly-concurrent workload, while increasing the number of CPUs available. We use the module explained in Section 4.5.2. The test machine is a quad-socket AMD Opteron(TM) Processor 6272. The kernel sees 64 CPUs, but the AMD Bulldozer architecture is a partial Simultaneous Multithreading (SMT) architecture. The FPU and L2 cache are shared between pairs of core, but the integer cores are independent. Before running the 64-cores tests, we evaluated that the overhead of running our workload on shared cores is between 10% and 15% more than running on independant CPU cores. Up to 32 simultaneous CPUs, we run on independant cores to limit this side-effect. For the tests at 48 and 64 cores, we have to use shared cores.

The test consists of running *hackbench* 50 times and compute the statistics about how long it takes to complete. *Hackbench* spawns 10 groups of processes and tries to exchange 1 byte, back and forth, 10000 times between the senders and receivers, over 40 different sockets. It

thus consists of 400 tasks trying to run simultaneously, exercising the scheduler under a high concurrency. When increasing the number of CPUs available between 1 and 64, we see a linear speedup when running without instrumentation.

From all the combinations in Table 4.1, we confirmed that the lockless linked-list has a major impact in terms of scalability and that the *try_lock* mechanism is more interesting when working with shared CPU cores. To limit the number of graphs, we only present here the best results for each hashtable implementation. We also included the test with a *SystemTap* script doing the same operation as our program. The graph 4.3 shows the results with varying CPU configurations : 1 to 32 non-shared CPU cores, and 2 to 64 shared (SMT) CPU cores.

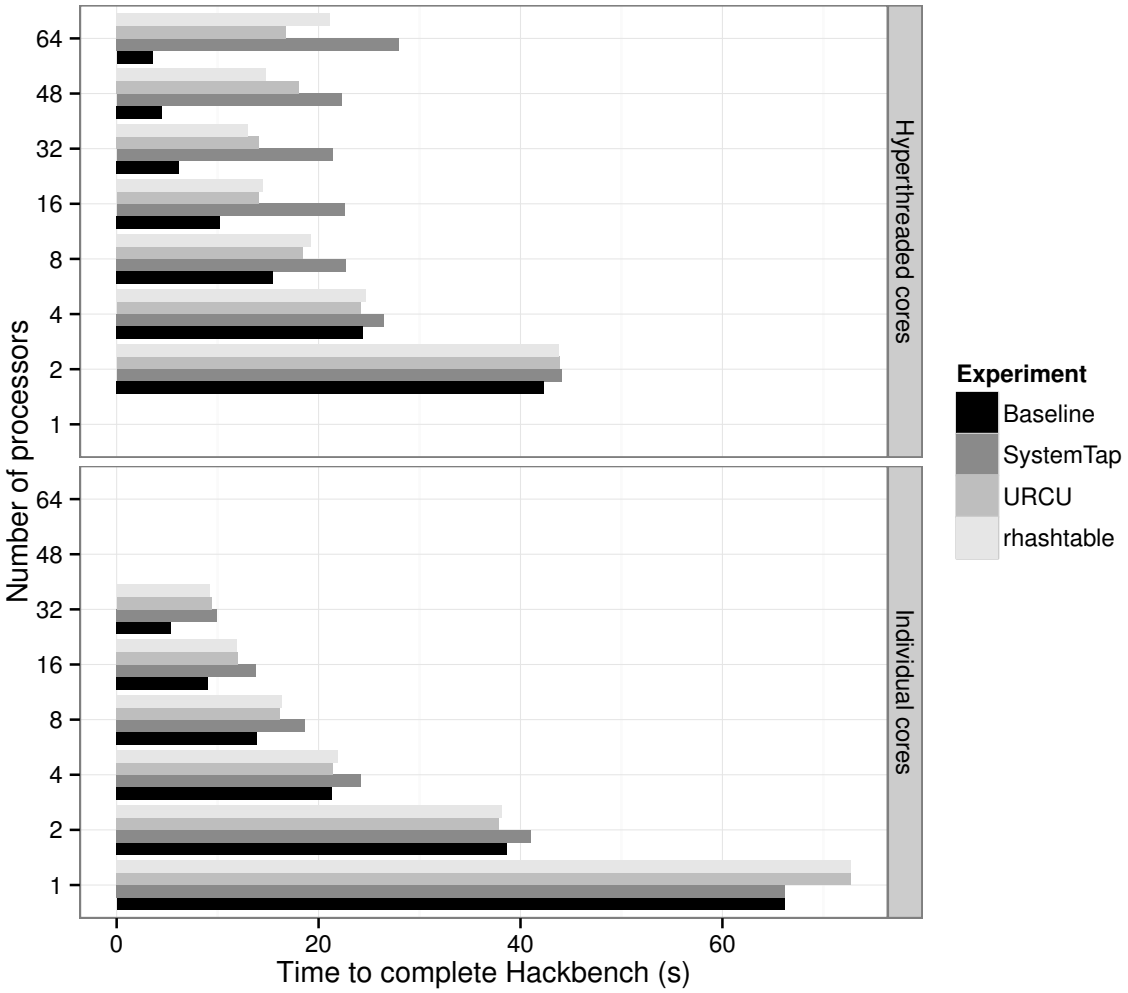

Figure 4.3 Overhead on hackbench

As we can see, the sharing of CPU ressources has an impact on the scalability of all the configurations. The RCU-based approaches have a clear advantage over the rwlock approach,

especially on shared cores. With only one core though, when there are no concurrency issues, we see that the lock prefix on instructions used by RCU has more impact. To limit this unncesseray overhead, we would need to detect the current number of CPUs and adapt the locking strategy accordingly.

4.6.2 I/O overhead

We tried to quantify the overhead imposed when tracking the block-related events, as explained in Algorithm 2, by running the *fileio* test of *sysbench* in various combinations (sequential read, sequential write, random read and random write) with 2GB of data split in 16MB files on a SSD (Samsung 850 Pro) and on a rotating drive (Western Digital Red). The only measurement above 1% overhead is the sequential read on SSD : 4%. It remains nonetheless within the standard deviation. Therefore, we did not find any significant overhead on the I/O.

4.6.3 MySQL stress test

This test is based on *sysbencholtp*, which simulates heavy access to a *MySQL* database. This test is interesting because it combines CPU, memory and I/O operations, and thus presents a good approximation of a real-world heavy workload. For this test, we measured the overhead imposed by various configurations of LTTng in flight-recorder mode (all events enabled, only *sched_switch* and *sched_wakeup*, only system calls), and the latency tracker with the best performing algorithm determined from the CPU overhead test. We also included a test with *latencytop* enabled from the Linux kernel.

Table 4.2 MySQL stress test overhead

Test	**Average**	**Overhead**
Baseline	63.26s	
LTTng sched	63.65s	0.61%
LTTng syscalls	64.95s	2.66%
Tracker	65.36s	3.31%
Latencytop	66.24s	4.70%
LTTng all	70.24s	11%

All the results are presented in Table 4.2. The standard deviation for all these tests was between 0.25 and 0.30 seconds.

4.7 Conclusion and Future Work

In this study, we created and tested a new approach to measure the latency of arbitrary events and react when a high latency is detected. This new method is scalable, and the overhead, as tested on micro-benchmarks, is constant and low-impact when running on mixed workloads.

The result of this study is an implementation of a "latency tracker" that can already be used on real servers. It detects concurrency and latency issues, previously hard to detect, without noticeable impact. In addition, the "callbacks" recording mechanism implemented, combined with the efficiency of the flight-recorder mode of LTTng, allows to gather relevant background information. This is used to understand offline the source of the latency at the time it happened.

This new approach for software tracing led us to start working on an *efficient stateful tracing* architecture. In contrast with the traditional tracing paradigm, where every event is recorded and then analyzed offline, in this new architecture, the context in which the events are triggered is what determines the action to perform when the event arrives. This approach, previously reserved for dynamic tracers such as *SystemTap* and *dtrace*, but limited by their scalability issues, can now be used efficiently and only relies on a unique identifier to match start and end events.

The result is a generic and lightweight monitoring solution producing accurate analyses for any operation that can be expressed as a request. The data produced by these analyses is concise and easier to process than full system traces, while still providing enough information to understand the problems.

In terms of future work, the early results collected with the sample analyses developed during this phase of the project encouraged us to test these analyses on large-scale infrastructures (such as a large number of instances in the Amazon EC2 Cloud) and extract common characteristics between latency events. This work is especially interesting for detecting and understanding latency outliers and we expect to find rare or unknown conditions with this new architecture.

4.8 Acknowledgments

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Naser Ezzati for reviewing this manuscript.

CHAPITRE 5 ARTICLE 3 : REAL-TIME LINUX KERNEL RESPONSE TIME MEASUREMENT

Authors

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

Mathieu Desnoyers
ÉfficiOS Inc.
mathieu.desnoyers@efficios.com

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

Submitted to

Real-Time Systems

Keywords

Response time, Real-Time, Latency, Tracing, Concurrency

5.1 Abstract

We propose a new class of online analysis to measure the response time of Linux-based systems and react if necessary. The target is every system that requires a bounded response time, including real-time systems, servers and desktop environments. The response time in such systems is either the delay between an interrupt and the start of processing, or the delay between an interrupt and the end of processing. We address the problem of following the complete control flow at run-time, starting at the interrupt, to measure the total response time. Our algorithm is designed to run as a monitoring service or for debugging purposes. It does not require any modification to the kernel, and is integrated with kernel tracers

to provide the same level of detail, without needing to store and post-process large traces continuously.

5.2 Introduction

Real-time operating systems and applications are designed to minimize and control the response time. Hard real-time systems are designed and tailored exactly for the target application to have a bounded response time. On the other hand, general purpose operating systems, such as Linux PREEMPT_RT, have to provide a good balance between usability and responsiveness to events, as they can be used for a variety of use-cases. When deploying a real-time application based on Linux, users expect a complete operating sytem with a bounded reactiveness.

There are two main kinds of real-time workloads : timer-driven and interrupt-driven. The only difference between the two types of workloads is the wake-up source : in timer-driven configurations, we know exactly when the next wake-up is supposed to happen, whereas in interrupt-driven configurations, it can happen at any time. An example of a timer-driven workload is motion control systems where we periodically have to check the current position and adjust if necessary. An example of an interrupt-driven workload is an input device like a button or a sensor.

When working with a full-fledged system such as Linux, since the core of the system is not tailored specifically for the desired use-case, there are many possible perturbations sources, and settings to adjust. The mainline Linux kernel supports various levels of priorities. However, the main risk with this kernel is that preemption can be disabled for various reasons, which can cause arbitrary scheduling delays even for high priority tasks. The PREEMPT_RT kernel patch converts Linux into a fully preemptible kernel Fu and Schwebel. That way, high priority tasks have a much higher chance of being scheduled quickly when needed.

This improves considerably the responsiveness of the system. Nonetheless, since this kernel tries to be as generic as possible, it is very complex. There are many sites where latencies can be introduced, and several options need to be configured to adjust the system to the requirements. Monitoring is essential to validate that deadlines are satisfied, and otherwise to understand why they were missed.

The responsiveness of a system is defined as the delay between an event and the reaction to this event. We are usually interested either in the delay for the target task to start processing the data, or in the delay to finish processing the data. In the operating system, the closest software event to a hardware event is an interrupt. A basic chain of reactions can

be summarized by : an interrupt is raised (by a device or a timer), the kernel is signaled, it calls the interrupt handler which reads the data (if any), it then raises a software interrupt (SoftIRQ) which awakens the target task (if it is waiting for input). The complete detailed chain of reactions is presented in Section 5.4.2.

To illustrate this behaviour, we built a small application that waits for a user input, and logs the data in parallel, gathers some statistics and does the actual computation. It then goes back to waiting for the next input. We launch this application through SSH, so the user input arrives from the network to SSH, and is then forwarded to the application. Figure 5.1 presents the timeline view of this application, as presented by Trace Compass tra, when it receives the data, and Figure 5.2 presents the resources usage during the same time interval. In this example, the whole period, from the interrupt arrival to the end of processing by all threads, takes 256 micro-seconds. The kernel trace, with the scheduling, interrupt and system call events for the whole system during this period, contains 576 events.

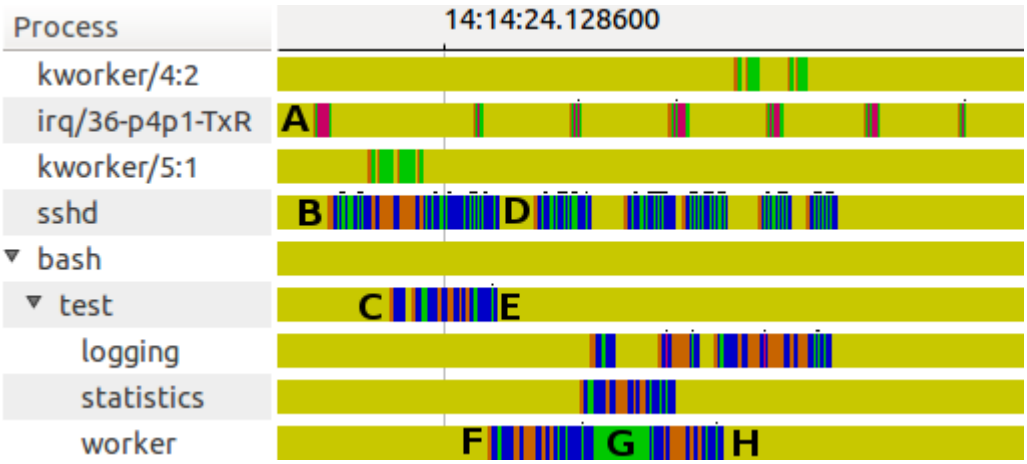

Figure 5.1 Test application timeline

In Figure 5.1, **A** corresponds to the threaded interrupt handler reading data from the device and waking up SSH. This handler has been woken up by the kernel just before the interrupt arrived. At **B** we see the awakening of `SSH` from the `select` system call, **C** is the awakening of the `test` application from the `read` system call. At **D**, `SSH` returns waiting for the next input. The activity of `SSH` and the IRQ thread after **D** is related to the output generated by the threads. At **E** the `test` application does the same. Between **C** and **E** the `test` application woke up the `worker` thread which starts running at **F**. The **G** area is where the actual computation on the data is performed. Just before that, the `worker` thread wakes up the `logging` and `statistics` threads. At **H** the worker thread returns waiting for the next input.

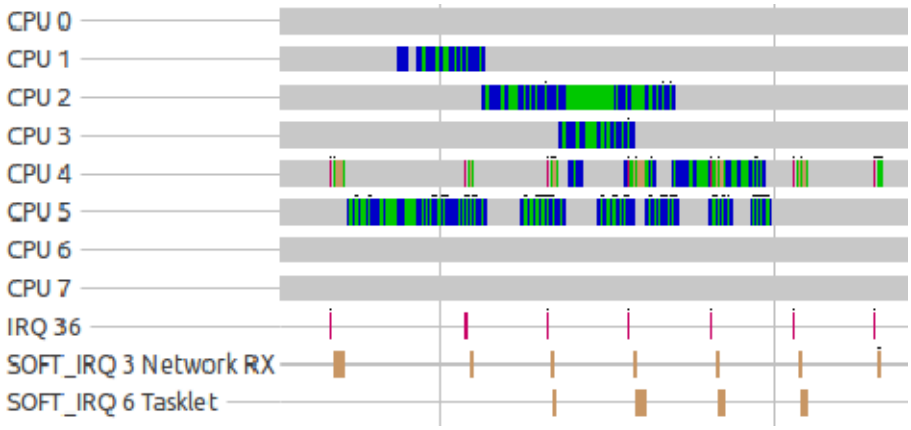

Figure 5.2 Test application resources usage

In order to monitor the response time of a system, we have to compute the delay between the interrupt entry point in the kernel (just before **A** in Figure 5.1) and the time when the interesting thread waiting for the interrupt starts to work (**F**) or when it finishes its processing (**H**). The first problem is that only a subset of the interrupts received by the kernel is relevant to the main workload. The information required to know if an interrupt is targeting a process of interest is only known later in the chain. We can filter on the interrupt number, but we also need an efficient way to track all the interrupts coming into the kernel and follow their execution flow up to the point where we can decide if it meets the desired conditions.

The second problem is that sometimes an interrupt handler Bovet and Cesati (2005) (or a process woken up by an interrupt) can wake up multiple other processes and softirqs, as in the previous example where the `worker` thread wakes up two other threads in addition to processing the data. This creates multiple branches in the execution path and we might not have all the information to select which of these branches is the relevant one. Therefore, we have to track all of the branches until we have enough information to make a choice. Moreover, the first user-space process woken up in the interrupt processing chain is not necessarily the target task. In the previous example, `SSH` is the first user-space process along this path, but its only role is to forward the data.

Finally, users might want to track the execution down to the end of data processing (**H**). In this case, we have to follow the processing even after it reached user-space, which adds more complexity since each application can handle the data differently.

The main contribution of this paper is an efficient algorithm to identify and measure at run-time the relevant chains of event across execution contexts, from an interrupt to the

end of processing in user-space. Another contribution is the research on the data structures supporting this algorithm to optimize its execution speed, limit its memory footprint and limit its impact for real-time systems (lock-free). Control options are provided to target specific processes and interrupts, to specify where to stop the tracking (entering in user-space, returning to a waiting state or finishing to handle the data in complex or asynchronous workloads), and to execute custom actions depending on the total response time.

The result of this work is a method to accurately identify when an unusually long response time happens, quickly observe the breakdown of where the time was spent in the whole processing chain, and give users a way to gather more background information to understand and fix the problem. For investigations that require more data, we also record a special event in a kernel trace to provide a clear entry point for advanced offline analyses.

5.3 Related Work

The tracing infrastructure in the Linux kernel for real-time is advanced and well maintained. `Ftrace` Edge (2009a) has two tracers dedicated to real-time problems : `irqsoff` and `preemptoff` Rostedt (2009) that can be combined and are focused on real-time tasks. The `irqsoff` tracer instruments the callsites where the interrupts are being disabled and enabled and records the highest latency between those two events. The `preemptoff` tracer is similar but focuses on the intervals during which preemption is disabled. These tracers work very well for finding the worst cases of these two kinds of problems. These analyses are complementary to the work we are doing here because they provide very detailed information about those two sources of latency, but they do not consider the delay of the whole chain, only between two specific events.

Usually, when trying to understand this kind of problem, the kernel tracers are the best solution, as they give a complete view of the system. The drawback with these systems is the time required to process and understand the trace afterwards. In Côté and Dagenais (2016), the authors provide interesting results about using models and comparison views to quickly extract problematic executions. They also provide a pattern discovery mechanism to assist users in creating an application model. This work is very useful but is meant to be run offline, after a detailed trace has been collected for the full execution duration.

Also based on offline processing, but particularly interesting, is the research on the analysis of the critical path leading to a specific point in the trace, presented in Giraldeau and Dagenais (2015). This work is useful to explain why a specific long latency occurred, but it needs an entry point in the trace ; it cannot be run on every state of every real-time task. We use this

algorithm to provide a detailed view of what happened when we know that a long latency occurred, but we cannot use this method to actually detect a long latency. Our method can provide the entry point required by the critical path analysis to fully explain a problem.

Another approach to detect real-time problems is to run a periodic task and measure its behaviour over a long period of time. The `rt-tests` tool suite provides benchmarking tools such as `cyclictest` Gleixner to evaluate the real-time capabilities of a system or to detect abnormal latencies. Cyclictest works by initializing a timer set to expire at a known timestamp. When the timer expires, the kernel schedules back `cyclictest` which checks the current timestamp. It then computes the time difference between the current time and the expected wake-up time. That way, it knows how long it took between the moment the timer interrupt fired and the moment the target process started to run. With this tool, we know the general profile of a system. In addition, it is possible to set a threshold and integrate it with `ftrace` to see a trace of when a high latency occurred. This tool is commonly used to benchmark Linux real-time systems. However, it is not designed to work as a monitoring tool and cannot work with aperiodical interrupts ; it needs to know the time at which the interrupt occurred.

Another research related to the timer interrupts and scheduling delays is presented in Kwon et al. (2010) and Kwon (2010). The authors use LTTng Desnoyers and Dagenais (2006) to collect traces related to `hrtimers` from the Linux kernel and parse the trace to extract high latency events between the expected expiration of a timer and the actual return to user-space, while generating a high load on the system. This work seems limited to parsing text-formatted offline traces on single-core systems.

To accurately measure event response latencies, a hardware-based approach has been taken in Rybaniec and Wieczorekb (2012). In that paper, the authors use a simple microcontroller to raise interrupts and wait for the output on a GPIO pin. By measuring the delay between sending the signal and the answer, under various conditions, they compare three approaches to handle the interrupts on Linux. This method is interesting for this kind of comparison because the exact measurement is not critical here, only the difference between the scenarios. Thus, the imprecision of the microcontroller clock is shared between all tests and can then be factored out.

A concrete example of how the research we are conducting is useful can be found in Yang et al. (2012b). In this paper the authors argue that synchronous completion of block I/O deliver higher performance than interrupt-driven I/O, especially for ultra-low latency devices. This paper provides a thorough study of the impact of context switching in terms of hardware resources (cache, TLB, power management). This study is interesting for our topic because

their measurement technique consists of modifying the kernel and user-space code to read the cycle counter of the CPU for all important steps, and processing the results offline. A lot of manual steps were required for this processing. This motivates the need for an automated way to do this type of analysis, so other research can rely on it to compare approaches and evaluate possible optimizations. It also highlights the fact that we need to make sure that our measuring method has a well defined and understood resources usage, so that it can be trusted for such low-level work.

The `latency-tracker` project Desfossez et al. (2016) provides a framework to track at run-time matching entry and exit events, and start an action if the delay between the two events is higher than a threshold. This project is used for example to track system calls and block I/O operations latencies. However, until now, it was limited to pairs of events, and could not follow multi-level chains of events. It was built from the start with a focus on efficiency and low intrusiveness. Thus, we chose to use this project as a base for our research.

5.4 Following Interrupts In The Kernel

5.4.1 Interrupts Processing Call Chains

When an hardware interrupt is raised, the processor notifies the kernel. When the interrupts are unmasked, this notification causes the `do_IRQ` kernel function to be executed, regardless of the code currently executing. Then, the kernel calls the appropriate interrupt handler which usually ends up raising a software interrupt or waking up a process to handle the data. The purpose of this multi-stage dispatch is to limit the amount of time spent in interrupt context, which interrupts the normal flow of execution and prevents other interrupts from coming in. The exact interrupt-handling chain differs between the mainline and `PREEMPT_RT` kernels. The following diagrams present the two callchains based on the trace points available in the kernel :

As we can see, the main difference between the two implementations is that the `PREEMPT_RT` kernel handles the interrupts in a thread, each interrupt number has its own threaded interrupt handler, which allows the user to set priorities for the interrupt handlers. On the other hand, in the mainline kernel, the interrupts usually raise software interrupts, processed when the interrupt handler returns.

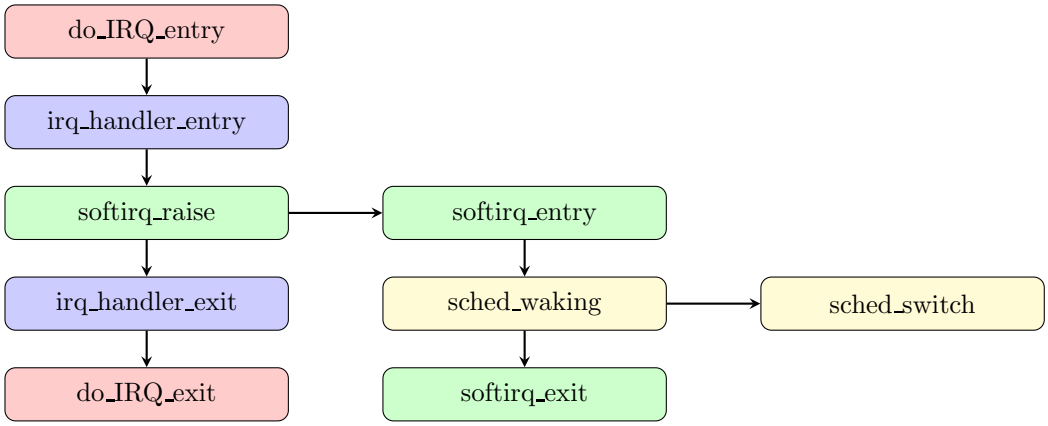

Figure 5.3 Interrupts flow in the mainline kernel

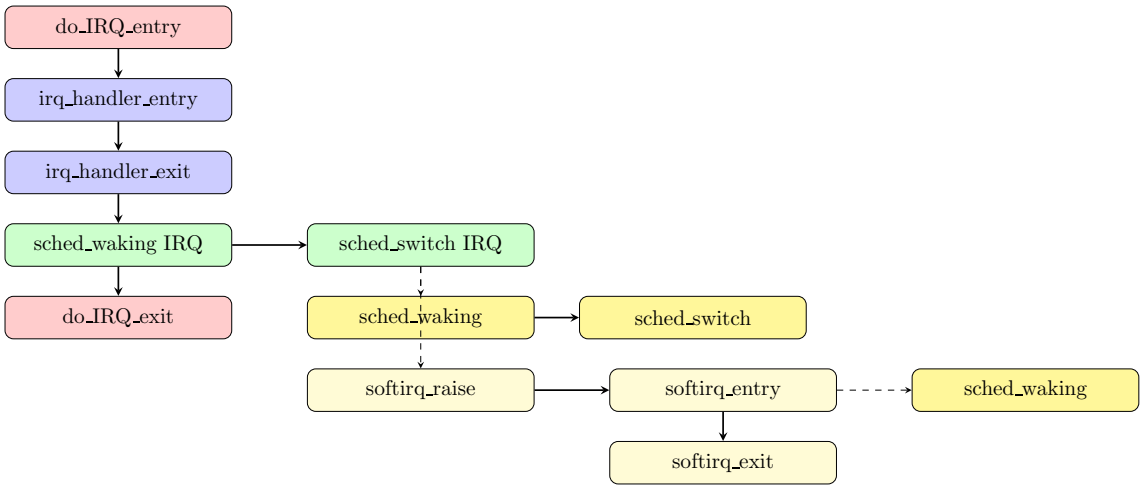

Figure 5.4 Interrupts flow in the PREEMPT_RT kernel

5.4.2 Interrupts Tracking

To follow the interrupts processing chain, we chose to create probes to hook on already available trace points, so we do not require any modification to the kernel code. Only the `do_IRQ` function is not yet instrumented with trace points, so we use `kretprobes` Rostedt (2005) to hook on the entry and exit of this function. After hooking up the probes on the relevant events, we need to lookup the current context to know whether a specific event is relevant for the current configuration. In order to do that, we created the transition table for the two kernel variants and extracted the common criterion that could be matched between each state change. Table 5.1 presents the lookup, insertion and the eventual deletion from the hash table that need to be performed for each of the tracepoint events that can be related to an interrupt. The "work_begin" and "work_end" events are additionnal sources of

information that are detailed in section 5.5.1.

Table 5.1 Linux interrupts handling transition table

Event	**Lookup key**	**Insert key**	**Clear**
do_IRQ_entry	None	"do_IRQ", CPU #	N/A
irq_handler_entry	"do_IRQ", CPU #	"HARDIRQ", CPU #	No
softirq_raise	"HARDIRQ", CPU #	"RAISE", CPU #, softirq #	No
irq_handler_exit	"HARDIRQ", CPU #	None	Yes
do_IRQ_exit	"do_IRQ", CPU #	None	Yes
softirq_entry	"RAISE", CPU #, softirq #	"SOFTIRQ", CPU #	Yes
sched_waking	"HARDIRQ", CPU #	"WAKING", PID #	No
sched_waking	"SOFTIRQ", CPU #	"WAKING", PID #	No
sched_waking	"SWITCH", CPU #	"WAKING", PID #	No
softirq_exit	"ENTRY", CPU #	None	Yes
sched_switch_in	"WAKING"	"SWITCH", PID #	Yes
sched_switch_out	"SWITCH", PID #	None	No
sched_switch_out (blocked)	"SWITCH", PID #	None #	Yes
work_begin	"SWITCH", PID #	"WORK", user cookie	No
work_end	"WORK", user cookie	None	Yes

These tables are required because we are working with an event-based mechanism on multi-processor systems. Therefore, all of these events can happen in any order and may look semi-random when observing them on a timeline. With this state-tracking methodology, we convert asynchronous events to a meaningful parallel state.

5.5 Critical Trees

When using the transition tables defined in the previous section, we can effectively follow an interrupt from the moment the kernel notices it until the scheduling of the task waiting for it. The path for this operation is often a single chain. With this method, however, we can go farther than the scheduling of the target task. With some simple heuristics, we can follow the process up to a point where it finishes handling the data related to the interrupt. For example, consider a thread that waits on the `poll` system call, for inputs on a file descriptor triggered by a key press on the keyboard. When a key is pressed, an interrupt is raised, the kernel handles it and wakes up the waiting task. This process then reads the data, does some processing and goes back waiting on `poll`. If we look at the scheduling events during the processing of the data by the task, we see that it tries to run as long as the work is not completed ; it can be preempted, but stays in the run queue. When it goes back waiting on `poll`, it goes out of the run queue (the `prev_state` field in the `sched_switch` trace point is not equal to 0). With this information, we can follow the whole operation, which gives us additional metrics and can help really understand where the time is spent.

The problem is that there might be multiple tasks woken up following a single input, for example a statistics threads or a logging thread. Also, the target task might just be a dispatch thread that waits for input and forwards it to the next available worker thread. In this case, we observe a new `sched_waking` event ; it means that we have to wait for the corresponding `sched_switch`, while still tracking the state of the waker task to see if it wakes up other tasks. Since we are only following the links between threads, it can become difficult to identify the relevant chain of events.

From this example, we understand that each `sched_waking` (and `softirq_raise`) causes a new branch in the critical path that we have to follow until we know which one is the important one for our use-case. This transforms our notion of *critical path* into a *critical tree*.

Following only the scheduling events works for relatively simple chains of events. For more complex chains (asynchronous or busy wait in user-space), the scheduling events might not be enough to get a clear picture of the program behaviour. For these complex cases, we created user-space notifiers so the threads can notify the tracker when the process starts and finishes its work. This solves the problem for the more advanced use-cases. Future work will automate increasingly advanced cases.

5.5.1 Efficiently Following All The Chains

Memory Allocator

The first step to be able to track all of these concurrent events efficiently is to use appropriate data structures. Building up on the existing `latency-tracker` infrastructure Desfossez et al. (2016), especially the lockless Read-Copy-Update (RCU) hashtable and pre-allocated memory in a lockless RCU free-list, we added the support for a NUMA-aware memory allocator. The `latency-tracker` pre-allocates all the memory required to keep track of the events when it is created, to avoid allocating memory in the critical path of the applications.

Previously, when a new event had to be tracked in memory, the allocator would try to use an event from its local CPU pool and, if it was empty, it would request 16 events from the shared pool. When releasing the event, it would add it to local CPU pool and, if the pool was too big, it would transfer 16 events back to the shared pool.

This approach had the advantage of limiting the rate of failed "compare-and-exchange" atomic instructions (`cmpxchg`) and false-sharing. However, we noticed that it was not efficient in terms of memory usage when working with highly unbalanced workloads on multi-processor systems. Indeed, if a processor had a lot of activity, it would often empty the local and the

shared pools and fail to allocate new events, while other processors still had a lot of unused events available. Moreover, for some workloads, we noticed a high level of NUMA node miss, which added overhead. We compared with `perf bench numa` the local and remote access to memory. The results are presented in Table 5.2, they clearly show a speedup of 106%.

TABLE 5.2 NUMA access timings

	Local	**Remote**
Latency (nsecs/byte)	0.133	0.258
Bandwidth (GB/sec)	7.525	3.878

The memory management algorithm is presented in Algorithm 4.

```
function INIT_TRACKER_FREELIST(max_events)
    nr ← NR_NUMA_NODES
    for all numa_node do
        POOLS[node] ← ALLOC_POOL(max_events/nr)
    for all online_cpu do
        local_pool ← POOLS[CPU_TO_NODE(cpu)]
function GET_EVENT
    if not EMPTY(local_pool) then
        return GET_FROM_LOCAL_POOL
    return GET_FROM_NODE_POOL
function PUT_EVENT(event)
    if EVENT_FROM_REMOTE_POOL(event) then
        return ADD_TO_REMOTE_POOL(event)
    if SIZE(local_pool) < LOCAL_CACHE then
        return ADD_TO_LOCAL_POOL(event)
    return ADD_TO_NODE_POOL(event)
```

Algorithm 4 NUMA-aware memory allocator

Since our keys are 1200 bytes long, and we have to manage multiple thousands of active keys in memory during heavy workloads, this optimization was noticeable immediately when implemented. To test it, we ran a `ping flood` (on the test machine described in Section 5.7.1) pinned sequentially on each of our processors that sent 5000 packets and we compared the two memory allocators. In terms of NUMA node miss, we decreased from 150,387 to 61,018, an improvement of 60%.

In terms of memory usage, we compared the memory allocation required to ensure that we did not miss any events in a I/O stress test. The test takes 2.1 seconds to complete with both

algorithm and we track 210,000 events during that period. With the previous approach, we had to allocate 1,000,000 events to have enough memory. With the new approach, we only need 300,000, saving 840MB with this new memory management algorithm.

Keeping The State Across Branches Of The Same Tree

Our goal is to keep track of the delay between the time when an interrupt is noticed by the kernel until a specific "end condition". We have defined three main end conditions :

1. when the target task gets scheduled in (measuring the time spent in the kernel before processing the data) ;
2. when the target task goes back to passively waiting for the next interrupt (measuring the time spent in the kernel and the processing of the data for simple processing loops) ;
3. when a related task informs the kernel that is has finished handling the data (measuring the time spent in the kernel and processing the data for complex/asynchronous chains of events).

We define the "target task" as the first user-space task woken up in the interrupt processing chain. The first case is usually only a chain of events (no branches), so it does need any special handling for the memory management. The second case, implies that the kernel has time to put the target task to sleep between events. It is interesting because it does not require any collaboration from the user-space task. However, we do not put too much emphasis on this scenario because it is highly dependant on the workload. Furthermore, it could create branches that never finish. For the third case, we require that a task in the interrupt processing chain informs the kernel when it has finished processing the data (usually the period end as defined in a real-time process). This case is the one described here because it provides the most accurate results.

The output of this tracking is the timestamp of the arrival of the interrupt in the kernel, and the delay until the end condition was detected. Optionally, we can provide the breakdown of what happened, but this work is more suited to be done offline by algorithms such as the ones in Giraldeau and Dagenais (2015). The main focus here is to identify abnormally long chains and extract entry points for post-processing the traces offline.

From Section 5.5, we know that we have to follow all wake up events as separate branches of the same tree, until we identify which branch is the relevant one. Since we are working with an event-based tracker that only matches pairs of event (entry and exit) and each "wake-up" and "raise" events create new branches, we have to duplicate some information at each

step. Additionally, we are interested in the end in only one branch, so when it has been identified, we need a way to remove the other branches. The Figure 5.5 presents an example of how a single interrupt source can create such a tree of related tasks. In this diagram, the interesting critical path is composed of the light gray nodes, the dark gray nodes are irrelevant to determine the delay between the interrupt and the `work done` event, but we only know that when we receive this last event.

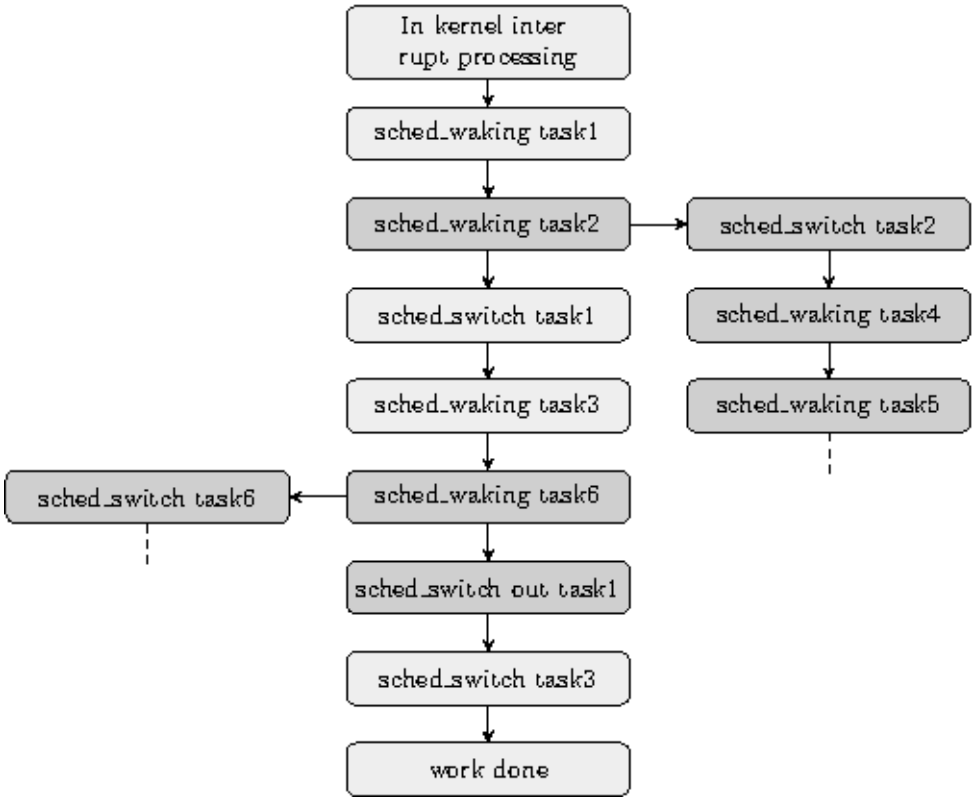

Figure 5.5 Example critical tree

As we can see from this example, the tree can become deep and wide very quickly. Since we are only interested in the total delay of the interesting branch, and we want to limit the memory consumption of this algorithm for production use, only the tip of each branch stays in memory at any time and a link to the origin node is kept at all time. All the intermediate events are used to follow the state transitions, as explained in Section 5.4.2, but the end of a branch always replaces the previous end. Thus, the memory consumption related to a particular interrupt depends on the number of branches it creates until the interesting branch is identified.

Asynchronous Workloads

Some low-latency and high throughput applications implement a listener-worker design pattern where a thread waits for an input, sends it to a worker thread or adds it to a queue, and returns waiting for the next input. This is usually implemented for multi-threaded applications. Depending on the communication mechanism to dispatch the work, we might not necessarily observe an explicit link between the listener and the worker, especially if it is based on a shared queue.

To guarantee that we can keep track of an input throughout such a chain, we introduced a `begin work` event which works in combination with the `work done` event. The match between those two events is based on a cookie, a value that can be found at the beginning and at the end of the work. After the end of the work, the value can be reused if needed for another work tracking. In Algorithm 5, we illustrate this mechanism with a `job_id` as cookie.

```
function LISTENER
    while True do
        job_id ← WAIT_INPUT
        EMIT_WORK_BEGIN(job_id)
        APPEND_WORK_QUEUE(job_id)
function WORKER
    while True do
        job_id ← DEQUEUE_WORK
        PROCESS_WORK(job_id)
        EMIT_WORK_DONE(job_id)
```

Algorithm 5 Asynchronous state tracking

This collaboration from the listener thread is the only information required to link an interrupt to any workload in user-space. This approach solves the problem for the more complex use-cases, where the processing of the data can follow multiple paths, and for asynchronous processing loops. The only drawback is that users are responsible to instrument the code and emit the two events at the appropriate sites.

5.5.2 Cleaning-up The Irrelevant Branches

When the interesting branch is identified, we have to close all the other branches to limit memory consumption and overhead introduced by having to handle the state transitions. The `latency-tracker` module already supports a reference count mechanism to keep events active in memory even if they are not in the hash table anymore. We use this mechanism to

keep some state information in the root of the tree (the `do_IRQ` event) along with the origin timestamp. Each event always keeps a link to the root and each branch takes a reference to the root. When a relevant branch is identified, it sets a flag in the root node that the relevant branch has been found and marks the current node as relevant. If the relevant branch has already been identified with a `begin_work` event and a new branch is created, we have to clear the flag because any of the following branches might be the relevant one again.

The memory management and reference counting procedure is presented in Algorithm 6.

```
function ON_STATE_CHANGE(event, params)
    if root_branch_terminated then
        TEARDOWN_CURRENT_NODE
        return RELEASE_ROOT_REFCOUNT
    if root_relevant_branch_found then
        if ! current_branch_relevant then
            TEARDOWN_CURRENT_NODE
            return RELEASE_ROOT_REFCOUNT
    return DO_STATE_CHANGE(event, params)
function DO_STATE_CHANGE(event, params)
    if event == "begin_work" then
        MARK_CURRENT_BRANCH_RELEVANT
        MARK_ROOT_RELEVANT_BRANCH_FOUND
    else if event == "work_done" then
        MARK_ROOT_BRANCH_COMPLETE
        return FINISH_BRANCH
    else if event == ("sched_waking" or "softirq_raise") then
        if relevant_branch_found then
            CLEAR_RELEVANT_BRANCH_FOUND
        TAKE_ROOT_REFCOUNT()
        return INSERT_NEW_BRANCH(event, params)
    return STATE_TRANSITION(event, params)
```

Algorithm 6 Tree reference count management

This algorithm ensures that all memory is reclaimed after a branch is completed. One limitation of this approach is that filtering on the target process can be difficult. Indeed, if the filtered process is not the first user-space program that is woken up in the processing chain, we cannot apply the filtering on the process name or PID, since we never know if and when we will reach it. The effect of not applying the filters is that the memory usage grows and we can never collect it, since even the blocking states are not closing the branches. We end up only closing the branches that have ramifications in a relevant branch, but all the others

(belonging to other applications or the kernel for example) remain active. To circumvent this problem, when using this mode, we require that the user filter on the process waiting in the blocked system call, and close each branch that did not enter in user-space when it goes back to a blocking state.

5.6 Real-time Problems

In this section, we present some common problems related to real-time systems that we can identify with our algorithm.

5.6.1 Hardware Delays

When a device has data ready to be read by the system, it sends an interrupt. The kernel receives it and ends up reading the data from the device. If, for some reason, the device reading time is longer than expected, our methodology triggers an alert that can be used to extract the tracing buffers and understand why it happened. An example, which is hard to debug with the existing methodologies, is when working with write-combining devices. With these devices, write requests are buffered before being actually sent to the device, which helps for the latency of the write requests. However, if a read request arrives, all the buffered data must be flushed before the read request can be processed, in order to guarantee consistency. Depending on the size of the buffer, and the activity of the device, the delay to answer a read request might be unusually long and cause problems for a real-time process.

Depending on the use-case, this class of problems can be very difficult to diagnose because it appears to be random. A user-space application can be instrumented to detect long reading requests and trigger the collection of tracing buffers, but it requires some modifications on the worker process and detailed knowledge of the hardware communication. With our approach, we only enable the module, set a threshold, eventually filter on the target process or interrupt number, and let it run without any application-specific configuration. Over time, it produces a serie of tracing snapshots (one for each time a long latency was detected) that can be later inspected to accurately understand the cause of delays.

To simulate this behaviour, we introduced a one second delay in an interrupt handler and configured our module with a threshold of one second. Here is the output of the trace point emitted from the tracker when the latency is detected :

```
latency_tracker_rt: comm=sshd, delay=1000197873,
breakdown={
```

```
    do_IRQ [000] = 24286151893541,
    to irq_handler_entry(27) [000] = 12222,
    to softirq_raise(3) [000] = 1000074429,
    to softirq_entry(3) [000] = 34652,
    to sched_waking(1935) [000] = 41657,
    to switch_in(sshd-1935, 20) [002] = 34913
}
```

The `comm` field shows the target process for the interrupt, the `delay` field shows the total delay for the whole operation (from the interrupt noticed by the kernel to the scheduling of the target process). The `breakdown` field outputs the critical path of the interrupt processing, the first line (`do_IRQ`) shows the timestamp of the `do_IRQ` operation, the next lines output the time difference between the previous operation in nanoseconds, the number between brackets (`[000]`) is the CPU number.

As we can see, the tracker automatically detected a delay of one second between the `irq_handler_entry` and the `softirq_raise` events. Since no other interrupt processing took longer than one second during this capture, only one event was output out of the 50 interrupts that arrived in the system.

In this case, the tracker is configured to collect the breakdown of the chain automatically, so it can be interpreted alone without enabling other kernel events. This text breakdown can be disabled to limit the overhead. When the text breakdown is disabled, the breakdown field is empty, so we only record an event which indicates that an interrupt-related latency ended at this timestamp and started 1,000,197,873 nanoseconds earlier. The breakdown can then be recomputed offline using the trace, if it is recorded. The `wakeup_pipe` of the tracker can be used to trigger this recording Desfossez et al. (2016).

5.6.2 User-space Hang

Another kind of common user-space latency is when the code in user-space takes longer than a pre-defined period. Maybe it is doing too much computation, or depends on CPU instructions that are slower than expected in certain conditions. If the period starts by an interrupt (hardware or timer), we can use our system to track this delay, as long as we configure the tracker to wait for the next blocked state (or with the work begin/end user-space collaboration).

In this output, the `bash` process associated with a SSH connection informs the kernel that it is starting to work on a job and completes the job five seconds later :

```
latency_tracker_rt: comm=bash, delay=5442747644,
breakdown={
    do_IRQ [000] = 1649968838900,
    to irq_handler_entry(28) [000] = 26520,
    to softirq_raise(3) [000] = 6161,
    to softirq_entry(3) [000] = 12004,
    to sched_waking(2008) [000] = 48951,
    to switch_in(sshd-2008, 20) [000] = 62347,
    to sched_waking(4) [000] = 168235,
    to switch_in(kworker/0:0-4, 20) [000] = 24379,
    to sched_waking(2056) [000] = 9254,
    to switch_in(bash-2056, 20) [000] = 6999,
    to sched_waking(4) [000] = 108459,
    to work_begin [000] = 227780,
    to work_done [000] = 5442046555
}
```

We clearly see the link between the network interrupt, that lead to starting the job, to the end of this job, regardless of what happened between the `work_begin` and `work_end` events five seconds later. If we need more information, we can use the callback related to this occurence to extract the content of a trace from memory, and then do more advanced analyses based on this entry point.

5.7 Measurements

5.7.1 Test setup

In this section, we measure the overhead introduced by the active tracking of the state in the critical path. All of the measurements are made on a dual-processor Intel(R) Xeon(R) E5-2630 clocked at 2.40GHz with 32GB of RAM. (NUMA configuration with 16GB on each socket). The Hyperthreading, TurboBoost and power management features are disabled, so we have 16 cores available. The test machine is running Ubuntu Trusty 14.04 with the PREEMPT_RT kernel 4.1.10-rt11 compiled manually.

5.7.2 Probes Overhead

Since we are adding code in different places in the kernel, and we are working with hardware interrupts, we need to measure accurately the time spent in each individual probe and sum it

at the end to evaluate the overhead of our method, without being impacted by the hardware latency. The additional work in each probe consists in a lookup in the hash table to see if the current event is related to a previous event. If it is, update the state to prepare for the next event in the chain. Therefore, the overhead varies depending on the current state and whether or not a state transition is required.

Since the added code is located in very sensitive code paths (mainly interrupt handler and scheduler) and the overhead depends on several factors (caches and TLB state), we noticed that the profilers and tracers available were not suited for this type of measurement : the sampling resolution was too low and tracing added too much overhead and jitter to the measurements.

To solve this problem, we had to design a custom measuring method to get accurate results. Our code is running in a tracepoint probe, so we know that preemption is disabled. For our measurements, we also disable the local interrupts and discard the results if NMI or MCE happened during the critical section. In addition to the time spent, we are also in an ideal situation to read the hardware `perf` PMU (Performance Monitor Unit) counters and accurately observe the impact of our code on the various cache levels and TLB state. The overall behaviour of this measuring framework is presented in Algorithm 7. For conciseness, in this example we only verify the NMI counter and record one PMU counter.

```
function ON_MODULE_LOADING
    INIT_PER_CPU_RESULTS_ARRAY
    START_PROCESSING
function ON_STATE_CHANGE(event, params)
    DISABLE_LOCAL_INTERRUPTS
    cpu ← CURRENT_CPU
    nmi ← NMI_COUNT(cpu)
    perf1 ← GET_PMU_COUNTER
    ts1 ← CLOCK_GETTIME
    DO_STATE_CHANGE(event, params)
    ts2 ← CLOCK_GETTIME
    perf2 ← GET_PMU_COUNTER
    if nmi == NMI_COUNT(cpu) then
        APPEND_PER_CPU(cpu, ts2-ts1, perf2-perf1)
    RESTORE_LOCAL_INTERRUPTS
function ON_MODULE_UNLOADING
    OUTPUT_PER_CPU_ARRAY
    FREE_PER_CPU_ARRAY
```

Algorithm 7 Overhead measuring algorithm

We inserted this measurement algorithm in the function responsible for handling the state transitions : if the entry is present in the hash table it performs a transition by inserting the next key and removing the old one if necessary (see Table 5.1). On the other hand, if the entry key is not present, it immediately returns without any transition.

For a test period of 4.5 seconds, on a mostly idle server, we observed a ratio of 1/161 between the events that lead to a transition and the events that immediately returned. The results from our measurements are presented in Table 5.3. The frequency distribution of the duration of transitions are presented in Figures 5.6 and 5.7.

TABLE 5.3 Transitions measurements

Metric	**Transition**	**No transition**
Ratio of requests	0.6%	99.4 %
Average latency	1136.93 ns	259.13 ns
Standard deviation	278.71 ns	28.42 ns
Minimum latency	565 ns	237 ns
Maximum latency	3028 ns	1938 ns
Average instruction count	2024	756
Average L1 misses	38.78	3.04
Average LLC misses	3.66	0.003
Average TLB misses	0.12	0.002
Average branch misses	3.08	0.15

Since these measurements are performed in a very controlled execution environment, we were able to use the PMU counters to prove that the outliers in the distributions are mainly related to the branch misses. We can also see a correlation with the L1 cache misses. Figure 5.8 presents the correlation between the latency of the transitions and the number of branch misses.

Since a typical path from an interrupt to user-space is usually composed of 7 transitions, we can conclude that our average overhead when an interrupt is tracked is 7.95 micro-seconds.

We measured that the hashing algorithm (`jhash`) is responsible for an average of 220 ns for each request, so it is the major factor when there are no transitions.

5.7.3 Impact Of The State Tracking

We are trying here to determine what is the impact of the additional work on various workloads. We start by doing some micro benchmarks to stress important ressources and then experiment with real workloads.

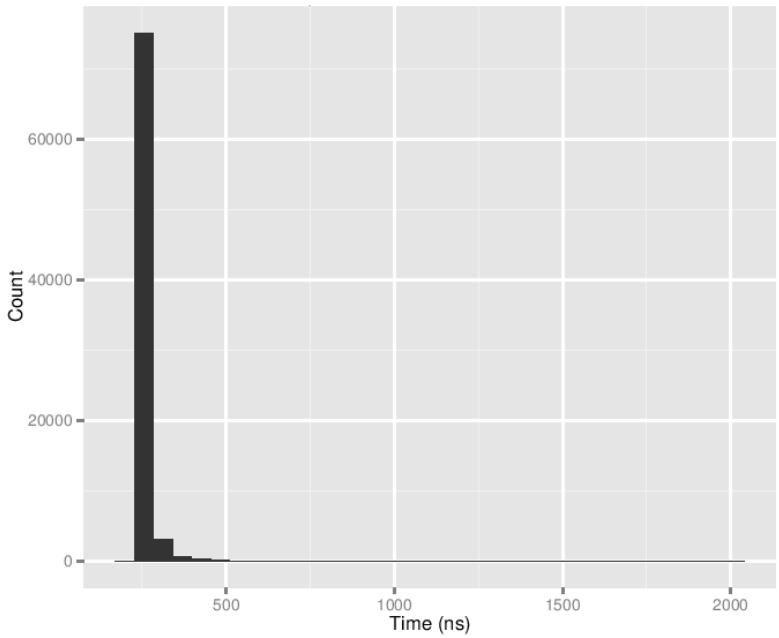

Figure 5.6 Overhead time distribution without transitions

As expected, the performance impact was linked to the number of interrupts generated during the workload.

For file I/O measurements, we used `sysbench` to create 100000 random read and write requests in 128 files of 16M in parallel (16 threads). We measured an overhead of 5.1% when the tracker is loaded. The test was run 10 times, each time lasted between 9.1 and 10.6 seconds, and during this time we tracked 1.1 million state changes.

We performed a network stress-test using `iperf` with a similar machine connected at 1Gbps to our test machine, once again we did not observe any performance difference : 942Mbps of TCP traffic with and without the tracker loaded. We tracked 81300 network interrupts during this test and 4713 wakeup events of the `iperf` process. During this 10 second test, 532745 state changes were recorded in the tracker.

We did the same network test but with 10Gbps/s network interfaces, we had to allocate 3.1Gbps to be able to track all the packets (1.2 million state changes for 10 seconds). In terms of throughput, we went from 8.02Gbps/s to 7.70Gbps/s so we impacted this workload by 3.89%.

To measure the overhead on a more real-world application, we used the `sysbench` oltp test. At 16 threads, 190000 read/write requests are issued in 10000 transactions. We repeated this test 10 times with and without the tracker. We noticed a slow down of 4.84% when the

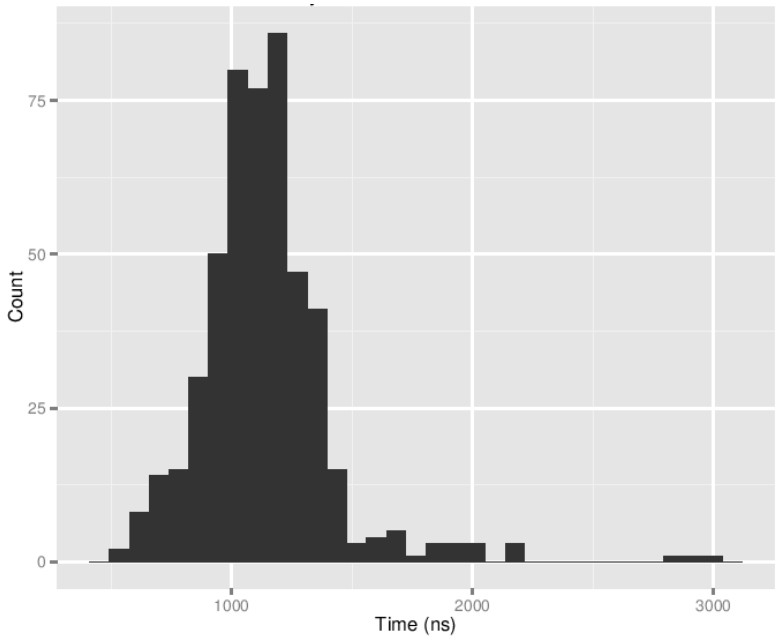

Figure 5.7 Overhead time distribution with transitions

tracker was loaded. During the 2.3 seconds test, it tracked 114000 state changes.

All the results are presented in Table 5.4.

Table 5.4 Transitions measurements

Test	Baseline	Tracker	Overhead
CPU	19.20s	19.20s	0.00%
Memory	32.33s	32.37s	0.30%
File Read/Write	9.04 s	9.50 s	5.10%
Network 1Gbps	942Mbps/s	942Mbps/s	0.00%
Network 10Gbps	8.02Gbps/s	7.70Gbps/s	3.89%
OLTP (MySQL)	2.27s	2.38s	4.84%

5.8 Limitations And Future Work

The link from the kernel to user-space depends on the `sched_waking` event being emitted. The problem with this approach is that this event is only emitted if the target process is blocked waiting for data when the interrupts arrive. If the thread is running, the event is not emitted and we miss the link. Arguably, if the target thread is not blocked, the response time should be low enough, but it could also be a sign of abnormal behaviour. When data is ready on a file descriptor, the Linux kernel calls a callback that checks if a task if waiting for

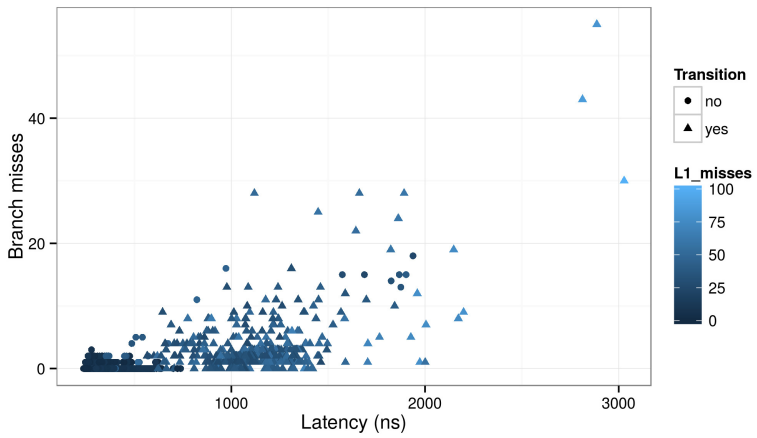

Figure 5.8 Latency of the transitions and amount of branch misses

data on this FD. If no task is waiting, no wake-up event is necessary so we miss the link ; the interrupt handling finishes and we cleanup the state. To circumvent this problem, we would have to add code in the callback and keep a mapping between all socket addresses and file descriptors in processes. We would also need to take into account duplicated and shared file descriptors among processes. This has not been implemented yet, it is left as future work.

In terms of performance improvement, the hashing algorithm is clearly a major factor in the total overhead and we should look at more efficient algorithms than `jhash` ; we could explore some platform-specific optimizations.

5.9 Conclusion

In this paper, we have demonstrated that we can accurately measure the whole event-response critical path, from the interrupts to the end of processing, in real-time with an average overhead of less than 8 micro-seconds. This method, currently limited to blocking processes waiting for data, gives us insight into how interrupts are handled in the kernel and in user-space. Our method can work automatically for simple use-cases or can be user-assisted for the more complex processing chains. It helps identify abnormal behaviours and patterns for sporadic problems. The events generated by our module can be used as entry points inside kernel traces, and we can use the alert callbacks to extract the trace buffers to disk. This combination of online and offline kernel tracing provides a good compromise between fast analyses, low overhead and high level of details.

5.10 Acknowledgments

This work was made possible by the financial support of Ericsson, EfficiOS, Prompt and NSERC.

CHAPITRE 6 RÉSULTATS COMPLÉMENTAIRES

Ce chapitre présente des résultats supplémentaires obtenus lors de cette recherche mais qui ne font pas partie des articles. Trois axes sont abordés :

— une approche de machine à état en direct basée sur les traces avec un niveau de granularité variable dans le temps ;
— un moyen de représenter en temps réel la distribution des latences des opérations d'entrée-sortie.
— un mécanisme pour réaliser des mesures très précises sur des sections courtes de code.

6.1 Machine à état en direct du système d'exploitation avec granularité variable

Au début d'une trace noyau LTTng, on active souvent des événements appelés `statedump`. Ces évènements consistent en une extraction rapide de l'état courant du système. On y retrouve la liste des processus existants, les descripteurs de fichiers ouverts par processus, les partitions, etc. Ces évènements sont très utiles car ils permettent de donner le contexte nécessaire aux évènements suivants. Ainsi, si une opération est faite par un processus sur un fichier déjà ouvert, il est possible de faire correspondre le numéro de descripteurs au nom du fichier, ce qui n'aurait pas été possible sans le `statedump` ni l'événement d'ouverture.

À la suite des travaux sur l'article présenté au Chapitre 3, qui portait sur l'analyse de traces en direct pendant la capture, il est apparu important d'avoir un moyen de connaître en tout temps l'état d'un système. En particulier, on cherchait à connaître en tout temps la liste des processus actifs sur le système ainsi que l'origine de tous les descripteurs de fichiers ouverts par tous les processus. Ainsi, si en cours de route un problème était détecté sur un descripteur de fichier particulier et un numéro de processus, il nous était possible de retracer son origine (nom du fichier et date d'ouverture) sans avoir à conserver toute la trace depuis le début. Ce système étant conçu pour fonctionner en permanence, il nous fallait également un moyen de limiter la quantité d'information stockée, donc avoir une résolution variable en fonction de l'âge des informations. Aussi, ce système était prévu pour fonctionner sur les serveurs dans les centres de donnée, donc il devait fonctionner en réseau.

Pour arriver à ces objectifs, une machine à état a été conçue et implémentée de manière expérimentale. Les principes de base étaient :

— Garder un historique détaillé de toutes les opérations (début/fin de processus, ouverture/fermeture de fichiers, appels systèmes d'entrées-sorties, etc) sur 10 secondes ;

— Après 10 secondes, garder seulement les événements de création de processus et d'ouverture de fichier pour lesquels l'évènement de fin ou de fermeture n'est pas encore arrivé ;
— Gérer la machine à état côté serveur, donc le fournisseur de données enverrait des événements pré-conditionnés, pas les événements bruts, par exemple : `OpenFile(timestamp, pid, filename, fd)` ; ainsi l'implémentation ne dépendrait pas d'un format de trace ou d'un client spécifique ;
— Concevoir des requêtes permettant d'obtenir de l'information sur un processus ou un descripteur de fichier par rapprt à l'état courant ;
— Service facile à mettre à l'échelle pour permettre d'augmenter le nombre de machines suivies au besoin.

Le côté permettant de s'attacher à une ou plusieurs sessions en direct a été mis en place sous forme d'un plugiciel associé à `Babeltrace`. Celui-ci communiquait ensuite avec une base de donnée `Redis` (choisie parce qu'il s'agit d'une base de donnée en mémoire de type clé-valeur). La machine à état quant à elle était directement dans le moteur de base de donnée sous forme de scripts `Lua` dédiés à chaque opération (création/suppression de processus, ouverture/fermeture de fichiers, appels systèmes d'entrées-sorties, gestion de la collecte des vieilles données (*garbage collection*), etc).

La preuve de concept a ainsi été faite qu'il est possible de garder un état cohérent en parallèle avec l'exécution de la machine en se basant uniquement sur les traces reçues en direct par le protocole `LTTng live`. Par contre, la quantité de calcul requise sur la machine devant gérer l'état en tout temps était beaucoup trop élevée pour que l'approche puisse se mettre à l'échelle. En effet, un serveur de base de données avec un processeur plus puissant que la machine qu'il devait superviser n'arrivait pas à suivre le débit de trace qu'il devait traiter et son retard augmentait avec le temps. Cela est dû au grand nombre d'événements inutiles qu'il recevait. En effet, au moment de ce travail, le traceur noyau de `LTTng` n'avait pas le concept de filtrage ni la possibilité de choisir les appels systèmes à activer. Le pourcentage de donnée de trace inutile revenait à 10% de ce que l'analyseur recevait. Le but étant de faire de ce système un service réseau partagé par potentiellement un grand nombre de serveurs, l'approche a été jugée non-viable. Un autre critère ayant poussé la décision de ne pas chercher à plus optimiser l'approche était que le système ne serait pas utile la majorité du temps. Seulement dans les cas de problèmes complexes, ce style de base de donnée d'état serait pertinent. Donc, avoir toute cette charge pour seulement des cas extrêmement rares semblait d'un faible intérêt.

La décision a conséquemment été de se concentrer sur la détection de problèmes en temps réel et la collecte d'informations de contexte au moment exact où le problème surgit, ce qui

a donné naissance aux idées des deux articles qui ont suivi.

6.2 Présentation en direct de la distribution des latences d'opérations d'entrées-sorties

Dans les environnements d'infonuagique, les instances sont réparties sur des serveurs physiques par un système d'orchestration. Ces systèmes essaient de répartir les instances sur les machines physiques disponibles afin de mieux utiliser les ressources et donner une expérience satisfaisante aux utilisateurs. Un des défis majeurs dans ces environnements est de gérer de manière adéquate la concurrence sur les périphériques partagés de type entrées-sorties. En effet, il s'agit de matériel physique avec des latences variables en fonction des opérations et de la concurrence, donc répartir les utilisateurs selon leur entente de service peut constituer un défi important.

Un phénomène commun dans ces environnements est le *noisy neighbour*, une instance déployée sur un serveur physique qui utilise tellement les ressources que le système n'arrive pas à arbitrer correctement et les autres instances cohabitant s'en trouvent ralenties.

Suite à l'article présenté au Chapitre 4, l'idée d'utiliser ce nouveau système pour présenter en direct le profil courant de la distribution des latences d'entrées-sorties est apparu. En effet, le surcoût très faible imposé par cette approche permettait de le garder actif en tout temps et d'extraire très facilement un profil de la machine.

L'objectif était ici d'être en mesure d'établir un profil d'une machine et par la suite de déterminer si une opération était dans les normes ou exceptionnelle pour cette machine. L'idée était ensuite d'automatiser la création de profil et la détection d'anomalies afin de détecter automatiquement si le profil des opérations d'entrées-sorties d'une machine changeait subitement de manière indépendante de la charge courante de l'instance. Lorsque de nouvelles latences étaient détectées, on voulait être en mesure de prouver qu'elles n'étaient pas causées par un changement local (tel qu'une mise à jour de code ou un plus grand nombre d'utilisateurs), mais bien par un changement dans l'environnement (tel que l'arrivée d'une instance aggressive sur le même serveur physique), sans avoir la vue de l'hyperviseur.

La création du profil d'entrées-sorties en temps réel a été conçue et intégrée au projet `latency-tracker`. Il s'agit d'un fichier dans `debugfs` qui peut être lu à tout moment et qui affiche un tableau de distribution des latences de plusieurs catégories d'opérations liées aux fichiers et aux périphériques de type `block`. Les compteurs peuvent éventuellement être remis à zéro à chaque lecture. Ainsi, il est facile d'obtenir le profil autour d'une commande.

Un exemple de profil capturé sur un serveur réel hébergeant des services de courriel et web

est présenté dans la Figure 6.1. On y voit le nombre de requêtes par catégorie qui se sont complétées dans un intervalle de temps. Les catégories sont dans l'ordre :

— les appels systèmes de lecture (ex : `read, readv, recvmsg`),
— les appels systèmes d'écriture (ex : `write, writev, sendmsg`)
— les appels systèmes lecture et écriture combinés (ex : `sendfile, splice`,
— les appels systèmes synchronisation des caches (ex : `sync, fsync, fdatasync`),
— les appels systèmes ouverture (ex : `open, connect`),
— les appels systèmes fermeture de fichiers (ex : `close, shutdown`),
— les opérations de lecture de blocs,
— les opérations d'écriture de blocs.

Range	s_read	s_write	s_rw	s_sync	s_open	s_close	b_read	b_write
[1ns, 2ns[	0	27	0	0	0	0	0	0
[2ns, 4ns[	0	0	0	0	0	0	0	0
[4ns, 8ns[	0	0	0	0	0	0	0	0
[8ns, 16ns[	0	0	0	0	0	0	0	0
[16ns, 32ns[	0	0	0	0	0	0	0	0
[32ns, 64ns[	0	0	0	0	0	211	0	0
[64ns, 128ns[	46896	358	0	4	115142	2784763	0	0
[128ns, 256ns[	454251762	4062745	0	239	708593	24746514	0	0
[256ns, 512ns[	222570993	13857969	0	2247	4011796	32044590	0	0
[512ns, 1us[	873102685	54628730	0	6899	10786875	8306124	0	0
[1us, 2us[	1009908482	146361897	3	4651	16535392	2003785	0	0
[2us, 4us[	485365620	220228207	6	5786	25022023	60742	0	0
[4us, 8us[	73471567	93383523	86	3205	13115679	10748	0	0
[8us, 16us[	29161187	134699476	246	640	3037085	67621	0	0
[16us, 32us[	38748776	290538329	1646	346	1012850	18565	0	0
[32us, 64us[	10269901	81184574	1157	110	407154	7697	285498	0
[64us, 128us[	3090275	2488445	407	22	383163	847	1618492	0
[128us, 256us[	2780181	89765	143	15	905194	89	10712425	0
[256us, 512us[	1468721	7844	17	7	616739	24	11362145	0
[512us, 1ms[	1329321	2912	4	1	37895	48	3062329	1
[1ms, 2ms[	674593	1793	2	56	63096	65	2305734	536
[2ms, 4ms[	532696	1759	15	4827	12688	38	3861904	20883
[4ms, 8ms[	857063	3991	41	320092	13662	3	1087695	852597
[8ms, 16ms[	421769	11745	51	300832	11591	0	743540	1364989
[16ms, 32ms[	95921	3795	13	266578	3834	0	266913	941398
[32ms, 64ms[	135097	1258	3	35695	1732	0	49218	458174
[64ms, 128ms[	11517	797	1	9855	806	0	13240	274447
[128ms, 256ms[	4843	539	0	1127	243	0	7735	182823
[256ms, 512ms[	31077	533	0	260	120	0	7487	535156
[512ms, 1s[	344954	101	0	252	139	0	1411	3162
[1s, 2s[	2850	74	0	262	135	0	678	2760
[2s, 4s[	2124	49	0	245	76	0	375	1655
[4s, 8s[	186	17	0	83	12	0	58	273
[8s, 16s[	53	0	0	0	2	0	0	0
[16s, 32s[	34523	0	0	0	0	0	0	0
[32s, 64s[	32	0	0	0	0	0	0	0
[64s, 128s[	50	0	0	0	0	0	0	0
[128s, 256s[	8	0	0	0	0	0	0	0
[256s, 512s[	10	0	0	0	0	0	0	0
[512s, 1024s[	8	0	0	0	0	0	0	0
[1024s, 2048s[	0	0	0	0	0	0	0	0

Figure 6.1 Exemple de distribution des latences d'opérations d'entrées-sorties après 3 jours d'exécution

Une fois ce profil créé, nous avons étudié ce qui était observable avec ces nouvelles métriques sur différentes instances Amazon. Nous avons commencé par faire une expérience où l'on créait une nouvelle instance avec un disque SSD, chargeait le module, démarrait un test de charge de base de donnée, on collectait le profil, et on enregistrait et détruisait l'instance pour en créer une nouvelle. Après 100 itérations, nous avons obtenu le résultat présenté à la Figure 6.2. On peut y voir très clairement deux modes presques identiques dans toutes les expériences. Les deux modes illustrent très bien comment Amazon contrôle les ressources, ils utilisent un algorithme de type seau de jeton (*token bucket*) qui se remplit à un certain rythme. Tant qu'il y a des jetons dans le seau, les requêtes vont à pleine vitesse, dés que le seau est vide, il faut attendre qu'il se remplisse pour pouvoir émettre de nouvelles requêtes. La vitesse de remplissage est variable en fonction du nombre total d'opération réalisées. Au début, le seau se remplit rapidement, donc il est possible de faire les installations de base, mais au bout d'un certain temps, le seau se remplit moins vite. Avec les résultats obtenus, on voit clairement l'impact de passer d'une classe à une autre.

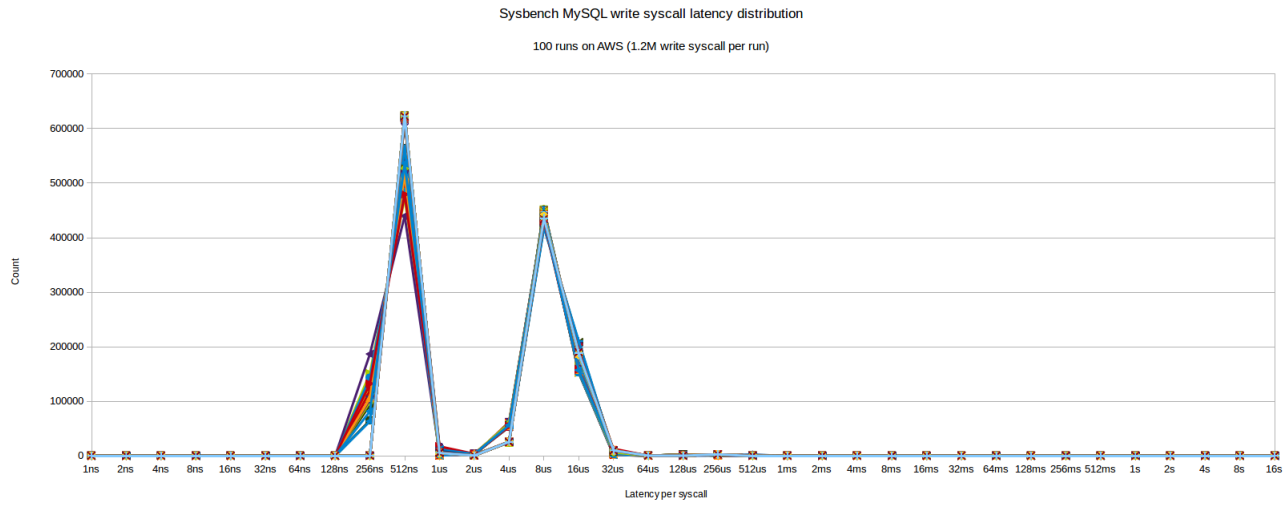

Figure 6.2 Distribution des latences sur 100 instances Amazon

Suite à ces résultats, il est apparu clair qu'Amazon a un grand contrôle sur l'utilisation des ressources et que les performances n'étaient pas si chaotiques que ce que la littérature sur le sujet semblait dire. Il est possible que si nous avions choisi des instances à base de disques durs rotatifs plutôt que des SSDs, nous aurions vu un peu plus de variabilité dans les résultats. Cependant, cette technologie est amenée à disparaître et cela ne semblait pas pertinent puisque les instances par défaut sont équipées de disques SSD maintenant.

Pour aller plus loin dans l'élaboration de notre algorithme, nous avons donc réalisé des

expériences dans des environnements contrôlés avec des machines physiques sur lesquelles nous avions le contrôle. Afin d'automatiser la détection de changement de comportement, nous avons utilisé la méthode statistique du *Chi-squared test* pour comparer les distributions enregistrées. Plus l'indice était élevé, plus la distribution était différente de la précédente.

Dans l'expérience suivante, nous avons lancé le même test de base de données dans une machine virtuelle, mesuré le nombre d'opérations par seconde dans la base de donnée, puis lancé des charges de travail sur le disque dur utilisé par la machine virtuelle depuis la machine physique. L'objectif était de perturber les opérations d'entrées-sorties de la machine virtuelle sans qu'elle puisse le savoir. Nous avons utilisé deux types de charges différentes : tout d'abord une compilation de noyau Linux, puis un `rsync`. Les mesures du nombre d'opérations par secondes à la base de données et le calcul de la distribution des entrées-sorties étaient calculés depuis l'intérieur de la machine virtuelle.

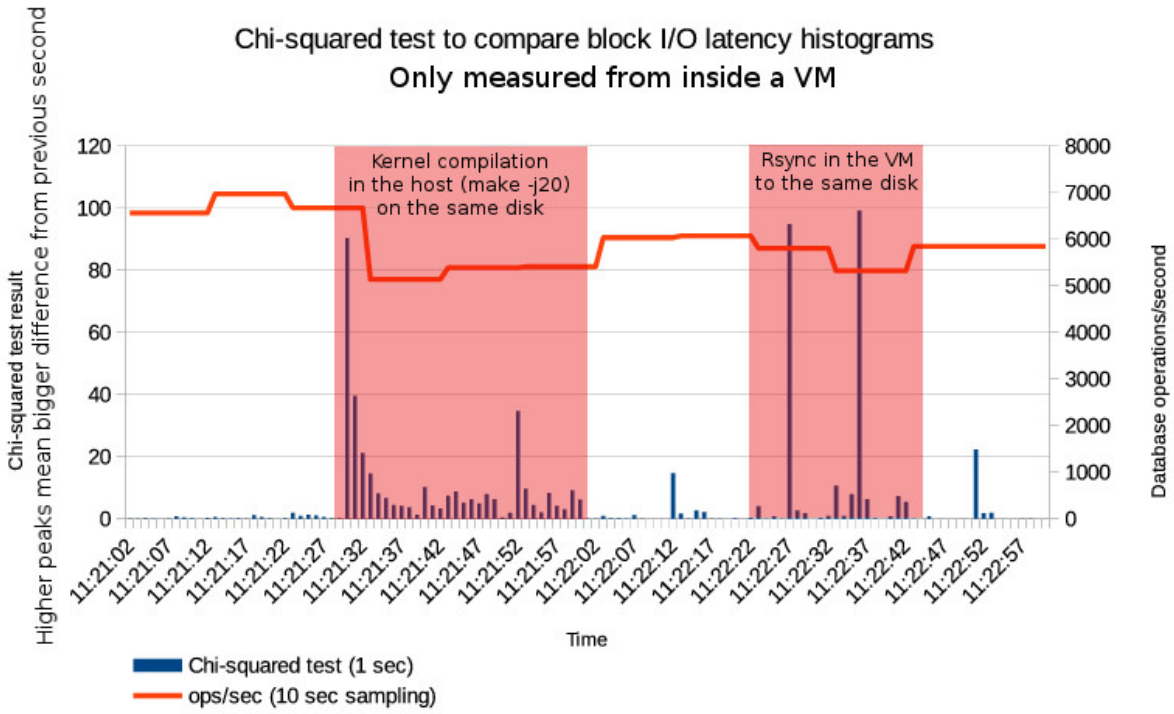

Figure 6.3 Mesure des perturbations sur un disque physique depuis l'intérieur d'une machine virtuelle

Comme on peut le voir sur la Figure 6.3, le *Chi-squared test* a clairement des valeurs plus élevées lorsqu'une perturbation arrive sur le disque, mais il est stable lorsque le test de base de donnée fonctionne de manière normale. La courbe représentant le nombre d'opérations par seconde présente la même tendance. Avec cette expérience, on voit qu'il est possible de faire corréler ces métriques. Il est ainsi possible de montrer que si soudainement le nombre d'opé-

rations par seconde change pour un serveur de base de données, ce n'est pas nécessairement parce qu'il y a moins d'utilisateurs.

Le problème majeur de cette approche est qu'il n'est pas possible d'interpréter la valeur du test de manière positive ou négative. Tout ce qui est mesuré est l'intensité du changement. Cela explique pourquoi on voit plusieurs changements durant les activités de perturbation : un pour le début de la perturbation et l'autre pour le début du retour à la normale. Avec ce problème, nous en sommes arrivé à la conclusion que l'approximation faite par le noyau sur la durée passée dans la file d'attente des disques durs apportait une information plus compréhensible pour l'utilisateur, bien que beaucoup moins précise.

6.3 Mécanisme de micro-mesures de latences

La méthodologie développée pour l'article présenté au Chapitre 3 a été créée dans le but d'instrumenter une section très courte de code noyau et de faire corréler la latence d'exécution de cette section avec l'évolution des différentes ressources de la machine. En particulier, on s'intéresse aux défauts de mémoire cache de niveau 1 et 3, aux défauts de TLB, aux cycles de processeurs passé en attente, etc. Pour réaliser ces mesures, un en-tête C a été élaboré pour être simple à utiliser dans n'importe quelle section courte du noyau. La méthodologie de mesure mise en place par cet outil consiste à :

- désactiver les interruptions,
- désactiver la préemption,
- lire le compteur d'interruptions non-masquables (NMI et MCE),
- lire les compteurs de performance voulus (`Perf PMU`) sur le processeur courant,
- lire le compteur de cycles (seulement s'il est garanti constant),
- exécuter la section de code,
- relire le compteur de cycles,
- relire les compteurs de performance,
- relire le compteur d'interruptions non-masquables,
- restaurer les interruptions et la préemption au même état qu'au début,
- si une NMI ou MCE a eu lieu pendant l'exécution du code, ne pas enregistrer les résultats,
- sinon, enregistrer la différence entre les deux lectures de compteurs de performance dans une région mémoire pré-allouée.

Le résultat de l'application de cette méthodologie permet de faire des mesures très précises et ainsi établir un lien entre une exécution et l'évolution de certains compteurs. Un exemple

simplifié de résultats produits par cette analyse est présenté à la Figure 6.4. Pour référence, la vue complète est présentée à l'Annexe A. L'expérience consistait à mesurer le temps exact d'une recherche dans la table de hashage du `latency-tracker` lorsque l'élément recherché n'était pas présent. Les résultats sont présentés sous forme d'une matrice mettant en lien toutes les métriques. Ainsi, on peut rechercher des tendances communes. Dans cet exemple, on peut y voir que les seules métriques qui ont un comportement différent lorsque la latence est élevée sont `cpu_cycles` et `branch_miss`. Pour le nombre de cycles, c'est attendu puisque la latence est liée au nombre de cycles, par contre pour les fautes de prédiction de branche, on voit bien que ce nombre élevé arrive seulement lorsque la latence est plus élevée, donc on peut faire l'hypothèse que la latence est liée à ce problème. Bien sûr, il ne s'agit pas d'une corrélation forte. Pour aller plus loin dans cette analyse et avoir la certitude, il faudrait utiliser du traçage matériel de processeur.

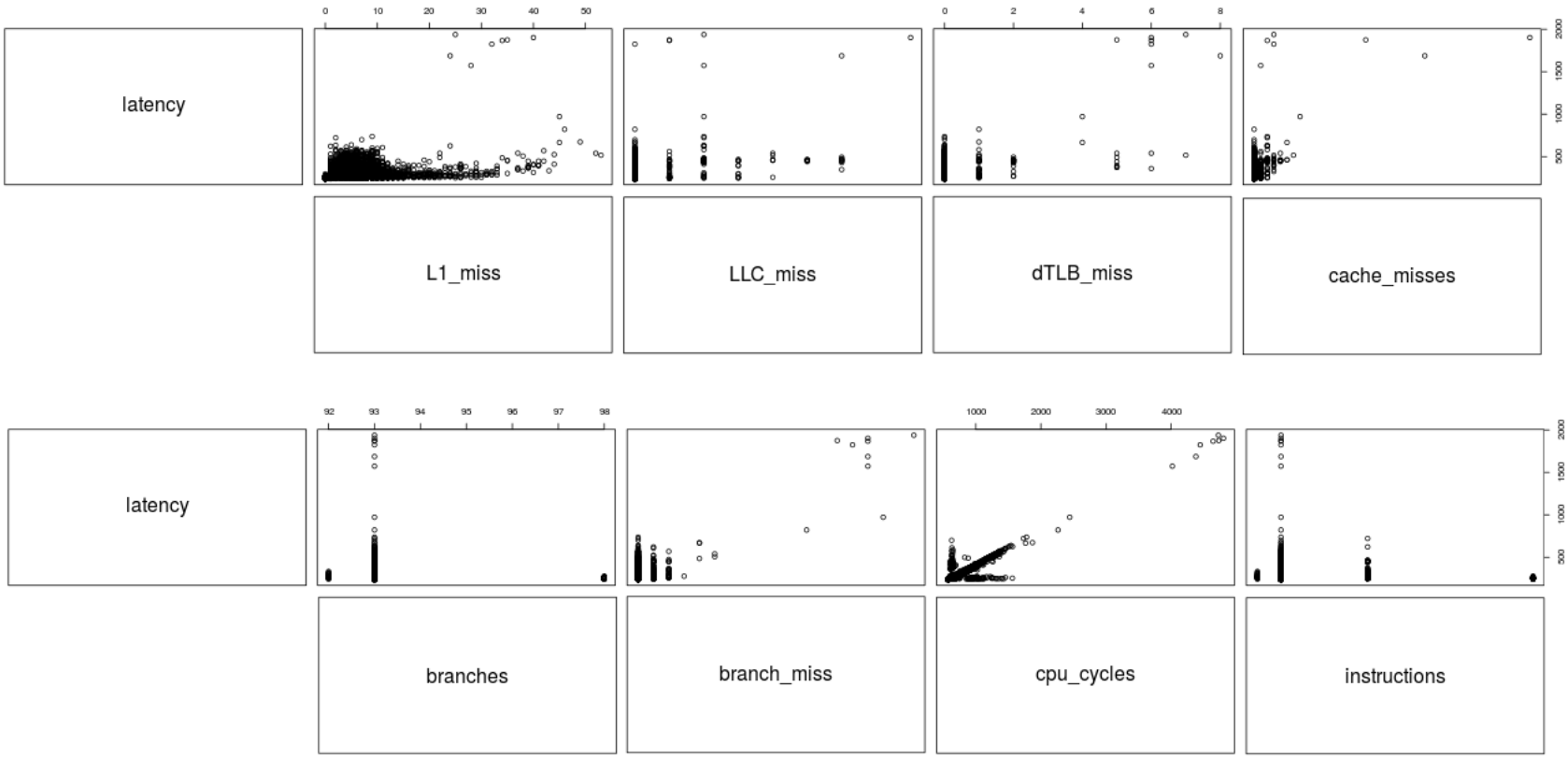

Figure 6.4 Matrice des différentes métriques système

CHAPITRE 7 DISCUSSION GÉNÉRALE

7.1 Atteinte des objectifs

Nous effectuons un retour sur les objectifs de départ pour évaluer dans quelle mesure ils ont été atteints.

Analyse de traces concurrentes en direct : Il a été possible de créer un algorithme permettant l'analyse de traces de systèmes multi-coeurs pendant que la capture est en cours. Le résultat permet de traiter des traces pendant leur capture tout en garantissant qu'aucun événement n'est accessible dans le désordre, que les méta-données (nécessaires à l'interprétation de la trace) sont toujours disponibles avant les données, que les flux de données s'ajoutant en cours de trace sont détectés et intégrés à la lecture au bon moment, que les processeurs inactifs ne sont pas réveillés trop fréquemment, et que la balance entre surcoût et latence d'accès aux données est contrôlable par l'utilisateur en fonction du besoin. Le résultat de la complétion de cet aspect permet un accès distant rapide à une trace en cours de capture, permettant d'extraire le plus vite possible les métriques nécessaires à la résolution de problèmes complexes distribués.

Mesures de latences dans le noyau Linux : Il a été possible de mettre en place et d'optimiser des structures de données et des algorithmes permettant de mesurer des latences entre deux points dans le noyau Linux de manière efficace. Les contraintes de cet aspect étaient liées à la parallélisation, aux environnements d'exécution (gestionnaires d'interruptions et ordonnanceur principalement), à la consommation des ressources, et à l'impact de garder et d'accéder un état global de manière efficace dans le chemin critique du noyau. Le résultat de cet aspect est une méthode générique et simple d'utilisation pour mesurer un délai entre deux emplacements arbitraires, et pour déclencher des actions lorsque le délai est supérieur à un seuil. Le seul critère requis est d'avoir un élément commun entre le début et la fin de la période à mesurer. Cette méthode a été testée dans différents sous-systèmes (interruptions, appels systèmes, périphériques de type disque et réseau) et a permis de déclencher des captures de *snapshots* LTTng pour des analyses plus détaillées par la suite.

Mesure du délai de réponse en temps réel : Il a été possible de suivre la chaîne de transitions d'états entre l'occurence d'une interruption et la fin du traitement en espace utilisateur lié à cette interruption. Le travail lié à ce suivi a été optimisé pour permettre son utilisation dans les conditions les plus exigeantes. Les contraintes de cet aspect étaient la performance de la solution, mais également la création et l'élagage d'un arbre d'état en

mémoire pouvant grandir de manière arbitraire tant que les indices concernant la validité d'une branche n'étaient pas connus. Les heuristiques pour déterminer la fin du traitement fonctionnent dans les cas simples. Pour les cas plus complexes (traitements asynchrones ou scrutation active principalement), des outils permettent à l'application de collaborer avec le module de suivi pour le renseigner sur l'état d'une requête. Beaucoup de travail est passée dans l'optimisation des performances et la possibilité de borner le surcoût de traitement pour le rendre compatible avec les systèmes temps-réel garantis.

Tout ce travail a été réalisé en gardant toujours comme requis que les systèmes cibles peuvent autant être des serveurs puissants que des appareils embarqués. En particulier, l'objectif a toujours été de rendre les algorithmes et structures de données assez efficaces pour qu'ils soient déployables de manière permanente en production sans impact majeur. Les garanties que nous avons respectées sont de toujours présenter les données de manière cohérente dans le temps, de donner la possibilité à l'utilisateur de trouver l'équilibre entre vitesse d'accès aux données et intrusivité, de faire le minimum de travail dans le chemin critique des applications et faire le travail plus demandant en dehors (analyses, allocations de mémoire, collecte des déchets), de respecter les contraintes d'architecture (NUMA, impact des structures de données sur les lignes de cache), et d'avoir un impact prévisible et borné.

Avec de nombreux exemples, nous avons prouvé que l'approche hybride entre agrégation et traçage fonctionne pour les cas complexes. Le traçage en mémoire est un outil essentiel pour ce travail et l'ajout de la détection automatique rend son utilisation très puissante. En effet, on combine ainsi l'avantage principal du traçage qui est de présenter un état complet du système avec la l'avantage de l'agrégation qui est de présenter des résultats concis. Avec ces deux mondes réunis, nous avons créé une nouvelle manière d'investiguer les problèmes de performance qui ne se limite pas à ceux illustrés dans cette recherche. Nous pouvons maintenant détecter pendant l'exécution les latences anormales, extraire les informations détaillées sur les quelques instants ayant précédé le problème, et lancer des analyses avancées sur cet historique pour comprendre les problèmes de manière non-équivoque. Cela élimine le besoin de reproduire le problème, de connaître à l'avance l'instrumentation à activer et d'essayer d'appliquer des correctifs à l'aveugle en espèrant avoir corrigé le problème. Tous les résultats de ce travail se retrouvent dans le projet `latency-tracker`.

CHAPITRE 8 CONCLUSION ET RECOMMANDATION

Cette recherche a permis de faire avancer l'état de la connaissance dans le domaine de l'analyse des problèmes de latence dans les systèmes en temps réel. La réalisation principale de cette recherche est la création d'algorithmes pour détecter et comprendre des problèmes complexes de latences anormales avec un impact minimal. Les contributions incluent :

— Un algorithme de lecture de trace de systèmes parallèles en direct avec un contrôle sur le surcoût ;
— Une méthode pour mesurer des latences arbitraires de manière efficace même dans les cas de concurrence ;
— Un algorithme de calcul du délai de réaction de systèmes temps réel entre l'arrivée d'une interruption et la fin du traitement en espace utilisateur.

En conclusion, l'approche préconisée alliant traçage noyau et agrégation optimisée dans le chemin critique permet de mesurer efficacement et d'expliquer en direct ou *a posteriori* des latences dans des conditions réelles d'exécution. Elle est adaptée aussi bien au noyau Linux officiel que sa variante temps réel, ce qui permet de l'appliquer dans de nombreuses conditions d'exploitation. Les mesures montrent que les différentes méthodes présentées sont applicables dans des situations réelles, et l'approche est assez générique pour détecter des problèmes sans nécessiter trop d'information préliminaire.

Les résultats obtenus automatiquement par ces analyses dans des cas d'utilisation réels montrent principalement des problèmes d'utilisation abusive des ressources, d'interactions mal contrôlées entre différents composants et d'inefficacités dans l'utilisation de la parallélisation des tâches. L'échelle de temps de ces problèmes commençant dans l'ordre des dizaines de micro-secondes, il n'existe pas d'alternative à cette approche.

8.1 Limitations de la solution proposée

La limitation principale actuellement est liée au fait que le suivi de l'exécution en espace utilisateur dépend de la coopération de l'application, si l'on souhaite une vue complète. Les heuristiques en place fonctionnent seulement dans les cas d'applications simples.

Une autre limitation actuelle est que le lien entre une interruption et l'application de destination est basé sur un événement de réveil, ce qui implique que l'application doit être en attente de l'événement. Dans un cas où une application serait active mais en train de travailler dans une autre section du code (potentiellement par erreur), nous ne verrions pas le lien entre

l'interruption et le reste du traitement. Donc, pour l'instant, la solution se limite aux cas où il y a toujours un processus en attente des événements et qui relaie éventuellement le travail à d'autres processus.

Du côté performance, la limitation principale est l'algorithme de hachage qui est responsable de la majeure partie du surcoût imposé. En effet, Ajouter 8 micro-secondes à chaque requête pertinente risque d'être un frein à l'utilisation de cette approche.

8.2 Améliorations futures

Il faudrait ajouter d'autres heuristiques dans le système pour qu'il soit capable de faire automatiquement le suivi du chemin critique d'une application en espace utilisateur lorsqu'elle a reçu des données sur un descripteur de fichier. Les données requises pour faire cette analyse de manière automatisée sont disponibles pour résoudre la majorité des cas. Nous pourrions ainsi limiter les besoins de collaboration des applications aux cas plus difficiles à modéliser.

Il serait également utile de faire le lien entre les adresses des *sockets* et les descripteurs de fichiers dans les processus afin de pouvoir faire un lien entre une interruption et un processus, sans nécessiter les événements de réveil. Ainsi, on pourrait facilement voir si une application n'est pas bien conçue pour répondre à la demande du réseau. Cela pourrait indiquer des cas où l'utilisation des ressources parallèles n'est pas optimisée.

L'algorithme de hachage pourrait utiliser des instructions matérielles (`SSE` par exemple) pour produire un résultat plus rapidement, cette piste est à investiguer. Le code est très flexible au niveau du choix de l'algorithme, il serait donc assez simple de faire des expériences.

Enfin, il serait possible et utile d'étendre le `latency-tracker` au suivi de latence en espace utilisateur. Ainsi, les utilisateurs pourraient simplement demander au système de gérer le lien entre les événements de début et de fin et réagir au besoin. L'intégration complète avec le traceur `LTTng` est une piste qui serait intéressante à étudier.

RÉFÉRENCES

"Amazon cloudwatch", http ://aws.amazon.com/cloudwatch/, accessed : 2012-09-30.

"The latency-tracker repository", https ://github.com/efficios/latency-tracker, accessed : 2015-12-16.

"The lttng-analyses repository", https ://github.com/lttng/lttng-analyses, accessed : 2015-12-16.

"Google rocksteady", https ://code.google.com/p/rocksteady/, accessed : 2012-09-30.

"Sysdig official website", https ://sysdig.com, 2014, accessed : 2016-01-12.

"Locating system problems using dynamic instrumentation".

"The tracecompass website", http ://tracecompass.org/, accessed : 2015-12-16.

"Distributed systems tracing with zipkin", `http://engineering.twitter.com/2012/06/distributed-systems-tracing-with-zipkin.html`, accessed : 2012-09-30.

A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, et J. Widom, "Stream : The stanford data stream management system", *Book chapter*, 2004.

W. Betz, M. Cereia, et I. C. Bertolotti, "Experimental evaluation of the linux rt patch for real-time applications", dans *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*. IEEE, 2009, pp. 1–4.

D. P. Bovet et M. Cesati, *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

R. Buyya, "Parmon : a portable and scalable monitoring system for clusters", *Software-Practice and Experience*, vol. 30, no. 7, pp. 723–740, 2000.

J. M. Calandrino, H. Leontyev, A. Block, U. Devi, et J. H. Anderson, "Litmusˆ rt : A testbed for empirically comparing real-time multiprocessor schedulers", dans *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 111–126.

E. Cecchet, M. Natu, V. Sadaphal, P. Shenoy, et H. Vin, "Performance debugging in data centers : Doing more with less", dans *Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International*, 2009, pp. 1–9. DOI : `10.1109/COMSNETS.2009.4808877`

A. Chandrasekar, K. Chandrasekar, M. Mahadevan, et P. Varalakshmi, “Qos monitoring and dynamic trust establishment in the cloud”, dans *Advances in Grid and Pervasive Computing*, série Lecture Notes in Computer Science, R. Li, J. Cao, et J. Bourgeois, éds. Springer Berlin Heidelberg, 2012, vol. 7296, pp. 289–301. DOI : 10.1007/978-3-642-30767-6_25. En ligne : http://dx.doi.org/10.1007/978-3-642-30767-6_25

B. Claise, “Cisco Systems NetFlow Services Export Version 9”, RFC 3954 (Informational), Internet Engineering Task Force, Oct. 2004. En ligne : http://www.ietf.org/rfc/rfc3954.txt

——, “Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information”, RFC 5101 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. En ligne : http://www.ietf.org/rfc/rfc5101.txt

S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero-Merino, L. M. Vaquero, K. Nagin, et B. Rochwerger, “Monitoring Service Clouds in the Future Internet”, 2010.

J. Corbet, “Extending extended bpf”, *Linux Weekly News*, 2014.

M. Côté et M. R. Dagenais, “Problem detection in real-time systems by trace analysis”, *Advances in Computer Engineering*, vol. 2016, 2016.

M. R. Dagenais, K. Yaghmour, C. Levert, et M. Pourzandi, “Software performance analysis”, *arXiv preprint cs/0507073*, 2005.

D. Davenport, G. Delp, J. Lynch, K. Plotz, et P. Leichty, “Apparatus and method for segmentation and time synchronization of the transmission of a multiple program multimedia data stream”, Juil. 29 1997, uS Patent 5,652,749. En ligne : http://www.google.com/patents/US5652749

J. Desfossez, M. Desnoyers, et M. R. Dagenais, “Runtime latency detection and analysis”, *Software : Practice and Experience*, 2016. DOI : 10.1002/spe.2389

M. Desnoyers et M. R. Dagenais, “The lttng tracer : a low impact performance and behavior monitor for gnu/linux”, dans *Proceedings of Ottawa Linux Symposium 2006*, 2006, pp. 209–223.

M. Desnoyers, “Common trace format (ctf) specification”, http ://www.efficios.com/ctf, 2011.

——, “Low-impact operating system tracing”, Thèse de doctorat, École Polytechnique de Montréal, 2009.

——, “Low-impact operating system tracing”, Thèse de doctorat, ECOLE POLYTECHNIQUE DE MONTREAL, 2009.

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

M. Dhingra, J. Lakshmi, et S. K. Nandy, “Resource usage monitoring in clouds”, dans *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, série GRID ’12. Washington, DC, USA : IEEE Computer Society, 2012, pp. 184–191. DOI : 10.1109/Grid.2012.10. En ligne : http://dx.doi.org/10.1109/Grid.2012.10

S.-S. Dragos, M.-F. Vaida, L.-A. Suta, M.-C. Ureche, et A. Voina, “Syncy : A software engine for data stream event synchronization”, dans *System Theory, Control and Computing (ICSTCC), 2012 16th International Conference on*. IEEE, 2012, pp. 1–6.

J. Edge, “A look at ftrace”, http ://lwn.net/Articles/322666/, 2009.

——, “Perfcounters added to the mainline”, http ://lwn.net/Articles/339361/, 2009.

L. Fu et R. Schwebel. Real-time linux wiki. rt preempt howto. https ://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO. Accessed : 2016-01-06.

Y. Fu, Y. Wang, et E. Biersack, “A general scalable and accurate decentralized level monitoring method for large-scale dynamic service provision in hybrid clouds”, *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1235 – 1253, 2013, <ce :title>Special section : Hybrid Cloud Computing</ce :title>. DOI : 10.1016/j.future.2012.11.001. En ligne : http://www.sciencedirect.com/science/article/pii/S0167739X12002075

F. Giraldeau et M. Dagenais, “Wait analysis of distributed systems using kernel tracing”, *IEEE Transactions on Parallel and Distributed Systems*, 2015.

T. Gleixner, “Cyclictest”, accessed : 2015-12-16.

A. Gohad, P. Rao, N. Narendra, et K. Ponnalagu, “Formation of dynamic collaborative cloud links based on provider capability cost and resource health monitoring”, dans *Cloud*

Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on, 2012, pp. 1–6. DOI : 10.1109/CCEM.2012.6354609

M. Gorawski et A. Chrószcz, "Synchronization modeling in stream processing." dans *ADBIS (2)*, série Advances in Intelligent Systems and Computing, T. Morzy, T. Härder, et R. Wrembel, éds., vol. 186. Springer, 2012, pp. 91–102. En ligne : http://dblp.uni-trier.de/db/conf/adbis/adbis2012-2.html#GorawskiC12

B. Gregg, *Systems Performance : Enterprise and the Cloud*. Pearson Education, 2013.

D. Gunter et B. Tierney, "Netlogger : a toolkit for distributed system performance tuning and debugging", dans *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, 2003, pp. 97–100. DOI : 10.1109/INM.2003.1194164

D. Gupta, R. Gardner, et L. Cherkasova, "Xenmon : Qos monitoring and performance profiling tool", Rapp. tech., 2005.

G. Heward, I. Muller, J. Han, J.-G. Schneider, et S. Versteeg, "Assessing the performance impact of service monitoring", dans *Software Engineering Conference (ASWEC), 2010 21st Australian*, 2010, pp. 192–201. DOI : 10.1109/ASWEC.2010.28

D. Hildebrand, "An architectural overview of qnx." dans *USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992, pp. 113–126.

M. Jabbarifar, M. Dagenais, R. Roy, et A. S. Sendi, "Optimum off-line trace synchronization of computer clusters", *Journal of Physics : Conference Series*, vol. 341, p. 012029, 2012.

V. Jacobson, C. Leres, et S. McCanne, "The tcpdump manual page", *Lawrence Berkeley Laboratory, Berkeley, CA*, 1989.

M. Kang, D.-I. Kang, M. Yun, G.-L. Park, et J. Lee, "Design for run-time monitor on cloud computing", dans *Security-Enriched Urban Computing and Smart Grid*, série Communications in Computer and Information Science, T.-h. Kim, A. Stoica, et R.-S. Chang, éds. Springer Berlin Heidelberg, 2010, vol. 78, pp. 279–287. DOI : 10.1007/978-3-642-16444-6_36. En ligne : http://dx.doi.org/10.1007/978-3-642-16444-6_36

R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, et A. Vahdat, "Chronos : predictable low latency for data center applications", dans *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 9.

G. Katsaros, G. Kousiouris, S. V. Gogouvitis, D. Kyriazis, A. Menychtas, et T. Varvarigou, “A self-adaptive hierarchical monitoring mechanism for clouds”, *Journal of Systems and Software*, vol. 85, no. 5, pp. 1029 – 1041, 2012. DOI : 10.1016/j.jss.2011.11.1043. En ligne : http://www.sciencedirect.com/science/article/pii/S0164121211002998

A. Kertesz, G. Kecskemeti, A. Marosi, M. Oriol, X. Franch, et J. Marco, “Integrated monitoring approach for seamless service provisioning in federated clouds”, dans *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, 2012, pp. 567–574. DOI : 10.1109/PDP.2012.25

A. Khandual, “Performance monitoring in linux kvm cloud environment”, dans *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, 2012, pp. 1–6. DOI : 10.1109/CCEM.2012.6354620

B. Khargharia, H. Luo, Y. Al-Nashif, et S. Hariri, “Appflow : Autonomic performance-per-watt management of large-scale data centers”, dans *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int’l Conference on Int’l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 103–111. DOI : 10.1109/GreenCom-CPSCom.2010.103

B. Kim, Y. Yoo, C. Yoo, et Y. Ko, “Design and implementation of resource management tool for virtual machine”, dans *Computer Applications for Communication, Networking, and Digital Contents*, série Communications in Computer and Information Science, T.-h. Kim, D.-s. Ko, T. Vasilakos, A. Stoica, et J. Abawajy, éds. Springer Berlin Heidelberg, 2012, vol. 350, pp. 17–24. DOI : 10.1007/978-3-642-35594-3_3. En ligne : http://dx.doi.org/10.1007/978-3-642-35594-3_3

K.-D. Kwon, “Analyzing real-time performance problems in embedded linux”, Thèse de doctorat, WASEDA UNIVERSITY, 2010.

K.-D. Kwon, M. Sugaya, et T. Nakajima, “Ktas : analysis of timer latency for embedded linux kernel”, *International Journal of Advanced Science and Technology*, vol. 19, pp. 59–70, 2010.

P. A. Laplante, *REAL-TIIME Systems design and analysis.* IEEE, 1993.

S. Larsen, P. Sarangam, R. Huggahalli, et S. Kulkarni, “Architectural breakdown of end-to-end latency in a tcp/ip network”, *International journal of parallel programming*, vol. 37, no. 6, pp. 556–571, 2009.

J. Leverich et C. Kozyrakis, “Reconciling high server utilization and sub-millisecond quality-of-service”, dans *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 4.

M. Lindner, F. G. Marquez, C. Chapman, S. Clayman, D. Henriksson, et E. Elmroth, “The cloud supply chain : A framework for information, monitoring, accounting and billing”, dans *2nd International ICST Conference on Cloud Computing (CloudComp 2010)*, 2011.

H. Liu, “A measurement study of server utilization in public clouds”, dans *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, Dec 2011, pp. 435–442. DOI : 10.1109/DASC.2011.87

D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, et C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads”, dans *Proceeding of the 41st annual international symposium on Computer architecture*. IEEE Press, 2014, pp. 301–312.

D. Luu, “The nyquist theorem and limitations of sampling profilers today, with glimpses of tracing tools from the future”, http ://danluu.com/perf-tracing/, 2016.

M. L. Massie, B. N. Chun, et D. E. Culler, “The ganglia distributed monitoring system : design, implementation, and experience”, *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

D. Mills, J. Martin, J. Burbank, et W. Kasch, “Network Time Protocol Version 4 : Protocol and Algorithms Specification”, RFC 5905 (Proposed Standard), Internet Engineering Task Force, Juin 2010. En ligne : http://www.ietf.org/rfc/rfc5905.txt

P. J. Mucci, S. Browne, C. Deane, et G. Ho, “Papi : A portable interface to hardware performance counters”, dans *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

A. Muñoz, J. Gonzalez, et A. Maña, “A performance-oriented monitoring system for security properties in cloud computing applications”, *The Computer Journal*, vol. 55, no. 8, pp. 979–994, 2012. DOI : 10.1093/comjnl/bxs042. En ligne : http://comjnl.oxfordjournals.org/content/55/8/979.abstract

R. Nikolaev et G. Back, “Perfctr-xen : a framework for performance counter virtualization”, *SIGPLAN Not.*, vol. 46, no. 7, pp. 15–26, Mars 2011. DOI : 10.1145/2007477.1952687. En ligne : http://doi.acm.org/10.1145/2007477.1952687

J. Quittek, S. Bryant, B. Claise, P. Aitken, et J. Meyer, “Information Model for IP Flow Information Export”, RFC 5102 (Proposed Standard), Internet Engineering Task Force, Jan. 2008, updated by RFC 6313. En ligne : http://www.ietf.org/rfc/rfc5102.txt

C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, et M. A. Kozuch, “Towards understanding heterogeneous clouds at scale : Google trace analysis”, *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, p. 84, 2012.

S. Rostedt. (2005, April) An introduction to kprobes. http ://lwn.net/Articles/132196/.

——, “Finding origins of latencies using ftrace”, *Proc. RT Linux WS*, 2009.

J. Rothschild, “High performance at massive scale-lessons learned at facebook”, dans *Seminar at UCSD*, 2009.

R. Rybaniec et P. Z. Wieczorekb, “Measuring and minimizing interrupt latency in linux-based embedded systems”, dans *Proc. of SPIE Vol*, vol. 8454, 2012, pp. 84 540Y–1.

H. Schulzrinne, A. Rao, et R. Lanphier, “Real Time Streaming Protocol (RTSP)”, RFC 2326 (Proposed Standard), Internet Engineering Task Force, Avr. 1998. En ligne : http://www.ietf.org/rfc/rfc2326.txt

H. Schulzrinne, S. Casner, R. Frederick, et V. Jacobson, “RTP : A Transport Protocol for Real-Time Applications”, RFC 3550 (INTERNET STANDARD), Internet Engineering Task Force, Juil. 2003, updated by RFCs 5506, 5761, 6051, 6222, 7022, 7164. En ligne : http://www.ietf.org/rfc/rfc3550.txt

Z. Shao, L. He, Z. Lu, et H. Jin, “Vsa : An offline scheduling analyzer for xen virtual machine monitor”, *Future Generation Computer Systems*, no. 0, pp. –, 2012. DOI : 10.1016/j.future.2012.12.004. En ligne : http://www.sciencedirect.com/science/article/pii/S0167739X12002245

B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, et C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure”, Google, Inc., Rapp. tech., 2010. En ligne : http://research.google.com/archive/papers/dapper-2010-1.pdf

H. Stokking, M. Van Deventer, O. Niamut, F. Walraven, et R. Mekuria, “Iptv interdestination synchronization : A network-based approach”, *ICIN2010, Nov*, 2010.

M.-S. Su, K. Thulasiraman, et A. Das, “A scalable on-line multilevel distributed network fault detection/monitoring system based on the snmp protocol”, dans *Global Telecommunications Conference, 2002. GLOBECOM ’02. IEEE*, vol. 2, 2002, pp. 1960–1964 vol.2. DOI : 10.1109/GLOCOM.2002.1188542

M. Taufer et T. Stricker, “A performance monitor based on virtual global time for clusters of pcs”, dans *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, 2003, pp. 64–72. DOI : 10.1109/CLUSTR.2003.1253300

B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, et D. Gunter, “The netlogger methodology for high performance distributed systems performance analysis”, dans *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, 1998, pp. 260–267. DOI : 10.1109/HPDC.1998.709980

B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, et R. Wolski, “A grid monitoring architecture”, 2002.

J. Triplett, P. E. McKenney, et J. Walpole, “Resizable, scalable, concurrent hash tables via relativistic programming.” dans *USENIX Annual Technical Conference*, 2011, p. 11.

P. Uthayopas, S. Phaisithbenchapol, et K. Chongbarirux, “Building a resources monitoring system for smile beowulf cluster”, dans *Proceedings of HPC Asia98 Conference, Singapore*, 1998.

G. Xiang, H. Jin, et D. Zou, “A comprehensive monitoring framework for virtual computing environment”, dans *Information Networking (ICOIN), 2012 International Conference on*, 2012, pp. 551–556. DOI : 10.1109/ICOIN.2012.6164438

J. Yang, D. B. Minturn, et F. Hady, “When poll is better than interrupt.” dans *FAST*, vol. 12, 2012, pp. 3–3.

W. Yang, J. Zhu, T. Zhou, et Q. Wang, “Research on the communication monitoring technology for host based on virtual machine monitor”, dans *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, 2012, pp. 972–974. DOI : 10.1109/MINES.2012.183

N. Yigitbasi, A. Iosup, D. Epema, et S. Ostermann, “C-meter : A framework for performance analysis of computing clouds”, dans *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, série CCGRID ’09. Washington, DC,

USA : IEEE Computer Society, 2009, pp. 472–477. DOI : 10.1109/CCGRID.2009.40. En ligne : http://dx.doi.org/10.1109/CCGRID.2009.40

ANNEXE A Corrélation entre latence et défauts de cache

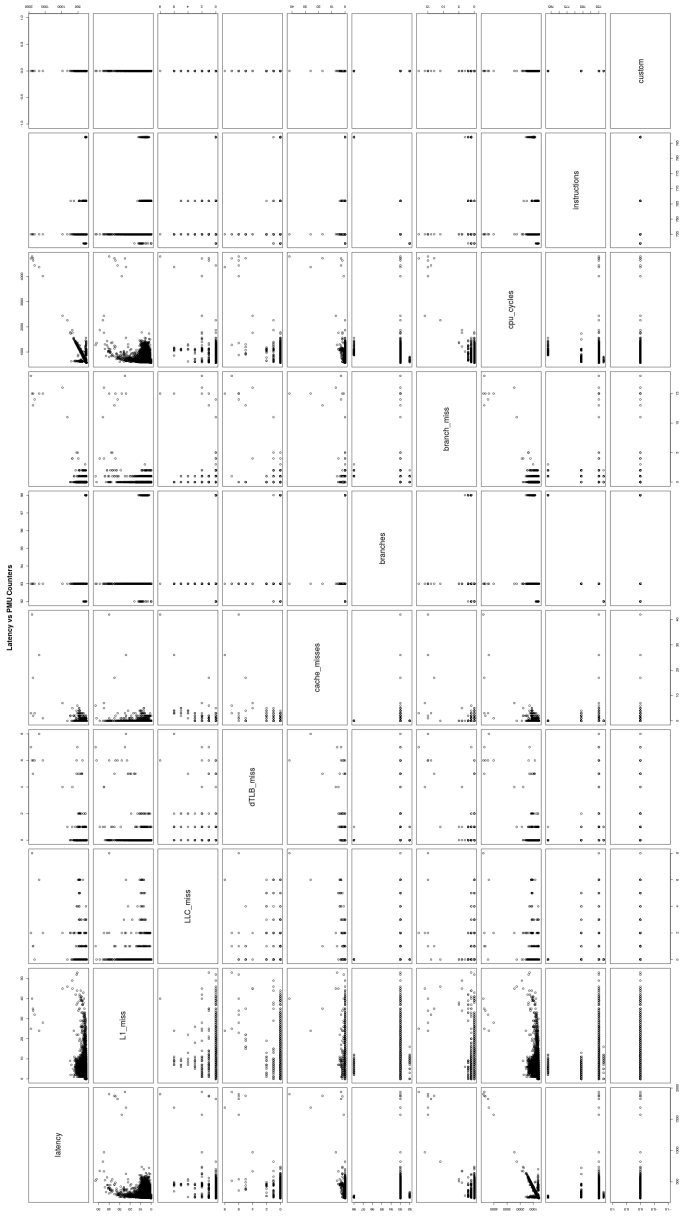

Figure A.1 Vue complète de la matrice des différentes métriques système