Titre: Title:	Traçage de systèmes linux multi-coeurs en temps réel
Auteur: Author:	Raphaël Beamonte
Date:	2013
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Beamonte, R. (2013). Traçage de systèmes linux multi-coeurs en temps réel [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1173/

UNIVERSITÉ DE MONTRÉAL

TRAÇAGE DE SYSTÈMES LINUX MULTI-CŒURS EN TEMPS RÉEL

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

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE DE SYSTÈMES LINUX MULTI-CŒURS EN TEMPS RÉEL

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

M. LANGLOIS J.M. Pierre, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. BOYER François-Raymond, Ph.D., membre

« Je passe tout mon temps à comprendre le temps. »

Alain Bosquet (Avoir empêche d'être – 1994)
Critique littéraire et poète français

À ma mère dont le soutien m'a permis
de prendre le temps de chercher
à réduire le temps.

REMERCIEMENTS

Je tiens premièrement à remercier mon directeur de recherche, Michel Dagenais, pour le support qu'il m'a apporté tout au long de ma maîtrise, la confiance qu'il m'a accordée dès le premier jour et qu'il a sans cesse renouvelée. Son intérêt profond pour la recherche de la connaissance et de la compréhension a été un vecteur d'envie d'aller toujours plus loin.

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

J'aimerais aussi adresser des remerciements aux personnes qui m'ont permis de pouvoir suivre ce programme d'études, que ce soit par le fait d'avoir envoyé un premier courriel pour mettre en avant mon intérêt pour ces études comme l'a fait Florent Retraint, par les lettres de recommandations écrites pour mon admission au programme de maîtrise en sciences appliquées par Timothée Toury, Guillaume Doyen, Michel Doussot et Giorgio Lanzarone, ou encore par les démarches entreprises à mon arrivée par Jean Dansereau et Julie Defretin pour concilier la fin de mon diplôme d'ingénieur et le début de ma maîtrise.

Les collègues du laboratoire DORSAL ne sont pas en reste non plus, puisque c'est grâce à eux que l'ambiance de travail est toujours restée dans le haut de l'échelle et que le partage des connaissances a toujours été présent. Un merci tout particulier aux associés de recherche passé et présents, Matthew, Yannick et Geneviève pour l'aide précieuse qu'ils m'ont apportée et le support moral dans les moments plus « chargés ».

Enfin, j'aimerais remercier ceux de mes proches, parents et amis, qui ont toujours été confiants dans mes compétences et m'ont poussé sans cesse à me dépasser tout en me soutenant et surtout en me supportant – ce qui est certainement le plus difficile !

RÉSUMÉ

Le traçage est une méthode d'analyse de plus en plus populaire compte tenu de la précision des données qu'il permet de réunir concernant les activités d'un système. Nombre de systèmes demandent cependant aujourd'hui de fonctionner dans des conditions restreintes : c'est le cas des systèmes temps réel. Ces systèmes reposent sur le respect d'échéances dans le traitement de données. Autrement dit, une donnée exacte n'aura de valeur dans un système de ce type qu'à condition qu'elle soit obtenue dans les délais, sans quoi le système sera considéré défaillant. Ces conditions excluent par conséquent tout outil d'analyse impactant de manière significative la latence et les performances du système. Le traçage repose sur l'ajout de points de trace au sein de l'application ou du système d'exploitation tracé. Ces derniers vont générer des évènements lorsqu'ils sont atteints durant l'exécution et ainsi permettre de suivre l'état d'un système au fur et à mesure de son exécution. Cependant, la génération de ces données implique une modification dans le déroulement normal de l'application ou du système tracé, et par conséquent un impact. Il devient alors important de pouvoir quantifier cet impact de manière à savoir sous quelles conditions une application temps réel pourra ou ne pourra pas être tracée, autrement dit, de savoir si l'ajout de ce délai de traitement rendra le système défaillant, et par conséquent inadéquat au traçage.

L'objectif de cette recherche est de montrer qu'il est tout à fait possible d'utiliser le traçage pour analyser de telles applications, et ce avec un impact en terme de latence que nous souhaitons quantifier pour le pire cas.

Pour ce faire, nous allons définir un environnement de travail temps réel en étudiant les divers systèmes et configurations spéciales mis à disposition. Ensuite nous verrons quels sont les outils permettant de valider la qualité de notre environnement de travail et quel est leur fonctionnement. Nous comparerons par la suite les différents outils mettant à disposition des traceurs, et lesquels parmi eux nous permettent de corréler des traces du noyau et de l'espace utilisateur.

Une fois l'environnement défini et vérifié et l'outil de traçage choisi, notre méthode expérimentale reposera sur un étalonnage du système temps réel permettant par la suite d'évaluer l'impact du traceur sous différents angles à l'aide de la création de différent scénarios d'analyse. Cette méthode expérimentale prendra en compte la latence ajoutée sur le système par l'instrumentation, le traçage mais aussi la qualité des traces obtenues. Ces éléments seront les marqueurs de qualité du traçage dans des conditions temps réel.

L'hypothèse servant de point de départ de ce travail est qu'en isolant le fonctionnement du traceur de celui de l'application temps réel tracée, l'impact de l'outil de traçage sur l'application pourra être radicalement réduit.

Les résultats de ce travail sont la découverte et l'intégration aux outils existants d'un modèle pour supprimer les communications entre l'application tracée et le traceur durant la période active de traçage, permettant ainsi de n'ajouter qu'une interférence minimale sur l'exécution normale de l'application, permettant ainsi de rendre le traçage apte à fonctionner dans nombre de conditions temps réel, du moment que le délai maximal qu'il ajoute entre dans l'échéance de celle-ci. De plus, l'outil `npt` créé comme outil d'étalonnage et d'analyse de l'impact a évolué au fur et à mesure de nos recherches et est maintenant un logiciel disposant de différentes options pour simuler de multiples scénarios d'exécution d'une application temps réel. Enfin, la méthodologie d'analyse de l'impact du traçage sur le système temps réel constitue elle aussi un des résultats de cette étude.

Le résultat final est que l'utilisation du traçage pour une application temps réel s'exécutant dans l'espace utilisateur est viable pour la plupart des systèmes temps réel avec une latence ajoutée de seulement quelques microsecondes.

ABSTRACT

Tracing is an analysis method increasingly popular. It provides a lot of information of high precision about a system or application. However, many systems need to operate under restricted conditions. This is the case of real-time systems. These systems are based on meeting deadlines in data processing. In other words, accurate data will only have a value if we can poduce it in a timely fashion, otherwise we will consider the system as defective. These conditions therefore exclude any analysis tool which could impact significantly the system latency or performance. Tracing is based on the addition of tracepoints directly in the application or operating system source code. There tracepoints will generate events when they are reached as part of the execution, and allow to monitor the system or application state during its run. Still, the generation of this data involves a change in the normal course of the traced system, hence an impact. It then becomes important to quantify this impact in order to know the conditions for tracing a real-time application, in other words, to know whether the added latency will invalidate the real-time condition, hence force to consider tracing as inedequate or not.

The objective of this research is to prove that it is possible to use tracing to analyze such applications, thus quantifying the maximum impact in terms of latency.

In order to do that, we will define a real-time work environment by studying the various systems and special configurations available, then we will see what tools allow to validate the quality of our work environment and how they work. We will then compare the various tracing tools and identify which allow us to correlate the kernel and userspace traces.

Once we have setup and verified our real-time environment and selected our tracing tool, our experimental method will be based on real-time system calibration to subsequently assess the impact of the tracer from different use cases, using different analysis scenarios. This experimental method will take into account the latencies added by the instrumentation and the tracing of the system, but also the quality of the generated traces. These elements will be considered as quality markers of real-time tracing.

The hypothesis serving as starting point for this work is that, by isolating the work of the tracer from that of the real-time application, the impact of the tracing tool on the application can radically be reduced.

The results of this work are the discovery and integration in existing tools of a model to remove the direct connections, between the traced application and the tracer, while the tracing operations are active. This allows to only add a minimal interference to the normal execution of the application, thereby making tracing able to operate multiple real-time cases,

as long as the maximum latency added can be supported by the traced application. In addition, the `npt` tool created as the calibation and impact analysis tool has evolved during our research and now offers options allowing to simulate multiple execution scenarios of real-time applications. Finally, the real-time tracing impact analysis methodology we developed is also one of the results of this study.

The final result is that using tracing to analyze real-time applications running in userspace is a valid and good option for most real-time systems, adding latencies in the low microsecond range.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BIOS	Basic Input/Output System
CLI	CLear Interrupts
CTF	Common Trace Format
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
GNU	GNU is Not Unix
HRT	Hard Real-Time
IRQ	Interrupt ReQuest
LTTng	Linux Trace Toolkit next generation
NMI	Non-Maskable Interrupt
npt	Non-Preempt Test
RAM	Random Access Memory
RCU	Read-Copy Update
RTOS	Real-Time Operating System
SMI	System Management Interrupt
SMP	Symmetric MultiProcessor
SRT	Soft Real-Time
STI	SeT Interrupts
TSC	TimeStamp Counter
URCU	Userspace Read-Copy Update
UST	UserSpace Tracer
VDSO	Virtual Dynamically linked Shared Objects

Chapitre 1

INTRODUCTION

Les systèmes temps réel ont toujours été plus difficiles à surveiller et déboguer à cause des contraintes temps réel qui excluent tout outil impactant de manière significative la latence et les performances du système. Compte tenu de ces restrictions, le traçage est l'outil le plus fiable pour l'étude de ces systèmes. Ces dernières années, le comportement en temps réel des systèmes GNU/Linux s'est grandement amélioré et, avec une utilisation correcte de l'isolation des processeurs sur des systèmes multi-cœurs, il est maintenant possible d'atteindre des latences du domaine de quelques microsecondes. Dans ce contexte, les traceurs doivent s'assurer que leurs ajouts de latence sont dans cette gamme, prévisibles et parfaitement adaptés à un fonctionnement multi-cœurs.

1.1 Définitions et concepts de base

Dans cette section, nous présenterons et définirons différents concepts qui seront utilisés dans ce mémoire.

1.1.1 Le temps réel

Système informatique temps réel

Un système temps réel est un système informatique prenant en compte les contraintes temporelles. Ces contraintes sont alors aussi importantes que l'exactitude des résultats obtenus, et un résultat sera invalidé s'il n'est pas obtenu dans les délais. On dit alors qu'un système temps réel doit respecter des déterminismes logique – exactitude du résultat – et temporel – résultat obtenu dans le temps imparti.

On distingue deux types de systèmes temps réel selon l'importance des contraintes temporelles à remplir :

– Les systèmes temps réel à contraintes souples (SRT), pour lesquels les évènements traités en dehors des délais ou simplement perdus n'ont pas de conséquence catastrophique sur la bonne marche du système. Par exemple, les systèmes multimédia – notamment la diffusion en flux – pour lesquels il est possible de perdre quelques secondes de vidéo sans que le fonctionnement général du système ne soit mis en péril ;

– Les systèmes temps réel à contraintes strictes (HRT), qui correspondent à des systèmes devant répondre à des sollicitations ou évènements, internes ou externes, dans un temps maximum nécessitant alors un déterminisme temporel fort. Par exemple, les systèmes de contrôle des centrales nucléaires ou encore les systèmes embarqués en aéronautique entrent dans cette catégorie.

Application temps réel

Une application temps réel est une application pour laquelle le respect de contraintes temporelles lors d'un traitement est aussi important que le résultat du traitement. Elle repose généralement sur un déterminisme à deux critères : logique, c'est à dire que les mêmes données traitées doivent donner le même résultat, et temporel, c'est à dire qu'une tâche donnée doit absolument être réalisée dans les délais impartis, autrement dit respecter l'échéance.

1.1.2 Fonctionnement du système d'exploitation

Préemption

Dans le domaine informatique, la préemption est l'acte d'interrompre temporairement une tâche en cours d'exécution sans nécessiter sa coopération, dans le but de la laisser terminer son exécution plus tard. Ce genre d'interruption est en général effectué pour donner du temps d'exécution à une tâche de plus haute priorité et sera effectué par l'ordonnanceur du système s'il s'agit d'un ordonnanceur préemptif. On appelle ceci un changement de contexte.

Interruption

En informatique, une interruption est un signal envoyé au processeur par le matériel ou un logiciel indiquant un évènement nécessitant une réaction immédiate. Une interruption informe le processeur d'arrêter l'exécution de la tâche en cours afin de s'occuper d'une condition de haute-priorité. Celui-ci interrompra alors temporairement ses activités tout en sauvegardant leurs états et exécutera un programme simple, appelé gestionnaire d'interruption, dont le but est de gérer ce type d'évènements. Une fois que le gestionnaire d'interruption a fini son exécution, le processeur restaurera l'exécution des programmes interrompus précédemment. Chaque interruption dispose de son propre gestionnaire d'interruption.

On différencie les interruptions matérielles, envoyées directement au processeur de manière électronique par un périphérique externe ou un composant matériel du système, des interruptions logicielles, qui peuvent être causées par une condition exceptionnelle dans le processeur lui-même ou encore une simple instruction d'interruption qui provoquera l'interruption lors de son exécution. Le nombre d'interruptions matérielles est limité par le nombre

de lignes de requêtes d'interruption (IRQ) du processeur. Il peut cependant y avoir des centaines d'interruptions logicielles, ces dernières n'ayant pas de limite matérielle.

Isolation des processeurs

L'isolation des processeurs est la pratique permettant dans un système multiprocesseur de séparer le travail d'un processeur du travail des autres. C'est une méthode principalement utilisée pour isoler un processeur ayant pour rôle de travailler sur des tâches temps réel de haute priorité. En isolant un processeur on s'assure que ses ressources sont réservées pour des tâches hautement prioritaires. On crée alors un environnement dans lequel le déterminisme nécessaire pour supporter des applications temps-réel est atteint. Dans cet environnement, on peut ainsi garantir un temps de réponse rapide aux interruptions externes.

1.1.3 Le traçage

Le traçage consiste à enregistrer des évènements durant l'exécution du système tracé. Ces évènements peuvent se situer dans le noyau du système d'exploitation ou au sein d'une application de l'espace utilisateur. Il permet, tout en minimisant l'impact qu'il a sur le système, d'obtenir des informations sur son comportement.

Évènement

Dans le cadre d'une trace, un évènement est une action survenant à un instant donné dans le système (dans le noyau) ou dans une application (dans l'espace utilisateur) qui est enregistrée. Un évènement est lui-même ponctuel, sa durée dans le temps est nulle. Chaque évènement est défini par la date à laquelle il est survenu, son type (par exemple un « appel système » dans le noyau) et des informations additionnelles. Ces informations additionnelles ne sont pas les mêmes selon l'évènement concerné. Les évènements d'une trace sont générés par des points de trace, qu'ils soient statiques ou dynamiques.

Point de trace

Un point de trace est moment dans le déroulement de l'exécution du noyau du système ou d'une application auquel un évènement sera généré. C'est lorsqu'on place ce point de trace que l'on pourra lui passer les informations additionnelles de l'évènement. Il existe deux types de points de trace, les points de trace statiques (qui existent avant l'exécution) et les points de trace dynamiques (qui vont être créés lors de l'exécution). L'étude que nous faisons ici se concentre sur des points de trace statiques, les points de trace dynamiques ayant de par leur définition un impact plus grand sur l'exécution de l'application.

1.2 Éléments de la problématique

Les systèmes temps réel nécessitent une surveillance accrue de la latence étant donné qu'elle fait partie intégrante de l'exactitude du résultat. Ces dernières années, les évolutions du noyau Linux et de son correctif `PREEMPT_RT` ont permis d'améliorer grandement les performances en temps réel de ce système. GNU/Linux est devenu un système d'exploitation polyvalent capable d'exécuter des applications dans des conditions de temps réel à contraintes souples. À l'aide du correctif `PREEMPT_RT` et d'une isolation adéquate des processeurs, il est aussi possible d'atteindre les contraintes fixées par le temps réel strict.

Lorsqu'on exécute une application temps réel, elle travaillera la plupart du temps exclusivement en espace utilisateur de façon à ne pas dépendre des temps de réponse des applications ou systèmes externes. Dans ce type d'applications, le déterminisme des délais de réponse devient aussi important que la validité des résultats obtenus. On parle ici de respect des échéances. Sans ce respect, le travail de l'application est invalidé et l'application elle-même sera jugée comme étant défaillante. Aussi, dans nombre de cas, ajouter un délai même déterministe dans l'exécution d'une application peut la faire agir différemment, par exemple en faisant disparaître un problème dont on souhaitait identifier la source, ou encore en noyant un pic de latence parmi d'autres latences ajoutées par l'analyse. Ajouter du traçage sur une application temps réel devient dès lors un défi.

En effet, pour fonctionner, un traceur doit s'insérer dans le fonctionnement régulier de l'application ou du système afin que celui-ci génère les évènements qui seront par la suite enregistrés dans la trace. Pour cela, des points de trace sont ajoutés aux endroits ciblés puis appelés lors de l'exécution de l'application. Ce sont lors de ces appels que les évènements sont créés. Ces points nécessitent au moins l'exécution d'une instruction, et ce même lorsque le traçage est inactif, permettant de déterminer si le traceur doit être appelé ou non. Selon la méthode utilisée, un point de trace inactif pourra ne prendre qu'un faible nombre de cycles. Utiliser le traçage pour permettre d'analyser des applications temps réel implique par conséquent le respect des contraintes temporelles nécessaires au fonctionnement de ces applications. Il est par conséquent nécessaire que la méthode utilisée pour l'ajout des points de trace n'ait pas ou peu d'impact de latence lorsque ces derniers sont inactifs, et une latence déterministe et la plus faible possible lors d'un traçage actif et de la création d'évènements. Bien que le traçage soit déjà ici un outil de choix face aux débogueurs qui nécessiteront l'utilisation de points d'interruption, arrêtant ainsi le déroulement de l'exécution de l'application, il est important que le traçage ajoute une latence raisonnable dans le pire des cas. Cette latence devra ainsi être de l'ordre de quelques microsecondes pour pouvoir être utilisable sur des systèmes basés sur des FPGA.

`LTTng-UST` est la composante de l'espace utilisateur du traceur `LTTng` et permet l'ajout de points de trace statiques dans une application. Son mode de fonctionnement permet de séparer la capture des évènements de leur écriture sur le disque en deux processus distincts pouvant donc s'exécuter sur deux processeurs différents dans un système multi-cœurs, et par conséquent de limiter l'impact du côté de l'application tracée. L'utilisation des points de trace statiques et la séparation de l'instrumentation et du consommateur d'évènements en font un outil de choix lorsque vient la question du traçage d'une application en temps réel.

1.3 Objectifs de recherche

Il est maintenant possible de préciser les objectifs de recherche :

1. Définir une méthodologie d'analyse de l'impact du traçage sur une application temps réel ;
2. Déterminer les capacités temps réel actuelles de `LTTng` ;
3. Identifier les problèmes potentiels et leurs causes (latences trop élevées dues aux changements de contexte ou aux inversions de priorité pouvant subvenir du fonctionnement du traceur) ;
4. Proposer les solutions pour palier à ces problèmes (limiter les changements de contexte, supprimer la situation provoquant les inversions de priorité) ;
5. Si les solutions sont efficaces, les faire intégrer au traceur ;
6. Comparer les performances de `LTTng` avec celles d'autres traceurs courants intégrés au noyau Linux.

L'ensemble de ces objectifs a pour but de répondre à l'objectif principal de permettre le traçage d'un système temps réel sans impact majeur sur la latence avec `LTTng`. L'objectif minimum est de réduire de moitié l'impact du traceur sur l'application temps réel.

1.4 Plan du mémoire

Au chapitre 2, nous presenterons une revue de littérature faisant le point sur l'état de l'art dans les domaines des systèmes temps réel ainsi que du traçage. Le chapitre 3 présentera la démarche de l'ensemble du travail de recherche en mettant de l'avant la cohérence, par rapport aux objectifs de la recherche, de l'article de revue du chapitre 4. Nous présenterons ensuite des résultats complémentaires à cet article au chapitre 5. Enfin, le chapitre 6 sera une discussion générale des travaux, qui précèdera directement la conclusion au chapitre 7, synthétisant les travaux et identifiant des possibilités d'évolutions futures.

Chapitre 2

REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art dans les domaines des systèmes temps réel et du traçage sous Linux. Nous étudions également les outils permettant de vérifier la qualité d'un système temps réel ainsi que les méthodes pour en améliorer le comportement.

2.1 Les systèmes temps réel

Outre la différence entre les systèmes SRT et HRT, il existe différents systèmes d'exploitation supportant le temps réel en utilisant différentes architectures spécifiques pour fonctionner de manière idéale dans des environnements temps réel [1]. L'architecture la plus exploitée encore aujourd'hui est l'approche micro-noyau. Cette architecture est basée sur l'ajout d'une couche d'abstraction – le micro-noyau – entre le matériel et le noyau Linux, comme présenté à la Figure 2.1. Le noyau Linux est ainsi exécuté en tâche préemptible de plus basse priorité du micro-noyau et sert à accueillir toutes les tâches ne nécessitant pas une priorité temps réel. Les tâches à priorité temps réel sont exécutées directement sur le micro-noyau. Si cette approche est efficace, c'est aussi parce que le micro-noyau a un rôle de gestion des interruptions systèmes. En effet, en interceptant les interruptions, le micro-noyau peut assurer que le noyau non-temps réel ne puisse préempter le fonctionnement des tâches temps réel. Le micro-noyau permet donc d'atteindre un support du temps réel à contraintes strictes.

Le premier système d'exploitation temps réel (RTOS) utilisant l'architecture micro-noyau et un noyau Linux est RTLinux [2] dont la première version date de 1996. Basé sur l'architecture micro-noyau, il mettait en avant le fait d'exécuter le noyau Linux en tant que tâche entièrement préemptible. Ce RTOS est cependant devenu un produit commercial lorsque ses créateurs ont fondé FSMLabs. En 2007, Wind River a racheté la solution et l'a rendue accessible au public sous le nom de *Wind River Real-Time Core for Wind River Linux*, dont la ligne de produit a finalement été discontinuée en août 2011.

Peu après la publication de RTLinux, un autre groupe de recherche a vu le fruit de son travail aboutir par la création de *Real-Time Application Interface* (RTAI) [3]. La différence principale entre RTAI et RTLinux se situe sur les changements effectués au noyau Linux. En effet, RTLinux modifie le comportement du noyau Linux pour accepter le micro-noyau en insérant ses modifications directement dans le code source de Linux. Ainsi, il devient

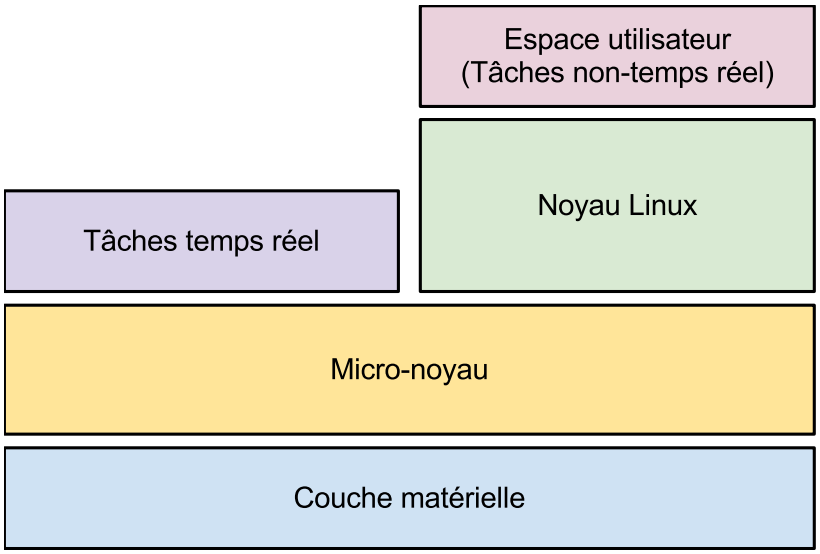
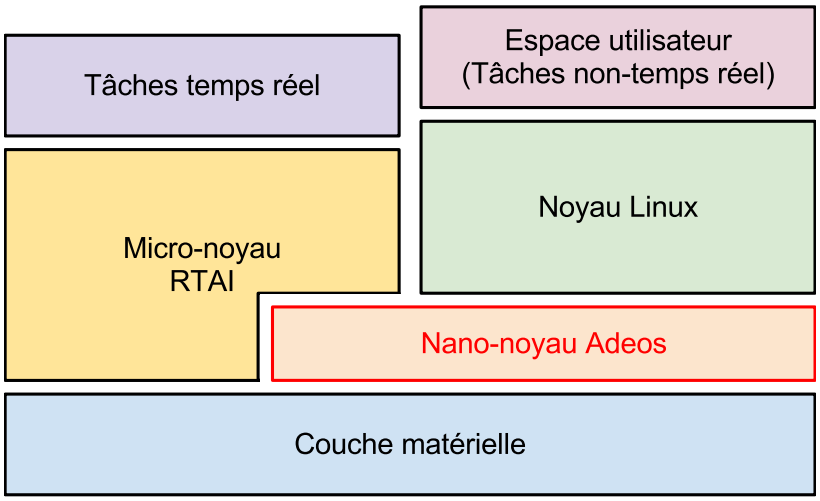


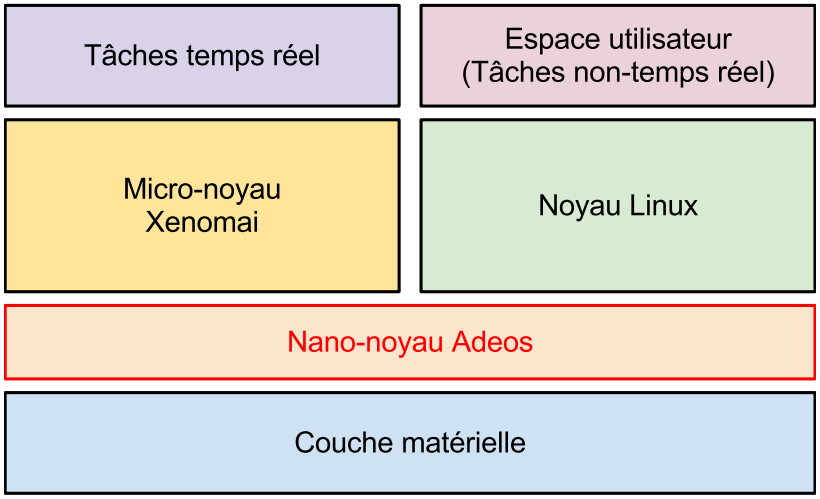
Figure 2.1 Représentation de l'architecture de fonctionnement du temps réel sur un système d'exploitation basé sur l'architecture micro-noyau

beaucoup plus difficile de maintenir l'état du noyau Linux au fur et à mesure de son évolution, de même que d'identifier, lors de l'apparition d'une erreur, si elle vient de la surcouche ajoutée par RTLinux ou de ses modifications du noyau Linux. RTAI de son côté a opté pour une couche d'abstraction matérielle composée d'une structure de pointeurs vers les adresses mémoires des gestionnaires d'interruptions ainsi que vers les fonctions permettant d'activer ou de désactiver ces interruptions. La création de cette couche d'abstraction permet de minimiser les retouches du code source de Linux puisqu'elle est implémentée en ne modifiant et n'ajoutant que quelques dizaines de lignes de code. De plus, l'utilisation d'une telle couche permet aisément de la désactiver pour isoler des bogues ou lorsque le temps réel n'est pas requis en changeant les valeurs des pointeurs pour les valeurs originales.

Alors qu'en 2001 les créateurs de RTLinux déposent un brevet concernant le comportement du micro-noyau, un nouveau projet voit le jour dans le but de le contourner, avec l'implication de RTAI. Il s'agit de Adeos (*Adaptive Domain Environment for Operating Systems*) [4]. Adeos vise à rendre le système temps réel modulable tout en permettant de séparer la couche d'abstraction matérielle du noyau. Il fournit un environnement flexible permettant de partager les ressources matérielles entre plusieurs systèmes d'exploitation ou plusieurs instances d'un même système d'exploitation, en contactant en séquence les différents systèmes déclarés de façon à ce que le système le plus prioritaire décide si oui ou non une interruption, par exemple, devra être propagée. L'approche micro-noyau adoptée par RTAI change alors quelque peu, le noyau temps réel interceptant toujours les interruptions mais reposant sur



(a) Implémentation de RTAI



(b) Implémentation de Xenomai

Figure 2.2 Architectures basées sur Adeos utilisées par RTAI et Xenomai

Adeos pour les propager au noyau Linux dans le cas où ces interruptions n'influent pas sur l'ordonnancement temps réel, comme le présente la Figure 2.2(a) [5].

Le développeur de ce « nano-noyau » est l'auteur et développeur principal de Xenomai, un autre RTOS à architecture micro-noyau [6]. Xenomai repose lui aussi sur Adeos mais laisse une plus grande place au nano-noyau dans l'architecture générale qu'il utilise, comme le montre la Figure 2.2(b). En effet, dans le cas de Xenomai, Adeos fonctionnera simplement de la façon dont il a été conçu : comme un hyperviseur pouvant supporter plusieurs systèmes d'exploitation en leur donnant à chacun une priorité. Tandis que RTAI est un système qui se concentre principalement sur l'obtention des latences les plus faibles possibles – d'où l'utilisation d'une architecture où RTAI reçoit directement les requêtes d'interruption –, Xenomai considère aussi l'extensibilité, la portabilité et la maintenabilité comme objectifs de développement.

Sur ce type d'architecture, cependant, la séparation du noyau supportant le temps réel et du noyau standard Linux pose par contre problème lorsqu'il s'agit de déboguer l'exécution d'une application composée de tâches temps réel et d'autres non-temps réel. En effet, un système d'exploitation basé sur cette architecture rend ces tâches indépendantes. De même, la communication entre ces tâches nécessite l'utilisation de tubes nommés (FIFO) ou de mémoire partagée.

Un travail est fait sur le noyau Linux pour le rendre de plus en plus apte à fonctionner avec des applications temps réel. Si l'on utilise un noyau Linux standard sans configuration spécifique, lorsqu'un processus de l'espace utilisateur fait un appel système il ne peut pas être préempté. Cela signifie que si un processus de haute priorité souhaite faire un appel système, mais qu'un processus de basse priorité est déjà en train d'en exécuter un, le processus de haute priorité devra attendre que le processeur ait fini d'exécuter l'appel système du processus de plus basse priorité. Depuis le noyau 2.6, l'option `CONFIG_PREEMPT` a été ajoutée pour permettre de palier à ce comportement. En utilisant cette option, si un processus de haute priorité apparaît, un processus de plus faible priorité sera préempté, y compris s'il est au milieu d'un appel système, permettant d'obtenir les performances requises par les systèmes temps réel à contraintes souples.

Il existe un correctif spécifique au temps réel pour le noyau Linux, appelé `PREEMPT_RT` [7]. Ce correctif permet de corriger le comportement du noyau pour celui d'un système temps réel à contraintes strictes, tout en limitant le nombre de modifications apportées. La majeure partie du noyau devient préeemptible grâce à quelques changements ciblés, notamment la réimplémentation de certaines primitives de verrouillage du noyau, la conversion des gestionnaires d'interruptions en fils d'exécution ou encore la mise en œuvre de l'héritage de priorité pour les mutex internes au noyau afin d'éviter les situations d'inversion de priorité [8]. En-

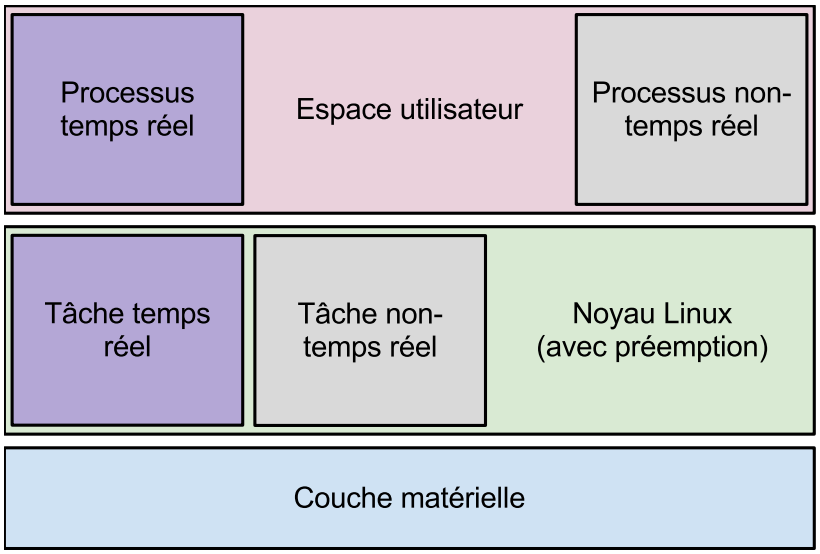


Figure 2.3 Représentation de l'architecture de fonctionnement du temps réel sur un système d'exploitation GNU/Linux

fin, nombre des fonctionnalités développées pour ce correctif ont été intégrées dans le noyau standard.

Lors de la création des premiers RTOS à architecture micro-noyau, il était reproché à Linux de ne pas préempter l'exécution d'une tâche pendant un appel système. On lui reprochait aussi qu'une tâche de haute priorité doive attendre après une tâche de plus faible priorité pour accéder à une ressource, tant que cette dernière ne l'aura pas relâchée, ou encore que son ordonnanceur pouvait donner du temps d'exécution à une application alors qu'une application de plus haute priorité était en attente. La gestion du matériel faite par le système pouvait aussi poser problème, réordonnançant par moments les tâches pour utiliser le matériel de manière plus efficace. Nombre de ces reproches ont été corrigés, soit dans le noyau standard, soit dans le correctif `PREEMPT_RT`, et ne sont désormais plus d'actualité.

De plus, utiliser GNU/Linux permet de combiner aisément des traces du noyau et des applications puisqu'il n'y a pas de séparation des applications temps réel et non-temps réel lors de leur exécution. Il est ainsi possible de tracer le noyau – unique – du système et les applications qui s'y exécutent et que nous souhaitons surveiller.

2.2 Isolation des processeurs

Pour obtenir le meilleur comportement temps réel, une approche consiste à isoler les processeurs. Chaque processeur pourra ainsi avoir ses tâches attribuées et ne pas interférer sur le fonctionnement des autres. Le processeur que l'on cherchera à isoler est celui qui sera

dédié aux activités associées à des tâches temps réel de haute priorité. Cette isolation permet de garantir une réponse rapide aux interruptions externes et de fournir un environnement d'exécution plus déterministe pour les tâches temps réel [9].

Bien qu'avec la technologie Hyperthread d'Intel il soit aujourd'hui possible d'isoler des processeurs sur des systèmes autres que ceux à multiprocesseurs symétriques (SMP), la structure même de cette technologie, basée sur la séparation d'un processeur physique en deux processeurs logiques, la rend déconseillée pour du travail temps réel. Les résultats obtenus dans [10] démontrent par ailleurs que l'utilisation de cette technologie décroit le déterminisme.

Sur un noyau Linux, plusieurs étapes sont nécessaires pour permettre une isolation correcte des processeurs. La première étape consiste à définir un ou plusieurs groupes auxquels nous pourrons assigner un processeur spécifique et, si jugé utile, de la mémoire. Des outils tels que les `cgroups` [11] et les `cpusets` [12] permettent un contrôle aisé des processeurs associés à une tâche. Ce contrôle est simplifié encore par l'utilisation d'outils tels que `cset` [13, 14]. Il est par la suite nécessaire de modifier certains comportements du système tels que la répartition automatique de charge des processeurs (`sched_load_balance`), la limite de temps d'exécution des applications temps réel (`sched_rt_runtime_us`) ou encore le délai après lequel on identifie qu'une tâche est bloquée (`hung_task_timeout_secs`). Enfin, une isolation complète des processeurs nécessite une meilleure gestion des interruptions. En d'autres mots, si l'on peut éviter d'avoir à traiter des interruptions sur le processeur chargé d'exécuter les tâches de haute priorité, il sera possible de gagner quelques microsecondes voire millisecondes. Le noyau Linux avec le correctif `PREEMPT_RT` dispose de processus légers du noyau chargés d'exécuter les interruptions (*threaded interrupts*). Ce processus est réveillé par le noyau lorsqu'une interruption est appelée, et est chargé de l'exécuter. Ce nouveau modèle permet de gagner en performance puisque l'appel à l'interruption ne prend désormais plus que le temps de demander au fil d'exécution du noyau de se réveiller, et ce fil d'exécution se retrouve ordonnancé de la même façon que n'importe quel autre processus du système, laissant donc libre champ à l'exécution des applications de plus haute priorité [15]. Cependant, cette évolution ne change pas le fait que l'interruption doit tout de même être traitée à un moment donné sur un processeur : il est alors possible de modifier, pour chaque interruption, les masques des processeurs chargés de les exécuter. De cette façon, il est possible d'isoler le traitement des interruptions à l'un des processeurs que nous aurons choisi pour la tâche, simplement via l'écriture de quelques octets dans des fichiers définis [16].

Certaines interruptions comme les interruptions non masquables (NMI) ne sont cependant pas désactivables ni ne permettent de choisir le processeur sur lequel elles s'exécuteront. En effet, ces interruptions d'un type tout particulier sont souvent utilisées pour des cas dans

lesquels le temps de réponse pour traiter l'interruption est critique, ou lorsque l'interruption levée ne devrait jamais être désactivée sur le système [17]. La raison la plus courante d'apparition de ces interruptions concerne les rapports d'erreurs matérielles à la suite desquelles il est impossible pour le système de reprendre ses activités. Elles peuvent cependant survenir sur certaines architectures pour d'autres raisons comme le débogage ou profilage du système ou des cas spéciaux comme une réinitialisation. Il existe dans le cœur du noyau Linux une procédure permettant de s'assurer que le système n'est pas bloqué, et, le cas échéant, de le débloquer. Pour fonctionner, cette procédure, appelée *NMI watchdog*, a recours à l'utilisation de NMIs réguliers lui permettant de reprendre le contrôle malgré le statut bloqué du système et d'ainsi vérifier si l'un des processeurs du système semble gelé [18]. Cette apparition régulière de NMI est néanmoins mauvaise pour l'intégrité de notre environnement puisqu'il s'agit d'interruptions dans le fonctionnement du système ou de l'application. Heureusement, il est possible de désactiver ce système de surveillance (`nmi_watchdog`).

L'utilisation de l'isolation des processeurs sur un noyau Linux demande cependant d'être vigilant. Il existe certains cas bien précis, appelés interférences processeur, où des systèmes de gestion du noyau Linux pourront tout de même reprendre le contrôle sur un processeur isolé. Les sources de ces potentielles interférences sont répertoriées et un travail est fait pour corriger leur comportement [19].

Un exemple d'obstacle au plein fonctionnement en temps réel de notre système, y compris avec l'isolation des processeurs, est le mécanisme de synchronisation sans blocage *read-copy update* (RCU) qui fait partie intégrante du noyau Linux depuis octobre 2002 [20]. Ce mécanisme permet de plus grandes performances dans un cas où une majorité de lecteurs souhaitent accéder à une variable. Il permet de séparer les opérations de destruction en deux étapes, la première étant le fait d'empêcher quiconque de voir la donnée supprimée, et la seconde qui effectue la suppression. Ces deux étapes sont séparées par une période appelée « période de grâce » durant le temps nécessaire pour que tous les lecteurs utilisant l'objet supprimé aient supprimé leur référence à celui-ci. Un exemple est donné dans la documentation du noyau Linux sur la suppression d'une liste chaînée protégée par RCU [21] : l'élément serait premièrement supprimé de la liste, il faudrait par la suite attendre que la période de grâce s'écoule, puis libérer la mémoire associée à l'élément supprimé. Si le fonctionnement de ce mécanisme améliore le parallélisme du noyau, il nécessite cependant un réveil régulier sur chaque processeur pour s'assurer de la fin des périodes de grâce. Ce réveil provoque par conséquent un retour au noyau, y compris si une application temps réel de très haute priorité est en train de s'exécuter, arrêtant dès lors son exécution, ce que nous souhaitons absolument éviter. À partir du noyau Linux 3.10, il sera possible de désactiver – ou plutôt limiter – ces réveils réguliers pour ne plus déranger le comportement temps réel d'une application [22].

Dans le cadre de cette étude, cependant, nous voulions travailler avec un noyau stable, il est donc nécessaire de trouver une solution pour ne pas être interrompu par le fonctionnement de RCU.

Plus globalement, nous ferons attention à ce que notre configuration ne se trouve pas dans une situation ou des interférences puissent se produire afin de ne pas casser les conditions d'isolation.

2.3 Outils de validation des systèmes temps réel

Pour évaluer les propriétés temps réel d'un traceur, les propriétés de l'installation de test doivent être validées. Cela consiste à mesurer les latences induites par le matériel et par le système d'exploitation. En utilisant un noyau Linux, la suite `rt-tests` semble prometteuse pour effectuer cette validation car elle contient plusieurs outils partageant cet objectif. Dans cette section, nous présenterons différents outils utiles pour l'analyse de la qualité d'un système temps réel.

2.3.1 Validation du matériel

Des latences anormales du matériel peuvent se produire sur du matériel mal configuré ou inadapté à des solutions temps réel. Pour mesurer ces latences, le module noyau `hwlat_detector` [23] peut être utilisé. Ce dernier isole le matériel du système d'exploitation via l'utilisation de l'appel système `stop_machine()` permettant de monopoliser tous les processeurs pour une durée spécifiée [24]. Pour ce faire, cet appel système désactive temporairement les interruptions, exécute la séquence de code qui lui est passée en paramètre, puis réactive les interruptions. Le code exécuté par cet appel ne peut donc pas se faire préempter.

En utilisant cet appel, il exécute donc un bout de code dont le but est d'interroger le compteur de temps processeur (TSC) à l'aide de l'appel système `get_ktime()` en continu pendant une période configurable et de rechercher les incohérences dans les données obtenues de ce compteur, tel que présenté dans le pseudocode en Figure 2.4. Si un écart est identifié, cela signifie que l'interrogation en continu a été interrompue. Étant donné la configuration du système utilisée de par l'appel à `stop_machine()` et le fait que, comme on peut le voir dans le pseudocode, les prises de temps permettant d'analyser cette différence sont contiguës, une interruption de ce genre ne peut être qu'une interruption du système de gestion (SMI). Les SMIs sont des interruptions matérielles utilisées au niveau des processeurs pour exécuter différentes tâches telles que rapporter des erreurs matérielles, qu'elles soient fatales ou non, s'assurer des limitations thermiques ou encore vérifier la santé du système [25]. Étant donné que les SMIs sont programmés par les développeurs des micrologiciels des processeurs, ils

1: $echantillon \leftarrow 0$
2: $diff \leftarrow 0$
3: $total \leftarrow 0$
4: $debut \leftarrow$ **read** CpuTsc()
5: **do**
6: $\quad t_1 \leftarrow$ **read** CpuTsc()
7: $\quad t_2 \leftarrow$ **read** CpuTsc()
8: $\quad total \leftarrow t_2 - debut$
9: $\quad diff \leftarrow t_2 - t_1$
10: $\quad echantillon \leftarrow$ **max**($echantillon$, $diff$)
11: **while** $total <= taille\_echantillon$

Figure 2.4 Pseudocode représentant l'algorithme de base de `hwlat_detector` pour détecter une latence due au matériel

peuvent être plus ou moins fréquents dépendamment du matériel. La nature même de ces interruptions provoque des latences qui sont très difficiles à détecter. À notre connaissance, seul un processus d'élimination permet de les détecter durant l'exécution d'une application. Ce sont pour ces raisons que nous voulons éviter d'avoir des SMIs durant l'exécution d'une application temps réel. Afin de simplifier l'utilisation de ce module, l'outil `hwlatdetect`, fourni dans la suite `rt-tests`, est un script python qui permet d'automatiser un test en montant le pseudo système de fichiers `debugfs`, le module `hwlat_detector`, puis en lui donnant les différentes options pour le test et enfin en enregistrant le rapport d'exécution.

Un exemple d'un tel rapport d'exécution est présenté suite à l'exécution de la commande affichée en première ligne :

```
# ./hwlatdetect --duration=1d

hwlatdetect:  test duration 86400 seconds
   parameters:
        Latency threshold: 10us
        Sample window:     1000000us
        Sample width:      500000us
     Non-sampling period:  500000us
        Output File:       None

Starting test
test finished
Max Latency: 0us
```

```
Samples recorded: 0
Samples exceeding threshold: 0
```

Ici, nous pouvons voir que sur une exécution ayant duré 86 400 secondes – soit 24 heures –, la latence maximale observée sur le système n'a pas atteint une microseconde. L'entête du rapport rappelle les paramètres utilisés pour l'exécution de l'outil. On peut voir ici que la période totale est de une seconde, tandis que la période durant laquelle les tests sont exécutés est de une demi seconde. Il est intéressant de noter que sur certains systèmes, une latence liée aux SMI de 19 ms a été détectée grâce à cet outil.

2.3.2 Validation du système d'exploitation

Une fois que l'on s'est assuré que le matériel est apte à fonctionner dans des conditions temps réel, il est nécessaire de vérifier que le système d'exploitation l'est aussi. En effet, si le noyau Linux est de plus en plus émergeant du côté des systèmes temps réel, il reste que certaines configurations particulières peuvent être nécessaires. Avoir des outils permettant de vérifier que le système d'exploitation n'ajoute pas de latence lors d'une exécution de haute priorité est donc important.

L'outil `cyclictest`

La suite `rt-tests` fournit l'outil `cyclictest`. Cet outil permet de vérifier la performance temps réel du système d'exploitation en exécutant de multiples processus sur différents processeurs, chacun exécutant une tâche périodique [26]. Chaque tâche peut avoir une période différente, et la priorité de chaque processeur peut être définie à n'importe quelle valeur jusqu'à une priorité temps réel.

La performance est alors évaluée en mesurant la différence entre la période désirée et la période réelle de la tâche exécutée, comme le présente le pseudocode en Figure 2.5. Autrement dit, on analyse la différence entre l'heure à laquelle l'application aurait dû être réveillée et l'heure à laquelle elle a effectivement été réveillée par le système. Ce test nous donne une indication sur les latences dues à l'ordonnanceur temps réel du système, devant normalement s'assurer qu'une tâche de haute priorité soit réveillée dès le moment où elle est prête à être exécutée. Plus cette différence est grande, moins la performance est élevée. A la fin de l'exécution, l'outil permet d'obtenir un rapport par processeur des latences minimum, moyennes et maximum entre la période désirée et la période obtenue.

Un exemple de rapport d'exécution généré par `cyclictest` est présenté suite à l'exécution de la commande affichée en première ligne :

```
1: duree_du_test ← 0
2: temps ← 0
3: latence ← 0
4: debut ← read SourceTemps()
5: while duree_du_test ≤ duree_totale do
6:     t0 ← read SourceTemps()
7:     Attendre(periode)
8:     t1 ← read SourceTemps()
9:     temps ← t1 − t0
10:    latence ← temps − periode
11:    Statistiques(latence)
12:    duree_du_test ← t1 − debut
13: end while
```

Figure 2.5 Pseudocode illustrant le fonctionnement pour un processus de `cyclictest` pour détecter une latence due à l'ordonnanceur

```
# ./cyclictest -m -p95 --smp -D 1h

# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.01 0.04 0.05 1/165 3662

T: 0 ( 3635) P:95 I:1000 C:3599996 Min:    1 Act:    3 Avg:    3 Max:    11
T: 1 ( 3636) P:95 I:1500 C:2399997 Min:    1 Act:    4 Avg:    3 Max:    14
T: 2 ( 3637) P:95 I:2000 C:1799998 Min:    1 Act:    3 Avg:    3 Max:    11
T: 3 ( 3638) P:95 I:2500 C:1439998 Min:    1 Act:    4 Avg:    3 Max:     9
```

Nous pouvons voir sur ce rapport les informations concernant les différents processus exécutés par l'outil, numérotés de 0 à 3, et s'exécutant chacun sur un processeur du système, comme demandé à l'aide de l'option `-smp`. Chaque information est précédée par une lettre ou un sigle permettant de la distinguer. La lettre « P » désigne la priorité du processus en train de s'exécuter, on voit ici qu'il s'agit d'une priorité de 95 utilisée pour chacun des processus, comme nous l'avons demandé via l'option `-p95`. La lettre « I » désigne l'intervalle après lequel le fil d'exécution va se réveiller pour prendre sa mesure. La lettre « C » désigne le nombre de cycles d'exécution déjà réalisés par le processus léger. Enfin, « Min », « Act », « Avg » et « Max » désignent respectivement les latences minimum, actuelle, moyenne et maximum en microsecondes.

L'outil `preempt-test`

L'outil `preempt-test`, qui ne fait pas partie de la suite `rt-tests`, permet de vérifier si une tâche de plus haute priorité est apte à préempter une tâche de plus basse priorité [27]. Cette vérification est réalisée par l'exécution consécutive de fils d'exécution avec des priorités de plus en plus élevées. Le parent exécutant ces fils d'exécution a la plus haute priorité. Chaque fil d'exécution doit exécuter une boucle dont la durée est paramétrable. Les temps pris pour réveiller les différents fils d'exécution, préempter les tâches de plus basse priorité et exécuter le test au complet sont alors mesurées. Si le système a un comportement temps réel correct, l'exécution du test devrait permettre d'observer que les temps de réveil des tâches sont courts et que les tâches de haute priorité finissent leur travail en premier.

Un exemple de rapport d'exécution généré par `preempt-test` est présenté suite à l'exécution de la commande affichée en première ligne :

```
# ./preempt-test -t20

Starting test with 20 threads on 4 cpus
0 -> p:17 a: -1 l: 6/4 t: 20002/20326/324
1 -> p:18 a: -1 l: 6/4 t: 20002/20336/334
2 -> p:19 a: -1 l: 6/4 t: 20001/20345/344
3 -> p:20 a: -1 l: 4/4 t: 20007/20359/352
4 -> p:16 a: -1 l: 6/4 t: 39962/40276/314
5 -> p:14 a: -1 l: 13/4 t: 39984/40278/294
6 -> p:15 a: -1 l: 6/4 t: 39977/40281/304
7 -> p:13 a: -1 l: 7/5 t: 40018/40294/276
8 -> p:12 a: -1 l: 7/5 t: 59820/60084/264
9 -> p:11 a: -1 l: 7/5 t: 59938/60190/252
10 -> p:10 a: -1 l: 7/5 t: 59969/60208/239
11 -> p:9 a: -1 l: 7/5 t: 59984/60211/227
12 -> p:8 a: -1 l: 6/5 t: 79782/79996/214
13 -> p:7 a: -1 l: 8/4 t: 79898/80101/203
14 -> p:6 a: -1 l: 7/5 t: 79929/80119/190
15 -> p:5 a: -1 l: 8/5 t: 79944/80122/178
16 -> p:4 a: -1 l: 7/6 t: 99743/99907/164
17 -> p:3 a: -1 l: 9/6 t: 99861/100012/151
18 -> p:2 a: -1 l: 13/8 t: 99896/100030/134
19 -> p:1 a: -1 l: 60/60 t: 99979/100040/61
```

```
---------------------------
Stats: [min: 4us @ row 0] [max: 60us @ row 19] {ave: 8us]
---------------------------
Test PASSED
```

Chaque information de ce rapport est précédée par une lettre indiquant de quoi il s'agit. La lettre « p » indique la priorité du processus. La lettre « a » indique son affinité s'il a été décidé de fixer les processus sur un processeur en particulier. La lettre « l » indique la latence selon deux valeurs séparées par une barre oblique, la première étant la latence de réveil et la seconde la latence de retour suite au réveil. Enfin, la lettre « t » indique les informations de temps d'exécution selon trois valeurs séparées par une barre oblique, la première concerne l'heure de démarrage du processus léger, la seconde son heure de fin d'exécution, et la dernière fait la différence entre ses deux valeurs pour afficher sa durée d'exécution. Enfin, le rapport affiche les statistiques des latences minimum, maximum, et moyenne sur l'ensemble des fils d'exécution. On peut voir finalement que le test a réussi pour l'exécution de 20 fils d'exécution, comme demandé par l'option en ligne de commande `-t20`. Effectivement, les données montrent bien que les processus de plus haute priorité ont eu la priorité d'exécution sur les processus de plus basse priorité. Le processus de priorité la plus faible démontre par ailleurs le temps d'exécution le plus court tandis que sa latence est la plus élevée. Cette situation s'explique par le fait qu'il ait été mis en attente d'exécution par l'ordonnanceur en raison de sa priorité faible, le temps que les autres processus terminent leurs exécutions, puis a pu s'exécuter à la fin, d'une seule traite, sans être arrêté puisqu'aucun autre processus de plus haute priorité n'était exécuté.

2.4 Outils de traçage sous Linux

Le traçage est une technique d'analyse de performance des systèmes informatiques qui permet de récupérer des informations sur une application durant son exécution en limitant au maximum l'impact sur cette dernière. Le traçage se différencie du débogage par le fait que son but n'est pas d'arrêter le fonctionnement du programme analysé pour l'observer à un moment donné, mais au contraire de le laisser s'exécuter pour pouvoir faire une analyse à postériori du déroulement de son exécution. En outre, le traçage permet d'obtenir une vue d'ensemble des interactions entre les composantes d'une application et celles du système d'exploitation.

Dans cette section, nous nous concentrerons sur les différents outils de traçage permettant de tracer le noyau du système d'exploitation puis sur ceux d'entre eux qui ont une composante permettant de tracer une application en espace utilisateur.

2.4.1 Outils pour tracer le noyau

Afin de tracer le noyau, deux approches différentes existent : l'instrumentation dynamique et l'instrumentation statique.

L'instrumentation dynamique dans le noyau est effectuée avec `kprobes` [28]. Elle permet d'insérer dynamiquement à l'endroit désiré un dispositif du même type qu'un point d'arrêt ainsi que le code permettant de le traiter automatiquement et de reprendre l'exécution qui avait été suspendue. Ce type d'instrumentation a pour point fort le fait de ne pas avoir à toucher au code source du noyau, et par conséquent de ne pas avoir besoin de le compiler à nouveau. Cependant, le dynamisme même de cette technique implique un surcoût dans le fonctionnement normal du noyau, ainsi qu'une modification à son comportement actuel. En effet, en insérant du code dynamiquement, le noyau n'agira pas de la même façon que lorsque ce code n'était pas présent. Ce changement de comportement et le surcoût ajoutés rendent ce genre d'instrumentation peu fiable pour le traçage d'un système temps réel.

L'instrumentation statique est quant à elle réalisée à l'aide de points de traces ajoutés de manière fixe dans le code source du noyau. Si elle oblige à recompiler le noyau lorsqu'on ajoute de tels points d'instrumentation, elle permet aussi un comportement cohérent entre une exécution tracée et une exécution non tracée. De même, puisque les points préexistent dans le noyau, ils n'ajoutent en soi pas de surcoût lors du traçage. La macro `TRACE_EVENT()` est utilisée pour créer ce type de points de trace [29, 30, 31]. Elle a vu le jour suite à des réflexions pour trouver un modèle apte à fonctionner dans différentes situations, pour différents besoins de différents utilisateurs. Étant flexible et simple d'utilisation, elle est aujourd'hui le standard de la création de points de trace dans le noyau Linux. Enfin, une fois un point de trace créé avec cette macro, n'importe quel traceur supportant son format pourra y être connecté. Des traceurs comme `perf`, `ftrace`, `SystemTap` ou encore `LTTng` sont aptes à l'utiliser. Étant donné la sensibilité aux conditions d'exécution et à la latence dans le contexte d'un système temps réel, des traceurs supportant l'instrumentation statique seront préférés dans le cadre de nos recherches.

Le traceur `Feather-Trace`

`Feather-Trace` [32] est un traceur utilisant des évènements statiques très légers. Il est utilisable sur les architectures Intel x86. Ayant été conçu pour tracer des systèmes et applications temps réel, il s'agit d'un traceur à faible surcharge de latence : les points de trace inactifs ne provoquent l'exécution que d'une instruction – un saut inconditionnel – tandis que les points de trace actifs ne provoquent l'exécution que de deux instructions – un saut inconditionnel et un appel système. `Feather-Trace` est capable de travailler dans un contexte multiprocesseurs et utilise des tampons de type FIFO à fil sécurisé et sans attente. Son fonctionnement est facilement portable puisque ce traceur ne nécessite pas d'utilisation des primitives de synchronisation du système d'exploitation ni de modification du gestionnaire d'interruptions. `Feather-Trace` a été utilisé à titre d'étude de cas pour analyser les verrouillages dans le noyau Linux. Cependant, il ne supporte pas d'évènements à tailles variables, étant donné que son mécanisme de réservation de mémoire est basé sur des indices de tableau, ce qui limite le contenu des évènements et leur multiplicité. Enfin, la source temporelle utilisée pour les traces est l'appel système `gettimeofday()` qui ne permet qu'une précision à la microseconde.

Le traceur `Paradyn`

`Paradyn` est un traceur effectuant de l'instrumentation dynamique en modifiant les exécutables binaires pour y insérer des appels aux points de trace [33]. Bien qu'il utilise de l'instrumentation dynamique pouvant être faite au moment de l'exécution, `Paradyn` permet aussi de réécrire le fichier binaire pour n'ajouter qu'une faible surcharge de latence. Cette technique a été utilisée pour surveiller et analyser l'exécution de codes malveillants. Bien que l'outil de traçage offre une interface de programmation d'application (API) extensive pour modifier les exécutables, appelée `Dyninst`, il n'inclut pas de système de gestion des tampons de trace, de définition des différents types d'évènements ou encore de mécanisme d'écriture des traces. Aucune garantie ne peut ainsi être donnée quant aux conditions de traçage sur des systèmes multi-cœurs. Les composants manquants de ce traceur doivent être implémentés séparément.

Le traceur `perf`

Le traceur `perf` [34] est l'un des traceurs intégrés au noyau Linux. Il a été originellement conçu pour permettre un accès facile aux compteurs de performance présents dans les processeurs. Son utilisation s'est par la suite étendue pour s'interfacer avec la macro `TRACE_EVENT()` et par conséquent lire les points de trace du noyau Linux. Il est désormais

possible de l'utiliser pour générer des statistiques sur le nombre de fois qu'un point de trace est exécuté, par exemple, ou encore d'analyser le nombre d'évènements qu'un processeur aura enregistrés durant une période. Un outil, appelé lui aussi `perf`, est disponible dans les sources du noyau pour contrôler le traceur.

Les statistiques générées par `perf` à partir des compteurs de performance matériels et logiciels, lors de l'exécution très simpliste de l'outil, dont l'appel est en première ligne, sont de la forme suivante :

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

Performance counter stats for 'hackbench 10':

  1568,083128 task-clock              #   3,352 CPUs utilized
       34 316 context-switches        #   0,022 M/sec
        3 503 CPU-migrations          #   0,002 M/sec
       29 148 page-faults             #   0,019 M/sec
4 133 963 671 cycles                  #   2,636 GHz                     [87,27%]
1 756 375 029 stalled-cycles-frontend #  42,49% frontend cycles idle    [88,52%]
  718 044 218 stalled-cycles-backend  #  17,37% backend  cycles idle    [66,34%]
4 267 624 983 instructions            #   1,03  insns per cycle
                                      #   0,41  stalled cycles per insn
                                                                      [83,08%]
1 190 870 838 branches                # 759,444 M/sec                 [86,29%]
    9 433 482 branch-misses           #   0,79% of all branches       [88,72%]

0,467785294 seconds time elapsed
```

Ce traceur fonctionne toutefois via de l'échantillonnage de valeurs. C'est à dire qu'à chaque fois que la limite fixée par le traceur est atteinte, une interruption est générée pour que l'information soit enregistrée. On peut considérer qu'en limitant ainsi l'impact du traceur sur le système, `perf` serait avantageux pour le traçage de systèmes temps réel, mais l'utilisation d'échantillonnage est très négative concernant la précision des traces obtenues et rend alors impossible d'analyser avec précision le comportement d'un système temps réel.

`Perf` peut aussi fonctionner comme un traceur conventionnel mais il n'a pas été optimisé pour cette usage et offre des performances sur multi-cœur moins intéressantes que `ftrace` [35].

Le traceur `ftrace`

Le traceur `ftrace` [36] est lui aussi intégré au noyau Linux. Il a été développé à l'origine pour le noyau temps réel de Linux mais a depuis été intégré au noyau standard. Ce traceur, appelé `ftrace` pour *Function Tracer* ou traceur de fonctions, a été créé dans le but de suivre l'ordre des fonctions appelées dans le noyau et ainsi de déterminer le chemin critique durant une période de temps définie. Il a depuis évolué et intègre aujourd'hui des modules d'analyse plus complets tels que de l'analyse de la latence ou des analyses du fonctionnement de l'ordonnanceur [37]. Comme `perf`, `ftrace` utilise la macro `TRACE_EVENT()` pour tracer le fonctionnement du noyau. Il dispose aussi de l'avantage de ne pas nécessiter d'outil externe pour le contrôle ou la lecture des données générées, tout en étant géré via le pseudo système de fichiers `debugfs`. Cependant, `ftrace` est un traceur du noyau exclusivement.

Le traceur `SystemTap`

`SystemTap` [38] est un système de surveillance pour Linux. Il vise principalement la communauté des administrateurs systèmes et leur fournit des scripts d'instrumentations permettant automatiquement d'être interfacés avec l'instrumentation statique fournie par la macro `TRACE_EVENT()`. `SystemTap` permet aussi d'ajouter de manière simplifiée des points de trace via `kprobes`. L'instrumentation est réalisée dans un langage de script spécial qui est compilé pour produire un module noyau. L'analyse des données est fournie dans l'instrumentation elle-même et les résultats sont affichés dans la console à intervalles réguliers ou encore écrits directement dans un fichier. L'analyse est par conséquent réalisée à la volée et, à notre connaissance, il n'existe pas de moyen permettant de sérialiser les évènements bruts pour les enregistrer de façon efficace et stable afin de les analyser plus tard. `SystemTap` laisse entrevoir ici une limite par l'absence de format de trace compact qui deviendrait une limitation dans le cas de traces volumineuses. Cependant, il offre la possibilité, en utilisant ses structures de données intégrées, de calculer des statistiques personnalisées durant la capture des évènements.

Le traceur `LTTng`

`LTTng` est un traceur qui a été créé en mettant de l'avant l'importance de l'impact sur le système [39, 40]. Jusqu'à la version 2.0, `LTTng` était un ensemble de correctifs à appliquer sur les sources du noyau Linux pour apporter nombre d'instrumentations qui n'existaient pas dans le noyau standard. Depuis la version 2.0, la partie noyau du traceur existe sous la forme de modules qui sont chargés lors du démarrage des outils de traçage. Les modules fonctionnent avec les différentes instrumentations statiques du noyau utilisant la macro

`TRACE_EVENT()`. Ils permettent aussi d'activer des points de trace en utilisant `kprobes`, de tracer des fonctions et de capturer les compteurs de performance. Lors du traçage, les évènements produits sont consommés par un processus externe, le consommateur, qui les écrit sur le disque. Des tampons circulaires sont alloués par processeur pour atteindre les propriétés de mise à l'échelle et ne pas nécessiter d'attente. De plus, les variables de contrôle pour les tampons circulaires sont mises à jour par des opérations atomiques au lieu de verrouillages. Les variables de traçage importantes sont protégées par l'utilisation de structures de données RCU. Enfin, contrairement à `Feather-Trace`, il est possible de supporter des types d'évènements arbitraires par l'utilisation du format *Common Trace Format* (CTF) [41]. L'outil `babeltrace` permet de lire et convertir les traces stockées dans ce format vers du texte simple.

Une trace lue avec `babeltrace` sera de la forme suivante :

```
[02:47:05.128959473] (+0.000003560) station11-64 exit_syscall:
    { cpu_id = 0 }, { ret = 0 }
[02:47:05.128962750] (+0.000003277) station11-64 sys_readv:
    { cpu_id = 0 }, { fd = 12, vec = 0x7FFF57CB7AF0, vlen = 1 }
[02:47:05.128965968] (+0.000003218) station11-64 exit_syscall:
    { cpu_id = 0 }, { ret = 141 }
[02:47:05.128976987] (+0.000011019) station11-64 kmem_cache_alloc:
    { cpu_id = 2 }, { call_site = 0xFFFFFFFF811098A6,
    ptr = 0xFFFF8801A2BE0B80, bytes_req = 208, bytes_alloc = 256,
    gfp_flags = 32976 }
[02:47:05.129004111] (+0.000027124) station11-64 sys_poll:
    { cpu_id = 0 }, { ufds = 0x7FC1D70B7E70, nfds = 3,
    timeout_msecs = 227 }
[02:47:05.129007224] (+0.000003113) station11-64 hrtimer_init:
    { cpu_id = 0 }, { hrtimer = 18446612139423701672, clockid = 1,
    mode = 0 }
[02:47:05.129008246] (+0.000001022) station11-64 hrtimer_start:
    { cpu_id = 0 }, { hrtimer = 18446612139423701672,
    function = 18446744071579214861, expires = 849575927150,
    softexpires = 849575700151 }
[02:47:05.129009489] (+0.000001243) station11-64 rcu_utilization:
    { cpu_id = 0 }, { s = "Start context switch" }
[02:47:05.129010026] (+0.000000537) station11-64 rcu_utilization:
    { cpu_id = 0 }, { s = "End context switch" }
```

On peut y voir la date de l'évènement, la différence temporelle depuis l'évènement précédent, le nom de la machine sur lequel la trace a été enregistrée, le type d'évènement (« `exit_syscall` » pour la première ligne), le processeur sur lequel l'évènement s'exécutait et finalement l'ensemble des variables associées à l'évènement tracé. Bien sûr, il est possible d'afficher nombre d'autres informations si elles ont été enregistrées lors de la trace.

2.4.2 Outils pour tracer en espace utilisateur

Le traçage en espace utilisateur est basé sur le même principe que le traçage du noyau du système d'exploitation. Autrement dit, ce traçage est basé sur la prise d'évènements au sein d'une application, sans interrompre son exécution, tout en limitant l'impact sur celle-ci. Nombre d'implémentations d'outils de traçage pour l'espace utilisateur sont basées sur des appels systèmes bloquants, sur du formatage de chaînes (avec `printf` ou `fprintf` par exemple) ou encore permettent d'atteindre une cohérence des fils d'exécution en bloquant les ressources partagées contre l'écriture concurrente. Le système de journalisation `Poco::Logger` est un exemple d'implémentation de ce type [42]. Cette catégorie de traceurs est lente et inadéquate pour une utilisation dans le cadre de systèmes et applications temps réel ou multiprocesseurs.

Deux autres types de traceurs de l'espace utilisateur sont identifiables : ceux nécessitant une collaboration du noyau et ceux fonctionnant entièrement en espace utilisateur. Pour cette étude, nous nous intéressons principalement aux traceurs capable de corréler des traces noyaux et des traces de l'espace utilisateur et fonctionnant sous Linux. Bien que `Feather-Trace` fasse partie de cette catégorie, les limitations que nous avons relevées dans la sous-section précédente nous paraissent trop importantes pour le retenir pour nos recherches.

`SystemTap` avec `uprobes`

`SystemTap` peut être utilisé pour tracer des applications de l'espace utilisateur en plus de tracer le noyau. Avant le noyau Linux 3.8, il était nécessaire d'ajouter un correctif au noyau appelé `utrace` [43]. Ce correctif est un ensemble de codes sources qui ne sont pas intégrés au noyau mais ont pour but de remplacer de façon transparente pour l'utilisateur l'ensemble de fonctionnalités offertes par l'appel système `ptrace`, tout en ajoutant des fonctionnalités de traçage en espace utilisateur. Depuis le noyau Linux 3.8 cependant, une nouvelle composante est apparue. Cette composante appelée `uprobes` [44] fonctionne de la même manière que `kprobes` mais en espace utilisateur. Il suffit désormais d'activer cette option dans la configuration du noyau pour pouvoir tracer en espace utilisateur avec `SystemTap`.

Tracer l'espace utilisateur avec `SystemTap` apporte cependant une contrainte majeure : à l'image du fonctionnement de `Feather-Trace`, un appel système est nécessaire lors de chaque instruction tracée. Un tel comportement implique un impact majeur sur les performances de l'application que l'on souhaite tracer, surtout lorsqu'il s'agit d'une application temps réel. En effet, retourner dans le noyau du système d'exploitation implique de rendre le contrôle à ce dernier qui pourrait, si des tâches internes au noyau sont en attente, essayer de les exécuter avant de rendre le contrôle à l'application. Ce problème est valable avec les points de trace dynamiques comme ceux statiques avec ce traceur. En effet, les points de trace statiques utilisant les mêmes méthodes pour le traçage de l'application que les points de trace dynamiques, le seul gain d'utiliser un point de trace statique est ici de ne pas avoir à modifier l'exécutable avant que l'application ne s'exécute.

`LTTng-UST`

UserSpace Tracer (`UST`) est la composante espace utilisateur du traceur `LTTng` [45]. Il fournit des macros permettant d'ajouter des points de traces compilés de manière statique à un programme. Il fonctionne de la même manière que le traceur noyau, y compris en ce qui concerne la protection des variables de traçage importantes qui est ici réalisée avec une version de RCU spécialement adaptée à l'espace utilisateur appelée URCU. URCU permet d'éviter les échanges de lignes de cache entre les lecteurs qui peuvent survenir avec les systèmes traditionnels de verrouillages en lecture et écriture [46].

`LTTng-UST` permet de plus d'éviter des retours dans le noyau puisque l'application tracée se charge d'écrire dans de la mémoire partagée, qui est par la suite lue par le consommateur de traces, un autre processus séparé de l'application temps réel. La Figure 2.6 présente un schéma très simplifié de la prise d'une trace par cet outil. On identifie deux processus mis en jeu lors de la trace : l'application et le consommateur. Le consommateur présent ici est spécifique aux traces de l'espace utilisateur, il s'agit par conséquent d'un processus différent du consommateur des traces du noyau. L'application écrit directement les évènements en mémoire et n'a par conséquent pas besoin de passer par le noyau du système d'exploitation. Une fois le tampon dans lequel elle écrit rempli, elle utilise un tube de contrôle non bloquant pour réveiller le consommateur, puis change de tampon pour continuer l'enregistrement des évènements. Le consommateur va alors vider le tampon situé en mémoire partagée de façon invisible pour l'application. L'application tracée et le consommateur sont donc deux processus bien distincts ne nécessitant que peu de communications l'un avec l'autre. L'application peut par conséquent être un processus à priorité temps réel tandis que le consommateur restera un processus régulier.

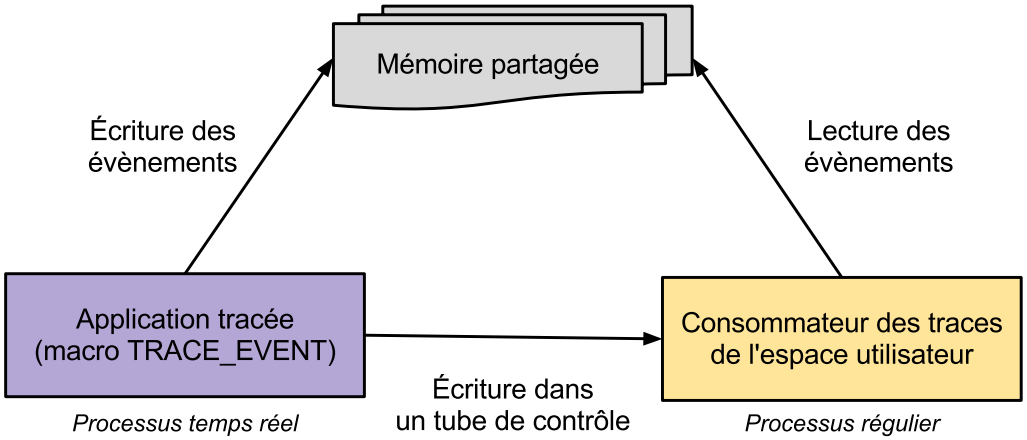


Figure 2.6 Schéma très simplifié de l'enregistrement d'une trace par `LTTng-UST`

Enfin, étant donné que les compteurs de temps du noyau et de l'espace utilisateur utilisent la même source de temps, les évènements tracés à ces deux niveaux peuvent être corrélés à l'échelle de la nanoseconde.

2.5 Conclusion de la revue de littérature

Le domaine du traçage étant relativement nouveau, on constate que peu de recherches ont été faites quant à l'intégration du traçage sur des systèmes temps réel. Les traceurs ont beaucoup évolués et sont aujourd'hui, pour certains, des composantes intégrées au noyau Linux. De même, les propriétés des systèmes temps réel sont de plus en plus recherchées sur des systèmes standards, et ainsi les évolutions faites dans le cadre de ces systèmes bien spécifiques sont petit à petit intégrées au noyau Linux. Il semble dès lors important d'arriver à corréler les deux domaines.

Un traceur ajoutant par nature de la latence pour jouer son rôle, il devient intéressant d'étudier les solutions les plus efficaces pour analyser et étudier quelles sont leurs barrières, et essayer de les franchir.

Chapitre 3

MÉTHODOLOGIE

La recherche des sources de latence dans un traceur passe tout d'abord par l'analyse des conditions actuelles de travail et par la mise en place d'une méthodologie de mesure des latences sur un système temps réel. Dans cette partie, nous présenterons les procédures utilisées pour réaliser notre étude.

3.1 Environnement de travail

3.1.1 Matériel

Deux ordinateurs, appelés ici station A et station B, étaient disponibles pour réaliser cette étude. La première étape a été de déterminer la configuration matérielle la mieux adaptée pour travailler sur du temps réel. Les configurations matérielles de ces deux systèmes sont présentées dans le Tableau 3.1.

Chaque station a reçu la même configuration avant analyse, y compris au niveau du BIOS. Cependant, en utilisant `hwlatdetect`, il a été aisé d'éliminer la station A qui laissait entrevoir des latences de près de 19 ms, ainsi causées par des interruptions matérielles, tandis que la station B souffrait de latences de l'ordre de 0 µs. Cette station, dont les résultats ont été vérifiés à l'aide de deux autres stations à configurations matérielles identiques, nous a permis d'obtenir les résultats pour l'ensemble des tests présentés dans cette étude.

Tableau 3.1 Configurations matérielles des stations A et B

Élément	Quantité	Type
Station A		
Carte mère	1	Supermicro X7DAL
Processeur	2	Intel® Xeon® E5405, avec quatre cœurs cadencés à 2.00 GHz
Mémoire vive	4	KINGSTON® DIMM DDR2 Synchrone 667 MHz de 2 GB
Station B		
Carte mère	1	Intel Corporation DX58SO
Processeur	1	Intel® Core™ i7 920, avec quatre cœurs cadencés à 2.67 GHz
Mémoire vive	3	KINGSTON® DIMM DDR3 Synchrone 1067 MHz de 2 GB

3.1.2 Logiciel

L'outil `cyclictest` a par la suite été utilisé pour vérifier les systèmes d'exploitation utilisés. La documentation de la suite d'outils `rt-tests` mentionne que `cyclictest` a été développé pour être utilisé dans des conditions de stress du système. L'outil `rteval` a été spécifiquement créé dans le but de préparer un environnement stressé puis d'exécuter `cyclictest`. Plus spécifiquement, cet outil développé en python a pour rôle d'exécuter de multiples fils d'exécution permettant de créer de la charge système à l'aide de compilations du noyau ou encore en faisant appel au programme `hackbench`. Cet outil était donc idéal pour exécuter les tests souhaités. Il a cependant été nécessaire de lui appliquer une série de correctifs permettant à l'outil de fonctionner peu importe la distribution Linux utilisée, cette série de correctifs a par la suite été intégrée à la branche principale de `rteval`.

Si nos premières analyses étaient basées sur des noyaux Linux 3.2 spécifiques à la distribution Debian, nous avons choisi d'utiliser aussi la version 3.8 du noyau Linux afin de pouvoir réaliser des tests avec le traceur `SystemTap`. Ces tests nous permettront de présenter la différence entre un traceur travaillant exclusivement en espace utilisateur et un traceur nécessitant des appels systèmes. De plus, travailler sur plusieurs versions du noyau nous permet d'apprécier la stabilité de nos résultats dans le temps.

Enfin, l'isolation CPU présentée à la Figure 3.1 est utilisée pour chacun des tests. Le cœur 0 sert à accueillir tous les processus du système, tandis que le cœur 1 est responsable de l'exécution de l'application temps réel, et le cœur 2 du traceur. Le cœur 3 n'a été utilisé que couplé à l'un des trois cœurs précédents pour vérifier l'impact d'un second cœur pour l'un de ces jeux de processeurs.

3.2 Traçage d'une application temps réel : l'outil `npt`

Si les outils présentés à la section 2.3 permettent d'étudier le comportement du système temps réel, aucun d'entre eux n'a été spécifiquement prévu pour servir d'étalon pour analyser la surcharge ajoutée par une autre application. L'outil de test de non-préemption (*Non-Preempt Test*, ou `npt`) a été développé dans ce but. Le principe de base de son fonctionnement est très simple : il s'agit d'une application qui, une fois fixée sur un processeur (via l'appel système `sched_setaffinity`), exécute en permanence des tours de boucle tout en prenant la mesure du temps à chaque tour afin de générer des statistiques sur la durée de ces tours. Le pseudocode en Figure 3.2 présente ce fonctionnement. Dans une telle situation, le processeur exécutant la tâche n'aura ainsi pas de pause du début de son exécution jusqu'à la fin. On notera que cinq tours de boucles seront exécutés en plus du nombre demandé pour laisser le

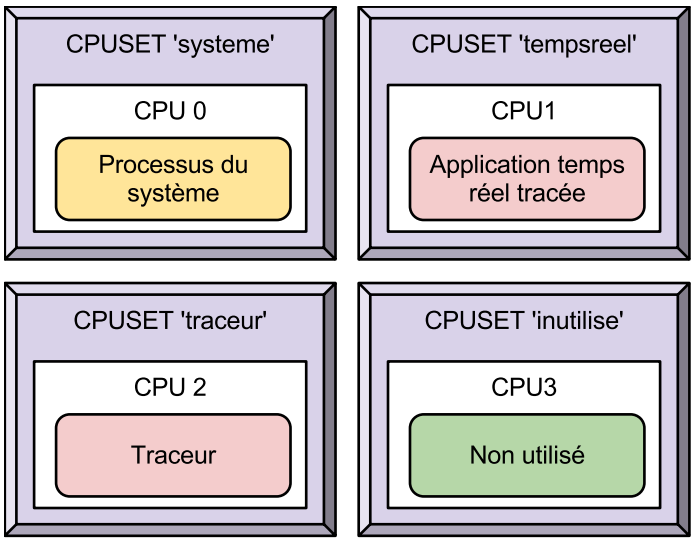


Figure 3.1 Schéma de l'isolation des processeurs pour l'exécution des tests

temps au système de prendre un rythme régulier avant les analyses, ce nombre est bien sûr paramétrable.

3.2.1 Suppression des interférences

La configuration de l'environnement, couplée à de l'isolation des processeurs, devrait normalement permettre à notre application de ne jamais être interrompue pendant sa tâche. Nous avons cependant vu que certaines sources d'interférence sont beaucoup plus difficiles à isoler, telles que les réveils réguliers de RCU. C'est pourquoi, s'il est compilé et exécuté avec les bonnes options, `npt` peut définir un environnement de travail encore plus restreint lors de son exécution. En regardant de plus près le principe de fonctionnement du module noyau `hwlat_detector`, nous avons vu qu'il utilisait l'appel système `stop_machine()` pour ne pas être dérangé. Cet appel système utilise lui-même les appels système `local_irq_save()` et `local_irq_restore()` pour respectivement désactiver les requêtes interruptions et les réactiver. Ces appels ne sont cependant disponibles que dans le noyau, et désactivent les IRQ sur l'ensemble des processeurs. Dans le cadre de fonctionnement de `npt`, nous ne voulions les désactiver que sur le processeur sur lequel il s'exécute, et pouvoir le faire à partir d'une application de l'espace utilisateur.

Les instructions assembleur CLI et STI sont des instructions privilégiées permettant respectivement de désactiver et réactiver les interruptions, incluant l'interruption nécessaire à la préemption. Des instructions assembleur peuvent être envoyées depuis une application de l'espace utilisateur directement au processeur sur lequel elle s'exécute et, en les utili-

```
1: i ← 0
2: t_0 ← read rdtsc
3: t_1 ← t_0
4:
5: while i ≤ loops_to_do do
6:     i ← i + 1
7:     duration ← (t_0 − t_1) × cpuPeriod
8:
9:     CalculateStatistics(duration)
10:    t_1 ← t_0
11:    t_0 ← read rdtsc
12: end while
13:
```

Figure 3.2 Algorithme de base de `npt`

sant, après s'être accordé les droits normalement réservés au noyau du système, `npt` n'a par conséquent d'impact que sur le fonctionnement de son processeur, tout en ayant la possibilité d'atteindre un niveau d'isolation beaucoup plus élevé.

Du côté des ressources mémoire, l'outil utilise l'appel système `mlockall` pour verrouiller la mémoire qu'il utilise, ou va utiliser, dans la mémoire vive (RAM) du système. Il évite par conséquent que ses pages mémoire soient déportées dans la mémoire *swap*, ce qui pourrait ajouter de la latence non déterministe en nécessitant des accès disque.

3.2.2 Choix d'une source de temps

Savoir sur quelle source de temps se baser est un choix difficile. Il faut prendre en considération les différents avantages de chaque système tout en tenant compte de leurs inconvénients, et ce dans les conditions très particulières d'une analyse temps réel. Étant donné le peu de temps s'écoulant entre deux boucles réalisées par `npt`, il nous semblait important d'avoir une source de temps nous permettant d'atteindre une précision très élevée tout en nécessitant une faible surcharge. De plus, nombre de sources de temps accessibles facilement, telles que `clock_gettime` peuvent nécessiter – hors utilisation de VDSO qui permet d'exporter des routines du noyau dans l'espace utilisateur – de passer par le noyau.

L'instruction assembleur `rdtsc` permet de récupérer la valeur du compteur de temps du processeur (*Time Stamp Counter*) sur lequel elle est exécutée [47]. Le compteur de temps du processeur est un compteur très précis qui a une fréquence fixe. Dans la plupart des cas aujourd'hui, cette source de temps est synchronisée entre les différents processeurs mais, même si ce n'est pas le cas, `npt` n'en souffrirait pas, étant donné qu'il s'exécute du début

à la fin sur un seul processeur. Utiliser ce compteur nécessite cependant de connaître la fréquence du processeur afin de convertir les valeurs obtenues en durées. Avant de récupérer cette fréquence, l'application se charge de stresser le processeur via une boucle vide tel que présenté dans [48], ce stress permet d'éliminer les effets de l'ajustement automatique de la fréquence du processeur. La fréquence peut finalement être récupérée directement depuis le fichier système `/proc/cpuinfo` ou être évaluée pour plus de précision. Dans l'ensemble des tests de cette étude, nous choisirons de passer par l'évaluation de la fréquence.

3.2.3 Évaluation du traceur

Étant donné la structure très simple de l'application `npt`, il est très facile de l'instrumenter pour surveiller son fonctionnement. En ajoutant un point de trace avant le début de la boucle, et un après la fin de celle-ci, on borne le travail intense effectué par le test. On peut par la suite ajouter un point de trace au cœur de la boucle pour qu'à chaque tour un évènement soit généré. Cette instrumentation est présentée sur la Figure 4.2. L'application instrumentée devra être vérifiée en premier lieu sans traçage actif, de manière à s'assurer que l'instrumentation elle-même, sans point de trace actif, n'ajoute pas de latence, puis des tests pourront être faits avec le traçage actif.

Lorsque l'on trace cette application, on obtient des résultats à analyser de deux types : la latence, fruit des statistiques générées par `npt`, et les traces d'exécution, obtenues du traceur. Afin d'évaluer la qualité du fonctionnement du traçage en temps réel, nous pouvons associer à ces deux types de résultats des marqueurs de qualité. Le premier de ces marqueurs permet ainsi de nous indiquer l'ajout de latence provoqué par la capture des évènements par le traceur. La qualité de ce marqueur sera meilleure plus la latence calculée sera faible. Le second marqueur nous informe de la qualité de la trace obtenue. La qualité de ce deuxième marqueur sera bonne si aucun évènement n'a été perdu lors de la trace, nous permettant alors de savoir en détails quels ont été les évènements générés par les points de traces actifs lors de l'exécution.

Dans un cas où le premier marqueur est de mauvaise qualité, en considérant que le deuxième marqueur est de bonne qualité et que la trace réalisée concerne le noyau et l'espace utilisateur, il sera donc possible d'analyser la série d'évènements ayant eu lieu au moment de l'ajout de latence dans l'exécution de `npt`.

Suivre cette méthodologie nous a permis de générer, sur un système configuré adéquatement et dont les capacités temps réel ont été vérifiées au préalable, des histogrammes d'impact en terme de latence pour les différents cas de traçage. Ces histogrammes sont présentés, comparés et discutés dans notre article de revue, au Chapitre 4.

Chapter 4

LINUX LOW-LATENCY TRACING FOR MULTICORE HARD REAL-TIME SYSTEMS

Authors

Raphaël Beamonte
Polytechnique Montréal
raphael.beamonte@polymtl.ca

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

Submitted to IEEE Access, July 5th, 2013

Keywords Linux, Real-time systems, Performance analysis, Tracing, Multicore processing

4.1 Abstract

Real-time systems have always been difficult to monitor and debug because of the real-time constraints which rule out any tool significantly impacting the system latency and performance. Tracing is often the most reliable tool available for studying real-time systems. In recent years, the real-time behavior of Linux systems has greatly improved and it is now possible to have latencies in the low microsecond range. In that context, tracers must ensure that their overhead is within that range, predictable and scales well to multiple cores.

The `LTTng` 2.0 tool chain has been optimized for multicore performance, scalability and flexibility. We used and extended the real-time verification tool `rteval`, using the `cyclictest` tool and `hwlat_detector` module, which allowed us to study the impact of `LTTng` on the maximum latency for serving hard real-time applications. We introduced a new real-time analysis tool to establish the baseline of real-time system performance and then to measure the impact added by tracing with `LTTng` kernel tracing and `LTTng` user-space tracing (UST). We then identified latency problems and accordingly modified the buffer switch

protocol in LTTng-UST, and the procedure to isolate the shielded real-time cores from the RCU interprocess synchronization routines. This work resulted in extended tools to measure the real-time properties of multicore Linux systems, a precise characterization of the real-time impact of LTTng kernel and UST tracing tools, and improvements to LTTng for tracing real-time systems.

4.2 Introduction

Tracing is a method to study the runtime behavior of a program's execution. It consists in recording timestamped events at key points of the execution. Because it can be used to measure latency, tracing is a fundamental tool for debugging and profiling real-time systems. To be suitable for real-time system instrumentation, a tracer must have low-overhead and consistent maximum latency in order to minimize execution timing changes and maintain determinism.

The Linux Trace Toolkit next-generation (LTTng) is a high performance tracer optimized for Linux. It supports both kernel and userspace tracing with coherent timestamps, which allow to observe system-wide execution. Earlier results for LTTng-UST show that the maximum tracepoint execution delay is 300 times the average [39]. Our goal was to assess the newer version of LTTng-UST 2.0 for use in real-time systems. Our contribution consists in a methodology to measure LTTng-UST tracepoint latency characteristics in a real-time environment, and modifications to LTTng and CPU shielding configuration to improve its real-time behavior. We measured the latency distribution in this real-time setup and compared it to results obtained on a regular setup. We developed the Non-Preempt Test (npt) tool to address these specific measurement requirements and thus were able to validate a real-time system and its tracing impact. In addition, we proposed and applied modifications to LTTng-UST in order to lower maximum latency and evaluate its effectiveness.

We present related work in section 4.3. We detail the test environment and the methodology in section 4.4. Baseline results are shown in section 4.5 while results obtained with our proposed then included improvements to LTTng-UST are discussed in section 4.6 and 4.7. Future work and the conclusion are in section 4.8.

4.3 Related work

This section presents the related work in the two main areas relevant for this paper, real-time systems and userspace tracing.

4.3.1 Existing Real-Time validation tools

To evaluate the real-time properties of the tracer, timing properties of the test setup must be validated. It consists in measuring latencies induced by the hardware and the operating system. We mainly used the rt-tests suite and related tools to perform the validation. In this section, the different tools corresponding to our needs are presented.

Hardware Abnormal hardware latencies can occur in misconfigured hardware. To measure these, we used the `hwlat_detector` kernel module [23]. This module uses the `stop_machine()` kernel call to hog all of the CPUs during a specified amount of time [24]. It then polls in a tight loop the CPU timestamp counter (TSC) for a configurable period and looks for the discrepancies in the TSC data. If there is any gap, this means that the polling was interrupted which, in a tight loop in kernel mode with interrupts disabled, could only be a non-maskable system management interrupt (SMI). SMIs are hardware interrupts used at the CPU level to perform different tasks such as reporting hardware errors, doing thermal throttling or system health checks [25]. The nature of these interrupts causes latencies that are hard to detect. Only an elimination process allows to detect such latencies while running applications. For this reason, we want to avoid SMIs during real-time application work. `Hwlatdetect` is a python script to simplify the use of the `hwlat_detector` module.

Software `Cyclictest` is a tool to verify the software real-time performance by running multiple processes on different CPUs, executing a periodic task [26]. Each task can have a different period. The priority of each process can be set to any value up to *real-time*. The performance is evaluated by measuring the discrepancy between the desired period and the real one.

The `preempt-test` tool [27] is also interesting. This tool is not part of the `rt-tests` suite but was analyzed before the development of the Non-Preempt Test tool presented in 4.5.1. It allows to verify if an higher priority task is able to preempt a lower priority one by launching threads with increasing priorities. It also measures the time it takes to preempt lower priority tasks.

4.3.2 Existing tracers

In this section, we presents characteristics of currently available tracers.

Some existing implementations of tracers rely on either blocking system calls, string formatting or achieve thread-safety by locking the shared resources for concurrent writers. For example, the logging framework `Poco::Logger` is implemented this way [42]. This

category of tracer is slow and unscalable, and thus is unsuitable for use in real-time and multi-core environment.

`Feather-trace` [32] is a low-overhead tracer implemented with thread-safe and wait-free FIFO buffers. It uses atomic operations to achieve buffer concurrency safety. It has been used to analyze locking in the Linux kernel. However, it does not support variable event size, since the reservation mechanism is based on array indexes. Also, the timestamp source is the `gettimeofday()` system call, which provides only microsecond precision instead of nanosecond.

`Paradyn` modifies binary executables by inserting calls to tracepoints [33]. The instrumentation can be done at runtime, or using binary rewriting in order to reduce the runtime overhead. This technique has been used to monitor malicious code. While the framework offers an extensive API to modify executables, it does not include trace buffer management, event types definition or trace write mechanisms. Therefore, the missing components must be implemented separately.

`SystemTap` is a monitoring tool for Linux [38]. It works by dynamically instrumenting the kernel using KProbes. It also provides a way to instrument user-space applications using UProbes since Linux kernel 3.8. In both cases, the instrumentation is done in a special scripting language that is compiled to produce a kernel module. The analysis of the data is bundled inside the instrumentation itself and the results may be printed on the console at regular interval. Hence, the analysis is done in-flight and there are no facilities, as far as we know, to efficiently serialize raw events to stable storage. Moreover, even if it is possible to determine precise places to put user-space probes to be statically compiled, these probes nonetheless incur an interrupt, just as for the dynamic probes, which is problematic for real-time tracing.

`LTTng-UST` provides macros to add statically compiled tracepoints to a program. Produced events are consumed by an external process that writes them to disk. Unlike Feather-trace, it supports arbitrary event types through the Common Trace Format [41]. The overall architecture is designed to deliver extreme performance. It achieves scalability and wait-free properties for event producers by allocating per-CPU ring-buffers. In addition, control variables for the ring-buffer are updated by atomic operations instead of locking. Moreover, important tracing variables are protected by read-copy update (RCU) data structures to avoid cache-line exchanges between readers occurring with traditional read-write lock schemes [46]. A similar architecture is available at the kernel level. Since both kernel and userspace timestamps use the same clock source, events across layers can be correlated at the nanosecond scale, which is really useful to understand the behavior of an application.

LTTng is thus the best candidate to work on real-time tracing. The rest of this paper focuses on LTTng version 2.2 which we used to perform our experiments.

4.4 Test environment

We used the tools presented previously to validate our test setup. The system consists of an Intel® Core™ i7 CPU 920 2.67 GHz, with 6 GiB of DDR3 RAM at 1067 MHz and an Intel DX58SO motherboard. Hyperthreading was disabled as it introduces unpredictable delays within cores by sharing resources between threads, both in terms of processing units and in terms of cache. This is something to avoid in real-time systems.

As expected, running `hwlatdetect` to verify the hardware latency did not find any problem; it measured no latencies for a duration of twenty-four hours. The `hwlat_detector` module often allowed us to find unexpected latencies on particular setups in our initial studies. This module thus helped us to choose a computer able to do real-time work.

The `cyclictest` tool was then used to verify the software latency. As the documentation of rt-tests specifies that `cyclictest` has been developed primarily to be used in a stressed environment, we made the test using `rteval`. The `rteval` tool is a python script written to run multiple threads which will load the system, and run `cyclictest` in a separate thread at the same time. It then produces a report giving information about the system tested and the results obtained under load. This tool was at first only working on some RedHat distribution but was intended to be working on most of the existing Linux distributions; we contributed the needed changes to work on other distributions. We performed the tests on the two different kernels used in the rest of this paper, the 3.8.13 stable kernel, (hereinafter referred as standard kernel), and the 3.8.13 stable kernel with the rt11 `PREEMPT_RT` patch, (hereinafter referred as `PREEMPT_RT` patched kernel, or RT kernel). We chose to do our tests on both these kernels to compare the performance of LTTng in a non-real-time environment versus a hard real-time one. We also expected that if LTTng was able to reach very good performance on a non-optimized system, it would most likely be able to reach it on a real-time one. Both kernels were compiled with `uprobes` support to be able to trace with `SystemTap` as well.

Table 4.1 shows the results of the `cyclictest` executions run by `rteval` on these kernels during one hour. These executions have been performed running `hackbench` [49] and a kernel compilation load (`make -j8`, to use 8 compilation threads). The `cyclictest` application was executed with command line arguments -i100 to set the base interval of the first thread, -m to prevent the memory used by `cyclictest` from being paged out, -p95 to set the priority to real-time and --smp to activate the standard options to test an SMP system.

Table 4.1 Results of the `cyclictest` executions performed on our standard (std) and `PREEMPT_RT` patched (rt) kernels

	Latencies in µs				Kernel
CPU core	0	1	2	3	type
Minimum	1	1	1	1	std
	1	1	1	1	rt
Average	2	2	2	2	std
	2	2	3	2	rt
Maximum	17	18	16	35	std
	8	5	7	5	rt

The results obtained show latencies up to 18 µs for three of the four CPU cores on which `cyclictest` was running with the standard kernel. The fourth shows a latency about two times higher than the other cores. The results are better on the `PREEMPT_RT` patched kernel. The maximum latency reached is 8 µs, instead of 18 µs on the other. We also see that the maximum of the processor with the worst latency under the `PREEMPT_RT` patched kernel is lower than the maximum of the processor with the best latency under the standard kernel (almost twice lower). The `PREEMPT_RT` patched kernel should thus be able to handle real-time tasks much better than the standard kernel.

4.5 Baseline results

In this part, we present the performance of `LTTng` in our test environment. To do so, we first introduce the Non-Preempt Test tool, developed for this purpose, and then present and discuss our latency results.

4.5.1 The Non-Preempt Test tool

One condition we wanted to test was the non preemption of a high priority process. To do so, we developed the Non-Preempt Test application, or `npt`. To isolate the effect of different latency sources, the tool can optionally first set up an ideal environment by disabling the interrupt requests (IRQ), (only when compiled with the *enable-cli-sti* command line option). The IRQs are hardware signals sent to the processor aiming to stop temporarily the running process and run an interrupt handler. Such events can add latency. In our case, we wanted to separate the latencies caused by the rest of the system from those linked to tracing, to be able to analyze the tracer. Even if disabling the IRQs is not mandatory, it allows to isolate

the factors that can cause unwanted latencies. For this reason, they were disabled for the experiments presented in this paper.

The tool then locks the process memory into RAM to prevent it from being swapped (with `mlockall`). The core of the application loops and calculates the time gap between the start of two consecutive loops, using the `rdtsc` instruction to get the Time Stamp Counter [47] of the CPU. This is similar to the `hwlat_detector` module in kernel mode. In an ideal situation, this time gap will be very short – just the time to execute the few instructions in the loop. At the end of its execution, `npt` computes latencies statistics for each loop and generates a histogram showing the different latencies reached and the number of times each one was reached. The `npt` tool was primarily designed to be executed in a CPU shielded environment, where one or more CPUs are exclusively dedicated to the real-time task. This is highly recommended but not mandatory, as `npt` automatically asks to be pinned on a specific CPU. Our CPU shielding configuration puts all the system processes on cpu0 and `npt` on cpu1, as shown in Figure 4.1. Npt tool version 1.0 was used for the tests presented in this paper.

The `rdtsc` time source is a precise counter and its frequency is fixed. Even in cases where it is not synchronized between cores, this does not affect our experiment because `npt` sets is own CPU affinity (with `sched_setaffinity`) to be scheduled on the same CPU at all time. Moreover, this is reinforced by the CPU shielding. In order to reduce the effect of transient state, `npt` also uses an empty loop to stress the CPU before getting its frequency, as presented in [48]. The frequency can then be recovered from `/proc/cpuinfo`, which is the default behavior of `npt`, but we choose to evaluate it for more accuracy (using the *eval-cpu-speed* command line option). The CPU stress allows to remove any effect of the frequency scaling, even if it is not disabled. However, the effect of the Intel® Turbo Boost Technology is not managed yet. We finally discard the first five – this number is configurable – iterations of the benchmark. The study of the pipeline warm-up latency is beyond the scope of this paper.

This tool is ideal to test the performance of the kernel and userspace `LTTng` tracers as it is easy to extend and add tracepoints in the main loop, while identifying any added latency added by the tracer, as shown in Pseudocode 4.2. The session daemon of `LTTng` is put on cpu2 during the tracing tests, to be CPU independent from `npt` and the system processes. The session daemon spawns the consumer daemons and thus they will also run on cpu2.

4.5.2 Latency results

Figure 4.3 presents the histograms generated by `npt` for an execution with 10^8 loops without tracing. As we can see, there is no latency peak. These results indicate a good

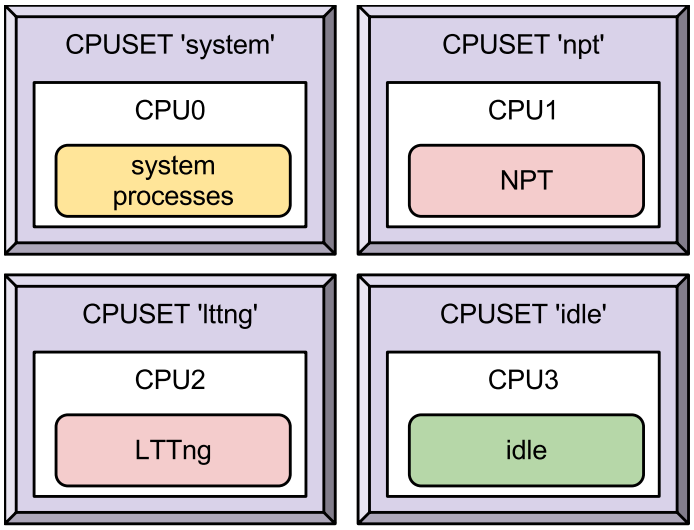


Figure 4.1 The `cpusets` organization for the running tests

1: $i \leftarrow 0$
2: $t_0 \leftarrow$ **read** *rdtsc*
3: $t_1 \leftarrow t_0$
4: **tracepoint** *nptstart*
5: **while** $i \leq loops\_to\_do$ **do**
6: $\quad i \leftarrow i + 1$
7: $\quad duration \leftarrow (t_0 - t_1) \times cpuPeriod$
8: $\quad$ **tracepoint** *nptloop*
9: $\quad$ CalculateStatistics($duration$)
10: $\quad t_1 \leftarrow t_0$
11: $\quad t_0 \leftarrow$ **read** *rdtsc*
12: **end while**
13: **tracepoint** *nptstop*

Figure 4.2 Tracepoints in `npt`

hardware and software basis, thus insuring that any added latency will be caused by the tracer.

In order to see the baseline performance of LTTng-UST and SystemTap, we ran npt for 10^8 loops with each tracer, one after the other, and then compared the results. We started our tests on kernel 3.2 at first, but as SystemTap needs uprobe to trace userspace, we then moved to the kernel 3.8. This change caused a serious performance regression in LTTng, resulting in dropped events, that we were able to trace to a change in the fadvice system call included in the first 3.8 stable release [50]. We then choose to remove the fadvice call from LTTng for our tests, as it was not necessary in our case. The Table 4.2 shows the data obtained. We can see in the table that the maximum latency of SystemTap is almost twenty times larger than the one of LTTng on a standard kernel, and around forty times larger on a PREEMPT_RT patched kernel. Moreover, the variance of the results obtained for SystemTap is much larger than the one obtained for LTTng. As the maximum latency and the variance are important values for a real-time application, LTTng is a better choice than SystemTap for this study.

Figures 4.4, 4.5 and 4.6 present the generated histograms for executions of npt with 10^8 loops with respectively kernel, UST, and kernel and UST tracers active.

We verified that no event was lost for each of the generated traces by using the babeltrace tool, which provides a command line interface to read Common Trace Format (CTF) traces.

As we can see, LTTng-UST adds many non-deterministic peaks to the execution of npt, up to 82 µs on the standard kernel and 35 µs on the PREEMPT_RT patched one. On both kernels, using kernel tracing alone doesn't seem to have any impact on the execution of npt. Latency peaks show that the impact is more important on the UST side, likely because there is an UST tracepoint directly added into the loop, therefore slowing it. As these peaks were

Table 4.2 Statistics per loop, in nanoseconds, generated by npt for 10^8 loops on both standard and PREEMPT_RT patched kernels for both LTTng-UST 2.2 and SystemTap 2.2.1

	Latencies in ns			
Kernel	standard		PREEMPT_RT patched	
Tracer	LTTng	SystemTap	LTTng	SystemTap
Minimum	270.0	581.6	262.5	911.2
Mean	498.2	777.0	497.6	1028
Maximum	82 180	1 498 000	35 260	1 476 000
Variance	3.620	23.36	4.872	33.74
Std deviation	60.17	152.8	69.80	183.7

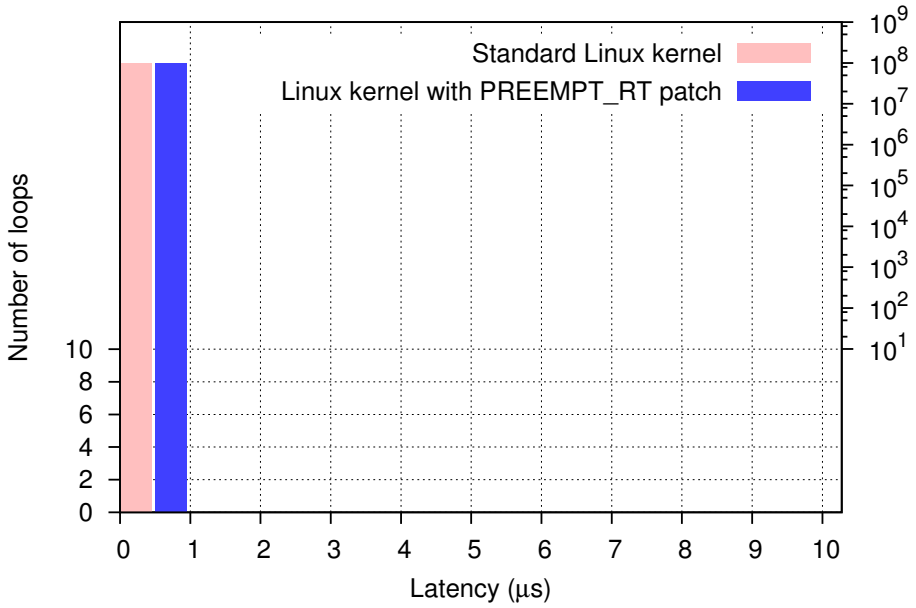


Figure 4.3 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels

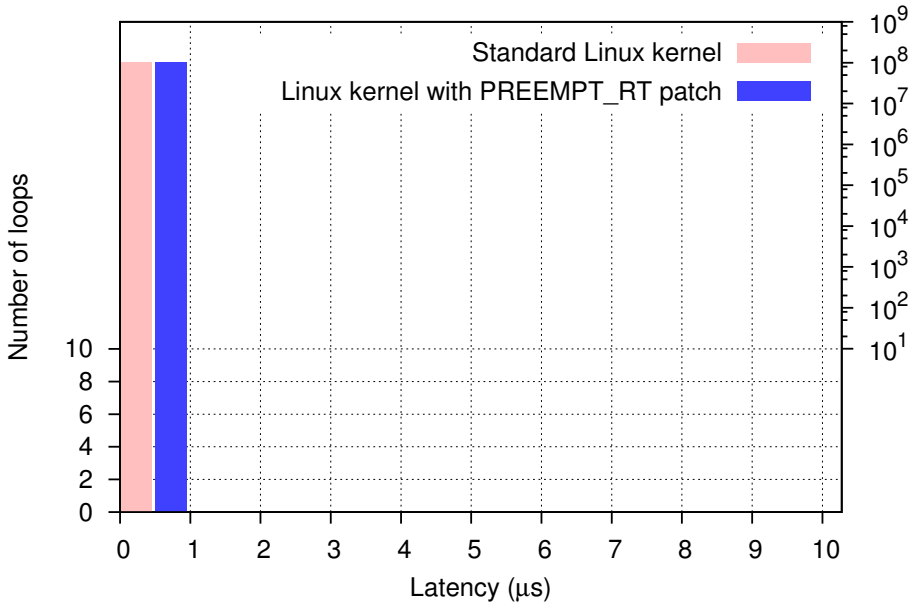


Figure 4.4 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels with `LTTng` kernel tracing

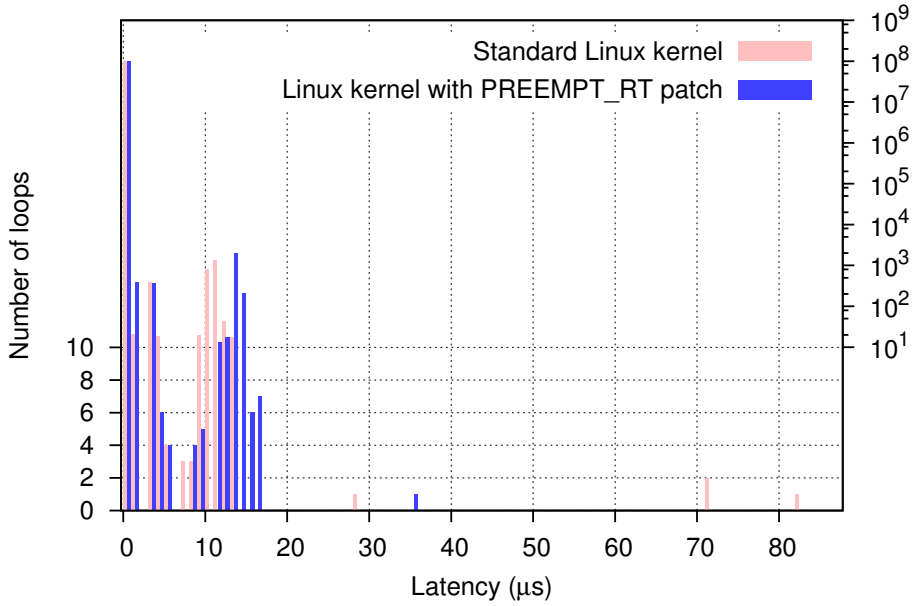


Figure 4.5 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels with `LTTng-UST` tracing

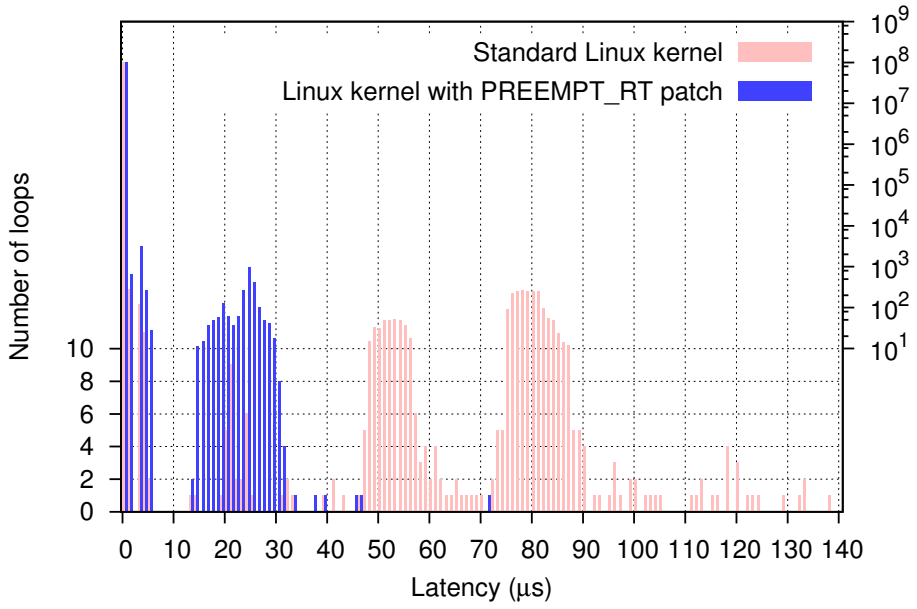


Figure 4.6 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels with `LTTng-UST` and kernel tracings

also visible in the execution of `npt` with both kernel and UST tracers, we used this trace to analyze the execution of `npt` on cpu1. Doing so, we identified that, at some point, `npt` was scheduled out from its CPU, and a lower priority `kworker` thread was scheduled for a short amount of time, before `npt` returned back to its execution. This point was in fact the exact moment where the application was using a `write` call. This call is part of UST and aims to inform the consumer, using a non-blocking `write` call on its control pipe, that the current tracing sub-buffer in the application is full.

4.6 Reducing maximum latency

The results presented in the previous section led us to modify `LTTng-UST` to create a test version in which the synchronization between the application and the consumer is removed to dissociate the work of `npt` and `LTTng`. Instead of using the kernel polling call in the consumer, we first changed it to active polling for the sake of this experimentation. Using active polling, the consumer would continuously check if the buffers were full and thus run at 100% of the CPU. However, with our shielded environment, it would not have any impact on the `npt` execution. This implementation was then improved to a timed polling using a `sleep` call to relieve the CPU which was running the `LTTng-UST` consumer. The timed polling, using delays selected between 20 and 200 microseconds, gave results as good as those of the active polling, while avoiding to overload the hosting CPU. For its part, the application (through the UST library) will not contact the consumer anymore to inform it of the sub-buffers state. We also discovered that the `getcpu` call in glibc version 2.13 was not a VDSO function yet, and thus was adding latency to `LTTng`. We upgraded our system to use glibc version 2.16 which corrects this behavior for our tests.

After further tests, these `LTTng-UST` design changes were included in `LTTng` version 2.2 as a new `read-timer` command line parameter, after the conference paper introducing them [51]. Without this parameter, `LTTng` 2.2 has the same behavior as `LTTng` 2.1. Figures 4.7 and 4.8 show the difference of added latencies using or not the `read-timer` command line parameter of `LTTng-UST` on a standard and a `PREEMPT_RT` patched kernel respectively. To avoid confusion, we will thereafter use the terms « timer `LTTng-UST` » when using the read timer mode, and « writer `LTTng-UST` » otherwise.

On the standard kernel, the maximum latency is lowered from 82 µs to 7 µs, while on the `PREEMPT_RT` patched kernel, it is lowered from 35 µs to 6 µs. If we compare the results of the timer `LTTng-UST` on both kernels in Figure 4.9, we can see that, unlike the writer `LTTng-UST` results shown in Figure 4.5, these are much more consistent between kernels.

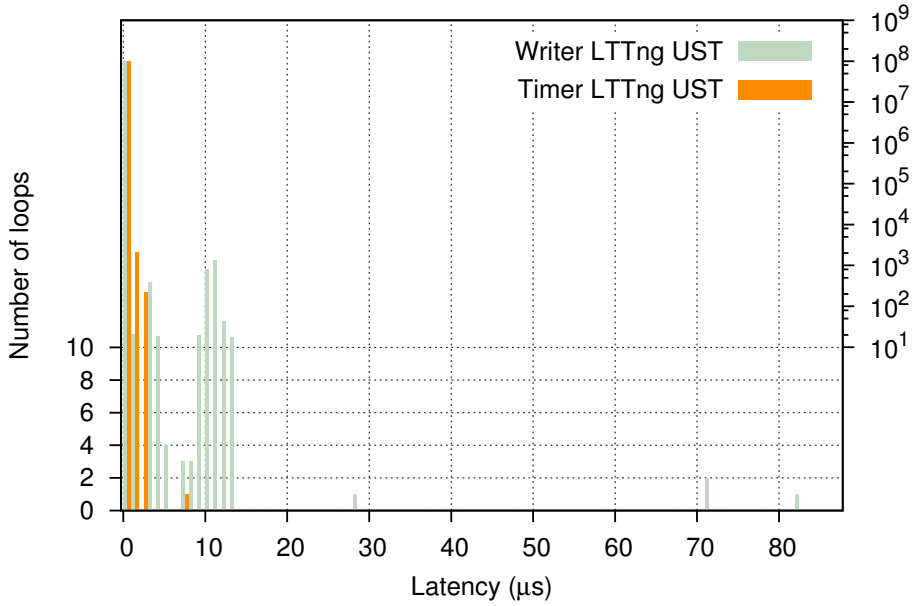


Figure 4.7 Histograms generated by `npt` for 10^8 loops on a standard kernel with writer and timer `LTTng-UST` tracing

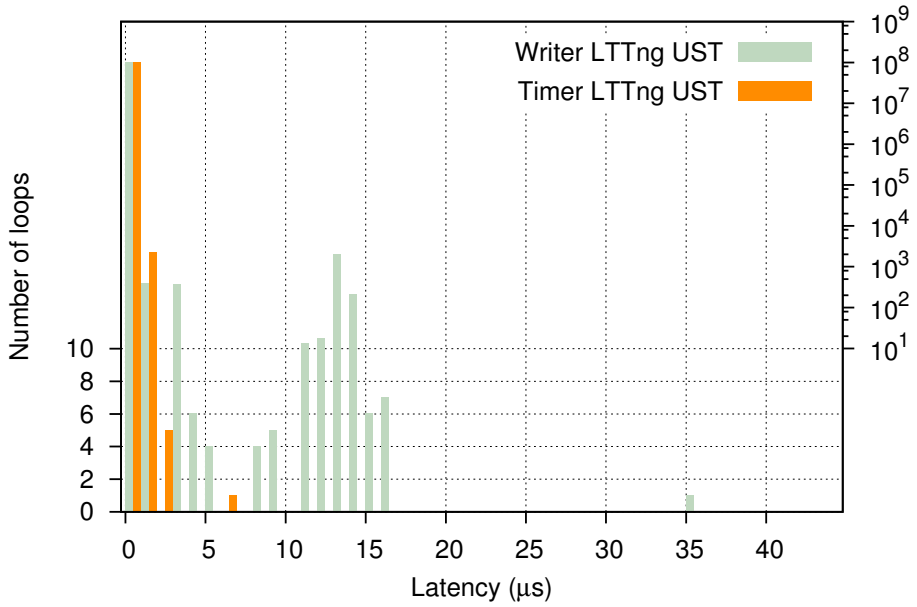


Figure 4.8 Histograms generated by `npt` for 10^8 loops on a `PREEMPT_RT` patched kernel with writer and timer `LTTng-UST` tracing

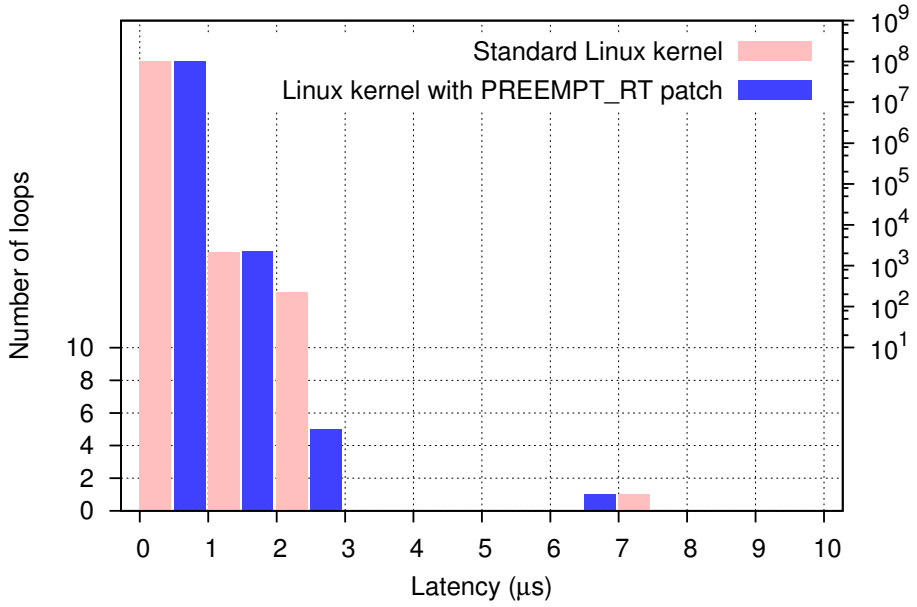


Figure 4.9 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels with timer `LTTng-UST` tracing

Moreover, Table 4.3 shows the statistics obtained from the execution of `npt` for the writer and timer designs of `LTTng` for comparison purposes. We can see that, even if the minimum duration is higher with the timer version for the standard kernel, the maximum duration, the variance and the standard deviation, which are the most important values in a real-time system, are lower.

Table 4.3 Statistics per loop, in nanoseconds, generated by `npt` on both standard and `PREEMPT_RT` patched kernels for both the writer and timer versions of `LTTng-UST`

	Latencies in ns			
Kernel	standard		`PREEMPT_RT` patched	
`LTTng-UST` 2.2	writer	timer	writer	timer
Minimum	270.0	369.4	262.5	258.0
Mean	498.2	424.2	497.6	286.8
Maximum	82 180	7569	35 260	6409
Variance	3.620	1.063	4.872	0.4541
Std deviation	60.17	32.60	69.80	21.31

4.7 Real-time tracing limits

We have seen in the previous section that the proposed design modification allows us to trace an application with a heavy UST load. However, `LTTng` still has limits when it comes to tracing the userspace application and the kernel at the same time. In the extreme case where an application would generate tracing events at the maximum rate, in a tight infinite loop, the system may be overwhelmed. In that case, where events cannot be consumed as fast as they are generated, either the generating program should be temporarily blocked, or some of the events generated will be dropped.

Npt is just such an atypical application doing almost nothing but generating events in a tight infinite loop. Interestingly, when only UST is used, `npt` on cpu1 generates a maximum volume of tracing data, but the consumer daemon on cpu2 is still able to cope with this very large volume. However, when kernel tracing is added, cpu2 has the added burden of generating kernel tracing data and consuming this additional tracing data and becomes overwhelmed. In this latest case, even if we can reach latencies as low as 6 µs, as shown in Figure 4.10, the UST part of the tracer drops many events, giving the priority to the kernel trace.

Since it's useful to have a trace with both correlated tracers (userspace and kernel), we wanted to know what is the maximum charge our setup can handle without dropping events. In most cases, a trace without any discarded events has more value than a trace with discarded ones. To measure the maximum load, we added a new tracepoint maximum frequency command line parameter to the `npt` tool, allowing to limit the maximum number of times a tracepoint will be called per second. This test aims to restrain the frequency of events, which will lighten the stress on the storing mechanism of `LTTng`.

We started a series of tests using this new option to find by binary search the number of UST events we could generate per second without discarding any of them. We chose to use 32 sub-buffers of 1 MB for the UST trace, and 32 sub-buffers of 4 MB for the kernel one. The kernel trace was started by enabling all the tracepoints currently available in `LTTng-modules` 2.2. The kernel was idle during our tests. We also ran our tests using

Table 4.4 Millions of tracepoints per second we are able to generate without any drops, in our system, with userspace and kernel tracing actives, using 32 sub-buffers of 1 MB for UST and 32 sub-buffers of 4 MB for the kernel

Kernel	all tracepoints	syscalls only
standard	2,0	2,4
`PREEMPT_RT`	2,2	2,9

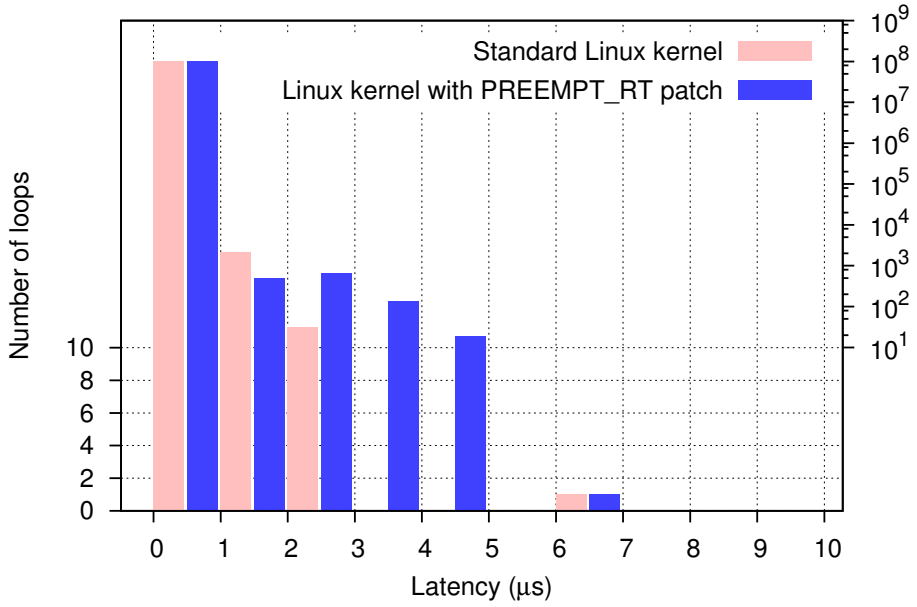


Figure 4.10 Histograms generated by `npt` for 10^8 loops on standard and `PREEMPT_RT` patched kernels with timer `LTTng-UST` and kernel tracing

only syscalls tracepoints to lighten the work of the kernel consumer. In real-life situations, one would not use all the kernel tracepoints but choose those which are really useful to the analysis of the behavior of his program. In such situations, as fewer events would be generated on the kernel side, we expect to be able to use a greater tracepoint frequency on the userspace tracing side. The results of these tests are presented in the Table 4.4.

For both standard and `PREEMPT_RT` patched kernels, we can see that `LTTng` is able to support a pretty heavy tracing charge on the userspace side, even when tracing the kernel, allowing to trace very demanding real-time applications. As expected, this charge is higher when using fewer kernel tracepoints.

4.8 Conclusion and Future Work

We have presented the effects of tracing with `LTTng` on both standard and `PREEMPT_RT` patched Linux kernels by using the Non-Preempt Test (`npt`) application. We changed the way the userspace instrumented application interacts with `LTTng` userspace tracer (UST) to reduce and improve the determinism of the added latency. Our results were promising and thus integrated upstream in the new `LTTng` 2.2 release, allowing us to lower the maximum latencies to 7 µs for the standard kernel and 6 µs for the `PREEMPT_RT` patched one when using

only userspace tracing. We also were able to determine the stress limits of `LTTng` when tracing both userspace and kernel by limiting the UST tracepoints frequency.

We believe that `LTTng` has a great potential for tracing real-time systems. Therefore, we are viewing the real-time work described in this paper as the beginning of a larger project in which collaborations and contributions are welcome. We intend to pursue our investigations to find if we can lower even more the `LTTng` latency, and create new test cases in `npt` to be able to evaluate more easily a real-time system and its real-time behavior. The latest version of `npt` can be obtained from `http://git.dorsal.polymtl.ca/?=npt.git`. Another feature of `LTTng` that could be useful for real-time applications tracing is being developped to take snapshot traces, allowing to only store the trace events in the vicinity of an identified problem.

Acknowledgments

This research is supported by OPAL-RT, CAE, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ).

Legal Statement

Chapitre 5

RÉSULTATS COMPLÉMENTAIRES

Nous évoquions à la section 3.1 le fait que nous avions commencé par réaliser l'étude sur les noyaux 3.2.0-4 fournis par les paquets de la distribution Debian. Dans l'article au Chapitre 4, les résultats présentés ne concernent que les noyaux 3.8.13 de la version stable de Linux. Dans cette partie, nous présenterons brièvement les résultats obtenus auparavant à l'aide des noyaux Linux Debian 3.2.0-4-amd64 standard (version 3.2.32-1) et Linux Debian 3.2.0-4-rt-amd64 utilisant le correctif `PREEMPT_RT` (version 3.2.32-1).

5.1 Ligne de base de résultats pour les noyaux Linux 3.2

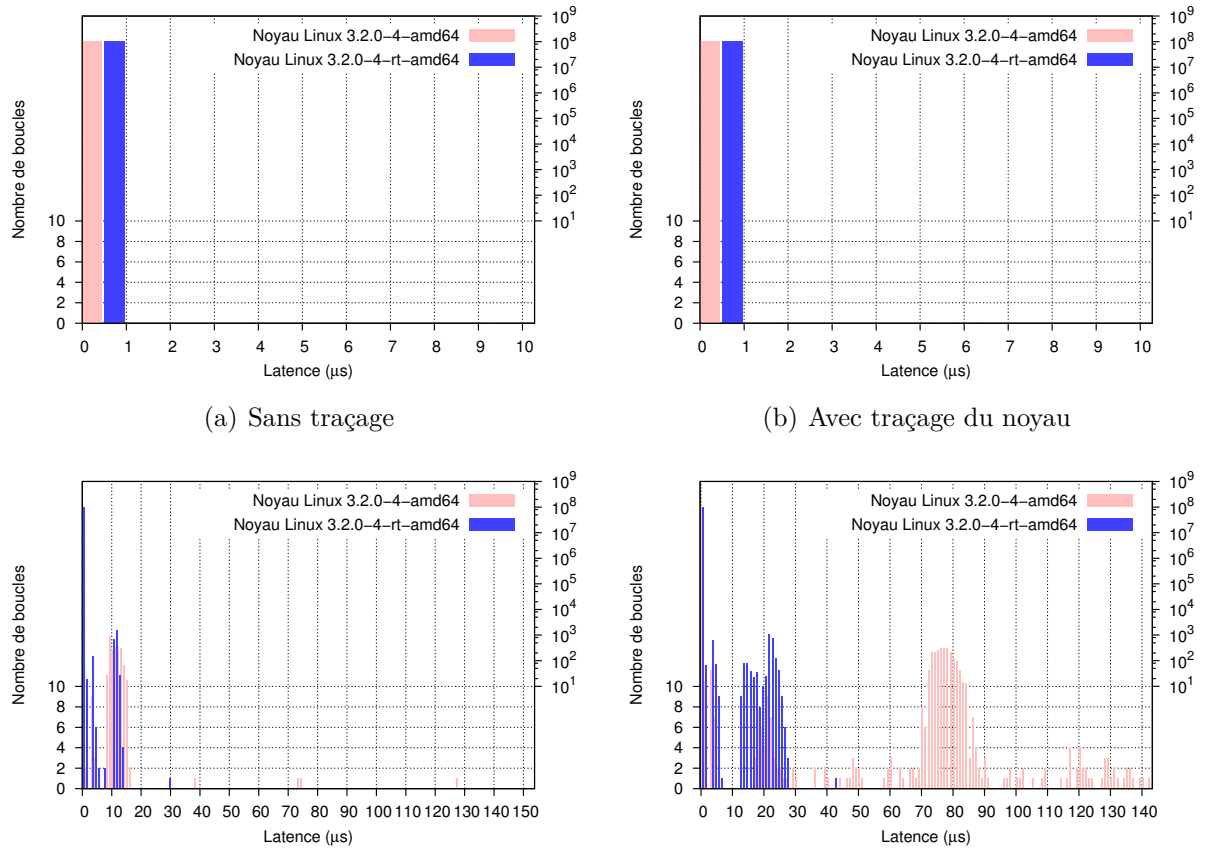
Cette section présente les résultats obtenus avec `npt` en utilisant la version 2.2 de `LTTng`, le traceur de l'espace utilisateur fonctionnant avec le tube de contrôle, ce qui est le comportement par défaut de `LTTng-UST`. Pour rappel, ce tube de contrôle est utilisé par l'application tracée lorsqu'un des tampons est plein afin de signaler au consommateur qu'il est nécessaire de le vider. La Figure 5.1 présente quatre histogrammes générés par `npt` sur les noyaux Linux Debian.

Si l'on compare les Figures 5.1(a) et 5.1(b) aux Figures 4.3 et 4.4, on remarque qu'il n'y a pas de différence dans les résultats. Sur les noyaux Linux 3.8 comme 3.2, le traçage du noyau n'a donc aucun impact sur le fonctionnement de l'application temps réel.

En comparant la Figure 5.1(c) à la Figure 4.5, cependant, on remarque que le traçage de l'application temps réel avec le traceur de l'espace utilisateur a légèrement plus d'impact sur le noyau Linux 3.8 (35 µs) que sur le noyau 3.2 (29 µs) lorsqu'il s'agit du noyau avec le correctif `PREEMPT_RT`, mais que c'est le contraire sur le noyau standard, où la version 3.8 a un impact de 82 µs, contre 127 µs pour le noyau de version 3.2.

Cette tendance est aussi identifiable en analysant les Figures 5.1(d) et 4.6, où le noyau Linux Debian 3.2 avec le correctif `PREEMPT_RT` n'ajoute que 42 µs de latence, là où le noyau Linux 3.8 en ajoute 71 µs. Sur les noyaux standards, la version 3.8 permet cependant d'obtenir de meilleurs résultats puisqu'elle ne provoque qu'une latence de 138 µs contre 142 µs pour la version 3.2.

De manière générale, cependant, en comparant l'ensemble des histogrammes de la Figure 5.1 aux Figures 4.3, 4.4, 4.5 et 4.6, nous pouvons voir une grande ressemblance dans les graphiques entre les cas de traçage semblables.



(a) Sans traçage

(b) Avec traçage du noyau

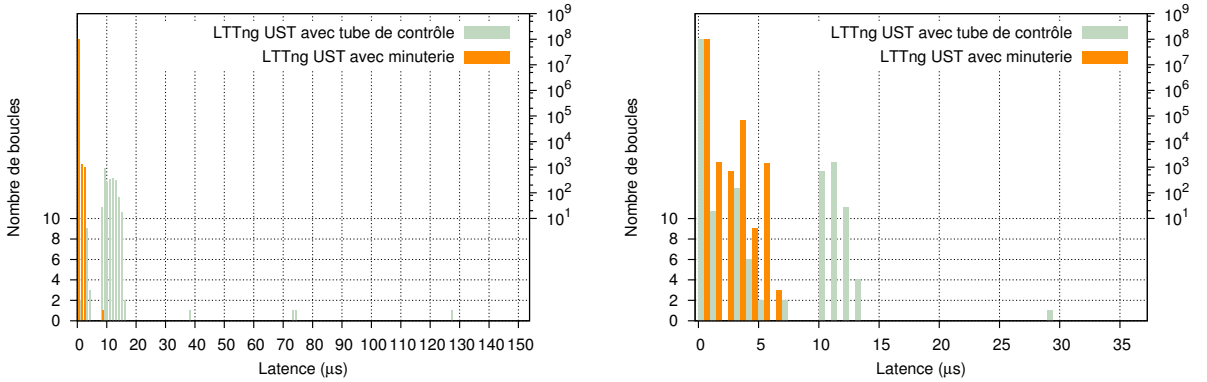
(c) Avec traçage de l'espace utilisateur utilisant le tube de contrôle

(d) Avec traçage de l'espace utilisateur utilisant le tube de contrôle et du noyau

Figure 5.1 Histogrammes générés par `npt` pour 10^8 boucles sur un noyau Linux Debian 3.2 standard et un noyau Linux Debian 3.2 utilisant le correctif `PREEMPT_RT`

5.2 Résultats en utilisant LTTng-UST avec la minuterie

Nous présentons ici les résultats obtenus avec npt en utilisant la version 2.2 de LTTng, en utilisant la minuterie pour réveiller le consommateur lors d'une trace en espace utilisateur. Pour rappel, la minuterie permet d'éliminer tout contact entre l'application tracée et le traceur en réveillant le consommateur des traces à intervalles réguliers paramétrables pour vider les tampons. La Figure 5.2 présente des histogrammes comparatifs des versions avec tube de contrôle et avec minuterie du traceur de l'espace utilisateur de LTTng, sur les noyaux Linux Debian 3.2.



(a) Sur un noyau Linux Debian 3.2 standard

(b) Sur un noyau Linux Debian 3.2 utilisant le correctif PREEMPT_RT

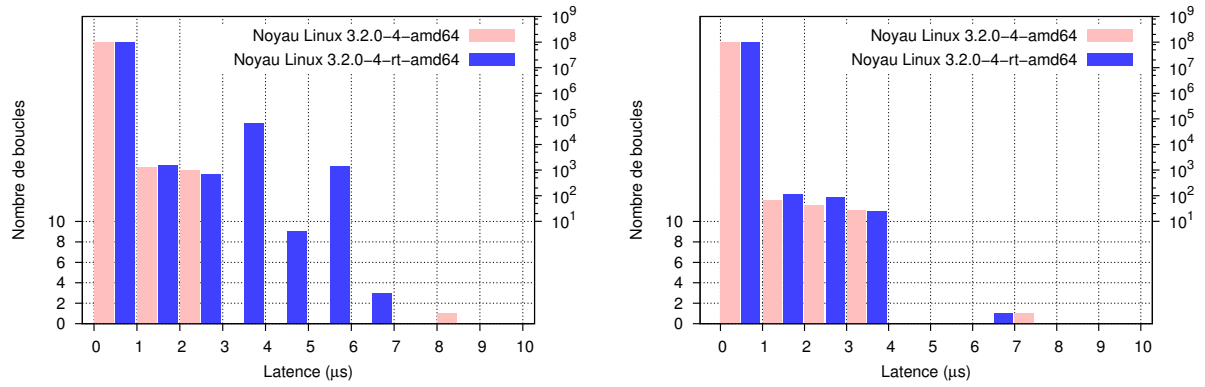
Figure 5.2 Histogrammes générés par npt pour 10^8 boucles en utilisant LTTng-UST avec minuterie et LTTng-UST avec tube de contrôle

En comparant les Figures 5.2(a) et 5.2(b) respectivement aux Figures 4.7 et 4.8, nous pouvons voir que peu importe le noyau utilisé entre la version 3.2 ou la version 3.8, nous arrivons à une latence ajoutée beaucoup plus faible en utilisant le réveil par minuterie du traceur de l'espace utilisateur de LTTng. Nous relevons de plus une forte similarité entre les résultats obtenus sur les deux versions de noyaux.

Le Tableau 5.1 montre les différentes statistiques obtenues de l'exécution de npt en utilisant le réveil par minuterie et le réveil par tube de contrôle de LTTng-UST 2.2 pour fins de comparaisons. En regardant le Tableau 4.3, nous pouvons voir que la tendance est très similaire entre les versions du noyau Linux 3.8 et 3.2. Nous pouvons cependant relever que les résultats sur le noyau Linux 3.8 concernant les valeurs clés que sont la latence maximum, la variance et l'écart-type sont plus intéressants que ceux obtenus sur le noyau Linux Debian 3.2, autant pour les noyaux standards que ceux utilisant le correctif PREEMPT_RT.

Tableau 5.1 Statistiques par boucle, en nanosecondes, générées par `npt` sur les noyaux Linux Debian 3.2 standard et Linux Debian 3.2 utilisant le correctif `PREEMPT_RT`, avec traçage de l'espace utilisateur pour la version 2.2 de `LTTng-UST` avec les réveils par minuterie et par tube de contrôle

	Latences en ns			
Noyau Linux Debian 3.2	standard		avec le correctif `PREEMPT_RT`	
`LTTng-UST` 2.2	tube de contrôle	minuterie	tube de contrôle	minuterie
Minimum	258.0	457.5	258.0	256.9
Moyenne	478.3	538.3	483.6	361.6
Maximum	127 800	8258	30 000	6934
Variance	3.545	2.002	3.394	2.071
Écart-type	59.54	44.74	58.26	43.87



(a) Sans traçage du noyau

(b) Avec traçage du noyau (sans charge)

Figure 5.3 Histogrammes générés par `npt` pour 10^8 boucles sur un noyau Linux Debian 3.2 standard et un noyau Linux Debian 3.2 utilisant le correctif `PREEMPT_RT`, avec le traceur `LTTng-UST` utilisant le réveil par minuterie

Enfin, la Figure 5.3(a) démontre vis à vis de la Figure 5.1(c) que pour le noyau Linux Debian 3.2, comme pour le noyau Linux 3.8, les résultats obtenus en terme de latence en utilisant le réveil par minuterie sont beaucoup plus constant entre les noyaux standard et utilisant le correctif `PREEMPT_RT` que ceux obtenus en utilisant le réveil par écriture sur un tube de contrôle.

Bien évidemment, les traces obtenues en n'utilisant que le traçage de l'espace utilisateur avec le réveil par minuterie sont ici aussi complètes et démontrent donc un marqueur de bonne qualité des traces.

5.3 Limites observées de points de trace par seconde sur les noyaux Linux Debian 3.2

Dans la section 4.7, nous avons présenté les résultats d'un test permettant d'évaluer par recherche binaire le nombre d'évènements pouvant être générés par seconde tout en conservant un bon marqueur de qualité de la trace. La Figure 5.3(b) démontre qu'il est ici aussi possible de tracer le noyau et l'espace utilisateur en même temps en ajoutant très peu de latence avec `LTTng-UST` et le réveil par minuterie.

Tableau 5.2 Millions d'évènements par seconde que nous sommes capables de générer sans aucune perte, dans notre système, avec les traçages de l'espace utilisateur utilisant le réveil par minuterie et du noyau actifs, en utilisant 32 tampons de 1 MB pour `LTTng-UST` et 32 tampons de 4 MB pour le noyau

Noyau	Points de trace actifs	
	tous	appels système
standard	1,8	2,4
PREEMPT_RT	2,2	2,9

Si dans ce cas aussi ne pas avoir de limite provoque des pertes d'évènements, le Tableau 5.2 permet de voir que, dans l'ensemble, les valeurs arrondies au dixième de million sont sensiblement les mêmes que celles du Tableau 4.4, excepté pour le noyau standard lorsqu'on active tous les points de trace. Dans ce dernier cas, les résultats avec le noyau Linux Debian 3.2 sont plus faibles que ceux obtenus avec le noyau 3.8.

Chapitre 6

DISCUSSION GÉNÉRALE

Dans ce chapitre, nous allons revenir sur les résultats présentés aux Chapitres 4 et 5, discuter nos contributions et leurs impacts, et enfin détailler les limitations des solutions présentées.

6.1 Retour sur les résultats

Une de nos principales contibutions est présentée plus en détails au Chapitre 3. Afin de remplir nos objectifs, il nous a en effet été nécessaire de développer une méthodologie afin de réaliser des tests et d'obtenir des résultats d'analyse nous permettant de nous orienter dans la bonne direction. Nous présentons ici la manière de définir la qualité de son environnement de travail, autant sur les plans matériel que logiciel, en utilisant un jeu d'outils spécifiques à cet effet. Nous présentons aussi la manière dont fonctionnent ces outils et les raisons pour lesquelles ils ont été choisis. Nous sommes par ailleurs intervenu sur l'un de ces outils, `rteval`, afin de le rendre disponible à l'ensemble de la communauté des utilisateurs alors qu'auparavant son fonctionnement était restreint.

Notre objectif premier était de réussir à diminuer la latence induite par le traçage, et ce afin que lors de l'analyse d'une application temps réel – application par conséquent très sensible à la latence – l'instrumentation n'affecte pas ou peu les conditions de fonctionnement régulières de celle-ci. Nous présentons dans cette étude la manière dont nous avons instrumenté une application et la façon dont nous avons utilisé cette instrumentation pour analyser, identifier puis réduire les impacts du traçage sur l'application temps réel tracée. Les résultats présentés permettent d'affirmer que notre méthode fonctionne, et qu'elle n'est pas seulement valable pour une version unique du noyau Linux mais du noyau 3.2 au noyau 3.8. Elle ne semble par conséquent pas altérée par les évolutions actuelles du noyau.

6.2 Le test de non-préemption : `npt`

L'un des résultats de cette recherche est l'outil de test de non-préeemption. Cet outil est présenté à la section 3.2 et a servi d'étalon à cette étude.

6.2.1 Fonctionnement général

En effet, `npt` est l'outil idéal pour les analyses que nous souhaitions faire : ne fonctionnant que dans l'espace utilisateur, sur un unique processeur, et dans un environnement très restreint qu'il aura – en partie – lui-même configuré ; le déroulement du test ne pourrait être perturbé que par l'intervention du matériel ou de l'outil lui-même. De plus, cet outil étant basé sur une algorithmie très basique, ajouter des points de trace aux endroits stratégiques est très facile. Lors de l'ajout de points de trace dans l'application, ces derniers nécessiteront, pour la création d'un évènement, une action de la part de l'application instrumentée.

Si `npt` a été développé dans le cadre de cette étude, son utilité est toutefois beaucoup plus large que le champ de notre travail et permet de manière générale de s'assurer qu'un système temps réel est capable d'exécuter une application temps réel de haute priorité sans que celle-ci ne se fasse préempter. La plupart des configurations faites par l'outil au système afin d'entrer dans des conditions idéales de travail sont disponibles sous forme d'options, permettant d'isoler un à un les éléments qui pourraient dégrader le déterminisme du système.

6.2.2 Données générées

Étant donné que `npt` prend des mesures de son fonctionnement en permanence, il est aisé, en comparant les résultats avec et sans traçage, de savoir l'impact du traçage sur l'application. Ces mesures sont formatées sous la forme de statistiques générales sur le comportement de l'application durant une exécution complète et permettent de connaître les durées minimum, maximum et moyennes d'un tour de boucle. La variance et l'écart-type pour ces valeurs sont aussi disponibles à titre d'indicateurs puisqu'ils permettent d'avoir une meilleure idée du déterminisme de ces valeurs. `Npt` génère aussi un histogramme permettant de savoir quels sont les différents pics de latence atteints durant l'exécution et à combien de reprise ils ont été observés. Nous présentons ces histogrammes et les informations qu'ils nous permettent d'obtenir.

Un exemple d'exécution de `npt` donne le résultat suivant :

```
# Data generated by NPT for 100000000 loops
# The time values are expressed in us.
#
# 100000000 loops done.
#
#General statistics of loops duration:
#       min:            0.369369
#       max:            7.569250
```

```
#       mean:           0.424197
#       sum:            42419727.668304
#       variance:       0.00106283
#       std dev:        0.032601
#Tracepoints generated: 100000005
#
#       time    nb. loops
#       ------------------
        0       99997711
        1       2067
        2       221
        7       1
```

Nous pouvons y voir dans l'entête les diverses statistiques d'ensemble calculées au fur et à mesure de l'exécution, suivies de l'histogramme des latences associées au nombre de fois qu'elles sont survenues durant l'exécution. Ici, l'exécution a été réalisée pour 10^8 tours de boucle. On peut remarquer la génération des 5 points de traces supplémentaires pour laisser le temps au système de prendre un rythme régulier.

6.3 Limitations de la solution proposée

La solution proposée tient compte d'une isolation parfaite de l'application temps réel. Il se peut toutefois qu'une application temps réel ne soit pas dans des conditions idéales de fonctionnement et, dans des cas de ce genre, il se peut que des interférences extérieures très difficiles à isoler viennent contraindre le fonctionnement temps réel.

De plus, les systèmes d'exploitation évoluant, il est nécessaire de tenir compte de leurs avancées pour assurer que les conditions de traçage ne se dégradent pas. En effet, nous avons pu voir qu'entre le noyau 3.2.0 et le noyau 3.8.13, une modification dans le fonctionnement de `fadvice` nous a contraint à appliquer une nouvelle modification au traceur afin d'isoler un comportement hasardeux. Par chance, cette modification n'a pas d'impact sur nos analyses compte tenu de notre méthodologie.

Un traceur fonctionnant par le biais de l'application tracée, il reste nécessaire à tout moment qu'une application fasse un arrêt – même bref – dans son exécution normale afin de laisser la place aux opérations de traçage. Nous avons pu diminuer le temps nécessaire à la réalisation de ces opérations en grande partie parce que ces opérations provoquaient des erreurs connues pour les systèmes temps réel, tel que l'inversion de priorité identifiée ou encore l'utilisation de l'appel système `clock_gettime`. Une limitation à ces améliorations

reste par conséquent les comportements du système d'exploitation et du traceur, et les étapes nécessaires pour générer un évènement lors de la trace d'une application instrumentée.

Enfin, nous avons pu identifier une limitation quant à l'écriture des traces lorsque nous activons trop de points de trace dans le traceur du noyau, et que nous traçons en même temps notre application de l'espace utilisateur générant énormément d'évènements. Nous avons pu quantifier cette limitation sur un noyau sans charge, en identifiant que cette limite était provoquée par un goulot d'étranglement lors de l'écriture des traces, les consommateurs des deux traceurs cherchant chacun à prendre la main pour écrire leurs traces.

Chapitre 7

CONCLUSION

7.1 Synthèse des travaux

Dans ce mémoire, nous avons étudié la problématique de l'utilisation du traçage pour des applications temps réel de haute priorité, sur des systèmes multi-cœurs. La performance du traceur, du point de vue de la latence ajoutée lors du traçage d'une application et de la qualité des traces obtenues suite à ce traçage, sont les critères traités dans cette étude.

Notre premier objectif était de définir une méthodologie d'analyse de l'impact du traçage sur une application temps réel. Pour cela, nous avons dû aborder les questions de validation de l'environnement de travail. À cette fin, nous avons premièrement regardé du côté du matériel pour nous assurer que la station de travail qui serait utilisée pour nos tests serait efficace dans le cadre d'un système temps réel. Nous avons ensuite analysé le système d'exploitation. Pour ces deux étapes, nous avons utilisé un ensemble d'outils permettant de valider le système sous divers aspects, chacun étant important pour le fonctionnement d'une application en temps réel. Nous avons par la suite étudié la façon d'isoler chacun de nos processeurs afin de définir un environnement idéal, l'application temps réel et le traceur pouvant ainsi avoir chacun leur processeur.

Cette méthodologie a par la suite conduit au développement du logiciel `npt`, qui nous a permis de répondre à notre deuxième objectif qui était de déterminer les capacités temps réel actuelles de `LTTng`. En effet, en instrumentant `npt` puis en traçant son exécution avec les traceurs du noyau et de l'espace utilisateur de `LTTng`, nous avons pu analyser la latence ajoutée par `LTTng` lors du traçage de notre application. Ces analyses nous ont permis de voir que le traceur noyau de `LTTng` n'impactait pas le fonctionnement de notre application s'exécutant exclusivement dans l'espace utilisateur. Nous avons vu que, cependant, tracer cette application à l'aide du traceur de l'espace utilisateur `LTTng-UST` provoquait des latences non déterministes. Les résultats obtenus en traçant l'application avec les traceurs du noyau et de l'espace utilisateur en simultané montraient eux aussi des latences de ce genre.

Les traces générées lors de ces exécutions, couplées aux statistiques calculées par `npt`, nous ont par la suite permis d'analyser les causes des latences ajoutées par le traceur et d'ainsi répondre à notre troisième objectif qui était d'identifier ces causes. En effet, en utilisant la valeur des pics de latence observés et en cherchant les évènements correspondants dans les traces, nous avons pu y voir la séquence d'évènements ayant menée à l'ajout de

ces latences. Nous avons ainsi déterminé que l'impact était provoqué par des communications entre l'application tracée et le consommateur des évènements. Nous avons ainsi pu proposer des solutions pour parer à ces interférences, en utilisant `npt` pour les valider. Ces solutions, basées sur la réduction des interractions entre les applications instrumentées et le consommateur des évènements, ont été intégrées à `LTTng` depuis la version 2.2.

Enfin, nous avons pu utiliser notre outil pour comparer les performances de `LTTng` à celles de `SystemTap`. Les résultats obtenus nous ont permis de valider, dans un contexte de traçage temps réel, la supériorité des points de trace statiques fonctionnant exclusivement dans l'espace utilisateur, tels qu'utilisés par `LTTng`, à ceux nécessitant des appels systèmes pour générer les évènements, tels qu'utilisés par `SystemTap`.

7.2 Améliorations futures

Les améliorations futures de nos travaux se distinguent en deux domaines : l'amélioration des outils d'analyse des systèmes temps réel et l'amélioration du traçage en temps réel.

En effet, nous avons pu identifier à l'aide de nos travaux une lacune des outils d'analyse des systèmes temps réel existants que nous avons comblée en créant `npt`. Cependant, notre application n'est pas complète et peut encore être améliorée en lui ajoutant des cas d'analyses qui pourraient par la suite nous donner plus d'information sur le système, mais aussi sur le traçage en temps réel. Un des cas utiles que nous avons identifié serait celui de la génération très soutenue d'évènements pendant une courte période de temps, suivie d'une période de calme, ces deux fenêtres s'enchaînant l'une après l'autre, durant une période de temps définie. Ce mode d'analyse représenterait un nouveau cas d'utilisation du traçage pour une application temps réel, où une série d'évènements peut ne se produire que très rarement, mais nécessiter la génération de beaucoup d'évènements lorsqu'elle se produit.

Du côté du traçage en temps réel, un nouveau mode de traçage de `LTTng` en cours de développement permettra de répondre à une autre situation temps réel. Lorsque l'on trace une application temps réel durant une longue période de temps, on ne veut pas nécessairement conserver l'entièreté de la trace mais plus précisément un moment identifié comme problématique. Ce nouveau mode de traçage fonctionne sur le principe d'un arrêt sur image et permettra de sauvegarder la trace enregistrée dans les tampons au moment où l'on identifiera un problème, levant ainsi la limitation liée à la vitesse d'écriture sur disque. Bien entendu, il sera intéressant d'analyser ce nouveau mode de traçage avec `npt` afin de comparer ses résultats à ceux présentés dans ce mémoire.

Enfin, les nouvelles options de configuration du noyau liées aux réveils de RCU seront intéressantes à analyser dès lors que le noyau Linux 3.10 sera disponible en version stable.

Elles pourront potentiellement nous permettre de ne plus avoir besoin de désactiver les interruptions locales lors de l'exécution de `npt`.

RÉFÉRENCES

[1] T. Jones. (2008, avr.) Anatomy of real-time linux architectures. [en ligne]. Disponible sur : http://www.ibm.com/developerworks/linux/library/l-real-time-linux/

[2] V. Yodaiken et M. Barabanov, "A Real-Time Linux," *Linux Journal*, vol. 34, 1997. [en ligne]. Disponible sur : http://users.soe.ucsc.edu/~sbrandt/courses/Winter00/290S/rtlinux.pdf

[3] P. Mantegazza, E. L. Dozio, et S. Papacharalambous, "RTAI : Real Time Application Interface," *Linux J.*, vol. 2000, no. 72es, avr. 2000. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=348554.348564

[4] P. Gerum, K. Yaghmour, et P. Mantegazza. (2002, juin.) A novel approach to real-time Free Software. [en ligne]. Disponible sur : https://lwn.net/Articles/1222/

[5] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, et C. Taliercio, "Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 1, pp. 435–439, févr. 2008.

[6] Xenomai. Xenomai : Real-Time Framework for Linux. [en ligne]. Disponible sur : http://www.xenomai.org/

[7] P. E. McKenney. (2005, août) A realtime preemption overview. [en ligne]. Disponible sur : http://lwn.net/Articles/146861/

[8] (2012, sept.) CONFIG PREEMPT RT patch. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch

[9] S. Brosky, "Shielded CPUs : real-time performance in standard Linux," *Linux Journal*, vol. 2004, no. 121, pp. 9–, mai 2004. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=982972.982981

[10] S. Brosky et S. Rotolo, "Shielded Processors : Guaranteeing Sub-millisecond Response in Standard Linux," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA : IEEE Computer Society, avr. 2003, pp. 120.1–. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=838237.838351

[11] P. Menage, P. Jackson, et C. Lameter. (2004) CGROUPS. [en ligne]. Disponible sur : https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[12] S. Derr, P. Jackson, C. Lameter, P. Menage, et H. Seto. (2004) CPUSETS. [en ligne]. Disponible sur : https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt

[13] (2011, sept.) Cpuset Management Utility. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/Cpuset_Management_Utility

[14] A. Tsariounov. (2010, sept.) How to Shield Your Linux Resources. [en ligne]. Disponible sur : http://www.suse.com/documentation/slerte_11/pdfdoc/how_to_shield_your_linux_resources/how_to_shield_your_linux_resources.pdf

[15] J. Corbet. (2009, août) The realtime preemption endgame. [en ligne]. Disponible sur : http://lwn.net/Articles/345076/

[16] D. Domingo et L. Bailey, édit., *Red Hat Enterprise Linux 6 Performance Tuning Guide*. Raleigh : Red Hat, 2011, ch. Interrupts and IRQ Tuning. [en ligne]. Disponible sur : https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-irq.html

[17] K. Adams. (2005, oct.) Linux NMIs on Intel 64-bit Hardware. [en ligne]. Disponible sur : http://x86vmm.blogspot.ca/2005/10/linux-nmis-on-intel-64-bit-hardware.html

[18] M. Slusarz, C. Gorcunov, I. Molnar, A. Rozanski, et F. Luis Vázquez Cao. (2008) NMI watchdog. [en ligne]. Disponible sur : https://www.kernel.org/doc/Documentation/nmi_watchdog.txt

[19] G. Ben-Yossef. (2012) Sources of CPU interference in core Linux code. [en ligne]. Disponible sur : https://github.com/gby/linux/wiki

[20] P. E. McKenney et J. Walpole. (2007, déc.) What is RCU, Fundamentally ? [en ligne]. Disponible sur : http://lwn.net/Articles/262464/

[21] P. E. McKenney. (2011) RCU Concepts. [en ligne]. Disponible sur : https://www.kernel.org/doc/Documentation/RCU/rcu.txt

[22] J. Corbet. (2013, mai) (Nearly) full tickless operation in 3.10. [en ligne]. Disponible sur : http://lwn.net/Articles/549580/

[23] (2011, nov.) The hwlat_detector module documentation. [en ligne]. Disponible sur : https://www.kernel.org/pub/linux/kernel/projects/rt/2.6.33/patch-2.6.33.9-rt31

[24] (2005) stop_machine.h. [en ligne]. Disponible sur : https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/include/linux/stop_machine.h?id=v3.8.13

[25] K. Mannthey. (2009, sept.) Running without Systems Management Interrupts. [en ligne]. Disponible sur : http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMI-plumers-2009.pdf

[26] (2012, mai) Cyclictest. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/Cyclictest

[27] (2012, mai) Preemption Test. [en ligne]. Disponible sur : https://rt.wiki.kernel.org/index.php/Preemption_Test

[28] S. Goswami. (2005, avr.) An introduction to KProbes. [en ligne]. Disponible sur : http://lwn.net/Articles/132196/

[29] S. Rostedt. (2010, mars) Using the TRACE_EVENT() macro (Part 1). [en ligne]. Disponible sur : http://lwn.net/Articles/379903/

[30] ——. (2010, mars) Using the TRACE_EVENT() macro (Part 2). [en ligne]. Disponible sur : http://lwn.net/Articles/381064/

[31] ——. (2010, avr.) Using the TRACE_EVENT() macro (Part 3). [en ligne]. Disponible sur : http://lwn.net/Articles/383362/

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

[33] A. R. Bernat et B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, ser. PASTE '11. New York, NY, USA : ACM, sept. 2011, pp. 9–16. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/2024569.2024572

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

[35] M. Desnoyers. (2010, août) A new unified Lockless Ring Buffer library for efficient kernel tracing. [en ligne]. Disponible sur : http://www.efficios.com/pub/linuxcon2010-tracingsummit/presentation-linuxcon-2010-tracing-mini-summit.pdf

[36] J. Edge. (2009, mars) A look at ftrace. [en ligne]. Disponible sur : http://lwn.net/Articles/322666/

[37] S. Rostedt. (2008) ftrace - Function Tracer. [en ligne]. Disponible sur : https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[38] F. C. Eigler, "Problem Solving With Systemtap," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, juill. 2006, pp. 261–268.

[39] M. Desnoyers et M. R. Dagenais, "The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, juill. 2006, pp. 209–224.

[40] M. Desnoyers, “Low-impact operating system tracing,” Thèse de doctorat, École Polytechnique de Montréal, Québec, Canada, déc. 2009.

[41] ——. (2013, mai) Common Trace Format (CTF) Specifications. [en ligne]. Disponible sur : http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.txt

[42] Applied Informatics Software Engineering GmbH. (2012) POCO C++ Libraries. [en ligne]. Disponible sur : http://pocoproject.org/

[43] J. Corbet. (2007, mars) Introducing utrace. [en ligne]. Disponible sur : http://lwn.net/Articles/224772/

[44] J. Keniston et S. Dronamraju. (2010, avr.) Uprobes : User-Space Probes. [en ligne]. Disponible sur : http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf

[45] P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, “Combined Tracing of the Kernel and Applications with LTTng,” in *Proceedings of the 2009 Linux Symposium*, juill. 2009. [en ligne]. Disponible sur : http://www.dorsal.polymtl.ca/static/publications/fournier-combined-tracing-ols2009.pdf

[46] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, et J. Walpole, “User-Level Implementations of Read-Copy Update,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, févr. 2012.

[47] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Intel Corporation, déc. 2009, vol. 3, no. 253669-033US.

[48] A. Thankashan. (2010, sept.) High Performance Time Measurement in Linux. [en ligne]. Disponible sur : http://aufather.wordpress.com/2010/09/08/high-performance-time-measuremen-in-linux/

[49] C. Williams et D. Sommerseth. (2010, févr.) Manpage for hackbench. [en ligne]. Disponible sur : http://man.cx/hackbench(8)

[50] M. Gorman. (2013, févr.) mm/fadvise.c : drain all pagevecs if POSIX_FADV_DONTNEED fails to discard all pages. [en ligne]. Disponible sur : https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit?id=bb01afe62feca1e7cdca60696f8b074416b0910d

[51] R. Beamonte, F. Giraldeau, et M. Dagenais, “High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems,” in *Proceedings of the 14th Real-Time Linux Workshop*. OSADL, oct. 2012.