UNIVERSITÉ DE MONTRÉAL

ANALYSE DE PERFORMANCE ET CARACTÉRISATION DE COMPORTEMENT À L'AIDE D'ENREGISTREMENT D'ÉVÉNEMENTS NOYAU

Karim Yaghmour
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES (M.Sc.A.)
GÉNIE INFORMATIQUE
JUIN 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE

Ce mémoire intitulé:

ANALYSE DE PERFORMANCE ET CARACTÉRISATION DE COMPORTEMENT À L'AIDE D'ENREGISTREMENT D'ÉVÉNEMENTS NOYAU

présenté par: Karim Yaghmour
en vue de l'obtention du diplôme de: Maîtrise ès science appliquées
a été dûment accepté par le jury d'examen constitué de:

Guy Bois, Ph.D., président
Michel Dagenais, Ph.D., membre et directeur de recherche
Richard Gourdeau, Ph.D., membre

À Sonia, pour sa patience et son amour.

Remerciements

En premier et avant tous, j'aimerais remercier Michel Dagenais pour la confiance qu'il m'a fait dès le départ. Le degré de liberté que j'ai eu d'explorer de nouvelles idées et de travailler dans l'environnement de mon choix a été déterminant au succès éventuel du projet. S'y ajoutent l'expérience et l'expertise de Michel qui ont été des atouts précieux lors des prises de décisions et aux choix directionnels du projet. Ses conseils et éclaircissements ont, de fait, façonné le projet et ont grandement contribué à son succès.

Avec un projet de logiciel libre comme celui-ci, plusieurs personnes ont contribué aux choix techniques et tactiques qui en ont guidé le dévelopement et la promotion. À cet égard, un merci particulier s'adresse à Jacques Gélinas qui a, plus d'une fois, partagé ses connaissances techniques avec moi et m'a fourni une vision d'expert sur les logiciels libres. Je suis aussi reconnaissant à Richard Stallman qui a pris le temps de m'expliquer les ramifications légales de l'usage de la license GNU GPL et des répercussions de l'attribution de droits d'auteurs à des entreprises. Ses conseils ont été précieux à la survie du projet en tant que logiciel libre.

Werner Almesberger a été un des premiers à reconnaître le potentiel de LTT et a contribué plusieurs recommandations que je me suis efforcé de suivre. L'équipe de Zentropix (depuis racheté par Lineo) a aussi été une des premières à reconnaître l'utilité de l'outil pour des applications industrielles et sont aussi les premiers à avoir financé certains ajouts. Je nomme ici particulièrement Stuart Hughes, Steve Papacharalambous et Jim Norton.

La collaboration de Jean-Hughes Deschênes dans le cadre de son Projet de Fin d'Études a été importante au dévelopement de l'interface graphique de LTT. Cette contribution est d'autant plus importante que l'aspect visuel est souvent celui qui accroche le premier les usagers potentiels. Dans le même esprit, Yoon-Ho Park et le comité technique de USENIX 2000 ont contribué plusieurs commentaires à l'article qui a été présenté à cette conférence et qui a permis d'asseoir la légitimité et l'innovation du projet.

L'ouverture de l'équipe de RTAI a aussi été une étape importante du développement de LTT. En plus de Steve et Stuart, je reconnais ici l'aide précieuse de Pierre Cloutier, Paolo Montegazza et David Schleef. Merci aussi à Guy Bois qui m'a donné la chance d'appliquer certains principes de RTAI et de LTT dans le cadre d'un cours sur les systèmes temps-réels.

Je suis aussi reconnaissant aux individus qui ont utilisé, testé, commenté et contribué au projet. Ils sont trop nombreux, mais leurs contributions n'en sont pas moins importantes.

Résumé

Plusieurs outils sont disponibles pour faire l'analyse de systèmes informatiques. Bien que plusieurs de ces outils soient assez répandus, l'information qu'ils rapportent demeure insuffisante ou incomplète. Ainsi, les outils basés sur l'échantillonnage ne présentent qu'une estimation du comportement du système et les outils basés sur l'instrumentation ont souvent un surcoût élevé.

Des variations sur les modes d'échantillonnage et sur les techniques d'instrumentation ont permis d'augmenter la fiabilité des outils et d'en réduire le surcoût. Toutefois, malgré ces améliorations, ces outils demeurent incapables de restituer la dynamique du système dans son ensemble.

Le but de ce travail est de fournir un ensemble d'outils capables de présenter à leurs usagers une reconstruction fidèle du comportement dynamique du système et, ce faisant, de leur fournir des mesures exactes sur sa performance. Ainsi donc, non seulement l'usager est-il en mesure d'identifier les surcoûts dus aux conflits de partage des ressources, mais il est aussi en mesure d'en identifier les causes. Il peut, par exemple, faire la différence entre le temps d'attente dû à un accès au disque et le

temps d'attente dû au partage du CPU avec d'autres processus.

L'ensemble d'outils décrit plus bas, le Linux Trace Toolkit (LTT), accumule ses données par moyen d'instrumentation du noyau. L'instrumentation consiste à enregistrer les événements noyau importants dans un tampon pour fins d'analyse. Un ensemble de modules logiciels est utilisé pour permettre cette fonctionnalité. Un outil d'analyse est aussi mis à la disposition de l'usager pour lui permettre d'analyser les données accumulées et d'en extraire les informations pertinentes.

Les données expérimentales montrent que les ensembles de données fournis par LTT sont uniques et permettent de reconstruire efficacement le comportement du système. De plus, le surcoût de LTT est minime et est plus petit que 2.5% lors de l'observation des événements noyau de base. Étant donné la disponibilité du code source de LTT sous forme de logiciel libre, il est facile d'en étendre la fonctionnalité quant aux événements enregistrés et le traitement des données accumulées. LTT est couramment utilisé pour fin d'analyse et de reconstruction de dynamique de systèmes et fait partie de plusieurs distributions Linux.

Abstract

Many tools are provided for the analysis of modern computer systems. Although some of these tools are commonly used, the information they provide about a system's behavior is often incomplete or insufficient. As such, tools based on sampling provide only an estimation of the system's behavior and tools based on instrumentation often have high overhead.

Variations on sampling modes and instrumentation techniques have contributed to a higher degree of accuracy and a lower degree of overhead. Yet, regardless of these enhancements, these tools remain incapable of providing the user with a reconstruction of the full dynamic behavior of the system.

The goal of this work is to provide a set of tools capable of presenting their users with an adequate reconstruction of the system's behavior and, doing so, of providing exact performance measurements. Therefore, not only is the user able to identify resource contention related overhead, but he is also able to identify the causes of this overhead. For instance, he can differentiate between time waiting for disk access and time waiting due to the sharing of the CPU with other processes.

The set of tools herein described, the Linux Trace Toolkit (LTT), collects its data by means of kernel instrumentation. This instrumentation consists of the logging of the important kernel events in a data buffer for further analysis. A set of software components is used in order to provide this functionality. An analysis tool is also provided to the user to enable him to analyze the data collected and to extract from it relevant information.

Experimental data shows that the data sets collected by LTT are unique and enable accurate system behavior reconstruction. In addition, the overhead caused by LTT is less than 2.5%, when core kernel events are monitored. Given the availability of LTT's source code as free software, extending its functionality in terms of events collected and later post-processing is easily done. LTT is currently used to help in the analysis and reconstruction of system behavior and is part of many Linux distributions.

Table des matières

Liste des tableaux

Table des figures

Introduction

À travers les différentes étapes de progression des systèmes informatiques, l'analyse de performance a été un catalyseur et un outil de changement important. Au départ, les problèmes à résoudre étaient surtout sur le plan matériel. Ainsi, il s'agissait de générer, de génération en génération, des processeurs, des mémoires, des disques, des bus, des réseaux et d'autres composantes de plus en plus rapides, de plus en plus efficaces et de plus en plus flexibles.

Cette recherche du nirvana matériel a amené son lot de contributions, du pipe-linage des instructions dans le CPU aux disques RAID en passant par les bus PCI, les interfaces SCSI et les liens ATM. À chaque génération incombait le devoir de démontrer qu'elle était meilleure que la précédente. Un des tests inconditionnel de passage était de démontrer l'augmentation de la capacité de traitement de la nouvelle génération. Pour ce faire, une analyse de la performance était effectuée pour soutenir la thèse de l'amélioration. Souvent effectué par le fabriquant, ce rite de passage était souvent secondé par les différents magazines traitant du sujet.

Toutefois, avec l'apparition de matériel de plus en plus standardisé et disponible à grande échelle, la place occupée par le matériel dans l'effort d'assemblage d'un système informatique a graduellement diminué au profit du logiciel. Avec cette progression, au matériel s'est graduellement ajoutée l'efficacité d'utilisation du matériel par le logiciel comme critère de performance. Ainsi, il ne suffisait plus qu'on détienne le matériel le plus performant. Il fallait aussi que le logiciel utilisé permette à l'usager de tirer le maximum de puissance de traitement du matériel disponible.

Ce besoin a contribué à créer un champ de spécialité en soi, l'analyse de performance des logiciels. À travers le temps, ce champ s'est enrichi de plusieurs techniques et outils d'analyse permettant de mesurer les différentes composantes de l'action du logiciel dans le temps. Si au départ de simples outils suffisaient, il est clair que la complexification des systèmes logiciels n'a fait qu'augmenter la difficulté de les analyser et d'en comprendre la dynamique.

À l'heure actuelle, il n'est plus suffisant d'utiliser un seul outil afin de comprendre le comportement d'un logiciel. Il est de plus en plus commun de faire usage d'une multitude d'outils logiciels afin de pouvoir caractériser adéquatement la performance d'un système logiciel. On utilisera souvent un profileur pour déterminer le temps passé à l'intérieur des différentes routines dans un même programme. De même, le système d'exploitation nous fournira souvent des statistiques concernant la manière dont les ressources sont partagées entre les différents processus.

Cependant, bien que les outils actuels permettent d'obtenir un grand nombre d'informations très précises sur une application en particulier et un grand nombre d'informations statistiques concernant l'ensemble du système, très peu d'outils permettent de comprendre la progression dynamique exacte du système tout en donnant des détails très précis sur la performance des différentes composantes du système. Lorsque disponible, ce type d'outil est souvent utilisé de manière marginale et n'est pas d'usage facile.

Le but de ce travail était donc de fournir un outil d'analyse dynamique qui soit facile d'usage tout en étant flexible, modulaire et ayant très peu d'impact sur la performance du système observé. Car, comme avec le principe d'incertitude d'Eisenberg, on ne peut observer un système logiciel sans en modifier le comportement. On peut, toutefois, diminuer l'impact causé par les mesures. C'était une des lignes directrices du travail effectué.

C'est avec ces objectifs qu'a commencé le travail sur le Linux Trace Toolkit (LTT) en janvier 1999. Une première relâche publique du projet fut effectuée le 22 juillet de cette même année. Depuis, plusieurs autres relâches publiques ont été effectuées et le projet a connu plusieurs améliorations et contributions. À l'heure actuelle, LTT en est à sa 7ième version et est utilisé à travers le monde dans plusieurs domaines d'application et par de grands noms tels que la NASA, Intel, HP, Boeing, IBM, Alcatel, HP, Nortel et d'autres afin de comprendre la dynamique des systèmes utilisés.

Initialement, le choix de Linux comme base du projet était purement académique puisque Linux était un des seuls systèmes Unix gratuits dont le code source était disponible et dont l'adoption et la maturité étaient relativement établies. Ce n'est que par la suite que ce choix fut confirmé par la popularité grandissante de Linux et de son adoption comme plate-forme de choix dans des systèmes commerciaux et académiques de pointe de toutes sortes.

Le document qui suit décrit les différents éléments qui entrent dans la conception de LTT et de son évaluation en tant qu'outil d'analyse de performance. Le chapitre 1 effectue une revue de littérature visant à discuter des différents outils d'analyse existants et de leurs limites. Le chapitre 2 présente les concepts sur lesquels LTT se base. Le chapitre 3 décrit les différents algorithmes et les différentes structures de données utilisés par LTT. Finalement, le chapitre 4 présente les différents résultats qui démontrent que LTT est facile d'usage, flexible, modulaire et a très peu d'impact sur le comportement et la performance du système observé.

Chapitre 1

Revue de littérature

Avant de détailler les concepts sous-jacents à LTT, il est important de faire une revue complète des différentes publications et des différents travaux effectués dans le champ de l'analyse de performance. Cette revue permettra de mieux apprécier les fonctionnalités permises par LTT et de comprendre où se situe LTT parmi l'ensemble des outils d'analyse de performance disponibles.

Pour ce faire, la discussion qui suit commence par énoncer les différents types de technologies utilisées pour mesurer la performance. Ensuite, les types d'information recueillie sont passés en revue. Après avoir couvert ces aspects conceptuels, des outils réels sont décrits et discutés. Cette description est faite en quatre parties. La première porte sur les outils d'analyse locale, la deuxième porte sur les outils d'analyse globale, la troisième porte sur les outils d'analyse dynamique et la dernière porte sur les simulateurs de systèmes. C'est à la troisième catégorie qu'appartient LTT.

1.1 Types de technologies utilisées pour mesurer la performance

Malgré l'existence d'une pléthore d'outils d'analyse de degrés de sophistication très variés, les principes de collecte de données sont relativement similaires. Il n'y a pas de prédominance d'une technique sur une autre parce que chaque technique possède ses avantages et ses aléas. Aussi, les techniques décrites ici ne sont pas limitées à une implémentation logicielle. Ainsi, une technique de mesure peut être aussi bien implémentée en logiciel qu'en matériel, chaque implémentation ayant, ici aussi, ses avantages et ses inconvénients.

Le premier moyen, et souvent le plus simple et le plus répandu, de collecter des données pour des fins d'analyse de performance est l'échantillonnage. Cette technique consiste à intervenir périodiquement pendant l'exécution du logiciel à observer et à noter différentes informations pertinentes. Au départ, les intervalles de temps utilisés pour échantillonner étaient toujours fixes. Comme on le verra plus loin, il s'est avéré que l'usage de l'échantillonnage à intervalles fixes pouvait générer des résultats erronés. Pour cette raison, et pour des raisons d'avancées technologiques au plan matériel, il est de plus en plus commun de trouver des systèmes d'analyse de performance basés sur des intervalles d'échantillonnage variables. Au stade actuel, il n'est plus possible de proposer de nouveaux systèmes basés sur des intervalles d'échantillonnage fixes sans se heurter aux critiques de la communauté scientifique ;

tant la possibilité d'erreur de cette technique est élevée.

En plus de l'intervalle fixe, il existe deux méthodes utilisées afin de produire des intevalles d'échantillonnage variables. La première consiste à utiliser un intervalle d'échantillonnage aléatoire bornée. Par exemple, on pourrait décider d'échantillonner le système dans X ms, où X se trouve entre 1 et 10. Encore que le choix de la fonction aléatoire doit être fait prudemment puisqu'il faut s'assurer d'une bonne distribution des valeurs échantillonnées. Aussi, il faut s'assurer que si la fonction aléatoire elle-même est implémentée en logiciel, qu'elle ne perturbe pas le comportement normal du système.

La deuxième technique d'échantillonnage aléatoire consiste à programmer les compteurs de performance de l'architecture sous-jacente de manière à ce que lorsqu'un certain nombre d'événements d'un certain type se produisent, un signal, souvent une interruption, soit émis afin de procéder à l'échantillonnage. Cette technique est très prometteuse puisqu'elle permet de tirer profit des capacités de mesure matérielles disponibles dans les processeurs récents. En effet, ces compteurs sont gérés par le matériel lui-même et ne nécessitent aucun support logiciel qui pourrait, par définition, ralentir la performance du système observé. Il demeure que les interruptions doivent, elles, être traitées par un gestionnaire adéquat.

En soit, l'échantillonnage peut souvent être effectué sans avoir à rajouter de code supplémentaire au logiciel observé. Il est toutefois souvent plus simple d'ajouter quelques informations concernant le code original du programme à l'intérieur du binaire résultant afin de faciliter l'étape d'analyse des échantillons. Il est clair que dans certains cas cet ajout d'information n'est pas possible, étant donné l'absence d'une portion ou d'une autre du ou des logiciels observés.

L'échantillonnage est suivi par l'instrumentation comme deuxième technique utilisée pour collecter des données concernant le comportement d'un logiciel. Contrairement à l'échantillonnage, l'instrumentation présente un degré d'intrusion plus élevé quant au programme à analyser. Ainsi, il faut parvenir, par un moyen ou un autre, à insérer à même le code exécutable des points de mesure qui permettront d'obtenir les informations voulues. Par contre, au delà du degré d'intrusion plus élevé, un logiciel instrumenté est souvent plus fidèle à son comportement original qu'un logiciel échantillonné puisque le parcours logique des instructions à exécuter est déterministe.

Il existe différents moyens d'insérer des points de mesure à l'intérieur d'un logiciel. Le premier consiste à les rajouter à même le code source original du logiciel. Cette technique est la plus simple et plusieurs programmeurs l'utilisent couramment en insérant des *printf()*s à même le code qu'ils écrivent afin de surveiller l'occurrence d'un événement ou un autre. Toutefois, l'insertion de tels appels est limitée puisque l'ordre d'affichage des données n'est pas nécessairement le même que celui de leur arrivée. Un autre moyen utilisé est de recourir à l'édition binaire. Avec cette technique, le binaire du programme est lu, instrumenté et réécrit de manière à produire des informations le concernant durant son exécution. Bien que difficile d'implémentation, cette technique a l'avantage de ne pas nécessiter de la part du programmeur un effort additionnel durant son travail et de demeurer applicable en l'absence du code source. Lorsque le code source est disponible, l'instrumentation peut être effectuée directement par le compilateur avec les options appropriées.

La dernière technique utilisée pour mesurer la performance diffère des deux premières du fait qu'elle n'agit pas en marge du matériel sur lequel le logiciel est supposé rouler mais elle en fait plutôt la simulation. Le degré de simulation peut varier mais il est clair que le matériel natif ne sera pas utilisé et que le logiciel ne roulera pas à la même vitesse que sur un système réel. L'avantage ici est que le logiciel sera observé à son insu, sans aucune forme d'intrusion ou d'instrumentation. Le désavantage majeur de cette technique est que les personnes qui en font usage devront s'assurer constamment que le simulateur qu'ils utilisent représente de manière fidèle le vrai matériel, ce qui n'est pas chose facile surtout avec un matériel de plus en plus complexe.

Enfin, bien qu'un outil n'utilise souvent qu'un seul type de collecte de données, il est clair que dans la pratique réelle les programmeurs utilisent une variété d'outils ayant des méthodes de collecte différentes afin d'obtenir un large éventail d'informations complémentaires. Il serait, ainsi, intéressant de voir s'il n'y aurait pas un moyen de bâtir des outils qui combineraient, à l'intérieur d'une même application ou interface commune, plusieurs techniques de collecte de données de manière transparente et cohérente.

1.2 Information type recueillie par les systèmes de mesure

Avant d'avoir instrumenté un logiciel ou d'avoir décidé du type de système de mesure à être utilisé, il faut au préalable décider de l'information à recueillir. Après tout, c'est bien pour recueillir cette information que l'instrumentation est faite. De manière générale, on peut classifier l'information recueillie en deux catégories : l'information locale et l'information globale. La différence entre ces deux types d'information se situant au niveau de leur utilité. Une information locale en est une qui sert à la compréhension d'une composante logicielle isolée et dont l'utilité pour la compréhension de l'ensemble des modules logiciels est plutôt réduite. À l'inverse, une information globale est une information qui sert principalement à comprendre l'interaction entre les diverses composantes logicielles. Néamoins, une information globale peut, tout de même, fournir certaines informations concernant une composante logicielle isolée, sans pour autant être assez significative pour permettre la caractérisation de la composante en question. Voyons maintenant quelques exemples des deux types d'information et leur utilité.

Souvent, l'unité de base de l'information locale est le compteur de programme. Ainsi, peu importe la corrélation temporelle, logique ou structurelle faite par la suite, il s'agira, plus souvent qu'autrement, d'enregistrer un compteur de programme et une autre information de corrélation. Dans sa forme la plus simple, c'est un compteur de temps qui sera enregistré avec le compteur de programme. Plus récemment, se sont rajoutés les compteurs de performance des divers processeurs. Aussi, on peut enregistrer des informations contenues dans la pile d'exécution telles que les séquences d'appels de fonctions. En utilisant ces informations, on peut rebâtir un arbre d'appels des fonctions avec pour chaque branche le nombre d'appels, le temps passé à exécuter les diverses branches, ainsi que les observations relatives aux compteurs matériels observés.

Du point de vue global, les informations peuvent varier énormément, mais il s'agira essentiellement d'enregistrer l'état actuel du système. Souvent, ceci passe par l'identification du processus ayant le contrôle du CPU. Il peut s'agir aussi d'identifier si le code s'exécutant actuellement appartient au noyau ou à l'application, déterminant ainsi le travail requis par le noyau afin de desservir les applications et l'usage fait par les différentes applications du CPU. Ce type de données sert à l'administrateur de système afin de déterminer les ressources de traitement nécessaires et lui permet aussi d'identifier les applications qui utilisent beaucoup de ressources. Ceci pourrait être ensuite utilisé pour choisir d'augmenter la capacité de traitement ou de changer les logiciels utilisés pour des logiciels moins gourmands.

1.3 Analyse locale

Maintenant que les types d'informations recueillies ont été couverts, examinons des exemples de systèmes de collecte de données locales. Pour chacun des systèmes couverts, les méthodes de collecte, les données recueillies et les analyses effectuées seront discutées.

1.3.1 GProf

De loin le plus répandu des outils d'analyse locale, GProf [26] est, en quelque sorte, une référence de par son ancienneté et de par la fonctionnalité qui y est implémentée, même s'il n'est qu'une réimplémentation de l'outil *prof* de AT&T. Conçu par Graham, Kessler et McKusick au début des années '80, il implémente un système de collecte et d'analyse basé sur l'échantillonnage de la tâche observée.

Afin d'utiliser GProf, l'application à observer doit être compilée avec les paramètres correspondants. Durant la compilation, le compilateur ajoutera le code nécessaire afin que l'application démarre une minuterie (timer) qui interrompra l'exécution chaque quelque temps afin de collecter les données pertinentes. Les minuteries utilisées sont les minuteries Unix conventionnelles. Il ajoutera aussi un appel à une fonction d'observation (monitoring) qui sera invoquée à l'entrée de chaque fonction instrumentée. La fonction d'observation aura pour rôle de compter le nombre de fois que la fonction observée a été appelée à travers un certain arc[1] d'appel.

Afin de réduire le nombre d'écritures au fichier de données collectées, la fonction d'observation utilise une table de hashing pour conserver les informations concernant

[1]Une fonction peut être appelée à divers endroits d'un même programme et à différentes profondeurs. Un arc d'appel correspond à l'appel d'une fonction selon une certaine séquence d'appels donnée.

les arcs et le nombre de fois qu'ils ont été parcourus. De même, la fonction de gestion de la minuterie bâtit un histogramme dynamique contenant pour chaque coup de minuterie le compteur de programme de l'application observée. La table contenant les informations concernant les arcs d'appels et l'histogramme d'appels sont écrits dans le fichier de données seulement lorsque l'application termine son exécution, diminuant ainsi l'impact dynamique qu'a l'instrumentation sur l'exécution du programme.

Une fois la collecte terminée et le fichier de données disponible, un outil est utilisé afin de rebâtir l'arbre d'appels et attribuer les temps aux diverses fonctions. Le temps passé dans les diverses fonctions est trouvé en utilisant l'histogramme d'appels. Plus il y a eu d'échantillons de compteur de programme se trouvant à l'intérieur d'une fonction, plus le CPU a passé du temps à l'intérieur de cette fonction. En utilisant cette mesure, les temps attribués aux fonctions sont propagés à leur appelant, permettant ainsi de connaître le coût réel de l'appel d'une fonction et la décomposition de ce coût sur les fonctions appelées par cette dernière. En fait, le temps d'exécution d'une fonction est le temps passé à exécuter du code appartenant à cette fonction plus la somme des temps attribués aux fonctions appelées par celle-ci.

La reconstruction de l'arbre d'appels est quelque peu plus complexe et implique l'utilisation de l'arbre d'appels statique[2] afin de s'assurer que tous les arcs possibles sont couverts, même ceux qui ne seront jamais parcourus. L'arbre statique est retrouvé en parcourant le fichier exécutable en y regardant le code objet. Ce faisant, on est en

[2] L'arbre des appels tels qu'il se trouve à même le code source de l'application.

mesure de retrouver, pour chaque fonction, les fonctions qui y sont appelées. Une fois cet arbre reconstruit, la table de hashing des arcs parcourus dynamiquement y est intégrée en attribuant un poids zéro aux arcs jamais parcourus. Les arcs parcourus auront le poids trouvé lors de l'exécution dynamique.

Avec l'arbre d'appels et les temps d'exécution combinés, GProf peut maintenant générer des données significatives au programmeur. Il demeure, toutefois, que GProf est limité dans sa capacité d'analyse. De fait, le taux d'échantillonnage utilisé est souvent beaucoup trop grossier (de l'ordre d'un centième de seconde) pour la rapidité des processeurs actuels (exécutant des instructions en quelques nanosecondes). Aussi, GProf compte sur l'effet statistique afin d'obtenir une gamme d'échantillons significatifs ; avec les dérives possibles que cette hypothèse implique.

1.3.2 Path Profiler

Contrairement à GProf qui se limite à analyser les temps d'appels des procédures et les différents arcs d'appels, Path Profiler [19] va plus loin en permettant, entre autres, de distinguer les différents chemins possibles à l'intérieur même des fonctions. Ainsi, les résultats obtenus sont non seulement répartis entres les diverses fonctions mais aussi aux divers blocs logiques formant une fonction. Ce faisant, cette technique supplante l'analyse de couverture conventionelle qui se limite souvent à compter le nombre de fois qu'une ligne de code est réellement exécutée par un programme.

Pour instrumenter l'application, Path Profiler utilise l'Executable Editing Library [28] afin d'éditer un exécutable binaire et d'y ajouter l'instrumentation nécessaire. En réalité, ce processus comporte plusieurs étapes, tel que décrit dans [19]. Premièrement, le programme est lu par l'éditeur binaire. Ensuite, la structure du programme est parcourue afin d'identifier les différents chemins d'exécution possibles. Chacun de ces chemins est instrumenté afin d'incrémenter un compteur qui lui est attribué. Cette étape implique l'ajout de code exécutable au code original. Ensuite, le tout est réécrit pour former un nouvel exécutable instrumenté.

Lors de l'exécution, les différents compteurs sont incrémentés et le tout est réécrit dans un fichier de données à la sortie du programme. Dans une version plus élaborée de ce schème décrite dans [17], les compteurs matériels du processeur sous-jacents sont enregistrés en combinaison avec les décomptes de chemin d'exécution. Ainsi, le code ajouté à l'application met les compteurs à zéro à l'entrée de la fonction et enregistre la valeur finale à la sortie de la fonction. Cette technique permet, par exemple, de pouvoir attribuer à chaque chemin d'exécution un certain nombre de fautes de cache en plus de pouvoir indiquer le nombre de fois qu'un chemin spécifique a été exécuté. Certains chemins pouvant, de cette manière, être identifiés comme étant des "hot paths".

Conjointement, une méthode a été développée pour permettre de lier un chemin d'exécution intra-fonction à un arc d'appel de fonction afin d'identifier les chemins suivis selon les arcs d'appels. En y conjuguant l'utilisation des compteurs de performance, Path Profiler peut ainsi attribuer, par exemple, un certain nombre de fautes de cache à un arc d'appel donné.

Bien que les résultats obtenus par Path Profiler soient beaucoup plus détaillés que ceux disponibles à travers l'utilisation de GProf ou d'un outil d'analyse de couverture, il n'en demeure pas moins que l'impact sur l'exécution des processus est énorme. De l'aveu même des auteurs de l'outil, Path Profiler ajoute en moyenne 30% au temps d'exécution de l'application observée. Selon les données disponibles, ce chiffre monte même jusqu'à 96%. Dans un environnement de production, ce type de surcoût est souvent inacceptable. De plus, il faut noter que cette technique se base sur l'édition binaire par EEL qui, encore de l'aveu de son auteur, n'est pas une technique tout à fait maitrisée. Aussi, Path Profiler a été implémenté sur SPARC seulement. Cette architecture étant peu répandue en comparaison avec l'architecture PC, l'usage de l'outil est restreint. De plus, PathProfiler n'étant pas un logiciel libre, les auteurs de Path Profiler ne peuvent utiliser à profit la main d'oeuvre disponible dans la communauté des logiciels libres afin de perfectionner leurs outils.

1.3.3 Quantify

À l'inverse de Path Profiler qui est surtout un projet de recherche, Quantify [1] est un produit commercial mature et qui bénéficie d'un certain succès grand public, surtout de par sa disponibilité pour le système d'exploitation Windows NT sur le PC. Bien que le fonctionnement interne de Quantify ne soit pas documenté publiquement, un des auteurs de Path Profiler, James Larus, confirme dans [17] à travers une communication privée avec le fabriquant de Quantify que le fonctionnement interne de Quantify est très similaire à celui de Path Profiler. Ainsi, les observations faites pour Path Profiler devraient s'appliquer aussi Quantify sous certaines réserves. En effet, bien que les auteurs de Path Profiler aient effectué des mesures de performance scientifiquement satisfaisantes, rien de tel n'a été fait pour Quantify et on ne peut donc tirer aucune conclusion concernant l'utilisabilité d'applications instrumentées par Quantify en milieu de production.

1.3.4 Digital Continuous Profiling Infrastrucutre

Au lieu d'observer les applications une à une, comme le font les outils vus plus haut, DCPI [22] échantillonne l'ensemble des logiciels s'exécutant sur le CPU. Ce faisant, il est en mesure d'observer le comportement des diverses composantes logicielles du système sans avoir à en instrumenter aucune. Pour y arriver, DCPI tire profit de l'existence de compteurs de performance matériels sur les CPUs récents et obtient, de ce fait, des résultats très détaillés en ce qui concerne les temps d'exécution, ainsi que les causes des divers ralentissements. Comme avec Path Profiler, cette analyse peut s'effectuer aussi bien au niveau des fonctions qu'aux niveau des instructions à l'intérieur même des fonctions. Toutefois, DCPI ne permet pas de reconstruire les divers arbres d'appels comme le font les outils vus plus haut.

Afin d'obtenir la fonctionnalité désignée, DCPI est composé de plusieurs blocs logiciels : un pilote noyau, un daemon et les outils d'analyses de données. Le tout repose sur l'utilisation des compteurs de performance qui peuvent être programmés afin de générer une interruption lorsqu'un certain nombre d'événements d'un type donné se produisent. Ainsi, on peut programmer ces compteurs pour générer une interruption chaque x fautes de cache. Dans DCPI, le pilote noyau programme ces compteurs avec des valeurs de décompte aléatoires afin de s'assurer que les échantillons soient uniformément répartis. Lorsqu'une de ces interruptions se produit, le pilote enregistre le pointeur de programme où s'est produite l'interruption, le type d'événement qui était compté par le compteur matériel (le nombre de fautes de cache, par exemple) et l'identificateur du processus actuel. Ces données sont entreposées par le pilote qui signale le daemon chaque fois que le tampon de données principal est plein. Ce faisant, le daemon parcourt chacune des entrées dans le tampon et trouve l'image binaire correspondant au pointeur de programme échantillonné, ainsi que l'emplacement correspondant au compteur de programme à l'intérieur de l'image. Ces données sont préservées par le daemon conjointement avec l'événement observé dans une table de hashing. Chaque quelques temps, les données de cette table sont écrites sur disque pour fins d'analyse utltérieure. En tout et pour tout, ce système est très efficace et consomme très peu de ressources. En fait, l'impact de l'utilisation de ce système est inférieur à 5% lorsque comparé au roulement du système en l'absence de DCPI.

Utilisant les données recueillies durant l'échantillonnage, les outils d'analyse permettront d'identifier pour chaque instruction les éléments d'information suivants :

- La fréquence, qui sera proportionnelle au nombre de fois que l'instruction a été exécutée.

- Le CPI (nombre de Cycles Par Instructions), permettant de connaître le nombre de cycles que le CPU a pris en moyenne pour exécuter l'instruction.
- Un ensemble de coupables, permettant d'identifier les raisons des différentes latences encourues.

Ces informations seront fournies à l'usager selon l'outil DCPI qu'il utilise et lui permettront d'identifier non seulement les fonctions qui sont problématiques, mais aussi les instructions à l'intérieur de ces fonctions qui ralentissent l'exécution de l'application, ainsi que la ou les causes de ce ralentissement.

Notons que l'analyse effectuée par DCPI est valide si et seulement si le CPU exécute les instructions dans leur ordre d'origine. De fait, plusieurs processeurs récents réordonnent les instructions qui leurs sont fournies afin d'optimiser leur exécution selon le contexte actuel. Ces optimisations font en sorte que la corrélation entre les interruptions dues au décompte des compteurs de performance et le pointeur de programme n'est plus possible. Les auteurs de DCPI ont discuté de ce problème dans [23] et proposent des amendements au matériel, ainsi qu'une technique différente que celle utilisée par DCPI afin de pallier aux problèmes causés par le réordonnancement des instructions. Ceci dit, ces amendements ne sont disponibles que pour le système expérimental étudié par les auteurs et ne semblent pas faire partie des processeurs disponibles sur le marché. Étant donné que la grande partie des processeurs modernes utilisent des méthodes de réordonnancement[3], il faudra attendre que les techniques présentées dans [23], ou des techniques similaires, soient adoptées par les concepteurs de CPUs afin de pouvoir utiliser DCPI, ou un logiciel similaire, de manière fiable.

1.3.5 Morph

Tous les outils vus plus haut nécessitent l'intervention d'un programmeur afin de mener à terme le processus d'optimisation. Morph [25], par contre, définit un système d'optimisation complètement automatisé. C'est à dire qu'il n'est pas nécessaire à un être humain d'intervenir ou même d'analyser les données recueillies pour que le système, ou une de ses parties, soit optimisé. Ainsi, la collecte des données, leur analyse et la modification des applications est complètement prise en charge par le système Morph. En ce sens, Morph est unique en son genre. Pour ce faire, Morph est composé des modules logiciels suivants : le moniteur, le gestionnaire et l'éditeur. Chacune de ces composantes s'occupe d'automatiser une partie ou une autre du processus d'optimisation.

Le moniteur se charge de la collecte des données et de leur stockage sur disque pour analyse future. Pour ce faire, la fonction de traitement de l'interruption de l'horloge système a été modifiée afin d'enregistrer la valeur du compteur de programme telle qu'elle était au moment de l'interruption. Ceci est analogue au fonctionnement de GProf du fait que l'importance accordée à une instruction quelconque dépend du

[3]Dans la famille des 386, tous les processeurs après le PentiumPro effectuent une forme où une autre de réordonnancement.

nombre de fois que celle-ci s'exécutait au moment de l'échantillonnage. Afin de pouvoir corréler le compteur de programme à un programme quelconque et à une adresse dans ce programme, d'autres informations sont aussi nécessaires et, conséquement, d'autres parties du noyau doivent être instrumentées afin de pouvoir extraire ces informations. De fait, les fonctions suivantes sont modifiées :

– *exec()*, pour l'information concernant l'initialisation et afin de sélectionner dynamiquement si un processus doit être observé ou non.
– *mmap()*, afin de connaître les adresses réelles des fichiers exécutables et des librairies partagées.
– *exit()*, afin de connaître le moment auquel le processus a cessé d'exister.

Avec 8 octets par échantillon, un taux d'interruption horloge de 1024 Hz et un tampon de données de 256 KB, il est nécessaire de vider le tampon environ toutes les 30 secondes. La gestion du tampon est faite par un pilote noyau et la tâche de vider le tampon et d'en écrire le contenu sur disque est prise en charge par un daemon usager.

Notons que, tel qu'il est indiqué dans [25], il serait possible d'utiliser DCPI afin de se charger de la collecte des données à la place du moniteur de Morph. Ceci permettrait une plus grande résolution et permettrait d'éviter les faiblesses de la technique utilisée par Morph ; telles que la possibilité d'aberrations d'échantillonnage étant donné l'utilisation de l'horloge système.

Chaque quelques temps, le gestionnaire de Morph s'exécutera et sera chargé du traitement et de la gestion des données collectées par le moniteur. Le gestionnaire a, en fait, deux fonctions. La première est de décider du moment opportun pour lancer une optimisation. La seconde est de transformer les valeurs du compteur de programme collectées durant l'échantillonnage en une forme plus compacte qui puisse être facilement utilisée pour les optimisations. L'équipe travaillant sur Morph s'est surtout concentrée sur le deuxième aspect du gestionnaire. Donc, seule la seconde fonction du gestionnaire sera discutée en détails.

Avant de continuer, il est important de noter que les concepteurs de Morph ont pris comme hypothèse que le code source original des applications s'exécutant sur le système n'est pas disponible, mais que le code de référence est en fait ce qu'ils appellent la "forme intermédiaire". Cette forme intermédiaire aurait l'avantage de ne pas nécessiter la publication du code source tout en permettant l'optimisation du programme observé. Ainsi, lorsque le gestionnaire de Morph analysera les données collectées, c'est aux fichiers de la forme intermédiaire, correspondant aux exécutables observés, qu'il fera référence et non aux fichiers sources originaux.

Le gestionnaire parcourt donc le fichier de données et associe à chaque valeur de compteur de programme un bloc de code à l'intérieur d'un fichier de forme intermédiaire. Le gestionnaire conserve ses données d'exécution en exécution et bâtit un profil pour chaque programme observé. Ce profil sera constamment mis à jour afin qu'il reflète l'utilisation courante du programme en question.

À son tour, l'éditeur se servira du profil génèré par le gestionnaire afin de diriger l'optimisation des applications. Cette optimisation utilise comme entrées le profil et le fichier de format intermédiaire désigné. Elle générera un nouvel exécutable qui remplacera celui qui avait été observé. En fait, l'éditeur a recours à trois types d'optimisations qui sont exécutées dans l'ordre suivant :

1. **L'alignement de branchement (branch alignment).** Cette optimisation consiste à réorganiser les blocs du programme afin de minimiser les aléas de contrôle, les erreurs de prédiction de sauts et les chargements inutiles tout en augmentant le degré de localité dans la cache d'instruction.

2. **L'élimination de trous (fluff removal).** Cette optimisation vise à augmenter la compacité du code en déplaçant le code inutilisé vers la fin de l'exécutable.

3. **Le positionnement des procédures (procedure layout).** Ce positionnement vise à placer les procédures en mémoire de manière à réduire les pénalités de cache dues aux limites de tailles et aux conflits d'adresses des instructions.

Les résultats présentés montrent que Morph a moins de 1% d'impact sur le système observé tout en permettant des optimisations très similaires à celles obtenues en utilisant des outils d'optimisation conventionnels. Ces résultats sont satisfaisants étant donné que, contrairement, aux autres outils, Morph est entièrement automatisé.

Toutefois, le système d'optimisation implémenté par Morph est difficilement applicable à grande échelle étant donné qu'il se base sur la disponibilité des différentes applications sous forme intermédiaire. Comme l'indiquent les auteurs de Morph, il n'est pas envisageable que les fabriquants de logiciels commerciaux soient enclins à distribuer leurs applications sous de tels formats. Notons, par contre, que ce type de problème ne se pose pas avec les logiciels distribués sous une license libre[4], tel que la GNU GPL [2], étant donné que le code source original de ces logiciels est disponible à tous. Dans un contexte pareil, il serait intéressant de voir un système semblable à Morph être implémenté afin d'optimiser constamment des logiciels libres.

1.4 Analyse globale

Tel qu'indiqué précédemment, l'analyse globale permet de recueillir des informations statistiques concernant l'ensemble des logiciels composants le système. La plupart des systèmes d'exploitation modernes incorporent la collecte de données globales sous une forme ou une autre et exportent des interfaces permettant d'accéder à ces données. Il est d'autant plus facile d'incorporer cette fonctionnalité que l'instrumentation et les calculs nécessaires sont simples à programmer et utilisent très peu de ressources. Voyons maintenant deux exemples de systèmes et d'outils d'analyse globale.

1.4.1 Windows NT Performance Monitor

Windows NT vient équipé avec un logiciel nommé Performance Monitor (PM). Cet outil permet d'avoir accès à des compteurs maintenus par le noyau de WinNT, tel que décrit dans [34]. Ces compteurs sont classés par catégories de services noyau et sont accessibles à travers un appel système. Lorsqu'un événement d'un certain type se

[4] C'est le cas de la plupart des logiciels formant le système GNU/Linux.

produit, WinNT incrémente le compteur qui lui est associé. La valeur de ce compteur peut ensuite être lue et affichée graphiquement par le PM. L'usager peut sélectionner les compteurs qu'il désire visualiser et le PM se chargera d'y accéder et des les afficher convenablement.

Bien que le décompte des événements se fasse en temps-réel au fur et à mesure qu'ils se produisent, la lecture des compteurs de performance se fait à la vitesse choisie par l'usager. Ainsi, l'usager peut choisir un taux de rafraîchissement plus ou moins rapide. La fréquence de lecture des compteurs sera proportionnelle à la vitesse de rafraichissement. Plus l'usager désire suivre la performance du système avec détails, plus il augmente la fréquence de rafraîchissement, plus il y a de lectures des compteurs et plus l'impact sur le système est grand. Malgré le fait que cet outil soit efficace pour obtenir un aperçu du fonctionnement du système, il est impossible de l'utiliser pour obtenir une image complète du fonctionnement du système.

1.4.2 Unix *ps*, *top* et */proc*

De manière similaire, les systèmes Unix modernes sont équipés d'une implémentation ou d'une autre de la commande *ps*. Cette commande permet de visualiser l'utilisation que font les divers processus des ressources du système. *top* permet d'afficher les mêmes informations mais, contrairement à *ps*, *top* rafraichit régulièrement l'affichage et permet à l'usager d'avoir une information constamment mise à jour. Ces deux outils utilisent les mêmes services du système afin d'obtenir les informations qu'ils affichent. Récemment, ceci implique l'utilisation du répertoire */proc* qui contient des fichiers virtuels dont le contenu est créé dynamiquement par le système d'exploitation lorsqu'ils sont accédés.

Peu importe la méthode d'accès utilisée ou l'outil d'affichage, la méthode de collecte de l'information est très semblable d'un système Unix à un autre. En fait, la plupart de ces systèmes utilisent l'interruption de l'horloge afin d'échantillonner l'identificateur du processus actuel et d'attribuer au processus correspondant le temps de traitement depuis la dernière interruption de l'horloge. Cette technique a ses limites puisqu'il est impossible de prendre en compte les changements de contexte et autres modifications aux ressources du système qui se sont produites entre les interruptions. Les valeurs obtenues peuvent, donc, être trompeuses. Ceci est d'autant plus vrai que les fréquences d'horloge des processeurs modernes et, conséquemment, la rapidité à laquelle ils exécutent les instructions, sont plus élevées que les fréquences d'interruption de l'horloge par plusieurs ordres de grandeur.

Bien que les résultats puissent être améliorés en utilisant un échantillonnage aléatoire, tel que décrit dans [30], on n'obtient qu'une vision statistique du système qui décrit mal le système dans toute sa complexité.

1.5 Analyse dynamique

Palliant à certaines lacunes présentes dans les outils vus précédemment, les outils d'analyse dynamique permettent d'observer le comportement dynamique d'un système à l'intérieur d'un intervalle donné. Ce faisant, l'usager peut identifier toutes les transitions ayant eu lieu, leur raison d'être et leurs conséquences sur le restant du système. Car, bien qu'il soit possible d'améliorer la performance d'une application individuelle ou l'ensemble des applications, la plupart des ralentissements sont dus à des conflits de partage de ressources, à des problèmes de synchronisation, à l'ignorance du concepteur de logiciel du traitement fait par le système pour desservir les appels systèmes, ou à d'autres problèmes structurels qui ne peuvent pas être solutionnés par une quelconque optimisation du code. Si, par exemple, un logiciel est souvent en attente d'entrées/sorties, rares sont les logiciels d'analyse locale ou d'analyse globale qui permettront de l'identifier. Un logiciel d'analyse dynamique, cependant, permettra au programmeur d'identifier ce problème facilement.

Essentiellement, l'analyse dynamique est basée sur l'instrumentation du noyau du système d'exploitation. À certains endroits stratégiques, des instructions de traçage sont ajoutées au noyau afin d'être appelées lorsque l'événement désigné se produit. Lorsque ces instructions sont appelées, elles conservent des données concernant l'événement qui s'est produit dans un tampon de données dans l'ordre d'arrivée des événements afin d'être en mesure de reconstruire le comportement dynamique du système. Il existe deux catégories de systèmes d'exploitation pour lesquels des outils d'analyse dynamique sont disponibles : les systèmes d'exploitation à usages généraux et les systèmes d'exploitation temps-réel et/ou pour systèmes embarqués.

Dans la catégorie des outils d'analyse pour systèmes d'exploitation à usages généraux, on retrouve deux outils répandus : l'outil de traçage pour AIX d'IBM [3] et les outils de Sun basés sur le format TNF[5] [4]. L'outil de traçage pour AIX a la particularité d'être facilement extensible. De fait, pour ajouter de nouveaux événements, il s'agit seulement de modifier le fichier */etc/trcfmt*. Par contre, aucun moyen n'est disponible pour afficher graphiquement le comportement du système avec cet outil hormis quelques projets de recherche qui utilisent des données extraites par le traceur pour les afficher [5]. Malgré la disponibilité d'outils de visulation parmi les outils de Sun, les graphes affichés sont difficiles à suivre et ne se comparent pas favorablement aux outils de visualisation disponibles pour les divers systèmes d'exploitation embarqués/temps-réels.

Toutefois, les limitations principales de ces outils sont : leur caractère non-extensible, l'impossibilité de modifier leur code source, leur incapacité de tracer les tâches temps-réels et le manque de détails des informations qu'ils fournissent.

Les outils qui composent LTT, que nous verrons plus loin, sont aussi des outils d'analyse dynamique pour un système d'exploitation à usage général, Linux, mais ont été bâtis avec une optique différente. Ils présentent des résultats qui sont uniques et comblent les lacunes des autres outils de trace de la même catégorie.

Dans la catégorie des outils d'analyse pour systèmes d'exploitation temps-réels et/ou pour systèmes embarqués, il existe une très longue liste d'outils d'analyse dynamique de qualité très variable. Le plus connu d'entre ces outils est sans doute Wind-

[5]Trace Normal Form.

View de WindRiver [6]. Cet outil permet d'analyser et de visualiser le comportement dynamique de systèmes basés sur le système d'exploitation VxWorks de WindRiver. C'est, de loin, l'outil d'analyse le plus recherché, permettant une configuration gaphique du processus de traçage, ainsi que différents niveaux de configuration de la visualisation de la trace. QNX, un autre fabriquant de systèmes d'exploitation temps-réels, possède aussi un outil similaire, DejaView [7]. D'autres outils se trouvent aussi dans cette catégorie, par exemple : TimeScan d'Etnus [8], Trace32 de Lauterbach [9], TimeTrace de TimeSys [10] et le Hyperkernel Trace Utility de Nematron [11].

Ces outils, néanmoins, sont limités à l'affichage du comportement dynamique du système et ne sont pas conçus de manière à permettre l'extraction d'informations concernant la performance du système. Aussi, ces outils sont des produits commerciaux dont le code source est inaccessible et ne peuvent donc pas être améliorés ou modifiés par leurs usagers.

1.6 Simulation de systèmes

Un autre moyen de récolter des informations concernant un système d'exploitation et les applications qu'il roule est de simuler le matériel sous-jacent. Ainsi, le système opère tel que prévu mais peut être observé dans un environnement contrôlé. C'est cette approche qui est suivie par SimOS [32]. SimOS simule une machine MIPS complète servant à rouler une version non-modifiée du système d'exploitation IRIX. Le matériel simulé inclut les composantes suivantes : processeur (CPU), unité de gestion de mémoire (MMU), caches, systèmes de mémoire, disques SCSI, cartes Ethernet, minuterie (clock) et consoles. Pour chaque type d'opération effectuée sur le matériel, il est possible d'attacher des opérations de traitements qui permettent d'extraire des informations concernant le logiciel qui s'exécute. De cette manière, il est, par exemple, possible de connaître le processus auquel le système d'exploitation attribue le processeur en s'attachant aux instructions propres aux opérations de changement de contexte. Cette approche est suivie à l'échelle des autres composantes du système afin d'extraire des informations très précises sur la dynamique du système. Pour cette même raison, SimOS a été utilisé à plusieurs reprises afin d'optimiser plusieurs programmes commerciaux et applications scientifiques.

Bien que les résultats obtenus par SimOS soient intéressants, il n'en demeure pas moins que SimOS reste et demeure un système de simulation qui ne roule pas à la même vitesse que le matériel réel. De plus, cette approche se généralise difficilement à d'autres systèmes basés sur d'autres processeurs ou même à des systèmes MIPS différents de ceux de SGI. Pour chaque configuration, il faudrait avoir un simulateur qui lui est particulier. Au delà de l'intérêt scientifique, il est difficile de voir une justification suffisante qui pourrait motiver une entreprise ou un groupe d'individus à étendre cette approche à toutes les configurations matérielles possibles. Il est beaucoup plus réaliste de voir déployer une infrastructure de collecte de données intégrée à un système d'exploitation qui roule sur plusieurs configurations matérielles, tel Linux.

Chapitre 2

Concepts de base

Afin de comprendre l'architecture de LTT et les compromis d'implémentation qui y sont trouvés, il est nécessaire de discuter de l'architecture des systèmes pour lesquels LTT a été conçu, Linux et RTAI.

2.1 Architecture de Linux

Linux est un système d'exploitation de type Unix. Le travail sur Linux a débuté en 1991 avec le lancement par Linus Torvalds. Depuis, un grand nombre de personnes ont contribué au projet et Linux est maintenant un système d'exploitation très mature dont la part de marché représente à peu près 5% des ordinateurs personnels et près de 30% des serveurs.

Bien que l'architecture des systèmes Unix conventionnels soit relativement bien connue, voir [18, 35], l'architecture de Linux est quelque peu différente. C'est pour

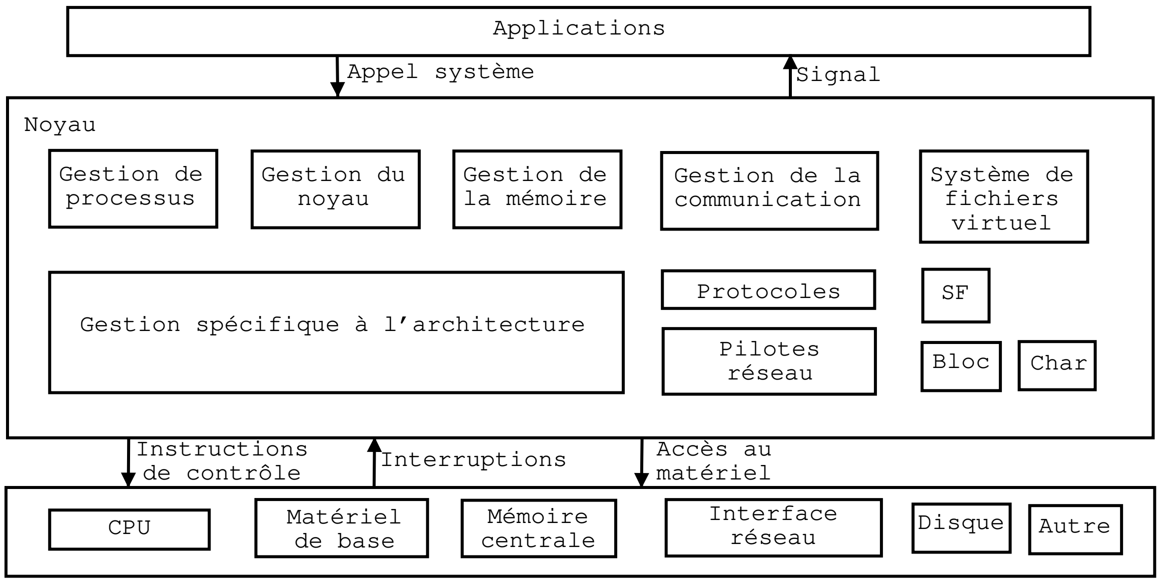


FIG. 2.1 – Architecture de Linux.

cette raison que certains ouvrages couvrent spécifiquement l'architecture de Linux, tels que [24, 33, 29, 20].

L'architecture générale de Linux est illustrée à la figure 2.1. Afin de couvrir adéquatement chacune des composantes du noyau, les sections suivantes présentent le détail des composantes importantes, de leurs interactions entre elles et de leurs interactions avec les autres composantes du système.

2.1.1 Gestion de processus

L'unité de gestion de processus s'occupe de réagir adéquatement aux appels systèmes et aux signaux qui ont trait aux processus s'exécutant sur le système afin de conserver un comportement cohérent des processus. En soi, un processus est une abstraction logicielle puisque le matériel ne reconnaît les tâches que sous leur forme la plus simpliste, soit une pile et des registres uniques. C'est l'unité de gestion des processus qui permettra de maintenir cette abstraction tout en permettant des manipulations et des comportements similaires aux systèmes Unix conventionnels.

À l'intérieur du noyau, chaque processus est représenté par une structure de type `task_struct`. C'est cette structure, ou un pointeur à celle-ci, qui permettra d'identifier un processus ou un autre et qui sera utilisée pour opérer sur le processus correspondant. De l'extérieur, c'est à dire en milieu usager, chaque processus est identifié à l'aide d'un identificateur de processus (PID). Lorsqu'une application désire agir sur un processus, c'est ce PID qui est utilisé ; les structures internes du noyau étant complètement invisibles aux applications usagers. C'est ce même PID qui apparaît lorsqu'on utilise une commande comme *ps*.

Les sous-sections suivantes discutent des tâches principales de l'unité de gestion de processus.

2.1.1.1 Création et destruction de processus

Lorsque les demandes de création et de destruction de processus parviennent au noyau, l'unité de gestion de processus s'occupe d'initialiser les structures nécessaires ou, à l'inverse, de libérer les ressources qui appartiennent au processus.

La création de processus se fait lorsque les fonctions Unix conventionnelles *fork()* et *vfork()* sont appellées. Aussi, pour faciliter l'implémentation de processus poids-léger (threads), Linux permet un autre appel, *clone()*. De manière interne, ces trois fonctions utilisent la même primitive, *do_fork()*. C'est cette fonction qui se chargera d'allouer les espaces mémoires nécessaires, d'initialiser des registres d'exécution, de trouver un PID au processus et d'insérer le processus dans la liste des processus existants, entre autres choses. Dans Linux, il est aussi possible de créer des processus de type noyau (kernel threads). Ces processus s'exécutent dans l'espace d'adressage du noyau et sont principalement utilisés pour paralléliser certaines opérations du noyau.

La destruction de processus se fait ordinairement lorsque l'appel *exit()* est fait, mais peut aussi résulter de l'envoi d'un signal au processus. Dans tous les cas, c'est la primitive *do_exit()* qui sera chargée de libérer les ressources acquises par le processus tout en s'assurant que toutes les références faites à ce processus par d'autres structures du noyau soient enlevées.

2.1.1.2 Ordonnancement

L'ordonnancement des processus inclut l'algorithme d'ordonnancement, ainsi que les fonctions influençant cet ordonnancement.

L'algorithme d'ordonnancement de Linux peut être appelé dans deux circonstances. Premièrement, lorsque le processus courant doit attendre pour une ressource ou une autre ou lorsqu'il en fait la demande. Deuxièmement, lorsque le temps alloué au processus actuel est échu ou lorsqu'un processus de plus grande priorité devient actif. Dans tous les cas, c'est la primitive *schedule()* qui se charge de réordonnancer les processus. Cette fonction est complexe et se charge de plusieurs autres tâches qui doivent être proprement accomplies afin que l'ordonnancement et les changements de contexte soient adéquatement effectués. L'algorithme d'ordonnancement implémenté dans *schedule()* est de type round-robin[1] avec niveaux de priorité et vieillissement. Ainsi, plus de temps passe entre chaque instance d'ordonnancement d'un processus, plus ce processus sera promu de priorité (vieilli) afin qu'il puisse finalement avoir une part du temps CPU. En réalité, Linux permet d'autres algorithmes d'ordonnancement afin de se conformer aux politiques d'ordonnancement de Posix, mais ces politiques sont très peu utilisées.

Voici quelques exemples d'appels qui peuvent influencer l'ordonnancement des processus :

- *nice()*, modifie la priorité de base du processus appellant.
- *getpriority()*, retourne la priorité du processus designé.
- *setpriority()*, modifie la priorité du processus designé.
- *sched_getscheduler()*, retourne le type d'ordonnancement appliqué au processus désigné.
- *sched_setscheduler()*, modifie le type d'ordonnancement appliqué au processus désigné.
- *sched_getparam()*, retourne les paramètres d'ordonnancement du processus désigné.
- *sched_setparam()*, modifie les paramètres d'ordonnancement du processus dési-

[1]Ne pas confondre avec la politique d'ordonnancement *SCHED_RR* implémenté dans Linux qui s'applique aux processus "temps-réels".

gné.

- *sched_yield()*, permet au processus appellant d'abandonner le temps CPU lui étant alloué.

2.1.1.3 Traitement de signaux

Les signaux permettent au noyau d'informer les processus du fait qu'un événement important le concernant a eu lieu. De base, il y a 31 signaux, numérotés de 1 à 31. Chacun des ces signaux correspond à un événement et peut posséder des particularités de traitement. Un processus peut choisir de réagir aux signaux de trois manières différentes :

- En les ignorant. Ce faisant, le signal ignoré n'a aucun effet. Certains signaux ne peuvent, cependant, pas être ignorés.
- En laissant la procédure par défaut s'exécuter. Pour la plupart des signaux ceci voudra dire une interruption de l'exécution du processus par un moyen ou un autre.
- En attrapant le signal. Ceci permet au processus de réagir au signal d'une manière adéquate. Notons, toutefois, que certains signaux, tel que *SIGKILL* ne peuvent être attrapés par le processus et forcent une réaction de la part du système d'exploitation.

Lorsqu'un signal est envoyé, soit par une composante du noyau ou un autre processus, c'est la primitive *send_sig_info()* qui finit par être appelée afin de livrer le signal à l'application désignée. Tout le long du chemin de propagation des signaux, les paramètres de configuration de signaux initialisés pour le processus recevant le signal sont pris en compte.

Voici quelques exemples d'appels qui peuvent influencer le traitement des signaux :

- *kill()*, envoie un signal donné au processus désigné.
- *sigaction()*, spécifie les actions à prendre en cas de signal.
- *sigpending()*, permet d'examiner les signaux non livrés.
- *siprocmask()*, modifie l'ensemble des signaux masqués.
- *sigsuspend()*, suspend l'exécution du processus jusqu'à ce qu'un signal lui soit livré.

2.1.2 Gestion de la mémoire

La gestion de la mémoire s'occupe de départager la mémoire physique et la mémoire virtuelle disponible entre les divers processus et les diverses composantes du noyau. La gestion de la mémoire dans Linux se base sur l'usage de tables de pagination telles qu'on en retrouve sur la plupart des architectures modernes. Ces tables permettent d'isoler l'espace d'adressage des processus tout en permettant au processus de voir un seul type de cartographie mémoire. Dans cette configuration, les processus détiennent le premier 3GB de mémoire virtuelle, sur un système 32bits, et le système d'exploitation se trouve dans les 1GB supérieurs. Cette division peut être modifiée sur les systèmes ayant une large mémoire physique afin d'en permettre un usage plus optimal.

C'est aussi grâce à l'unité de gestion de mémoire que des fichiers peuvent être cartographiés en mémoire (memory mapped). Cette technique permet d'accéder au contenu d'un fichier comme s'il s'agissait d'un tableau de données en mémoire, permettant de ce fait un accès rapide et transparent.

L'unité de gestion de mémoire se chargera de la pagination en enlevant de la mémoire physique les pages les moins souvent utilisées. Sous Linux, elle met aussi en place un système de cache fichier afin que les fichiers, ou leurs parties, récemment utilisés, soient disponibles en mémoire pour accélérer les lectures et écritures qui en sont faites.

2.1.3 Gestion de la communication

La communication inter-processus est une composante importante de tout système d'exploitation moderne. S'inspirant des autres systèmes Unix, Linux possède des primitives de communication très similaires à celles trouvées dans les systèmes Unix. Ces systèmes de communication sont : les pipes, les FIFOs, les primitives IPC Système V et les sockets. Chacun de ces services de communication nécessiterait une discussion qui dépasse le cadre du travail actuel, mais l'essentiel est de souligner leur disponibilité dans Linux puisque certains sont pertinents au travail d'instrumentation effectué pour LTT.

2.1.4 Système de fichiers virtuel

Le système de fichiers virtuel offre aux applications une vision unifiée de l'arborescence des fichiers et des répertoires dans Linux. Cette technique permet d'avoir une racine unifiée et permet de rendre les accès aux divers fichiers transparents aux protocoles, structures et matériels sous-jacents. De la sorte, il n'est pas nécessaire qu'une application sache qu'un fichier se trouve sur disque SCSI dans un système de fichier ext3 ou sur un disque IDE dans un système de fichier DOS. Le système de fichiers virtuel permet aussi d'abstraire le matériel puisque toute composante matérielle est vue comme une entrée dans le répertoire */dev*. Ainsi, chaque opération fichier usuelle, telle que *open()*, *close()*, *read()*, *write()*, passe par le système de fichier virtuel qui se charge d'appeler les fonctions sous-jacentes en fonction du système de fichiers et du matériel.

2.1.5 Gestion spécifique à l'architecture

Malgré le fait que Linux roule sur plusieurs architectures, les algorithmes importants sont tous implémentés de manière indépendante du matériel. Seules certaines opérations spéciales, telles les changements de contexte ou les manipulations de bas niveau, ont besoin d'être codées spécifiquement pour chaque matériel différent. C'est le rôle de la gestion spécifique à l'architecture de s'assurer que les requêtes de haut niveau soient converties en des instructions réelles au matériel sous-jacent. C'est aussi cette unité qui se charge de faire parvenir aux plus hauts niveaux du noyau les événements matériels de bas niveau tels que les interruptions.

Pour chaque architecture supportée par Linux, se trouve un répertoire contenant le code spécifique à cette architecture à même le code source du noyau. Lorsque le noyau est configuré pour la compilation, le répertoire adéquat sera choisi. En plus des interfaces entre les abstractions du noyau et le matériel, ce répertoire contient aussi le code de démarrage de Linux qui permet de charger le noyau sur l'architecture désignée.

2.1.6 Protocoles de communication

Les protocoles de communication permettent à Linux de communiquer avec le monde extérieur à travers un réseau quelconque. La discussion des protocoles supportés par Linux et leur implémentation dépasse le cadre de ce travail, mais il suffit de dire que l'unité de gestion de communication du noyau fait appel à ces divers protocoles afin de permettre aux applications usagers de communiquer avec d'autres applications roulant sur d'autres machines sur le réseau.

2.1.7 Systèmes de fichiers

Les systèmes de fichiers, dénotés par "SF" sur la figure 2.1, implémentent une structure spécifique sur un média de conservation physique. Un même média physique peut être utilisé par divers systèmes de fichiers différents, chacun ayant sa propre structure. L'existence de plusieurs types de systèmes de fichiers est surtout due à des raisons historiques, mais aussi pour des raisons de divergences d'opinions sur le meilleur moyen d'implémenter un système de fichier. Sur Linux, le système de fichier le plus utilisé est *ext2*, bien que ce système soit appellé à être remplacé par un des divers système de fichiers journalisé faits pour Linux. D'autres systèmes, comme MSDOS et VFAT, sont préservés pour maintenir la compatibilité avec d'autres systèmes d'exploitation qui ne reconnaissent pas *ext2*. Ce sont les divers systèmes de fichiers qui sont chargés de gérer les opérations de bas niveau effectuées par le système de fichiers virtuel.

2.1.8 Pilotes de périphériques

Les pilotes de périphériques sont des composantes logicielles permettant de contrôler un certain type de matériel. Par exemple, pour contrôler le port série du PC, il faut un pilote de périphérique correspondant. Dans Linux, il existe trois grandes catégories de pilotes de périphériques : pour périphériques à accès par blocs, pour périphériques à accès par caractère et pour cartes réseaux. Les périphériques de type bloc sont pour la plupart des disques. La catégorie des périphériques à accès par caractères est de loin la plus peuplée puisque la plupart des périphériques qui ne sont pas un disque en font partie.

Chaque type de pilote de périhpérique spécifie une interface spécifique à être implémentée, pour que les composantes du noyau puissent l'utiliser, et utilise des interfaces spécifiques à son tour pour communiquer avec le noyau.

2.2 Architecture de RTAI

Le Dipartimento di Ingegneria Aerospaziale, Polytecnico di Milano (DIAPM) Real-Time Application Interface [12] tient ses racines de recherches faites par Paolo Montegazza dans les années '80 afin d'obtenir des performances temps-réels sous DOS. Dans le temps, le système était construit en programme TSR[2]. Avec l'arrivée de systèmes d'exploitation 32-bits, une demande de plus en plus grande de passer à ce type de système fut ressentie. C'est dans ce contexte que le passage à Linux fut proposé. Toutefois, la série de noyau 2.0.x étant jugée trop peu mature pour la fonctionnalité nécessaire, l'idée fut mise au rancart. Lorsque RTLinux [13] fut porté avec succès à Linux 2.0.x, l'idée ressurgit et une variante DIAPM de RTLinux fut créée. Lorsque la série Linux 2.2.x fut atteinte, Linux était devenu assez mature pour incorporer facilement les concepts mis de l'avant par Paolo Montegazza. RTAI naquit de la combinaison de ces concepts avec Linux 2.2.x. RTAI est aujourd'hui largement répandu et est souvent préféré à RTLinux par les concepteurs travaillant dans l'industrie [27]. L'architecture de RTAI a déjà été couverte dans [31, 21], voyons-en les aspects les plus importants.

Principalement, RTAI insère une couche sous Linux qui prend le contrôle de la gestion des interruptions et en redirige le flot vers des modules qui peuvent assurer un temps de réaction déterministe. L'architecture de RTAI est illustrée à la figure 2.2. Le noyau est modifié (patched) pour y insérer une couche d'abstraction permettant

[2]Terminate and Stay Resident.

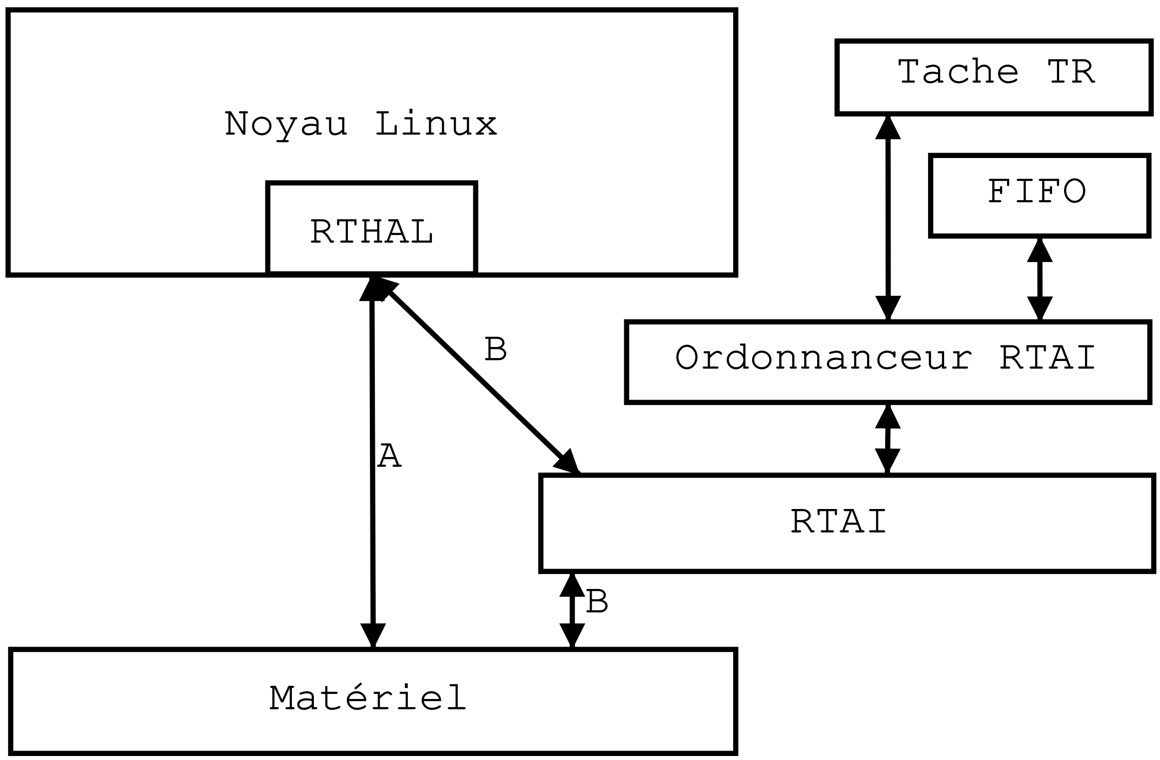


Fig. 2.2 – Architecture de RTAI.

d'aiguiller la gestion des interruptions. Cette couche est la Real-Time Hardware Abstraction Layer ou RTHAL. Lorsque RTAI n'est pas encore chargé, le flot de contrôle des interruptions est tel qu'indiqué par la flèche A. Lorsque RTAI est chargé, ce flot devient tel qu'indiqué par les flèches B. On peut ainsi voir qu'au chargement de RTAI, toutes les décisions concernant les interruptions passent par RTAI. C'est ce qui permet à RTAI de gérer proprement les interruptions et de les aiguiller de manière à obtenir leur livraison en temps-réel.

Une fois qu'il aura pris le contrôle, RTAI offre à Linux une gestion transparente des interruptions. C'est à dire que Linux ne verra aucune différence sur ses activités usuelles de contrôle d'interruptions. En plus, RTAI offre un certain nombre de services et d'abstractions aux tâches temps-réels. Ceux-ci incluent l'allocation d'interruption et le contrôle des minuteries (timers).

Afin d'offrir à Linux un contrôle virtuel des interruptions, RTAI implémente des versions de remplacement du couple *cli()/sti()*. Ces remplacements sont en fait des fonctions de RTAI qui changeront la valeur d'une variable d'état qui indiquera si Linux ignore présentement les interruptions (*cli()*) ou s'il les accepte (*sti()*). Puisqu'en temps normal toute interruption latente est activée au moment d'un *sti()*, RTAI implémente un mécanisme d'enregistrement des interruptions qui prend en compte toutes les interruptions latentes à être livrées et qui les livre au moment opportun. Peu importe l'état de réception des interruptions de Linux à un moment donné, RTAI livre les interruptions temps-réels de manière identique.

Au-dessus du module de base de RTAI, plusieurs modules offrent divers abstractions et services. Certains de ces modules peuvent être utilisés indépendamment des autres et certains dépendent d'autres modules. L'ensemble de RTAI et ses composantes sont chargeables dynamiquement puisque chaque module est un module noyau dynamique. Les sections suivantes décrivent certains aspects des composantes de RTAI.

2.2.1 Ordonnanceurs

Les ordonnanceurs de RTAI utilisent les services et les abstractions fournis par le module de base de RTAI décrit précédemment afin de fournir à leur tour des services et des abstractions de haut niveau au programmeur tels que les tâches et les divers mécanismes de communication. Trois ordonnanceurs viennent avec RTAI : un ordonnanceur pour uniprocesseur (UP), un ordonnanceur SMP et un ordonnanceur pour multi-uniprocesseur (MUP). Chacun de ces ordonnaceurs peut être utilisé soit en mode “one-shot” ou en mode “periodic”. Ces modes font référence à la manière dont les minuteries sont configurées et prennent racine dans l’architecture PC où la minuterie 8254 peut être configurée de différentes façons.

Sur cette architecture, les minuteries peuvent être configurées de manière à émettre une interruption selon une période de temps fixe, d’où le mode periodic, soit en les reprogrammant à chaque interruption, d’où le mode one-shot. Bien qu’il en prenne à peu près 3 microsecondes pour reprogrammer les minuteries, il n’est pas toujours avantageux d’utiliser le mode periodic pour éviter la reprogrammation. Lorsque les intervalles d’interruption varient, par exemple, on est forcé d’utiliser le mode one-shot puisqu’il s’agit du seul mode qui nous permet de générer les interruptions aux moments voulus.

L’ordonnanceur uniprocesseur implémente un algorithme d’ordonnancement qui sélectionne une tâche pour exécution sur un seul CPU. Son opération est très simple et consiste à donner le contrôle du CPU à la tâche ayant la plus haute priorité. En fait, il s’agit d’un ordonnanceur multi-liste basé sur la priorité et qui inclut du support pour l’héritage de priorité. Dans ce contexte, Linux est une tâche temps-réel qui détient la plus basse priorité. L’implémentation de l’ordonnanceur est faite dans deux fonctions séparées : *rt_schedule()* et *rt_timer_handler()*. La première est invoquée par les services de RTAI qui ont besoin d'effectuer un changement de contexte. La seconde est appellée en réaction à une interruption de la minuterie.

L'ordonnanceur SMP diffère de l'ordonnanceur UP du fait qu'il peut ordonnancer des tâches à plus d'un processeur. Cet ordonnanceur possède des services additionnels afin de permettre l'assignation du traitement d'interruption aux divers CPU, ainsi que de fournir un service d'affinité de CPU aux tâches. Aussi, l'ordonnanceur SMP peut utiliser les minuteries de l'APIC au lieu de se limiter au 8254 puisque l'APIC possède ses propres minuteries. Toutefois, l'ordonnanceur SMP reste un ordonnanceur basé sur la priorité.

L'ordonnanceur MUP traite un système SMP comme un ensemble de plusieurs uniprocesseurs. Ainsi, les minuteries de chaque processeur peuvent être configurées différemment des autres. Par exemple, un CPU peut être en mode periodic tandis qu'un autre pourrait être en mode one-shot. Cet ordonnanceur est très pratique pour certains types d'applications.

2.2.2 Unités de communication et autres services

RTAI est équipé d'une grande variété de services de communication. Certains de ces services font partie du module d'ordonnancement et d'autres sont des modules à part entière. Les sous-sections suivantes traitent des divers services de communication.

2.2.2.1 Boîtes de courrier

Les boîtes de courrier permettent d'échanger des données de tailles variables entre divers tâches en utilisant le pointeur à une boîte de courrier comme point de référence. Habituellement, il s'agit d'envoyer n octets à une boîte de courrier donnée. La tâche de réception peut ainsi lire m octets dans son tampon de données.

2.2.2.2 Messages et RPCs

Contrairement aux mailboxes, les messages et les RPCs utilisent un pointeur vers une structure de tâche pour identifier leurs correspondants. Ainsi, lorsqu'un message est envoyé, il est envoyé d'une tâche à une autre et l'appel à la fonction de service nécessite le passage d'un pointeur à une tâche. Aussi, les données échangées ne sont pas de taille variable, mais sont toujours un entier.

2.2.2.3 Sémaphores

Un sémaphore est un mécanisme de syncrhonisation qui permet à plusieurs tâches de synchroniser leur travail adéquatement. Dans RTAI, les sémaphores sont identifiés par leur structure. Ainsi, les tâches désirant se synchroniser sur un sémaphore devront utiliser un pointeur vers la structure de ce sémaphore. Autrement, les sémaphores dans RTAI fonctionnent de la même manière que les sémaphores conventionnels.

2.2.2.4 FIFOs

Les FIFOs permettent aux tâches d'écrire des données dans un tampon et de les lire dans un ordre de premier-entré-premier-sorti. L'utilité principale des FIFOs est de permettre l'échange d'information entre les tâches temps-réels et les processus Linux usuels. Lorsque les tâches temps-réels désirent opérer sur un FIFO, ils utilisent un identificateur de FIFO pour y avoir accès en utilisant l'interface de programmation exportée par le module de FIFO. Dans l'espace usager, chaque FIFO correspond à une entrée dans le répertoire */dev*. De cette manière, le FIFO # 1 est désigné par */dev/rtf1*. Les FIFOs permettent aux tâches temps-réels de collecter des données en temps-réel tout en permettant à une tâche Linux usuelle d'y avoir accès. L'acquisition de données est une application directe des FIFOs. Inversement, les processus Linux peuvent envoyer des données aux tâches temps-réels en utilisant les opérations Unix conventionnelles de *open, read, write, etc.*. Pour l'application Linux, le FIFO est un fichier comme un autre.

2.2.2.5 Mémoire partagée

Un autre moyen d'échanger une grande quantité d'information est possible grâce à l'utilisation de mémoires partagées. Principalement, l'unité de mémoire partagée gère l'allocation et la libération de régions de mémoire partagée. L'identification des régions mémoire est possible grâce à un mécanisme d'identification basé sur un nom. En utilisant ce nom, les différentes tâches peuvent retrouver une région mémoire leur étant désignée. Si une tâche essaie d'ouvrir une région mémoire déjà ouverte, cette région est cartographiée dans l'espace d'adressage de la tâche.

2.2.2.6 Posix

Le standard Posix est très large et il existe plusieurs composantes du standard Posix pour le temps-réel. Le module Posix de RTAI implémente le standard 1003.1c pour les pthreads et une partie du standard 1003.1b, les queues de messages. Étant donné la taille du module Posix dans RTAI, il n'est pas possible d'en discuter longuement dans le cadre de ce texte. Le lecteur intéressé est invité à consulter la documentation accompagnant RTAI pour plus d'information concernant le module Posix de RTAI.

2.2.2.7 Gestion de mémoire

RTAI permet l'allocation dynamique de mémoire grâce à un algorithme déterministe. Cette allocation fonctionne en utilisant un ensemble initial de morceaux de mémoire et en s'approvisionnant chez Linux en cas de besoin. Un des avantages de cette unité est la possibilité de programmer en C++ pour RTAI.

2.2.3 LXRT

LXRT intègre le meilleur des deux mondes en offrant les services RTAI aux processus Linux usuels. En fait, LXRT permet aux tâches temps-réels et aux applications Linux de communiquer ensemble de manière tout à fait transparente. Les possibilités de communication entre les tâches des différents domaines sont illustrées à la figure 2.3.

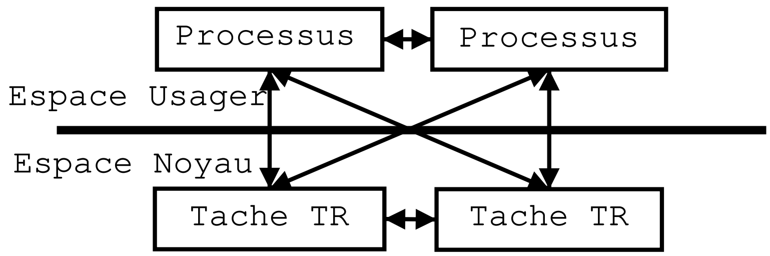


FIG. 2.3 – Possibilités de communication avec LXRT.

Les sous-sections suivantes discutent des différentes particularités de LXRT.

2.2.3.1 Services au domaine usager

La communication avec LXRT à partir du domaine usager est possible grâce une interruption logicielle. En utilisant cette interruption, les applications usagers peuvent demander des services à RTAI de la même manière qu'elles demandent des services à Linux à travers les appels systèmes. Dans les services exportés par RTAI, on retrouve tous les services qui n'étaient disponibiles qu'aux modules chargés dans le noyau tels que les boîtes de courrier, les messages, les RPCs et les sémaphores. En fait, en utilisant LXRT, il est possible de rendre les appels aux divers services complètement transparents au contexte. Ainsi, la programmation des applications Linux pour utiliser les services RTAI se fait de la même manière qu'avec un module qui se charge à l'intérieur du noyau. La seule différence réside dans l'usage d'un *main()* au lieu d'un *init_module()* et d'un *cleanup_module()*. On peut ainsi tester des applications en milieu usager afin de vérifier leur fonctionnement adéquat pour ensuite les insérer en tant que modules à l'intérieur du noyau. Notons, toutefois, que les applications Linux utilisant LXRT ne deviennent pas temps-réels déterministes (hard real-time), elle demeure des tâches temps-réels non-déterministes (soft real-time). Au delà des fonctions initialement exportées, il est possible d'étendre LXRT afin d'offrir des services additionnels.

2.2.3.2 Voler des tâches à Linux

En plus de permettre aux processus Linux d'utiliser les services de RTAI, LXRT permet aussi aux processus Linux de devenir des tâches temps-réels RTAI déterministes (hard real-time) à part entière à l'aide d'une fonction de service, *rt_make_hard_re al_time()*. Contrairement aux modules chargés dynamiquement, ces tâches temps-réels roulent dans leur environnement mémoire isolé et on obtient ainsi une protection mémoire pour les tâches temps-réels. Une fois dans ce mode, les tâches obtiennent le même degré de déterminisme que celui offert aux autres tâches temps-réels. Comme il est possible de migrer au domaine temps-réel, il est aussi possible de retourner au mode Linux normal avec un appel similaire.

2.2.3.3 Traitement d'exceptions

Afin de profiter au maximum de la protection mémoire offerte par l'utilisation de processus Linux et pour permettre l'utilisation de déboggeurs conventionnels avec les tâches temps-réels, LXRT implémente aussi un mécanisme de traitement des exceptions. Puisque RTAI est le premier à recevoir les interruptions, il est en mesure de renvoyer ces interruptions à la fonction de traitement appropriée ou à Linux, dans le cas où cette interruption s'adresse à Linux.

2.2.3.4 Services QNX

Comme certains systèmes d'exploitation sont largement répandus, il est souhaitable que RTAI puisse fournir des interfaces de compatibilité. C'est le but des services QNX qui permettent une interface de programmation très similaire à celle offerte par QNX pour la communication synchrone.

2.2.3.5 Serveur Unix

Bien qu'il soit possible aux processus Linux de migrer au domaine temps-réel, il leur est impossible d'utiliser les services Linux conventionnels une fois la transition faite. Le serveur Unix permet de pallier à cette limitation en permettant aux tâches hybrides d'avoir accès aux services Linux usuels. Pour ce faire, une tâche est lancée en parallèle à la tâche hybride et échangera des informations avec cette dernière grâce à une mémoire partagée. Toutefois, l'utilisation de ce service est non-déterministe et la tâche temps-réel devra attendre que Linux finisse son traitement avant de reprendre la main, peu importe les délais encourus.

Chapitre 3

Algorithmes et structures de données

Maintenant que nous avons couvert les différents concepts sur lesquels LTT est basé voyons les concepts implémentés dans LTT. Cette discussion couvrira toutes les composantes de LTT en détails incluant les détails d'implémentation pour Linux et RTAI. LTT est en fait composé de plusieurs modules logiciels indépendants les uns des autres, chacun des modules ayant été conçu de manière à en faciliter l'extension et à minimiser l'impact sur la performance.

3.1 Architecture globale de la collection de données

La figure 3.1 illustre l'architecture de collecte de données de LTT sous Linux. Les flèches indiquent le flot de données à travers les différents modules qui composent

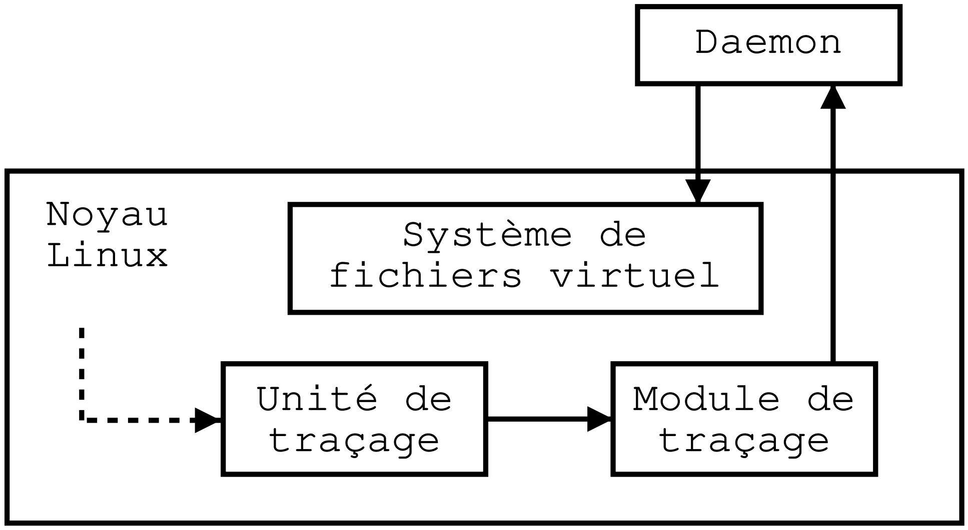


Fig. 3.1 – Architecture de collecte de données de LTT sous Linux.

LTT, la source primaire de données étant le noyau instrumenté. Aucune composante du noyau n'est désignée comme source d'information de traçage, mais plusieurs des composantes décrites à la section 2.1 sont tracées.

Principalement, les événements tracés sont envoyés au module de traçage à travers l'unité de traçage du noyau. Le module de traçage, visible du domaine usager comme une entrée dans le répertoire */dev*, enregistre alors l'événement dans son tampon de données. Finalement, le daemon lit les données du module de traçage et les enregistre dans un fichier fourni par l'usager.

Lorsque RTAI est présent avec Linux, l'architecture de LTT est telle que présentée à la figure 3.2. Encore une fois, les flèches représentent le flot de données entre les différentes composantes de LTT. À part l'ajout de l'unité de traçage de RTAI et les sources additionnelles d'événements dans RTAI, l'architecture de LTT demeure sensiblement la même.

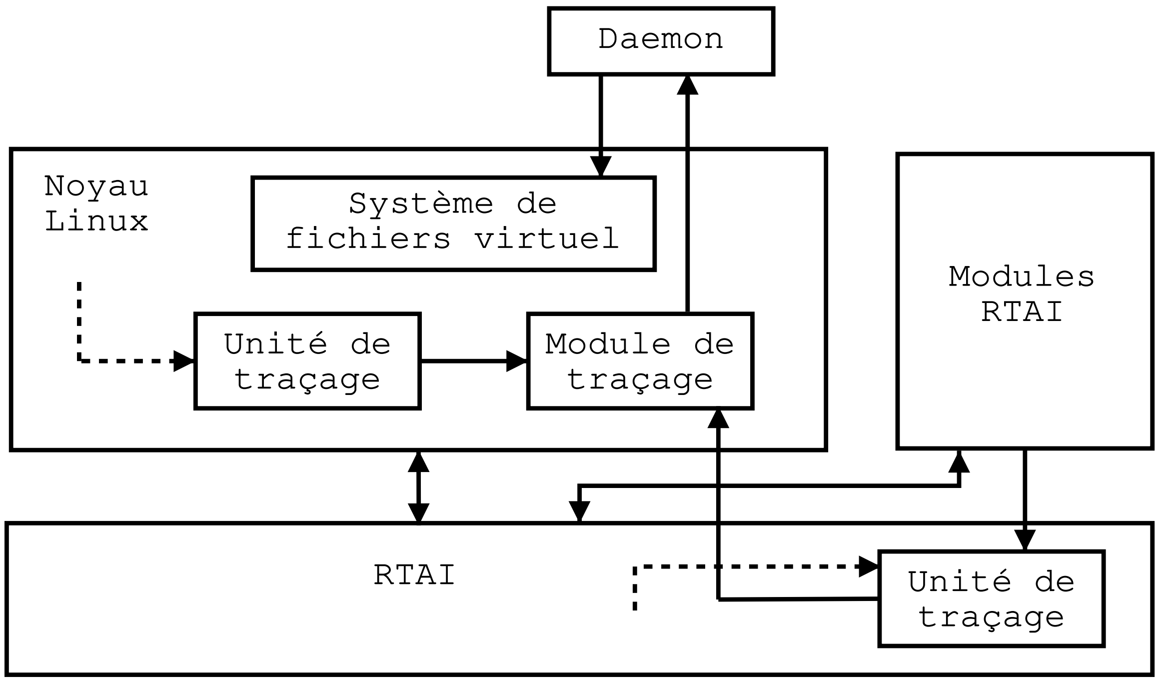


FIG. 3.2 – Architecture de collecte de données de LTT sous Linux avec RTAI.

3.2 L'unité de traçage du noyau

L'unité de traçage du noyau est une extension aux services de base du noyau. Sa fonction principale est de fournir un point d'entrée unique à tous les autres services du noyau qui ont besoin de tracer des événements. Toutefois, l'unité de traçage n'enregistre aucun événement. Elle renvoie, plutôt, les événements au module de traçage.

Pour offrir ces services, le module de traçage doit s'enregistrer auprès de l'unité de traçage au démarrage du système, si le module était compilé comme faisant partie du noyau. Si le module était compilé pour chargement dynamique, l'enregistrement s'effectue au chargement du module. Lorsqu'il s'enregistre, le module de traçage fournit à l'unité de traçage une fonction de rappel qui doit être appelée lorsqu'un événement se produit. Si aucun module de traçage n'est enregistré, les événements sont ignorés. Aussi, l'unité de traçage fournit une interface d'appel au module de traçage lui permettant de configurer la manière dont les compteurs de programme sont enregistrés lors d'un appel système. Puisque la plupart des appels système se font à travers des librairies, et non directement par le programme, il est intéressant de pouvoir spécifier la profondeur de la pile d'appels à laquelle il faut remonter ou une plage d'adresses dans laquelle devrait se trouver l'adresse originale d'appel. Lorsque cette valeur est initialisée, le noyau parcourra la pile lors d'un appel système afin de trouver une adresse qui réponde à la configuration donnée. Voici les interfaces de programmation de l'unité de traçage :

– Enregistrement du module de traçage :

  `register_tracer()`

  Cette fonction prend une fonction de rappel en paramètre, l'enregistre comme étant la fonction à appeler lorsqu'un événement se produit et retourne un code d'erreur. C'est cette fonction qui est utilisée par le module de traçage pour enregistrer sa fonction de traçage.

– Désactivation du module de traçage :

  `unregister_tracer()`

  Prend la fonction déjà enregistrée, vérifie que c'est bien celle qui est couramment utilisée et la désactive. Ici aussi, un code d'erreur est retourné.

– Configuration du traçage :

`trace_set_config()`

Permet au module de traçage d'initialiser différents paramètres de traçage, tels que la profondeur à laquelle l'adresse d'appel doit être trouvée lors d'un appel système ou la plage d'adresses à laquelle l'adresse d'appel appartient. Ces paramètres sont ensuite utilisés par la fonction qui trace les appels système afin d'extraire l'information désirée.

- Obtention de la configuration :

  `trace_get_config()`

  Fournit à l'appelant les paramètres de configuration initialisés avec la fonction précédente.

- Enregistrement d'une fonction de rappel sur événement :

  `trace_register_callback()`

  Cette fonction prend en paramètres une fonction de rappel et un identificateur d'événement. Par la suite, si l'événement designé se produit, la fonction de rappel sera immédiatement appelée.

- Désactivation d'une fonction de rappel sur événement :

  `trace_unregister_callback()`

  Utilisé pour désactiver une fonction de rappel enregistrée avec la fonction précédente.

- Traçage d'événement :

  `trace_event()`

La fonction unique de traçage du noyau qui est appelée à chaque fois qu'un événement tracé se produit. C'est ici que la fonction de rappel enregistrée par le module de traçage est appelée. C'est ici aussi que les fonctions de rappel d'événements sont invoquées.

En résumé, l'unité de traçage agit comme un lien entre les différentes unités du noyau et le module de traçage.

3.3 L'unité de traçage de RTAI

L'unité de traçage est une addition aux autres unités déjà disponibles dans RTAI. Comme ces autres unités, elle est implémentée en tant que module noyau chargeable dynamiquement. Principalement, elle sert à fournir aux différents modules RTAI un point unique d'entrée pour le traçage d'événements. Toutefois, elle n'enregistre aucun événement. Elle ne fait que les renvoyer au module de traçage, s'il est chargé. Sinon, les événements tracés sont ignorés par l'unité de traçage.

Afin de fournir ces services, l'unité de traçage de RTAI fournit trois fonctionnalités principales : une fonction unifiée de traçage pour RTAI, une fonction d'enregistrement du module de traçage et une fonction de désactivation du module de traçage. Pour que le module de traçage reçoive les événements, il faut qu'il s'enregistre auprès de l'unité de traçage en fournissant une fonction de rappel. Cette fonction de rappel sera invoquée à chaque fois qu'un événement RTAI se produit. En réalité, cette fonction est la même que celle fournie par le module de traçage à l'unité de traçage du noyau.

Donc, la base de données d'événements est unique pour tout le système. Voici les interfaces de programmation de l'unité de traçage de RTAI :

- Enregistrement du module de traçage :

  `rt_register_tracer()`

  Comme la fonction d'enregistrement du noyau, cette interface prend une fonction de rappel en paramètre et l'enregistre comme étant la fonction à invoquer, lorsqu'un événement se produit, et retourne un code d'erreur. Cette fonction est utilisée par le module de traçage pour enregistrer sa fonction de rappel. Cette dernière est la même fournie à l'unité de traçage du noyau.
- Désactivation du module de traçage :

  `rt_unregister_tracer()`

  Prend la fonction de rappel déjà enregistrée, vérifie qu'il s'agit bien de celle qui est couramment utilisée et la désactive. Encore une fois, un code d'erreur est retourné.
- Traçage d'événement :

  `rt_trace_event()`

  La fonction de traçage unifiée de RTAI appelée à chaque fois qu'un événement se produit. C'est ici que la fonction de rappel enregistrée par le module de traçage est appelée.

Contrairement à l'unité de traçage du noyau, l'unité de traçage de RTAI n'offre aucune interface de configuration de paramètres, du moins, pour l'instant. Toutefois, elle a une fonctionnalité similaire, celle d'offrir un lien entre les différents modules RTAI et le module de traçage.

3.4 L'instrumentation du noyau

L'instrumentation du noyau se compose des différents événements tracés dans le noyau. Il y a différents types d'événements. Chaque type d'événement possède ses propres ensembles de données, et la taille d'une entrée de traçage varie, ainsi, selon le type d'événement enregistré. Sous un type d'événement, il peut même y avoir des sous-types. La liste des types d'événements, de leurs sous-types, ainsi que les données enregistrées pour chacun des types et sous-types est présentée au tableau 3.1.

Étant donné le grand nombre de modifications au code du noyau et le nombre de fichiers modifiés, un ensemble de macros a été utilisé, au lieu d'un appel direct à l'unité de traçage, pour tracer les événements noyau. Durant la compilation du noyau, si le traçage est désactivé alors le code ajouté n'a aucun effet. Si, toutefois, le traçage est activé, alors les macros sont remplacées par une portion de code qui initialise certaines valeurs et appelle ensuite l'unité de traçage.

Voici un exemple d'un macro utilisée pour le traçage :

```
TRACE_SCHEDCHANGE();
```

Il s'agit en fait d'une macro C à qui on peut passer des paramètres comme avec une fonction usuelle. Dans ce cas-ci, l'appel complet est :

```
TRACE_SCHEDCHANGE(prev->pid, next->pid, prev->state);
```

Tab. 3.1 – Événements noyau tracés.

Type d'événement	*Sous-type*	*Détail de l'événement*
Début de trace	Aucun	Spécifique au module de traçage
Entrée d'appel système	Aucun	ID de l'appel et compteur de programme
Sortie d'appel système	Aucun	Aucun
Entrée de faute	Aucun	ID de faute et compteur de programme
Sortie de faute	Aucun	Aucun
Entrée d'interruption	Aucun	ID de l'interruption
Sortie d'interruption	Aucun	Aucun
Changement de contexte	Aucun	Tâche entrante, tâche sortante et état de la tâche sortante
Minuterie du noyau	Aucun	Aucun
Bottom half	Aucun	ID du bottom half
Processus	Création de tâche noyau	Addresse de début de tâche et PID
	Fork	PID du processus créé
	Exit	Aucun
	Wait	PID sur lequel on attend
	Signal	ID du signal et PID de destination
	Wakeup	PID du processus et état avant wakeup
Système de fichier	Début d'attente sur tampon	Aucun
	Fin d'attente sur tampon	Aucun
	Exec	Nom du fichier
	Open	Nom du fichier et descripteur
	Close	Descripteur de fichier
	Read	Descripteur de fichier et quantité lue
	Write	Descripteur de fichier et quantité écrite
	Seek	Descripteur de fichier et offset
	Ioctl	Descripteur de fichier et commande
	Select	Descripteur de fichier et temps d'attente
	Poll	Descripteur de fichier et temps d'attente
Minuterie	Expiré	Aucun
	Ajuster itimer	Type et temps
	Ajuster minuterie	Temps
Mémoire	Allocation de page	Ordre de grandeur
	Libération de page	Ordre de grandeur
	Échange en entrée	Adresse de page
	Échange en sortie	Adresse de page
	Début d'attente sur page	Aucun
	Fin d'attente sur page	Aucun
Communication par socket	Appel socket	ID de l'appel et ID du socket
	Création de socket	Type de socket et ID du socket créé
	Envoie par socket	Type de socket et quantité envoyé
	Réception par socket	Type de socket et quantité reçue
Communication inter-processus	Appel système V IPC	ID de l'appel et ID de l'entité
	Création de queue de message	ID de la queue de message
	Création de sémaphore	ID du sémaphore
	Création de mémoire partagée	ID de la mémoire partagée
Réseau	Entrée de paquet	Type de protocole
	Sortie de paquet	Type de protocole

Cette instruction de traçage sert en fait à tracer les changements de contexte qui ont lieu dans Linux. Toutes les autres instructions de traçage ressemblent beaucoup à celle-ci. Ce sont toutes des macros C qui génèrent du code si et seulement si le traçage est activé. Voici la définition complète de la macro précédente lorsque le traçage est activé :

```
#define TRACE_SCHEDCHANGE(OUT, IN, OUT_STATE) \
           do \
           {\
           trace_schedchange sched_event;\
           sched_event.out       = OUT;\
           sched_event.in        = IN;\
           sched_event.out_state = OUT_STATE; \
           trace_event(TRACE_EV_SCHEDCHANGE, &sched_event);\
           } while(0);
```

3.5 L'instrumentation de RTAI

Comme avec l'instrumentation de Linux, l'instrumentation de RTAI est une composante très importante du support de LTT pour RTAI puisqu'elle détermine l'information qui pourra être collectée concernant le fonctionnement des divers modules de RTAI. Ici encore, il existe différents types d'événements avec chacun un ensemble de données pour le décrire. Puisque certains événements appartiennent au même type de fonctionnalité ou au même module RTAI, des sous-types sont souvent utilisés pour identifier un événement parmi les autres événements faisant partie du même groupe. Contrairement à Linux qui n'a qu'un seul point d'entrée pour ses services, l'interruption logicielle d'appel système, RTAI a plusieurs points d'entrées pour ses services. De fait, lorsque les applications veulent accéder aux services du noyau, elles utilisent l'interruption d'appel système. Toutefois, lorsque les tâches temps-réels accèdent aux services de RTAI en tant que modules, ils appellent directement la fonction de service. Il en résulte un grand nombre de points d'entrées dans RTAI qui forcent l'instrumentation de chacun de ces points d'entrées. Conséquemment, il y a plus d'événements instrumentés dans RTAI que dans Linux. Ces événements sont présentés dans le tableau 3.2.

Comme avec le noyau, l'instrumentation de RTAI peut être désactivée à la configuration du noyau. Le même type de macro utilisée pour instrumenter Linux a été utilisée pour instrumenter RTAI.

3.6 Le module de traçage

Le module de traçage est un élément clé de l'architecture de LTT. La performance du processus de traçage dépend largement de ce module. En théorie, le but du module de traçage est simple : enregistrer les descriptions des événements qui se produisent et les livrer de manière efficace au daemon. En pratique, son implémentation est beacoup plus complexe, pour plusieurs raisons.

TAB. 3.2 – Événements RTAI tracés.

Type d'événement	*Nombre de sous-types*	*Détail de l'événement*
RTAI mount	N/A	Aucun
RTAI un-mount	N/A	Aucun
Entrée d'IRQ global	N/A	ID de l'IRQ et contexte
Sortie d'IRQ global	N/A	Aucun
Entrée d'IRQ au CPU	N/A	ID de l'IRQ et contexte
Sortie d'IRQ au CPU	N/A	Aucun
Entrée de faute	N/A	ID de la faute et adresse
Sortie de faute	N/A	Aucun
Entrée de SRQ	N/A	ID du SRQ et contexte
Sortie de SRQ	N/A	Aucun
Entrée dans Linux	N/A	ID du CPU en entrée
Entrée dans RTAI	N/A	ID du CPU en entrée
Changement de contexte	N/A	Tâche entrante, tâche sortante et état de la tâche sortante
Tâche	12	Sous-type et 3 champs
Minuterie	5	Sous-type et 2 champs
Sémaphore	6	Sous-type et 2 champs
Message	6	Sous-type et 3 champs
RPC	4	Sous-type et 3 champs
Mailbox	10	Sous-type et 3 champs
FIFO	26	Sous-type et 2 champs
Mémoire partagée	5	Sous-type et 3 champs
POSIX	44	Sous-type et 3 champs
LXRT	8	Sous-type et 3 champs
LXRT-Informed	10	Sous-type et 3 champs

Premièrement, le module de traçage doit extraire de l'information additionnelle concernant chaque événement qui se produit. Cette information additionnelle est composée du temps auquel l'événement s'est produit et de l'identificateur du CPU sur lequel il s'est produit. Puisque le temps absolu est une donnée large (8 octets dont 4 pour les secondes et 4 pour les microsecondes), seule la différence entre le moment actuel et le moment auquel le dernier changement de tampon de données s'est produit est enregistré. Le moment auquel le dernier changement de tampon a eu lieu est nécessaire pour implémenter un mécanisme de double-tamponnage (double-buffering) expliqué plus bas. Étant donné la taille des tampons et la fréquence à laquelle les événements se produisent, 4 octets suffisent pour enregistrer la différence. Notons que le module de traçage utilise la fonction *do_gettimeofday()* du noyau afin d'obtenir un temps absolu d'arrivée de l'événement. Sous un PC utilisant un processeur Pentium, cet appel utilise le Time Stamp Counter (TSC) du processeur qui permet une précision à la microseconde près.

Deuxièmement, le module de traçage doit être configurable. Voici les options de configuration possibles :

- Taille des tampons de données.
- Masque de traçage.
- Masque de détails des événements.
- Enregistrement de l'ID du CPU.
- Suivre un certain PID.

- Suivre un certain groupe de processus.
- Suivre un certain GID.
- Suivre un certain UID.
- Profondeur d'appel à laquelle le compteur de programme doit être extrait lors d'un appel système.
- Plage d'adresses à laquelle le compteur de programme extrait doit appartenir, lors d'un appel système.

Le masque de traçage est utilisé pour déterminer si un événement doit être enregistré. Le masque de détails détermine si les détails de l'événement doivent être enregistrés. Lorsqu'un PID, un groupe de processus, un GID ou un UID est suivi, seuls les événements qui se produiront durant l'exécution des processus désignés seront enregistrés. La profondeur d'appel et la plage d'appel sont telles que discutées précédemment. La configuration du module de traçage est effectuée à l'aide de l'appel système *ioctl()*.

Troisièmement, le module de traçage doit être réentrant puisqu'un événement peut se produire à deux niveaux de prioriotés différents en même temps. Par exemple, une interruption peut se produire pendant que le module de traçage est en train de traiter un événement d'appel système. Puisque les interruptions ont priorité sur tout autre événement et différentes interruptions ont différentes priorités, le traitement de l'appel système sera suspendu et un événement d'interruption sera envoyé au module de traçage. Pour résoudre ce problème, un verrou noyau est utilisé durant la section critique de l'enregistrement de l'événement. Toutefois, ceci pourrait être un problème si le verrou est maintenu trop longtemps étant donné la fréquence à laquelle certains événements se produisent. Ceci n'est pas un problème dans LTT puisque le temps d'utilisation des verrous est très restreint. De plus, puisque la fonction d'enregistrement n'appelle aucune autre fonction, il ne peut y avoir de problèmes d'inversions de priorités. Tout enregistrement d'événement peut être interrompu par un événement de plus haute priorité du moment que le verrou n'est pas en action.

Afin d'être accessible comme un pilote de périphérique par le système de fichiers virtuel, le module de traçage fournit quelques interfaces d'opérations sur fichier. Ceux-ci sont :

- *open()*, la tâche y faisant appel est enregistrée comme étant celle qui doit recevoir un signal au moment où le tampon de données est plein.
- *ioctl()*, offre une interface de configuration au daemon.
- *release()*, pour que le module soit libéré à la mort du daemon ou lorsque le traçage est terminé.
- *fsync()*, pour réinitialiser le module.

Afin de traiter efficacement une grande quantité de données, le module utilise un mécanisme de double-tamponnage. Avec cette technique, un tampon est utilisé pour y insérer les données jusqu'à ce qu'il soit plein. À ce moment, un signal de type SIGIO est envoyé au daemon et le tampon plein devient le tampon de lecture de daemon. L'ancien tampon de lecture du daemon est recyclé comme nouveau tampon d'écriture pendant que le daemon effectue la lecture du tampon rempli. Ce processus d'échange de tampons continue tout le long des opérations de traçage et permet de ne perdre aucune donnée ; à condition que le daemon ait le temps de lire le tampon qui lui est assigné avant que le second ne soit rempli. Toutefois, puisque la taille des tampons est configurable, il en revient au daemon de configurer le module adéquatement. Aussi, même si le module a été ouvert en utilisant *open()*, aucun événement n'est enregistré jusqu'à ce que la commande de début de traçage soit envoyée au module en utilisant l'interface *ioctl()*. Inversement, le traçage est arrêté lorsque la commande d'arrêt est envoyée par la même interface.

3.7 Le daemon

La fonction principale du daemon est d'extraire l'information accumulée par le module de traçage et de l'écrire dans un fichier de sortie. Aussi, il fournit à l'usager un certain nombre d'options afin de contrôler le processus de traçage. En plus de permettre à l'usager de configurer le module de traçage selon les paramètres de configuration possibles, il lui permet aussi de spécifier la durée de la session de traçage.

Une fois le daemon démarré, il ouvre le module, le configure et initialise une minuterie, si une durée de temps avait été spécifiée. Sinon, l'usager doit tuer le daemon manuellement pour arrêter le traçage. En temps normal, le daemon attend le signal de SIGIO pour lire du module de traçage, ou les événements de mort ou de minuterie pour arrêter le traçage.

Malgré le fait que la technique de collection de données décrite plus haut permette d'obtenir les détails du fonctionnement dynamique du système, un dernier élément d'information manque pour permettre une reconstruction adéquate du comportement du système : l'état du système précédent le traçage. Pour combler cette lacune, le daemon parcourt le répertoire */proc* pour enregistrer des informations concernant les divers processus. Ces informations sont : PID, nom de l'exécutable et PID du parent. Cet tâche est effectuée tout de suite après la configuration du module de traçage et le démarrage du processus de traçage. L'information extraite est conservée dans un fichier différent de la trace qui sera utilisé par l'outil d'analyse.

3.8 L'outil d'analyse et de visualisation

Contrairement aux autres composantes de LTT, l'outil de visualisation et d'analyse roule après l'accumulation des traces. L'outil utilise l'état initial du système et le fichier de trace créé par le daemon afin de recréer le comportement dynamique du système dans l'intervalle de temps donné. Voyons les différentes composantes de l'outil d'analyse et de visualisation.

3.8.1 La base de données des événements

Le fichier contenant l'état initial du système avant le traçage est le premier à être chargé par l'outil d'analyse. Les PIDs et PPIDs sont utilisés pour bâtir un arbre de processus liant tous les parents à leurs descendants. Cette arbre sera enrichi par les informations contenues dans le fichier de trace puisque certains processus ont pu être créés durant le traçage.

Une fois le fichier d'état initial chargé, le fichier de trace est cartographié en mémoire à l'aide de l'appel *mmap()*. Cet appel retourne un pointeur à une adresse en mémoire à partir de laquelle on trouve le contenu du fichier cartographié. Pour accéder à un certain élément du fichier à une certaine distance du début du fichier, on accède à la région mémoire à cette même distance de l'adresse retournée par *mmap()*. L'avantage ici est que le système d'exploitation n'est pas obiligé de maintenir le fichier en mémoire physique. De fait, il choisira certains morceaux qui seront effectivement accessibles par l'application mais retirera de la mémoire physique les parties non-utilisées. Lorsque ces dernières seront accédées, une faute de page sera signalée et il se chargera de les placer en mémoire pour que l'application y accède. Le tout étant transparent à l'application.

Une fois le fichier cartographié, il est parcouru une fois afin de bâtir des statistiques concernant le système et afin d'extraire certaines informations qui seront nécessaires au fonctionnement adéquat de la base de données des événements. Puisque les événements sont lus dans leur ordre d'arrivée, il est possible d'effectuer des mesures sans difficultés. Les informations suivantes sont extraites de cette première lecture :

- Les processus créés durant la session de traçage sont ajoutés à l'arbre de processus.
- Le temps passé entre les entrées dans les appels système et les fins des appels

sont calculés. Cette information est répertoriée par appel système pour chaque processus tout en tenant compte du nombre d'appels de chaque type.

- Le temps attribué par l'ordonnanceur aux divers processus est calculé.
- Le temps d'attente pour entrées/sorties est calculé pour chaque processus.

On procède aussi au décompte du nombre de changements de contexte, d'interruptions, de fautes, de minuteries et de paquets réseau. La base de données RTAI complète cette analyse avec des informations concernant les tâches RTAI et les événements gérés par RTAI.

Les propriétés de *mmap()* sont ensuite utilisées pour former la base de données des événements de LTT qui permet de parcourir les événements récoltés et d'en extraire les informations pertinentes. Ainsi, la base de données permet la fonctionnalité suivante :

- `DBEventNext()`, aller au prochain événement.
- `DBEventPrev()`, aller à l'événement précédent.
- `DBEventID()`, obtenir l'ID de l'événement.
- `DBEventTime()`, obtenir le temps auquel l'événement s'est produit.
- `DBEventCPUID()`, obtenir l'ID du CPU sur lequel l'événement s'est produit.
- `DBEventStruct()`, obtenir la structure native décrivant l'événement.
- `DBEventString()`, obtenir une chaîne de caractères décrivant l'événement.
- `DBEventSize()`, obtenir la taille occupée par l'événement dans le tampon de données.
- `DBEventProcess()`, obtenir le processus auquel appartient l'événement.

Ces mêmes fonctions sont valides aussi bien pour les traces contenant des données pour Linux que celles contenant des données pour RTAI. Toutefois, une extension à la base de données initiale traite spécifiquement les données RTAI. Souvent, la base de données initiale est en mesure d'utiliser la base de données RTAI pour extraire les informations nécessaires. C'est le cas des services de parcours de la trace et des autres services permettant d'extraire de l'information concernant un événement. Une seule information n'est disponible qu'auprès de la base de données RTAI et c'est l'identification de la tâche RTAI à laquelle appartient un événement. Ce service est fournit par `RTAIDBEventTask()`.

En plus, la base de données offre des services permettant d'identifier si un événement marque un changement de contrôle ou pas. Un changement de contrôle se produit lorsqu'il y a une transition d'une entité logicielle à une autre. Un appel système, par exemple, crée un changement de contrôle puisque le système passera de l'exécution du code d'un processus à celui du noyau. Une interruption qui se produit durant l'exécution d'un processus aura aussi le même effet. Une interruption qui se produit pendant l'exécution de code appartenant au noyau ne marque cependant pas un changement de contrôle puisque ce sera le noyau qui aura le contrôle pour traiter l'interruption. Aussi, un retour d'appel système ne marque pas nécessairement un retour de contrôle au code d'une application puisque le système d'exploitation peut décider d'effectuer un changement de contexte tout de suite après le retour d'appel. Ainsi, un événement en soi ne marque pas un changement de contrôle.

Pour un système Linux ne roulant pas RTAI, on utilise une suite de deux événements pour déterminer un changement de contrôle. Lors d'un retour d'appel système, il y aura transition à l'application si et seulement si le prochain événement marque une nouvelle entrée dans le noyau. En réalité, tout événement ayant lieu dans le noyau pourrait être la source d'une transition à un processus si l'événement qui le suit marque une entrée définitive au noyau. Une entrée définitive au noyau pourrait être : une interruption en milieu usager, une faute en milieu usager ou un appel système. Il serait difficile d'énoncer une liste de règles permettant de décrire complètement et adéquatement le comportement du noyau Linux et, conséquemment, la signification exacte des suites d'événements. Néanmoins, les règles de base peuvent être élaborées et corrigées ultérieurement selon les différents tests effectués. C'est de cet esprit que sont nées les règles trouvées dans LTT. Afin de simplifier l'utilisabilité de ces règles, la base de données fournit les services suivants :

- `DBEventControl()`, permet de savoir si l'événement désigné crée un changement de contrôle.
- `DBEventControlEntry()`, permet de savoir si l'événement désigné marque un passage d'un processus au noyau.
- `DBEventControlExit()`, permet de savoir si l'événement désigné marque un passage du noyau à un processus.

Contrairement aux états possibles dans un système Linux sans RTAI, dans un processus ou dans le noyau, un système Linux avec RTAI est plus complexe et peut se trouver dans quatre états différents. Dans un tel système on peut être dans un processus, dans le noyau Linux, dans une tâche temps-réel RTAI ou dans le noyau RTAI. Ici, il n'est plus possible de se fier sur une suite de deux événements pour connaître l'état actuel du système. On doit plutôt maintenir une variable d'état qui identifie l'état actuel et qui servira au calcul du prochain état selon la séquence d'événements. À toute fin pratique, la modélisation du comportement du système est basée sur une machine à états qui possède quatre états distincts et où les transitions sont dues aux événements successifs. Ainsi, un retour d'interruption peut vouloir dire un retour à l'exécution du noyau ou un retour à un processus quelconque. C'est la machine à états qui permettra de déterminer l'action. Ceci pose un problème supplémentaire puisqu'il est souvent nécessaire de se déplacer à grand pas dans la trace tout en étant en mesure de déterminer l'état dans lequel on se trouve à un événement. Afin de résoudre ce problème, une technique de retour arrière est utilisée. Cette technique consiste à reculer un certain nombre d'événements à partir de l'événement actuel jusqu'à ce qu'on trouve un événement déterminant. Un événement déterminant est un événement après lequel l'état du système est connu peu importe les événements précédents. Une interruption, par exemple, marque toujours une entrée au noyau RTAI. Une fois l'événement marquant trouvé, on avance de nouveau jusqu'à l'événement désigné, identifiant ainsi l'état actuel. Encore une fois, il n'est pas possible d'élaborer une description exhaustive du comportement du système. On peut néanmoins établir quelques règles de base et les corriger au fûr et à mesure que les séquences de tests de trace sont effectuées. Comme avec un système Linux sans RTAI, la base de données RTAI offre des services permettant d'identifier les états correspondants aux événements dans la trace d'un système Linux avec RTAI :

- `RTAIDBGetCurrentState()`, permet d'identifier l'état du système au moment où se produit l'événement désigné. Il s'agit en fait de l'état du système avant que ne se produise l'événement désigné.
- `RTAIDBEventState()`, permet d'identifier l'état du système suite à l'événement désigné.

3.8.2 L'outil de visualisation

L'outil de visualisation se base sur la fonctionnalité fournie par la base de données des événements pour présenter une vue cohérente du système à l'usager. Afin d'en faciliter l'usage, l'outil de visualisation peut aussi bien être utilisé à la ligne de commande, permettant à l'usager de générer un fichier de sortie texte contenant les informations concernant la trace, ou dans une interface graphique, permettant à l'usager d'interagir avec l'application et de parcourir la trace à sa guise.

Dans les deux modes, l'outil d'analyse permet à l'usager d'avoir accès à la liste complète des événements et aux statistiques extraites de la trace. Dans le cas de la liste complète des événements, il s'agit de donner pour chaque événement : l'ID du CPU sur lequel il s'est produit, son type, le temps auquel il s'est produit, le PID du processus auquel il appartient, le nombre d'octets qu'il occupe dans la trace et une chaîne de caractères décrivant en détails l'événement. Dans le cas des statistiques extraites, il s'agit de donner les informations suivantes pour chaque processus : le nombre d'appels système, le nombre de fautes, la quantité de données lues dans des fichiers, la quantité de données écrites dans des fichiers, le temps passé à exécuter du code appartenant au processus, le temps octroyé par l'ordonnanceur, le temps d'attente total pour des entrées/sorties ainsi que la liste des appels système effectués, le nombre de fois que chacun a été effectué et le temps total passé à l'intérieur de cet appel. Les statistiques suivantes concernant l'ensemble du système durant la trace sont aussi fournies : le temps de début de la trace, le temps de fin, la durée de traçage, la quantité de temps octroyée au processus inactif (idle) par l'ordonnanceur, le nombre d'événements, le nombre de changements de contexte, le nombre de minuteries noyau, le nombre d'appels système, le nombre de fautes, le nombre d'interruptions, le nombre de pages allouées, le nombre de pages libérées, le nombre de paquets sortants, le nombre de paquets entrants, le vecteur d'interruption ainsi que la liste des appels système effectués à travers tout le système, le nombre de fois que chacun a été effectué et le temps total passé à l'intérieur de cet appel.

Dans le mode graphique, l'outil de visualisation affiche aussi à l'usager un graphe illustrant les divers changements de contrôle et leurs causes. Ce graphe est très utile puisqu'il présente à l'usager un modèle du comportement du système dans un format très accessible. Dans ce graphe, le temps passé à l'intérieur d'un même état, dans un processus usager par exemple, est représenté par une ligne horizontale proportionnelle à la durée de cet état telle que vue dans la trace et les changements de contrôle sont représentés par des lignes verticales qui relient l'état de départ et l'état d'arrivée. Afin de différencier les divers types de changements de contrôle, chaque type d'événement causant un changement de contrôle est représenté par une couleur différente : un appel système par le bleu, une faute par le gris et une interruption par le blanc. De manière similaire, le temps passé dans un processus est représenté en vert tandis que le temps passé dans le noyau est représenté en orange. Aussi, les événements importants sont accompagnés d'une icône qui en donne quelques détails. Un appel système, par exemple, est accompagné d'une icône distinctive et du nom de l'appel système en question.

Ici, les fonctionnalités de la base de données des événements sont utilisées à fond. Les changements de contrôle sont identifiés et les descriptions des événements sont extraites à l'aide des fonctions prévues à cet effet. La difficulté principale étant d'identifier les événements se trouvant aux extrémités de l'affichage et de compléter le dessin à ces extrémités tout en tenant compte des états attribués aux événements qui précèdent de peu ou dépassent de peu l'espace d'affichage.

En premier, on identifie les bornes de dessin. Ces bornes sont formées par l'instant dans le temps auquel correspond le premier point de dessin et l'instant dans le temps auquel correspond le dernier point de dessin. Puisque l'usager peut agrandir et rétrécir la période de temps affichable, le facteur de conversion entre le temps et la position dans l'échelle d'affichage changent. Ce facteur de conversion est conservé en nombre de microsecondes par pixel. Lorsqu'on agrandit la période affichée (zoom out), le nombre de microsecondes par pixel augmente. À l'inverse, lorsqu'on diminue la période affichée (zoom in), le nombre de microsecondes par pixel diminue. Ce facteur est initialisé à une valeur par défaut et sert au calcul de la taille en nombre de pixels que prendrait le dessin complet de la trace qui équivaut à la durée de la trace en microsecondes fois le nombre de microsecondes par pixels.

Une fois les bornes identifiées, on parcourt la trace afin d'identifier les événements qui y correspondent. On commence par le premier événement courramment affiché. Si c'est la première fois qu'on dessine le graphe, on démarre avec le premier événement dans la trace. Ensuite on avance ou on recule selon le temps de l'événement. S'il dépasse la borne de départ on recule, sinon on avance. On continue le parcours jusqu'à ce qu'on trouve le premier événement qui s'est produit avant la borne de départ. Une fois trouvé, on avance dans la trace jusqu'à ce qu'on trouve le premier événement qui suit la borne de fin.

Avec les événements limites connus, on commence le dessin du graphe de trace. Le dessin se fait par suite de valeurs x, y entre lesquelles on dessine des lignes verticales ou horizontales selon la signification de la suite des événements. Chaque ligne dessinée utilise le dernier point de la dernière ligne dessinée comme point de départ. Lorsqu'une entrée dans le noyau se produit, on dessine une ligne verticale de la dernière sortie jusqu'au processus designé. On dessine ensuite une ligne horizontale jusqu'à l'événement d'entrée dans le système. Pour terminer, on dessine une ligne verticale du processus jusqu'au noyau. Lorsque les événements dessinés appartiennent déjà au noyau, on ne fait que continuer la ligne horizontale jusqu'à l'événement courant. Lorsqu'on traite une trace qui contient des informations concernant RTAI aussi, l'engin de dessin utilise la machine à états décrite précédemment pour dessiner la trace d'une manière similaire. Pour quelques événements significatifs, on dessine aussi une icône distinctive avec un texte correspondant.

Afin de diminuer le nombre de lignes et d'îcones dessinées, on ne dessine que le strict minimum. Pour ce faire, chaque requête de dessin de ligne est différée le plus longtemps possible et les icônes qui chevauchent les précédentes ne sont pas dessinées. Les coordonnées de chaque ligne qui demande à être dessinée sont conservées jusqu'à ce qu'une ligne suivante ait des coordonnées différentes. À ce moment, la ligne différée est effectivement dessinée dans le graphe. Cette technique accélère la dessin de la trace lorsque le nombre de microsecondes par pixel est élevé (zoom out) puisque plusieurs lignes se superposent. De même, une icône n'est dessinée que si l'icône précédente est assez éloignée. Le critère de distance est subjectif mais est fixé de manière à présenter un aspect visuel cohérent.

Chapitre 4

Résultats

Avec les principes de base et les concepts internes de LTT présentés, voyons les résultats d'utilisation de LTT. Cette présentation permettra d'illustrer que LTT comble effectivement les lacunes des autres outils d'analyse en permettant de reconstruire le comportement dynamique du système tout en ayant peu d'impact sur le comportement du système.

4.1 Exemple de trace de Linux

Commençons la discussion par une présentation d'un exemple de trace de Linux. Dans le tableau 4.1 on peut voir un exemple de trace où un processus est mis en attente d'entrée/sortie à cause d'une faute de page le temps que la page en question soit lue du disque et mise en mémoire pour le processus. Pour simplifier l'illustration, l'ID du CPU n'est pas illustré puisque les événements se sont tous produits sur le même CPU.

TAB. 4.1 – Exemple de trace : Service d'une faute de page.

Trap entry	993,455,180,020,524	605	12	TRAP : Data Access ; EIP : 0xC0014290
Memory	993,455,180,020,528	605	11	PAGE ALLOC ORDER : 0
File system	993,455,180,020,556	605	19	START I/O WAIT
Sched change	993,455,180,020,582	0	18	IN : 0 ; OUT : 605 ; STATE : 2
IRQ entry	993,455,180,020,743	0	8	IRQ : 19, IN-KERNEL
Process	993,455,180,020,752	0	15	WAKEUP PID : 605 ; STATE : 2
IRQ exit	993,455,180,020,759	0	6	
Sched change	993,455,180,020,762	605	18	IN : 605 ; OUT : 0 ; STATE : 0
File system	993,455,180,020,763	605	19	END I/O WAIT
Sched change	993,455,180,020,819	0	18	IN : 0 ; OUT : 605 ; STATE : 2
IRQ entry	993,455,180,025,522	0	8	IRQ : 19, IN-KERNEL
Process	993,455,180,025,531	0	15	WAKEUP PID : 605 ; STATE : 2
IRQ exit	993,455,180,025,537	0	6	
Sched change	993,455,180,025,539	605	18	IN : 605 ; OUT : 0 ; STATE : 0
Trap exit	993,455,180,025,591	605	6	
Syscall exit	993,455,180,025,602	605	6	
Trap entry	993,455,180,025,606	605	12	TRAP : Instruction Access ; EIP : 0x30010664

La première colonne à partir de la gauche donne le type de l'événement, la deuxième donne le moment auquel il s'est produit en microsecondes, la troisième donne le PID du processus auquel appartient l'événement, la quatrième donne la quantité d'octets occupés par la description de l'événement dans la trace et la dernière colonne donne quelques détails sur l'événement. Ici, la séquence d'événements est assez simple. Le processus essaie d'accéder à une donnée dans son espace d'adressage. Une faute est générée, toutefois, étant donné que la donnée n'est pas disponible. Le processus est donc mis en attente par le noyau qui ordonnance un autre processus à sa place. Étant donné qu'aucun autre processus n'a besoin du CPU, le noyau donne la main au processus inactif. Quelques temps plus tard, le disque répond à la requête qui lui avait été envoyée et le processus est réordonnancé.

Malgré la simplicité de l'exemple, il permet de voir le niveau de détails qu'on peut obtenir avec LTT. Le déboggeur *gdb*, par exemple, aurait été incapable de nous aider

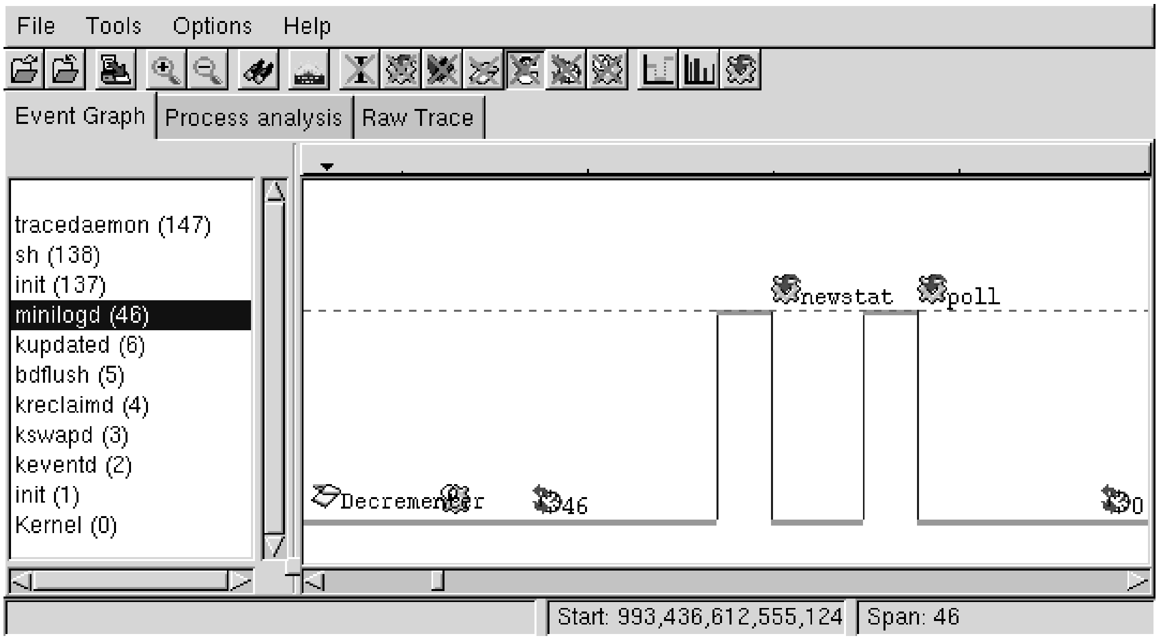


FIG. 4.1 – Exemple de graphe de trace.

à identifier cette séquence d'événements.

Un exemple de graphe dessiné par LTT est donné à la figure 4.1. Dans le haut de la fenêtre affichée on peut voir un menu et une barre d'outils. Trois onglets sont ensuite utilisés pour afficher les données. Le premier présente le graphe. Le second présente les informations concernant la performance. Le dernier contient la liste complète des événements. L'onglet de trace est séparé en deux régions. Dans la partie de gauche, on voit une liste complète des processus actifs durant la trace. La partie de droite contient un graphe d'événements dessiné tel qu'expliqué précédemment. Dans le graphe actuel, on peut voir que le processus *minilogd* est ordonnancé suite à une interruption d'horloge. *minilogd* fait ensuite appel à l'appel système *newstat*. Le noyau effectue un travail en conséquence et retourne le contrôle à *minilogd* de nouveau qui effectue un

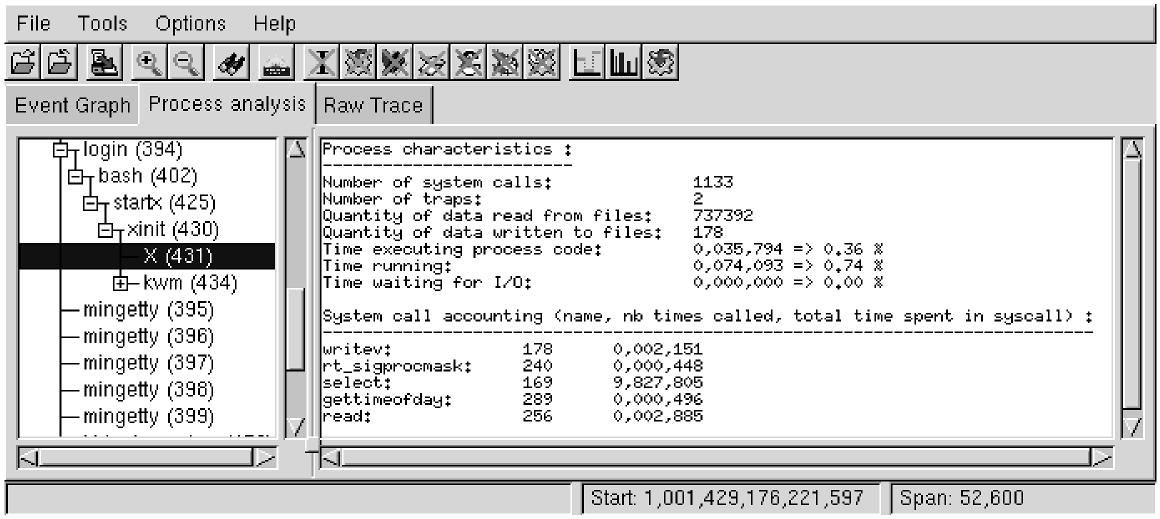


FIG. 4.2 – Exemple d'analyse de performance d'un processus.

appel à *poll*. Le noyau ne trouvant rien qui satisfait le processus, il le met en attente et ordonnance un autre processus.

La figure 4.2 illustre un exemple de l'analyse de performance présentée dans le deuxième onglet. À gauche, on peut voir l'arborescence complète des processus qui existaient durant le traçage. À droite, on peut voir des informations détaillées concernant l'exécution des processus durant l'intervalle de temps tracé. Dans cet exemple-ci, le processus observé est le serveur *X*. On peut ainsi voir que 0.36% du temps a servi à exécuter du code appartenant au serveur même si le processus était ordonnancé pour 0.74% du temps, la différence ayant été utilisée par le noyau pour servir l'application. Aussi, on voit les appels systèmes effectués par l'application, le nombre total de fois qu'ils ont été effectués ainsi que le temps total passé dans chaque type d'appel.

4.2 Impact du traçage sur le système

La principale nouveauté de LTT, hormis l'extensibilité permise par sa structure modulaire ouverte, est la quantité d'information qu'il permet d'obtenir tout en demeurant un outil d'analyse léger et non-instrusif. Cette section étudie l'impact qu'a LTT sur le comportement du système.

Les résultats expérimentaux montrent que LTT est en mesure de fournir des ensembles de données uniques avec très peu de surcoût. En général, un système observé est ralenti d'un maximum de 2.5% lorsqu'on observe les événements de base.

Les expériences ont été menées sur une Pentium II d'Intel roulant à 350MHz avec 128Mo de mémoire principale. La distribution Linux utilisée était RedHat 6.0. Bien sûr, le noyau par défaut était remplacé par un noyau modifié pour LTT. À moins d'indications contraire, toutes les tâches ont été exécutées à partir de la ligne de commande sans aucune interface graphique.

4.2.1 Surcoût global sur le système

Afin d'évaluer le surcoût causé par LTT à un système en fonction, un certain nombre de modifications ont dû être appliquées à la méthode de collecte pour isoler l'impact de chacune de ses composantes sur l'architecture globale. Les configurations suivantes ont été testées :

1. Noyau 2.2.13 d'origine.

2. Noyau 2.2.13 modifié où les événements sont ignorés par l'unité de traçage du

noyau.

3. Noyau 2.2.13 modifié où les événements sont enregistrés, mais le daemon ne roule pas.

4. Noyau 2.2.13 modifié où les événements sont enregistrés et lus, mais pas écrits dans le fichier de sortie.

5. Noyau 2.2.13 modifié où les événements sont enregistrés, lus et écrits dans le fichier de sortie final.

6. Même configuration que la précédente sauf que les détails d'événements sont enregistrés seulement pour les événements de base.

Pour chaque configuration, une série de tâches ont été démarrées et le temps requis pour les compléter a été enregistré. On peut ensuite comparer les temps d'exécution pour déterminer quelles composantes du système de collecte de données ralentissent le système observé. Pour ce faire, les tâches suivantes ont été exécutées :

1. Compilation complète du noyau en utilisant make. Cette tâche utilise de manière intensive le CPU et le système de fichiers.

2. Création d'une archive tar en utilisant un grand nombre de fichiers (800Mo). Cette tâche utilise le système de fichiers de manière intensive.

3. Compression d'une archive du code du noyau (50Mo) en utilisant bzip2. Cette tâche utilise le CPU de manière intensive.

4. Pendant l'utilisation de l'environnement KDE, les tâches suivantes ont été démarrées deux fois en parallèle :

(a) *netscape* a ouvert des pages web de 14 sites différents.

(b) *acroread* a ouvert 7 documents différents.

(c) *staroffice* a ouvert 8 documents différents.

(d) *gnuplot* a dessiné 4 fonctions 14 fois.

Cette tâche utilise intensément les entrées/sorties et les services du système d'exploitation[1].

Les résultats de ces tests sont présentés au tableau 4.2. Pour chaque configuration, exceptée la première, le pourcentage de différence de temps d'exécution est donné entre cette configuration et la configuration de base afin de faciliter l'analyse. Les temps donnés sont en secondes et représentent la valeur moyenne de 10 exécutions[2]. Étant donné la complexité des systèmes Unix modernes, les valeurs de temps données varient. Le tableau 4.3 donne les variances trouvées. Le tableau 4.4 donne les tailles des traces générées et les quantités de données générées par secondes. Ces résultats sont discutés dans les sous-sections suivantes.

4.2.2 Composantes du surcoût

Afin de bien comprendre l'impact des différentes composantes du surcoût de LTT, la discussion qui suit discute des différentes étapes du traçage, du moment où les données sont générées au moment où elles sont disponibles pour traitement sur le

[1]Le système d'exploitation doit s'occuper d'ordonnancer plusieurs tâches en compétition tout en gérant les communication inter-processus avec le serveur X à travers les sockets

[2]Chaque test a en fait été exécuté 11 fois avec le résultat du premier test étant ignoré.

TAB. 4.2 – Temps d'exécution des différentes tâches en secondes selon la configuration.

Configuration	*Compilation*	*Archivage*	*Compression*	*Env. KDE*
1	240.2	357.4	141.2	246.8
2	240.8	358.0	141.4	249.2
△	0.25 %	0.17 %	0.14 %	0.97 %
3	243.6	359.1	141.1	252.2
△	1.42 %	0.48 %	-0.07 %	2.19 %
4	245.7	358.1	141.6	252.9
△	2.29 %	0.20 %	0.28 %	2.47 %
5	246.9	365.5	141.4	258.3
△	2.79 %	2.27 %	0.14 %	4.66 %
6	246.3	363.6	141.6	252.9
△	2.54 %	1.74 %	0.28 %	2.47 %

TAB. 4.3 – Variances sur les temps d'exécution en secondes selon la configuration.

Configuration	*Compilation*	*Archivage*	*Compression*	*Env. KDE*
1	0.4	1.6	0.4	0.9
2	0.6	1.9	0.5	1.9
3	0.5	1.7	0.3	1.6
4	0.5	2.0	0.5	1.1
5	1.1	1.6	0.5	3.9
6	0.5	1.3	0.5	1.8

TAB. 4.4 – Tailles des traces d'événements en mégaoctets selon la tâches.

Tâche	*Conf.*	*Une exéc.* (Mo)	*Taux* (Mo/s)
Compilation	5	33.4	0.135
	6	25.0	0.101
Archivage	5	27.9	0.076
	6	16.5	0.045
Compression	5	1.76	0.012
	6	1.14	0.008
Env. KDE	5	139	0.538
	6	62.9	0.249

disque.

Il est intéressant de noter que l'instrumentation des événements de base cause un surcoût de moins de 2.5% peu importe la tâche exécutée, comme on peut voir en comparant la sixième configuration avec la première au tableau 4.2.

4.2.2.1 Instrumentation du noyau

L'impact de l'instrumentation du noyau peut être vu en comparant la seconde configuration à la configuration de base. Les pourcentages montrent que l'impact de l'instrumentation est minime, sinon insignifiant. L'impact le plus élevé se produit lors de la tâche de test dans l'environnement KDE et est dû à l'utilisation intensive des ressources systèmes. Sinon, il ne semble pas y avoir de ralentissement visible. Nous sommes donc en mesure d'affirmer que le traçage du noyau a peu ou pas d'effet sur la performance du système tant et aussi longtemps que le daemon n'a pas donné d'instructions au module de traçage pour qu'il enregistre les événements dans ses tampons.

4.2.2.2 Enregistrement de la trace

L'impact de l'enregistrement de la trace peut être vu en comparant la troisième configuration à la deuxième. Ici, l'impact varie selon le type de tâche exécutée par le système. Dans les cas où il n'y a qu'un seul processus qui roule en tout moment, tels les tâches d'archivage et de compression, le surcoût est négligeable. Dans les autres cas, ceux de la compilation du noyau et de l'environnement KDE, les résultats semblent indiquer que l'enregistrement des événements dans le module de traçage à l'intérieur d'un verrou noyau (*spin_lock_irq_save*) soit la source de conflits d'ordonnancement.

4.2.2.3 Lecture de la trace

L'impact de la lecture des données du module de traçage par le daemon peut être vu en comparant la quatrième configuration à la troisième. Le surcoût de cette lecture est petit, mais est en corrélation avec le surcoût de l'enregistrement de la trace.

4.2.2.4 Écriture de la trace

L'impact de l'écriture de trace de données sur disque par le daemon peut être vu en comparant la cinquième configuration à la quatrième. Les tâches qui sont composées de plusieurs processus qui se partagent le CPU sont les plus touchées par le traçage. Dans ces cas, les divers processus doivent partager le CPU avec le daemon qui doit écrire de grandes quantités de données au disque. La compilation est composée de plusieurs étapes et nécessite l'utilisation de plusieurs fichiers temporaires qui sont effacés avant d'être écrits sur disque et utilise, donc, plus le CPU que le disque. La tâche d'archivage voit son surcoût augmenter puisque le processus d'archivage doit maintenant partager le CPU et le disque avec une autre tâche, le daemon.

4.2.3 Espace utilisé

Lorsque compilé avec le support pour le traçage, le noyau modifié voit sa taille augmenter de 4Ko. Cette augmentation inclut le module de traçage et les modifications au noyau. Étant donné les ressources actuelles, cette augmentation ne cause aucun problème. Lorsque le traçage est activé, l'espace utilisé par le double-tampon s'ajoute à la taille dynamique du noyau.

La quantité d'espace disque utilisée par les traces peut être vue en regardant le tableau 4.4. On peut voir que la taille varie selon la tâche exécutée et peut être assez grande. Il est clair que la taille de la trace est proportionnelle au nombre d'événements qui se produisent. Pour les tâches qui utilisent le CPU intensivement, comme la compression, les résultats montrent que la quantité de données est minime. Pour les tâches qui utilisent le système d'exploitation intensivement mais qui n'utilisent pas l'environnement graphique, telles que la compilation et l'archivage, la quantité de données générée est sous 10Mo par minute.

Les traces les plus volumineuses sont générées lorsque le système est en mode graphique. Dans ce cas, la quantité de données générée par minute est au dessus de 30Mo dans le pire des cas. Ces chiffres sont dus à l'interaction constante entre les différentes applications pour permettre l'affichage à travers le serveur X. Lorsqu'on ne trace que les détails des événements de base, la quantité de données générée baisse à 15Mo par minute.

4.2.4 Discussion des coûts d'usage

Comme on a pu le voir dans la section précédente, certains aspects de LTT utilisent une grande quantité de ressources systèmes : les accès disque et l'espace disque principalement. Afin de réduire le surcoût causé par la lecture de la trace et son écriture au disque, plusieurs mécanismes peuvent être pris en considération. Une solution serait l'utilisation du mécanisme *mmap* disponible dans Linux. Cette technique permettrait au daemon de voir le tampon de données du module de traçage dans son espace d'adressage. On éviterait ainsi d'avoir à copier des données du noyau à l'espace usager puisque les écritures auraient comme paramètre le tampon noyau maintenant accessible aux deux.

Une autre technique possible serait d'alimenter les traces directement au système de fichiers ou au pilote de périphérique du disque de l'intérieur du noyau sans passer par le daemon. Cette technique comporte des problèmes qui dépassent le cadre de la présente discussion.

La réduction de la taille de la trace diminuerait le nombre d'accès au disque et l'espace utilisé sur le disque. Les types et les tailles de données et de structures ont déjà été optimisés afin d'en réduire la taille. Néanmoins, on pourrait envisager l'utilisation d'algorithmes de compression afin de prendre avantage de la régularité trouvée dans les traces et, ainsi, réduire leur taille. En utilisant des outils de compression disponibles, on a trouvé que les traces peuvent être réduites par un facteur de 10. Ceci voudrait dire que le taux de 30Mo/minute observé pourrait être réduit à 3Mo/minute. Des tests seraient nécessaires afin de déterminer le coût en temps CPU de la méthode de compression choisie.

TAB. 4.5 – Temps (en secondes) d'activité pour chaque application observée.

Application	*time*		*LTT*		
	Usager	*Système*	*Usager*	*Système*	*Attente E/S*
Netscape	6.20	2.52	6.44	1.74	0.80
StarOffice	3.02	3.40	4.36	0.84	0.43
X11amp	1.64	1.48	1.61	0.66	0

4.3 Caractérisation d'une station de travail

Afin de caractériser une station de travail, un ensemble d'applications doit être choisi pour représenter l'utilisation d'un système comme station de travail. Ensuite, des mesures peuvent être prises avec LTT et avec des outils conventionnels. L'exactitude des outils conventionnels peut ensuite être évaluée en comparaison avec les données trouvées dans la trace.

Pour cet exemple, les applications suivantes ont été observées sur une période de 90 secondes : Netscape, StarOffice et X11amp. Durant cette période de temps, le daemon de LTT enregistrait les événements et l'outil *time* était utilisé pour extraire les informations accumulées par échantillonnage par Linux. Les applications ont été utilisées d'une manière typique. *Netscape* a été utilisé pour visualiser 2 sites web. Un document était modifé dans *StarOffice*. *X11amp* lisait un fichier mp3. Les résultats des 90 secondes d'exécution sont donnés au tableau 4.5.

Pour chaque outil, la colonne *Usager* donne le temps que le système exécutait du code appartenant à l'application, tel que vu par l'outil désigné. La colonne *Système* donne te temps que le noyau a passé à desservir l'application, tel que vu par l'outil désigné. La colonne *Attente E/S* montre le temps passé par la tâche en attente d'entrées/sorties, tel que vu par LTT.

Les résultats montrent que les outils d'analyse conventionnels, *time* dans ce cas-ci, ne réussissent pas à fournir à l'observateur une appréciation correcte du fonctionnement du système. Entre autres, *time* rapporte que *StarOffice* a roulé 3.40s en mode système alors que le vrai chiffre est 0.84s. Il en fait de même avec *X11amp* et rapporte que cette application a roulé 1.48s en mode système alors que le vrai chiffre est 0.66s. En plus, avec les outils conventionnels il n'y a aucun moyen de connaître la quantité de temps pendant laquelle l'application n'a pu être ordonnancée étant donné qu'elle était en attente d'entrées/sorties. LTT est en mesure de fournir cette information. Il est aussi important de connaître le temps pendant lequel le processus inactif était ordonnancé. Ici, le processus inactif était ordonnancé pendant 75.4% du temps observé.

4.4 Comparaison avec les outils conventionnels

L'intérêt principal de LTT étant le fait qu'il fournit de nouvelles informations, il est important de comparer LTT avec des outils d'analyse existants. Pour ce faire, les paragraphes qui suivent comparent LTT avec *gdb*, *ps*, *gprof* et *time*.

La section 4.1 présentait un exemple de trace qui démontrait que *gdb* est un outil inadéquat dans certaines situations. Un exemple encore plus clair des limites de *gdb* est la résolution de problèmes de synchronisation. Étant donné qu'il modifie le comportement de l'application observée par l'utilisation du service *ptrace* du noyau, il est souvent impossible de reproduire des problèmes de synchronisation avec *gdb* et, encore moins, de les résoudre.

De plus, certains problèmes de synchronisation peuvent disparaître lorsque *gdb* est utilisé. Dans les cas les plus simples, l'insertion de *printf*s dans le code observé permet aux développeurs de résoudre les problèmes de synchronisation. Dans des systèmes logiciels complexes, cette technique est cependant inefficace. En ce sens, LTT offre une solution à ces problèmes en fournissant aux développeurs la séquence exacte des événements tels qu'ils se sont produits sur le système observé. La résolution des problèmes de synchronisation se fait par l'analyse de la séquence des événements. De plus, LTT ne modifie pas la séquence des événements du système observé puisque les événements sont enrégistrés dans leur ordre d'arrivée et les verroux d'écriture sont maintenus pour des prériodes de temps très courtes.

Comme il a été démontré plus haut, LTT aide aussi à la compréhension des différentes composantes de la performance d'un système. En fait, les informations fournies par LTT reflètent plus fidèlement le comportement du système ou du processus observé. Ceci a été démontré pour *ps*. Il est important de noter que même si *ps* n'est pas le seul outil d'analyse utilisé dans les systèmes Unix modernes et n'est certainement pas le plus précis, il demeure quand même le plus important étant donné son usage très répandu et son adoption comme moyen légitime pour mesurer la performance des systèmes.

Un autre outil souvent utilisé pour mesurer la performance est *gprof*. Contrairement à *ps*, il est utilisé pour analyser la performance d'une seule application. Ici encore, les résultats recueillis sont tout au plus statistiques. Pour illustrer cette réalité, la portion de code suivante a été analysée avec *gprof* :

```
int i, j;
void fct_A(void)
{ for(i = 0, j = 0; j < 10000000; j++)
    i++;
  printf("i's value : %d \n", i);
}
void fct_B(void)
{ for(j = 0; j < 1000000; j++)
    sched_yield();
}
int main(void)
{
  fct_A();
  fct_B();
}
```

En utilisant *gprof*, on apprend que l'application a pris 250ms pour s'exécuter. 170ms ont été passées dans *fct_A* et le reste, 80ms, ont été passées dans *fct_B*. En utilisant LTT on trouve que l'application a, en fait, été ordonnancée pendant 3.28s et que 2.34s ont été passées à exécuter du code appartenant à l'application. Le reste du temps a été pris par le noyau pour exécuter des services pour l'application. Il est aussi intéressant de noter que LTT rapporte que l'application a passé 972ms à l'intérieur des appels consécutifs à *sched_yield*. Ce temps n'a pas été vu par *gprof* qui a attribué seulement 80ms à *fct_B*. De plus, *time* corrobore les résultats de LTT et rapporte que l'application s'est exécutée pendant 3.64s. *time*, par contre, rapporte que 2.71s ont été passées à exécuter du code système contre seulement 0.61s pour le code de l'application. Ces données ont été calculées en fonction des statistiques accumulées par le noyau. Il s'agit, en fait, de résultats similaires à ceux rapportés par *ps* qui est connu pour son manque de précision.

La différence entre les résultats de LTT et ceux de *gprof* est due à l'usage de méthodes différentes pour la collecte de données. *gprof* utilise l'horloge système pour échantillonner le comportement de l'application. Avec LTT, on peut voir les minuteries de *gprof* s'enclencher pour échantillonner le code et générer des *sigreturn*s par la suite. En observant la trace, on peut voir que l'échantillonnage se produit souvent à l'intérieur d'un appel à *sched_yield*. Ce faisant, le système est en train d'exécuter du code noyau et le temps depuis le dernier échantillon est attribué au système. C'est pour cette raison que *time* indique des valeurs éronnées et c'est pour cette même raison de *gprof* se trompe dans ses chiffres. La même expérience a été effectuée avec *gettimeofday* à la place de *sched_yield* et a produit des résultats similaires.

4.5 Observations et caractérisations de tâches temps-réels

Maintenant que nous avons couvert les différents aspects de performance et de comparaison de LTT, voyons un exemple d'observation de tâche temps-réel avec LTT. Pour ce faire, l'exemple Jepplin distribué avec RTAI est utilisé.

L'exemple Jepplin est composé d'un ensemble de tâches temps-réels qui communiquent ensemble en utilisant des mailboxes et des sémaphores. L'ordonnancement des tâches dépend des messages qu'elles attendent. Inversement, les messages envoyés par une tâche peuvent enclencher l'ordonnancement d'une autre tâche temps-réel. Une portion des interactions de contrôle des diverses tâches de cet exemple est illustrée à la figure 4.3. La figure 4.4 montre la liste textuelle des événements illustrés.

On peut voir les transitions d'états sur le graphe, les détails des événements qui les causent et leur impact sur le système. De fait, on peut voir comment les communications entre les divers processus génèrent des changements de contexte. S'il y avait des problèmes de synchronisation, il serait facile de les identifier avec ce type de vue.

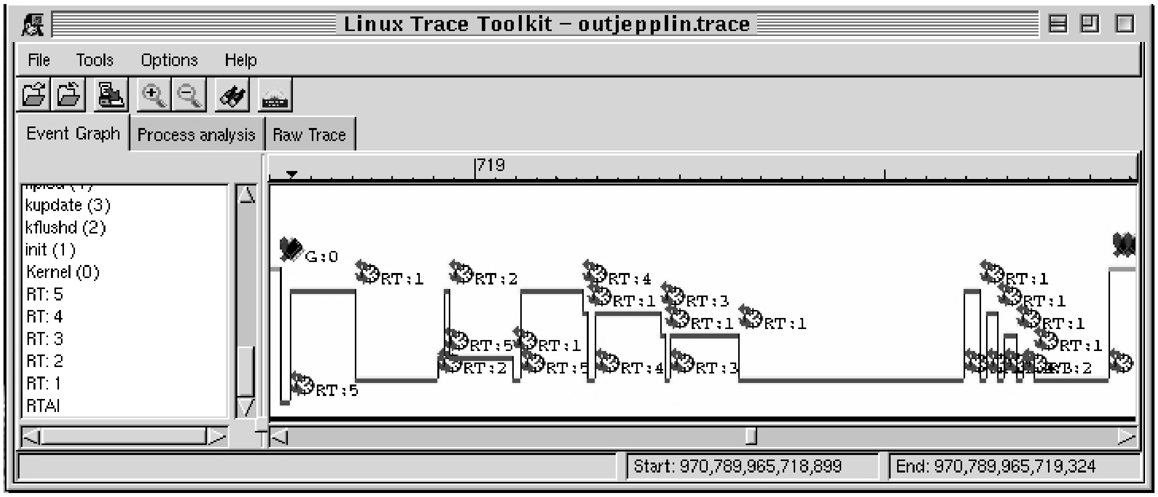
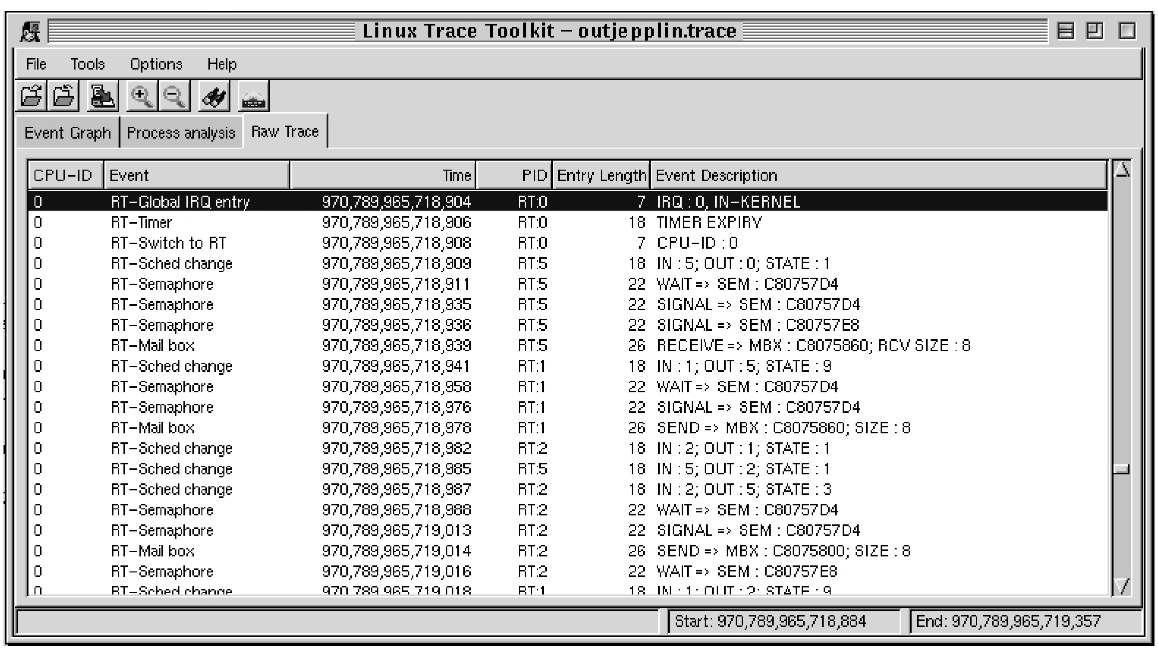


FIG. 4.3 – Graphe de contrôle des communications dans l'exemple Jepplin.



CPU-ID	Event	Time	PID	Entry Length	Event Description
0	RT-Global IRQ entry	970,789,965,718,904	RT:0	7	IRQ : 0, IN-KERNEL
0	RT-Timer	970,789,965,718,906	RT:0	18	TIMER EXPIRY
0	RT-Switch to RT	970,789,965,718,908	RT:0	7	CPU-ID : 0
0	RT-Sched change	970,789,965,718,909	RT:5	18	IN : 5; OUT : 0; STATE : 1
0	RT-Semaphore	970,789,965,718,911	RT:5	22	WAIT => SEM : C80757D4
0	RT-Semaphore	970,789,965,718,935	RT:5	22	SIGNAL => SEM : C80757D4
0	RT-Semaphore	970,789,965,718,936	RT:5	22	SIGNAL => SEM : C80757E8
0	RT-Mail box	970,789,965,718,939	RT:5	26	RECEIVE => MBX : C8075860; RCV SIZE : 8
0	RT-Sched change	970,789,965,718,941	RT:1	18	IN : 1; OUT : 5; STATE : 9
0	RT-Semaphore	970,789,965,718,958	RT:1	22	WAIT => SEM : C80757D4
0	RT-Semaphore	970,789,965,718,976	RT:1	22	SIGNAL => SEM : C80757D4
0	RT-Mail box	970,789,965,718,978	RT:1	26	SEND => MBX : C8075860; SIZE : 8
0	RT-Sched change	970,789,965,718,982	RT:2	18	IN : 2; OUT : 1; STATE : 1
0	RT-Sched change	970,789,965,718,985	RT:5	18	IN : 5; OUT : 2; STATE : 1
0	RT-Sched change	970,789,965,718,987	RT:2	18	IN : 2; OUT : 5; STATE : 3
0	RT-Semaphore	970,789,965,718,988	RT:2	22	WAIT => SEM : C80757D4
0	RT-Semaphore	970,789,965,719,013	RT:2	22	SIGNAL => SEM : C80757D4
0	RT-Mail box	970,789,965,719,014	RT:2	26	SEND => MBX : C8075800; SIZE : 8
0	RT-Semaphore	970,789,965,719,016	RT:2	22	WAIT => SEM : C80757E8
0	RT-Sched change	970,789,965,719,018	RT:1	18	IN : 1; OUT : 2; STATE : 9

FIG. 4.4 – Liste des événements de communication de l'exemple Jepplin.

Conclusion

Dans ce mémoire nous avons présenté une nouvelle technique de collecte et d'analyse de systèmes telle qu'implémentée dans le Linux Trace Toolkit. Les résultats accumulés montrent que le surcoût de LTT est minime et que les ensembles de données accumulés par LTT sont uniques. Ces ensembles de données ont été utilisés pour reconstruire efficacement le comportement dynamique des systèmes observés. Nous avons aussi montré que les outils d'analyse conventionnels sont imprécis et qu'il serait préférable d'utiliser l'instrumentation noyau lorsqu'une caractérisation précise est nécessaire.

LTT est disponible depuis un certain temps à travers le site web du projet [14]. Plusieurs usagers l'ont déjà utilisé pour tracer et mesurer la performance de leurs systèmes. LTT a aussi été le sujet de plusieurs articles [39, 36, 38, 37]. Au stade actuel du dévelopement de Linux, LTT s'est démarqué comme l'outil de traçage de choix et fait partie de deux distributions commerciales, celle de Lineo et celle de MontaVista. Afin de promouvoir l'usage de LTT dans plusieurs situations, nous avons collaboré avec IBM afin de fournir un système de traçage universel en permettant l'ajout dynamique de points de traçage à l'aide du Dynamic Probes d'IBM [15]. Cette interface a nécessité l'ajout d'une fonctionnalité permettant la création dynamique d'événements dans LTT. Ceci augmente la flexibilité de LTT puisqu'il n'est plus limité aux événements prédéfinis.

Plusieurs possibilités de recherches futures sont intéressantes. Premièrement, il serait intéressant de pouvoir collecter des traces de plusieurs machines différentes pour ensuite effectuer une analyse de l'ensemble des machines. Ce type d'analyse donnerait accès à plusieurs informations sur les systèmes distribués qui ne sont pas couramment disponibles.

Étant donné la précision des résultats obtenus par LTT, il serait intéressant de voir les résultats obtenus par LTT réutilisés dans l'ordonnancement futur des tâches et le partage des ressources. Une compagnie utilise déjà les résultats accumulés sur les tâches temps-réels par LTT pour prévoir l'ordonnancement futur des tâches [16].

Aussi, étant donné les événements noyau tracés, il serait possible d'utiliser LTT comme base pour un outil d'analyse d'intrusion de sécurité. Des événements ou des suites d'événements pourraient ainsi déclencher des réactions qui pourraient être programmées.

Afin de simplifier l'usage de LTT et encourager son adoption, nous travaillons à faire intégrer les instructions d'instrumentation et le module de traçage à l'intérieur du code du noyau. De la même manière, nous travaillons à l'intégration de LTT dans diverses distributions. À cet égard, Debian contient déjà un paquetage LTT dans sa branche de dévelopement.

Les outils dévelopés dans le cadre du Linux Trace Toolkit sont modulaires, extensibles et disponibles ouvertement, facilitant ainsi leur extension et adaptation. Ce type d'outils sera de plus en plus nécessaire aux processus de dévelopements de logiciels étant donné l'accroissement constant de la complexité des logiciels et des composantes matérielles faisant partie des systèmes informatiques modernes.

Bibliographie

[1] Rational Software's Quantify,

http ://www.rational.com/products/quantify.

[2] GNU General Public License version 2,

http ://www.gnu.org/copyleft/gpl.html.

[3] IBM's AIX trace facility,

http ://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/

aixprggd/genprogc/trace_facility.htm.

[4] SUN's Trace Normal Format tools,

http ://www.sun.com/smcc/solaris-migration/docs/whitepapers.html#TNF.

[5] Program Visualization at IBM Research

http ://www.research.ibm.com/pvres/.

[6] WindRiver's WindView,

http ://www.windriver.com/products/html/windview2.html.

[7] QNX's DejaView,

http ://www.qnx.com.

[8] Etnus' TimeScan,

http ://www.etnus.com/products/tsproduct/index.html.

[9] Lauterbach's Trace32,

http ://www.lauterbach.com.

[10] TimeSys' TimeTrace,

http ://www.timesys.com.

[11] Nematron's HyperKernel Trace Utility,

http ://www.nematron.com/solutions/software/hyperkernel/hyperkernel.html.

[12] RTAI,

http ://www.aero.polimi.it/projects/rtai/.

[13] RTLinux,

http ://www.rtlinux.org.

[14] Linux Trace Toolkit home page,

http ://www.opersys.com/LTT.

[15] IBM's DProbes,

http ://oss.software.ibm.com/developer/opensource/linux/projects/dprobes.

[16] Tripac Software,

http ://www.tripac.com.

[17] G. Ammons, T. Ball, and J. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[18] M. Bach. *The Design of the Unix Operating System.* Prentice Hall, 1986.

[19] T. Ball and J. Larus. Efficient Path Profiling. In *Proceeding of MICRO-29*, 1996.

[20] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 2001.

[21] P. Cloutier. DIAPM RTAI Position Paper. In *Proceedings of the 2nd Real Time Linux Workshop*, 2000.

[22] J. Anderson et al. Continuous Profiling : Where Have All the Cycles Gone. In *16th ACM Symposium on Operating Systems Principles*, 1997.

[23] J. Dean et al. ProfileMe : Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of IEEE Micro-30*, 1997.

[24] M. Beck et al. *Linux Kernel Internals : Second Edition.* Addison-Wesley, 1998.

[25] X. Zhang et al. System Support for Automatic Profiling and Optimization. In *16th ACM Symposium on Operating Systems Principles*, 1997.

[26] S. Graham, P. Kessler, and M. McKusick. gprof : A call graph execution profiler. In *SIGPLAN Notices*, 1982.

[27] A. Ivchenko. Real-time linux. *Embedded Systems Programming*, pages 35–57, May 2001. Sidebar p.36.

[28] J. Larus and E. Schnarr. EEL : Machine-Independent Executable Editing. In *Proceeding of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[29] S. Maxwell. *Linux Core Kernel Commentary.* Coriolis, 1999.

[30] S. McCanne and C. Torek. A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling. In *Proceedings of the Winter USENIX Conference*, 1993.

[31] P. Montegazza. DIAPM RTAI for Linux : WHYs, WHATs and HOWs. In *Proceedings of the 1rst Real Time Linux Workshop*, 1999.

[32] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, pages 78–103, January 1997.

[33] A. Rubini. *Linux Device Drivers*. O'Reilly, 1998.

[34] David A. Salomon. *Inside Windows NT : Second Edition*. Microsoft Press, 1998.

[35] Uresh Vahalia. *Unix Internals : The New Frontiers*. Prentice Hall, 1996.

[36] K. Yaghmour. Monitoring and Analyzing RTAI System Behavior Using the Linux Trace toolkit. In *Real Time Linux Workshop*, 2000.

[37] K. Yaghmour. Tracing real-time application interface using the linux trace toolkit. *Embedded Linux Journal*, pages 39–42, March/April 2001.

[38] K. Yaghmour and M. Dagenais. The linux trace toolkit. *Linux Journal*, pages 54–57, May 2000.

[39] K. Yaghmour and M. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.