Titre: Title:	Analyse automatisée des causes de blocage de processus à partir d'une trace d'exécution
Auteur: Author:	Pierre-Marc Fournier
Date:	2009
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Fournier, P.-M. (2009). Analyse automatisée des causes de blocage de processus à partir d'une trace d'exécution [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/117/

UNIVERSITÉ DE MONTRÉAL

ANALYSE AUTOMATISÉE DES CAUSES DE BLOCAGE DE PROCESSUS À PARTIR D'UNE TRACE D'EXÉCUTION

PIERRE-MARC FOURNIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MARS 2009

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

ANALYSE AUTOMATISÉE
DES CAUSES DE BLOCAGE DE PROCESSUS
À PARTIR D'UNE TRACE D'EXÉCUTION

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

M. BOIS Guy, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. BOYER François-Raymond, Ph.D., membre

REMERCIEMENTS

Je remercie Michel Dagenais, mon directeur de recherche, pour ses conseils, sa disponibilité et l'aide financière qu'il m'a versée. Je le remercie également de m'avoir donné l'opportunité d'assister à diverses conférences, ce qui a largement contribué à ma formation et à mes contacts.

Je remercie Dominique Toupin et Marco Massé d'Ericsson de m'avoir donné la chance de travailler avec eux durant l'été 2008.

Finalement, je remercie mon collègue Mathieu Desnoyers pour le temps qu'il a consacré aux discussions avec moi et qui ont permis de faire avancer mon projet et mes connaissances dans le domaine du traçage.

RÉSUMÉ

La tendance actuelle à la parallélisation fait que les développeurs sont de plus en plus fréquemment confrontés à des bogues de performance dont la cause est difficiles à repérer. Ceux-ci sont fréquemment dus à des interactions imprévues entre des composants logiciels qui s'exécutent parallèlement. L'intégration grandissante et la complexité liée au débogage des grands systèmes informatiques exacerbent ce problème. Un type de bogue de performance particulièrement difficile est celui dont la réelle cause est séparée du symptôme par une chaîne de blocages. Les outils actuels sont d'une aide limitée pour résoudre ces bogues.

L'objectif de ce travail est donc la conception d'un outil aidant au débogage de problèmes de performance impliquant des chaînes de blocages. Ce mémoire introduit cette nouvelle approche et traite de son implémentation dans l'Analyseur de délais de LTTV. Le Linux Trace Toolkit (LTTng) est utilisé pour l'enregistrement de traces et la majorité de l'instrumentation, permettant le traçage de systèmes en production avec grande précision et un impact minimal sur la performance. Cette approche utilise exclusivement de l'instrumentation noyau et ne requiert pas la recompilation des applications. L'outil d'analyse produit un rapport qui indique en détail de quelle façon le temps a été consommé dans un processus donné entre deux événements. Pour chacune des catégories, un autre rapport indique la liste des intervalles de temps où le processus a été dans cet état. Enfin, dans les cas où le processus était bloqué, la chaîne de blocages complète est affichée.

L'Analyseur de délais de LTTV a été utilisé pour analyser et corriger rapidement des problèmes de performance complexes, chose impossible avec les outils existants. Le temps d'analyse est linéaire par rapport à la taille de la trace. L'usage de la mémoire lors de l'analyse de grandes traces est linéaire par rapport à la taille de la trace, mais une méthode pour la rendre constante est décrite.

La méthode conçue pourrait servir de base à des travaux futurs, dont l'analyse de

chaînes de blocages qui impliquent plusieurs ordinateurs ou qui impliquent à la fois des machines physiques et les machines virtuelles qu'elles hébergent.

ABSTRACT

The current trend towards parallelization puts developers more and more in situations where they are confronted with performance bugs whose cause is difficult to indentify. These are frequently due to unexpected interactions between software components that execute concurrently. The tighter integration and the complexity of debugging large systems worsen this problem. One type of bug which is particularly difficult to locate is a performance problem whose root cause is separated from its symptom by a chain of blockings. Current tools provide little help with these problems.

The aim of this work is therefore to design a tool that helps debugging performance problems involving chains of blockings. This thesis introduces this new approach and discusses its implementation in the LTTV Delay Analyzer. The Linux Trace Toolkit (LTTng) is used for trace recording and most of the instrumentation, allowing the tracing of production systems with great precision and minimal performance impact. This approach uses solely kernel instrumentation and does not require the recompilation of applications. The analysis tool produces a report that shows in detail in what way time was spent in a process between two given events. For each category, another report shows the list of time spans during which the process was in that state. Finally, in cases where the process was blocked, the complete chain of blockings is displayed.

The LTTV Delay Analyzer was used to analyze and fix quickly complex performance problems, something impossible with existing tools. Analysis time grows linearly with trace size. Memory usage during the analysis of large traces grows linearly with trace size, but a strategy to make it constant is described.

This new method could serve as a starting point for future work, including the analysis of blocking chains that involve many computers or that involve physical machines as well as the virtual machines they host.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API :	Application programming interface (interface de programmation)
CPU :	Central processing unit (microprocesseur)
ID :	Identificateur
IO :	Input/output (entrée/sortie)
IP :	Instruction pointer (pointeur d'instruction)
IRQ :	Interrupt request (requête d'interruption)
KDE :	K Desktop Environment
LTTng :	Linux Trace Toolkit Next Generation
LTTV :	LTTng Viewer
MPI :	Message Passing Interface
NMI :	Non-maskable interrupt (interruption non masquable)
RAM :	Random access memory (mémoire vive)
TLB :	Translation lookaside buffer
USB :	Universal Serial Bus
WIB :	Working/Interrupted/Blocked
X :	X Window System

INTRODUCTION

Plusieurs éléments dans le contexte actuel rendent le débogage des systèmes informatiques de plus en plus difficile.

Premièrement, on observe une tendance à la parallélisation. Il s'agit d'une approche adoptée afin de maintenir une croissance maximale des performances malgré la pression de plus en plus forte de contraintes physiques et de murs technologiques[1] sur l'architecture utilisée aujourd'hui dans la grande majorité des ordinateurs[1]. La manifestation la plus récente de cette tendance est l'apparition massive sur le marché de processeurs multi-coeurs. La diffusion de cette technologie est telle qu'il est aujourd'hui difficile de trouver un ordinateur neuf qui ait un seul coeur. Une autre manifestation de la tendance à la parallélisation se situe non pas au niveau des coeurs, mais à celui des ordinateurs. Les compagnies qui ont besoin de quantités considérables de puissance de traitement font de plus en plus appel à de grands nombres d'ordinateurs qui travaillent conjointement pour effectuer des calculs ou offrir des services. Un des exemples les plus éminents est la grappe Google[2]. Pour tirer profit de ces nouvelles technologies, les logiciels multi-processus, multi-fils et la communication interprocessus sont de plus en plus utilisés.

Deuxièmement, il existe une tendance à l'intégration des logiciels, autant à l'échelle d'un ordinateur qu'à l'échelle d'Internet. En effet, les logiciels sur un même ordinateur communiquent de plus en plus entre eux, mais également avec d'autres logiciels sur d'autres ordinateurs, via Internet. Ce nombre accru de liens de communications

[1] Ces contraintes et murs sont : (1) L'augmentation du courant de fuite des transistors menant à une consommation de puissance excessive. (2) Les gains d'une plus haute fréquence de CPU sont en partie annulés par la latence des accès mémoire, dont la performance n'a pas augmenté aussi rapidement que celle des processeurs. (3) Le goulot d'étranglement de Von Neumann (*Von Neumann bottleneck*) fait en sorte que certaines applications deviennent moins efficaces à mesure que la vitesse des processeurs augmente. Pour plus de détails, voir [1].

complexifie les systèmes informatiques en augmentant les dépendances entre leurs divers composants logiciels.

Troisièmement, les ordinateurs sont de plus en plus utilisés dans des contextes de production où le débogage est sévèrement contraint car il risque de changer le comportement du système de façon inacceptable. Par exemple, un système de manipulation de vidéo en temps réel qui a des problèmes de performance peut difficilement être débogué avec un outil qui change de façon notable son temps d'exécution ou qui demande une intervention interactive. Un autre exemple est celui du serveur en production. Ainsi, il est fréquent que la surcharge système engendrée par l'utilisation d'outils de débogage soit inacceptable. Or, certains bogues sont impossibles à reproduire hors d'un environnement de production[2].

Que ces facteurs se manifestent simultanément ou un à la fois présente des défis en matière de débogage. La parallélisation et l'intégration multiplient le nombre de bogues dont la cause dépasse les bornes d'un seul processus. Souvent, ces bogues affectent seulement la performance sans offrir au programmeur un processus planté dont la pile donnerait un point de départ pour la recherche de bogue. En outre, la performance des sytèmes en production doit être maintenue.

Or, les outils de débogage traditionnels sont souvent inadéquats pour la résolution de tels bogues[3, 4, 5, 6]. Il existe donc un besoin pour des outils pouvant aider les programmeurs à trouver les bogues de performance dans des systèmes informatiques modernes. Ce projet de recherche vise à développer un tel outil.

Lorsqu'il recherche la cause d'un bogue lié à la performance, le programmeur cherche à savoir ce qui s'est passé dans le système durant la période où les per-

[2]Ce peut être le cas parce que l'infrastructure à reproduire est trop élaborée, ou encore parce que le bogue se produit très rarement sur une machine donnée (mais souvent dans une grappe de milliers d'ordinateurs).

formances n'étaient pas satisfaisantes. Cela revient à examiner ce qui a empêché un processus de passer d'un état à un autre plus rapidement. Un bogue dans ses algorithmes internes qui fait qu'il a besoin de plus de temps de processeur qu'il ne devrait représente le problème le plus facile à trouver ; les profileurs tels que gprof[7] sont très aptes à aider le programmeur à trouver les lignes de code problématiques dans ce cas. Mais ce cas n'est que la pointe de l'iceberg.

La cohabitation d'applications indépendantes sur les mêmes systèmes fait qu'une mauvaise performance dans une application peut être due à une autre, complètement indépendante. Les profileurs pour système complet[8, 9, 10] aident à comprendre comment s'est réparti le temps d'exécution entre les processus et même à l'intérieur de chaque processus.

Un autre type de problème de performance est lié au blocage d'un processus, typiquement dans un appel système, en attente d'une réponse du matériel (par exemple disque dur) ou d'un autre processus dont il dépend. Souvent, une chaîne de plusieurs blocages sépare le symptôme de la cause. Dans ces cas, le fait de savoir qu'un processus est bloqué, même lorsqu'on sait dans quel appel système, est d'une utilité limitée. Il faut une manière de suivre la chaîne des blocages jusqu'à sa source pour comprendre pourquoi elle n'a pas été débloquée plus tôt. Bien que les systèmes parallèles soient très vulnérables à ce type de problème, les outils de débogage actuels sont d'une aide limitée pour y faire face. Ce projet de recherche vise à remédier à cette situation par la conception d'un outil aidant au débogage de problèmes de performance impliquant des chaînes de blocages.

Objectifs

Les objectifs suivants ont été fixés.

- Instrumenter le noyau de Linux de façon à obtenir dans des traces toute l'information nécessaire pour extraire les causes de blocage des processus, tant dues à d'autres processus qu'à du matériel.
- Concevoir des algorithmes permettant d'extraire les liens de dépendance entre les processus d'un système, à partir d'une trace de ce système, en utilisant une quantité raisonnable de mémoire et dans un temps raisonnable.
- Présenter ces liens au programmeur d'une façon qui aide à la résolution de bogues de performance.

Plan du mémoire

Au chapitre 1, une revue de littérature est présentée dans laquelle sont énumérés les outils et techniques actuellement disponibles pour le débogage de problèmes de performance et plus spécifiquement pour ceux liés au blocage. Ensuite, le chapitre 2 consiste en un article intitulé *Analyzing Blockings to Debug Performance Problems on Multi-Core Systems*. Cet article présente la solution utilisée ainsi que les résultats principaux obtenus. Vient ensuite le chapitre 3 où des résultats supplémentaires sont décrits. Enfin, le chapitre 4, consiste en une discussion complémentaire à celle qui se retrouve dans l'article.

CHAPITRE 1

REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art des techniques de débogage pour les bogues de performance. D'abord, l'application des techniques classiques de débogage aux problèmes de performance est discutée dans la section *Techniques classiques de débogage*. Ensuite, la section *Techniques récentes de débogage* décrit des techniques plus avancées pour résoudre les bogues sur les ordinateurs modernes, mais qui ne sont pas directement conçues pour aider à l'analyse des blocages. Enfin, les outils directement conçus pour le débogage de blocages sont abordés dans la section *Outils d'analyse des causes de blocage*.

L'objectif est de donner, pour chaque outil ou technique présenté, une bonne idée des problèmes de performance qu'il aide à déboguer tout en mentionnant son niveau d'aptitude au débogage de problèmes de blocage. Ceci permettra une comparaison avec la solution proposée dans le chapitre suivant.

1.1 Techniques classiques de débogage

Des techniques de débogage développées durant les années 1970, 1980 et 1990, plusieurs sont encore utilisées aujourd'hui. Les débogueurs, les profileurs, les métriques de performance du système d'exploitation, par exemple, sont encore utiles dans bien des cas. En effet, si on retrouve aujourd'hui plus de programmes complexes qu'il y a 10 ou 20 ans, il s'en développe encore de fort simples. Ces outils sont donc encore valables et ils ont été raffinées depuis leur invention jusqu'à aujourd'hui.

Dans cette section, nous présentons ces outils « classiques » et discutons de leur efficacité pour résoudre des problèmes de performance d'aujourd'hui.

Nous n'aborderons pas les débogueurs, comme gdb[11] par exemple, malgré qu'ils soient un des premiers outils qui viennent à l'esprit lorsqu'on parle d'outil de débogage traditionnel. Le débogueur est un outil précieux pour étudier le fonctionnement structurel d'un programme. Il permet au développeur de le stopper selon diverses conditions, d'en inspecter des variables, voire de modifier dynamiquement son état. Cependant, il ne fournit pas d'information sur l'aspect temporel de son exécution. Son utilité est donc très limitée dans le contexte de problèmes de performance, dont la nature même est temporelle. C'est pourquoi il n'en sera pas question ici.

1.1.1 Techniques élémentaires de traçage

Une des techniques les plus élémentaires et les plus anciennes de débogage consiste à imprimer des messages, à l'écran ou ailleurs, à des moments précis de l'exécution. S'il choisit correctement ces moments et les informations à imprimer, le programmeur peut, en relisant cette trace d'exécution, comprendre où se situe le bogue. En imprimant également l'heure à laquelle chaque message est généré, et ce avec un bon niveau de précision, il obtient la capacité d'isoler des problèmes de performance simples.

Cette technique comporte plusieurs limites. Par exemple, chaque nouveau problème demande généralement une nouvelle instrumentation manuelle, qui ne peut être réutilisée facilement, car elle doit souvent être retirée pour limiter la sortie et le ralentissement du programme. Chaque module instrumenté doit être recompilé. Ensuite, si la cause du problème de performance n'est pas liée à des lignes de

code précises du programme, mais à des activités asynchrones, il sera difficile de l'isoler avec cette technique. De plus, l'analyse doit être faite manuellement par des humains, ce qui contraint sévèrement la taille de trace pouvant être analysée et les liens faits entre les messages. Par ailleurs, le débogage de problèmes intermittents est difficile.

Une forme de traçage plus évoluée, le traçage d'appels systèmes et de signaux, est apparue assez tôt dans Unix. Cette technique consiste à démarrer un programme à l'aide d'une commande spéciale (strace, truss, ktrace, selon la variante du système d'exploitation). L'utilisateur peut alors voir la liste des appels systèmes faits par le programme, leurs arguments et leur valeur de retour. Chaque appel est accompagné de sa durée et de l'heure à laquelle il a eu lieu. Le mécanisme utilisé varie d'une implémentation à l'autre ; une méthode courante, par exemple avec le système Linux, consiste à utiliser l'appel système ptrace()[12] pour surveiller un processus, tandis qu'une autre utilise le système de fichiers /proc. Dans les deux cas, c'est un second processus qui fait la surveillance. Cette méthode est peu efficace puisqu'elle génère un grand volume de changements de contexte et de passages de l'espace noyau à l'espace utilisateur ; en effet, à chaque entrée et sortie d'un appel système, le processus tracé doit bloquer pendant que le processus de surveillance reçoit son état et inspecte sa mémoire.

Ces limites ont incité les programmeurs à développer des outils conçus spécialement pour le débogage de problèmes de performance, dont nous discutons dans la suite de cette section.

1.1.2 Métriques de performance

Les systèmes d'exploitation, même les moins modernes, compilent un certain nombre de métriques de performance. Celles-ci concernent l'activité globale d'un sous-système (réseau, mémoire, disques, etc), d'une ressource matérielle ou logique précise, ou d'un processus donné. Elles prennent généralement la forme de variables numériques qui peuvent être consultées directement ou par le biais d'outils.

Par exemple, ps est un utilitaire Unix ancien qui affiche la quantité de mémoire et le temps CPU consommés par chaque processus. Top, basé sur le code de ps, propose un affichage qui se rafraîchit périodiquement et permet d'afficher les plus grands consommateurs de ces ressources. Vmstat affiche, pour l'ensemble du système, la quantité de mémoire consommée et l'activité disque. Ces métriques sont typiquement conservées dans des structures de données du noyau et transmises aux programmes utilisateur qui en ont besoin via des appels systèmes ou des systèmes de fichiers spéciaux comme /proc ou /sys (voir figure 1.1). Le contenu des fichiers de ceux-ci reflète non pas le contenu de fichiers physiques sur un support de données, mais le contenu, souvent en texte, de structures de données noyau.

Ces métriques ont l'avantage que leur collecte est peu coûteuse en temps machine mais surtout en mémoire. À défaut de pointer vers des lignes de code précises, elles permettent dans certains cas de trouver le processus trop gourmand ou la ressource qui ne fournit pas. Par exemple, si le système exhibe de mauvaises performances parce qu'un processus utilise le processeur presque exclusivement pour une période de plusieurs secondes, top l'affiche en haut de la liste avec une utilisation du CPU près de 100%. Si la cause du ralentissement est que la mémoire est pleine, vmstat affiche de l'activité de mémoire virtuelle sur disque.

Ces métriques sont des totaux et leurs valeurs sont mises à jour en temps réel.

```
           CPU0       CPU1
  0:          16          0   IO-APIC-edge      timer
  1:           1          1   IO-APIC-edge      i8042
  4:           1          1   IO-APIC-edge
  8:           0          0   IO-APIC-edge      rtc0
  9:           0          0   IO-APIC-fasteoi   acpi
 12:           1          3   IO-APIC-edge      i8042
 17:     6388928    6391656   IO-APIC-fasteoi   uhci_hcd:usb3, ehci_hcd:usb4, pata_marvell
 18:           0          0   IO-APIC-fasteoi   uhci_hcd:usb1, uhci_hcd:usb7
 19:           1          2   IO-APIC-fasteoi   uhci_hcd:usb6, ohci1394
 21:      274680     274306   IO-APIC-fasteoi   uhci_hcd:usb2
 22:     2191614    2191955   IO-APIC-fasteoi   HDA Intel
 23:        7007       7036   IO-APIC-fasteoi   uhci_hcd:usb5, ehci_hcd:usb8
1272:      241668     238891   PCI-MSI-edge      eth0
1273:      149031     149081   PCI-MSI-edge      ahci
NMI:           0          0   Non-maskable interrupts
LOC:    17703068   17783937   Local timer interrupts
RES:      180260     187937   Rescheduling interrupts
CAL:      303885     317294   function call interrupts
TLB:       29999      29036   TLB shootdowns
TRM:           0          0   Thermal event interrupts
THR:           0          0   Threshold APIC interrupts
SPU:           0          0   Spurious interrupts
ERR:           0
```

Figure 1.1 Exemple de métriques dans un fichier de /proc. Ce fichier, /proc/interrupts, montre, pour chaque numéro d'interruption, le nombre de déclenchements sur chaque CPU. Les codes d'interruption non numériques (NMI, LOC...) sont des interruptions locales (générées par le contrôleur d'interruption du CPU lui-même) par opposition aux autres, qui sont des interruptions globales au système.

Elles ne peuvent fournir de l'information que par rapport aux problèmes qui sont assez importants pour produire un effet visible sur le total, ou qui persistent assez longtemps pour être visibles sur la valeur en temps réel.

1.1.3 Profileurs traditionnels

Lorsqu'un processus consomme trop de temps CPU, comme dans l'exemple de la section 1.1.2, le programmeur est intéressé à savoir dans quelles fonctions, voire dans quelles lignes de code, le programme a passé le plus de temps. Il est également utile de savoir quels sont les lignes de code à partir desquelles une fonction ayant consommé beaucoup de temps processeur a été le plus appelée. Les profileurs traditionnels, tels que gprof [7] aident à obtenir cette information.

La méthode utilisée par gprof comporte deux volets. Premièrement, une option du compilateur permet d'ajouter, dans le préambule de chaque appel de fonction du programme, un appel à une fonction spéciale de monitorage. Cette fonction sauvegarde dans une table le nombre d'appels à chaque fonction, en fonction de l'adresse appelante. Le fait de sauvegarder le lieu d'appel permet d'afficher, pour chaque fonction, le nombre d'appels par site d'appel.

L'autre volet du profilage consiste à programmer une interruption périodique permettant d'échantillonner à intervalle régulier la fonction dans laquelle se trouve le pointeur d'instruction. En pondérant le nombre d'échantillons pour chaque fonction par rapport au temps total d'exécution, gprof estime le temps, en secondes, passé dans chacune. En distribuant ce temps sur les lieux d'appel proportionnellement, il est également possible d'estimer le temps passé dans la fonction en fonction de la fonction appelante.

Une fois l'exécution du programme terminée, mais juste avant que sa mémoire ne

soit libérée, les structures de données décrites ci-haut sont écrites dans un fichier. Ensuite, l'utilitaire gprof peut être utilisé pour convertir ce fichier de données brutes en un rapport de profilage. Les figures 1.2 et 1.3 montrent des sections d'un tel rapport.

1.2 Techniques récentes de débogage

La section précédente présentait les techniques de débogage traditionnelles et comment elles peuvent s'appliquer aux problèmes de performance. Ces techniques ont été développées dans un contexte où la cause des problèmes de performance était généralement limitée à un seul processus.

Avec la croissance de la taille des systèmes et de leur niveau de parallélisme, de nouveaux outils spécialisés en débogage de problèmes de performance ont été développés. Contrairement à leurs prédécesseurs, ils sont axés sur l'étude de l'ensemble des logiciels d'un ordinateur, voire d'une grappe complète. En effet, ils sont conçus de manière à résoudre des problèmes de performance impliquant plusieurs composants du système. Également, ils permettent d'accélérer le débogage de problèmes difficiles à reproduire ou se produisant rarement.

Ces avantages viennent au prix que ces techniques requièrent une connaissance très profonde de l'ensemble du système. Connaître en détail le fonctionnement de l'application dans laquelle se manifeste le problème n'est pas suffisant car la cause peut souvent se situer ailleurs, et c'est au programmeur qui utilise ces techniques qu'incombe la tâche de lier symptôme et cause.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 17.39      0.04     0.04   697263     0.06     0.13  lttv_hooks_call_merge
 13.04      0.07     0.03   888442     0.03     0.03  lttv_attribute_find_subdir
  8.70      0.09     0.02  4151288     0.00     0.00  ltt_time_compare
  8.70      0.11     0.02   700900     0.03     0.03  ltt_interpolate_time_from_tsc
  8.70      0.13     0.02   698102     0.03     0.04  ltt_update_event_size
  8.70      0.15     0.02   697455     0.03     0.13  ltt_tracefile_read
  8.70      0.17     0.02                             compare_tracefile
  4.35      0.18     0.01   713208     0.01     0.01  marker_get_info_from_id
  4.35      0.19     0.01   703703     0.01     0.01  ltt_get_uint64
  4.35      0.20     0.01   697889     0.01     0.01  ltt_seek_next_event
  4.35      0.21     0.01   118180     0.08     0.08  push_state
  4.35      0.22     0.01    50340     0.20     0.20  lttv_trace_get_hook_field
  4.35      0.23     0.01                             lttv_process_traceset_middle
  0.00      0.23     0.00  1397256     0.00     0.00  ltt_align
  0.00      0.23     0.00  1396334     0.00     0.00  ltt_get_uint16
  0.00      0.23     0.00  1110240     0.00     0.00  ltt_tracefile_get_event
  0.00      0.23     0.00   996553     0.00     0.00  lttv_attribute_get_type
  0.00      0.23     0.00   968751     0.00     0.00  lttv_iattribute_get_type
```

Figure 1.2 Rapport de profilage généré par gprof - *Flat Profile*

```
index % time    self  children    called     name
[1]     85.1    0.01    0.19                 lttv_process_traceset_middle [1]
                0.02    0.07  697263/697455      ltt_tracefile_read [2]
                0.04    0.05  697263/697263      lttv_hooks_call_merge [3]
                0.01    0.00 1394507/4151288     ltt_time_compare [10]
                0.00    0.00       5/5           lttv_traceset_context_ctx_pos_compare [63]
                0.00    0.00  697263/1110240     ltt_tracefile_get_event [76]
                0.00    0.00  697263/743163      lttv_hooks_by_id_get [79]
                0.00    0.00  697235/697842      ltt_event_time [82]
-----------------------------------------------
                0.00    0.00     192/697455      ltt_tracefile_seek_time [48]
                0.02    0.07  697263/697455      lttv_process_traceset_middle [1]
[2]     39.0    0.02    0.07  697455          ltt_tracefile_read [2]
                0.00    0.06  697347/698102      ltt_tracefile_read_update_event [4]
                0.00    0.01  697455/697717      ltt_tracefile_read_seek [14]
-----------------------------------------------
                0.04    0.05  697263/697263      lttv_process_traceset_middle [1]
[3]     38.8    0.04    0.05  697263          lttv_hooks_call_merge [3]
                0.00    0.01    8390/8390        after_schedchange [20]
                0.00    0.01   17441/17441       after_syscall_exit [21]
                0.00    0.01   17440/17440       after_syscall_entry [22]
                0.00    0.01    8390/8390        before_schedchange [23]
                0.00    0.00   54157/54157       syscall_entry [24]
                0.00    0.00   11237/11237       after_soft_irq_entry [25]
                0.00    0.00   11237/11237       after_soft_irq_exit [26]
                0.00    0.00   35446/35446       soft_irq_entry [27]
                0.00    0.00    7058/7058        after_irq_exit [29]
                0.00    0.00    7058/7058        after_irq_entry [28]
                0.00    0.00   22134/22134       irq_entry [30]
                0.00    0.00     205/205         after_enum_process_state [32]
                0.00    0.00    2126/2126        after_trap_exit [33]
                0.00    0.00    2125/2125        after_trap_entry [34]
```

Figure 1.3 Rapport de profilage généré par gprof - *Call Graph*

1.2.1 Traceurs généraux

Une des façons de déboguer un problème de performance est d'utiliser un traceur. Le traçage est une technique qui consiste à enregistrer des événements durant l'exécution d'un programme. Le programmeur examine ensuite ces événements afin d'en déduire de l'information pouvant mener à la cause du problème. Chacun de ces événements comporte un libellé qui donne une information sur l'état actuel du système. Il est accompagné de l'heure à laquelle il est survenu ainsi que, parfois, d'informations supplémentaires appelées les *arguments*.

Le traçage ressemble à l'enregistrement de journaux ou *logs*. Cependant, le terme « traçage » réfère davantage à des approches optimisées pour soutenir en continu de très hauts débits d'événements, souvent plusieurs milliers par seconde ou plus. La haute performance des traceurs permet d'enregistrer des événements de très bas niveau (ex : l'entrée dans un gestionnaire d'interruption), ce qui constitue un avantage par rapport aux journaux. Le traçage génère cependant une quantité telle de données qu'il est habituellement moins facile de consulter une trace qu'un journal.

Le mécanisme du traçage peut être divisé en trois modules.
– Les fournisseurs de données (*data providers*)
– La collecte des données (*data collection*)
– L'analyse des données (*data analysis*)
Un événement est généré lorsque le flot d'exécution atteint un point d'instrumentation dans le code. Les fournisseurs de données sont les mécanismes qui opèrent aux sites d'instrumentation et qui préparent l'événement pour l'envoi au collecteur de données. Il existe deux grandes catégories de fournisseurs de données, soit les fournisseurs utilisant une instrumentation statique, et ceux qui utilisent une instrumentation dynamique. La première est très efficace, mais requiert une re-

compilation pour être ajoutée ; en revanche la seconde est moins efficace mais peut être ajoutée pendant l'exécution. Typiquement, un fournisseur de données permet de désactiver un point de trace lorsqu'il n'est pas utilisé et d'ainsi diminuer son impact sur la performance.

Une forme très simple de fournisseur de donnée utilisant de l'instrumentation statique est le simple appel de fonction ; il s'agit de la méthode utilisée dans le Java Logging API[13]. L'appel est inséré par le programmeur durant le développement du programme ou au moment du débogage. Dans le Java Logging API, chaque appel à la fonction de traçage est accompagné d'un niveau d'importance (de FINEST à SEVERE). Selon les besoins du moment, un niveau d'importance minimal donné est sélectionné et seuls les événements plus importants que ce seuil sont envoyés au collecteur. Ce filtrage a lieu dans la fonction de traçage.

Une forme plus avancée de fournisseur de données, utilisé dans le cadre de l'instrumentation statique, est le projet des Linux Kernel Markers, ou « marqueurs »[14]. Il s'agit d'une technique qui minimise le nombre de cycles du processeur nécessaire et l'impact sur la cache. L'approche utilisée consiste à insérer au point d'instrumentation une instruction qui charge une valeur immédiate dans un registre, puis un saut conditionnel basé sur cette valeur immédiate. La destination du saut est un bloc de code à la fin de la fonction actuelle. Dans ce bloc se trouve un appel à une fonction de traçage, puis un retour à l'instruction qui suit le saut conditionnel. L'activation et la désactivation du point de trace se fait par la modification dynamique de la valeur immédiate dans l'instruction de chargement. Pour une valeur immédiate donnée, le saut n'a pas lieu et le point de trace est sauté. Ce mécanisme complexe nécessite plusieurs précautions lors de l'implémentation à cause de la modification dynamique de code, mais garantit un impact minimal lorsque le point de trace est désactivé, en plus de permettre une activation par site. Pour un Pentium 4 à 3 GHz, le mécanisme des marqueurs cause une pénalité d'environ 16 ns[15] par

événement, ce qui est minime.

Une forme d'instrumentation dynamique est celle utilisée par DTrace[5] pour instrumenter l'entrée ou la sortie de fonctions du noyau de Solaris. Davantage de détails sur cette technique sont donnés à la section 1.2.1.3.

La collecte de données intervient lors du passage sur un point de trace pour préparer l'événement et le mettre dans la trace. Divers traitements peuvent alors être appliqués à l'événement. Dans le cas d'un traceur uniquement en espace utilisateur comme le Java Logging API, les informations peuvent être mises directement dans un fichier ou traitées par d'autre code. Dans le cas de DTrace, lorsqu'un point de trace intéressant est rencontré, une machine virtuelle exécute un petit programme qui peut mettre à jour des variables d'état ou imprimer des messages en sortie.

Dans le cas de LTTng[16], l'événement est placé dans un tampon dans la mémoire noyau. Un programme copie le tampon vers un fichier ou l'envoie via un réseau à mesure qu'il se remplit. Le tampon est divisé en au moins deux zones de tailles égales appelées sous-tampons. Les sous-tampons sont réutilisés à mesure qu'ils sont consommés ; le tampon est donc de type circulaire. Le format d'écriture des événements dans le tampon a été conçu pour minimiser la taille mémoire des événements. Le numéro unique de type d'événement est écrit, suivi de l'heure à laquelle il est survenu, puis de ses arguments. Les arguments sont des informations supplémentaires qui accompagnent un événement ; par exemple l'entrée dans un gestionnaire d'IRQ est accompagné du numéro d'IRQ. Le nombre et le type des arguments, qui sont fonction du type d'événement, sont décrits au début de la trace. Des événements peuvent se produire simultanément dans des contextes d'exécution réentrants. Un mécanisme d'exclusion mutuelle est donc requis pour protéger le système de traçage. LTTng cause un minimum d'impact en performance, notamment parce que son collecteur de données est « sans verrouillage », n'utilisant que des primitives

atomiques pour garantir l'exclusion mutuelle. Il est donc complètement réentrant ; en effet, il peut fonctionner avec des points de trace dans des interruptions non-masquables (NMI).

Quant aux analyseurs de données, leur rôle est d'afficher et d'analyser les traces pour mettre en évidence les informations importantes qu'elles contiennent. Ils prennent souvent la forme d'applications interactives. Plusieurs exemples d'analyseurs sont cités dans les sections suivantes.

1.2.1.1 LTTng

LTTng[16], le *Linux Trace Toolkit Next Generation*, est un traceur pour le système d'exploitation Linux. Il s'agit d'un traceur d'usage général qui vise à être approprié pour tous les cas où une instrumentation statique est souhaitable. Les Linux Kernel Markers jouent le rôle de fournisseur de données pour le code noyau, tandis qu'un appel système remplit cette fonction en espace utilisateur. Un module de sérialisation d'événements et le système de fichiers Relay[17] en sont le collecteur de données. Finalement, l'outil LTTV[18] en est l'analyseur de données. LTTng est fourni avec une instrumentation de base pour le noyau de Linux. Il est en cours d'intégration dans la version officielle du noyau de Linux.

L'instrumentation fournie avec LTTng permet de générer les événements du tableau 1.1. LTTng permet aux développeurs d'ajouter un nouveau point d'instrumentation, incluant des arguments, en une seule ligne de code.

L'outil d'analyse LTTV accompagne LTTng. Il comporte une interface graphique, une interface en ligne de commande et plusieurs vues. LTTV a la capacité d'ouvrir plusieurs traces utilisant une base de temps commune pour les fusionner. Il s'agit d'une fonction essentielle pour l'étude de traces enregistrées simultanément sur

Tableau 1.1 Points d'instrumentation accompagnant la version 0.26 de LTTng.

Catégorie	Événements
Événements internes	Définition d'un marqueur, format des arguments d'un marqueur
Appels système	Entrée et sortie
IRQ	Entrée et sortie
softIRQ	Mise en queue, entrée et sortie
Trap	Entrée et sortie
Appels système fichier	Événements pour les appels open(), read(), write(), readv(), writev(), pread64(), pwrite64(), llseek(), lseek(), ioctl(), exec(), poll(), select() donnant leurs arguments
Gestion de la mémoire	mm_filemap_wait_end, mm_filemap_wait_start, mm_page_free, mm_page_alloc, mm_handle_fault_exit, mm_handle_fault_entry, mm_swap_in
Réseau	net_socket_call net_socket_create net_socket_recvmsg net_socket_sendmsg net_dev_receive net_dev_xmit net_insert_ifa_ipv4 net_del_ifa_ipv4
Système *input*	Événement d'entrée par l'humain (clavier, souris, joystick, etc)
List	Utilisé au début de la trace : listage des modules noyau chargés, de l'état de processus, des descripteurs de fichiers, des noms des interruptions, des mappages de mémoire virtuelle, des interfaces réseau,

des machines qui sont membres d'une même grappe de calcul, du même système distribué, ou s'exécutant sur une même machine physique grâce à un système de virtualisation. Également, plusieurs vues peuvent être ouvertes simultanément sur une trace.

LTTV est optimisé pour lire et afficher de très grandes traces, incluant celles dont la taille dépasse la quantité de mémoire vive de l'ordinateur sur lequel l'analyse a lieu. LTTV ne charge donc pas la trace complète en mémoire. La lecture fonctionne plutôt par un mécanisme de requêtes. Les modules qui affichent des résultats demandent à l'engin d'obtenir les événements dans un certain intervalle de temps. L'engin combine les requêtes des modules, fait un mmap() du début de la région demandée, et indique aux modules où ils peuvent trouver les événements en mémoire. Lorsque la fin de la zone mappée en mémoire est atteinte, un munmap() est effectué, puis un mmap() de la zone suivante est fait et ainsi de suite jusqu'à la fin de l'intervalle de temps demandé par les requêtes. Les données de la trace n'ont donc jamais à être copiées pendant l'analyse, ce qui permet d'atteindre de bonnes performance malgré le volume important de données.

LTTV est livré avec un certain nombre de vues. La figure 1.4 en montre quelques-unes. La vue *Event View* permet de naviguer à travers la liste d'événements bruts et de la filtrer. Une autre, le *Control Flow View*, affiche, pour chaque processus, une ligne du temps dont la couleur indique un état du processus parmi ceux du tableau 1.2. Une autre vue similaire, le *Resource View*, affiche une ligne du temps colorée pour des ressources comme le processeur (occupé par un processus, IRQ ou inutilisé), les gestionnaires d'interruption (actif, inactif), les gestionnaires de trappes (actif, inactif) et les gestionnaires de softIRQ (inactif, en attente, actif). La vue des statistiques présente toute une gamme de métriques, comme le nombre d'événements de chaque type produits dans chaque contexte. LTTV possède également d'autres vues qui ne seront pas décrites ici. Chaque vue dont un des axes

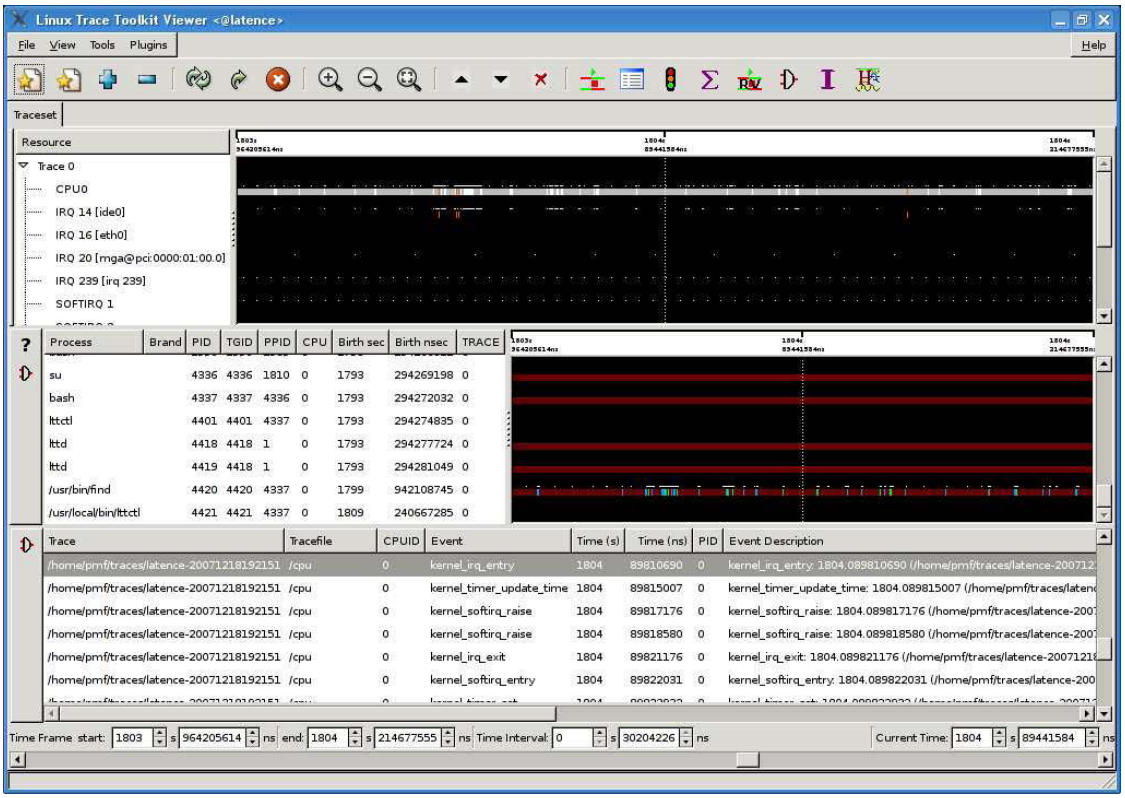

Figure 1.4 Quelques vues de LTTV
De haut en bas : Resource View, Control Flow View, Event View

est le temps possède un curseur dont l'emplacement est synchronisé avec celui des autres vues.

Pour afficher graphiquement l'état de divers composants du système, LTTV comporte un module de conservation d'état. Lorsqu'une nouvelle trace est ouverte, le module fait une requête à l'engin pour en obtenir tous les événements. Il considère les événements *statedump* ainsi que tous les événements qui annoncent des changements d'état du système par la suite. Le *statedump* est un mécanisme par lequel l'état du système est décrit en absolu au début de la trace (liste des appels système disponibles, des zones de mémoire mappées, etc). Les événements de changement d'état subséquents sont des événements ordinaires qui se produisent dans le système, par exemple des changements de contexte ou l'entrée dans un gestion-

Tableau 1.2 États de processus dans LTTV (type de code exécuté).

Modes Control Flow View
Unknown
Running in user mode
Running in system call
Running in trap
In IRQ
In softIRQ
Waiting for I/O
Waiting for CPU
Waiting for fork
Process exited, waiting for cleanup

naire d'interruption. Le module de conservation d'état utilise ces informations pour sauvegarder l'état absolu du système à intervalle régulier.

Les modules d'analyse et de visualisation peuvent quant à eux interroger le module de conservation d'état. Lorsque l'utilisateur de LTTV modifie la fenêtre de temps affichée par les vues, celles-ci doivent rafraîchir leur affichage. La sauvegarde de l'état du système la plus proche, mais précédant le début de la nouvelle fenêtre de temps à afficher, est chargée. Ensuite, l'intervalle de trace entre cette sauvegarde et le début de la fenêtre de temps est lue, afin de retrouver l'état au début de cette fenêtre. Ceci évite d'avoir à relire la trace à partir du début pour connaître l'état du système en un point donné, diminuant grandement le temps de réaction de LTTV lors de la navigation dans une trace. Les valeurs du tableau 1.2 sont un des types d'état qui sont conservés par le module de conservation d'état.

L'enregistrement d'une trace avec LTTng peut se faire de deux manières différentes. Premièrement, une trace peut être enregistrée sur le disque. Ce mode est utile lorsque la trace à enregistrer est grande. Deuxièmement, une trace peut être enregistrée en mémoire en mode *flight recorder*. Ce mode utilise des tampons circulaires, ainsi seuls les événements les plus récents sont conservés. Il est plus efficace

lorsque l'impact sur le système tracé doit être minimal. Il est aussi utile lorsque le traçage doit fonctionner durant une longue période dans l'attente qu'un certain bogue se produise — parfois plusieurs jours. Dans ces cas, un éventuel script conçu pour détecter le bogue peut stopper la trace. La fin de celle-ci contient les événements survenus autour du moment où le bogue s'est manifesté.

LTTng et LTTV ont été utilisés avec succès pour trouver la cause de bogues liés à la performance autant dans des systèmes temps-réel que dans la grappe de la compagnie Google ou d'autres systèmes d'envergure [3, 4].

LTTV peut être utilisé pour obtenir une vue générale de l'activité dans le système tracé grâce notamment au *Control Flow View*. Une fois une région problématique identifiée, la vue peut être agrandie jusqu'à l'échelle de la nanoseconde. De plus, une consultation des événements bruts permet l'accès au niveau de détails le plus précis. Ceci permet le débogage d'un large éventail de problèmes.

LTTV permet le débogage de problèmes de lenteur simples dus à des blocages, grâce au Control Flow View. La ligne du temps colorée permet de détecter visuellement le fait que beaucoup de temps a été passé en blocage. Dans certains cas, cette information est suffisante pour permettre au développeur de réaliser qu'un comportement n'est pas celui qui était anticipé et d'en voir la cause.

Cependant, généralement, l'interprétation est plus difficile. La lenteur peut être due à l'accumulation de plusieurs courts épisodes de blocage, ou il peut être difficile de dire quels blocages sont normaux et lesquels sont anormaux, ou encore une chaîne de blocages peut séparer le processus où se produit le symptôme de celui où se situe la véritable cause du problème. Ces cas demandent une exploration manuelle fastidieuse de la trace. En effet, la cause de chaque blocage doit être vérifiée manuellement avec la liste des événements. De plus, dans les cas où les

blocages sont chaînés, la chaîne doit être remontée manuellement. Dans les systèmes moindrement complexes, ce genre d'opération est si long et complexe qu'il est difficilement envisageable. Ainsi, LTTV est relativement limité pour le débogage de problèmes de performance liés à des blocages.

1.2.1.2 Wind River Workbench

Le Wind River Workbench[19] est un environnement de développement qui inclut un traceur. Il utilise LTTng comme fournisseur et collecteur de données. Son outil d'analyse fournit des vues équivalentes à celles de LTTV.

Pour l'analyse de problèmes de lenteur impliquant des blocages, le Workbench souffre des mêmes limites que LTTV.

1.2.1.3 DTrace

DTrace[5] est un traceur dynamique développé par Sun Microsystems et intégré aux versions récentes de Solaris et de Mac OS. Il est à la fois un fournisseur et collecteur de données. L'utilisateur de DTrace peut coder, à l'aide du langage D de DTrace, des analyses simples qui ont lieu de façon synchrone pendant l'enregistrement de la trace. De plus, des interfaces graphiques comme Shark[20] et Instruments[21] permettent de le contrôler.

L'instrumentation de DTrace est fournie par des *providers* ou fournisseurs, des modules offrant une interface standard d'accès à de l'instrumentation de nature variée. Le tableau 1.3 montre des types d'instrumentation disponibles. DTrace permet d'activer des points d'instrumentation dans le noyau et dans les processus durant l'exécution. Dans certains cas, il s'agit d'une instrumentation statique qui

Tableau 1.3 Points d'instrumentation de DTrace.

Points d'instrumentation de DTrace
Entrée et sortie de chaque fonction du noyau
Points d'instrumentation statique définis par les programmeurs
Point spécial qui donne la valeur du IP régulièrement pour profilage
Instrumentation des appels système
Instrumentation des primitives de verrouillage

est alors activée. Dans d'autres cas, comme les entrées et sorties de fonction, l'activation d'un point d'instrumentation correspond dans les faits à l'ajouter. En effet, la phase de compilation n'instrumente pas de manière particulière chaque fonction. Ce n'est qu'au moment de l'activation d'un point d'instrumentation dynamique que DTrace modifie le code au site d'instrumentation pour y ajouter un point de trace.

Lorsqu'un événement se déclenche, DTrace exécute les actions spécifiées par l'utilisateur dans un script en langage D. Celui-ci peut filtrer l'événement en fonction de la valeur de variables accessibles au site de l'instrumentation, mettre à jour des variables d'état ou imprimer des informations. Le script n'a cependant accès qu'en lecture à la mémoire du noyau afin d'en garantir la stabilité.

DTrace a été utilisé avec succès pour déboguer des bogues de performance complexes[5]. La méthodologie consiste à repérer une manifestation du problème de performance sur une variable, comme une métrique système par exemple. Ensuite, il faut instrumenter la modification de cette variable pour trouver le contexte où les modifications problématiques ont lieu, puis instrumenter ce contexte et ainsi de suite pour remonter la chaîne de causalité jusqu'au processus ou à la ressource qui est la cause du problème. Il s'agit d'une méthodologie itérative où à chaque fois, l'utilisateur doit décider de la nouvelle instrumentation à utiliser, ce qui demande une bonne connaissance du fonctionnement interne de toutes les couches du système.

Cette méthodologie itérative prend plusieurs minutes, voire plusieurs heures à appliquer. Pour que le problème de performance puisse être trouvé, il faut donc idéalement qu'il ait lieu en continu ou qu'il se produise à haute fréquence. Pour les problèmes qui se produisent rarement, DTrace est désavantagé par rapport aux traceurs purement statiques qui sont optimisés pour enregistrer de grandes quantités d'événements.

Sur plateforme x86, l'instruction INT3 que DTrace utilise pour instrumenter dynamiquement l'entrée et la sortie de fonctions est très coûteuse. Sur un Pentium 4 à 3 GHz, on compte une pénalité de plus de 1 μs par événement[22], soit 100 fois plus que pour les Markers (voir section 1.2.1). Cette forme d'instrumentation a donc des limites quant à la gamme de problèmes pour lesquels elle peut être utilisée.

Le projet SystemTap[23] est similaire à DTrace, mais est conçu pour usage avec le noyau de Linux. Son développement n'est pas encore terminé. SystemTap vise à dépasser certaines limites de DTrace dont l'une des plus importantes est la possibilité d'instrumenter chaque ligne de code plutôt que chaque entrée ou sortie de fonction.

1.2.1.4 QNX Momentics

Momentics[24] est une suite de développement qui permet de tracer le noyau du système d'exploitation QNX. La suite joue le rôle de fournisseur de données, de collecteur de données et d'analyseur. Les événements sont placés dans un tampon circulaire puis écrits progressivement au disque.

Un outil d'analyse de trace accompagne Momentics. Il permet de consulter les événements bruts ainsi qu'un histogramme du débit d'événements en fonction du temps, superposé à un diagramme de l'utilisation des CPU en fonction du temps.

Un graphique en pointe de tarte montre le temps où le système était inactif, dans une interruption, en travail dans le noyau, et en travail dans des applications utilisateur. Le temps d'exécution de chaque processus s'étant exécuté sur le système est divisé en catégories. Ces temps s'appliquent à la trace complète. Ils sont affichés dans un tableau.

– *Running time* : temps où le processus s'exécutait
– *Ready time* : temps où le processus était prêt à s'exécuter mais ne s'exécutait pas
– *Blocked time* : temps où le processus était bloqué dans un appel système

Pour les cas où le programmeur peut réussir à isoler dans une trace le bogue de performance, ces chiffres permettent de confirmer si celui-ci est dû à du blocage ou à une autre cause. Mais aucune information n'est donnée sur les causes du blocage.

1.2.2 Traceurs ad-hoc

Les traceurs ad-hoc sont des traceurs conçus par des programmeurs pour investiguer des problèmes de performance et d'autres types de problèmes dans un contexte très précis. Ils n'utilisent généralement pas d'infrastructure commune avec d'autres traceurs.

Comparativement aux traceurs généraux, ils ont l'inconvénient que les événements qu'ils recueillent ne peuvent être corrélés avec ceux qui sont recueillis par d'autres traceurs.

1.2.2.1 blktrace

Le traceur blktrace[25] a été conçu pour tracer les événements liés aux *block devices*[1] dans Linux. Les événements suivants peuvent être enregistrés : allocation d'une entrée dans la queue de requêtes, insertion d'une requête dans la queue, fusion de requêtes, réinsertion d'une requête dans la queue, remappage d'une requête, complétion d'une requête, etc.

Le projet utilise les Linux Kernel Markers aux sites d'instrumentation. Des fonctions propres à blktrace sérialisent les événements pour les envoyer dans un canal Relay. Un programme d'analyse conçu expressément pour blktrace consomme les données et affiche une liste simple d'événements ou une compilation de métriques.

Le traçage de chaque *block device* est activé séparément, ce qui permet de se concentrer sur le problème à l'étude sans nuire aux performances davantage que nécessaire.

Blktrace présente de l'information intéressante pour déboguer des problèmes de performance liés aux *block device*. Le fait que ce soit un traceur distinct rend cependant difficile la corrélation entre les événements blktrace et les autres qui se produisent sur le système. En outre, cette association est encore plus difficile si une chaîne de dépendances à plusieurs étapes relie le processus qui exhibe le symptôme de lenteur au disque.

1.2.2.2 ftrace

Ftrace[26] est un mécanisme de traçage pour le noyau de Linux. Il inclut un mécanisme de fournisseur de données, de collecteur de données et des analyses simples.

[1] Dans Unix, un *block device* est une abstraction qui correspond à un matériel ou entité logique, qui contient des données accessibles par adresse.

Le formatage des données sous forme de texte et l'analyse ont lieu dans le noyau. La trace en format texte peut être lue directement d'un fichier du système de fichiers DebugFS[27]. Ftrace est constitué de divers « traceurs » conçus chacun pour un type de problème de performance donné. Voici ces traceurs, une liste des événements qu'ils tracent et les principales informations qui les accompagnent.

- Changements de contexte
  - Pour chaque réveil et chaque changement de contexte : le temps, les processus impliqués, le mode du processus qui se fait réveiller, le numéro du processus.
- Zones de désactivation des interruptions
  - Pour chaque zone de désactivation des interruptions : processus en cours d'exécution, durée de désactivation, fonction où la désactivation et la réactivation ont eu lieu.
- Zones de désactivation de la préemption
  - Idem, pour chaque zone de désactivation de la préemption.
- Zones de désactivation des interruption ou de la préemption
  - Idem, pour chaque zone de désactivation des interruptions ou de la préemption.
- Latence entre le réveil et l'ordonnancement de la tâche temps-réel la plus prioritaire
  - Pour chaque réveil et chaque ordonnancement de la tâche temps-réel la plus prioritaire du système : tâche, durée, fonction du réveil, fonction du changement de contexte.
- Exécution de fonctions
  - Pour chaque fonction exécutée dans le noyau : processus, numéro de processeur, temps, fonction, fonction appelante (il est aussi possible de filtrer les fonctions déclenchant un événement).

Ftrace n'est pas conçu pour être facilement étendu pour tracer d'autres événements.

Ftrace utilise l'approche de fournir un système de traçage précis pour une tâche précise. Chacun de ses traceurs est donc simple et efficace pour le débogage du genre de problème précis pour lequel il est conçu. Son utilité est cependant limitée dans un contexte général où le bogue fait intervenir plusieurs modules du noyau, voire une ou plusieurs applications.

1.2.3 Outils de profilage système

Les outils de profilage système profilent un système au complet. En ce sens, ils sont une généralisation des outils profilage traditionnels abordés à la section 1.1.3.

1.2.3.1 Sysprof

Sysprof[8] consiste en un module pour le noyau de Linux qui échantillonne à intervalle régulier la pile du processus qui roule sur chaque processeur et l'envoie à un programme utilisateur via le système de fichiers proc. Ce programme inspecte l'exécutable et ses bibliothèques pour en déduire le nom des fonctions qui sont sur la pile et en faire un rapport. Un profil de type *flat profile* est affiché. Lorsqu'une fonction est sélectionnée, ses descendants sont affichés sous forme d'arbre dans une autre partie de la fenêtre. Tous les échantillons où le contrôle est dans le noyau sont regroupés en une seule ligne « noyau ». La figure 1.5 montre un exemple de rapport Sysprof.

Sysprof est utile pour déboguer les problèmes de lenteur lorsque la cause se situe à l'intérieur d'un processus ou qu'un processus est ralenti par d'autres processus qui accaparent le processeur. Il ne montre cependant pas les liens de dépendance qui existent à cause de blocages. Lorsqu'un processus est bloqué, donc dans le noyau, Sysprof ne fait que comptabiliser l'échantillon dans la ligne « noyau ».

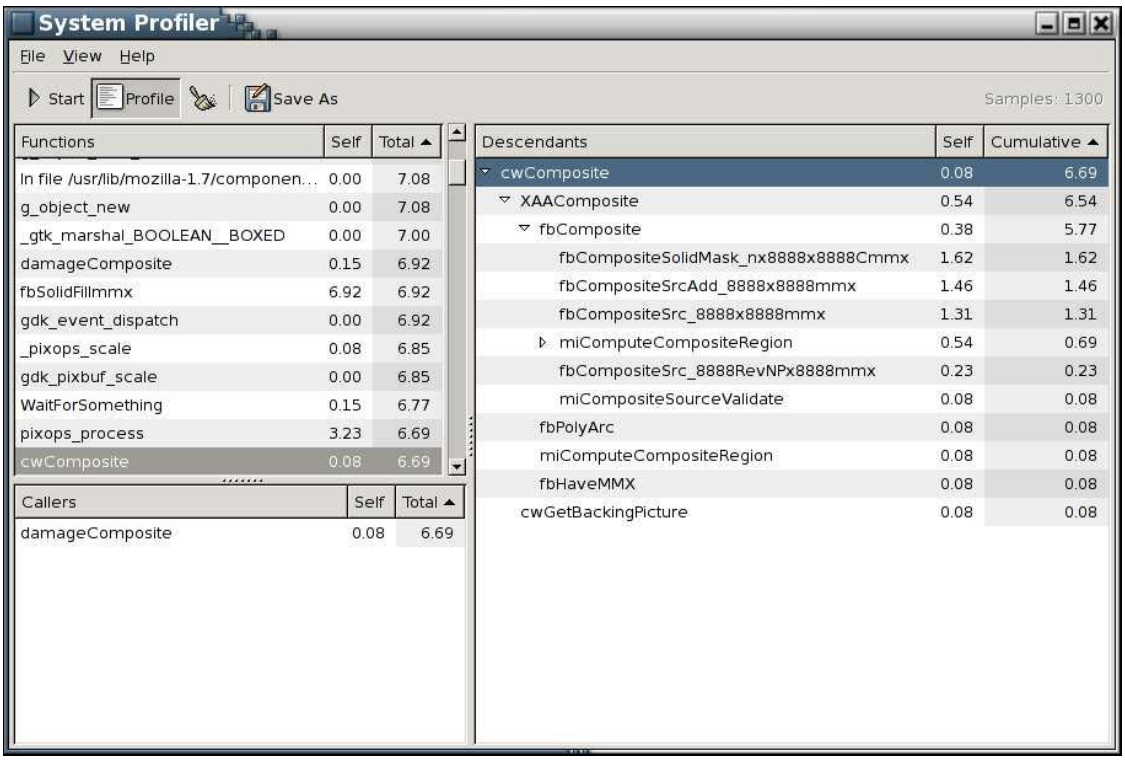

Figure 1.5 Interface utilisateur et rapport de Sysprof

1.2.3.2 Intel VTune

VTune[10] est un outil développé par Intel qui a sensiblement les mêmes fonctionnalités que Sysprof. De plus, il peut instrumenter un exécutable pour en extraire un profil aussi détaillé que ceux générés par gprof par exemple. Il souffre des mêmes limites en ce qui a trait à l'analyse de la causalité des blocages.

1.2.3.3 Oprofile

Oprofile[9] est un outil qui se sert de l'échantillonnage pour lire les compteurs de performance (pour architecture Intel, voir [28]) du microprocesseur et corréler les changements avec les adresses de code où elles ont eu lieu. Il est donc possible d'obtenir, pour chaque processus, fonction ou ligne de code, un profilage du temps qui y a été passé, mais également du nombre de fautes de cache de données, de fautes de cache d'instructions, de fautes de TLB, etc. qui y sont survenues.

Oprofile est un outil très utilisé pour l'optimisation de très bas niveau (dépendantes de l'architecture). Il recueille des informations sur le comportement du logiciel pendant son exécution. Les blocages sont donc hors de sa portée.

1.3 Outils d'analyse des causes de blocage

Dans la section précédente, nous avons passé en revue les techniques de débogage importantes ayant vu le jour relativement récemment, mais n'ayant pas été conçues spécifiquement pour déboguer des problèmes de performance liés au blocage. Certains des outils présentés permettent, dans une certaine mesure toutefois, d'examiner ces problèmes, mais demandent de le faire de manière relativement

Cause	Maximum	Average
process fork	1097.7 msec	2.5 msec
Reading from file	1097.0 msec	0.1 msec
updating atime	850.4 msec	60.1 msec
Locking buffer head	433.1 msec	94.3 msec
Writing to file	381.8 msec	0.6 msec
Synchronous bufferhead read	318.5 msec	16.3 msec
Waiting for buffer IO	298.8 msec	7.8 msec

Figure 1.6 Exemple de sortie de LatencyTOP, extraite du site du projet. Cette sortie a été générée pendant une compilation du noyau de Linux avec gcc.

manuelle.

Dans cette section, nous examinons les outils qui ont été conçus spécifiquement pour examiner les causes de blocage, ou qui rendent leur investigation facile, du moins dans certains contextes.

Nous commentons l'aptitude de ces outils à aider au débogage d'un problème de performance sur un système complexe en production.

1.3.1 LatencyTOP

LatencyTOP est un outil conçu par Intel pour pouvoir classer le temps de blocage total par cause, pour chaque processus du système ou un système complet. Le support noyau est inclus dans la distribution du noyau de Linux. La figure 1.6 montre un exemple de sortie.

LatencyTOP a l'avantage de pouvoir spécifier avec beaucoup de détails la cause du blocage. Par exemple, chaque appel système peut être instrumenté pour potentiellement causer de la latence de plusieurs façons différentes. Cependant, cette approche statistique requiert que la latence ait une assez longue durée pour être visible dans le total. De plus elle ne fournit pas d'indication au programmeur quant

à l'emplacement dans le code du processus de l'entrée dans le blocage. Finalement, elle ne permet pas de faire le suivi de chaînes de blocage.

1.3.2 Vampir

Vampir[29] est un outil de traçage et de visualisation visant à aider au débogage de problèmes de performance dans les applications parallèles, notamment celles basées sur la bibliothèque MPI.

L'application tracée doit être liée à la bibliothèque Vampir. Le développeur peut instrumenter le passage de l'application dans certains modes, nommés *activités*, en appelant une fonction prévue à cet effet à l'entrée et à la sortie de ceux-ci. Certaines activités sont fournies automatiquement, comme l'exécution de fonctions MPI. De la même manière, le traçage est démarré et stoppé par un appel de fonction de la bibliothèque Vampir.

Après analyse des fichiers de trace générés par les noeuds durant leur exécution, Vampir peut afficher un sommaire du temps passé dans chaque activité (par noeud ou global). Il peut également afficher un graphe de l'activité en fonction du temps pour chaque processus. L'échange de messages MPI est indiqué par des flèches perpendiculaires à l'axe du temps, entre les processus.

Ces vues permettent au développeur de comprendre plusieurs problèmes de performance liés aux interactions MPI et à la logique de l'application, notamment les chaînes de blocages n'impliquant que des échanges MPI et d'autres activités explicitement instrumentées. En revanche, comprendre les blocages qui se prolongent à cause d'un autre processus sur la même machine ou à cause du matériel (disques par exemple), nécessite l'utilisation d'une autre méthode.

1.3.3 Paradyn

Paradyn[30] est un outil de mesure de performances pour les systèmes parallèles et distribués. Il utilise de l'instrumentation dynamique qui est activée et désactivée pendant l'exécution du programme pour tester des hypothèses quant à l'emplacement des goulots d'étranglement de performance.

L'instrumentation qui est utilisée pour calculer des métriques, soit des compteurs et des minuteries, est implantée au moyen de trampolines[2].

Paradyn tente d'informer le programmeur à propos du problème de lenteur sous trois axes : le « pourquoi », le « où » et le « quand ». Pour chacun de ces axes un ou des arbres de causes sont définis. Chaque noeud, sauf les feuilles, contient un certain nombre de noeuds enfants qui représentent une cause plus précise. Par exemple, un arbre du « où » peut avoir une racine qui représente les objets de synchronisation en général ; ses enfants pourraient être « sémaphore », « message », « spinlock » et « barrière » ; ensuite, les enfants de chacun de ces noeuds pourraient être les instances de ces objets dans le code. Un autre arbre peut également être défini pour les différents processeurs. Enfin, un autre peut contenir les fichiers source et leurs fonctions.

Lorsqu'il est déterminé qu'un problème existe avec un noeud, des tests sont effectués avec ses enfants pour déterminer si la cause peut être raffinée davantage. Il en va de même pour tous les arbres, de sorte qu'après le test d'un certain nombre d'hypothèses, Paradyn converge vers des noeuds feuille qui informent le programmeur à propos du problème.

[2]Une trampoline, au sens de Paradyn, est une façon d'ajouter des instructions dans du code déjà compilé. Le mécanisme consiste à remplacer une instruction dans le code par un saut vers le code à ajouter. L'instruction écrasée est également insérée dans ce code. À la fin de ce code, une instruction ramène le fil d'exécution vers l'instruction qui suit celle qui a été écrasée.

Pour que Paradyn converge vers un noeud particulier d'un arbre, il faut que le problème ait été visible au noeud supérieur. Par exemple, pour qu'il teste un envoi de message particulier pour des délais, il faut que l'instrumentation de la catégorie globale « message » ait exhibé un problème. Or, de nombreux problèmes de performance ne sont pas visibles dans les métriques globales.

Juger convenablement de l'existence d'un problème de performance lié à un noeud donné est difficile dans un système complexe, or Paradyn dépend de l'élaboration d'un modèle permettant de prendre automatiquement cette décision.

Lors d'une situation où il y a une chaîne de blocages, le problème trouvé pourrait ne pas être la véritable cause, mais plutôt un symptôme. Paradyn n'offre pas de moyen d'explorer plus profondément ces cas, ni de remonter la séquence de blocages.

Paradyn nécessite également qu'un problème se reproduise plusieurs fois et suffisamment longtemps pour qu'un nombre suffisant d'hypothèses pour permettre une connaissance précise de la cause du problème puissent être testées.

1.3.4 Pip

Reynolds et al.[31] proposent un langage permettant de spécifier le comportement structurel et temporel d'un programme. Si un programmeur, en se basant sur le design de l'application, décrit les échanges de messages, et leurs contraintes temporelles, qui doivent se produire entre les fils d'exécution d'un même *path instance*, Pip peut leur indiquer si une exécution a respecté ces spécifications. Pip peut également analyser l'exécution d'une application pour écrire lui-même les spécifications à l'aide du langage, en découvrant la chaîne de causalité. Un programmeur peut alors les consulter et détecter certains problèmes dans le déroulement de l'application.

Théoriquement, avec une bonne spécification, Pip devrait pouvoir repérer le composant fautif lors d'un problème de performance. Mais dans les faits, il est nécessaire de faire un compromis entre la difficulté d'élaborer une bonne spécification et la proportion de problèmes pouvant potentiellement être trouvés. Or, les problèmes de performance dont la localisation demande le plus de temps au développeur se produisent sur les systèmes particulièrement grands, complexes et en évolution. Sur ce type de système, il est difficilement pensable d'établir manuellement une spécification du comportement du système suffisamment détaillée et précise pour pouvoir détecter une part importante des problèmes de performance liés au blocage. Quant à l'élaboration automatique, plus la spécification qu'elle produit est détaillée et plus elle risque d'être trop conservatrice et, en même temps, ardue à comprendre pour des humains.

1.3.5 TIPME

TIPME[32] aide à l'identification des causes de la lenteur perçue par l'utilisateur d'une interface graphique. Pour ce faire, il trace les requêtes et les réponses X Window en plus de l'activité d'ordonnancement et l'état des processus (*running*, *runnable*, *blocked*) dans des tampons circulaires en mémoire. Lorsque l'utilisateur perçoit une lenteur, il appuie sur une combinaison de touches qui déclenche une sauvegarde sur le disque l'activité récente du système. Le programmeur peut ensuite étudier ces données pour comprendre dans quel état les processus étaient. L'intervalle à examiner peut être facilement déterminé à partir des événements X Window qui sont enregistrés.

La méthodologie présentée est cependant limitée à des délais observés par les utilisateurs et à des délais d'une granularité perceptible par ceux-ci. De plus, bien que TIPME puisse détecter que le processus est bloqué sur une ressource donnée, il

ne permet pas de comprendre pourquoi la ressource n'était pas disponible, soit de suivre la chaîne de blocages.

1.3.6 Magpie

Magpie[33] est un outil de débogage de problèmes de performance pour les requêtes dans un système distribué. Il est axé sur la détection automatique de problèmes de performance.

Les systèmes sont tracés avec le Event Tracing for Windows[34] et la trace est traitée en ligne ou *a posteriori*. Un algorithme d'extraction de requêtes[35] sépare les événements qui concernent chaque requête, recréant ainsi une trace par requête, ne contenant que les événements y ayant trait. Pour ce faire, les composants qui traitent les requêtes sont instrumentés afin de produire des événements qui indiquent le début et la fin du traitement d'une requête. Il n'est pas nécessaire de propager un identificateur unique de requête à travers tout le système. Magpie fait plutôt la corrélation entre les identificateurs communs à un sous-ensemble de modules afin de retracer le chemin d'une requête. La manière de faire ces corrélations est définie manuellement dans un *schéma d'événements*.

Ces traces, chacune spécifique à une requête, sont regroupées en grappes de requêtes ayant eu un comportement similaire, par l'application d'un algorithme de type « String Edit Distance ». L'usage des ressources système de chaque requête, déduit par examen de la trace, est également utilisé pour le regroupement. Magpie se base sur l'hypothèse que les requêtes qui sont membres de petits regroupements sont celles qui ont connu des problèmes.

Une fois qu'une requête problématique est trouvée, les auteurs proposent de trouver les événements problématiques à l'intérieur par la comparaison de la trace de celle-ci

à une machine à états à laquelle chaque transition est associée à une probabilité[36].

Ainsi, Magpie permet d'imputer à chaque composant (serveur web, base de données, etc) une part du temps de traitement de la requête, dans la mesure où ce composant à été instrumenté convenablement et qu'il a été intégré au *schéma d'événements*. Il s'agit de deux contraintes importantes. Premièrement, il faut que le code source soit disponible et, comme l'indiquent les auteurs, l'instrumentation d'un application nécessite une étude approfondie de celle-ci, ce qui est une tâche ardue. Deuxièmement, l'établissement et la tenue à jour d'un schéma est une tâche également ardue ; elle nécessite de plus de faire un compromis entre la précision du schéma et la granularité des problèmes pouvant être trouvés.

1.3.7 Performance Debugging for Distributed Systems of Black Boxes

Aguilera et al.[37] détectent les goulots d'étranglement de performances dans des systèmes distribués de boîtes noires. Il n'instrumentent pas les noeuds du système, ni ne connaissent les détails de leur fonctionnement interne. Seule une inspection du trafic réseau est nécessaire. En analysant le moment de la transmission de chaque message, ainsi que son destinataire et son expéditeur, ils trouvent des séquences de messages ayant un lien de causalité entre eux. Lorsqu'un échange est trop long, ils peuvent voir quels noeuds induisent le plus de latence. Cette approche a l'avantage d'être très peu intrusive et donc facile à implanter, en plus d'être générique donc de pouvoir s'appliquer à presque n'importe quel système communiquant par messages et dont les messages peuvent être écoutés de manière passive. Elle n'est utile cependant que pour identifier le noeud et le chemin de causalité problématiques. Elle ne dit rien sur ce qui cause une latence à l'intérieur d'un noeud. Pour le savoir, l'usage d'une méthode différente est nécessaire.

1.4 Conclusion de la revue de littérature

Ainsi, il existe toute une gamme d'outils pour assister les développeurs lors du débogage de problèmes de performance. Cependant, il semble qu'aucun n'est tout à fait efficace face aux problèmes de performance impliquant des chaînes de blocages. Certains des outils permettent de faciliter les recherches dans certains cas, mais tous ont des limites qui, en pratique, les rendent très souvent difficiles à utiliser ou inutilisables dans les contextes où ce genre de bogue se produit généralement.

La nouvelle approche décrite dans ce mémoire est une tentative de combler ce manque. Le chapitre suivant décrit cette approche et l'outil dans lequel elle a été implémentée.

CHAPITRE 2

ANALYZING BLOCKINGS TO DEBUG PERFORMANCE PROBLEMS ON MULTI-CORE SYSTEMS

Authors : Pierre-Marc Fournier and Michel R. Dagenais.

Submitted to Operating Systems Review (ACM).

Abstract

Multi-core systems are rapidly becoming more prevalent. Consequently, developers frequently face performance bugs caused by unexpected interactions between parallel software components. The location of these bugs is difficult to identify with current tools. Indeed, a long nested chain of blockings may separate the process exhibiting the slowness from the root cause.

This article introduces a new approach for analyzing blockings on multi-core systems and reports on its implementation in the LTTV Delay Analyzer. It enables developers to quickly understand the dependencies among processes and see how the total elapsed time is divided into its main components. The LTTV Delay Analyzer was used to analyze and rapidly correct complex performance problems, something not possible with the existing tools. The Linux Trace Toolkit, LTTng, is used for most of the instrumentation and the trace recording, allowing the tracing of production systems with great accuracy and minimal impact. This approach uses solely kernel instrumentation and does not require the instrumentation or recompilation of processes. The analysis time is linear with respect to trace size. The

memory usage for large traces analysis is also discussed.

2.1 Introduction

As systems become more parallel, developers face an increasing number of performance problems. These problems often have a system-wide scope and occur only on production systems. Since they do not produce execution errors or incorrect results, but only take more time than expected, traditional debugging tools are ineffective against them, making them challenging and time-consuming for developers.

The observable manifestation of a performance bug is that a process takes more time than expected to accomplish a given task – that is, to bring the system from one user observable state to another. In order to fix the underlying problem, the developer must work his way down to the root cause by first finding out how the total elapsed time was spent. This time might have been consumed in three ways.

1. The process was using the processor. Too much time spent this way indicates an intrinsically challenging problem with unavoidable high processor usage, or an algorithmic bug internal to the process.
2. The process was too frequently interrupted by external and often asynchronous events like IRQs, bottom halves or preemption by the operating system.
3. The process was blocked on an operating system wait queue for an excessive amount of time, waiting for some hardware or another process to do something. For example, blocked in a read() system call waiting for data to arrive from a file descriptor, possibly pending a disk read access.

When encountering a performance problem on a small system, developers might guess in which of these states the system is spending too much time by a combination of the usage of operating system metrics (as displayed by tools such as time, ps

or strace), the general feel of the system and instinct. On larger systems, however, doing so is much more difficult, especially if the problem is hard to reproduce or lasts for very short periods of time.

Once the excess time is categorized, the way to fix it depends on its category. Problems of type 1) have been studied for a long time and can be further investigated with the traditional gprof[7] profiler, or, on a live system, with less costly profiling methods like Sysprof[8] or Oprofile[9]. The nature of problems of type 2) can be further understood by looking at operating system metrics such as the number of time each IRQ fired, the number of schedule outs of the process, or the time other processes on the systems ran while the performance problem occurred. Means to investigate problems of type 3) (blocked process) are however much less readily available.

When a process spends too much time blocked, the problem can ultimately be tracked down to hardware contention. This means that some other processes in the system need to use the CPUs or that some hardware like a hard disk or a network card needs to produce data before the process can be unblocked. Although the process itself can be directly blocked on this hardware resource, it is often not the case. Frequently, there is a chain of processes blocked on other processes for several levels until the direct blocking on hardware.

In this paper, we describe a method to debug performance bugs involving blockings of processes. In order to do so, we define a set of instrumentation points to add to the operating system in order to get the necessary data in an execution trace. We propose a trace analysis method that enables the user to determine easily whether the performance problem he is facing is blocking-related. Finally, we contribute an algorithm to extract the chains of blockings that lead down to hardware root causes.

The LTTV Delay Analyzer, which implements the proposed approach, is designed to run on production systems with minimal performance impact. Furthermore, the approach does not necessitate recompilation of code, as it relies on operating system instrumentation only. This black box model of processes enables the use of the method even if the source code to the processes involved is unavailable.

2.1.1 Previous Work

The DTrace dynamic tracer allows the debugging of blocking-related performance problems, as shown in the case study in [5]. The approach for debugging such problems with DTrace consists in initially writing a D script that instruments the symptom and procures more knowledge about the condition in which it occurs. This in turn allows to write other scripts that instrument that condition, walking the causality chain down to the root cause and writing numerous scripts along the way. Such a method requires advanced knowledge of the operating system in order to know where to instrument the code at each step. Also, the bug must be highly reproducible to allow for this iterative method. Finally, because DTrace works on a live system, a D script must contain code to filter out events not related to the bug, which can be a challenge.

Trace analysis tools with control flow views also allow debugging of blocking related performance problems to a certain extent. Such tools include LTTV[18], the Wind River Workbench[19] and QNX Momentics[24]. Unlike DTrace, they work on a recorded trace which, if the instrumentation was sufficient, contains enough information to understand the problem without need for further instrumentation and tracing iterations. The trace can also be recorded with minimal resource usage for a long period of time, waiting for a rare bug to occur, using the flight recorder mode. The approach with these tools is however a manual one. The developer must

inspect the Gantt chart of the control flow view in order to detect the unexpected behavior. This view contains a huge quantity of information and, unless one knows exactly what to look for, is mostly suited to help locate long states. Delays that result from the accumulation of short episodes are more difficult to identify visually. Another disadvantage is the absence of distinction between blockings with different causes on the screen. One must manually verify each one of them, which is a time consuming operation.

TIPME[32] helps identifying causes of user-perceived latency in interactive environments by summarizing the time passed in running, runnable and blocked states between a user input and the resulting graphical update. It does not however extract the chain of blockings.

Pip[31] generates a symbolic specification of the behavior of the program. This output can then be reviewed by the programmer to find anomalies. This specification can also serve as a reference that Pip can use to automatically compare a system behavior with. Alternatively, the programmer can manually write this specification, optionally including temporal constraints. Such an approach makes a compromise between the detail level of the specification and the amount of bugs that can be detected. On a complex and evolving system, maintaining such a specification is challenging.

Magpie[33] traces the progression of a request through the various software components of a system as well as its resource usage. The path is deduced by correlating arguments of events generated by these components using an *event schema*. Magpie helps the programmer understand in what component a request that exhibited performance problems spent too much time. The detail level of this information is, however, related to the detail of the application instrumentation and event schema provided.

Paradyn[30] is a tool to find performance bottlenecks in parallel and distributed systems. It uses dynamic instrumentation that is activated and deactivated during the trace while testing various hypotheses. Paradyn tries to locate the performance problem under tree axes : “why”, “where” and “when”. For each of these axes, a tree of possible causes is defined. Instrumentation is activated selectively for at most one node of a given depth at a time. Such an approach requires that the performance problem lasts for long enough and is dependent on the precision of the model.

In the subsequent sections, we first explain the architecture of the proposed analysis ; we describe the instrumentation, the tracing method and the analysis algorithm itself. We then present some results regarding the performance and memory usage of our technique as well as a case study. We conclude by discussing the advantages of the method and future work avenues.

2.2 Architecture

The approach used consists in instrumenting the Linux kernel and tracing the system with LTTng. The trace is thereafter analyzed by a specially designed LTTV plugin. This analysis tool can execute on a machine whose architecture is different from that on which the trace was recorded. State machines track blocking-related process data and make it available to report creation modules. These produce outputs that may be used by developers in order to understand blocking-related problems in the system.

2.2.1 Instrumentation

We instrumented the Linux kernel with the Linux Kernel Markers[14, 38], an in-kernel API for instrumenting its code. Markers offer virtually undetectable performance impact when deactivated and minimal performance impact when activated.

We instrumented scheduling changes (and whether they are caused by preemption or blocking), process wakeups, IRQ entry/exit, softIRQ entry/exit, trap entry/exit and process forking. Additionally, special *state dump* events are generated at the beginning of the trace that indicate in what control flow state (see section 2.2.3.1) each process is initially. All this instrumentation is kernel-based, no application instrumentation is necessary.

In order to display detailed information in reports, the Delay Analyzer also consumes events that give the following information.

– Mapping between IRQ number and name of hardware using it
– Mapping between softIRQ number and the function handling it
– Mapping between a process ID and its name
– Arguments of the open() system call, in order to save the mapping between file descriptor and file name
– Arguments of the read() system call, in order to know what file caused the system call to block
– What file descriptors caused a poll() system call to wake up

Work is under way to instrument yet more specific system calls.

2.2.2 Tracing

To trace the system, we use the LTTng[16] tracer. Its performance impact is very low, which allows recording traces on production systems. It provides accurate timestamping of events even on multi-processor systems, where the hardware permits. It is partly integrated in the Linux kernel and its integration is ongoing. Furthermore, it allows for an easy extension of its default instrumentation of the kernel.

A trace may be recorded in buffered write-to-disk mode or in *flight recorder* mode. The latter consists in recording the trace in circular buffers in memory. This mode minimizes impact on performance. It is well suited when trying to catch a bug that occurs randomly or rarely and necessitates to record a trace for a long period. The tracing can go on for days if necessary ; a specially designed script may then detect the first occurrence of the bug and stop the recording. The circular buffers are then transferred to disk. Hybrid approaches that use flight recorder for some buffers and write-to-disk for others are also available. Write-to-disk has the advantage of supporting traces larger than the memory.

2.2.3 Trace Analysis

An *a posteriori* analysis of the trace file produces reports that developers may use to study the performance problems at hand. We implemented this analysis in the form of a specially designed LTTV plugin[1] of about 2500 lines. LTTV is a trace viewing and analysis infrastructure. It is designed to handle efficiently traces many times larger than the workstation memory. LTTV may also read traces that

[1]This implementation is available in the LTTV package and repository at `http://ltt.polymtl.ca/`.

were recorded on a system with different byte ordering and integer sizes than the analyzing system.

When debugging a performance problem, the developer wants to know why a process did not get in a certain state (the *target* state) more quickly, starting from a given time, at which it was in a different state (the *initial* state). For example, he might want to know why a reply from a server did not arrive more quickly, starting from the time the client sent the request. He could also want to know why an application was not started more quickly, starting from the moment he clicked on the icon to start it.

Normally, the recorded trace will cover the time span between these initial and target states. The trace will also generally include extra time spans at the beginning and at the end. Therefore, the developer must specify what these initial and target states are to the Analyzer[2]. He does so by specifying a timestamp that corresponds to the initial state, and a trace event that indicates the entry in the target state. The timestamp of this target state event is used as the time at which to end the analysis; moreover this event's process – the process in the context of which it occurred – is the process that will be analyzed. This process needs not be the process that is the root cause of the delay. It only needs to be a process that is exhibiting performance problems, mere symptoms. The analyzer will automatically follow the dependency chain down to the root cause of any blocking problem.

If this target process did not exist yet during a part of the time range that is analyzed, its closest parent is analyzed instead during that period. What the developer wants to know is indeed what the closest parent was doing that was preventing it from creating a closer parent (or the target process itself) earlier.

[2]In many cases however, this is not necessary because performance bottlenecks will often stand out even if there are extra time ranges that were traced at the beginning and at the end of the trace.

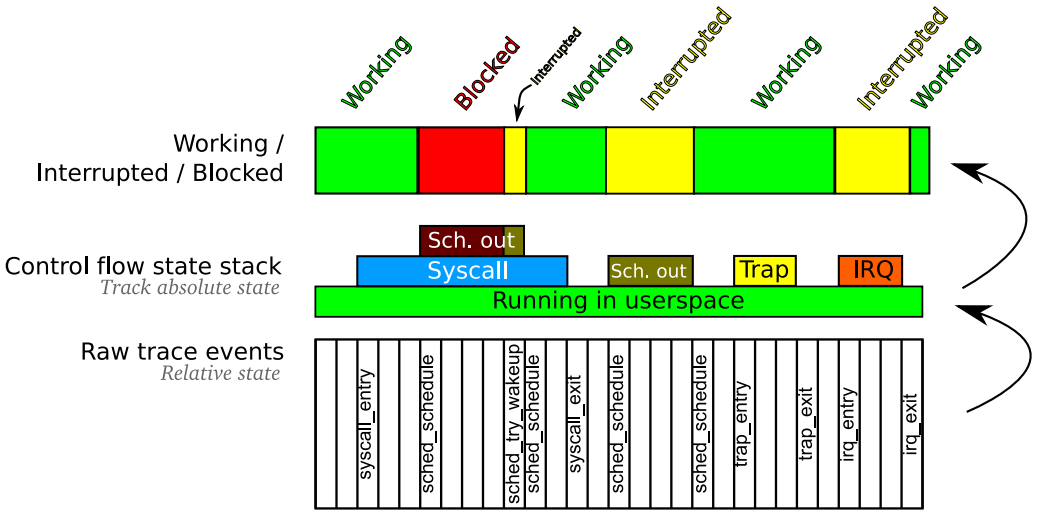

Figure 2.1 The entry and exit events from the raw trace (bottom) are used to infer the Control flow states, which are kept in a stack. These are in turn used to deduce the WIB states, which serve as main input to the report generation.

At this point, the start and end timestamps are used solely for the purpose of creating the reports. The part of the trace that precedes the analysis area is read fully, regardless of the requested initial and target states, in order to build the state of each process. In future versions, however, we plan to integrate the state information into the LTTV checkpoint mechanism, which will require starting to read at the closest preceding checkpoint, rather than the beginning of the trace.

The architecture of the analysis consists in two state machines. Reports are created using the state inferred by these state machines. Figure 2.1 shows the interaction between them. The next sections explain the meaning of these states and how they are inferred.

2.2.3.1 First State Machine : Control Flow State Stack

During its execution, a process enters and leaves control flow states. The first column of table 2.1 lists these states. Control flow states refer to the type of code

that the control flow is executing at a given time, or to the fact that it is not executing. Control flow states can be nested in various ways. For instance, the kernel code that handles an interrupt nests over the interrupted userspace process. When the execution of the kernel interrupt handler is finished, the kernel looks at the stack to find out the return address, back to the previous state, running the userspace process. The kernel instrumentation of the control flow state transitions of processes yields events about relative state transitions. For example, an event might tell that a process just exited the IRQ context, but it will not tell what state it is going back to. Therefore, in order to be able to deduce the control flow state of a process at any time, when reading the trace, a model of the system must be maintained with a stack of these nested states.

This is the purpose of the control flow state stack. While the trace is being read, when an event indicating a state transition is encountered, the control flow state stack of the active process is updated. When an "entry" event occurs, a state is pushed onto the stack ; when an "exit" event is encountered, the state at the top of the stack is popped and the new state is taken to be the one that was underneath. Each state has its specific entry and exit events. The context change event (event kernel_sched_schedule) is a special case. It indicates both that a process is being scheduled out (*Scheduled out* is then pushed on its stack) and that another is being scheduled in (*Scheduled out* is then popped from its stack) on a given CPU.

The *Scheduled out* state can occur in three different circumstances. 1. When a process is preempted by the operating system. In this case, it is pushed and later popped as a consequence of distinct kernel_sched_schedule events. 2. When a process gets scheduled out while blocked, waiting, via a wait queue, for a resource to become available. This state is pushed following a sched_schedule event, but popped following a sched_try_wakeup event that wakes it up. 3. Finally, the third case occurs just after (2) : the process has been woken up but is still not running, it

Tableau 2.1 Control flow states and the additional information kept with them.

Control flow state	Additional information
In userspace	–
In system call	syscall #
In trap	trap #
In IRQ	IRQ #
In softIRQ	softIRQ #
Scheduled out (not executing)	Type : "blocked", "preempted" or "waiting for schedule after blocking"

is waiting to be scheduled. Therefore, when a blocked process is woken (announced by a sched_try_wakeup event), a "blocked"-type *Scheduled out* is popped and a "waiting for schedule after blocking"-type *Scheduled out* is pushed on its control flow state stack.

Some control flow states are accompanied by additional important information. Some of this information is available immediately upon state entry. For example, the kernel_arch_syscall_entry (entry in a system call) event is accompanied by the syscall_id argument, which identifies the system call. Other information is not included directly in the event ; it is rather delivered in later events. For example, the file name associated with an open() system call is announced in a distinct event that comes after the system call entry event. This is necessary because the events that announce entry into a control flow state must fire as close as possible to the actual state boundary in order to obtain a trace that reflects accurately the system state. At such an early point in the system call, its arguments have yet to be decoded.

2.2.3.2 Second State Machine : Working / Interrupted / Blocked (WIB) State

While the Control flow state corresponds to the real process stacks, the WIB (working/interrupted/blocked) state is a higher-level abstraction with no direct equivalent in kernel or user memory. It is more suitable for the study of the blocking behavior of the process. The WIB state is deduced from the Control flow state. Therefore, each time the Control flow state changes, the WIB state is updated if needed. The values it can take are the following.

When a process is running in userspace, it is running code that makes it progress towards fulfilling its purpose ; it is therefore **Working**. When running in a system call, even though the code being executed was not designed by the application programmer, it is still helping the application progress towards its goal by manipulating hardware and kernel structures for it. The same applies to traps.

If an IRQ or a softIRQ occurs in the context of the process, then it is considered **Interrupted**. It is also the case if the process is preempted. Additionally, if the process was blocked but has been woken up but not rescheduled yet, it is considered Interrupted.

If the process is scheduled out and is not runnable, it is considered **Blocked**. In this case, it asked for a resource and that request blocked. This can be either directly or accidentally. Direct requests are done via system calls (for example, a blocking file reading operation, a blocking wait for a network packet). Accidental requests are done via traps (for example, a page fault). A process that is in a trap or a system call is Blocked only while it is actually scheduled out. Otherwise, it is in the Working state.

The time spent in any process p between times t_1 and t_2 is composed of the sum of

time spans during which process p was working, interrupted and blocked. Therefore,

$$T_{p,t_1,t_2} = t_2 - t_1 = T^W_{p,t_1,t_2} + T^I_{p,t_1,t_2} + T^B_{p,t_1,t_2}.$$

These components themselves are the sum of each occurence of the state in the process. For instance,

$$T^W_{p,t_1,t_2} = \sum_{i=1}^{n} (t\_end^W_i - t\_start^W_i)$$

gives the total working time for process p by adding the time spans of the n occurences of the Working state the process was in. The same applies to the other WIB states (Interrupted and Blocked).

Unlike control flow states, WIB states are not stacked, as they can be deduced by examining the Control flow state stack. Table 2.2 shows how the WIB state is deduced from the Control flow state stack.

WIB states are also accompanied by additional information about the state of the process (Table 2.3). This information is extracted from the Control flow state stack and the additional information that accompanies the states in the stack.

The WIB state of each process serves as input to the report generation which is described in the next sections.

2.2.3.3 State Holdback

As explained previously, some information associated to the control flow state comes after the event that indicates the entry in that state. However, the WIB state must be updated at each change in the control flow state. If this is done as soon as

Tableau 2.2 Correspondence between control flow states and WIB states. The reason for which a state was scheduled out, which is enclosed in parentheses, accompanies each *Scheduled out* state in the memory structure.

Control flow state (top of stack)	Resulting WIB state
Running in userspace	Working
Running in system call	Working
Running in trap	Working
Running in IRQ	Interrupted
Running in softIRQ	Interrupted
Scheduled out (runnable)	Interrupted
Scheduled out (preempted)	Blocked
Scheduled out (waiting for schedule after blocking)	Interrupted

Tableau 2.3 Information that accompanies each WIB state.

WIB State	Accompanying information
Working	start time, end time
Interrupted	start time, end time
Blocked	start time, end time, wakeup time, waking process, control flow state stack of waking process at wakeup time

the control flow state changes, some needed information will not be available for the WIB state. For example, a process enters an open() system call. When the kernel_arch_syscall_entry event occurs, the name of the file to be opened is not known yet. The process may be scheduled out because of a blocking in this open(), and the file name may still not be know. In fact, the event that announces the file name arrives just before the event that announces the exit from the system call. The file name must, however, be stored with the Blocked state as its cause.

In order to compensate for this, our approach uses a holdback mechanism. Some Control flow state push operations are not executed immediately on the stack. Rather, they are kept in a queue, if necessary as long as the event commanding the corresponding state pop does not arrive. When it does, its push and pop are processed and, in between, those of the states that were nested on top of it. Therefore, when the WIB state machine sees the control flow states, all their accompanying information is present. This mechanism is only used where necessary.

2.2.3.4 Report : WIB State Summary

The last part of the processing is the production of reports. The first of these reports is the WIB State Summary. See Figure 2.11 for an example.

This summary indicates how much time was spent in each WIB state for a given process, during the time span being analyzed. The summary takes the form of a tree where each node is a state that is a subpart of its parent. The root represents the total elapsed time spent in all states. The Interrupted WIB state is further divided in “IRQ”, “softIRQ”, “preempted” and “waiting for schedule after blocking”. Time passed in IRQs and softIRQs is further classified by its ID. As for the blocked time, it is further classified as having occurred within a system call or in userspace

```
make_state_summary(process, t1, t2)
{
    for each state of process between t1..t2  {
        node = find_node_for_state(state)
        node.total_time += state.t_end - state.t_start

        while not is_root(node) {
            node = node.parent
            node.total_time += state.t_end - state.t_start
        }
    }
}
```

Figure 2.2 Overview of the algorithm for generating the State Summary report. This function is initially called with t1 and t2 representing the requested analysis interval. The nodes are those of the tree-like State Summary report.

(in which a trap occurred), then even further by syscall ID or trap ID. Yet even further, the time blocked in some system calls is classified by special system call-specific criteria. For example, a blocking in the read() system call will be classified by the file that was being read. Figure 2.2 shows the algorithm used for the creation of this report.

In most cases, this report should give a first indication as to what caused a performance problem between the initial and the target states. The programmer can, by comparing the time spent in each of the three WIB states with estimations of normal values, deduce which one lasted too long.

2.2.3.5 Report : WIB State Instances

The WIB State Instances report shows, for each line of the WIB State Summary, the list of time spans during which the process was in that state. The label (for example "<8>") in the state summary refer to such a list of state instances. These instances are ordered by decreasing duration because usually the longest ones will

be the ones that a programmer will want to investigate first. This list can serve to locate instances of the states in the trace in order to study them with another LTTV view. The Control Flow View is most useful for this. The timestamps of each time span can also be used to find instances of blockings in the Blockings Causality report, which is described next.

Figure 2.12 shows an example of WIB state instances report.

2.2.3.6 Report : Blockings Causality

The Blockings Causality report shows the chronological list of time spans during which a process was blocked. It shows in what circumstances it was blocked, as well as how it was woken up (by what process, what IRQ or what softIRQ). Additionally, it recursively shows what the processes that unblocked the process were doing in turn.

Figure 2.13 shows an example of the report. A line that is indented one level further than the previous line means it refers to a blocking that occurred within the waking process of the blocking of the previous line. Blockings at the same indentation level occurred in the same process sequentially.

Therefore, a developer wanting to explore a blocking found in the WIB State Summary can use the WIB State Instances report to find the timestamps of the blocking instances, which will lead him to the right point in the Blockings Causality report. In it, the chain of blockings may be followed down to the root cause of the blocking. The root cause can either be a process (that is CPU-bound or waiting to be scheduled) or a busy hardware resource.

When the deepest cause of a long blocking is a blocking on a process, the system call

```
print_blocking_chain(indent_lev, process, t1, t2)
{
    for each blocking in process between t1..t2 {
        /* Print the blocking time and the context in which
                the blocked process was */
        print_header_line(indent_lev, blocking)
        print_blocking_chain(indent_lev+1, blocking.waker,
                blocking.tstart, blocking.tend)
        /* Print the waker of the blocking and
                the wakeup time */
        print_footer_line(indent_lev, blocking)
    }
}
```

Figure 2.3 Overview of the algorithm for printing the Causality of Blockings report. This function is initially called with indent_lev=0, and t1 and t2 representing the requested analysis interval. See Figure 2.13 for a report example.

it blocked on gives information about the cause. If this is not enough, the analysis can be run on the time span of that last blocking to get a WIB State Summary for that process, which will provide more detailed information about what it was doing. When the deepest blocking is woken by an IRQ or softIRQ, then its ID indicates the hardware that was responsible for the delay.

The method used to produce this report is the following. The WIB states of the process are read sequentially until a blocking is found. The blocking is printed. The control flow state stack of the waking process at the moment it woke the process up is examined. If it was in a softIRQ or IRQ, this fact is printed and the unblocking cause is considered hardware based and unrelated to the process in the context in which it occurred. Otherwise the process was Working and is responsible for the unblocking. In this case, the WIB states of the waking process are read from the time the blocking began to the time it ended. The exact same algorithm is applied to it, but the printing is nested, indented one more level. The next blocking is then read and so on. Figure 2.3 shows the pseudo-code for the algorithm used.

2.2.4 Implementation

In this section, we describe and discuss our experimental implementation.

The analyzer reads a trace once, from its beginning to the end of the range to analyze. The trace must be read from the beginning in order to build the initial system state at the point where the analysis must start. Afterwards, at each interesting event, the control flow state of the corresponding process is updated (possibly with holdback), and its WIB state is computed. If it changed, the new one is added to an in-memory list of the WIB states through which the process passed.

Once the trace is fully read and the WIB state list is complete, the reports are prepared. The creation of the WIB State Summary and of the WIB State Instances report is straightforward. It necessitates a single pass through the array of WIB states for each process. The creation of the Blockings Causality report requires to iterate through the WIB states of each process, printing each blocking. For each blocking that depended on another process, the corresponding range is found in the WIB state array of that process using a dichotomic search. The embedded blockings are all explored this way, recursively.

We have not found a way to build the Blockings Causality report incrementally while reading the trace in one pass, freeing the old WIB states. These old states cannot be freed because it is always possible that a blocking that lasts since the beginning of the trace will be unblocked. The old states of the waker will therefore need to be accessible to print the chain of blockings. Unfortunately, the waker is not known until the waking time.

Our implementation builds WIB state arrays for all processes because the process being analyzed can be unblocked by any process, and we will then want to know what that process was blocked on in order to display it in the Blockings Causality

report.

This approach to the analysis has the advantage that its run time is roughly linear with respect to the size of the trace. However, its memory usage also grows linearly with respect to the trace size because every WIB state is kept in memory. This limits its ability to analyze long traces taken on very busy systems.

We plan to make the memory usage close to constant by not saving these states in memory. Instead, the algorithm will read the trace for only one process at a time – initially, the process containing the *target state* – and create the reports incrementally while reading the trace. For the Blockings Causality report, when a blocking is found in the process, the trace will be read until the end of the blocking is found. When it is, and the waking process is known, the analyzer will seek in the trace to the beginning of the blocking and read events related to its waking process, providing the indented content for that blocking in the report. If further nested blockings are found, another seek will occur, and so on, until the analyzer is back to the initial process and done reading it.

This improved implementation will use much less memory, but will likely take more time, as the trace will be read more than once. However, the analysis time will be bounded by the nesting level of blockings, which is quite low on the traces we saw. Furthermore, it will permit analysis of huge traces on any recent workstation.

2.3 Results

2.3.1 tbench - Analysis Time

We recorded several traces on a system running the tbench[39] benchmark. Tbench generates a workload of network traffic between an arbitrary number of client and

server processes. Both the tbench server and client processes were run locally, on the traced system, as the goal was to produce blockings between local processes, rather than real network traffic. All traces were recorded and analyzed on a dual-core 3 GHz system with 4 GB of RAM.

Figure 2.4 shows the analysis time with several trace sizes, both with one and two processors. For the single-processor traces, one of the cores was deactivated using the Linux CPU hotplug facility. The analysis itself was always run with both cores enabled, but is single-threaded and therefore used only one core. Figure 2.5 shows similar measurements, but with a varying number of tbench clients.

Figure 2.4 indicates that with either one or two processors, the analysis time grew linearly with the size of the trace. The nth point for one processor corresponds to a trace of the same duration as the nth point for two processors. The clusters from left to right represent traces of 1 to 8 seconds, at 1 second intervals. For a given trace duration, the analysis time doubles when the number of processors doubles. This is not surprising, because the system can accomplish twice as much work, therefore the trace size doubles.

Figure 2.5 indicates that the analysis time grows linearly with the trace size, with different numbers of tbench clients. The analysis time appears independent of the number of clients.

2.3.2 tbench - Memory Usage

Figures 2.6 and 2.7 show the memory usage of the analyzer while it is analyzing 8 second traces recorded on a system loaded with tbench. The memory usage is quite high for such short traces because of the correspondingly high rate of system calls done by tbench. Figure 2.6 shows two traces, one recorded on the same dual-core

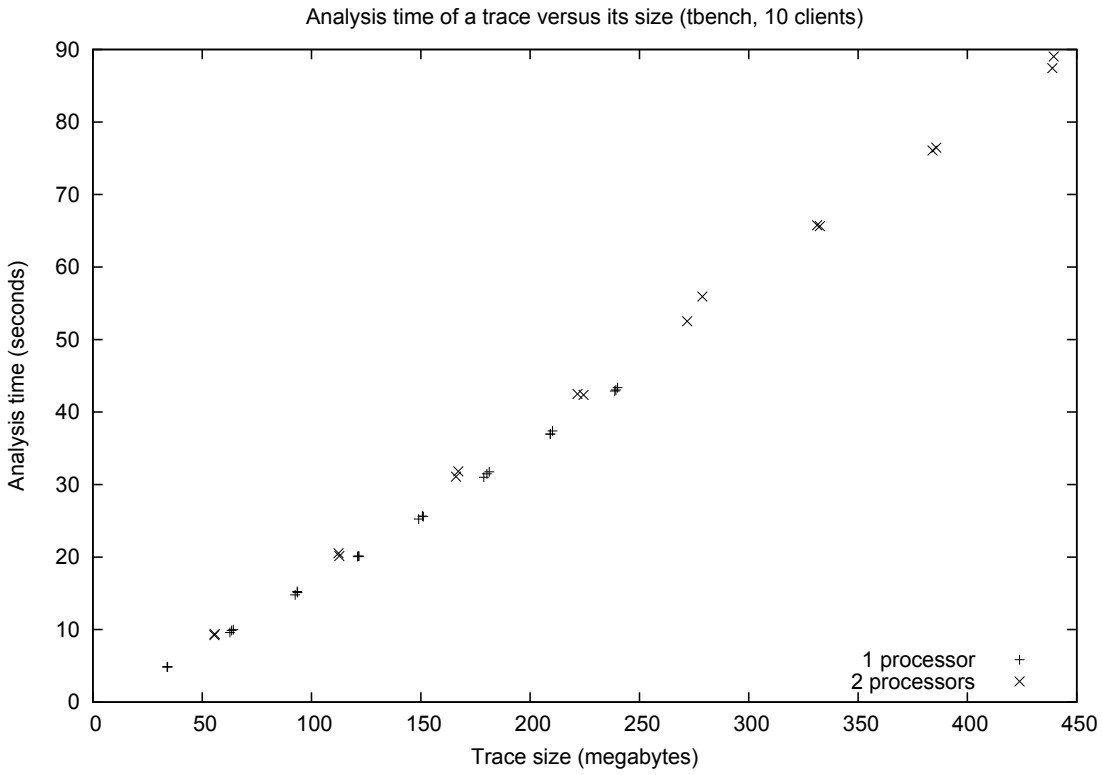

Figure 2.4 Analysis time versus trace size, system traced while running tbench, with one and two cores. Clusters, from left to right, correspond to traces that lasted 1 to 8 seconds.

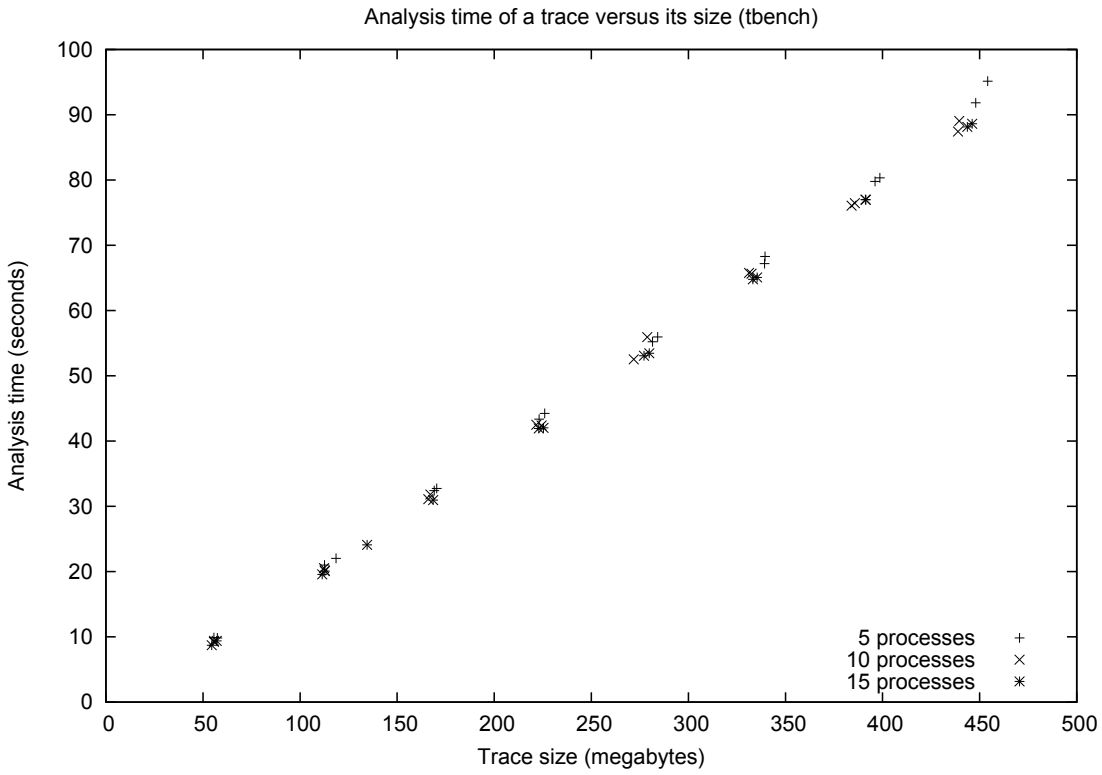

Figure 2.5 Analysis time versus trace size, system traced while running tbench, with various numbers of tbench client processes. Clusters, from left to right, correspond to traces that lasted 1 to 8 seconds.

machine as before, the other recorded on the same system, with one core disabled. Figure 2.7 shows similar measurements done with both cores active, but with a varying number of tbench clients.

In Figure 2.6, the memory usage increases linearly as the trace is read. The analyzer working on the one-core trace stops increasing its memory usage about twice as fast as its counterpart. This is because the 8 second trace taken on a two-core system is twice as big as the one taken on the single-core system, because the system was able to do twice as much work. Its parsing takes twice as long. Afterwards, both have a constant memory usage for some time while the reports are being generated. The largest portion of the report creation time is for the Blockings Causality report. Creating this report for the two processor trace takes about twice as long because there are about twice as many WIB states.

Figure 2.7 shows that the memory usage is about the same for all client counts. In all cases, the two processors were saturated by a number of clients greater than the number of CPUs, resulting in traces roughly equal in size and in parsing time.

2.3.3 Web Server

To get an impression of the performance of the analyzer under a real-life load, we traced a web server.

We hosted a copy of the Tracing Wiki[40] on the same system as before. We used the MediaWiki[41] engine 1.13.2 with memcached[42] 1.2.2, that we hosted with Apache[43] 2.2.9 and MySQL[44] 5.0.51a.

We traced the server while two other machines were downloading each a full copy of the wiki (pages, images, documents). We used wget[45] as client. For each run,

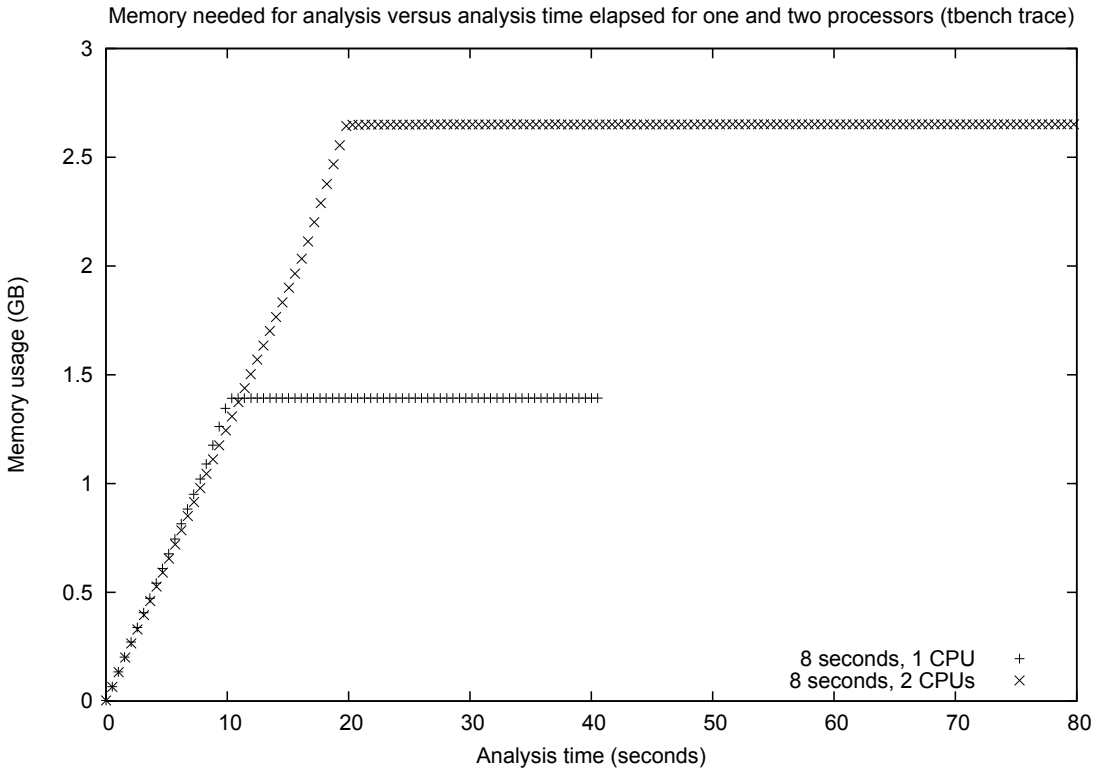

Figure 2.6 Evolution through time of the analyzer memory usage while processing 8 second traces recorded on a system running tbench on one and two cores.

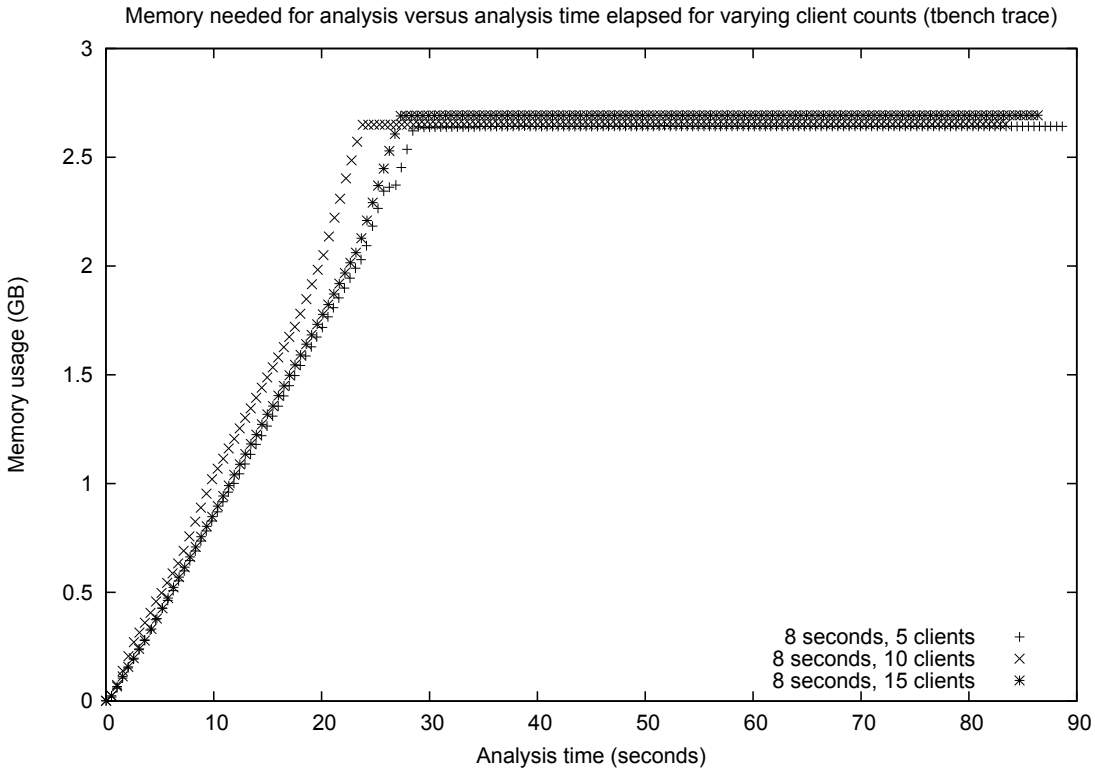

Figure 2.7 Evolution through time of the analyzer memory usage while processing 8 second traces recorded on a system running tbench with varying client counts.

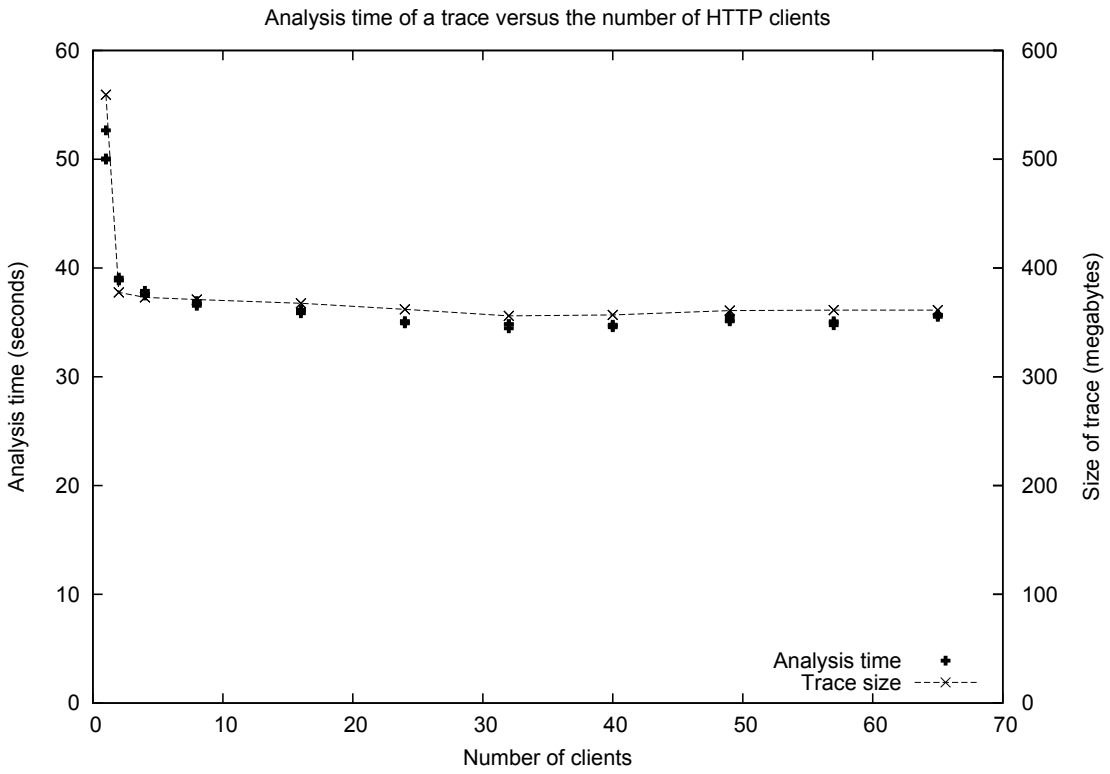

Figure 2.8 Analysis time and trace size versus number of web clients.

we varied the total number of wget clients (half on each server) between which the full download was spread.

Figure 2.8 confirms that the analysis time does not vary with the number of clients but is rather constant. The abnormally high value for two clients accompanies a correspondingly large trace size. This is probably due to the fact that the system is not fully used with only two clients. It therefore takes longer for the downloads to complete, resulting in a larger trace that takes longer to analyze.

Figure 2.9 shows the increase of analysis time versus the number of events in a trace. To obtain this graph we used one of the traces recorded for Figure 2.8. We ran the analysis simulating the end of the trace after varying event counts. As expected, the analysis time seems to grow linearly with the number of events read.

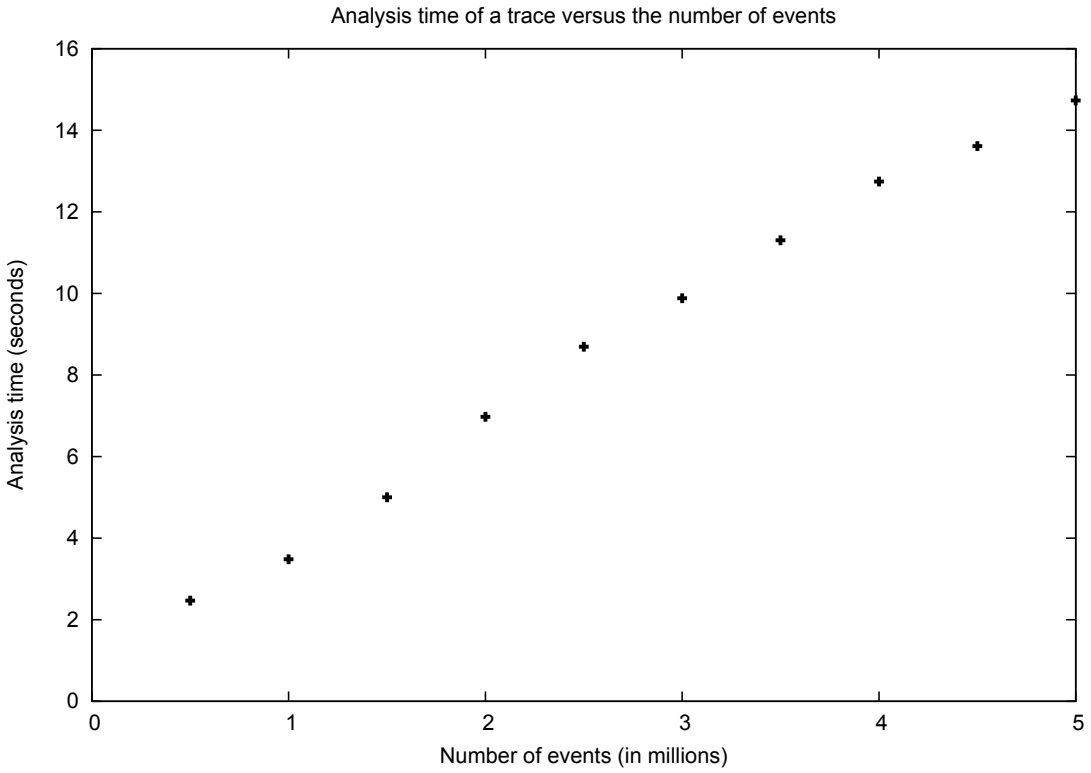

Figure 2.9 Analysis time versus the number of events in a web server trace.

2.3.4 Case Study

On the computer described in section 2.3.1, starting Xorg[46] and the KDE[47] desktop took around 2.5 seconds. Although this is much better than some of our older systems, the fact that this machine had a very fast feel once the desktop was loaded let us under the impression that X was blocked at some point during the starting sequence. We thus traced the starting of X and KDE.

On this system, running the startx script starts X and KDE. We started the trace just before running startx, in the same command. In order to know when the desktop was completely loaded, we put a script in the KDE Autostart directory (~/.kde/Autostart). Scripts in this directory are run as soon as the desktop is loaded, enabling us to locate quickly that point in the trace. The script ran a command to stop the trace. Although it was convenient to have the trace contain exclusively the interesting time span, it was not mandatory. We could have manually started the trace well before running startx and stop it well after the execution of the Autostart script.

We then ran the Delay Analyzer on the trace. As start and end points for the analysis, we used respectively an event at the beginning of the startx process and an event at the beginning of our Autostart script. As process to analyze, we chose the Autostart script, since the goal was to understand why it was not started earlier.

The Autostart script is run at the end of the trace, so the analyzer must look at its closest parent while analyzing the time range before its creation. This resulted in a chain of eight processes illustrated in Figure 2.10. The startup operation of X/KDE is relatively complex; other processes not involved in this parental chain run concurrently with it. One of them is Xorg, the process of the X server.

The WIB State Summary for this critical path showed the usage of time in each of

After the start event,
startx (pid 420) *ran for* **0.005269443 s**
before creating
xinit (pid 438) *which ran for* **1.309108247 s**
before creating
x-session-manager (pid 445) *which ran for* **0.116666736 s**
before creating
kdeinit (pid 500) *which ran for* **0.029303945 s**
before creating
kdeinit (pid 503) *which ran for* **0.240340942 s**
before creating
kdeinit (pid 519) *which ran for* **0.009239575 s**
before creating
kdeinit (pid 520) *which ran for* **0.806551758 s**
before creating
the autostart script (pid 542) *which ran for* **0.000002343 s**
before reaching the end event

Figure 2.10 Parental relationships from startx to the autostart script.

these processes before it created the next one in the chain – that is, the time that delayed the execution of the Autostart script. This time is shown in Figure 2.10.

The first process we looked at was xinit, as it accounted for about half the start time. Figure 2.11 shows that xinit basically spent the 1.3 seconds waiting in select() system calls.

```
Process 438 [/usr/bin/xinit]
Total (1.309108247) <0>
  Blocked (1.294384393) <1>
    Syscall 23 [sys_select+0x0/0x16c] (1.294379908) <8>
    Syscall 57 [stub_fork+0x0/0x11] (0.000002859) <2>
    Syscall 59 [stub_execve+0x0/0xc0] (0.000001626) <4>
  Interrupted (0.009927037) <5>
    Scheduled out (0.009908719) <7>
    Waiting for schedule after blocking (0.000018318) <6>
  Working (0.004796817) <3>
```

Figure 2.11 WIB State Summary for xinit before it created x-session-manager.

```
Node id: <8>
439924.976620220-439926.270851832 (1.294232)
439926.270856657-439926.270922847 (0.000066)
439926.271033308-439926.271045577 (0.000012)
439926.271070723-439926.271079202 (0.000008)
439926.271061869-439926.271068187 (0.000006)
439926.271231493-439926.271237701 (0.000006)
[...]
```

Figure 2.12 WIB State Instances report for blockings on select() in xinit.

Jumping to the WIB State Instances report for those select() calls gives Figure 2.12. Only one select() is responsible for 1.29 seconds of blocking. Let's look at the Blockings Causality report near that precise blocking.

The Blockings Causality report (Figure 2.13) shows that the 1.29 second long select() was woken by Xorg, the X server process. We were surprised to see that during that time, Xorg was itself blocked on a series of more than 40 nanosleep() calls lasting 0.01 to 0.03 seconds each.

An examination of the trace events shows that between the nanosleep() calls, X reads and writes data to a file descriptor, which is opened just before the nanosleep() sequence starts. This file is /dev/psaux, the character device to talk to PS/2 devices.

Looking at the Xorg configuration file revealed that it was setup to use /dev/psaux to communicate with the mouse. This character device is a PS/2 mouse emulator. A USB mouse is actually connected to the system, but X was using a much slower, emulated PS/2 interface. Even worse, the driver was configured for protocol autodetection, which consumed even more time.

After changing the X configuration to use the more modern *evdev* interface to communicate with the mouse, X took around 1.5 seconds to start, a 40% improvement

```
Process 438 [/usr/bin/xinit]
[...]
Blocked in RUNNING, SYSCALL 59 [stub_execve+0x0/0xc0], (times: [...], dur: 0.000002)
Woken up in context of 3 [migration/0] in WIB state UNKNOWN
Blocked in RUNNING, SYSCALL 23 [sys_select+0x0/0x16c], (times: [...], dur: 1.294232)
        Blocked in RUNNING, SYSCALL 2 [sys_open+0x0/0x17], (times: [...], dur: 0.013940)
        Woken up by a SoftIRQ: SoftIRQ 8 [rcu_process_callbacks+0x0/0x47]
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.200006)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 23 [sys_select+0x0/0x16c], (times: [...], dur: 0.203262)
        Woken up by a SoftIRQ: SoftIRQ 1 [run_timer_softirq+0x0/0x21a]
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010001)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010005)
        Woken up by an IRQ: IRQ 239 []
        Blocked in RUNNING, SYSCALL 35 [sys_nanosleep+0x0/0x62], (times: [...], dur: 0.010002)
        [... 37 other nanosleeps() lasting 0.01 to 0.03 seconds ...]
Woken up in context of 440 [/usr/bin/Xorg] in WIB state WORKING
[...]
```

Figure 2.13 Blockings Causality report showing the blockings inside the select() blocking of xinit. Time ranges were removed for lack of space. Each "Blocked in..." line shows the contents of the control flow stack of the blocked process.

from the original value.

2.4 Conclusion

We have described a method for analyzing performance problems on multi-core systems using LTTng and the LTTV Delay Analyzer. The instrumentation required and the method used for recording traces were explained. Afterwards, we described the method used for analyzing the trace and producing reports that allow the programmer to understand performance problems exhibited by a system.

Our current implementation may exhibit memory usage problems on extremely large traces, since it increases slowly but linearly with the trace size. However, modifications that would result in nearly constant memory usage while retaining a linear execution time increase, with respect to trace size, were described.

2.4.1 Future Work

Our approach could be extended to follow dependencies between processes on different computers communicating through a network. This case is similar the one of communicating processes on the same computer. For example, when a process is blocked waiting for a packet to arrive from the network, the Blockings Causality report could show what the process that eventually sends that packet was blocked on. Tracing of several nodes simultaneously would be necessary. Algorithms to reconcile packet sends and receives would also be required.

Another similar application is to follow blocking chains across the physical/virtual machine boundary. Applying our approach as presented in this paper to virtual machines presents challenges. Indeed, the virtual hardware causing a delay may be

software emulated. LTTV is already able to combine traces taken simultaneously on a virtual machine and its host, provided a common time base is used by the tracer.

The addition of a graphical tool to explore the reports would allow to display more information and to cross-reference it with that of other LTTV plugins. Finally, more instrumentation, notably for system call arguments, would permit the production of more detailed reports.

2.5 Acknowledgements

We wish to thank Mathieu Desnoyers for his insightful comments about tracing and performance analysis, and for answering the first author's numerous questions about LTTV and LTTng.

CHAPITRE 3

RÉSULTATS COMPLÉMENTAIRES

L'article du chapitre précédent présentait un cas où un bogue de performance a pu être trouvé grâce à l'Analyseur de Délais de LTTV. Dans ce chapitre, un second cas est présenté.

3.1 Étude de cas : démarrage de Vim

Lors d'une session de travail sur un terminal d'une machine distante avec ssh, il a été observé que le démarrage de l'éditeur Vim, en mode texte, était lent. Chaque démarrage nécessitait environ 3 secondes. Il s'agit d'une situation irritante lorsqu'on a plusieurs fichiers à éditer ; Vim démarre normalement instantanément. Sous tous les autres aspects, le terminal était aussi rapide qu'un terminal local.

La question s'est donc posée à savoir qu'est-ce que Vim faisait durant ces trois secondes. L'analyseur de délais a été utilisé pour y répondre.

Premièrement, une trace puis Vim ont été démarrés de façon synchrone. Ensuite, dès le démarrage de Vim terminé et l'écran de bienvenue visible, la trace a été stoppée et analysée avec l'Analyseur de délais.

Le premier rapport obtenu est le sommaire des états du processus de Vim pour l'ensemble de la trace (figure 3.1). On voit rapidement que le temps n'a ni été consommé par des IRQ, softIRQ, ou par des préemptions (0,02 s), ni parce que le code de Vim travaillait (0,02 s). En revanche, le programme a été bloqué pendant

```
Process 27541 [/usr/bin/vi]
Total (3.414892) <185>
    Blocked (3.253863) <188>
        Syscall 168 [sys_poll+0x0/0x6c] (3.040153) <203>
        Syscall 142 [sys_select+0x0/0x16c] (0.203307) <205>
        Syscall 162 [sys_nanosleep+0x0/0x94] (0.010003) <202>
        Syscall 3 [sys_read+0x0/0x93] (0.000356) <189>
            Unknown filename, fd 3 (0.000356) <190>
        Syscall 240 [sys_futex+0x0/0xc2] (0.000043) <199>
    Unknown (0.109651) <186>
    Working (0.027885) <187>
    Interrupted (0.023494) <191>
        Scheduled out (0.020775) <198>
        IRQ (0.001437) <192>
            IRQ 239 [(null)] (0.001184) <193>
            IRQ 19 [eth0] (0.000253) <200>
        SoftIRQ (0.001282) <194>
            SoftIRQ 3 [net_rx_action+0x0/0x182] (0.000795) <201>
            SoftIRQ 1 [run_timer_softirq+0x0/0x161] (0.000340) <195>
            SoftIRQ 5 [tasklet_action+0x0/0x10b] (0.000138) <196>
            SoftIRQ 6 [run_rebalance_domains+0x0/0x9f] (0.000005) <197>
            SoftIRQ 2 [net_tx_action+0x0/0xd4] (0.000005) <204>
```

Figure 3.1 Rapport du sommaire des états de Vim.

```
Format: timestart-timeend (duration)
5618707.647785-5618707.883659 (0.235874s)
5618707.426038-5618707.647437 (0.221399s)
5618710.111334-5618710.258759 (0.147425s)
5618709.218282-5618709.336185 (0.117903s)
[... plus de longs états ci-dessous ...]
```

Figure 3.2 Rapport des instances d'états de Vim.

environ 3 secondes, soit la quasi totalité de son temps de démarrage. L'analyseur indique dans quels appels système Vim a été bloqué. La majorité du temps de blocage a été consommé à l'intérieur de l'appel poll(), donc Vim était bloqué sur un descripteur de fichier. Pour en savoir plus, il faut consulter le rapport des instances d'états en cherchant à la section <203>.

Cette section du rapport des instances d'états est illustré à la figure 3.2. Le rapport montre plusieurs cas où Vim a été bloqué pendant plus de 0,1s. Le blocage le plus long (0,23s) est particulièrement intéressant. La connaissance de son intervalle de temps grâce à ce rapport permet de le retrouver dans le rapport des chaînes de blocages.

La chaîne est reproduite à la figure 3.3. L'intervalle de temps du rapport des instances d'états permet d'y retrouver le plus long blocage dans l'appel poll(). La ligne en-dessous indique que c'est dans le softIRQ 3 qu'a eu lieu le réveil. Ce softIRQ, associé à la fonction net_rx_action(), est appelé pour traiter un paquet qui arrive dans le système. En-dessous, on peut voir que d'autres longs blocages ont été réveillés dans le même contexte.

Vim attend donc du trafic réseau pendant son démarrage. Reste à voir pourquoi un éditeur de texte en console tel que Vim aurait besoin de communiquer par réseau durant son démarrage.

Une capture du trafic réseau a montré que Vim communiquait avec le serveur X de la machine locale (celle de l'utilisateur, non celle où Vim s'exécutait) via le tunnel X ssh. En effet, Vim, même en mode texte, se connecte à X lorsque possible afin de permettre l'utilisation du presse-papiers. Ainsi il est possible pour l'utilisateur de copier du texte de Vim vers d'autres applications sur son serveur X local. Une consultation du manuel de Vim a permis de découvrir que l'option -X désactivait cette connexion. Elle a été ajoutée et Vim démarre maintenant instantanément.

```
Process 27541 [/usr/bin/vi]
        [...]
        Blocked in SYSCALL 3 [sys_read+0x0/0x93], (times: 5618707.286697-5618707.287053, dur: 0.000356)
        Woken up in context of 17640 [bash] in high-level state RUNNING
        Blocked in SYSCALL 240 [sys_futex+0x0/0xc2], (times: 5618707.327408-5618707.327451, dur: 0.000043)
        Woken up in context of 27542 [/usr/bin/vi] in high-level state RUNNING
        Blocked in SYSCALL 162 [sys_nanosleep+0x0/0x94], (times: 5618707.396723-5618707.406726, dur: 0.010003)
        Woken up by an IRQ: IRQ 239 [(null)]
        Blocked in SYSCALL 168 [sys_poll+0x0/0x6c], (times: 5618707.426038-5618707.647437, dur: 0.221399)
        Woken up by a SoftIRQ: SoftIRQ 3 [net_rx_action+0x0/0x182]
        Blocked in SYSCALL 168 [sys_poll+0x0/0x6c], (times: 5618707.647785-5618707.883659, dur: 0.235874)
        Woken up by a SoftIRQ: SoftIRQ 3 [net_rx_action+0x0/0x182]
        Blocked in SYSCALL 168 [sys_poll+0x0/0x6c], (times: 5618707.884066-5618707.993661, dur: 0.109595)
        Woken up by a SoftIRQ: SoftIRQ 3 [net_rx_action+0x0/0x182]
        Blocked in SYSCALL 168 [sys_poll+0x0/0x6c], (times: 5618707.993982-5618708.104997, dur: 0.111015
        [...]
```

Figure 3.3 Rapport des chaînes de blocages de Vim.

CHAPITRE 4

DISCUSSION GÉNÉRALE

Le but de cette section est d'ajouter quelques éléments de discussion à ceux déjà présentés dans l'article, et de comparer l'Analyseur de délais de LTTV avec les outils existants.

4.1 Performance

Les résultats obtenus indiquent que le temps d'analyse d'une trace par l'Analyseur de délais LTTV est linéaire par rapport à la taille de la trace. Il ne serait pas possible d'obtenir une meilleure complexité algorithmique sans pré-traitement étant donné que la lecture d'un fichier est une opération dont le temps est lui-même linéaire par rapport à la taille du fichier.

4.2 Instrumentation

La méthode d'analyse de délais présentée est relativement générique. Elle attribue à chaque processus un état, dont le changement est gouverné par une instrumentation compacte, générique et facile à intégrer à bon nombre de systèmes d'exploitation actuels. Elle pourrait donc être adaptée à d'autres systèmes que Linux. Cette instrumentation est résistante à des changements au code du noyau, notamment à l'ajout d'appels système ou à la modification de la structure du code des appels système actuels. En effet, elle se situe à un niveau plus bas et instrumente le code générique de blocage plutôt que les cas particuliers. Une instrumentation supplé-

mentaire est utilisée dans le cas de certains appels système afin de fournir des informations supplémentaires dans les rapports, mais elle n'est nullement obligatoire. Il pourrait notamment être avantageux de la désactiver lors du traçage d'un système en production ou embarqué, où le débit d'événements doit être limité.

Il s'agit là de plusieurs avantages par rapport à une approche par le haut comme celle de LatencyTOP, par exemple. Son instrumentation se situe dans le code de chaque appel système pouvant bloquer. Elle est donc plus intrusive, moins compacte et moins portable.

4.3 Analyse de requêtes

Pip et Magpie sont des outils axées vers l'analyse de performance du traitement de requêtes. L'Analyseur de délais peut aussi aider à cette tâche. Si, comme événement d'état initial (*initial state event*) on choisit un événement lié à l'arrivée de la requête dans le système et, comme événement d'état cible (*target state event*), un événement lié à la complétion de la requête, l'Analyseur de dépendances peut dire ce qui s'est passé entre les deux. Il est possible cependant que certains algorithmes de traitement de requêtes, notamment ceux qui impliquent des entrées/sorties asynchrones (*asynchronous I/O*) rendent l'analyse difficile[1], mais ce n'était pas le cas des applications qui ont été tracées dans le cadre de ce projet.

Pip et Magpie peuvent analyser les requêtes dans les systèmes distribués. Ce n'est pas le cas de l'Analyseur de délais. Bien que des travaux aient été faits dans cette direction[48], LTTng ne peut pour l'instant réconcilier des traces prises sur plu-

[1] L'approche des entrées/sorties asynchrones permet aux programmes de faire des opérations d'entrées/sorties, sans toutefois demeurer bloqués sur ces opérations. Il est ainsi possible pour un même processus de traiter simultanément plusieurs requêtes. Dans ce contexte, une analyse des blocages peut être insuffisante pour lier le symptôme d'un problème de performance à sa cause. Cette lacune pourrait être comblée par l'instrumentation de l'application.

sieurs machines en parallèle. L'Analyseur de délais pourrait facilement être étendu pour suivre les chaînes de blocage à travers les frontières d'une machine. En effet, lorsqu'un processus sur un ordinateur est bloqué en attente d'un autre processus sur un autre ordinateur, il s'agit essentiellement du même patron que si les deux processus étaient sur le même ordinateur, mais les événements qui indiquent les changements d'états sont différents.

Pip et Magpie ont en outre des fonctionnalités de détection automatique de fautes que l'Analyseur de délais n'a pas, étant donné que ce n'est pas son but.

4.4 Préparation

Le fait d'utiliser une instrumentation noyau exclusivement permet de simplifier grandement la tâche du développeur qui fait le débogage. Il peut ainsi déboguer tous les composants d'un système sans avoir à les recompiler ni à les instrumenter. Il s'agit là d'un avantage substantiel par rapport à Pip, Magpie et Paradyn.

Ces outils requièrent également de spécifier d'avance certains patrons à vérifier. DTrace, quant à lui, requiert que son utilisateur spécifie explicitement ce qui est recherché dans un script D au moment de l'exécution. L'Analyseur de délais permet de procéder immédiatement au traçage et à l'analyse.

4.5 Durée et fréquence des problèmes pouvant être tracés

L'approche par traçage utilisée par LTTng et l'Analyseur de délais LTTV a l'avantage d'avoir peu d'impact sur les performances. Il permet donc de tracer des systèmes en production durant une longue période afin de recueillir de l'information sur des problèmes qui se produisent rarement. La faible quantité d'instrumentation

nécessaire minimise encore davantage cet impact. Magpie et Pip semblent permettre de le faire également. L'approche de DTrace requiert une intervention itérative qui n'est pas idéale pour les problèmes qui se produisent rarement.

4.6 Croisements avec autres événements

Un avantage très important du fait que l'Analyseur de délais utilise un traceur générique comme LTTng est la possibilité de faire des croisements avec les autres événements recueillis par le traceur, grâce aux autres vues fournies par le visualisateur. Il a été constaté que cette information permettait de beaucoup mieux saisir le comportement du système étudié.

Pour d'autres approches plus *ad-hoc* comme LatencyTOP, il est impossible d'en faire autant. Magpie et Pip cependant en ont également la possibilité.

Cet avantage devient particulièrement intéressant lors du débogage d'un bogue se produisant très rarement. Une fois que la trace est enregistrée, elle contient des informations sur une foule d'aspects du système qui permettent de comprendre substantiellement mieux un problème avant de passer à une itération de traçage subséquente. Il s'agit d'un net avantage par rapport à l'approche de DTrace.

4.7 Chaînes de blocages

L'intérêt principal de l'Analyseur de délais réside dans sa capacité d'aider au débogage en présence d'une chaîne de blocages.

Premièrement, le sommaire des états WIB permet au développeur de repérer qu'il a affaire à une situation impliquant des blocages.

Deuxièmement, l'Analyseur extrait les chaînes de blocages des traces. Ces chaînes permettent de retrouver, à partir d'un processus qui exhibe des symptômes de lenteur due à un blocage, la cause de cette lenteur.

L'Analyseur de délais de LTTV semble être le premier outil à extraire cette information.

CONCLUSION

Les systèmes informatiques sont de plus en plus grands, parallèles et intégrés. Cette complexité engendre de nombreux bogues de performance liés à l'interaction de composants et dont une chaîne de causalité parfois longue lie le symptôme à la cause. Conçus pour des bogues séquentiels et isolés, les outils de débogage traditionnels sont inefficaces face à ces problèmes. Leur repérage et leur correction sont donc très difficiles et coûteux. Ainsi, la disponibilité d'outils qui permettent le débogage efficace de problèmes de performance sur des systèmes complexes en production est de plus en plus important.

Les trois facteurs pouvant causer un délai dans un processus sont (1) un problème algorithmique dans le code du programme, (2) une interruption de l'exécution due à une cause externe et (3) un blocage sur une ressource ou un autre processus. Parmi ceux-ci, le blocage est celui pour lequel le moins d'outils existent. Pourtant, les chaînes de blocages peuvent causer des bogues extrêmement difficiles.

Dans ce mémoire, l'Analyseur de dépendances de LTTV a été présenté. Cet outil attribue à chaque instant écoulé entre deux événements dans une trace, une cause. Il permet ainsi au développeur de voir quelle cause a pris plus de temps que prévu. Si un blocage est responsable de cette consommation excessive de temps, ses causes peuvent être examinées grâce à l'affichage de la chaîne de blocages.

L'instrumentation utilisée par l'Analyseur de délais a été décrite. Celle-ci est simple, compacte, générique, portable et résistante aux changements dans le code du système d'exploitation. Combinée à l'utilisation de LTTng, elle permet un impact en performance minimal pour le traçage de systèmes en production.

Les algorithmes permettant d'analyser les traces et d'en extraire des rapports dé-

crivant l'activité du système et ses chaînes de blocages ont été décrits.

L'Analyseur de dépendances de LTTV est vraissemblablement le premier outil à pouvoir extraire automatiquement les chaînes de dépendances. Il est également un des mieux adaptés au débogage de problèmes de performance rares sur des systèmes en production, lorsqu'une recompilation des applications n'est pas souhaitable.

Les mesures prises ont indiqué un temps d'analyse qui croît linéairement avec la taille de la trace. La consommation mémoire est quant à elle également linéaire par rapport à la taille de la trace, et une méthode pour la rendre quasiment constante a été décrite.

Finalement, les études de cas présentées démontrent l'utilité de l'approche pour déboguer des problèmes de performance dus à des blocages.

Travaux futurs

Il serait souhaitable d'étendre cette approche pour que des analyses de délais puissent être faites sur plusieurs machines, constituant un système distribué, qui communiquent par réseau. Il serait alors possible, par exemple, en traçant un client et les serveurs, d'expliquer le temps consommé à toutes les étapes d'une requête, du clic de souris fait par l'utilisateur dans son navigateur web jusqu'à l'affichage du résultat sur son écran, en passant par le traitement de la requête par le serveur.

Une autre avenue prometteuse est le traçage de machines virtuelles et de leur machine physique. Beaucoup de composants « matériels » d'une machine virtuelle sont en fait implantés en logiciel. Le contrôleur réseau, le contrôleur de son, le contrôleur vidéo et les disques en sont des exemples. Lorsqu'une commande leur est envoyée, une chaîne de blocages peut être déclenchée sur la machine physique en

plus de celle qui a déjà mené à la requête dans la machine virtuelle. Cette complexité supplémentaire rend plus difficile le débogage de problèmes de performance sur les machines virtuelles. L'Analyseur de délais pourrait être modifié pour montrer les causes physiques ayant entraîné des délais.

L'ajout d'une interface graphique pour l'exploration des rapports générés ainsi que son intégration aux autres interfaces graphiques de LTTV permettrait l'affichage de davantage d'informations et faciliterait la consultation simultanée d'autres aspects du problème.

Enfin, l'instrumentation spécifique aux appels système pourrait être augmentée afin de détailler davantage l'utilisation des ressources dans le sommaire des états WIB.

RÉFÉRENCES

[1] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, et J. Rattner, "Platform 2015 : Intel Processor and Platform Evolution for the Next Decade," p. 12, 2005.

[2] U. Hoelzle, J. Dean, et L. Barroso, "Web Search for a Planet : The Architecture of the Google Cluster," *IEEE Micro Magazine*, p. 22–28, 2003.

[3] M. Bligh, M. Desnoyers, et R. Schultz, "Linux kernel debugging on google-sized clusters," dans *Linux Symposium*, (Ottawa, Ontario, Canada), June 2007.

[4] R. W. Wisniewski, R. Azimi, M. Desnoyers, M. Maged, J. Moreira, D. Shiloach, et L. Soares, "Experiences understanding performance in a commercial scale-out environment," *Lecture Notes in Computer Science*, vol. 4641, p. 139–150, 2007.

[5] B. Cantrill, M. Shapiro, et A. Leventhal, "Dynamic instrumentation of production systems," (Boston, MA, USA), p. 15–28, 2004.

[6] B. Cantrill, "Hidden in plain sight," *Queue*, vol. 4, no. 1, p. 26–36, 2006.

[7] S. Graham, P. Kessler, et M. McKusick, "gprof : a call graph execution profiler," vol. 17, (Boston, MA, USA), p. 120–6, 1982.

[8] S. Sandmann, "Sysprof–a system-wide linux profiler." `http://www.daimi.au.dk/~sandmann/sysprof/`. Vérifié le 2009/01/05.

[9] J. Levon et P. Elie, "Oprofile : A system profiler for Linux," 2005.

[10] "Intel VTune." `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`. Vérifié le 2009/01/05.

[11] R. Stallman, R. Pesch, et S. Shebs, *Debugging with GDB : The GNU Source-level Debugger*. Cambridge, MA : Free Software Foundation, 2002.

[12] "ptrace(2) Linux man page,"

[13] "Java Logging API." http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html. Vérifié le 2009/01/13.

[14] J. Corbet, "Kernel markers," *LWN.net*, août 2007. http://lwn.net/Articles/245671/. Vérifié le 2009/01/05.

[15] "Performance analysis of linux kernel markers 0.20 for 2.6.17." http://sourceware.org/ml/systemtap/2006-q3/msg00793.html. Vérifié le 2009/01/13.

[16] M. Desnoyers et M. R. Dagenais, "The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux," dans *Linux Symposium*, (Ottawa, Ontario, Canada), June 2006.

[17] K. Yaghmour, R. Wisniewski, R. Moore, et M. Dagenais, "relayfs : An efficient unified approach for transmitting data from kernel to user space," dans *Linux Symposium*, (Ottawa, Ontario, Canada), 2003.

[18] "LTTV." http://ltt.polymtl.ca. Vérifié le 2009/01/05.

[19] "Wind River Workbench." http://www.windriver.com/products/workbench/. Vérifié le 2009/01/05.

[20] "Shark." http://developer.apple.com/tools/performance/optimizingwithsystemtrace.html. Vérifié le 2009/01/13.

[21] "Instruments." http://developer.apple.com/documentation/developertools/Conceptual/InstrumentsUserGuide. Vérifié le 2009/01/13.

[22] "DTrace information on the TracingWiki." http://ltt.polymtl.ca/tracingwiki/index.php/DTrace. Vérifié le 2009/01/05.

[23] F. Eigler, "Problem Solving with Systemtap," dans *Proceedings of the Ottawa Linux Symposium*, 2006.

[24] "QNX Momentics." http://www.qnx.com. Vérifié le 2009/01/05.

[25] J. Axboe et A. D. Brunelle, "blktrace user guide." http ://www.kernel.org/pub/linux/kernel/people/axboe/blktrace/.

[26] S. Rostedt, "ftrace." `http://lwn.net/Articles/290277/`. Vérifié le 2009/01/05.

[27] T. Ts'o, "Debugfs(8) Linux man page," 2008.

[28] Intel Corp., *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[29] W. E. Nagel, A. Arnold, M. Weber, et K. Solchenbach, "Vampir : Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, p. 69–80, 1996.

[30] P. Miller Barton, D. Callaghan Mark, M. Cargille Jonathan, *et al.*, "The Paradyn Parallel Performance Measurement Tool," *IEEE Computer*, vol. 28, no. 11, p. 37–46, 1995.

[31] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, et A. Vahdat, "Pip : Detecting the unexpected in distributed systems," dans *Symposium on Networked Systems Design and Implementation*, p. 115–128, 2006.

[32] Y. Endo et M. Seltzer, "Improving interactive performance using TIPME," *SIGMETRICS Perform. Eval. Rev.*, vol. 28, no. 1, p. 240–251, 2000.

[33] P. Barham, A. Donnelly, R. Isaacs, et R. Mortier, "Using Magpie for request extraction and workload modelling," dans *Symposium on Operating Systems Design and Implementation*, p. 259–272, 2004.

[34] Microsoft Corp., "Event Tracing for Windows (ETW)," 2002. `http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp`. Vérifié le 2009/01/05.

[35] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, et D. Narayanan, "Request extraction in magpie : events, schemas and temporal joins," dans *EW11 : Proceedings of the 11th workshop on ACM SIGOPS European workshop*, (New York, NY, USA), p. 17, ACM, 2004.

[36] P. Barham, R. Isaacs, R. Mortier, et D. Narayanan, "Magpie : Online modelling and performance-aware systems," dans *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[37] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, et A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," dans *Proceedings of the nineteenth ACM symposium on Operating systems principles*, p. 74–89, ACM New York, NY, USA, 2003.

[38] M. Desnoyers et M. R. Dagenais, "LTTng : Tracing across execution layers, from the hypervisor to user-space," dans *Linux Symposium*, 2008.

[39] "tbench." `http://samba.org/ftp/tridge/dbench/README`. Vérifié le 2009/01/05.

[40] "Tracing Wiki." `http://ltt.polymtl.ca/tracingwiki`. Vérifié le 2009/01/05.

[41] "MediaWiki." `http://www.mediawiki.org`. Vérifié le 2009/01/05.

[42] "memcached." `http://www.danga.com/memcached/`. Vérifié le 2009/01/05.

[43] "Apache." `http://httpd.apache.org`. Vérifié le 2009/01/05.

[44] "MySQL." `http://www.mysql.org`. Vérifié le 2009/01/05.

[45] "GNU wget." `http://www.gnu.org/software/wget/`. Vérifié le 2009/01/05.

[46] "Xorg." `http://www.x.org`. Vérifié le 2009/01/05.

[47] "K Desktop Environment (KDE)." `http://www.kde.org`. Vérifié le 2009/01/05.

[48] É. Clément, "Synchronisation de traces dans un réseau distribué," mémoire de maîtrise, École Polytechnique de Montréal, 2006.