View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

ANALYSE DE PERFORMANCE DE SYSTÈMES DISTRIBUÉS ET HÉTÉROGÈNES À L'AIDE DE TRAÇAGE NOYAU

FRANCIS GIRALDEAU
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
NOVEMBRE 2015

© Francis Giraldeau, 2015.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ANALYSE DE PERFORMANCE DE SYSTÈMES DISTRIBUÉS ET HÉTÉROGÈNES À L'AIDE DE TRAÇAGE NOYAU

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

M. ANTONIOL Giuliano, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. PIERRE Samuel, Ph. D., membre
M. FRANKS R. Gregory, Ph. D., membre

DÉDICACE

À Dominic Gauthier,
grâce à qui tout est possible.

REMERCIEMENTS

Merci tout d'abord au professeur Michel Dagenais de m'avoir donné cette occasion unique d'approfondir mes connaissances dans le domaine des systèmes d'exploitation. Je tiens à souligner Benoît des Ligneris pour les projets que nous avons faits ensemble et qui ont été l'élément déclencheur de cette recherche. Merci au Centre National de Recherche du Canada, Ericsson et EfficiOS pour leur soutien financier, et notamment Lothar Wengerek pour sa supervision lors d'un stage au siège social d'Ericsson à Stockholm. Merci également aux collègues actuels et anciens du laboratoire DORSAL, pour l'entraide et les échanges d'idées. En particulier, je tiens à souligner la collaboration spéciale de Mathieu Desnoyers, Julien Desfossez, Yannick Brosseau, Geneviève Bastien, Matthew Khouzam, Alexandre Montplaisir et Suchakrapani Sharma. Merci également aux étudiants Raphaël Beamonte et Mohamad Gebai, avec qui j'ai eu la chance de réaliser des articles connexes. Un merci également à Philippe Doucet-Beaupré pour la révision de tous mes articles. Je tiens également à remercier Linus Torvalds pour Linux, sans qui il serait très difficile d'étudier les systèmes d'exploitation. Merci finalement à ma famille et mes amis pour leur aide et leur appui qui a été essentiel à la réussite de ce projet.

RÉSUMÉ

Les systèmes infonuagiques sont en utilisation croissante. La complexité de ces systèmes provient du fait qu'ils s'exécutent de manière distribuée sur des architectures multicoeurs. Cette composition de services est souvent hétérogène, c.-à-d. qui implique différentes technologies, librairies et environnements de programmation. L'interopérabilité est assurée plutôt par l'utilisation de protocoles ouverts. L'espace de configuration résultant croît de manière exponentielle avec le nombre de paramètres, et change continuellement en fonction des nouveaux besoins et de l'adaptation de la capacité. Lorsqu'un problème de performance survient, il doit être possible d'identifier rapidement la cause pour y remédier. Or, ce problème peut être intermittent et difficile à reproduire, et dont la cause peut être une interaction transitoire entre des tâches ou des ressources. Les outils utilisés actuellement pour le diagnostic des problèmes de performance comprennent les métriques d'utilisation des ressources, les outils de profilage, la surveillance du réseau, des outils de traçage, des débogueurs interactifs et les journaux systèmes. Or, chaque composant doit être analysé séparément, ou l'utilisateur doit corréler manuellement cette information pour tenter de déduire la cause du problème. L'observation globale de l'exécution de systèmes distribués est un enjeu majeur pour en maitriser la complexité et régler les problèmes efficacement.

L'objectif principal de cette recherche est d'obtenir un outil d'analyse permettant de comprendre la performance d'ensemble d'une application distribuée. Ce type d'analyse existe au niveau applicatif, mais elles sont spécifiques à un environnement d'exécution ou un domaine particulier. Nos travaux se distinguent par l'utilisation d'une trace noyau, qui procure un niveau plus abstrait de l'exécution d'un programme, et qui est indépendant du langage ou des librairies utilisées.

La présente recherche vise à déterminer si la sémantique des évènements du système d'exploitation peut servir à une analyse satisfaisante. Le surcout du traçage est un enjeu important, car il doit demeurer faible pour ne pas perturber le système et être utile en pratique.

Nous proposons un nouvel algorithme permettant de retrouver les relations d'attente entre les tâches et les périphériques d'un ordinateur local. Nous avons établi que le chemin critique exact d'une application nécessite des évènements qui ne sont pas visibles depuis le système d'exploitation. Nous proposons donc une approximation du chemin critique, dénomée *chemin actif d'exécution*, où chaque attente est remplacée par sa cause racine.

Les approches antérieures reposent sur l'analyse des appels système. L'analyse doit tenir en compte la sémantique de centaines d'appels système, ce qui n'est pas possible dans le cas

général, car le fonctionnement d'un appel système dépend de l'état du système au moment de son exécution. Par exemple, le comportement de l'appel système `read()` est complètement différent si le fichier réside sur un disque local ou sur un serveur de fichier distant. L'appel système `ioctl()` est particulièrement problématique, car son comportement est défini par le programmeur. Le traçage des appels système contribue aussi à augmenter le surcout, alors qu'une faible proportion d'entre eux modifie le flot de l'exécution. Les tâches d'arrière-plan du noyau n'effectuent pas d'appels système et ne peuvent pas être prises en compte par cette méthode. À cause de ces propriétés, l'analyse basée sur des appels système est fortement limitée.

Notre approche remplace les appels système par des évènements de l'ordonnanceur et des interruptions. Ces évènements de plus bas niveau sont indépendants de la sémantique des appels système et prennent en compte les tâches noyau. Le traçage des appels système est donc optionnel, ce qui contribue à réduire le surcout et simplifie drastiquement l'analyse. Les bancs d'essais réalisés avec des logiciels commerciaux populaires indiquent qu'environ 90% du surcout est lié aux évènements d'ordonnancement. En produisant des cycles d'ordonnancement à la fréquence maximale du système, il a été établi que le surcout moyen au pire cas est de seulement 11%. Nous avons aussi réalisé une interface graphique interactive montrant les résultats de l'analyse. Grâce à cet outil, il a été possible d'identifier avec succès plusieurs problèmes de performance et de synchronisation. Le fonctionnement interne et l'architecture du programme sont exposés par l'outil de visualisation, qui se révèle utile pour effectuer la rétro-ingénierie d'un système complexe.

Dans un second temps, la dimension distribuée du problème a été ajoutée. L'algorithme de base a été étendu pour supporter l'attente indirecte pour un évènement distant, tout en préservant ses propriétés antérieures. Le même algorithme peut donc servir pour des processus locaux ou distants. Nous avons instrumenté le noyau de manière à pouvoir faire correspondre les paquets TCP/IP émis et reçus entre les machines impliqués dans le traitement à observer. L'algorithme tient en compte que la réception ou l'émission de paquets peut se produire de manière asynchrone. Les traces obtenues sur plusieurs systèmes n'ont pas une base de temps commune, car chacun possède sa propre horloge. Aux fins de l'analyse, toutes les traces doivent être synchronisées, et les échanges apparaitre dans l'ordre de causalité. Pour cette raison, les traces doivent être préalablement synchronisées. L'algorithme a été utilisé pour explorer le comportement de différentes architectures logicielles. Différentes conditions d'opérations ont été simulées (délais réseau, durée de traitement, retransmission, etc.) afin de valider le comportement et la robustesse de la technique. Il a été vérifié que le résultat obtenu sur une grappe d'ordinateurs est le même que celui obtenu lorsque les services s'exécutent dans des machines virtuelles. Le surcout moyen nécessaire pour tracer une requête

Web s'établit à 5%. La borne supérieure du surcout pour des requêtes distantes est d'environ 18%. Pour compléter l'analyse, nous avons réalisé des cas d'utilisation impliquant six environnements logiciel et domaines différents, dont une application Web Django, un serveur de calcul Java-RMI, un système de fichier distribué CIFS, un service Erlang et un calcul parallèle MPI. Comme contribution secondaire, nous avons proposé deux améliorations à l'algorithme de synchronisation. La première consiste en une étape de présynchronisation qui réduit considérablement la consommation maximale de mémoire. La deuxième amélioration concerne la performance de la fonction de transformation du temps. Le temps est représenté en nanosecondes et le taux de variation à appliquer doit être très précis. L'utilisation de l'arithmétique à point flottant de précision double n'est pas assez précis et produit des inversions d'évènements. Un couteux calcul à haute précision est requis. Grâce à une simple factorisation de l'équation linéaire, la plupart des calculs à haute précision ont été remplacés par une arithmétique entière 64-bit. Les bancs d'essai ont mesuré que cette optimisation procure une accélération de 65 fois en moyenne et que la précision du résultat n'est pas affectée.

Le troisième thème de la recherche porte sur le profilage des segments du chemin d'exécution. L'échantillonnage des compteurs de performance matériel permet le profilage du code natif avec un faible surcout. Une limitation concerne le code interprété qui peut se retrouver dans une application hétérogène. Dans ce cas, le code profilé est celui de l'interpréteur, et le lien avec les sources du programme est perdu. Nous avons conçu une technique permettant de transférer à un interpréteur l'évènement de débordement du compteur de performance, provenant d'une interruption non masquable du processeur. L'analyse de l'état de l'interpréteur peut être effectuée en espace utilisateur. Un module d'analyse pour Python a été développé. Nous avons comparé le cout des méthodes pour obtenir la pile d'appel de l'interpréteur Python et celle du code interprété. Ces données sont sauvegardées par l'entremise de LTTng-UST, dont la source de temps est cohérente avec les traces produites en mode noyau, ce qui permet d'associer les échantillons produits avec le chemin d'exécution. Nous avons validé le profil à l'aide d'une application d'étalonnage. Nous avons mesuré une erreur inférieure à 1%, et ce résultat est équivalent à celui produit par un profileur déterministe. La période d'échantillonnage est établie selon un compromis entre le surcout et la résolution de l'échantillonnage. Nos tests indiquent que, pour un chemin d'exécution de 50 ms, une plage de taux d'échantillonnage existe et satisfait à la fois une marge d'erreur inférieure à 5% et un surcout de moins de 10%.

ABSTRACT

Cloud systems are increasingly used. These systems have a complex behavior, because they run on a cluster of multi-core computers. This composition of services is often heterogeneous, involving different technologies, libraries and programming environments. Interoperability is ensured using open protocols rather than standardizing runtime environments. The resulting configuration space grows exponentially with the number of parameters, and constantly changes in response to new needs and capacity adaptation. When a performance problem arises, it should be possible to quickly identify the cause in order to address it. However, performance problems can be intermittent and difficult to reproduce, and their cause can be a transient interaction between tasks or resources. The tools currently used to diagnose performance problems include resource utilization metrics, profiling tools, network monitoring, layout tools, interactive debuggers and system logs. However, each component must be analyzed separately, or the user must manually correlate that information to try deducing the root cause. Observing the performance of globally distributed systems is a major challenge, to master their complexity and solve problems effectively.

The main objective of this research is to obtain an analysis tool for understanding the overall performance of a distributed application. This type of analysis exists at the application level, but they are specific to a runtime environment or a particular application domain. To address this issue, we propose to use kernel tracing, which provides a more abstract information about the execution of a program and is independent of the language or the libraries used.

This research aims to determine whether the semantics of the operating system events are effective for performance analysis of such systems. The additional cost of tracing is an important issue because it must remain low, to avoid disturbing the system and be useful in practice.

We propose a new algorithm to find the waiting relationships between tasks and devices on a local computer. We established that the exact critical path of an application requires events which are not visible from the operating system. We therefore propose an approximation of the critical path, that we named execution path. Previous approaches rely on system call analysis. However, the analysis must take into account the semantics of hundreds of system calls. Tracing all system calls increases the overhead, while most system calls do not change the flow of execution. Furthermore, the background kernel threads do not perform system calls and are not taken into account. Our approach relies instead on lower-level events, namely from the scheduler and the interruptions. These events are independent of

the semantics of system calls and take into account the kernel threads. Tracing system calls is optional, which helps reduce the overhead and simplifies the analysis. The benchmarks made with popular commercial software indicate that about 90% of the overhead is related to scheduling events. By producing scheduling cycles at the maximum frequency, we established that the average worst case overhead is only 11%. Finally, we implemented an interactive graphical view showing the results of the analysis. With this tool, it was possible to identify quickly several performance and synchronization problems in actual applications. The tool also exposes the internal functioning and architecture of the program, which is useful for performing reverse engineering of a complex system.

Secondly, we addressed the distributed dimension of the problem. The basic algorithm has been extended to support indirect network wait, while preserving its previous properties. The same algorithm can therefore be used for local or remote processes. We instrumented the kernel for matching the TCP/IP packets sent and received between machines involved in the processing. The algorithm takes into account the fact that reception and transmission of packets can occur asynchronously. The traces obtained on several systems do not have a common time base, as each has its own clock. The analysis requires that all traces have the same time reference and exchanges must appear in the causal order. For this reason, the traces must first be synchronized. The algorithm was used to explore the behavior of different software architectures. We simulated various operating conditions (network delays, processing delays, retransmission, etc.) to validate the behavior and robustness of the technique. We verified that the result on a cluster of physical computers is the same as the one obtained when the services are running inside virtual machines. The average overhead to trace Web requests is about 5%. The worst case overhead measured with the higest frequency remote procedure call (empty remote call) is approximately 18%. To complete the analysis, we implemented use cases and software environments involving six different application domains, including a Django Web application, a Java-RMI server, a CIFS distributed file system, an Erlang service and a MPI parallel computation. As a secondary contribution, we proposed two improvements to the synchronization algorithm. The first is a pre-synchronization step that dramatically reduces the maximum memory consumption. The second improvement concerns the performance of the time transformation function. The time is represented in nanoseconds and the rate of change to apply must be very precise. The use of double precision floating point arithmetic is not accurate enough and produces event inversions. Expensive high-precision calculation is required. We replaced most of high-precision calculations by integer arithmetic of native register size, providing an average acceleration of approximately 65 times for the synchronization.

The third area of research focuses on profiling the execution path segments. Sampling hardware performance counters allows efficient profiling of native code. One limitation concerns the interpreted code that may be found in an heterogeneous application. In this case, the native code running is the interpreter itself, and the link with the actual sources of the interpreted program is lost. We developed a technique to transfer to an interpreter the performance counter overflow event from the non-maskable interrupt of the processor. The analysis of the interpreter state can then be performed in user-space. To demonstrate the feasability of the approach, we implemented the analysis module for Python. We compared the cost of methods to get the call stack of the Python interpreter and the interpreted code. This data is saved through LTTng-UST, which has a time source consistent with the kernel mode trace and allows the association of the samples produced with the execution path. We validated the profile using a calibrated program. We measured less than 1% profile error, and this result is equivalent to the error rate of a deterministic profiler. The sampling period is a compromise between the overhead and the profile resolution. Our tests indicate that, for an execution path of 50 ms, a range of sampling exists that satisfies both a margin of error lower than 5% and an overhead of less than 10%.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT viii

TABLE DES MATIÈRES xi

LISTE DES TABLEAUX xiv

LISTE DES FIGURES xv

LISTE DES SIGLES ET ABRÉVIATIONS xvii

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.2 Éléments de la problématique 3
1.3 Objectifs de recherche 4
1.4 Plan 5

CHAPITRE 2 REVUE DE LITTÉRATURE 6
2.1 Travaux du laboratoire DORSAL 6
2.1.1 Méthodes de corrélation statistique des paquets réseaux 8
2.2 Méthodes précises d'analyse de l'exécution distribuée 8
2.3 Calcul du chemin critique 13
2.4 Détection de goulots d'étranglement 14
2.5 Traceurs 15
2.6 Synthèse 16

CHAPITRE 3 MÉTHODOLOGIE 18
3.1 Étude du système 18
3.2 Mesures du cout de l'instrumentation 19
3.3 Validation 20
3.4 Environnement 21

CHAPITRE 4 ARTICLE 1 : Approximation of Critical Path Using Low-level System Events . . . 22
4.1 Abstract . . . 22
4.2 Introduction . . . 22
4.2.1 Related Work . . . 23
4.3 Execution model . . . 25
4.3.1 Critical path of execution . . . 26
4.3.2 Critical path approximation . . . 29
4.4 Empirical results . . . 32
4.4.1 Approximation algorithm . . . 34
4.5 Evaluation . . . 37
4.5.1 Environment . . . 38
4.5.2 Uneven Parallel Computation . . . 38
4.5.3 Block Device I/O . . . 39
4.5.4 Use Case . . . 40
4.5.5 Tracing Cost . . . 41
4.6 Future Work . . . 44
4.7 Conclusion . . . 44
4.8 Acknowledgements . . . 45

CHAPITRE 5 ARTICLE 2 : Host-based method to recover wait causes in distributed systems . . . 46
5.1 Abstract . . . 46
5.2 Introduction . . . 46
5.3 Analysis Architecture . . . 48
5.3.1 Trace Synchronization . . . 50
5.3.2 Trace Analysis . . . 51
5.4 Evaluation . . . 54
5.4.1 Effect of Host Type . . . 56
5.4.2 Effect of Network Conditions . . . 58
5.4.3 Effect of Asynchronous Processing . . . 59
5.4.4 Use-cases . . . 61
5.4.5 Analysis Cost . . . 66
5.4.6 Analysis Optimizations . . . 71
5.5 Future Work . . . 72
5.6 Related Work . . . 73

5.7 Conclusion 74

CHAPITRE 6 ARTICLE 3 : Execution path profiling using hardware performance counters 76
6.1 Abstract 76
6.2 Introduction 76
6.3 Related work 79
6.4 Architecture 81
6.5 Evaluation 84
6.5.1 Factoring out monitoring cost 84
6.5.2 Accuracy 85
6.5.3 Signal cost 85
6.5.4 Monitoring cost 86
6.5.5 Profiling overhead 87
6.5.6 Sampling resolution 89
6.6 Discussion 90
6.6.1 Unix signals limitations 90
6.6.2 Performance counter scope 92
6.7 Future work 92
6.8 Conclusion 93

CHAPITRE 7 DISCUSSION GÉNÉRALE 94
7.1 Atteinte des objectifs 94
7.2 Retombées connexes 96

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS 98
8.1 Limitations de la solution proposée 98
8.2 Améliorations futures 99

LISTE DES TABLEAUX

Tableau 2.1 Concepts et mots clés de recherche . . . 7
Tableau 3.1 Suite d'outils de `workload-kit` . . . 20
Table 4.1 Critical paths of basic execution graphs . . . 31
Table 4.2 Required kernel tracepoints . . . 34
Table 4.3 Percentage of execution time spent in each task on the critical path for the program `wk-imbalance` . . . 38
Table 4.4 Critical task flow of `wk-ioburst` . . . 39
Table 4.5 Results of sysbench tracing experiments, with and without tracing of system call events, and with a single internal disk (Int.) or with an added external disk (Ext.) . . . 45
Table 5.1 Kernel events required for the analysis . . . 51
Table 6.1 Profile measurement accuracy . . . 86
Table 6.2 Monitoring cost . . . 88
Table 6.3 Average profiling overhead for interpreter unwinding, interpreted code traceback and both monitoring combined, according to the sampling period of the cycle counter. . . . 90

LISTE DES FIGURES

Figure 4.1 Task state finite state machine 26
Figure 4.2 Basic execution graphs 30
Figure 4.3 Example of wake-up event from sub-task 36
Figure 4.4 Example of wake-up event in timer interrupt 36
Figure 4.5 Execution of `wk-imbalance` 39
Figure 4.6 Time line view of the critical task flow of `wk-ioburst` 40
Figure 4.7 Overview of the `APT` critical task flow 42
Figure 4.8 Call chain of mandb locating inefficient process execution 42
Figure 5.1 Task state and TCP packet transmission 47
Figure 5.2 Example of a synchronization graph between three hosts. 52
Figure 5.3 Resulting timestamp transforms according to the reference host for the synchronization graph example of Figure 5.2. 52
Figure 5.4 Active path of `wk-rpc` according to the network latency. The execution in (a) has natural network latency, in (b) the latency is set to 10 ms and in (c), the latency is set to 100 ms. Green intervals repesent CPU usage and pink intervals and edges represent network latency. 60
Figure 5.5 Active path of `wk-rpc` according to asynchronous processing level of 0% in (a), 50% in (b) and 100% in (c), and asynchronous processing based on event loop in (d). 62
Figure 5.6 Example of Java RMI compute engine execution. 62
Figure 5.7 Example of a CIFS remote directory listing. 63
Figure 5.8 Execution of the post HTTP request across the client, the web server and the database. 64
Figure 5.9 Execution of the echo Erlang example. 65
Figure 5.10 MPI imbalanced computation execution. Below: the user-space trace displays the running (green) and waiting (red) of each thread. Above: the kernel trace corresponding to the first MPI thread. The top interval shows a zoomed in view of the active wait section. The first cycle of the computation is highlighted. 66
Figure 5.11 Effect of tracing on request latency density for the wk-rpc benchmark. Tracing causes the distribution to shift to the right. 67
Figure 5.12 Relative event frequency according to workload. Interrupts are the most frequent event type for every workload. 69

Figure 5.13 Analysis time according to the number of traced web requests. Both graph construction and active path extraction have linear scalability. 70
Figure 6.1 This task execution path example shows the behavior a distributed and heterogeneous Web application according to time. The client, Apache HTTP server and the PostgreSQL database are running on different computers and communicate using TCP/IP. Green intervals represents computation, purple is disk I/O and edges are communication between tasks. 80
Figure 6.2 Performance counter creation and configuration. 82
Figure 6.3 Processing sequence of counter overflow event. An NMI is raised on counter overflow. The NMI handler raises a local APIC IRQ, that setup the signal stack and make sure the task is in the run queue. On return from the kernel space, the signal handler is executed, which contains arbitrary monitoring routine. Finally, `sigreturn()` removes the signal stack frame and the application resumes. 83
Figure 6.4 Monitoring cost according to call stack depth, for unwind and traceback. 88
Figure 6.5 The overhead is reduced by increasing the sampling period, which increase the margin of error. 91

LISTE DES SIGLES ET ABRÉVIATIONS

ABI	Application Binary Interface
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
APT	Advanced Packaging Tool
CFG	Control Flow Graph
CIFS	Common Internet File System
CPI	Cycle Per Instruction
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DNS	Domain Name Service
ELF	Executable and Linkable Format
ETW	Event Tracing for Windows
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter-Process Communication
IPC	Instruction Per Cycle
IPI	Inter-Processor Interrupt
IP	Internet Protocol
IRQ	Interrupt Request
JIT	Just In Time
JVM	Java Virtual Machine
KVM	Kernel Virtual Machine
LTT	Linux Tracing Toolkit
LQN	Layered Queuing Network
ME	Margin of Error
MPI	Message Passing Interface
NMI	Non-Maskable Interrupt
NTP	Network Time Protocol
PAG	Program Activity Graph
PMU	Performance Management Unit
RAM	Random Access Memory
RCP	Rich Client Platform

RCU	Read-Copy Update
RMI	Remote Method Invocation
RMS	Root Mean Square
RPC	Remote Procedure Call
SMP	Symmetric multiprocessing
SQL	Structured Query Language
SSD	Solid State Drive
TCP	Transmission Control Protocol
TLA	Three Letter Acronym
TMF	Tracing and Monitoring Framework
TSC	Time Stamp Counter
UDP	User Datagram Protocol
ULI	User-Level Interrupt
UST	User-Space Tracing
VMM	Virtual Machine Monitor
WSGI	Web Server Gateway Interface

CHAPITRE 1 INTRODUCTION

Cette thèse aborde l'analyse de performance de systèmes distribués et hétérogènes. Un système distribué est défini comme un ensemble d'ordinateurs indépendants qui apparaissent à l'utilisateur comme un seul système cohérent [1]. Plus précisément, notre étude porte sur le comportement des tâches, s'exécutant sur un ensemble d'ordinateurs et qui communiquent et se synchronisent pour réaliser leur traitement. La propriété hétérogène des systèmes étudiés fait référence à la diversité des logiciels impliqués, soit des langages de programmation et leur environnement, des librairies et des intergiciels. L'utilisation de standard ouvert de communication permet l'interopérabilité et la substitution de composants de manière transparente. Notre objectif général est développer une méthode permettant d'analyser efficacement la performance de ce type de système.

1.1 Définitions et concepts de base

Une latence faible est un critère primordial pour la satisfaction de l'utilisateur de systèmes informatiques [2], [3]. Or, l'augmentation de la vitesse des processeurs depuis plus d'une décennie évolue à un rythme moins rapide [4]. Il est plus simple aujourd'hui d'accroitre le débit, grâce notamment aux architectures multicoeurs, que de réduire la portion série du traitement. Comprendre l'origine de la latente d'un traitement est donc primordial pour assurer un niveau de performance optimal.

Or, les outils actuellement disponibles pour comprendre le comportement de systèmes distribués présentent tous des limitations. Les outils de profilage ne se concentrent que sur les instructions du programme, alors que le temps d'attente n'est pas pris en considération. Par exemple, le ratio d'instructions par cycle (IPC) du processus `sleep` est évidemment insuffisant pour comprendre le délai d'exécution. Ce ratio est calculé lorsque l'application s'exécute, et ne prend pas en compte l'attente de l'expiration du minuteur. Un phénomène similaire se produit en cas de contention pour l'accès à une ressource, comme un processeur.

Les outils de monitorage qui rapportent le taux d'utilisation moyen des ressources, comme le processeur, la mémoire, le disque et le réseau, sont pour la plupart basés sur la scrutation périodique des statistiques systèmes. La fréquence de cette scrutation est habituellement très faible, telle que 1 Hz, de manière à limiter la perturbation du système. Au mieux, ces données donnent une vague idée de l'état général du système. Évidemment, cette catégorie d'outil n'est pas adaptée pour identifier précisément la cause d'un délai anormal de l'ordre

de la microseconde, car la moyenne absorbe les cas spéciaux. Ces données sont également locales à un ordinateur, et la corrélation temporelle des séries de données ne peut être que grossière.

Un débogueur interactif est idéal pour trouver les problèmes de logique dans une application, mais celui-ci change les propriétés temporelles de l'exécution, ce qui peut le rendre inopérant pour des applications distribuées. Par exemple, un client est habituellement configuré pour échouer après l'expiration d'un certain temps si aucune réponse n'est reçue, ce qui peut se produire pendant une session interactive pour inspecter l'état du serveur. Aussi, certains problèmes ne sont pas causés par l'application elle-même, mais plutôt par son interaction *in situ* avec les autres composants du système, et ce d'une manière non déterministe.

L'attente entre les composants du système est l'élément clé de la recherche. Un processus peut attendre pour le traitement d'un périphérique, pour l'accès à une ressource déjà occupée, ou une autre tâche du système, qu'elle soit locale ou distante. En général, les systèmes distribués utilisent une attente passive pour une raison d'efficience. Une attente active, ou par scrutation, serait un gaspillage de ressources, alors que l'attente pour un paquet réseau peut être plusieurs ordres de grandeur plus lent qu'un simple changement de contexte. L'attente passive libère le processeur pour d'autres tâches et contribue à réduire la consommation d'énergie. Lorsque l'attente se termine, la tâche est remise dans la file d'exécution de l'ordonnanceur, qui peut ensuite la remettre en fonctionnement. La provenance du signal, remettant en fonction une tâche en attente, indique la cause réelle de l'attente.

Comprendre les interactions exactes entre les composants du système nécessite une approche par traçage. Cette technique consiste à enregistrer le temps, le type d'évènement et des données optionnelles, ce qui forme une séquence ordonnée. La trace peut ensuite être analysée pour en déterminer certaines propriétés.

Le projet *Linux Tracing Toolkit next generation* (LTTng) représente l'état de l'art dans le domaine du traçage et est l'outil de base que nous utilisons. Il est le fruit de nombreuses années de recherche et de développement de la part du laboratoire DORSAL et de ses collaborateurs. Il inclut le traçage en mode noyau et utilisateur, des temps monotoniques d'une précision de la nanoseconde, des tampons circulaires sans verrou, efficients pour une utilisation portable sur des architectures multicoeur, et un format de trace universel [5]. Notre but n'est pas d'optimiser directement le traceur lui-même, mais plutôt d'exploiter l'information qu'il est possible de tracer. L'instrumentation additionnelle nécessaire pour les besoins du projet tire avantage de cette infrastructure de traçage sophistiquée et performante. Il est clair que l'analyse manuelle des évènements a atteint une limite, et que des abstractions sont requises pour exploiter efficacement la quantité astronomique d'information produite.

Nos analyses sont développées dans le cadre applicatif de Trace Compass du projet Eclipse, qui est la continuité du *Tracing and Modeling Framework* (TMF). Cet environnement graphique permet de naviguer de manière interactive dans une représentation visuelle du résultat de notre analyse, et ainsi en faciliter l'interprétation. En ce sens, le projet ne se limite pas à produire une analyse, mais s'intéresse aussi aux outils pour la rendre accessible et exploitable facilement.

1.2 Éléments de la problématique

Déterminer les portions de l'exécution contribuant au temps de fin se rapporte à l'analyse du chemin critique. Cette tâche consiste essentiellement à trouver le chemin le plus long dans un graphe dirigé acyclique (en anglais *Directed Acyclic Graph* ou DAG), problème pour lequel les algorithmes sont établis depuis fort longtemps [6]. Un tel chemin peut être trouvé par une adaptation mineure à l'algorithme classique du chemin le plus court dans un DAG dont les arcs ont tous une longueur positive de Dijkstra [7].

D'autre part, une trace noyau concerne les évènements se produisant depuis le système d'exploitation. La sémantique des évènements se rapporte, entres autres, à l'ordonnanceur, aux appels système et aux interruptions. Ces évènements sont exactement ceux qui pourraient permettre d'étudier l'attente passive d'une application distribuée, de manière indépendante de l'environnement d'exécution. Or, nous ne savons pas s'il est possible de construire un modèle, tel qu'un DAG, à partir de ces évènements, puis d'utiliser les algorithmes connus sur cette structure. Il se peut que l'information requise ne soit pas accessible, inexploitable ou trop couteuse à obtenir en pratique. Même si un tel modèle existe, il est nécessaire de démontrer de manière expérimentale si l'abstraction produite est utile pour étudier la performance de logiciels existants et d'en explorer les cas limites.

Dans le cadre du projet, nous utilisons Linux comme système d'exploitation de référence, car son code source est libre et ouvert. Sans la possibilité d'étudier et d'instrumenter le système d'exploitation, cette recherche serait impossible. À cet effet, toute modification requise pour les analyses est théoriquement possible, mais ces changements ne seront pas forcément acceptés par le projet en amont. Le fait de devoir modifier les sources du noyau puis de le compiler représente une barrière majeure qui augmente en pratique la difficulté de déploiement de l'instrumentation sur des systèmes existants. Pour les composants optionnels, le noyau Linux comprend un mécanisme permettant de charger dynamiquement des modules. Or, nous ne savons pas si les modifications requises aux fins d'instrumentation peuvent se faire par l'entremise de l'API limité exposé par le noyau Linux. Rendre disponible l'ensemble de l'instrumentation sous forme de modules optionnels, compatibles avec la plupart des versions en

cours d'utilisation, constitue donc un défi technologique considérable pour son utilisation en pratique.

Le traçage permet d'enregistrer la dynamique du système mais, comme toute méthode de mesure, le traçage altère le comportement du système lui-même. Le surcout du traçage est un élément important à considérer, tout comme la latence additionnelle et les autres perturbations impliquées. Cette incertitude est un enjeu de recherche. Le surcout moyen est obtenu par des mesures sur des logiciels existants et, lorsqu'applicable, le surcout au pire cas est mesuré.

Une autre incertitude majeure du projet concerne la base de temps commune requise pour effectuer une analyse globale du système. Or, chaque ordinateur possède sa propre horloge, dont la fréquence de l'oscillateur varie de manière non linéaire en fonction de différents facteurs, comme la température et la tension de fonctionnement. La source de temps abstrait le *Time Stamp Counter* (TSC) avec une précision d'une nanoseconde et garantit la propriété monotonique des lectures du temps. Il se pourrait que la synchronisation des traces ne soit pas assez précise en pratique, ce qui empêcherait de consolider, sur une référence de temps commune, les traces obtenues sur plusieurs ordinateurs simultanément.

1.3 Objectifs de recherche

L'objectif général de cette recherche est de fournir des algorithmes et des outils d'analyse de traces qui permettent à des administrateurs système et des programmeurs de comprendre les performances de l'ensemble de l'exécution d'une application distribuée et hétérogène.

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

1. Développer l'instrumentation noyau servant d'entrée à la construction d'un modèle de l'exécution d'une application distribuée.
2. Extraire le chemin d'exécution d'une application distribuée.
3. Calculer les ressources utilisées dans chaque composant pour une exécution.
4. Relier la trace noyau obtenue au code source du programme.
5. Vérifier le fonctionnement de l'analyse pour un large éventail de configurations.
6. Mesurer l'impact en fonctionnement du traçage des événements requis par ces analyses.

La disponibilité d'outils basés sur les résultats de cette recherche améliorerait radicalement la manière dont les développeurs et analystes évaluent et trouvent les problèmes de performance sur des systèmes distribués et, en ce sens, représenterait une contribution significative. À

notre connaissance, aucun autre outil d'analyse ne permet de produire les résultats désirés en respectant les contraintes de la problématique.

1.4 Plan

La revue de la littérature est présentée au Chapitre 2. On y présente les travaux antérieurs relatifs à notre sujet de recherche. Le Chapitre 3 porte sur la méthodologie de la recherche. Les programmes développés pour supporter la recherche sont présentés, et incluent des logiciels de validation et de mesure de performance. Nous détaillons aussi l'environnement d'analyse dans lequel notre projet s'inscrit.

Les trois articles scientifiques issus de la recherche suivent successivement. L'article « Approximation of Critical Path Using Low-level System Events » au Chapitre 4 présente les principes fondamentaux de l'analyse. Cet article démontre les limitations inhérentes à l'analyse du chemin critique uniquement à l'aide de l'instrumentation noyau. On y discute d'une approximation du chemin critique, le chemin d'exécution, donnant des résultats utiles en pratique pour l'analyse de la performance. Cet article a été soumis au journal *Operating System Review* de l'*Association for Computing Machinery*. Le second article, au Chapitre 5, porte sur l'aspect distribué de la recherche. Le titre est « Host-based method to recover wait causes in distributed systems » et a été soumis au journal *IEEE Transactions on Parallel and Distributed Systems*. Le troisième article, au Chapitre 6, étudie les méthodes d'échantillonnage pour faire le lien entre le chemin d'exécution distribué et le code exécuté, dont le titre est « Execution path profiling using hardware performance counters ». Cet article a été soumis au journal *Software : Practice and Experience*.

La thèse se termine avec la discussion générale et la conclusion, aux Chapitres 7 et 8 respectivement.

CHAPITRE 2 REVUE DE LITTÉRATURE

D'après le contexte et les spécifications établies, cette section décrit l'état de l'art dans ce domaine, afin d'exposer les bases sur lesquelles repose cette recherche et démontrer l'originalité de la contribution.

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

Les concepts et les mots clés utilisés pour la revue de littérature sont présentés au Tableau 2.1. Une première requête a été réalisée en combinant ces concepts. Les articles pertinents retenus après cette première recherche ont été utilisés par compléter la recherche d'après les articles en référence, afin de s'assurer de couvrir l'ensemble des travaux du domaine. La recherche cible principalement les 10 dernières années, sans toutefois s'y limiter.

Au moment de rédiger cette thèse, l'inventaire bibliographique comprend 424 articles et références. Nous constatons qu'il s'agit d'un domaine de recherche actif de par le volume considérable de contributions récentes. Malgré cette recherche extensive, nous n'avons pas trouvé de travaux qui se penchent directement sur la problématique décrite en introduction, de la manière dont nous le proposons.

La revue commence par la présentation des travaux antérieurs du laboratoire DORSAL à la Section 2.1. Cette section positionne la présente proposition dans le cadre des travaux du groupe de recherche. Ensuite, il est question des deux catégories principales d'analyse distribuée de systèmes en boite noire, soit les méthodes basées sur la corrélation statistique des paquets réseau et les méthodes précises d'analyse de l'exécution distribuée [8]. Ces deux catégories sont expliquées et comparées aux sections 2.1.1 et 2.2 respectivement. La section 2.3 recense les techniques pour calculer de chemin critique d'exécution, un algorithme essentiel à notre étude. La revue se termine par l'inventaire des traceurs systèmes à la section 2.5, de manière à comparer leurs fonctionnalités respectives, et par une synthèse à la section 2.6.

2.1 Travaux du laboratoire DORSAL

Le laboratoire Distributed Open Reliable Systems Analysis Lab (DORSAL) conduit un ensemble de recherches sur les systèmes distribués multicoeurs pour les applications critiques, en particulier dans le domaine du traçage. Les partenaires actuels du laboratoires sont notam-

Tableau 2.1 Concepts et mots clés de recherche

Logiciel	Distribué	Observation	Performance
operating system	transaction	tracing	critical path
kernel	remote procedure	reverse engineering	scaling
virtual machine	distributed	observation	queuing model
program	network protocol	recording	profiling
source code	message passing	benchmarking	throughput
binary executable	data center	instrumentation	latency
assembly	high availability	execution graph	performance counter
software library	cloud computing	debugging	hardware counter
	client server	runtime verification	

ment CAE, la Défense Nationale du Canada, Efficios, Ericsson et Opal-RT. La proposition de recherche s'inscrit dans la démarche globale du laboratoire. Voici un résumé des contributions principales.

Le laboratoire a démarré le projet Linux Tracing Toolkit (LTT) en 1999 [9], un traceur noyau pour le système d'exploitation Linux. Son successeur Linux Tracing Toolkit Next Generation (LTTng) offre une meilleure mise à l'échelle pour des architectures multicoeurs par l'utilisation de structures de données sans verrous et la modification dynamique de code pour l'activation des points de trace [5], [10], [11]. LTTng a été étendu pour supporter le traçage haute performance en mode utilisateur [12], [13]. Des travaux portant sur la synchronisation temporelle de traces distribuées ont été réalisés [14]–[16]. La version de LTTng 2.6, publiée en février 2015, représente l'état de l'art dans le domaine du traçage. Cette version supporte le traçage en mode noyau, en mode utilisateur, les points de trace dynamiques et les compteurs de performance. Cette infrastructure de traçage utilise des tampons par processeur, sans verrou couteux, et enregistre au format Common Trace Format (CTF) [1], un format standard adapté au traçage.

Du point de vue de l'analyse des traces, on retrouve deux outils de visualisation, soit LTT Viewer (LTTV) et le plugin LTTng pour Eclipse. Ces deux outils fournissent un graphique de l'état des processus selon le temps, un histogramme et des statistiques de base [17]. Des progrès ont été réalisés dans l'utilisation de traces pour des analyses de sécurité [18], la reconnaissance de séquence d'événements à l'aide de machines à états finis [19], l'étude des relations de blocage entre les processus [20] et la rétro-ingénierie [21]. Un index hiérarchique

1. http ://www.efficios.com/ctf

d'intervalles a été développé, ce qui permet de retrouver l'état du système à n'importe quel moment de la trace en temps logarithmique [22].

2.1.1 Méthodes de corrélation statistique des paquets réseaux

Les méthodes statistiques retrouvent des liens entre les événements par une technique de corrélation temporelle [23]–[25]. Les événements en entrée et en sortie sont observés et le temps écoulé est enregistré. Avec un échantillon suffisant, l'algorithme rapporte la probabilité du lien causal entre les événements.

Constellation [26] se base sur une technique statistique pour déterminer un réseau de dépendances entre des ordinateurs. Le trafic réseau est enregistré passivement. Les adresses et les ports de communication sont utilisés pour l'analyse. L'hypothèse est faite qu'une requête est probablement la cause d'une autre si elles sont rapprochées dans le temps. En enregistrant un profil sur une longue période, un test statistique est utilisé pour discerner les requêtes reliées. Le taux de faux positif obtenu est de l'ordre de 2% pour les requêtes HTTP. Une approche similaire a été proposée par Aguilera et al. [27].

Les méthodes basées sur l'observation passive du réseau présentent l'avantage de ne pas ajouter de latence supplémentaire lors de l'exécution, car l'instrumentation n'est pas sur le chemin critique. Aussi, cette approche s'applique pour des applications quelconques. Les désavantages sont la présence de faux positifs ou de relations qui ne sont pas identifiées. En particulier, une faible précision est obtenue pour les chemins de communication rares. Enfin, l'observation unique du réseau ne permet pas de déterminer les liens entre les processus spécifiques impliqués, ni les liens entre les processus locaux d'un ordinateur, à moins d'avoir une instrumentation supplémentaire.

2.2 Méthodes précises d'analyse de l'exécution distribuée

L'approche précise nécessite la connaissance de la sémantique des événements pour mettre à jour un modèle du système. Le modèle et les événements doivent correspondre. L'emplacement de l'instrumentation change la nature des événements disponibles. Les méthodes précises présentent une difficulté à l'égard de la mise à l'échelle, car la quantité d'événements à traiter croit de manière proportionnelle à la taille du système observé, contrairement aux techniques statistiques qui font un échantillonnage ou une corrélation pour déduire un comportement probable.

L'instrumentation de l'application ou des librairies est une manière simple et efficace pour instrumenter une application distribuée de manière précise [28]–[37]. L'instrumentation sta-

tique du programme nécessite la modification du code source et ajoute une dépendance sur une librairie de traçage. Il existe des méthodes d'instrumentation dynamique modifiant sur place le programme exécuté [38]–[40], mais les techniques varient en fonction du langage de programmation et de l'environnement d'exécution. L'instrumentation d'une librairie a l'avantage d'éviter la modification du code source, mais restreint l'analyse aux applications utilisant cette librairie, ce qui ne permet pas d'observer une application quelconque. Cette technique ne fonctionne pas non plus dans le cas des programmes liés statiquement. L'instrumentation au niveau applicatif n'a pas accès aux événements du système d'exploitation, comme les interruptions et l'ordonnancement, ce qui limite la possibilité de caractériser le temps écoulé. Pour la suite de cette section, les approches les plus intéressantes et pertinentes au sujet sont détaillées.

Google Dapper [29] se base sur l'instrumentation de la librairie de communication *Remote Procedure Call* (RPC) pour l'analyse des requêtes distribuées. La librairie assigne un numéro unique aux requêtes au point d'entrée du système. Cet identifiant sert ensuite à grouper les événements relatifs à une requête. Chaque requête RPC est tracée récursivement pour constituer l'arbre des requêtes selon les composants impliqués. Pour minimiser le surcout et activer le système en production, les requêtes sont échantillonnées, ce qui permet de mettre à l'échelle l'observation. Une requête est tracée complètement ou pas du tout. Le temps passé dans un composant est décomposé entre le temps de communication, d'attente et de traitement. Les requêtes sont agrégées par similarité pour calculer la variance et détecter des anomalies. Cette méthode ne nécessite pas de modifier l'application elle-même, mais ne s'applique que pour les applications utilisant la librairie de communication instrumentée, ce qui limite sa généralité. D'ailleurs, le modèle de performance utilisé est réducteur et spécifique aux applications RPC utilisées dans l'environnement contrôlé de Google. Finalement, les auteurs se demandent eux-même comment associer l'instrumentation du noyau dans leur modèle.

Jumpshot [41] est un outil de visualisation de trace d'une application *Message Passing Interface* (MPI). Le programme est instrumenté en le liant avec la librairie `libmpe`. Cette librairie détourne les appels aux fonctions MPI, génère des événements reliés aux communications des programmes, puis effectue l'appel réel de la fonction interceptée. La vue de la trace montre l'ensemble des processus distribués impliqués dans le traitement. La vue temporelle montre le temps passé en calcul et en communication. Pour chaque communication, une ligne montre le lien entre l'émission et la réception d'un message. L'utilisation de Jumpshot requiert de refaire l'édition des liens de l'application, mais ne nécessite pas la modification du code source lui-même. Seules les applications MPI peuvent être instrumentées de la sorte, ce qui réduit la portée de cette instrumentation au même titre que Google Dapper. D'autre part, le trai-

tement fait par l'application en dehors des appels à la librairie de communication MPI n'est pas visible, ce qui limite la capacité de caractériser le temps écoulé.

Stardust [33] est capable de déterminer les ressources utilisées (temps processeur, accès disque, transfert réseau et utilisation de la mémoire) pour le traitement d'une requête distribuée, ainsi que la latence induite par chaque composant. En particulier, le système produit une analyse très précise en imputant les écritures sur disque effectuées en arrière-plan aux processus qui en sont la cause. Les événements sont reliés par la relation transitive d'un identifiant propagé aux sous-systèmes. Le système a été évalué par l'instrumentation du serveur NFS Ursa Minor. Environ 200 points de trace ont été ajoutés manuellement dans le code des composants du système pour obtenir le résultat, ce qui n'est pas approprié pour l'observation de type boîte noire.

Magpie [35], [36] extrait le chemin d'exécution d'une requête distribué. Le traceur Event Tracing for Windows (ETW) est utilisé pour obtenir les événements du système d'exploitation, des applications et du réseau. L'analyse de la trace est généralisée de deux manières. Premièrement, l'utilisation d'un schéma décrit la sémantique des événements disponibles. Deuxièmement, la jointure temporelle décrit les liaisons à créer entre les événements pour reconstituer la requête. La jointure exploite le lien transitif existant entre les champs des événements. La propriété de clôture transitive forme la requête complète. La requête obtenue comprend les accès disques, l'ordonnancement et les autres événements du système d'exploitation. Les chemins d'exécutions similaires sont regroupés avec un algorithme calculant la distance de Levenshtein. Le désavantage majeur de Magpie concerne la nécessité de l'instrumentation requise du cadre applicatif avec ETW. Cette approche nécessite l'accès au code source du cadre applicatif et sa modification, ce qui n'est pas envisageable dans le cas d'une infrastructure hétérogène. Aussi, Magpie a été évalué dans le cadre de l'analyse d'un service Web, ce qui est un sous-ensemble des applications que nous souhaitons observer. Une approche similaire a été proposée par Mysore et al. [42].

X-Trace [43] retrouve le chemin causal de communication en instrumentant la couche réseau avec un identifiant. Cet identifiant est relayé au prochain composant ou propagé aux sous-requêtes de manière récursive. L'identifiant est répliqué à travers les couches réseau, soit à travers les couches applicatives, TCP et IP. Les champs d'options de TCP et IP sont utilisés pour conserver les métadonnées. La manière de propager les métadonnées à la couche applicative dépend du protocole. La trace obtenue permet de relier le chemin causal complet de manière précise à travers les zones réseau. Cette technique requiert une modification invasive des applications de manière à déterminer le type de propagation de l'identifiant à effectuer, il ne s'agit pas à proprement dit d'une méthode d'instrumentation de type boite

noire. D'autre part, aucune autre information de profilage n'est disponible pour analyser la performance de l'application.

BorderPatrol [23] retrouve le chemin d'exécution d'une application distribuée. Les appels à la librairie standard C sont interceptés par le préchargement d'une librairie dynamique. Des événements sont enregistrés lors de l'établissement d'une connexion et lors de la transmission de messages. Ce qui distingue cette étude est l'utilisation de processeurs de protocoles standard pour observer et identifier la nature des messages échangés entre les composants. L'exactitude du chemin d'exécution retrouvé repose sur trois hypothèses concernant le comportement des applications observées, soit l'honnêteté de l'application (absence de bogue et comportement non malicieux), le traitement immédiat (sans multiplexage) et le traitement identique de requêtes reçues simultanément et séquentiellement. L'application n'a pas besoin d'être modifiée pour être instrumentée, ce qui constitue un avantage. Les processeurs de protocoles fournissent un contexte pour l'identification d'une requête, mais ne fonctionne pas si le contenu des paquets est chiffré. L'algorithme d'analyse hors ligne de la trace fonctionne par une copie dans une base de données SQL et souffre d'un problème de mise à l'échelle. Aussi, le format fixe des événements restreint la portée de l'instrumentation aux appels systèmes et aux adresses de communications et ne permet pas de caractériser précisément le temps de traitement.

vPath [44] extrait le chemin d'exécution d'une requête par l'enregistrement d'événement lors de l'envoi et de la réception de message de chaque fil d'exécution. Les événements sont enregistrés par l'hyperviseur Xen, ce qui ne nécessite aucune modification au code source des applications, ni l'inspection du contenu des messages. L'implémentation fonctionne pour l'architecture x86. Chaque fils d'exécution est identifié uniquement par le tuple formé de l'identifiant du domaine et les valeurs des registres CR3 et EBP, représentant respectivement le processus et le fil d'exécution. L'analyse fonctionne pour des applications respectant deux idiomes de programmation, soit le traitement synchrone des requêtes au-dessus d'une couche de communication fiable et la répartition du travail par fil d'exécution. L'étude démontre que la plupart des applications et des intergiciels courants respectent ces idiomes. La causalité du point de vue des fils d'exécution est capturée par l'observation des événements d'ordonnancement du système d'exploitation, tandis que la causalité entre les composants est déterminée par les messages échangés. Le modèle de traitement par un ensemble de fils d'exécution de travail satisfait l'hypothèse du comportement, et ce modèle est prévalent. Le système ne supporte pas les modèles d'exécution utilisant de la mémoire partagée, tel que certains pipelines, à cause de la limitation de l'instrumentation.

Whodunit [45] extrait l'exécution d'une transaction distribuée et produit un profil de l'exécution selon le composant. Le profilage de chaque composant est obtenu avec csprof [46]. L'instrumentation des messages est effectuée pour déterminer les liens entre les processus distribués et est implémentée par la surcharge de la librairie C. D'autre part, dans le but de détecter les liens entre des processus locaux, la communication en mémoire partagée est tracée. Ceci est accompli en interceptant les appels aux fonctions de verrou, puis en exécutant le code de la section protégée avec l'émulateur QEMU. Les adresses utilisées sont enregistrées pour identifier les processus producteur et consommateur. Whodunit est spécifiquement conçu pour les application multi-tiers.

PreciseTracer [47] extrait le chemin d'une requête distribuée de manière précise et en ligne. La construction du graphe d'activité de la requête se base sur les échanges de messages entre les applications. Chaque événement enregistré associe un message TCP au processus impliqué. Ces événements sont ensuite reliés entre eux pour former le graphe d'activité. Les graphes similaires sont superposés dans le but de présenter une vue synthèse de l'exécution. Deux approches sont utilisées pour améliorer la mise à l'échelle de l'analyse. Le traçage est activé globalement pour une courte période, ce qui diminue l'impact par rapport à si le traçage était toujours actif. Ceci diminue la charge moyenne sur une longue période, mais ne diminue pas le surcout lorsque l'instrumentation est activée, ce qui a un impact sur les requêtes traitées pendant ce temps. L'autre approche consiste à effectuer un échantillonnage aléatoire des événements. Pour un taux d'échantillonnage de 90% des événements du banc d'essai RUBiS, 90% des graphes sont exacts. La méthode a l'avantage de montrer que l'algorithme est tolérant à un certain taux de pertes, mais un taux d'échantillonnage de 90% ne diminue que marginalement le surcout du traçage, et l'algorithme n'a pas été testé avec un taux d'échantillonnage plus faible.

Self-Propelled Instrumentation [48] instrumente dynamiquement une application distribuée en suivant le flot de contrôle de son exécution. Une librairie contenant l'instrumentation est chargée et l'exécutable est modifié sur place pour insérer un saut vers l'instrumentation correspondante. La librairie contient des fonctions de rappel définies par l'utilisateur. L'instrumentation est propagée aux processus enfants en interceptant les appels aux fonctions de création de processus de la librairie C. Lorsque l'application se connecte à un serveur, alors celui-ci est instrumenté en arrière-plan lors de l'établissement de la connexion avant de retourner. L'opération s'effectue sur l'ordinateur pair par SSH. Nous avons mesuré que l'établissement de la connexion SSH locale sur un Intel core i5 à 2.5 Ghz et le serveur OpenSSH avec l'authentification par clé publique de 2048 bits à elle seule est de l'ordre de 250ms. Par conséquent, cette technique ne préserve pas les propriétés temporelles de l'application. Aussi, l'implémentation de la propagation de l'instrumentation comporte des implications au

niveau de la sécurité et de la configuration du système. Le client doit avoir le privilège de modifier l'exécutable du serveur, ce qui n'est clairement pas désirable dans un environnement de production. Aussi, le client et le serveur doivent s'exécuter sur des ordinateurs de même architecture, ce qui ajoute une contrainte sur le matériel supporté. Finalement, l'instrumentation de binaire ne fonctionne que pour les programmes compilés et exclu les exécutables intermédiaires comme le bytecode Java.

2.3 Calcul du chemin critique

Miller et al. [49] présentent un algorithme pour le calcul du chemin critique d'une application distribuée basée sur les messages échangés. Le Program Activity Graph (PAG) ou graphe d'activité d'un programme, est défini comme un graphe dirigé acyclique, pour lequel les nœuds sont les événements délimitant le début et la fin d'une activité et dont les arcs représentent le temps processeur utilisé. Le chemin le plus long correspond au chemin critique. Sa longueur est la somme du poids de ses segments. Leur étude compare un algorithme centralisé à un algorithme distribué pour le calcul du chemin critique hors ligne. L'accent est donc sur l'accélération du calcul hors ligne du chemin critique par un algorithme parallèle.

Hollingsworth et al. [50] proposent une méthode pour le calcul en ligne du chemin critique d'un graphe d'activité. La technique présentée évite de construire le graphe complet et de stocker les événements. La technique consiste à adjoindre la valeur courante du chemin critique à un message envoyé. Lors de la réception d'un tel message, cette valeur est copiée dans une variable locale au processus. Le temps de traitement du processus est ajouté. La valeur à jour est transmise avec la réponse. Lors de la réception d'un message, la valeur la plus élevée entre celle locale et celle reçue est conservée.

Le calcul du chemin critique proposé par Miller et Hollingsworth est efficace pour les applications dont la performance est limitée par le temps processeur. Or, ce type d'analyse est insuffisante pour déterminer le chemin critique d'un système qui serait borné par les entrées sorties. Si le temps d'attente et de communication est inclus au poids des arcs, alors tous les chemins du graphe ont le même poids, ce qui rend les algorithmes de calcul du chemin critique existants inopérants. Aussi, les dépendances entre les processus locaux utilisant d'autres mécanismes de communication, comme les tubes et les signaux, ne sont pas supportés, ce qui restreint le nombre de programmes qui peuvent être observés efficacement.

LTTV est un visualiseur de trace noyau développé au laboratoire DORSAL. Il contient plusieurs modules, dont un histogramme du nombre d'événements selon le temps, une vue des ressources du système et un module d'affichage de l'état des processus. L'analyseur recons-

titue l'état du système depuis la trace. En ce qui concerne l'analyse du chemin critique, le module d'analyse de dépendance de LTTV [20] utilise le traçage noyau pour déterminer la cause de l'attente dans un programme. L'analyse porte sur les blocages se produisant lors d'appel système. L'émetteur de l'événement de réveil indique la cause de l'attente. L'algorithme utilisé assume une structure d'attente en arbre. Cette structure n'est pas adéquate pour une application distribuée à cause des messages échangés, qui doivent être représentés par un graphe dans le cas général.

2.4 Détection de goulots d'étranglement

Saidi et al. [51] présentent une méthode d'extraction du chemin critique de l'exécution pour en déterminer les goulots, qu'ils soient logiciels ou matériels. La méthode consiste à modéliser sous forme d'automate les composants à analyser, dont les événements sont les transitions. Les liens entre les automates sont définis par l'utilisateur et forment le flot de contrôle et d'attente. Les états des automates correspondent aux arcs du graphe de dépendance. Ceux-ci sont annotés avec la durée pendant laquelle l'état est maintenu. Le système de mesure a été implanté avec un simulateur matériel. L'analyse à bas niveau du matériel se fait en instrumentant le simulateur avec des fonctions de rappel. L'analyse logicielle utilise la capacité du simulateur d'instrumenter l'entrée et la sortie des fonctions. Les automates sont annotés avec les étiquettes des fonctions. Le chemin critique est calculé sans reconstruire le graphe complet à la manière décrite par Hollingsworth et al. La méthode de visualisation du graphe proposée consiste à regrouper les graphes isomorphes et annoter les arcs par le nombre d'exécutions, le temps total écoulé et la proportion du temps sur le chemin critique. L'analyse vise spécifiquement la détection des goulots dans le code source et le matériel, tandis que nous nous intéressons au cas général du graphe d'exécution au niveau du système. En outre, l'analyse requiert la connaissance du code source des applications, du noyau et un modèle du matériel, ce qui n'est pas envisageable pour l'analyse de type boîte noire.

L'analyse d'un réseau de files d'attente en couche (en anglais Layered Queuing Network, ou LQN) a été démontré efficace pour déterminer la capacité et les goulots d'un système distribué [52]. En particulier, cette analyse a été appliquée avec succès pour déterminer les goulots d'un système temps-réel souple de téléphonie IP [53]. Il s'agit d'une généralisation d'un modèle de la théorie des files d'attentes pour des systèmes distribués. Le temps d'attente moyen en file et la longueur moyenne de la queue sont des exemples de mesures de performance que l'analyse peut produire. Pour certains modèles simples, il existe une solution analytique. Dans les autres cas, les résultats peuvent être obtenus par simulation discrète du modèle. La difficulté reliée à l'utilisation de LQN concerne la génération du modèle de performance depuis

le système. L'automatisation de cette approche est proposée pour faciliter la construction du modèle [54].

L'analyse du comportement statistique des paquets réseaux a été utilisée pour déterminer l'atteinte d'un goulot d'étranglement, sans pour autant connaître à l'avance la capacité maximale du système [55]. La distribution de la latence de communication est compilée pour une fenêtre de temps. D'après cette distribution, le facteur de dissymétrie de la distribution est calculé. Une dissymétrie positive indique que la distribution est compressée à la limite du système et donc l'atteinte d'un goulot.

2.5 Traceurs

Le traceur LTTng a été présenté dans la Section 2.1. Cette section présente les autres traceurs disponibles et leurs particularités.

Event Tracing for Windows (ETW) [56] est le système de traçage développé par Microsoft pour Windows. L'infrastructure trace autant le noyau que les applications en espace utilisateur. Des tampons par CPU sont utilisés pour diminuer le surcout lié à la synchronisation sur des architectures multicoeurs. L'outil Windows Performance Analyzer (WPA) affiche des graphiques de l'utilisation des ressources du système, calculés depuis les événements de la trace. L'analyseur utilise les symboles de débogages des exécutables tracés, pour relier les métriques au code source des applications.

DTrace [57] est un traceur disponible pour Solaris, Mach et FreeBSD. Ses fonctionnalités comprennent le traçage du noyau et des programmes en espace utilisateur, la définition d'une fonction de rappel par type d'événement, le filtrage et le traçage spéculatif. Le traçage est défini par un script en langage D, dans lequel sont spécifiés les points de trace à activer et leur traitement.

SystemTap [58] est un traceur dynamique pour Linux similaire à DTrace. Il permet d'accéder aux points de trace du noyau et aux compteurs de performance. Il permet aussi d'insérer dynamiquement un point de trace dans un programme en espace utilisateur à l'aide d'un point d'arrêt, une technique utilisée par les débogueur interactifs. Un script STP définit les points de trace à activer et la manière d'agréger les données en vue de leur affichage dans une console texte. La fonction d'affichage est appelée périodiquement, ce qui constitue la seule manière pour consulter les données. Le script STP est compilé dans un module noyau, puis chargé dynamiquement pour effectuer l'analyse en fonctionnement. Les analyses offertes sont limitées par les structures de données disponibles dans le langage. L'impact de performance

est important, puisque l'analyse se fait *in situ*, et l'implémentation courante comprend un verrou unique qui sérialise tous les accès au module d'instrumentation.

Perf [59] est un traceur regroupant les compteurs de performance matériels et logiciels du noyau Linux. Il est utilisé pour profiler l'exécution du noyau ou d'un programme en espace utilisateur. Les compteurs de performance disponibles varient selon l'architecture, comme le nombre d'instructions par cycle d'horloge et le nombre d'accès invalides à la cache. Les compteurs logiciels incluent par exemple les fautes de page mineures et majeures et les changements de contexte. La trace est accessible depuis un bloc de mémoire partagé et peut être sauvegardée ou traitée en continu.

Ftrace [60] permet d'obtenir le détail de l'appel des fonctions du noyau et le temps passé dans chacune d'elle. Il existe des modules d'analyse spécialisés, comme celui pour mesurer la latence de l'ordonnanceur pour des tâches temps réel.

LTTng sera utilisé pour obtenir les résultats de cette recherche. Ceci se justifie parce qu'il offre un niveau de fonctionnalité supérieur aux autres traceurs disponibles sous Linux et que nous avons accès au code source pour effectuer nos expérimentations. Du point de vue de la performance, des bancs d'essais démontrent que le traceur noyau LTTng requiert 119 ns pour enregistrer un événement en mémoire sur un processeur Intel Xeon cadencé à 2 GHz, dans des conditions de cache optimale [5]. Nous pensons que ces performances sont adéquates pour la réalisation du projet.

2.6 Synthèse

Les applications réparties sont très importantes et complexes ce qui motive beaucoup de travaux dans ce domaine. Les approches existantes basées sur l'instrumentation en espace utilisateur, comprenant le programme lui-même, les librairies et les intergiciels, sont limitées en terme de portée, car elles sont spécifiques à un langage (C/C++, Java, Python, etc.), un domaine d'application (MPI, HTTP, SQL, etc.) ou un intergiciel (Django, protobuf, RMI, etc.). Un système hétérogène est caractérisé par l'utilisation d'une combinaison de ces technologies, dont l'instrumentation respective a été développée indépendamment. Le résultat est un ensemble d'évènements bigarrés et incompatibles, ce qui rend impossible en pratique d'obtenir une vue d'ensemble du système à l'aide d'un modèle global. Standardiser l'instrumentation de tous les programmes est une utopie considérant l'ampleur de l'effort et de la coordination requise pour y arriver. Une autre solution technologique est donc nécessaire.

D'autre part, les profileurs traditionnels ne prennent pas en compte l'attente se produisant dans une tâche ni les liens existants entre les tâches. Pour illustrer cette limitation, considé-

rons la commande `sleep 1`. Mesurer le ratio d'instruction par cycle de ce programme en vue d'une optimisation n'aura qu'un effet négligeable sur le temps d'achèvement, car le temps d'attente pour l'expiration du minuteur est dominant. Cette particularité fait en sorte qu'un rapport de profileur peut être trompeur et doit être interprété avec précaution. Une autre limitation concerne le profilage des tâches connexes à celle profilée. Certains profileurs tiennent en compte les processus enfants de la tâche racine, mais aucun ne tient en compte les tâches en arrière plan déjà en exécution au lancement du profilage, tel qu'un serveur ou un démon. Quant au profilage global du système, il ne permet pas d'isoler le traitement relatif à un groupe de tâches reliées. Il est aussi à noter que toute optimisation à un serveur produira une accélération du client que si ce dernier attend pour la réponse. Aucun des profileurs étudiés ne prend en compte cette relation d'attente existante dans un traitement multitâche.

Les travaux sur les blocages initiés par Fournier et.al. [20] démontrent l'intérêt d'utiliser les évènements du système d'exploitation pour comprendre le temps écoulé dans une application complexe. L'analyse utilisant les évènements noyau est indépendante de l'environnement logiciel. Cette technique identifie l'attente entre les tâches en mode utilisateur et les périphériques. Cependant, l'attente entre les tâches noyau n'est pas incluse dans l'analyse, les appels systèmes supportés sont limités, l'analyse est limitée à un seul ordinateur et la correspondance entre la trace noyau et l'application n'est pas disponible. Le travail qui suit s'appuie donc sur ces travaux antérieurs et vise à éliminer ses limitations et étendre sa portée.

CHAPITRE 3 MÉTHODOLOGIE

Cette recherche est appliquée et expérimentale. Les expériences conçues visent à comprendre le comportement du système d'exploitation en lien avec son instrumentation disponible au départ. Cette rétro-ingénierie du noyau a permis ensuite de concevoir l'instrumentation additionnelle requise pour atteindre nos objectifs, puis de réaliser les analyses exploitant cette instrumentation. Les résultats ont été évalués pour vérifier la présence de cas particuliers ou de limitations. À chaque itération, l'instrumentation et l'analyse ont été améliorées de manière à la rendre plus générale. L'aspect expérimental comprend également la mesure du surcout du traçage et les performances de l'analyse elle-même. En dernier lieu, des études de cas ont été réalisées pour démontrer l'intérêt de la méthode proposée en pratique. Le reste de ce chapitre explique plus en détail les aspects méthodologiques.

3.1 Étude du système

La base de l'étude nécessite de comprendre certains comportements spécifiques du système d'exploitation. La dynamique étant non déterministe, le comportement à étudier peut ne pas se manifester. Nous avons donc développé la suite d'outils `workload-kit`. La liste des principaux outils développés est présentée au Tableau 3.1. Chaque programme a pour objectif de forcer le système dans un état particulier pour en étudier efficacement le comportement. Le blocage entre des tâches concurrentes et distribuées et au coeur de l'étude. Tous les mécanismes de communication interprocessus sont étudiés, ce qui comprend les verrous (mutex et sémaphore), les tubes, les fichiers et les sockets, en plus de l'attente pour la terminaison d'une tâche. Seule la mémoire partagée est un cas particulier. Si les accès concurrents sont protégés par des verrous, alors ceux-ci sont inclus dans l'analyse. Les accès concurrents atomiques ne bloquent pas, et donc ne modifient pas le flot d'exécution du point de vue du système d'exploitation. Par l'entremise de ces programmes, nous avons également étudié la relation d'attente avec les périphériques, comprenant les disques, les minuteurs et les interfaces réseau.

Ces outils sont programmés en C et utilisent des librairies de base, dont la relation entre le code source et les appels système résultants est simple à établir. Cette caractéristique confère un meilleur contrôle sur la trace produite et réduit le non-déterminisme. Une machine virtuelle ou un interpréteur produit une trace plus compliquée, considérant la compilation juste à temps et les abstractions supplémentaires, tandis que le comportement à étudier représente une plus faible proportion de l'exécution totale, d'où l'intérêt d'utiliser des programmes compilés statiquement pour cet aspect de la recherche.

Certaines situations nécessitent de simuler un calcul à durée fixe, indépendamment de la vitesse du processeur. Par exemple, tester un algorithme, calculant l'utilisation du processeur obtenu à partir d'une trace, requiert de comparer la valeur obtenue à celle attendue. Un calcul de durée fixe est obtenu par une procédure de calibration qui détermine la fréquence d'incrément du processeur. L'incrémentation est ensuite effectuée proportionnellement à la durée d'activité à simuler. Cette technique permet de simuler un calcul de 1 ms avec une précision de l'ordre de 1% (IC 95% [0.993, 0.991]). Cette variation est tenue en compte dans la vérification des résultats.

Lors de mesures de performances, une source importante de variation a été identifiée. Par défaut, la fréquence du processeur varie dynamiquement par rapport à la charge du système, dans un souci d'économie d'énergie. Pour éliminer cette source de variation, nous avons fixé la fréquence du processeur à son maximum pour l'ensemble des mesures.

Le comportement d'un programme peut être très différent selon l'état de la cache des pages du système. Lors de la lecture d'un fichier sur disque dont le contenu est absent de la cache, la tâche bloque en attente du périphérique. La tâche ne bloque pas si la page est déjà présente dans la cache. La cache des pages est vidée avant les expériences sensibles à son état.

Les traces sont générées de manière non interactive et peuvent être régénérées à la demande, ce qui est un critère important pour reproduire les expériences de manière fiable. Deux utilitaires ont été développés pour les produire, soit `lttng-simple` et `lttng-cluster`, qui gèrent respectivement une session de traçage localement et sur un groupe d'ordinateurs simultanément. Les paramètres d'exécution sont modifiables afin d'en étudier l'effet.

3.2 Mesures du cout de l'instrumentation

La mesure de l'impact du traçage est centrale à notre étude. La démarche générale est de mesurer le surcout moyen, en fonction du type de charge. Le surcout est décomposé entre l'augmentation du temps d'exécution du programme observé, des E/S correspondant à l'écriture ou le transfert de la trace, et finalement l'utilisation du processeur par les démons de traçage en arrière-plan. Le surcout moyen pour différents cas d'utilisation est utile pour évaluer l'ordre de grandeur de l'impact dans la réalité. Le programme libre `sysbench` a été utilisé pour mesurer l'impact de traçage en fonction du type de charge. Le surcout moyen du traçage a aussi été mesuré pour des applications typiques, tel qu'un service Web.

Or, le surcout est étroitement lié à la dynamique du programme utilisé. Pour cette raison, nous avons aussi évalué la borne supérieure du surcout, indépendamment d'une application particulière. La stratégie mise en oeuvre consiste à imposer une charge qui produit la plus

Tableau 3.1 Suite d'outils de `workload-kit`

Programme	Description
wk-cpm	Génère des combinaisons d'attente récursive entre processus parent et enfant.
wl-imbalance	Simule une répartition inégale de travail dans un calcul parallèle.
wl-deadlock	Provoque un interblocage entre deux fils d'exécution.
wk-ioburst	Produit des E/S synchrones et asynchrones, en cache froide et chaude.
wk-lockfight	Contention aléatoire sur un verrou entre plusieurs fils d'exécution.
wk-pipeline	Traitement producteur-consommateur par l'entremise d'un tube.
wk-pulse	Force la contention périodique de deux tâches sur un processeur.
wk-reparent	Simule une tâche parente quittant avant son enfant.
wk-rpc	Serveur de requête minimal par TCP/IP et le client correspondant.
wk-schedfreq	Force des cycles d'ordonnancement à la fréquence maximale.

haute fréquence d'événements possible. Par exemple, il a été déterminé que, pour un programme local, environ 90% du surcout est causé par les événements d'ordonnancement. Le programme `wk-schedfreq` démarre deux fils d'exécution s'échangeant un sémaphore, ce qui force un cycle d'ordonnancement, dont la fréquence est de l'ordre de 10^5 Hz.

L'analyse de la trace doit être elle-même aussi performante que possible, bien que cela ait une importance moins critique. Les algorithmes proposés ont été analysés en terme de temps et d'espace requis par rapport à la taille de la trace. D'un point de vue pratique, quelques secondes suffisent pour analyser les traces de nos expériences, puis la visualisation interactive depuis l'interface graphique est possible.

3.3 Validation

La validation des algorithmes proposés a été faite en trois étapes. La première consiste à appliquer l'algorithme de calcul du chemin actif de l'application directement sur un modèle de graphe, pour chaque cas de base déterminé lors de l'étude du comportement du système. Le chemin obtenu est comparé à celui attendu en termes de structure, de nombre de noeuds et de longueur des arcs. Le fait de valider l'algorithme sur un graphe synthétique évite la variation reliée au non-déterminisme d'une trace réelle, ce qui simplifie le développement et le débogage.

Dans un second temps, l'algorithme de construction du graphe a été validé pour s'assurer qu'il possède les propriétés attendues par l'algorithme d'extraction du chemin actif. Les

programmes de `workload-kit` sont idéals pour produire les traces nécessaires à la validation. Ces traces ont servi d'entrée à l'algorithme de construction du graphe, puis la structure du graphe produit a été vérifiée par rapport à celle attendue dans chaque situation.

D'autres programmes sont nécessaires pour démontrer que la technique proposée fonctionne dans des cas d'utilisation rencontrés en pratique. À cette fin, un ensemble de systèmes distribués ont été déployés. Ces systèmes sont implémentés en C/C++, Java, Python et Erlang, utilisant les principaux protocoles de communication et librairies courantes, comme un service Web, un programme MPI et l'accès à des objets distribués.

3.4 Environnement

Les expériences ont été réalisées sur l'architecture x86 SMP. Le jeu d'instruction est peu important pour la recherche, car les traces noyau sont généralement indépendantes de la microarchitecture. L'analyse elle-même de dépend pas d'évènements propres à une architecture. Les expériences ont été réalisées aussi bien sur un ordinateur individuel, qu'à l'aide de machines virtuelles et d'une grappe d'ordinateurs.

Les logiciels utilisés dans le cadre de la recherche sont tous à code source ouvert. La communauté scientifique peut facilement reproduire nos résultats, ainsi qu'étudier et étendre les expériences. En particulier, l'accès au code source de Linux a été essentiel à la conduite du projet.

Les résultats obtenus pour les trois thèmes suivants l'ont été grâce aux éléments méthodologiques présentés. Les détails du matériel et des versions mises en oeuvre ont évolué au cours de la recherche, sont précisés dans leur section respective, et reflètent l'état de la technologie à ce moment.

CHAPITRE 4 ARTICLE 1 : Approximation of Critical Path Using Low-level System Events

4.1 Abstract

This paper addresses the challenge of understanding the system-level critical path of applications with several interacting tasks, which is beyond the reach of traditional instruction profilers. The goal is to improve application responsiveness by identifying execution segments affecting the completion time with minimal intrusiveness.

We start with a model of fundamental interactions between cooperating tasks at the system level and analyse their critical path. We conclude that system level events are not sufficient to handle the case of locking primitives in user-space. We evaluate an approximation of the critical path by recovering blocking causes recursively. We found that the resulting path is more precise for synchronous programs, and that error related to lock contention is low under conservative assumptions.

To perform this analysis, the prior approach is extended to take into account kernel threads. The new method has also the advantage of reducing the runtime overhead. We validated our approach using a set of workloads that represent common programming patterns. Finally, we demonstrated the benefits of the proposed approach by using our tool to quickly identify easy optimization opportunities in the important software package management application `APT`, used in many of the most popular Linux distributions. We measured the tracing overhead required to perform the analysis for three types of workloads to assess its impact on the execution, and found that overhead less than 10% can be achieved in all cases.

4.2 Introduction

Reducing perceived latency is critical for user satisfaction [2], [3]. The cause of an intermittent performance problem is hard to identify because of the numerous software and hardware components involved [61]. Unfortunately, general code profiling tools, such as Valgrind [62] or OProfile [63], do not take into account the system state of programs. For instance, many processes waste time in non obvious ways while waiting on timers, events, resources or other processes. Aggregated statistics report the performance of the overall execution, but the time dimension is flattened.

Tracing is a type of monitoring, well suited to understand latencies. It consists in recording key events of the execution along with a timestamp. Drawbacks of tracing are related to the overwhelming size of the data set generated, the runtime overhead and the cost of developing and maintaining the instrumentation.

Our objective is to provide an automatic analysis tool to recover segments of execution affecting the waiting time of a given computation with minimum intrusiveness. To perform this analysis, we propose to rely only on operating system traces. User-space traces can provide more insight about the application logic, but they are not mandatory for the analysis. This choice to use only kernel traces is motivated by the following reasons. Firstly, kernel tracing allows to identify system-wide execution wait-for relationships. Secondly, all programs are supported without any modification to code or binary. Thirdly, the instrumentation is portable across hardware architectures. Finally, very low-overhead can be achieved with state-of-the-art kernel tracing facilities and carefully selected instrumentation.

We first describe the execution model we are proposing to perform the analysis in Section 4.3. We analyze execution patterns and their corresponding critical path at the system level. We demonstrate that it is not possible in every cases to recover precisely the critical path in the case of lock interference between threads using only kernel events. We discuss a possible approximation of the critical path considering only the blocking window, and evaluate the precision of the approximation. We then present empirical results of the analysis with Linux in Section 4.4. We describe how to perform the analysis without intercepting system calls, therefore making it suitable to account for kernel threads. Then, we explain the algorithm to resolve waiting dependencies in linear time according to the graph size. We implemented the analysis as plug-ing within the Eclipse Tracing and Monitoring Framework (TMF). We show the result of the analysis for a representative set of benchmark programs. Finally, we used the proposed tools to study the performance of `APT`, a popular and complex program. We discuss methods available to relate the kernel traces to the actual source code. We conclude with the runtime cost of recording events required for the analysis according to various type of workloads and configurations.

4.2.1 Related Work

Previous works focused on retrieving the critical path of a distributed computation [49], [50]. The Program Activity Graph (PAG) is defined as a directed acyclic graph, where nodes represent the events defining the beginning and end of an activity, and whose edges represent the CPU time used. The longest path in this graph represents the critical path. The length of a path is the sum of its segments' weights. This method works for CPU bound

programs. However, it does not account for waiting time that does not consume CPU cycles, but increases latency.

Critical path analysis of TCP transactions has been studied in [64]. TCP packets are traced and matched to produce a dependency graph. The computed critical path reports the proportion of the completion time due to the server, the client and the network. This method does not apply to local IPC. In addition, network tracing does not account for how time is spent inside endpoint machines [65].

Magpie [36] instruments the system at various levels to track processing of queries. Queries are tagged when they enter the system and this is used to associate processing resources to each request. Multiplexed queries are correctly handled. The analysis is therefore tied to domains where queries apply, such as three-tiers servers. The analysis requires that many userspace applications be instrumented to follow queries, such as the database and the web server. In comparison, our approach does not require application-specific information. In addition, the tool does not extract the critical path from the graph.

vPath [44] uses a virtual machine monitor (VMM) to observe the processing path. The technique does not require modification to the guest systems. System calls related to communication are intercepted, and TCP connections are monitored. Each event includes generic thread identifiers. It requires running the monitored system in a virtual machine, and does not allow monitoring the host itself.

OSprof [66] performs latency distribution analysis. An histogram is computed by measuring system calls durations under specific workloads. The purpose is to analyze the performance of the kernel code itself, and isn't suitable for userspace programs.

Android's Systrace [1] provides a complete system view of the execution. The tool aims at debugging deadline misses of screen refresh, affecting the responsiveness of the UI. Manual inspection is required to find the root cause of these misses.

Panappticon [67] aims at understanding the user perceived latency of Android applications, from the input to the screen update. It is a host based analysis, using instrumentation from the kernel and the Dalvik virtual machine. Events related to inter-process communications are recorded to identify connected tasks. Task blocking indicates the end of work item processing.

The Linux perf utility [59] can perform waiting analysis of a program. The result is a function tree indicating the location of wait in source code. However, the analysis does not recover the waiting cause.

1. http://developer.android.com/tools/help/systrace.html

The work in [68] introduces a method to predict the performance of a parallel program on multi-core systems from the execution on a single processor. The threading library is instrumented to record synchronization operations and function calls. This data serves as input for a simulator that highlights functions to optimize in order to reduce the execution time on a given number of processors. The analysis is limited to the CPU resource alone, and does not take into account I/O for instance. Our graph data structure and critical task flow computation is an extension of this work.

A previous study [20] discusses the detection of blockings in system calls. By following the source of wake-up signals, the chain of events ending a blocking call is recovered. This technique allows to effectively detect the root cause of waits. Our work extends and improves this principle of operation.

4.3 Execution model

Task state representation varies between operating systems. We begin by defining a canonical state machine to abstract these differences. Figure 4.1 shows the corresponding state machine including four states: running, preempted, interrupted and blocked. The task is in the running state when it executes on a given CPU. The preempted state applies when the task could run, but exhausted its time slice, or when a task with higher priority is running. The task is interrupted each time an interrupt service routine executes on that CPU. Note that the interrupt state can be nested, a case which isn't represented in Figure 4.1 for simplification. The blocked state occurs when the task inserts itself into a wait queue. The blocked state ends when the task is removed from the wait queue and added back to the running queue of the scheduler. We define this action as the wake-up signal. The blocked state is a mechanism for passive waiting, in contrast to active waiting that repeatedly polls for some condition to occur. Blocking always occurs in kernel mode, either when a user-space process performs a system call, or at any point for a kernel thread.

The context in which the wake-up occurs reveals the wait cause. We define two categories of wake-up, namely direct and indirect. Direct wake-up occurs when it is performed by a task in the running state, while indirect wake-up is done in the interrupt state. For the indirect wake-up, the interrupt vector indicates the device involved, e.g. an incoming network packet, a completed disk request, or a key press. Direct wake-up indicates a change in the control flow between tasks, while indirect wake-up does not. The wake-up source is known *a posteriori*. For instance, `sys_select` returns either because a file descriptor becomes ready (it may be a direct wake-up in the case of an IPC) or a time-out expires (indirect timer wake-up).

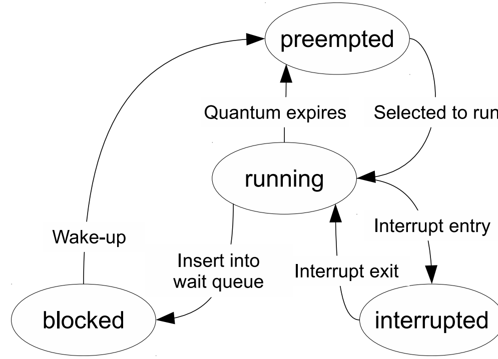


Figure 4.1 Task state finite state machine

We observed that the system-level flow of execution diverges and converges because of waiting relationships. In consequence, we model the execution and inter-task dependencies using a Directed Acyclic Graph (DAG) built from the trace. The data structure is similar to the one used in [69]. We present in Definition 1 an optimized sparse graph specifically for this purpose. Intuitively, the execution graph can be viewed as a two dimensional mesh, where vertices are ordered. Unlike ordinary adjacency lists, the execution graph allows binary search to seek at a given timestamp of a given task.

Definition 1 (Execution graph)
An execution graph G is a 5-tuple (V, S, F, E, Σ), where V is the set of vertices, $S \subseteq V$ is the set of initial vertices, $F \subseteq V$ is the set of final vertices, $E \subset V \times V$ is the set of edges written $v \to v'$ and Σ is the set of edge labels. Each vertex has at most four edges $e_{up}, e_{down}, e_{left}, e_{right}$ to neighbours. Horizontal edges e_{left}, e_{right} denote state changes of a task, while vertical edges e_{up}, e_{down} are inter-task links. Edges are either incoming e_{left}, e_{down} or outgoing e_{up}, e_{right}. Each vertex has a timestamp t representing causality such that $v \to v'$ must respect $t \leq t'$.

4.3.1 Critical path of execution

In Figure 4.2 we present basic interactions between tasks observed while tracing various types of programs. Complex execution graphs are combinations of these sub-graphs. Plain horizontal edges represent the running state and dashed edges represent the blocked state. Vertical edges represent wake-up signals. There are up to three tasks, task A, B and C. Vertices marked with X are not part of the graph, but are used for later discussion. For each sub-graph, we identify edges on the critical path to reach the last vertex w.r.t. task A in Table 4.1. The critical path is defined as the sequence of execution segments such that reducing their duration reduces the completion time [68].

The critical path is retrieved by first zeroing all blockings with direct wake-up, and then applying longest path computation with a backward topological sort from the end vertex of interest. This algorithm works in linear time for DAG [7].

Graph (a) shows the case where task A wakes-up task B and then waits until its processing is completed. This pattern represents for instance the behaviour of a parent and child relationship or a producer and consumer communicating on a pipe. The graph can be viewed as the union of two processing paths. If the edge (B0,B1) is reduced by a small amount ϵ, the wait duration will be reduced and task A will complete faster. However, reducing edge (A1,A2) will cause the waiting (A2,A3) to increase, and will not provide any speed-up.

Graph (b) is an example of repetition, and can be viewed as the concatenation of Graph (a) with itself. The critical path is simply the concatenation of sub-graphs' critical paths.

Graph (c) is typical of parallel programs where multiple tasks are spawned at once. If the edge (B0,B1) is reduced, then the wait time (A3-A4) will be reduced, but the wait edge (A5,A6) will increase, and the completion time will not improve. However, the edge (C0,C1) does affect the end time. We can also state that the edge (A1,A2) starting task C is on the critical path. It reflects the fact that the last thread to finish in a group limits the achievable speed-up.

Graph (d) occurs in the case of nested processing. In this example, task A wakes-up task B, which in turn wakes-up task C. Edges (A1,A2) and (B1,B2) do not affect the end time.

The last two cases (e) and (f) can occur if the trace is truncated for the time window of interest. Otherwise they have important implications in retrieving the critical path.

Graph (e) is an example of asynchronous processing. Task A wakes-up task B, but never waits for the result. In this case, no blocking occurs and we state that task B is never on the critical path of task A.

In Graph (f) task A waits for task B, but there is no prior signal from task A to task B as in previous cases. It may represent a barrier in a computation, but also a lock contention on a shared resource. In the case of a barrier, the edge (B0,B1) is affecting the end time, and thus presents no special issue.

Locking, on the other hand, is different. Suppose now that task B holds a lock between X1 and B1. Task A blocks while trying to enter the critical section, and is woken-up when task B releases the lock at B1. If the lock was released earlier by task B, then it would reduce the completion time of task A. However, we cannot speculate on the importance of the edge (B0,X1) before the lock, because if this edge is reduced enough, then task A may have obtained the lock. This would have the effect of reversing the waiting relationship. In

consequence, we state that while the theoretical critical path is (B0,B1,A2,A3), the critical path of interest is (A0,X0,X1,B1,A2,A3) or even better (A0,A1,X2,B1,A2,A3). Indeed, the only practical and interesting impact of B is the resource contention, in segment (X1,B1). Furthermore, the section of greater interest is when A is actually delayed, waiting on a resource held by B, shown by segment (X2,B1).

The problem is to distinguish between real dependencies and simple interference, through resource contention, between otherwise independent tasks. Four criteria may be used to distinguish between the two. The first, *lock holder*, is to look at the lock on which A is waiting. If the lock was taken and released by B, we can assume that B is independent and this is a case of lock contention. In that case, the critical path should not continue on B beyond the waiting time of A for the lock, yielding (A0,A1,X2,B1,A2,A3). On the other hand, if the lock was taken by A and released by B, this outlines a relation between the two tasks, for instance typical of a wait condition, and it is likely that the critical path should indeed go through B.

A second criterion, *reconvergence*, is based on the existence of a reconvergence to A. When the critical path search follows segment (B1,A2), because A is woken-up by B, but then stays on B for the remainder (B0,B1), this is a sign for concern since B appears otherwise unrelated to A. In that case, the algorithm could backtrack and go back to A just past the (A1,A2) blocking, yielding (A0,A1,X2,B1,A2,A3) as critical path. A closely related third approach, *main task*, is to come back to the task studied as soon as it is not blocked. In that case, it would yield the same result but would avoid scanning B up to its beginning and then backtracking. It is important to note that, in either case, no segment is searched more than once and the complexity remains linear with the graph size.

A fourth criterion, *marked unrelated*, is the manual identification of unrelated tasks. The critical computation would be carried without checking for lock holders or reconvergence, yielding (B0,B1,A2,A3). The user could then mark B as unrelated task and the critical path search would be restarted. When the critical path reaches a task marked as unrelated, it would go back to the previous task at the beginning of the blocking state that caused the task jump. Thus, in that case, when following wake-up link (B1,A2), connecting A to unrelated task B, the critical path would go back to A at A1, the start of blocked segment (A1,A2), yielding critical path (A0,A1,X2,B1,A2,A3).

While the first approach, *lock holder*, is particularly interesting, the problem is to obtain information about lock acquisition, and the lock holder, with low intrusiveness and overhead. When the lock is acquired by B, before contention with A occurs, we should have instrumentation in place to obtain the lock acquisition holder and time. Locking can occur either at

kernel or user-space level. The kernel locking routines could easily be modified to record such information. However, handling user-space locking represents a greater challenge because of fast user-space locking (futex) when the lock is available. This locking mechanism does not perform a system call on locking for performance reasons, and instead uses atomic operations on integers in shared memory. Consequently, the beginning of a critical section and associated information about the lock acquirer is not visible from kernel space. We conclude that it is not possible with the current kernel instrumentation to get the required information for using the *lock holder* approach.

One solution consists in instrumenting user-space locking. On Linux, most programs are using the `pthread` locks from the standard library. Library pre-loading could then be used to overload locking functions. However, this technique does not work for statically linked programs or for programs that would use a custom locking library. Hence, this solution is implementation dependent. This solution would carry a higher performance overhead, increase intrusivenes and defeat the universal approach based solely on kernel instrumentation.

Among the three remaining solutions, the *main task* approach appeared the most intuitive and easiest to grasp for users. This is especially important when such a powerful new analysis tool becomes available. As will be shown in Section 4.5, this approach was already very effective to quickly uncover subtle problems having a significant performance impact on widely used software packages. The other approaches will likely be investigated at a later time. Nonetheless, the other approaches would have little or no effect on the analysis time complexity presented in the experimental results. In the remainder of the article, the critical path approximation, critical path of interest based on the *main task* approach, is detailed and validated.

4.3.2 Critical path approximation

We discuss now a possible approximation of the critical path provided only kernel events are available. For this discussion, we consider horizontal edges as intervals according to time and use interval arithmetic to compare these sets. Neglecting preempted and interrupted intervals, the elapsed time of a task is defined as the set of intervals

$I_e = I_r \cup I_{bi} \cup I_{bd}$

where I_r corresponds to the set of intervals in the running state, I_{bi} to blocking intervals with indirect wake-up and I_{bd} to blocking with direct wake-up. We define the approximation of the critical path as the set of intervals of a task where I_{bd} is substituted recursively by the

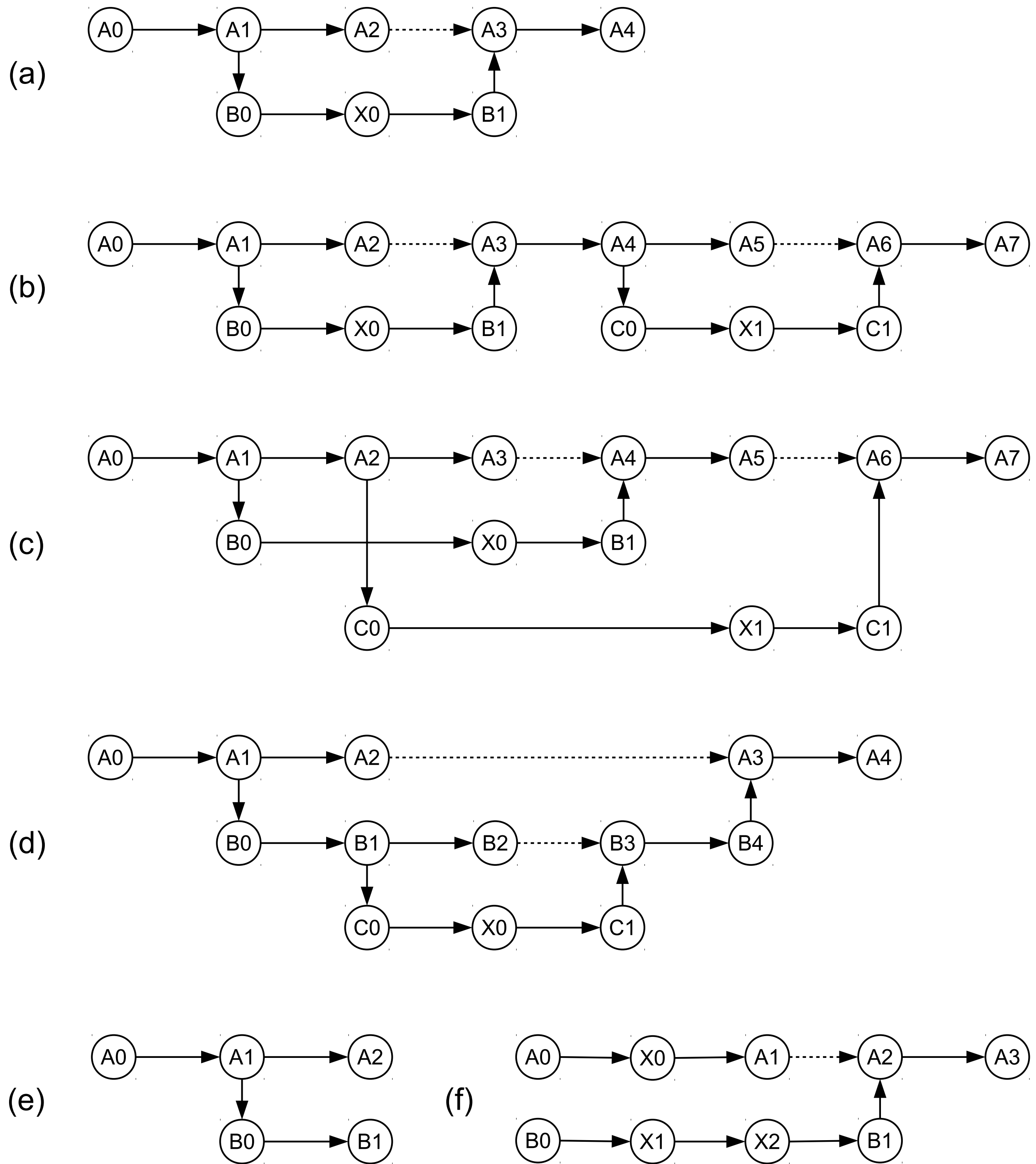


Figure 4.2 Basic execution graphs

Table 4.1 Critical paths of basic execution graphs

Sub-graph	Exact	Approximation	Incorrect	Missing
(a)	A0,A1,B0,B1,A3,A4	A0,A2,X0,B1,A3,A4	(A1,A2)	(B0,X0)
(b)	A0,A1,B0,B1,A3,A4, C0,C1,A6,A7	A0,A2,X0,B1,A3,A4, A5,X1,C1,A6,A7	(A1,A2),(A4,A5)	(B0,X0),(C0,X1)
(c)	A0,A2,C0,C1,A6,A7	A0,A3,X0,B1,A4,A5, X1,C1,A6,A7	(A2,A3),(A4,A5)	(X0,B1),(C0,X1)
(d)	A0,A1,B0,B1,C0,C1, B3,B4,A3,A4	A0,A2,B1,B2,X0,C1, B3,B4,A3,A4	(A1,A2),(B1,B2)	(B0,B1),(C0,X0)
(e)	A0,A2	A0,A2	$\emptyset$	$\emptyset$
(f)	A0,X0,X1,B1,A2,A3	A0,A1,X2,B1,A2,A3	(X0,A1)	(X1,X2)

overlapping interval of the task where the wake-up signal is emitted. The procedure ends when $I_{bd} = \emptyset$. This is essentially the same algorithm as in [20].

We define three sets of intervals to analyse the difference between the critical path I_{exact} and its approximation I_{approx} :

— $I_{good} = I_{exact} \cap I_{approx}$ are correct intervals identified by the approximation
— $I_{incorrect} = I_{approx} \setminus I_{exact}$ are intervals included in the approximation not affecting the completion time
— $I_{missing} = I_{exact} \setminus I_{approx}$ are missing intervals from the approximation

Notice that $I_{incorrect} \triangle I_{missing} = \emptyset$, because each interval in $I_{incorrect}$ has a complement in $I_{missing}$. Thus, the relative error is $\sum I_{incorrect} / \sum I_{exact}$. We also discuss the ability of the approximation to recover the set of tasks that contributes to the critical path.

We computed the difference between horizontal edges of the critical path and the approximation for basic graphs of Figure 4.2. In Graph (a), the difference is related to the edge (A1,A2) representing the execution performed after signalling the sub-task and before blocking. It represents the asynchronous part of the execution. Even programs with a synchronous design do exhibit some level of asynchronous behaviour, because there are always instructions running after the wake-up and before blocking. Unfortunately, there is no upper-bound to the asynchronous level of a task. Continuing with Graph (a), if the edge (A1,A2) is increasing because task A participates to the computation, the relative error of the approximation will

increase, up to the point the where blocking disappears, Graph (e) will be obtained instead, and there is no error. Both tasks are correctly identified as contributing to the critical path.

In Graph (c), the edge (X0,B1) of task B is incorrectly identified by the approximation. However, if waiting for task C is done before task B, then no blocking would occur waiting for task B, the error would be reduced, and task B would not be part of the approximation. In consequence, the wait order does have an effect on the approximation. When multiple results are expected, it is generally a good programming approach to use system calls waiting for any resource to be ready (e.g. `select()` and `wait()`) than to wait for a specific event (e.g. `read()` and `waitpid()`). The pattern of Graph (c) is therefore common in practice. In all cases the last task to finish is identified correctly and this knowledge is important for performance analysis.

For the nested case in Graph (d), all three tasks contributing to the critical path are correctly identified by the approximation provided the blocking window is large enough. The error is concentrated on the first half of the graph, while the last part is error free.

For Graph (f), the case for the barrier is similar to the reasoning made for Graph (a). The approximation is a measure of the unbalance w.r.t. the given task. We now consider the case of lock contention under two assumptions, namely that the critical section duration τ_{cs} is short, and that the blocking duration τ_b is uniformly distributed between $0 < \tau_b < \tau_{cs}$. Then we can state that the average error of the approximation is $\tau_{cs}/2$. The overall relative error will be proportional to the the probability of lock contention and is usually a rare situation for most programs. We conclude that under these assumptions the error of the approximation related to lock contention is low.

We conclude that the approximation presented will produce good results for mostly synchronous programs, that the last task in a group is correctly identified, and that the error related to lock contention should be low for most programs. Interpreting the approximation in the case of asynchronous or parallel applications should take into account the specific aspects of the method.

4.4 Empirical results

We now detail the actual events required from the Linux kernel to build the execution graph and the result of the critical path approximation on actual programs. While the current implementation of the analysis is focused on Linux, the method is general and could apply to other platforms if equivalent events are available. The event tracing for Windows [56] and DTrace [57] are examples of tracers for other operating systems.

Built-in static tracepoints of Linux are used as instrumentation. The minimal set of events and related fields required are listed in Table 4.2. The Linux Tracing Toolkit next generation (LTTng) is used to record the trace on disk. We selected this tracer for its high performance characteristics [10].

The `sched_switch` event indicates that thread `prev_tid` is replaced by thread `next_tid` on the CPU where the event occurs. The `prev_state` is the state `prev_tid` is going into. Values for this field are defined in file `sched.h` of the Linux source tree. Value 0 corresponds to `TASK_RUNNING`, thus we can conclude that `prev_tid` is preempted. If the value is greater than 0, it means that the task is either in `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state, corresponding to the blocking state.

The `sched_wakeup` occurs when the task `tid` is inserted into the run queue of the scheduler. Event `sched_wakeup_new` is similar, but occurs when the task is inserted for the first time in the run queue, usually upon creation after a call to `clone()`. It means that the task is runnable, but waits to be selected by the scheduler. It thus corresponds to the preempted state.

Three kinds of interrupts are traced: hardware interrupt (or IRQ), high-resolution timers (which is a type of hardware interrupt), and software interrupt (SoftIRQ, also called deferred interrupt or bottom-half [70]). Fields `irq` and `vec` indicate the type of interrupt. This information is used to categorize the wait time, along with the device involved in case the interrupt triggers a wake-up.

Often, the IRQ handler raises a SoftIRQ and returns. Immediately after, pending SoftIRQs are processed. The maximum number of consequent SoftIRQ is fixed to avoid starvation of the interrupted task and is typically fixed to 10. The per-cpu kernel thread `ksoftirqd` is woken-up if there are more SoftIRQs to handle. Of course, a task may wait for the processing of a given SoftIRQ by the `ksoftirqd` thread, and our analysis is able to detect it.

The graph construction algorithm from the Linux trace is detailed in Algorithm 3. It demultiplexes trace inputs according to the canonical task model. Error handling is not shown for simplicity. The $TASK$ set contains tasks of the system when tracing began. The array CPU stores the running task for each CPU. The CTX variable is a per-CPU array of stacks used to track interrupt nesting. The main procedure processes each event according to its type. The `sched_switch` event (lines 11-14) creates one vertex for each task involved. The task `prev_tid` was running and the edge label is set accordingly. The state of `next_tid` was preempted because it was in the running queue but not running. In the case of `sched_wakeup` (lines 15-25), a new vertex is appended to the target task. If the wake-up occurs inside an interrupt, then the edge label is assigned according to the type of interrupt. If no interrupt

Table 4.2 Required kernel tracepoints

Events	Fields
sched_switch	prev_tid, prev_state, next_tid
sched_wakeup	tid, target_cpu
sched_wakeup_new	tid, target_cpu
irq_handler_entry	irq, name
irq_handler_exit	irq, ret
hrtimer_expire_entry	hrtimer, now, function
hrtimer_expire_exit	hrtimer
softirq_entry	vec
softirq_exit	vec

is executing, it means that the wake-up comes from the current task and that the target task was blocked. A vertex is appended to the current task and its state is running. A vertical edge is created from the tail of the current task to the tail of target task. Interrupt related events (lines 27-31) push or pop CPU context. No vertex is created for tracking interrupt entry and exit in order to reduce the graph size. If a more detailed graph is desired, these vertices can be trivially added.

We illustrate the two main patterns that the graph encodes, i.e. a direct and indirect wake-up. In Figure 4.3, a task blocks while waiting for the sub-task signal. Events in Table (a) produce state changes shown in (b) and the resulting graph is shown in (c). Three new vertices are created and indicate that the task was waiting 10 units of time for the task to finish, and 20 units of time are spent in the preempted state before the task is scheduled. The example in Figure 4.4 shows the resulting graph in the case where a task is awaken from interrupt context.

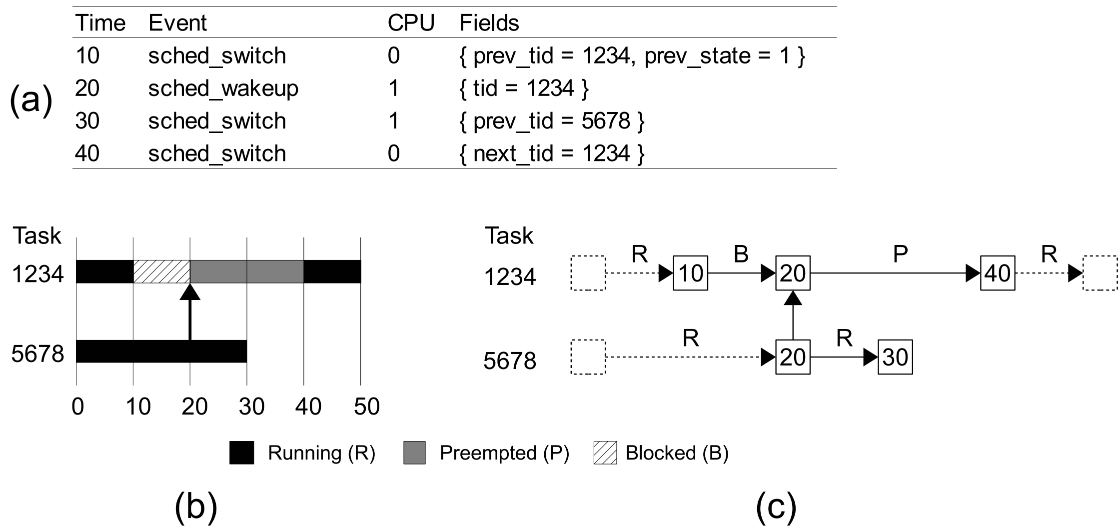
4.4.1 Approximation algorithm

We now present the algorithm to traverse the graph and build the approximation of the critical path presented in Section 4.3.2.

Algorithm 2 works by iterating from a start vertex. If a blocking edge is encountered, the incoming edge representing the wake-up is followed backward. The blocking interval is accounted to the sub-task. If the sub-task itself blocks, then this process is applied recursively. The iteration on the main task continues when the blocking interval is completely computed.

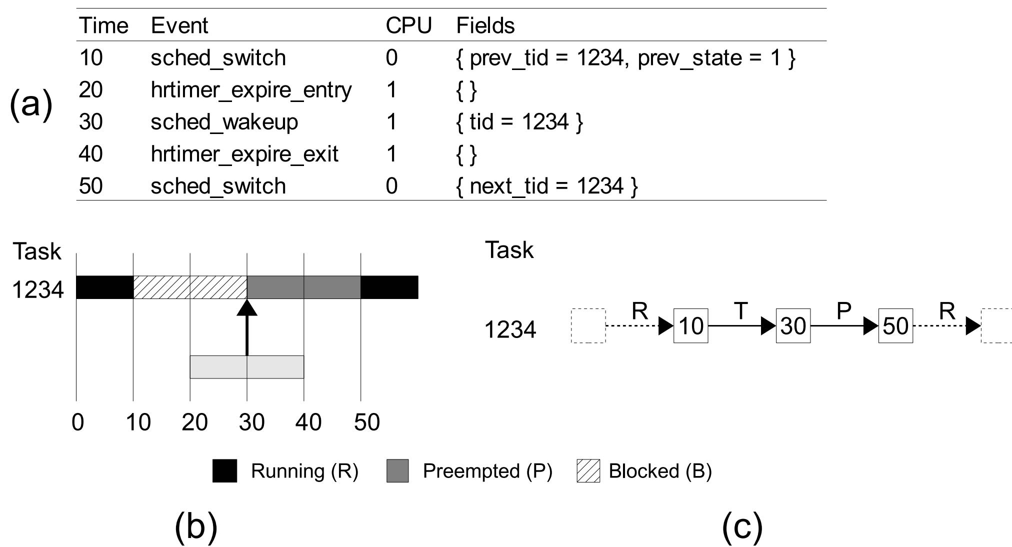
Algorithm 1 Execution graph construction from trace

```
Input: trace T
Output: execution graph G
 1: TASK ← {initial tasks}                                        ▷ Declarations
 2: CPU ← ∅
 3: CTX ← ∅
 4: for all task t ∈ TASK do                                      ▷ Initialization
 5:     if t.state is running then
 6:         CPU[t.cpu] ← t
 7:     end if
 8:     Create initial vertex of task t with timestamp t.begin
 9: end for
10: for all event e ∈ T do                                        ▷ Main procedure
11:     if e.type is sched_switch then
12:         LINK_HORIZONTAL(e.prev_tid, e.ts, running)
13:         LINK_HORIZONTAL(e.next_tid, e.ts, preempted)
14:         CPU[e.cpu] ← e.next_tid
15:     else if e.type is sched_wakeup or
16:           sched_wakeup_new then
17:         interrupt ← Peek CTX[e.cpu]
18:         if interrupt is not null then
19:             LINK_HORIZONTAL(e.tid, t.ts,
20:               labelof(interrupt))
21:         else
22:             v1 ← LINK_HORIZONTAL(e.tid, e.ts, blocked)
23:             v2 ← LINK_HORIZONTAL(CPU[e.cpu], e.ts,
24:                     running)
25:             LINK_VERTICAL(v1, v_images2)
26:         end if
27:     else if e.type is interrupt entry then
28:         Push e to CTX[e.cpu]
29:     else if e.type is interrupt exit then
30:         Pop from CTX[e.cpu]
31:     end if
32: end for
33: function LINK_HORIZONTAL(task, ts, l)                         ▷ Utilities
34:     tail ← last vertex of task from G
35:     Create vertex v with timestamp ts
36:     Create edge tail[right] → v[left] with label l
37:     return v
38: end function
39: function LINK_VERTICAL(from, to, l)
40:     Create edge from[up] → to[down] with label l
41: end function
```



Time	Event	CPU	Fields
10	sched_switch	0	{ prev_tid = 1234, prev_state = 1 }
20	sched_wakeup	1	{ tid = 1234 }
30	sched_switch	1	{ prev_tid = 5678 }
40	sched_switch	0	{ next_tid = 1234 }

Figure 4.3 Example of wake-up event from sub-task



Time	Event	CPU	Fields
10	sched_switch	0	{ prev_tid = 1234, prev_state = 1 }
20	hrtimer_expire_entry	1	{ }
30	sched_wakeup	1	{ tid = 1234 }
40	hrtimer_expire_exit	1	{ }
50	sched_switch	0	{ next_tid = 1234 }

Figure 4.4 Example of wake-up event in timer interrupt

The complexity to build and the resulting size of the execution graph is proportional to the number of events in the trace. The critical path approximation algorithm traverse the whole graph in the worst case $O(|V| + |E|)$. The computed path is a linked-list where we assume append and prepend operations works in constant time. As a result, the overall complexity $O(n)$ is linear according to the trace size.

Algorithm 2 Critical path approximation

Input: execution graph G, task T, v_s, v_e
Output: path P

```
 1: v ← v_s
 2: while v is not v_e do                          ▷ Forward iteration
 3:     E ← v.right
 4:     append PROCESS(E) to P
 5:     v ← E.to
 6: end while
 7: function PROCESS(edge)
 8:     if edge.label is blocking and
 9:          edge.to has incoming vertical edge then
10:         return RESOLVE(edge)
11:     else
12:         return E
13:     end if
14: end function
15: function RESOLVE(edge)
16:     TMP ← ∅
17:     v ← edge.to.down.from                      ▷ Wake-up vertex
18:     while v.ts > edge.from.ts do               ▷ Backward iteration
19:         E ← v.left
20:         prepend PROCESS(E) to TMP
21:         v ← E.from
22:     end while
23:     return TMP
24: end function
```

4.5 Evaluation

The analyser is implemented in Java as a plug-in extension to the Eclipse Linux tools project. To validate the correctness of results, we designed a set of workloads that present specific runtime behaviours and compared the output of the analysis with the expected result. This

workload-kit is now freely available [2]. We describe two representative examples to illustrate analysis results. We then proceed to describe a use case with an existing widely used application `APT`. We conclude with a study of the tracing runtime cost according to the type of load.

4.5.1 Environment

Results were collected on Ubuntu Linux 13.04, running the default 64-bit kernel 3.8.0 and with LTTng 2.1. The workstation is running an Intel i7-3770, featuring 4 hyperthreaded cores for a total of 8 virtual CPUs. The host has 8 GiB of memory. Traces are written to the local hard-drive. We configured system parameters with large tracing buffers to prevent event losses.

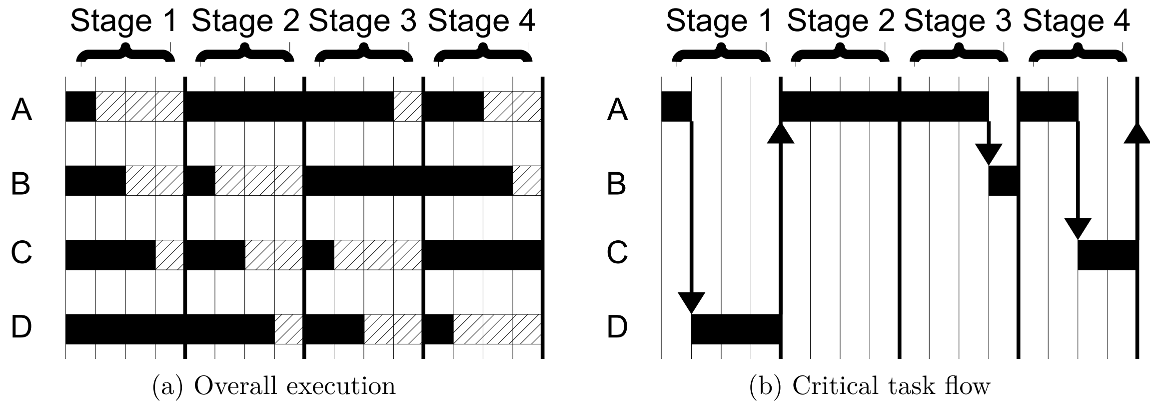
4.5.2 Uneven Parallel Computation

Speedup of parallel applications is limited by the serial part of the computation. The program `wk-imbalance` simulates a parallel computation with four cooperating threads with uneven work distribution, affecting the speedup obtained. The computation consists of cycles of 4 stages, where each stage is synchronized with `pthread_barrier()`. The work is distributed, with the duration increasing with the rank of the thread, and changes in a round-robin fashion at each stage. With this scheme, each thread should be on the critical task flow once. Figure 4.5 displays one cycle of the execution (a) and the corresponding critical task flow (b) w.r.t. task A. Table 4.3 compares expected and actual experimental results for 1000 cycles run. The expected and experimental results are within 1% of each other. A small variation in experimental results is expected and is attributed to the tracing overhead, microarchitecture pipeline conditions, scheduling latency, interrupt handlers and other background tasks.

2. http://github.com/giraldeau/workload-kit

Table 4.3 Percentage of execution time spent in each task on the critical path for the program `wk-imbalance`

Task	Expected	Experiment
A	62.5%	63.1%
B	6.3%	5.7%
C	12.5%	12.3%
D	18.8%	18.9%



(a) Overall execution (b) Critical task flow

Figure 4.5 Execution of `wk-imbalance`

4.5.3 Block Device I/O

We wanted to evaluate the ability of our method to compute correctly the time spent in block device writes. In particular, we wanted to observe synchronous I/O, which is usually much slower than asynchronous flushes of dirty pages. The utility `wk-ioburst` simply writes zeros in a file with the `O_SYNC` flag. This flag causes writes to return only when data is effectively written on the device.

Results are shown in Table 4.4. Most of the time is spent waiting for the `jbd2/sda1-8` thread. According to Linux documentation, this thread is responsible for the filesystem journaling. An inspection of the critical task flow view, shown in Figure 4.6, confirms the results' correctness.

A small portion of the critical flow is related to `lttng-consumerd`, the daemon in charge of writing the kernel trace to disk. This result was not expected at first, because there is no explicit relation between the trace consumer and the traced application. To understand why these threads wait for each other, we recorded the call chain of the kernel on each wake-up. We found that the function `journal_end_bu-ffer_io_sync()` locks a buffer for a short

Table 4.4 Critical task flow of `wk-ioburst`

Task	% Time
wk-ioburst	29.5%
jbd2/sda1-8	70.4%
lttng-consumerd	0.1%

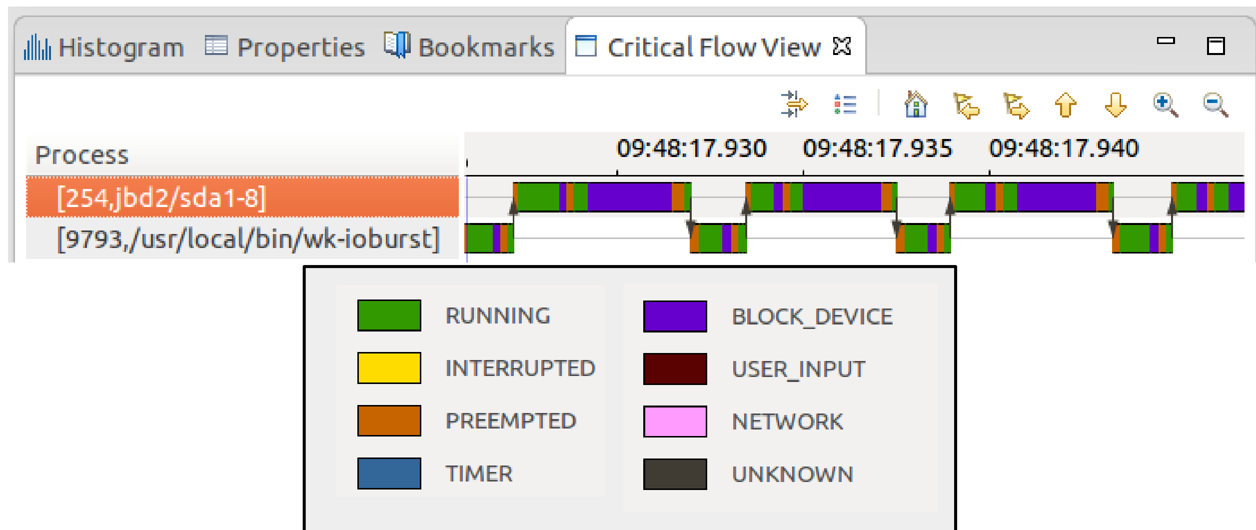


Figure 4.6 Time line view of the critical task flow of `wk-ioburst`

period of time and this was causing the unexpected switch in the critical task flow. In fact, this situation can potentially occur almost randomly between any processes sharing a global resource, such as a block device. Indeed, the contention for the lock actually delays our application.

4.5.4 Use Case

The Advanced Packaging Tool (APT) is a core component of many popular Linux distributions. This software is itself a front end for `dpkg`, which in turn executes numerous scripts to perform its operations. It involves network and disk I/O for the download and installation of packages, in addition to CPU load to resolve package dependencies and to decompress files. Consequently, retrieving the critical task flow of this application is nontrivial. The scenario we traced consists in the installation of the package `tree`. The package cache was cleaned to force the package to be downloaded, and we specified the option for non-interactive installation.

The installation takes about 7 seconds to complete wherein blocked states account for 37% of the elapsed time. The Figure 4.7 shows the critical flow view of the trace, where each processing step of `APT` is visible according to time. In addition to performance numbers, this view acts as a reverse engineering tool for understanding the execution structure of arbitrary related processes. At label (1) of Figure 4.7 the `apt-get` process is mostly computing in user-space. Running the program under a conventional profiler would focus on this part. At label (2) we observe interactions with the `dpkg` back end. At label (3) a high arrow density

is visible and it requires zooming to display its details. It is related to the installation of man pages, that triggers the creation of 543 `mandb` processes, for a total of 665 milliseconds. The execution suggests that the overhead of spawning new processes is high compared to the small amount of work done. Finally, we notice at label (4) two sleeps of 500 milliseconds each, at the end of the `apt-get` execution. We repeated the experiment and the number of sleeps is not constant, and thus suggests that the triggering condition is non-deterministic.

To fix performance issues, a call chain is required to locate the source code causing this behaviour. There are two main methods to recover the call chain in ELF executables. The first is based on frame pointers iteration. However, executable binaries on many systems are compiled by default without frame pointers, and therefore call chain recovery from frame pointers is not reliable. Recompiling all programs with frame pointers is possible, but not very practical. The other method is based on stack unwinding using static frame information tables, a feature used to handle exceptions in C++, that works with unmodified binaries. Offline unwinding consists in saving all registers and a large portion of the stack. Then, the instruction pointer is recovered for each stack frame from the information in the `.eh_frame` section of the ELF executable. Memory mappings of libraries are also required for symbol resolution. We observed that offline unwinding is an order of magnitude slower than frame pointers.

We used the `perf` utility to record the call chain from the kernel using the unwind method. Figure 4.8 shows the location associated with the inefficiency of `mandb` observed previously in Figure 4.7 (3). The problem is related to the inner working of `libpipeline.so`. In this case, the performance issue could be addressed by an architecture modification, such as using a thread pool. We used the same technique to locate the source code related to the sleeps in Figure 4.7 (4). The method `DoTerminalPty()` calls `nanosleep()` when reading the file descriptor `master` returns the `EIO` error. The comment in the source code above the sleep suggests that a race condition may occur in the caller if the child is about to exit. However, there is no guarantee the child has actually exited when the sleep completes, because it can be preempted for an undefined period of time while exiting (the Linux kernel is fully preemptive). In consequence, this is both a performance bug and a race, and proper synchronization and error handling in the caller should be used instead.

4.5.5 Tracing Cost

The tracing overhead has two sources: each event produced incurs additional instructions in the code path and the consumer daemon must write the trace from memory buffers to the disk. The activity of these daemons is also part of the trace, such that there is a small

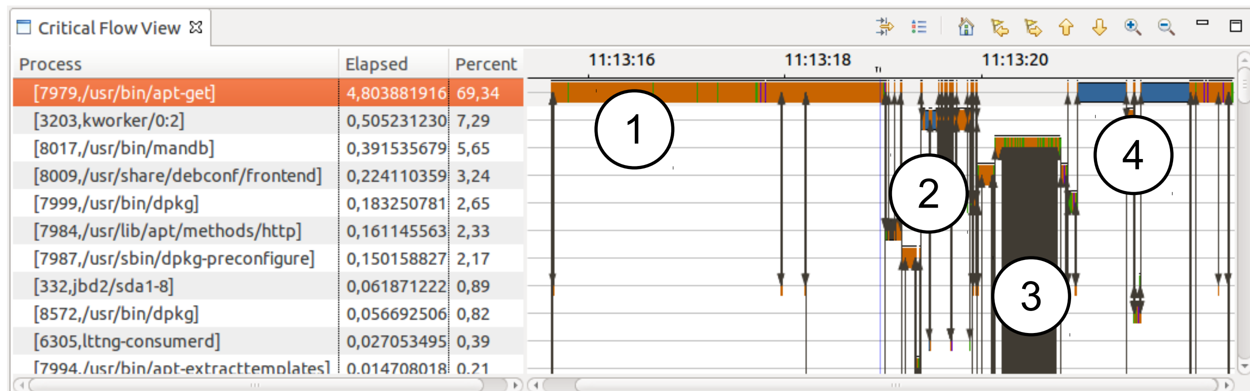


Figure 4.7 Overview of the APT critical task flow

Frame	Function	Code location
0	__execve	libc-2.17.so
1	__execvpe	libc-2.17.so
2	pipecmd_exec	libpipeline.so.1.2.2
3	[unknown]	mandb
4	pipecmd_exec	libpipeline.so.1.2.2
5	pipeline_start	libpipeline.so.1.2.2
6	find_name	mandb
7	test_manfile	mandb
8	testmandirs	mandb
9	create_db	mandb
10	mandb	mandb
11	process_manpath	mandb
12	main	mandb
13	__libc_start_main	libc-2.17.so
14	_start	mandb

Figure 4.8 Call chain of mandb locating inefficient process execution

but positive feedback of event production. Our objective is to measure empirically the global overhead and storage requirements associated with recording to disk the events needed by the critical task flow analysis. Our method does not require system call information. Therefore we wanted to measure the benefit associated with disabling system call tracing. Finally, we evaluated whether using an external USB drive for trace recording is suitable. An external drive represents a low-cost and convenient device for trace recording.

The experiment contains three benchmarks having different workload profiles. The first load is computation intensive, the second generates disk I/O and the third is a mixed load involving database queries on `MySQL`. Benchmarks are from the utility `sysbench`. Each experiment runs for one minute and is repeated ten times. One warm-up run is performed before experiments. Table 4.5 shows the results regarding the average overhead and the resulting trace size.

We observe that the tracing overhead for the CPU benchmark is less than 1% for all configurations. The overhead is low because the program runs mostly in user-space, and the scheduling event rate is limited by the timer tick, which is set to 1 kHz on the workstation.

The I/O benchmark shows overhead of the order of 50% when the trace is written to the internal drive. The slowdown is mostly due to seeks between the trace and the load that affects the effective bandwidth of the drive. When the trace is recorded to the external drive, the measured overhead is close to zero. The I/O occurs independently on each devices and reduces seek time. The application mainly waits for the disk, thus limiting the rate of operations. Consequently the resulting trace size is small.

The performance impact of system call tracing is clearly visible for the `MySQL` benchmark. Disabling system call tracing reduces the overhead by about 9% for both internal and external drive setup. The trace is smaller by a factor of 1.8 compared to when system calls are enabled. The trace size is larger than for other experiments. The data production rate reaches 37.7 Mb/s with system calls and 20.5 Mb/s without. Therefore, the disk throughput should be taken into account to avoid event losses in steady state.

To further understand the overhead source for the `MySQL` benchmark, we computed the proportion of each event type. When system calls are enabled, they account for 64% of events, while scheduling and interrupt events represents 33% and 3% respectively. Without system calls, we found that scheduling events represents 91% of all events, and that interrupts account for 9%.

Following the fact that most events are related to scheduling, we conducted an experiment to determine the upper bound of tracing overhead of these events in the code path. By running two threads that signal each other in turn using two semaphores, one forces the

scheduler to alternate between the two tasks at the fastest rate possible. We repeated the cycle one million times. The rate of context switches per-core was reduced from 1.55×10^5 s^{-1} to 1.38×10^5 s^{-1}, equivalent to 11%. Each iteration includes two `sched_switch` and one `sched_wakeup`. Considering an average running period per cycle of 4.85 μs when tracing is enabled, the cost of each event is 177 ns.

4.6 Future Work

Lock contention is causing a challenge to recover the exact critical path, but the approximation is generally precise for this situation. It may be possible to use the classical critical path algorithm for most of the processing, and switch to the approximation only when required. This heuristic would bring the best of both algorithms.

The memory scalability of the algorithm may be concern for very large traces. One solution consists in computing the statistics of the critical task flow continuously and discard older graph nodes that will not be accessed anymore. This may be interesting for batch processing, but is unsuitable for an interactive viewer. The solution in this case could be to adapt the state history tree [22] for the purpose of storing the graph data. It has been successfully used to store state history of processes for the control flow viewer.

The concept of request (such as RPC) does not belong to the operating system domain, and interpreting results must be done accordingly. For example, if a server reorders or merges requests, there is no way with the current implementation to isolate the processing of a request, and no assumptions are made in this area. However, the result does represent the complete processing from a system-level point of view, and thus has an intrinsic value to understand elapsed time. The system should process only one request to isolate the processing related to it, which is suitable for application development. In particular, it may be possible to compare critical task flow to detect changes in application performance [71].

4.7 Conclusion

We demonstrated the ability of our approach to recover runtime behaviour of complex applications on multi-core architectures at the system-level. In particular, our tool was able to very quickly find a major performance problem in a complex but widely used application, `APT`. The approach described here fills the need for a general latency analysis tool as well as runtime execution reverse engineering tool. Neither user-space applications nor the kernel required additional modification. We described how to build the execution graph from a kernel trace and to compute the critical task flow. We demonstrated the usefulness of our

Table 4.5 Results of sysbench tracing experiments, with and without tracing of system call events, and with a single internal disk (Int.) or with an added external disk (Ext.)

		Overhead		Trace size (MB)	
		Int.	Ext.	Int.	Ext.
CPU	w/ sys.	0.5%	0.5%	4.0	10.4
	w/o sys.	0.3%	0.4%	3.4	5.9
I/O	w/ sys.	51.9%	-0.1%	13.0	34.4
	w/o sys.	48.0%	-0.1%	8.3	16.4
MySQL	w/ sys.	11.3%	16.2%	2261.0	2130.0
	w/o sys.	2.6%	7.4%	1227.1	1167.1

approach to analyze the performance of actual complex programs. We measured the runtime tracing cost for various workloads and analyzed the impact of using an external drive for trace recording. We found that an external drive is beneficial in case of high I/O load, and that avoiding system call tracing for mixed load reduces overhead drastically.

4.8 Acknowledgements

This work was made possible thanks to the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Geneviève Bastien for the TMF view, Philippe Doucet Beaupré for additional reviews, Matthew Khouzam and Alexandre Montplaisir-Goncalves for their help regarding Eclipse, and finally Mathieu Desnoyers and Lothar Wengerek for their precious advice.

CHAPITRE 5 ARTICLE 2 : Host-based method to recover wait causes in distributed systems

5.1 Abstract

We describe a new class of profiler for distributed and heterogeneous systems. In these systems, a task may wait for the result of another task, either locally or remotely. Such wait dependencies are invisible to instruction profilers. We propose a host-based, precise method to recover recursively wait causes across machines, using blocking as the fundamental mechanism to detect changes in the control flow. It relies solely on operating system events, namely scheduling, interrupts and network events. It is therefore capable of observing kernel threads interactions and achieves user-space runtime independence. Given a task, the algorithm its active path from the trace, which is presented in an interactive viewer for inspection. We validated our new method with workloads representing major architecture and operating conditions found in distributed programs. We then used our method to analyze the execution behavior of five different distributed systems. We found that the worst case tracing overhead for a distributed application is 18 percent, and that the typical average overhead is about 5 percent. The analysis implementation has linear runtime according to the the trace size.

5.2 Introduction

A distributed and heterogeneous system is a set of threads running on multiple computers, and implemented in various programming languages. The hidden nature of the processing and the incompatibilities between runtime environments makes the task of performance profiling and debugging more difficult. Our goal is understanding the elapsed time of a computation in such systems to improve the response time and to diagnose performance problems.

Popular profilers based on hardware counters sampling [59], [63], dynamic binary translation [72] and call-graph elapsed time are useful to identify code hot spots, but are limited in two ways. Firstly, instruction profilers do not take into account the time spent waiting. Secondly, they are restricted to the local host, which limits their use for distributed applications. Hence, the performance of each component must be analyzed independently.

Previous work considered instrumentation of libraries and middleware to monitor performance of specific distributed systems [29], [33], [36], [41], [45], [50], [67]. The resulting instrumentation provides straightforward performance measures, but is domain dependent and tied to a runtime environment. Considering the large number of languages, components

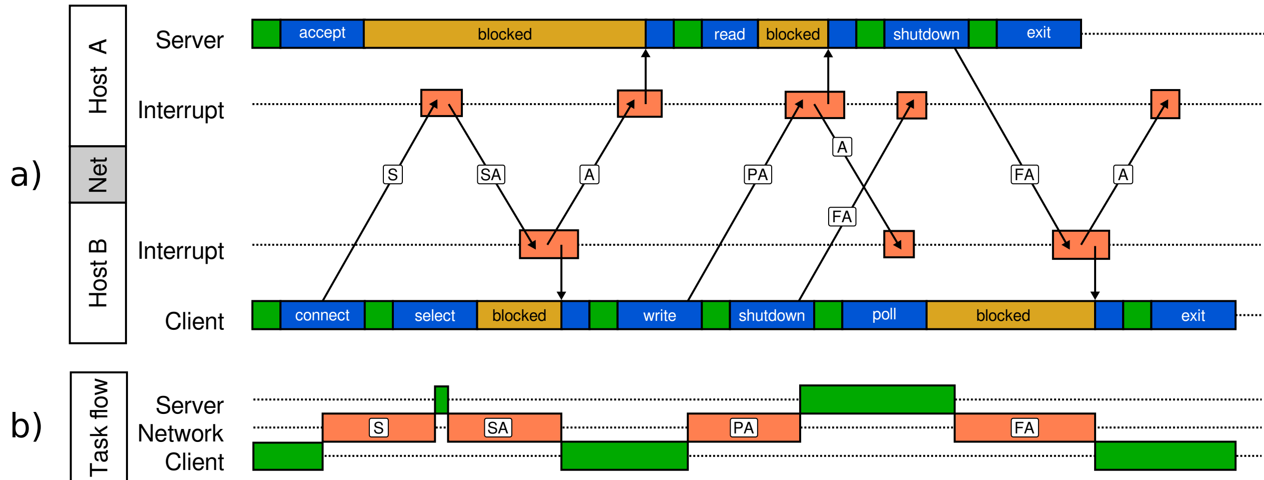


Figure 5.1 Task state and TCP packet transmission

and frameworks, the cost of instrumenting each of them is high. System call tracing was proposed as a less invasive instrumentation technique [20], [23], [44]. The request processing path of a distributed application is recovered by recording send and receive operations. These system calls are not sufficient in general, because communication can occur from any kernel code, such as other system calls, interrupt contexts and kernel threads. A technique based on recording network traffic was proposed to characterize the elapsed time between the client, the server and the network [64]. However, internal processing of endpoint machines is not visible using network events only.

Kernel tracing allows the wait occurring between threads to be observed. It works with unmodified executables and is system-wide, two properties important for actual heterogeneous distributed systems. By carefully choosing the instrumentation, low overhead and disturbance can be achieved. This paper aims to study methods providing meaningful representation of a broad range of actual distributed applications execution using kernel traces. The contributions are as follows :

- The design of the kernel instrumentation required for the analysis. The implementation is available as Linux loadable modules and works with an unmodified Linux kernel.
- A graph model of the system execution generated from the trace and the corresponding algorithm to extract the active path of a given task.
- Experiments on actual software to study the program behavior with regards to wait, according to host type, software architecture and network conditions.

— Evaluation of the analysis cost of the runtime overhead and the trace processing.

Note that our goal is not to recover the flow of requests in specific application protocols. This knowledge is outside the domain of the operating system, and we consider data payload in network packets as a black-box. Also, because the method recovers the active path at the system level, if the runtime environment multiplexes processing, additional user-space instrumentation may be required to untangle the data.

The next section describes in detail the instrumentation and the analysis algorithms.

5.3 Analysis Architecture

For our discussion, we consider that a task (or a thread) on a computer can be in four canonical states, namely `running` on a given CPU, `preempted` when ready but not executing, `interrupted` when an interrupt handler nests over the application code, and `blocked` when the task passively waits for an event and yields the CPU. All states other than `running` prevent the program from making progress and should be reduced whenever possible.

Interrupts can be trivially monitored by recording handler entry and exit. Simple statistics can be computed from these events, such as frequency and duration. Tracking interrupts allows us to identify if an event is emitted from task or interrupt context.

Preemption mostly occurs when processors must be shared between tasks. The scheduler switches the running task on a given CPU when its quantum expires. The cause of the preemption can be assigned to the tasks running on the corresponding CPU, because they affect the completion time of the preempted task. Preemption also occurs between the time a task becomes ready, after blocking, and the time it effectively executes.

Unlike other type of waits, `blocking` changes the control flow of the program. The behavior depends on the structure of the application. A task going to the blocked state is moved from the run queue to the wait queue and then the scheduler is invoked to yield the CPU. The key idea is that the event unblocking the task (hereafter referred to as the wake-up event) indicates the cause of the wait, which is unknown *a priori* and is non-deterministic in general. We distinguish two types of blockings, when the wake-up occurs from another task in kernel mode, or from an interrupt. Task wake-up examples are contention on a mutex, empty or full pipe conditions, and other inter-process communication. By improving the performance of the sub-task, the wait of the main task is reduced. In contrast, when the wake-up comes from interrupt context, the interrupt vector indicates the device upon which the task was waiting, for instance, a timer or a disk. In particular, programs waiting for an incoming network packet are generally woken-up from a network interrupt. By tracking the source of

the packet related to the interruption, we can identify the emitter task. The reasoning is that if the network was faster or if the task sent the packet earlier, then the blocking time of the receiver would have been reduced.

Consider for example the system call `select()`, that returns either because a file descriptor is ready or the specified timeout occurs. If data is written to the file by a local task, the wake-up source indicates which thread made the write. If the wake-up comes instead from the timer interrupt, it indicates that the timeout occurred. If the wake-up comes from the network interrupt, it means a remote task sent a message over a socket. Therefore, this mechanism is independent of the system call, its parameters or its return value.

One limitation of this approach is related to active waiting, such as spinlock and polling used in low-latency applications. Busy wait in user-space is not visible from the operating system. For distributed application, the network delays are usually an order of magnitude greater than the CPU speed, therefore it is reasonable to assume that most applications are blocking during the processing for efficiency.

We review briefly the behavior of the Linux operating system regarding the task states, the interruption context and the network exchanges, which are the foundation for the active path analysis. Packet transmission and reception always occur in kernel mode, either from a system call (such as `send()` or `write()`), or a deferred interruption (hereafter referred to as softirq). The reception is done asynchronously inside softirq, and then any task waiting for the data is awakened. Packet transmission also occurs from the softirq context. We observed that TCP control packets are sent immediately after packet reception, and that TCP retransmissions are sent from the timer softirq. To prevent user-space starvation due to high frequency softirq, the processing is deferred to the `ksoftirqd` kernel thread.

Figure 5.1 (a) shows the execution according to time of `netcat` transmitting a short string. The elapsed time is adjusted for proper display. The client establishes a TCP connection to the server, sends a short string and closes the connection. Eight messages are exchanged between the client and the server. The server blocks in `accept()` for an incoming connection. The client sends the synchronize packet and the server host acknowledges it directly from the interrupt handler, without the intervention of the server task. The client blocks in `select()` for the connection to be established. The server is then awakened when the handshake is completed. The client writes the data to the socket, closes the connection, and waits for the connection termination in `poll()`. The server blocks while the data is transmitted in `read()`, and finally closes the connection, which unblocks and terminates the client.

The execution path affecting the completion time is identified by following backward the source of the wake-up. The result w.r.t. the `netcat` client is shown in Figure 5.1 (b).

The last `poll()` is unblocked by the `FIN-ACK` packet from the server, sent when the server calls `shutdown()`. When reaching the blocking in `read()`, the `SYN-ACK` is followed, and the corresponding `write()` of the client is attained. Finally, the blocking in `select()` is resolved by traversing the `SYN-ACK` and `SYN` packets.

The events required for the analysis and the instrumentation method are shown in Table 5.1. The instrumentation is implemented as loadable kernel modules. We use three instrumentation methods. The first method consists in adding a probe to an existing static tracepoint in the kernel. All scheduler and interrupt events are recoded in this way, except for `sched_wakeup`. To reduce the wake-up latency, the tracepoint is executed on the destination CPU inside inter-processor interrupt (IPI). This has the effect of losing the source of the wake-up. We therefore use a kprobe hook to the function `try_to_wake_up()`, which is called before sending the IPI. The third mechanism uses netfilter to register a hook for recording TCP packet headers.

5.3.1 Trace Synchronization

One challenge of distributed event analysis is the absence of a global clock. The analysis is sensitive to message inversion, and requires partial order on network events. The convex hull synchronization algorithm using network packets has this property [15]. The algorithm produces a linear clock relation between two hosts, that models the clock drift and offset. The final transform for a given clock is composed relative to a reference host, computed from the synchronization graph.

The convex hull algorithm works as follows. For each pair of hosts, an x-y plane is built with two sets of points, either incoming and outgoing, w.r.t. a reference host. The convex hull of these two sets is approximated using minimum and maximum slopes dividing the two regions. The bisector of the the two slopes is taken as the final approximation. The minimum convex hull requires at least two points in each set. The algorithm fails if a slope intersects a convex hull. This situation can occur for lengthy traces because of the physical non-linear properties of crystal clock oscillators.

We define the following relations for the linear timestamp transformation.

- function : $f(t) = mt + b$
- inverse : $f^{-1}(t) = t/m - b$
- compose : $f(g(t)) = f(t) \circ g(t) = m_1 m_2 t + m_1 b_2 + b_1$ with $f(t) = m_1 t + b_1, g(t) = m_2 t + b_2$
- identity : $f(f^{-1}(t)) = I(t) = 1t + 0 = t$

Table 5.1 Kernel events required for the analysis

category	event	method
scheduler	sched_ttwu	kprobe
scheduler	sched_switch	tracepoint
interrupt	hrtimer_expire_entry	tracepoint
interrupt	hrtimer_expire_exit	tracepoint
interrupt	irq_handler_entry	tracepoint
interrupt	irq_handler_exit	tracepoint
interrupt	softirq_entry	tracepoint
interrupt	softirq_exit	tracepoint
network	inet_sock_local_in	netfilter
network	inet_sock_local_out	netfilter

When more than two traces are synchronized, there must be a transitive transform between a reference host and every other host. We compute such transform using a directed graph where vertices represent hosts and where edges represent timestamp transforms. If the graph is connected, then a global time can be recovered. The graph is built by adding two edges for each transform, namely a forward edge representing the computed transform and a reverse edge with the inverse transform. The transitive transform is obtained by composing transforms for the path from the reference host to the peer host. A composed transform does not guard from inversion, and the partial order is not guaranteed. In practice, the composed error margin is lower than the network transmission, and no inversion occurs.

As an example, consider the synchronization of traces from three computers q_0, q_1 and q_2, where packet exchanges occurred between (q_0, q_1) and (q_1, q_2) and the transform $f(t)$ and $g(t)$ respectively for these two pairs of hosts, obtained using the convex hull method. The resulting graph and each transform according to the reference computer are shown in Figure 5.2 and 5.3 respectively.

5.3.2 Trace Analysis

The recovery of wait causes needs efficient navigation between related events. We achieve this with a directed acyclic graph (DAG) built from the synchronized traces. Then, the wait cause is recovered by traversing the graph backward.

More formally, we define an *execution graph* data structure as a two dimensional doubly linked list, where horizontal edges are labeled with task states, and where vertical edges are

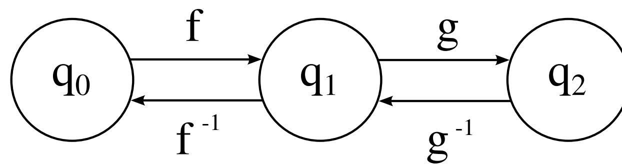


Figure 5.2 Example of a synchronization graph between three hosts.

Reference	Transform
q_0	$q_0 : I(t)$ $q_0 \rightarrow q_1 : f(t)$ $q_0 \rightarrow q_2 : f(t) \circ g(t)$
q_1	$q_1 : I(t)$ $q_1 \rightarrow q_0 : f^{-1}(t)$ $q_1 \rightarrow q_2 : g(t)$
q_2	$q_2 : I(t)$ $q_2 \rightarrow q_1 : g^{-1}(t)$ $q_2 \rightarrow q_0 : g^{-1}(t) \circ f^{-1}(t)$

Figure 5.3 Resulting timestamp transforms according to the reference host for the synchronization graph example of Figure 5.2.

signals between tasks (either a wake-up or a network packet). A vertex v represents an event and has a timestamp t, representing causality, such that every edge $v \rightarrow v'$ must satisfy $t \leq t'$. Algorithm 3 details the trace to graph transformation. For simplicity, error handling is not shown and a data structure representing the machine state is implicitly defined per-host.

The algorithm iterates over trace events, and processes them according to their type. The `sched_switch` event adds two new edges to the graph, one for the previous task that was running, and the other for the next task that was preempted prior to run. The `sched_ttwu` event adds an edge to represent the blocking state of the target task, while the task emitting the signal is necessarily running. Notice that the source may be either a thread or a per-CPU place-holder thread representing the interrupt context. Finally, a vertical edge is added from the source to the target with the wake-up label. The events `interrupt_entry` and `interrupt_exit` are managing the corresponding place-holder thread stack to account for nested interrupts. The network events are processed as follows. On transmission, a new vertex is added to the emitter task. This new vertex and its packet are added to the unmatched packet set. When the corresponding packet is found, a new vertex is created on the receiver, and a vertical edge with the label network is created from transmission to reception vertices.

We define the *active path of execution* as the execution path where all blocking edges are substituted by their corresponding sub-task. The algorithm is shown in Algorithm 5 and works as follows. The states of the main task are iterated forward, and visited edges are appended to the active path. If a blocked state is found, the incoming wake-up edge is followed, and the backward iteration starts. In the backward direction, the visited edges are prepended to a local path. If an incoming packet is found, the source is followed backward. If a blocking edge is found while iterating backward, this procedure is repeated recursively. The backward iteration stops when the beginning of the blocking interval is reached, the accumulated path is appended to the result and the forward iteration resumes.

It is immediately apparent from the graph construction algorithm that the runtime complexity is $O(n)$, because it consists of a single loop over events of the trace. The same observations applies to the active path computation, where only the connected components of the graph are traversed, and only once. We conclude that the sequential execution of both algorithms is linear according to the number of events. This property is verified experimentally using the actual implementation and is presented in Section 5.4.5.

5.4 Evaluation

We evaluated the system in three steps. First, we studied a single blocking call according to various operating conditions. We compare the execution on the local host to the execution in virtual machines and on physical machines. We show how the result changes according to the level of asynchronous processing of the application. We observe the effect of network latency and bandwidth on the result.

Algorithm 3 Execution graph construction

```
Input: synchronized trace T ← {T1, T2, ...Tn}
Output: execution graph G
 1: TASK ← {initial tasks}
 2: CPU ← {p0, p1, ...pn}
 3: IRQ ← {interrupt stub tasks}
 4: PKT ← ∅
 5: for all event e ∈ T do                                   ▷ Main procedure
 6:     now ← e.timestamp
 7:     if e is sched_switch then
 8:         LINK_HORIZONTAL(prev task, now, running)
 9:         LINK_HORIZONTAL(next task, now, preempted)
10:         set current task on CPU
11:     else if e is sched_ttwu then
12:         target ← e.tid
13:         source ← CURRENT_TASK()
14:         v1 ← LINK_HORIZONTAL(target, now blocked)
15:         v2 ← LINK_HORIZONTAL(source, now, running)
16:         link_vertical(v1, v2, wake-up)
17:     else if e is interrupt_entry then
18:         push interrupt
19:         LINK_HORIZONTAL(CURRENT_TASK(), none)
20:     else if e is interrupt_exit then
21:         pop interrupt
22:     else if e is inet_sock_local_out then
23:         tx ← LINK_HORIZONTAL(CURRENT_TASK(),
24:                 now, running)
25:         add (packet, tx) to PKT
26:     else if e is inet_sock_local_in then
27:         if packet match found then
28:             (packet, tx) ← remove match in PKT
29:             rx ← LINK_HORIZONTAL(CURRENT_TASK(),
30:                     now, running)
31:             LINK_VERTICAL(tx, rx, network)
32:         end if
33:     end if
34: end for
```

The second evaluation step focuses on analyzing five distributed systems under typical operating conditions. Use-cases were selected to represent the diversity of runtime environments used in the industry, and includes program written in C/C++, Java, Python and Erlang.

Algorithm 4 Execution graph construction (continued)

```
 1: function CPU_TASK()(cpu)                                  ▷ Utilities
 2:     if in_interrupt(cpu) then
 3:         return IRQ peek interrupt task of cpu
 4:     else
 5:         return task of cpu
 6:     end if
 7: end function
 8: function LINK_HORIZONTAL(task, ts, l)
 9:     tail ← last vertex of task from G
10:     Create vertex v with timestamp ts
11:     Create edge tail[right] → v[left] with label l
12:     return v
13: end function
14: function LINK_VERTICAL(from, to, l)
15:     Create edge from[up] → to[down] with label l
16: end function
```

The last step consists of evaluating the analysis cost. We measured the worst-case and average runtime overhead. We also studied the scalability of the analysis implementation.

For all experiments, the operating system is Ubuntu 14.04, running Linux 3.13 and LTTng 2.4. The machine used for local and virtual machine experiments is an Intel i7-4770, with 16 GB of RAM and an 1 TB SSD. The cluster used to run bare-metal experiments has four nodes, where each node is a dual-core AMD Opteron 246 processor, with 4 GB of RAM and 100 GB hard drive, communicating through dedicated Gigabit Ethernet subnet. The analyzer is implemented in Java as Eclipse plug-ins. All the code to reproduce the experiments is freely available on GitHub [1].

5.4.1 Effect of Host Type

We studied the effect of the host type on the analysis results. We compare three type of hosts configuration: a single host, two virtual machines and two distinct computers. We compare the execution of an RPC request for each host configuration. We implemented `wk-rpc`, a minimal RPC implementation in C to control precisely the system calls performed. The remote procedure computes for the amount of time specified by the client. The client connects to the server, writes the command and the parameter to the socket, and then calls read to

1. http://github.com/giraldeau

Algorithm 5 Active path computation

Input: execution graph G, task T, v_s, v_e
Output: path P

```
 1: v ← v_s
 2: while v is not v_e do                                  ▷ Forward iteration
 3:     E ← v.right
 4:     append PROCESS(E) to P
 5:     v ← E.to
 6: end while
 7: function PROCESS(edge)
 8:     if (edge.label is blocking) and
 9:            edge.to has incoming vertical edge then
10:         return RESOLVE(edge)
11:     else
12:         return E
13:     end if
14: end function
15: function RESOLVE(edge)
16:     TMP ← ∅
17:     v ← edge.from                                      ▷ Follow incoming
18:     while v.ts > edge.from.ts do                       ▷ Backward iteration
19:         E ← v.left
20:         if v.down.label is network then
21:             prepend RESOLVE(E) to TMP
22:         else
23:             prepend PROCESS(E) to TMP
24:         end if
25:         v ← E.from
26:     end while
27:     return TMP
28: end function
```

retrieve the return value of the command. When the server-side operation is completed, it sends the return value to the client, which causes the read call to return.

When the client and the server are running on the same host, the operating system transmits network packets on the loopback interface using softirq. It produces the same events structure as a physical interface. The synchronization stage is not necessary, because only one clock is involved.

We executed the client and the server processes each in their own virtual machine running on the same host. We used Kernel Virtual Machine (KVM) as the hypervisor. In this case, traces must be symchronized, because each virtual machine scales the Time Stamp Counter (TSC) to nanoseconds independently.

The third experiment consists in executing the client and the server, each on their own physical computer. Compared to the virtual machine experiment, the communication is done through physical network interfaces.

The results of the three experiments are producing the same structural result as described in Figure 5.1. The communication mechanism on the localhost interface works the same way as a remote socket for the analysis. We conclude that our implementation works for local and remote sockets, and is independent of the host type.

We observed a difference between local and remote executions for large data transfers. When the client and the server run locally, the client may be preempted inside `sendto()` by the server executing `recvfrom()`. When run remotely, the client blocks in `sendto()` instead. The local preemption is however immediate from the trace.

Another case involves a user-space program detecting if its peer processes are on the same computer. The program may use shared memory instead of sockets for efficiency, which changes the local behavior as compared to the distributed execution. The OpenMPI library has this capability, and is discussed in Section 5.4.4. If the wait related to shared memory synchronization is done through blocking system calls such as `futex()`, the wait dependencies between threads is taken into account by the proposed method, without the need to trace accesses to shared memory itself.

5.4.2 Effect of Network Conditions

We used the traffic shaper tool `tc` to increase packet transmission latency for the `wk-rpc` synchronous remote procedure call. The client and the server are running inside virtual machines, and the traffic shaping is applied to the virtual network interface on the host operating system. Figure 5.4 shows the three executions for natural latency in (a), a latency

of 10 ms in (b) and 100 ms in (c). For each execution shown, the server task is above the client.

Each execution begins with two network intervals in pink. They represent the DNS resolution prior to the connection. These intervals are not resolved, because the UDP packets and the DNS server are not traced. If the data was available, the DNS timing could be recovered, but it is left as future work. The second part of the execution shows, as expected, that the proportion of the network transmission increases according to latency.

We also observed the effect of available bandwidth on the behavior of a large network transfer. This experiment involves the `apache` web server and the `wget` client, where traffic shaping is used to limit the bandwidth. We observed that due to the TCP window and the fact that the processing time of the server is low, the transmission delay is greater than the associated blocking window, even when the bandwidth is not limited. The analysis accurately reports that almost all the wait time is caused by the network delay.

5.4.3 Effect of Asynchronous Processing

Asynchronous processing is a computation occurring simultaneously with input and output [1]. We simulate asynchronous processing in the client of `wk-rpc` using a busy-loop between sending the command and receiving the result. The transmission of a small message does not block, allowing the busy-loop to proceed. The effect is to reduce the blocking window of the subsequent read call.

Figure 5.5 shows the active path of the client according to time (server process above client process). Asynchronous processing is the amount of computation done after sending the request and before waiting for the reply. Asynchronous levels are 0, 50 and 100 percent of the blocking time respectively in (a), (b) and (c). When synchronous processing is used, the blocking window reveals the entire server processing related to the request. This window is reduced proportionally to the amount of asynchronous processing, until the point where the process does not block. In this situation, no change in the control flow occurs in the active path.

Another type of asynchronous processing consists in an event loop to keep a single thread responsive, despite long blocking waits. A typical example of an event loop is shown in Figure 5.5 (d). The event loop is implemented with the `poll()` system call, blocking for a resource to become ready up to a maximum timeout. The timeout period is set to 16 ms, corresponding to the screen vertical sync of 60 Hz and simulating the periodic refresh of a graphical user interface. The execution contains four intervals, where the first three (blue) are

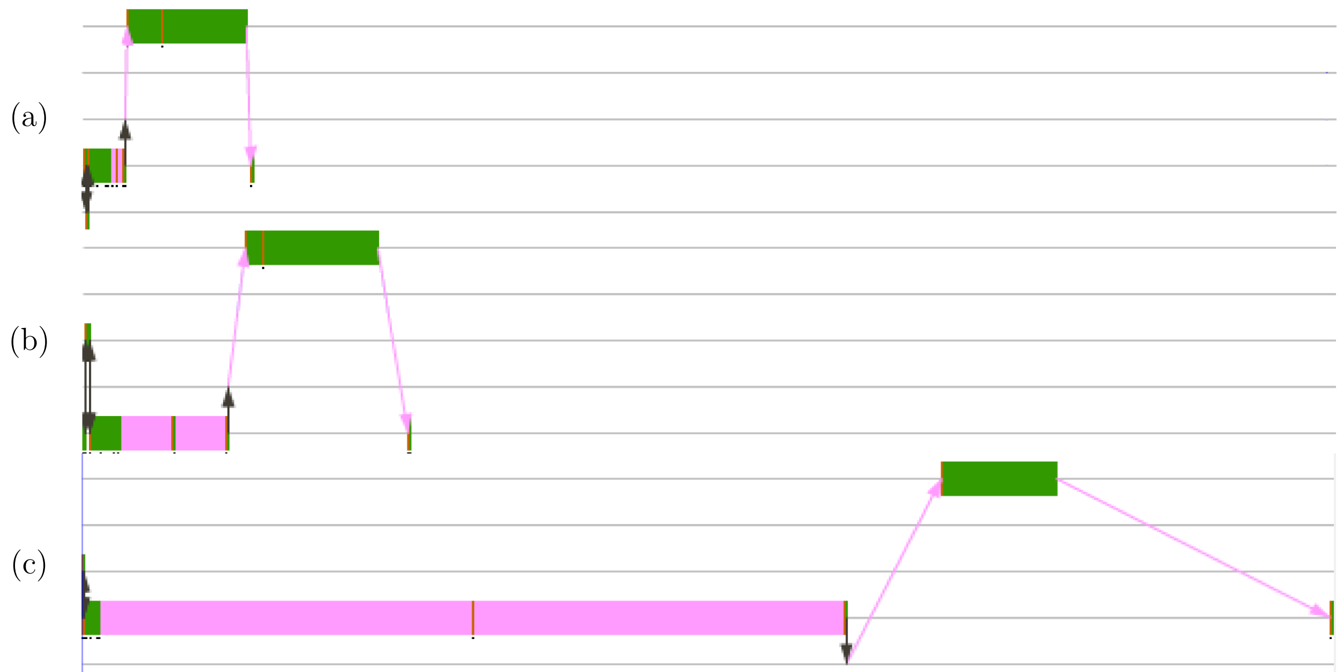


Figure 5.4 Active path of `wk-rpc` according to the network latency. The execution in (a) has natural network latency, in (b) the latency is set to 10 ms and in (c), the latency is set to 100 ms. Green intervals repesent CPU usage and pink intervals and edges represent network latency.

timeouts and where the last one is unblocked by the server's reply. The resulting active path associates wait time to the server only for the last blocking interval of the event loop. Because a large blocking window is decomposed into multiple arbitrary small timeouts, the control flow of the active path changes only for the blocking window related to the completion of the background request. Assuming uniform probability of event inter arrival while blocking, the resulting blocking window will not reflect the actual wait for the resource. Better handling of this execution pattern is a direction for future work, discussed in Section 6.7. However, the analysis is still useful in the event of a missed deadline for screen refresh. The active path would show the cause of the delay at the system level.

5.4.4 Use-cases

Java RMI

The Java Remote Method Invocation (RMI) is a framework to access objects on different computers over the network. We traced a classical example of RMI, where a client invokes a method on a remote server to compute the value of π with a given decimal place [73]. When the server starts, it registers the compute engine object to the `rmiregistry`. The client contacts the registry and obtains a reference to a proxy to access the remote object. The active path of the client is shown in Figure 5.6. The client waits two times for the registry and three times for the compute engine server. The last interval represents the actual computation of π. Java RMI is synchronous by nature, and the method produces accurate results in this condition.

Network Share

A remote file system is a storage device accessed through the network. The operating system handles transparently the I/O for the applications, either on a local drive or on a remote server. This experiment is about evaluating whether the wait for the file system is correctly recovered by our method. The experiment consists of a Samba server, providing a CIFS network share, and a client mounting and accessing files on this share.

Figure 5.7 shows the execution of the remote directory listing, where the server processing accounts for about 18 percent of the active path. The wait occurs in `newstat()` to get the file attributes, and then in the `getdents()` system call that returns the directory entries. These two system calls are actually sending network packets, because of the virtual file system implementation. In other words, even the most trivial program `ls` can be indirectly a distributed program. The correct active path is obtained because no assumption is made

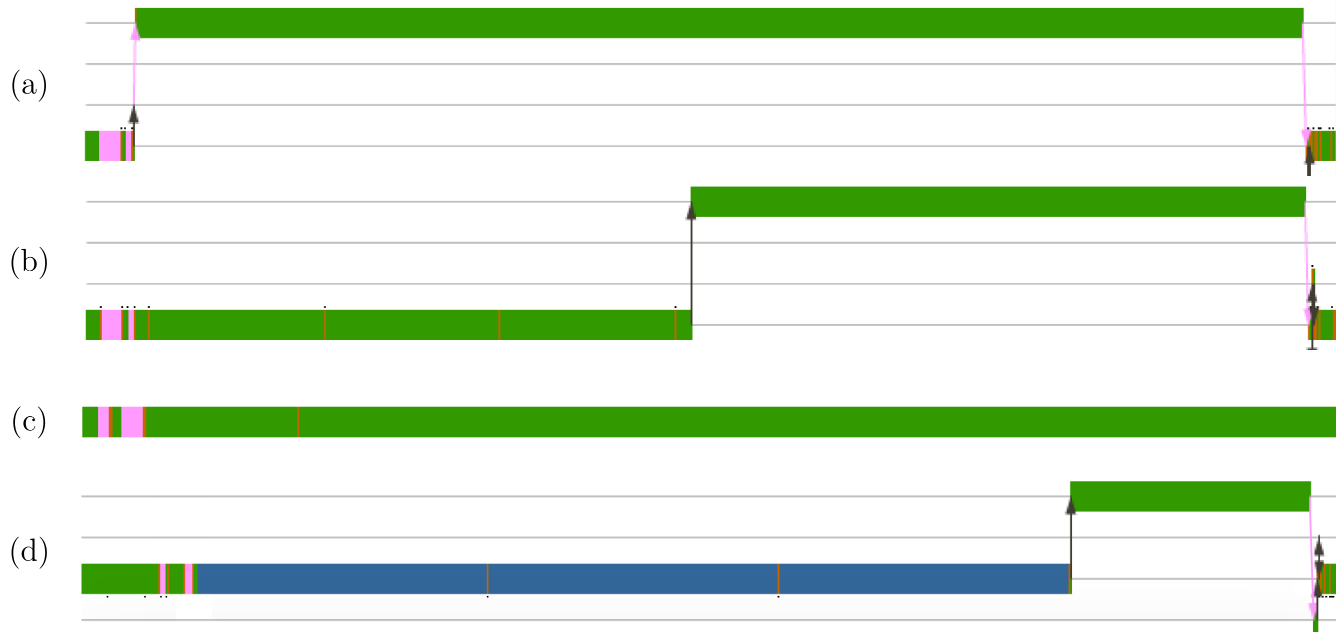


Figure 5.5 Active path of `wk-rpc` according to asynchronous processing level of 0% in (a), 50% in (b) and 100% in (c), and asynchronous processing based on event loop in (d).

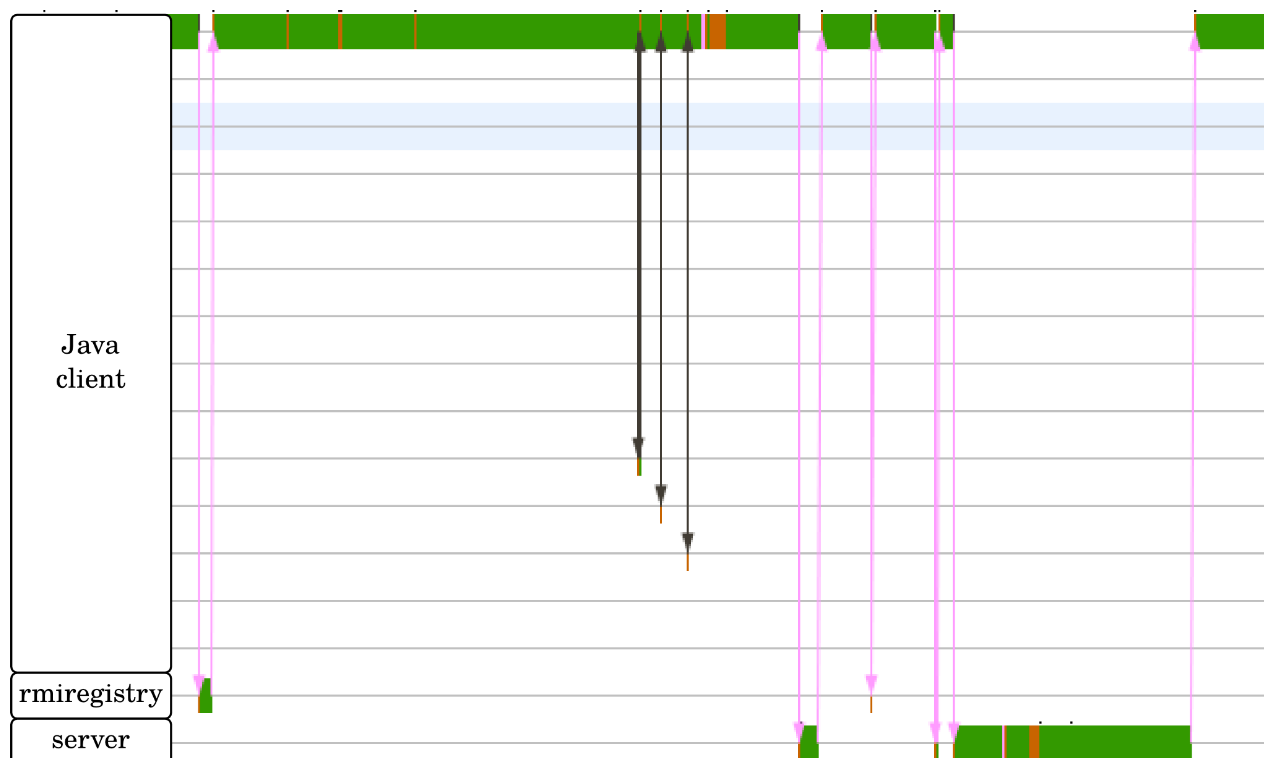


Figure 5.6 Example of Java RMI compute engine execution.

about which system call could send network packets. Another interesting finding concerning the inner working of the kernel is that the `cifsd` daemon receives the answer from the server and then wakes up the main client for both calls. This observation is only possible because the tracing is system wide.

Web Application

We observed the result of the analysis for a simple, yet actual and unmodified Web application. We used the Poll application of the Django project [74], a typical and popular Web framework written in Python. The user's vote is simulated with a non-interactive script using the python library `mechanize`. The voting process has three steps. First, the client performs a get request to download the form, then transmits the data using post, and finally the client is redirected to the poll result page. The application is deployed with WSGI using an Apache HTTP server and a PostgreSQL database. Each tier runs on its own KVM instance and is traced while the vote occurs.

The result of the post is shown in Figure 5.8. The client connects to the HTTP server, sends the POST data, and waits for the reply. The server immediately dispatches the request to a worker thread. The control flow changes frequently between the apache worker thread and the database. Near the end of the request, the postgres process performs a call to `fdatasync()` (purple), blocking until all dirty pages are flushed to permanent memory, and is related to the SQL update statement.

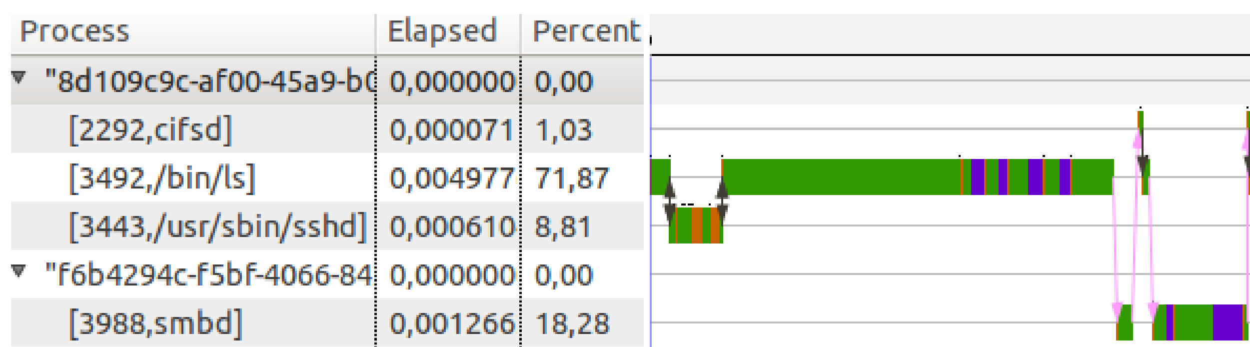


Figure 5.7 Example of a CIFS remote directory listing.

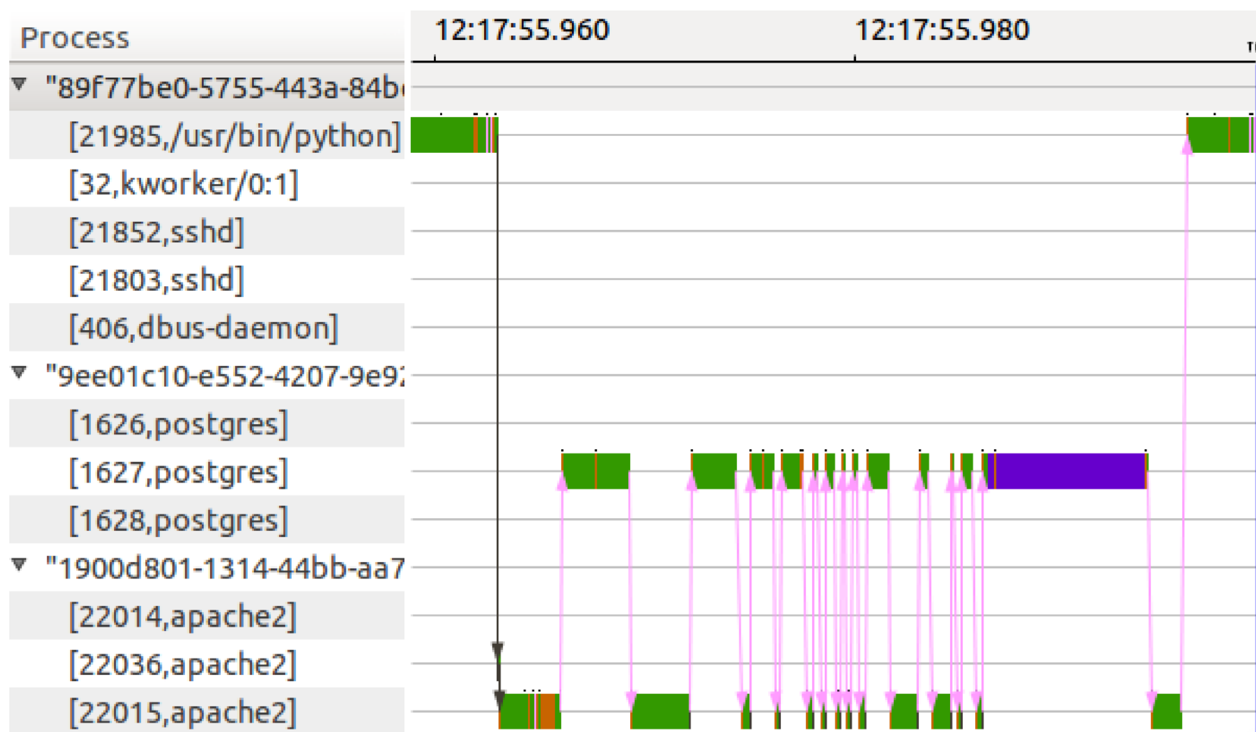


Figure 5.8 Execution of the post HTTP request across the client, the web server and the database.

Erlang Service

We verified that the method works for the Erlang runtime. We implemented a small echo server and its corresponding client in Erlang. The system was deployed in two virtual machines. The experiment consists of ten round-trips between the client and the server. The result is shown in Figure 5.9. The client blocks for the answer from the server, and the active path works for Erlang distributed processes.

MPI Computation

We tested our method for an MPI program using an imbalanced parallel computation. The function `MPI_Barrier()` is called at the end of a cycle to force the synchronization between distributed threads. We found that the OpenMPI barrier is implemented with a busy wait, and appears as normal processing from the kernel perspective. Busy-wait reduces the latency by avoiding the invocation of the scheduler, but at the cost of reduced resource efficiency. This design decision is justified where resources are dedicated and power consumption is a secondary concern, such as in scientific computing.

In this particular case, we used user-space instrumentation to record events before and after the barrier, in addition to the kernel trace. This data is displayed as time intervals and serves as an overlay to highlight the underlying kernel trace corresponding to the wait. Both user-space and kernel traces are using the same clock and timestamp transform, and are therefore synchronized on a per-host basis. Figure 5.10 shows the result for the execution of the MPI program involving two compute nodes. It allowed us to pinpoint that the MPI barrier repeatedly performs non-blocking calls to `poll()`. Future work could consist in evaluating whether it is possible to reliably detect an execution pattern for the active waiting at the barrier using only kernel events.

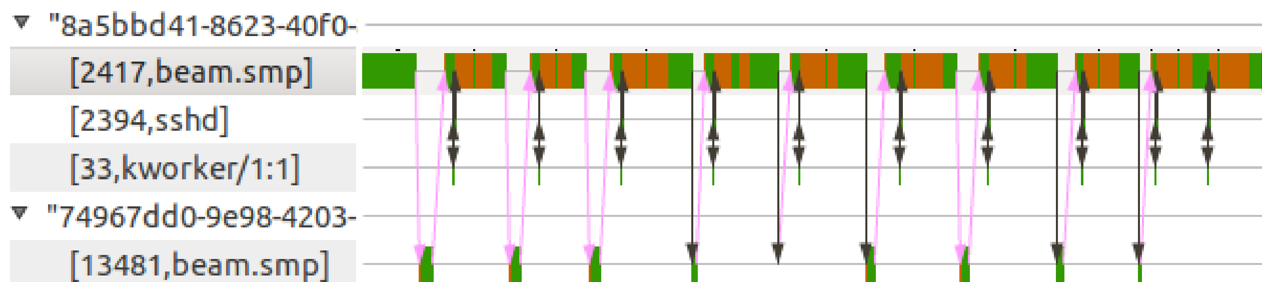


Figure 5.9 Execution of the echo Erlang example.

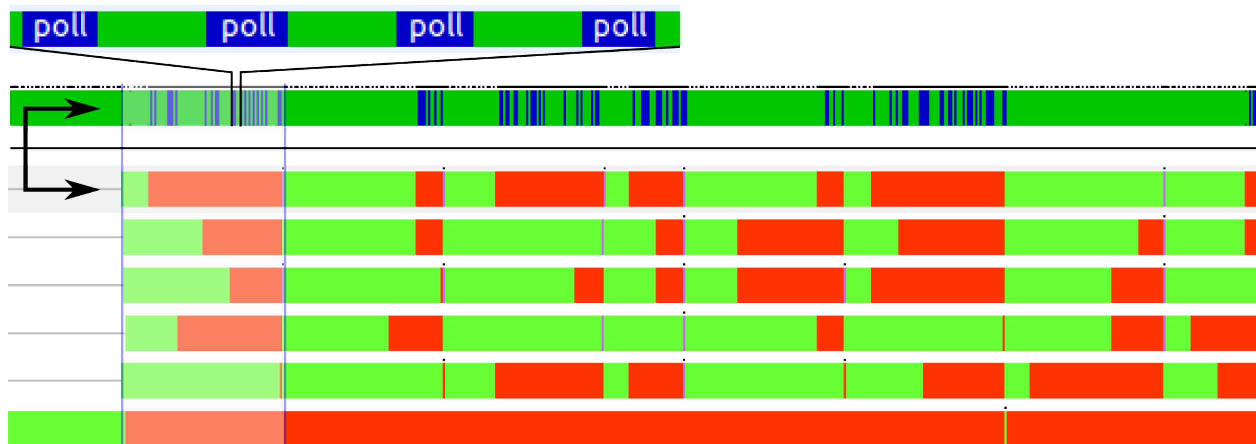


Figure 5.10 MPI imbalanced computation execution. Below: the user-space trace displays the running (green) and waiting (red) of each thread. Above: the kernel trace corresponding to the first MPI thread. The top interval shows a zoomed in view of the active wait section. The first cycle of the computation is highlighted.

5.4.5 Analysis Cost

We present the results of the analysis cost, both in terms of the tracing overhead and the implementation of the analysis algorithm.

The tracing cost includes the tracepoint instructions in the code path, and the execution of the trace consumer daemon, which is responsible for writing the event buffers to disk. This daemon is working as a background thread, and does not increase latency of the workload if no preemption occurs between them.

The total tracing cost is proportional to the number of events produced, and is about 200 ns per event. However, the overhead ratio is proportional to the event production rate. The first part of the experiment attempts to produce the highest possible frequency for a distributed workload, in order to measure the upper-bound of the overhead. The second part focuses on the average overhead for a typical use case. Experiments were performed on the cluster hardware described in Section 6.5. The scheduler governor was set to `performance` instead of `ondemand` to reduce variations caused by processor frequency changes.

The first experiment uses `netperf` to measures the effect on network throughput. Messages of size 16 Kio are sent from the client to the server. The throughput measured is 424 Mbits/s with and without tracing. Tracing has no significant impact on the network throughput for this experiment. This result could be explained by the fact that the probe latency is hidden by the TCP transmission window.

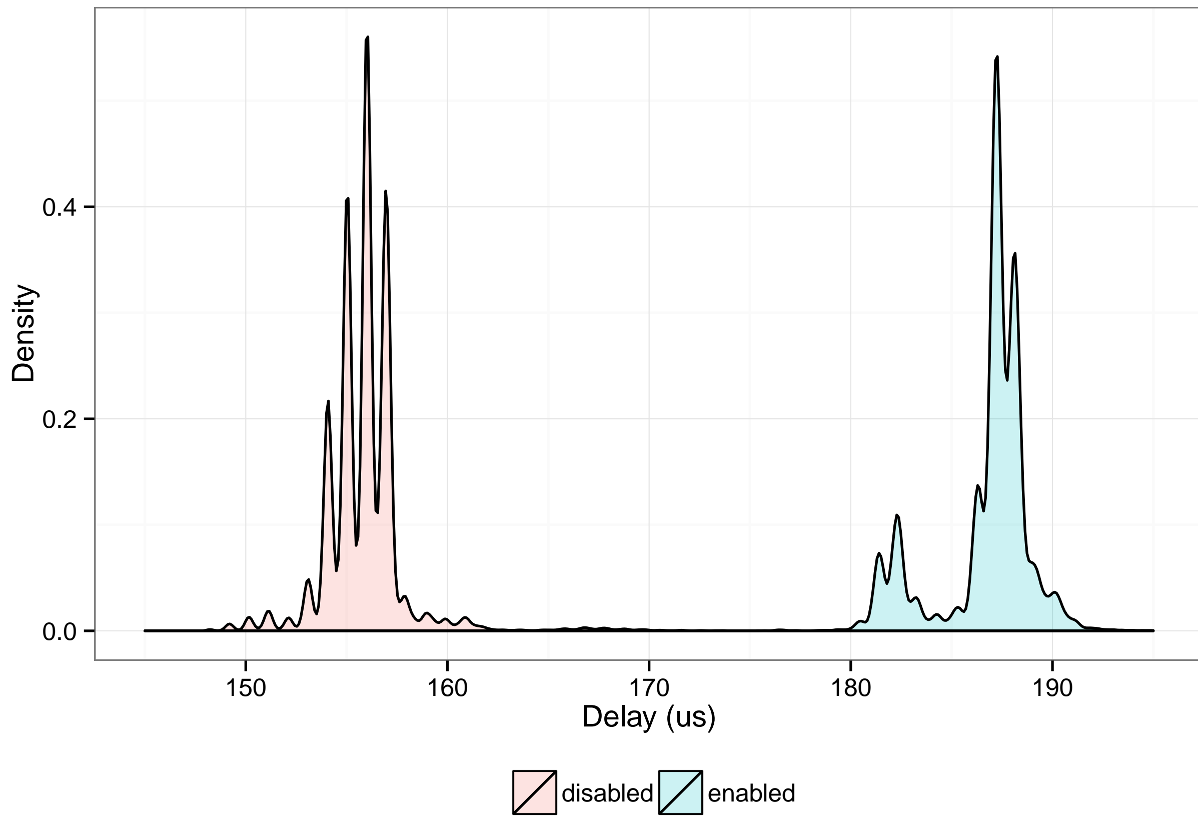


Figure 5.11 Effect of tracing on request latency density for the wk-rpc benchmark. Tracing causes the distribution to shift to the right.

The second experiment uses `wk-rpc` performing requests in a tight loop. The objective is to make round-trip queries at the fastest possible rate. The messages exchanged are only 32 bytes in size. We measured that tracing increased request latency by 18.3 percent, from 155.9 µs to 190.8 µs (mean difference in change, 34.9 [95% CI, 34.7 to 35.1]; $p<0.01$). Ideally, tracing should add a constant delay to requests. Figure 5.11 shows the effect of tracing on the request delay density. The delay density without tracing is multi-modal, but its envelope is uni-modal and nearly normal. The main effect of the tracing is a nearly a constant offset. A slight second mode appears at about 5 µs below the main mode. An hypothesis to explain this phenomenon may be that different code paths are executed depending on runtime conditions. Further analysis is necessary to verify this hypothesis.

The third experiment uses `wkdb`, the Django Poll web application. The test measures the latency for loading and submitting the vote form. The request latency increased by 5.1 percent, from 116.3 ms to 122.5 ms (mean difference in change, 6.3 [95% CI, 4.8 to 7.7]; $p<0.01$).

We measured the CPU usage of the tracing daemon to evaluate its relative impact on the system. We used the scheduling events from the trace to recover the average CPU usage for a window of one second during the peak workload activity. We found that the average CPU usage of the trace consumer daemon is comprised between 0.8 and 8.2 percent of one CPU, and is proportional to the event production rate (R^2=0.91).

To further categorize the cause of the overhead, we computed the event proportions according to their category, namely scheduler, network and interrupt. The results for the three experiments are shown in Figure 5.12. For every experiment, the most frequent events are interrupts. Reducing the tracing overhead for the active path analysis should therefore target interrupt events.

We studied the scalability of the Java implementation of the graph construction and path extraction algorithms. The input is traces of HTTP requests of the `wkdb` application, where the number of requests increases by power of two, up to 2^{12} or 4096. The largest trace has a size of 2.5 GB. Figure 5.13 shows that both algorithms have linear runtime according to the the number of traced requests. The graph construction is the most expensive step of the analysis, being two orders of magnitude more expensive than the path extraction. This is due to the fact that the graph construction includes the trace reading, and this operation is expensive compared to the in-memory graph traversal of the path extraction.

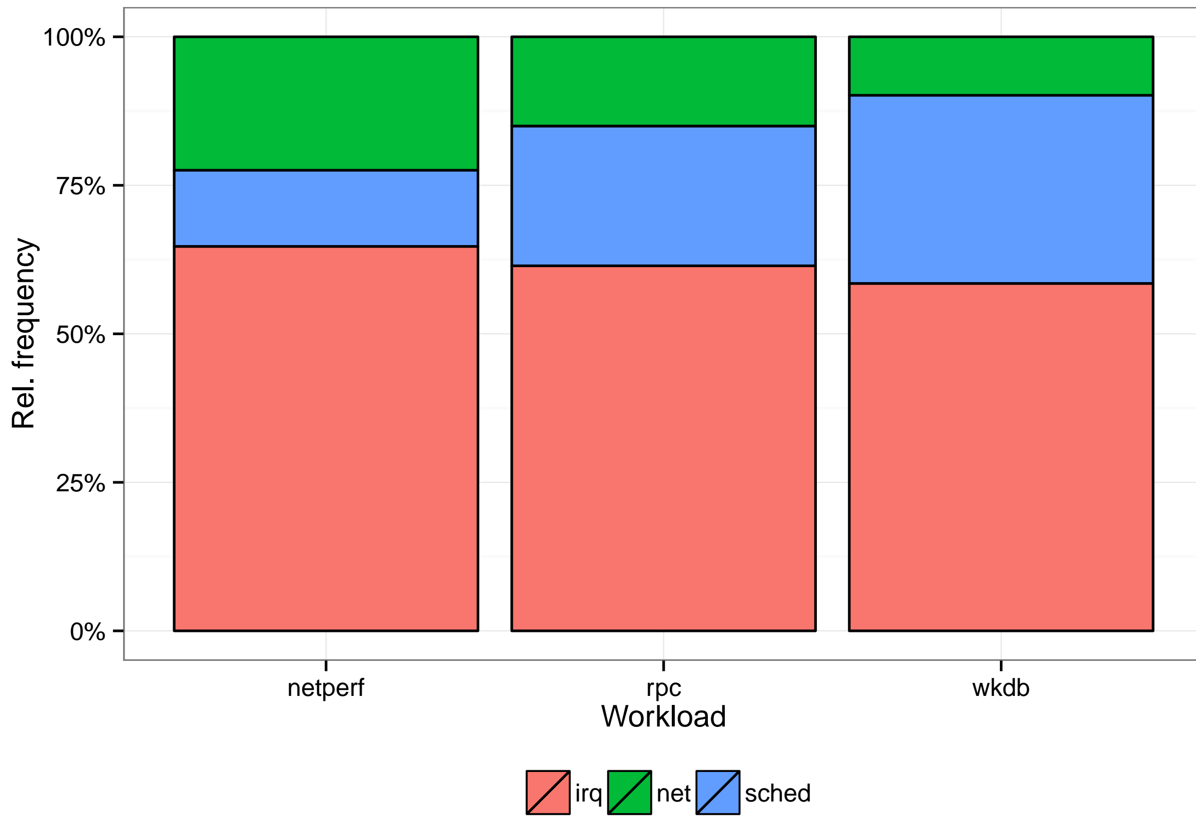


Figure 5.12 Relative event frequency according to workload. Interrupts are the most frequent event type for every workload.

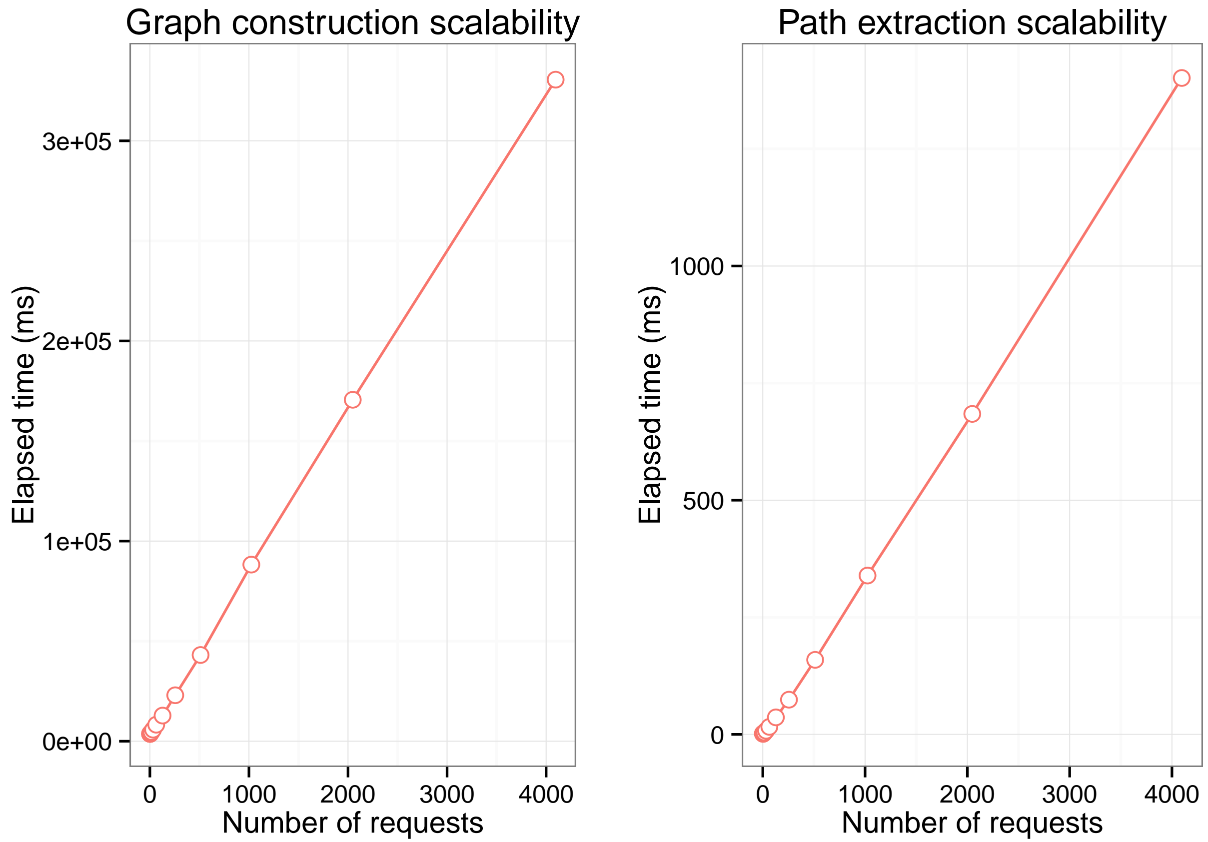


Figure 5.13 Analysis time according to the number of traced web requests. Both graph construction and active path extraction have linear scalability.

5.4.6 Analysis Optimizations

We present two optimizations improving efficiency of trace synchronization for actual traces.

We observed a high peak memory consumption when synchronizing traces, caused by the packet matching algorithm. Packets are added to a hash map and removed once a match is found. However, traces may be highly skewed such that, in the worst case, all events from a trace are read before the other. This has the effect of keeping all packets in memory. Old unmatched packets cannot be removed because a match may be found later, and it increases the memory consumption for the processing. This problem would not occur if the traces were synchronized, because matches would be found in a continuous manner and removed from the map. Packets still unmatched after six standard deviations of the average delay can be safely removed when traces are synchronized, because they will probably never match. Unmatched packets occur if one of the endpoints is not traced, or in case of packet loss.

The basic use case for an experiment is to start traces in parallel on multiple computers. In this situation, it is safe to assume that traces begin about at the same time. We therefore resolve the bootstrapping by shifting the traces to the origin of the reference trace by applying the timestamp transform

$$f(t) = t - (t_s - t_0)$$

where t_s is the start time of the trace and t_0 is the start time of the reference trace. We then perform a first coarse synchronization using the convex hull algorithm. The synchronization is stopped when the graph is connected with timestamp transform of precision greater than 1 percent. This condition is evaluated efficiently by using the weighted quick-union find [75]. When the precision threshold is reached, then a link between the two hosts is added. The coarse synchronization ends if the number of partitions equals one. Then, the normal synchronization procedure can be performed, including unmatched packet expiration.

We evaluated the effect of the two-step synchronization using a trace of a three-tier web application. The trace size is 126 MB. For this experiment, the coarse synchronization reduces the peak memory consumption by 99 percent while obtaining the same precision, but at the expense of reading twice a small proportion of the trace, which increases slightly the processing time (mean difference in change, 16.8 percent [95% CI, 14.5% to 19.1%]; $t(10)=14.1$, $p<0.01$).

The other enhancement is related to the computation of the transform. The transform slope is typically close to one, because all timestamps are in nanoseconds (the TSC scaling is already performed by the operating system). The timestamp in nanoseconds from the epoch is in the order of 10^{18}. When multiplying these two numbers using double precision floating

point arithmetic, the rounding produces non-monotonic timestamps. For this reason, an expensive 128-bit arithmetic is used for every event, specifically Java BigDecimal.

We designed a fast timestamp transform using integer arithmetic that guarantees monotonic time without overflow. The details are shown in Algorithm 6. It is based on the following equivalent rewriting of the linear function

$$f(t) = \frac{mc(t - t_0)}{c} + (mt_0 + b)$$

where the first term is the dynamic part of the timestamp, and the second term is constant over a period of time. The floating point slope is scaled by $c = 2^{30}$ to capture the nanosecond precision and then converted to integer. The effect of the factorization is to reduce the width of the timestamp to the 32-bit range. The multiplication result of the scaled slope and time difference fits into standard 64-bit registers. The division itself is implemented using bit shift for efficiency. Overflow is avoided by recomputing the constant factor mt_0 if $t - t_0 > 2^{30}$ using large decimal, but it occurs only once per second of elapsed time in the trace. As for the offset b, the decimal part is simply dropped because the number is already in the nanosecond range, which is the highest precision of timestamps in the analysis.

We ran a micro-benchmark that computes 2^{25} consecutive timestamps with 200 ns increments, a delay simulating the highest event frequency. Using the same machine as the previous test, the baseline transform function took 10.1 s to complete, compared to 65 ms for the fast transform, representing an average speed-up of 155 times. We performed a benchmark to evaluate the overall effect of the fast timestamp transform on trace reading. Using the same trace as above, the fast transform reduces trace reading time significantly (mean difference in change, -20.8% [95% CI, -16.4% to -25.3%]; $t(10)=9.2$, $p<0.01$).

5.5 Future Work

We showed that, in order to reduce the tracing overhead of the analysis, optimizing the selection of interrupt events to be traced would yield the greatest improvements. Interrupt entry and exit are recorded to know if a given event occurs from interrupt context. Tracing interrupts could be replaced by a per-event context carrying this information. It would reduce the number of events generated, but on the other hand some event sizes would increase slightly. Further study is required to determine the net impact of such optimization.

Concerning the analysis, the current approach has limited memory scalability. The graph size is proportional to the number of state changing events. Working in constant memory

Algorithm 6 Fast timestamp transform

Input: slope m, offset b, timestamp t
Output: transformed timestamp t_x

```
1:  t_0 ← 0                                         ▷ initialization
2:  c ← (1 << 30)
3:  cst ← 0
4:  m_int ← (int)(m × c)
5:  b_int ← (int)b
6:  function FAST_TRANSFORM(t)
7:      if ABS(t − t_0) > c then                    ▷ Rescale
8:          cst ← t × m + b_int
9:          t_0 ← t
10:     end if
11:     t_tmp ← (m_int × ABS(ts − t_0)) >> 30
12:     if ts < start then                          ▷ Rectify sign
13:         t_tmp ← −t_tmp
14:     end if
15:     return t_tmp + cst
16: end function
```

is required to handle traces larger than available memory. One solution would consist in computing a tree of blocking intervals incrementally in a bottom-up manner, and deleting unused vertices of the graph once a blocking is resolved. This method could work because a future event does not invalidate past computations.

During our experiments, we evaluated the effect of page cache conditions on the analysis time for traces stored on SSD. A cache cold run was only 2 percent slower than when the trace is in the page cache. Because the process is CPU bound, parallel processing of the trace may speed-up the analysis.

Another area for improvement concerns the ability to use the active path and relate it to the source code. The active path could be annotated using user-space tracing, call-stack sampling and performance counters. Then, the developer would be able to relate the source code to the underlying system-level execution.

5.6 Related Work

User-space and domain dependent instrumentation was proposed to record request flow and thread interactions in a distributed system. The techniques differ in the instrumentation method and semantics.

In [50], the critical path of a distributed processing system is computed by propagating the CPU usage along the communication edges between processes. The path using the most CPU time limits the completion time. This technique works for CPU bound processing, but does not account for the I/O wait time affecting the completion time.

Panappticon [67] combines user-space and kernel instrumentation to monitor responsiveness of UI on Android. The execution model recovers asynchronous processing and I/O activity correctly. Its scope is limited to the client-side processing.

Request tagging [29], [36] consists in assigning a unique identifier to incoming requests, in order to identify processing related to it. The instrumentation targets server-side frameworks, and is thus transparent to applications using them. The request flow can be augmented with system state. The analysis does not extend to the client.

MPE [41] is an interposition library that can trace calls to MPI. The Jumpshot interactive viewer uses the trace to display thread states and communications. The user can see the dynamics of thread execution according to time. The scope of the analysis is tightly coupled to the MPI domain.

BorderPatrol [23] uses library overloading to intercept calls to the C standard library functions. This instrumentation method is ineffective for statically linked programs. A more robust approach, used in vPath [44], intercepts system calls at the hypervisor level, but requires the workload to run in a virtual machine. Kernel tracing [20] has the same benefit without the virtual machine constraint. All methods based on the system call interface make assumptions regarding the underlying communication that do not hold for the general case.

5.7 Conclusion

Kernel tracing is a system-wide method to understand the actual latency of programs, independently of their runtime environment. We describe a method to visualize the execution of distributed systems using scheduling, network and interrupt events. Thread blocking indicates a change in the control flow, and the wake-up source identifies the wait cause. We demonstrated that this principle of operation can be applied on traces synchronized using the convex-hull algorithms, and that the analysis produces insightful results for a broad range of distributed systems, with a moderate impact on the system.

Acknowledgements

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Geneviève Bastien for code quality, Mohamad Gebai, Suchakra Sharma and Naser Ezzati for their comments, Matthew Khouzam and Alexandre Montplaisir for their help regarding Eclipse, and finally Mathieu Desnoyers and Lothar Wengerek for their precious advice.

CHAPITRE 6 ARTICLE 3 : Execution path profiling using hardware performance counters

6.1 Abstract

The task critical execution path, obtained from a kernel trace, reports the time spent waiting for each task involved in an heterogenous and distributed application. However, additional profiling is needed to understand and identify the problematic code associated with the long-lasting path edges. Hardware counters sampling provides insight on software performance at the microarchitecture level, for instance extracting the call stack every 100K execution cycles to understand where the execution time is spent. Similarly, extracting the call stack at the end of a long waiting system call is often useful. This technique is readily available for either statically or just in time (JIT) compiled code. However, interpreted code is indirectly executed on the processor, and the link between the statements and the executed assembly is missing. We describe an architecture to efficiently record call stacks along the execution path, including interpreted programs, in a low intrusive way that maintains the abstraction boundary between the kernel, the interpreter and the user code. The method consists in sending a signal from within the performance counter interrupt handler. The user-space code receiving the signal can inspect and record the state of the program. We implemented a profiler for the CPython interpreter using this technique. We studied the benefit, the accuracy and the cost of the technique compared to an all-kernel monitoring solution.

6.2 Introduction

In this paper, we consider the requirements for performance profiling of distributed and heterogeneous systems. Services on the cloud may be implemented using a variety of programs executed on different computers. Current profiling tools are either limited to the local host or are restricted to a given runtime environment or communication middleware. This complicates the performance analysis of such systems, because each component must be analyzed separately. A global view of the system behavior is necessary for performance profiling and debugging of actual running systems.

The *task execution path* method has the properties required to address this issue. The segments where tasks wait for each other are extracted from a kernel trace [13]. The kernel trace is itself independent of the runtime environment and works for heterogeneous applications (e.g., C, Java, Python...). Based on that principle, distributed applications can be monitored

by adding network packet send and receive events to the trace. The links between remote tasks can be recovered by matching the packet events, and this also serves to synchronize the traces on a common time reference [15]. Figure 6.1 shows an example of a classical three-tier Web application execution according to time. While the elapsed time allows to pinpoint a slow component along the execution path, it is not sufficient to identify and optimize the offending code. It is especially important in the case of interpreted code, considering that their run-time efficiency is lower. Writing the main components in an interpreted language, for quicker development, and then rewriting the performance sensitive components into a faster language like C, is considered a useful strategy [76]. For the purpose of identifying such code, we use hardware performance counter sampling, which has the potential to be used as a universal profiler. However, supporting performance counter sampling of interpreted code brings additional constraints.

The Performance Monitoring Unit (PMU) counts micro-architecture events, such as cycles, instructions and cache misses [77]. The counters can be read at predefined code locations, namely on function entry and exit, to produce an exhaustive call graph profile. Another operating mode consists in programming the counter to trigger an interrupt when it overflows a given value [78]. The application can be monitored from the interrupt service routine to produce a statistical profile. For example, hot spots in code can identified by simply recording the instruction pointer register (IP) of the program when the cycle counter overflows. The resulting profiling overhead is a compromise between the sampling rate, the monitoring features and the profile resolution.

The operating system is responsible to manage the PMU. It is responsible to program the device and to run the interrupt handlers. Also, maintaining per-task counters requires a tight integration with the scheduler. On Intel processors, a Non-Maskable Interrupt (NMI) is triggered when a counter overflows. A NMI can nest over normal interrupt routines, allowing to sample the interrupt handler themselves. However, the processing inside NMI context is limited in multiple ways. It cannot use locks from other kernel context and it cannot sleep. In addition, the NMI handler is not a schedulable work unit and it must complete quickly to keep the system responsive to other interrupts. Perf is the component managing the PMU in Linux.

Linking the counter overflow to the corresponding source code call path requires the call stack of the program. There are two main methods to get the call stack of the user-space program from the kernel mode. The first relies on frame pointers; they form a linked-list of the return addresses on the stack. Frame pointers thus allow efficient traversal of the call stack. However, frame pointers are usually optimized out in major Linux distributions

and often cannot be relied upon. The second method is based on stack unwinding, which is mandatory to catch C++ exception. It uses section `.eh_frame` of ELF executable files produced by the compiler. The unwinding procedure consists in walking up the stack while restoring clobbered registers, using the values pushed on the stack and the information from `.eh_frame` sections. The call stack is obtained by restoring the instruction pointer for each frame. Other registers restored in the process are not recorded. The unwinding can be done either online or offline. Online unwinding is done inside the code path, while the offline counterpart saves enough context and defers the actual computation outside the code path [79]. Offline unwinding requires saving a portion of the stack, the registers, and the state of executable memory mapped files. Therefore, there exists a trade-off between processing and storage between these two techniques. In every case, debugging information is required to map the addresses to the source code. Mapping instructions to source code is slightly different for JIT engines, because the code generated at run-time is written to anonymous memory maps. Saving the symbol table allows function-level granularity profiles. Another solution is to save the dynamically generated code as ELF files, in which case the rest of the analysis is the same as with statically compiled code.

An interpreter is defined as a runtime environment where the intermediate representation of the source code is executed without compilation [80]. Four programming languages out of the IEEE top 10 ranking, namely Python, PHP, Ruby and MATLAB, do have an interpreter as their reference run-time implementation [81]. For these runtimes, the assembly instructions running on the processor are those of the interpreter itself. For this reason, PMU sampling records the state of the interpreter, instead of the state of the interpreted code, and the link to the actual source code is lost. Call stack recovery of the interpreted code depends on the interpreter internal implementation, which may vary between versions and with compilation options.

In this paper, we study methods to forward the PMU event to the user-space applications. This approach respects the kernel and user-space boundary and allows the application to monitor its own internal state. The context of this work is more demanding than typical profiling use cases where profiles are accumulated over the complete execution of a process, lasting several seconds or more. Indeed, we want to analyze individual execution path components which may be of relatively small duration. When a typical request to a service lasts for instance 200ms, a subcomponent that consumes 100ms instead of 50ms may need to be analyzed. To obtain sufficient precision, many samples will be required during this relatively short period. Similarly, it is often useful to sample the call stack upon returning from a longer than usual system call. In this context, more call stack samples are required than for existing typical use cases, which explains the emphasis of this paper on the efficiency and wide

applicability of obtaining such call stacks, and on the profile precision. As a consequence, the main contributions of this work are :

— The design of PMU event forwarding to user-space.
— The implementation of `PyPMU`, a PMU-based CPython profiling module.
— The evaluation of the accuracy and the performance of the proposed approach, for the context of execution path components profiling.

In Section 6.3, we review existing profilers. The architecture proposed for hardware counter event forwarding is presented in Section 6.4. The evaluation of the design is in Section 6.5, followed by a discussion, proposal for future work and the conclusion in Sections 6.6, 6.7 and 6.8 respectively.

6.3 Related work

Monitoring an interpreter from the kernel is possible. For example, SystemTap can recover the call stack of CPython on PMU overflow [82]. It uses online unwinding to recover the interpreter call stack. Then, for each call to the `PyEval_EvalFrameEx` function, the data structure representing the interpreted code is accessed to identify the running code. This architecture has multiple drawbacks. The monitoring code runs from the interrupt handler and is thus subject to the limitations mentioned in the introduction. In addition, it requires a different kernel module for each interpreter (and possibly version) running on the system, which is a major problem from the system management and security point of view. Inserting a kernel module that depends on the internal structures of a user-space application violates the isolation principle.

The usage of hardware performance counters on Alpha processors is studied in [83]. When a counter overflow occurs, the IP is saved from the interrupt handler, which is a fast but minimal form of monitoring. When optimizing a software component, the programmer may either optimize the callee or another function higher in the call chain. Reporting only IP statistics does not allow one to make such design decisions. While they were reporting an overhead of 1-3%, the system does not perform call stack recovery, nor the analysis of interpreted code.

The PAPI library is a portable layer for accessing performance counters [84]. On Linux, it is essentially a wrapper around Perf. It can configure a counter to send a signal when a memory mapped ring-buffer is full of samples, and ready to be read. Our approach is related to this principle of operation.

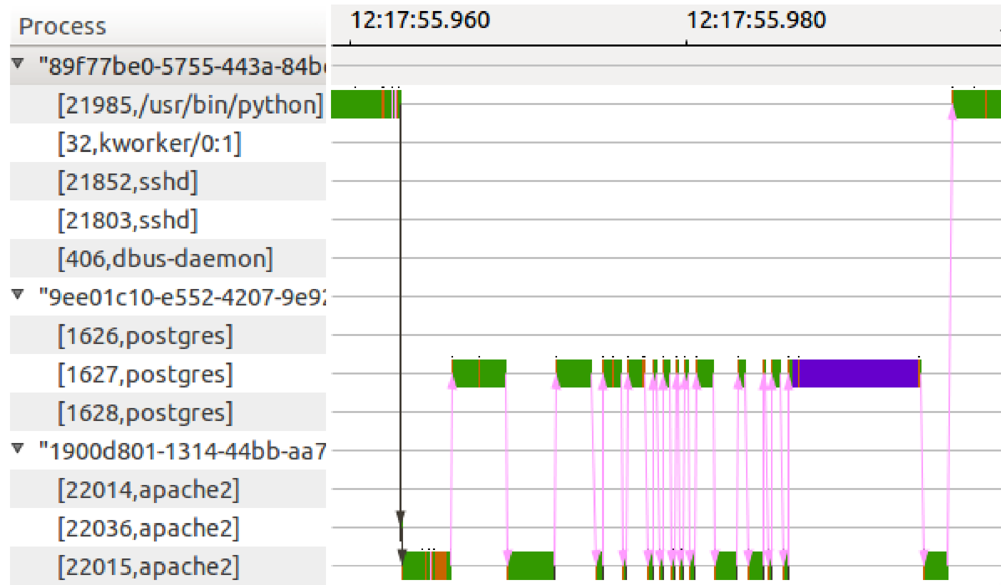


Figure 6.1 This task execution path example shows the behavior a distributed and heterogeneous Web application according to time. The client, Apache HTTP server and the PostgreSQL database are running on different computers and communicate using TCP/IP. Green intervals represents computation, purple is disk I/O and edges are communication between tasks.

Hardware performance counters are also used for online memory optimization on multicore NUMA systems [85]. The memory profile is the basis for moving pages closer to the core using it the most, in order to improve memory locality at runtime. Several other optimizations to Java memory management are provided by this study, improving initial object placement and garbage collection based on the collected profile.

Instruction level parallelism performance may be difficult to interpret in today's complex processors. The Statistical Stall Breakdown (SSB) is a metric to indicate the potential speedup gain by reducing pipeline stalls, based on hardware performance counter sampling [86]. The sampling operates at a rate of 100 us, with a reported overhead of about 2%. The resulting metric can be feed into a run-time optimizer to improve the efficiency of the code.

An online critical path profiling for distributed programs was described in [50]. The CPU usage is transferred along the communication edges between distributed processes and allows the identification of the computation bottleneck. It assumes that the same software stacks is used for all components, which does not hold for heterogeneous systems.

The call stack provides the context of a sample. An extension to that is the flow sensitive profile, where the cost is assigned to the acyclic path belonging to the Control Flow Graph

(CFG) of a procedure. The use of hardware performance counters for the metric source was evaluated in [87]. The reported overhead is about 60-80%, but the result is an exhaustive and detailed profile, not based on sampling.

The sample analysis can be done online or offline. Defering work offline aims at reducing the latency as compared to online analysis. An alternate design is to copy the data needed in temporary memory and wake-up a monitoring thread to perform the analysis on the fly [88]. The program resumes immediately after the copy and the analysis executes in parallel on another core. The latency of the monitored program is reduced, if copying the data is faster than the analysis otherwise occurring in the critical path. The agent thread can access all the memory state of the process for the analysis. This state is being updated by the main thread, such that careful error checking is required. The analysis may fail if the state has changed in such a way that the data structures are not valid anymore, such as if objects are freed. However, the study found that with 32 Kio of stack (or 8 pages), about 90% of samples of the JVM can be decoded using this technique. As a point of reference, copying 32 Kio takes on average 1.1 us in our environment.

6.4 Architecture

On Linux, each active performance counter is accessed by a file descriptor. It represents the handle to control the counter. When the file descriptor is memory-mapped, the kernel allocates a ring-buffer and provides access to it. The kernel produces samples in the ring-buffer which are consumed in user-space through the memory map. The consumer thread can wait synchronously for events to consume using `select()`. Alternately, the file descriptor can be configured with `fasync()` to send the asynchronous signal `SIGIO` when data is ready.

The principle of operation consists to set the counter attribute `wakeup_events` to 1 and use `fasync()` to configure the file descriptor to send a per-thread signal. The memory-mapped ring-buffer is not even required. This simple mechanism is effective to monitor user-space code, because the signal is processed immediately when returning to user mode after the interrupt. The program state does not change between the NMI and the end of the signal processing, and therefore the signal handler is able to inspect the state of the program as it was when the NMI occurred. The Figure 6.3 shows an overview of the steps involved in the signaling. The main steps to setup the counter are shown in Figure 6.2.

The signal handler has relaxed constraints compared to code running inside the NMI handler. The signal handler can sleep, fault and call any other signal safe function. Naturally, it cannot use locks from the main execution context, to prevent deadlock caused by lock nesting. For

```
struct perf_event_attr attr = { ... };
uint64_t val;
int tid, ret, flags, fd;
struct sigaction sigact;

/* open counter */
attr.wakeup_events = 1;
tid = syscall(__NR_gettid);
fd = sys_perf_event_open(
         &attr, tid, -1, -1, 0);

/* fasync setup */
struct f_owner_ex ex = {
    .type = F_OWNER_TID,
    .pid = tid,
};
fcntl(fd, F_SETOWN_EX, &ex);
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL,
    flags | FASYNC | O_ASYNC);

/* start the counter */
ioctl(fd, PERF_EVENT_IOC_REFRESH, 1);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
```

Figure 6.2 Performance counter creation and configuration.

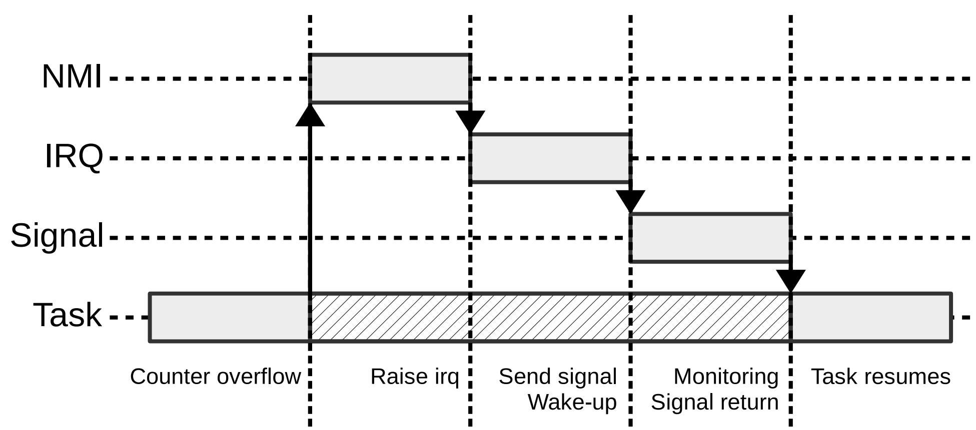


Figure 6.3 Processing sequence of counter overflow event. An NMI is raised on counter overflow. The NMI handler raises a local APIC IRQ, that setup the signal stack and make sure the task is in the run queue. On return from the kernel space, the signal handler is executed, which contains arbitrary monitoring routine. Finally, `sigreturn()` removes the signal stack frame and the application resumes.

example, any function calling `malloc()` is unsafe and must be avoided. Also, we must insure that the thread state is consistent at the time when the asynchronous signal runs. For instance, the compiler can reorder assignments to data structures in such a way that a pointer to a structure may be accessible before its fields are set. This is problematic for the information required for the stack dump, such as the frame structures in an interpreter. One solution to this problem is to publish the data structure reference only when it is ready, as done in RCU [89]. This is achieved by a memory barrier between the field assignments and the update of the current frame pointer.

Modifications to the CPython interpreter 3.5 are necessary to use with SystemTap. This modification exposes the current frame reference. Consequently, an alternate run-time environment must be setup, with the modified Python interpreter, including all library dependencies. It represents an additional burden for the user, unnecessarily complicating performance measurement. Moreover, it is preferable not to deviate from the default environment in order to reproduce bugs, which may disappear if using a different configuration.

For these reasons, we favored the use of an unmodified interpreter. For our prototype, we use the Python module API to obtain the current thread state and to access the top frame reference. The structure fields are checked and carefully accessed to avoid any illegal operations at run-time, upon receiving a profiling signal. Similar accesses are already performed from other signal handlers inside the interpreter.

The events recording (e.g., stack dump), from within the signal handler, is handled with LTTng-UST. This tracer is signal safe and uses efficient per-cpu lock-less buffers that contribute to reduce the overhead [89].

6.5 Evaluation

In this section, we assess the stability, the accuracy and the run-time performance of our approach. All experiments were performed on an Intel i7-4770, with 16 GB of RAM and an 1 TB SSD, running Ubuntu 14.04, with Linux 3.13. The code is freely available on GitHub for others to study and reproduce the experiments [1].

6.5.1 Factoring out monitoring cost

The performance counter does not distinguish between the main program context and the monitoring code running from the signal handler, introducing a positive feedback loop in the system. If the signal handler does enough work to cause a counter overflow, another signal will be queued and may cause an infinite loop of signals leading to program starvation.

To prevent this situation, and to insure accurate measurements that exclude the monitoring code, the signal handler must not increment the performance counter. We achieve this by configuring the counter in single shot mode. The counter is deactivated immediately after the overflow and is enabled again at the end of the signal handling routine, just before the task resumes. It requires only one `ioctl()` call per event to re-enable the counter. This operation takes on average 128 ns.

We tested the behavior of the system by sampling the instruction counter with a period of 10^4 for a CPU intensive workload. We simulated increasing amount of work inside the signal handler using a busy loop, where the number of iterations is doubled each time, up to 10^6, which covers multiple sampling periods. This experiment confirmed that the counter deactivation prevents the program starvation. There is no correlation between the amount of work performed inside the signal handler and the number of samples produced (cor=0.02, $R^2 = 6.9 \times 10^{-4}$). This result shows that the direct cost of the instrumentation is factored out from the performance counter and therefore should have a minimal effect on the profiling accuracy. Side effects caused by the instrumentation might affect the program behavior transiently while resuming execution. For example, eviction of cache lines can cause misses that would not have occurred without the instrumentation. The overall effect should be small

1. https://github.com/giraldeau/

and is not specific to our method. Nonetheless, the monitoring routine increases the elapsed time and must be as efficient as possible. The monitoring cost is studied in Section 6.5.4.

6.5.2 Accuracy

The precision is evaluated using a crafted profile containing ten functions, labeled f0 to f9. Each function calls a busy loop routine, where the number of iterations is proportional to the ideal profile. We measure the actual profile using the deterministic Python profiler `cProfile`. The result is compared to the profile obtained using `PyPMU` with the instruction counter and a period of 10^4. The results of both profilers are shown in Table 6.1.

This statistical significance is important to determine the accuracy of the resulting profile. As a guideline to determine the minimal sample size required, we assume a nearly normal sampling distribution of a single proportion [90]. This assumptions means that a sample belongs to the hot spot or not. It is certainly an approximation, but we are using it based on the fact that software profiles often reflect the Pareto Principle [91]. For a margin of error (ME) of 1% at 95% confidence interval, the ME is given by

$$Z^{\star} \times \sqrt{\frac{(p \times (1-p))}{n}} \leq ME$$

Using $p = 0.5$ to maximize the sample size and solving for n, the number of samples must be greater than 7140. The result set presented in Table 6.1 is more than 50 times this minimal sample size. We are confident that this number of samples is sufficient to factor most of the sampling variations from the profile error.

We measure the accuracy using the Root Mean Square (RMS) error for each function in the profile. We found that the accuracy of the two profilers are equivalent and the RMS error is less than 1%. This result suggests that the profiler has no obvious bias or discrepancy and should be acceptable in practice for measuring program performance.

6.5.3 Signal cost

This section details the cost components of a counter overflow event. The time includes the perf NMI handler, the local IPI sending the signal, the signal handler execution and the system call to reactivate the counter, but with an empty monitoring routine. Thus, it represents the baseline sample cost. The analysis of the monitoring task is presented in the next section.

Table 6.1 Profile measurement accuracy

Fn	Exp.	cProfile		PyPMU	
		time (s.)	%	count (10^3)	%
f0	50.0	1.997	50.1	186.6	50.0
f1	20.0	0.799	20.0	74.6	20.2
f2	15.0	0.599	15.0	55.8	14.9
f3	5.0	0.199	5.0	18.6	5.0
f4	5.0	0.199	5.0	18.6	5.0
f5	1.0	0.040	1.0	3.8	1.0
f6	1.0	0.039	1.0	3.7	1.0
f7	1.0	0.039	1.0	3.6	1.0
f8	1.0	0.039	1.0	3.7	1.0
f9	1.0	0.039	1.0	3.8	1.0
RMS (%)			0.1		0.1

We measured the average time to service an event using a busy loop benchmark and the cycle counter. The average time is calculated by measuring the difference between the time to execute the benchmark, with and without the instrumentation, and dividing by the number of events processed. The average is computed for 10^6 samples. On average, event forwarding takes 2.93 us. For comparison, we made an experiment where a thread sends `SIGUSR1` to itself in a tight loop. The signal handler executes before the next signal is sent. The mean iteration time is 879 ns, or about a third of the event forwarding cost.

For completeness, we measured the time needed to create and destroy the performance counter and to setup the signal handler. The micro-benchmark indicates that the counter management takes on average 4.15 us. Since this cost is amortized over the whole execution, it is negligible for practical applications.

6.5.4 Monitoring cost

The objective of the monitoring routine is to record the state of the interpreter and/or the interpreted code from the signal handler. We evaluate the monitoring alternatives in terms of their time and storage cost. The interpreter state is recovered using call stack unwinding. We compare the online and offline unwinding approaches. In the online case, the instruction pointer is recovered for each frame and the result is recorded in the trace buffer

using `libunwind`[2]. The offline unwind writes two pages of the stack (8 kio) and the values of general purpose registers to the trace buffer. This is the unwind method used by the Linux Perf tool inside the NMI handler. For the interpreted code, the built-in Python interpreter traceback is compared to tracing using LTTng-UST.

The benchmark recursively calls a function, and the monitoring routine is called when reaching a given call stack depth. We measured the mean time to execute the monitoring routine according to the recursion depth, up to a depth of 100, as shown in Figure 6.4. The Table 6.2 includes the linear fit of the data and the average event size for the whole trace.

The offline unwinding cost is nearly constant. This is expected, because the same amount of work is done for each event. As a comparison, a simple, cache hot `memcpy()` of two pages and the registers takes about 375 us, which is about 8 times faster. Recording such a large sample to the trace buffer adds an important extra cost.

On the other hand, the online unwinding cost is strongly proportional to the call depth. Solving the linear fit equations, online unwinding is faster than offline unwinding for depths less than 108. In the current implementation, we fixed the maximum stack depth to 100. The call stack is truncated if it is greater than the maximum. In these conditions, online unwinding is always faster. Regarding the trace size, offline unwinding uses on average 30 times more space than online unwinding in the test, where the mean depth is about half the maximum, or 50. We conclude that online unwinding is more efficient both in terms of time and storage.

The Python traceback is a simple traversal of linked frames. With the built-in traceback method, this information is formatted as a string and uses the `write()` system call to save the result to a file. The LTTng-UST method saves the frame information directly to a ring-buffer instead. The cost benefit of using LTTng-UST in this case is obvious, being at least 25 times faster than the built-in counterpart. The trace sizes are in the same order of magnitude, with a slight advantage for LTTng-UST, which uses on average 25% less storage.

6.5.5 Profiling overhead

We now consider the average overhead for the overall instrumentation. Unlike for our micro-benchmark, the measurements reported here take into account the sampling rate. A higher sampling rate increases the resolution, but also increases the run time. We first explore the overhead according to the sampling period. We use the profile benchmark described in Section 6.5.2 to measure the overhead as a function of the sampling period of the cycle counter.

2. http://www.nongnu.org/libunwind/

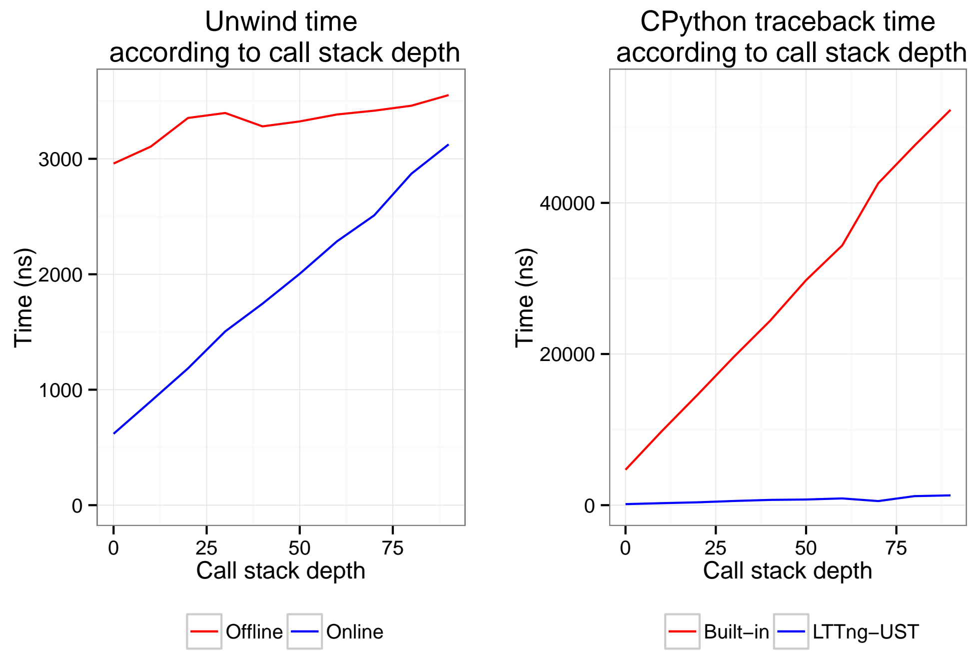


Figure 6.4 Monitoring cost according to call stack depth, for unwind and traceback.

Table 6.2 Monitoring cost

Experiment	Linear fit (ns) $mx+$	b	Avg. event size (byte)
Unwind Offline	$4.9x+$	3101	9106
Unwind Online	$27.6x+$	631	300
Traceback Built-in	$535.0x+$	3895	5943
Traceback LTTng-UST	$11.4x+$	155	4539

The results are presented in Table 6.3. Then, the overhead is compared to SystemTap, which is to our knowledge the only other profiler based on hardware performance counter sampling capable to monitor interpreted Python code.

As expected, the overhead is inversely proportional to the sampling period and can be arbitrary low, depending on the resolution requirements. We notice that the Python code traceback is slightly more costly than the interpreter unwinding, which is the opposite of the micro-benchmark.

Next, the overhead is compared to SystemTap. While making measurements, we observed that the number of samples produced is not proportional to the requested sampling frequency. In fact, the operating system throttles the counter if overflow frequency is too high and the CPU usage for the monitoring is above a given threshold, essentially putting an upper-bound on the overhead. In our case, the monitoring processing is not accounted for the throttling, because it is defered to the signal handler. We observed no throttling with our method. It allows us to sample at the actual period specified by the user, independently of the overhead. The average SystemTap event cost is 13.9 us, which is 42% more than our combined monitoring solution at a period of 10^6. The overall performance of forwarding the event to user-space for monitoring has the same order of magnitude in performance as processing the event entirely in kernel mode. The additional cost to forward the event to user-space can be offset by using a more efficient tracer and monitoring code.

The current SystemTap script for CPython requires two user-space probes, or `uprobes`, on frame entry and exit. These probes are implemented with a software interrupt and, consequently, the overhead is proportional to the frequency of function calls in the program. The amount of work per function is highly dependent on the code implementation and programming styles. The profile benchmark discussed previously has a low function call frequency. To better evaluate the overhead for the normal case, we use the π calculation benchmark from the CPython distribution. The profiling with SystemTap slows down the execution by a factor of 3.4 and produced about 4744 events. The equivalent result can be obtained with only 10.8% of overhead with our approach, independently of the function call frequency.

6.5.6 Sampling resolution

A proper sampling period should be defined to get a precise profile and maintain a reasonable overhead at the same time. As an example, we consider the task execution path presented in Figure 6.1, having a duration of roughly 50 ms. Figure 6.5 shows the relation between the overhead and the margin of error according to the sampling period of the cycle counter. We use data from the combined monitor and assume a sampling of a single proportion. Sampling

Table 6.3 Average profiling overhead for interpreter unwinding, interpreted code traceback and both monitoring combined, according to the sampling period of the cycle counter.

Monitor	Period (ns)	Overhead (%)
Unwind	10^4	231.4
	10^5	18.7
	10^6	0.6
	10^7	0.1
Traceback	10^4	316.6
	10^5	34.1
	10^6	2.4
	10^7	0.3
Combined	10^4	334.8
	10^5	35.3
	10^6	3.0
	10^7	1.2

at 10^7 nanoseconds (or 10 ms) has very low overhead, but the margin of error is high and could produce a misleading profile. On the opposite side, a low ME is costly and would produce more samples than required for practical profiling purposes. The interval $[3.5 \times 10^5, 5.5 \times 10^5]$ nanoseconds for the sampling period simultaneously satisfies an overhead below 10% and a margin of error smaller than 5%. This seems a good compromise between the two parameters, where the hot spots can be identified at a reasonable cost.

6.6 Discussion

In this section, we complete the analysis by presenting issues related to signals and design decisions of the system.

6.6.1 Unix signals limitations

Using a signal implies some general considerations that may impact the profiler. We review these issues and present possible solutions.

In the prototype, the `SIGIO` signal is used by the profiler. However, this signal can already be used for another purpose in the profiled application. To mitigate this problem, Linux

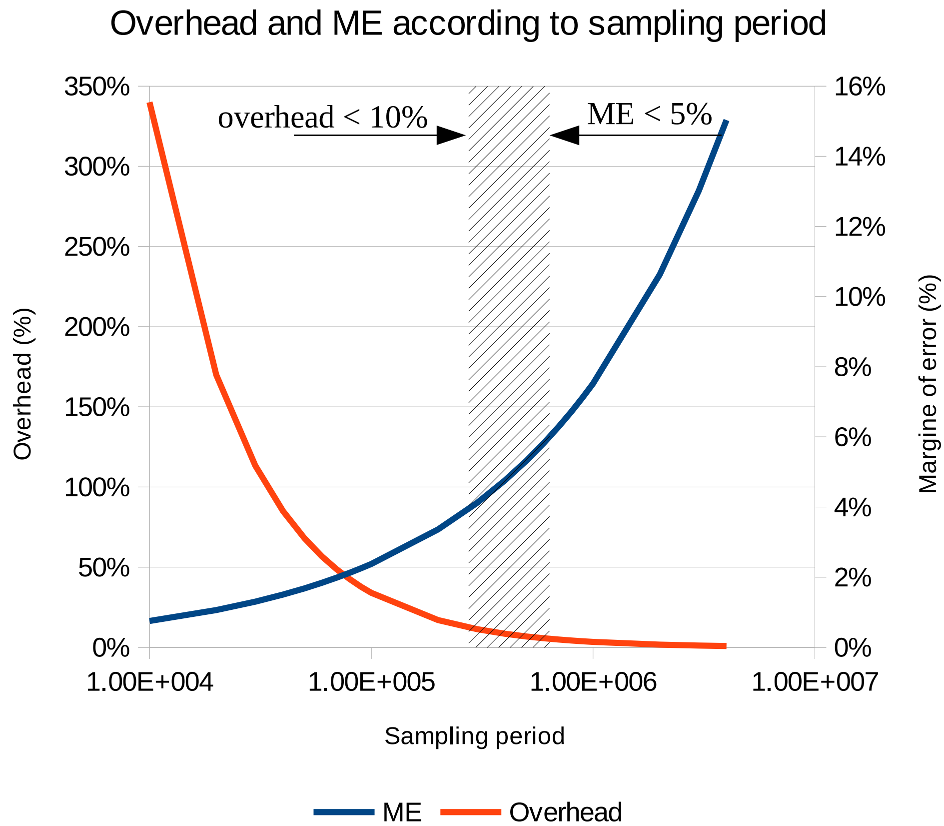


Figure 6.5 The overhead is reduced by increasing the sampling period, which increase the margin of error.

provides 32 user-defined signals. Instead of hard-coding a signal number, the next available signal should be used instead.

Signals can be temporarily disabled. The user code can block signals using the `sigprocmask()` system call. A common motivation for blocking signals is to prevent deadlocks between the main context and the signal context. In other words, disabling signals before locking prevents nesting over the same lock inside the signal handler. If a signal is sent while it is blocked, it is delivered immediately after the signal is unblocked. The sampling may therefore be delayed for an indefinite period of time. The same limitation already applies to the `SIGPROF` signal used for timer based profiling. It is possible to detect if signals are blocked for a long period of time using kernel tracing and report this to the user if it is an issue for the application in question. In the specific case of deadlock prevention, the best solution consists in avoiding locks completely. For instance, no lock is required if the data is not shared between execution contexts.

6.6.2 Performance counter scope

The design presented in Section 6.4 allows us to control the performance counters from within the program. The analysis can therefore address a specific code section of the program, excluding irrelevant code that would otherwise add noise to the analysis. However, the current design requires integration with thread management, and to start one counter per-thread.

On the other hand, counters can be enabled per-CPU in a system wide manner. This operating mode reduces the configuration burden to start counters from each observed process. A combined approach could use system wide performance counters, with a subscription mechanism to overflow events. This way, special run-times, such as interpreted code, could receive signals for any global active counter, without requiring to manage the counter themselves.

6.7 Future work

Performance counter sampling of all major run-time environments is a prerequisite for integration with the task execution path analysis. This integration also needs trace synchronization and task execution path extraction from kernel trace. This tooling is implemented into the TraceCompass analyzer, an Eclipse Foundation project [3]. An extension plug-in can then built on that foundation to aggregate performance counter samples within the intervals of

3. http://eclipse.org/tracecompass

the execution path and produce a profile representing computations that may spawn several computers.

Traditional signal mechanisms could be replaced by a more lightweight User-Level Interrupt (ULI) to forward interrupts to user-space [92]. The requirements are more stringent than signals however. Forbidden actions in ULI are page faults, floating point operations, system calls and illegal instructions. It should be possible to implement the profiler handler within these constraints. Additional study should be conducted to evaluate the potential speedup compared to the increased complexity of the handler.

The signaling mechanism is not limited to the performance counter events and can be extended to other events. For example, scheduling events of the operating system are not communicated to the program and thus preemption and blocking is not visible. In the case of high waiting latency, the program may record its state for performance debugging purposes. Since lengthy latencies within a given task are necessarily low frequency events, by definition, the related overhead of the signal handling should be relatively low.

The call stack samples may have a large proportion of redundancy. The functions at the root of the tree do not change often. Recording only the changes relative to the previous call stack would reduce the tracing time and space overhead. One approach to detect unchanged frames and limit call stack walk is to maintain a sentinel bit [46] and could be a possible optimization avenue.

6.8 Conclusion

We evaluated a mechanism to forwarding hardware performance counter overflow events to user-space. This technique provides a universal profile data source for distributed and heterogenous application, to use with the task execution path method. We demonstrated that the resulting profile is accurate and that the signal cost can be offset by efficient monitoring and tracing. An average overhead below 10% and a margin of error smaller than 5% can be achieved to identify hot spots in short intervals of 50 ms that occur in distributed processing.

Acknowledgements

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Mathieu Desnoyers and Pawel Moll for their precious collaboration.

CHAPITRE 7 DISCUSSION GÉNÉRALE

7.1 Atteinte des objectifs

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

Instrumentation noyau : Il a été possible d'instrumenter dynamiquement le noyau à partir de modules chargeables, ce qui évite à l'utilisateur l'étape délicate et fastidieuse de modifier et de recompiler le noyau à partir des sources. Cet aspect était une contrainte technologique importante pour la mise en oeuvre. La majorité de l'instrumentation requise a été réalisée grâce aux points de trace statiques du noyau, appellés `tracepoint`. Une fonction de rappel est ajoutée dynamiquement pour enregistrer l'évènement. Dans certains cas particuliers, il a été nécessaire d'utiliser `kprobe` pour insérer une fonction de rappel à un endroit précis du noyau. Lorsque défini à l'entrée de la fonction, le site d'instrumentation peut généralement être optimisé par le mécanisme de `ftrace`, qui consiste à modifier, pendant son exécution, le code du noyau pour remplacer un instruction `nop` en début de fonction par un renvoi vers l'instrumentation. Par contre, l'instrumentation basée sur des symboles du noyau est potentiellement moins stable dans le temps que celle basée sur les points de trace statiques, parce qu'elle peut être affectée par une réorganisation du code source. Finalement, l'interface `netfilter`, utilisée pour intercepter les paquets réseau, s'est avérée simple à exploiter.

Modélisation de l'exécution distribuée : Il a été possible de modéliser l'exécution du point de vue du système d'exploitation sans l'utilisation des appels systèmes. Ce dernier aspect permet d'oberver les tâches noyau, ce que la méthode précédente ne permettait pas. Par contre, le chemin critique exact du point de vue du système d'exploitation ne peut être obtenu sans l'information concernant les verrous. Lorsqu'une contention survient, le moment du début de la section critique est requis. Les accès aux sections critiques peuvent survenir très fréquemment, et leur traçage exhaustif serait prohibitif. L'approximation qui consiste à remplacer récursivement les périodes d'attentes produit une approximation intuitive, et souvent plus utile dans la plupart des cas.

Extraction du chemin d'exécution d'une application distribuée : Le modèle de base a été étendu pour supporter les applications distribuées. L'enregistrement des paquets réseau a été nécessaire. Lorsqu'une application bloque, la cause est identifiée en suivant le paquet réseau ayant provoqué le réveil. Cette technique fonctionne particulièrement bien pour les

applications synchrones. Les programmes asynchrones, basés sur une boucle d'évènements, peuvent produire un résultat incomplet.

Calcul des ressources : Le chemin actif de l'exécution est parcouru et, pour chaque tâche, le temps écoulé est classifié entre temps de calcul, préemption, attente pour le réseau et attente pour un autre périphérique. Cette classification est efficace pour caractériser le temps écoulé. La proportion du temps pour chaque composante est identifiée.

Relier la trace noyau au code source : Des solutions existent pour identifier le code source de programmes compilés statiquement ou juste à temps. Le déroulement de la pile d'appels *native* peut s'effectuer en ligne ou hors ligne, tant depuis le noyau que de l'espace utilisateur. Nous avons démontré qu'il était aussi possible d'identifier le code source d'un programme interprété en utilisant des compteurs de performance matériels, et qu'un cout raisonnable peut être atteint par un compromis entre la fréquence d'échantillonnage et la marge d'erreur cible.

Application de l'analyse à un large éventail de configurations : La méthode a été utilisée pour analyser un échantillon représentatif de différents environnements logiciels rencontrés en pratique. Cette étude montre des résultats concluants pour tous les types de moteurs d'exécution, comprenant les interpréteurs, le code compilé statiquement et juste à temps. La méthode fonctionne autant pour un ordinateur local, une grappe d'ordinateurs et des machines virtuelles. La méthode de synchronisation de la trace basée sur l'algorithme de *convex-hull* s'est avérée suffisamment précise pour éviter les inversions d'évènement, et répond au besoin de l'analyse. Le chemin actif calculé est particulièrement adapté lorsque l'attente est synchrone.

Mesure du cout de l'analyse : Chaque élément composant du cout en fonctionnement a été identifié, et comprend la latence directe de l'instrumentation, le temps processeur des démons d'arrière-plan, ainsi que les E/S requises pour l'enregistrement. Le surcout moyen dépend du type de charge de l'application. La principale source du surcout est liée à l'ordonnancement. Un programme produisant des changements de contextes fréquents aura un surcout généralement plus élevé. Notre méthodologie a permis d'établir le surcout au pire cas pour les évènements d'ordonnancement à 11%. L'observation de la distribution de la latence montre que l'instrumentation a comme effet principal un décalage constant, et qu'elle induit une faible distorsion de la distribution. Nous avons vérifié que l'implémentation des algorithmes de construction du graphe et de l'extraction du chemin actif affichent un temps d'exécution linéaire par rapport à la taille de la trace, et que la lecture de la trace constitue l'opération la plus couteuse du processus.

En résumé, les objectifs de la recherche ont été largement atteints. La technique proposée permet d'étudier l'attente dans les systèmes distribués et hétérogènes de manière efficace.

7.2 Retombées connexes

Les travaux présentés à la conférence *Ottawa Linux Symposium* visaient à exposer les options disponibles pour utiliser l'instrumentation noyau aux fins de monitorage de l'utilisation des ressources [93]. Il s'agit des résultats de l'analyse préliminaire de la recherche, entre autres pour se familiariser avec l'instrumentation du noyau. Les principaux sous-systèmes sont considérés, soit l'utilisation du processeur, du réseau, des disques et de la mémoire. Un prototype d'analyseur calculant le taux d'utilisation du processeur à l'aide des évènements d'ordonnancement a été développé. Cette méthode procure des statistiques d'une précision de la nanoseconde et garantit l'identification des tâches de courte durée, deux limitations de la scrutation des fichiers de statistiques habituellement utilisée pour le monitorage. Cette fonctionnalité est maintenant incluse dans l'analyseur Trace Compass.

Les outils développés dans le cadre de la recherche ont été appliqués à l'enseignement à l'École Polytechnique de Montréal. En particulier, le traçage noyau fait maintenant partie intégrante des laboratoires du cours de système d'exploitation de premier cycle. Des activités originales ont été conçues de manière à exposer les comportements du système d'exploitation. Les outils utilisés permettent de rendre explicites et visuels des phénomènes complexes et abstraits. Ces activités et les outils nécessaires découlent directement de l'expérience acquise au fil de la recherche. L'article issue de ce développement pédagogique a été présenté dans le cadre de la conférence de l'*American Society for Education Engineering* [94].

Les résultats concernant l'analyse des systèmes distribués ont été présentés au *Tracing Summit*, qui est concomitant à la conférence `LinuxCon 2014`. Il s'agit d'une conférence technique visant en premier lieu les développeurs du noyau Linux.

L'approximation du chemin critique a été utilisée pour comparer des exécutions semblables [95]. Une paire d'évènements identifie une section d'exécution à étudier. Le chemin critique de chaque section est calculé, puis une euristique regroupe les exécutions ayant la même structure. Une interface permet de sélectionner deux ensembles d'exécutions en vue de la comparaison. Le rapport indique les différences en termes de latence et d'utilisation de ressources. En sélectionnant les exécutions normales comme référence et celles aberrantes comme groupe de comparaison, cette méthode permet d'isoler efficacement les causes potentielles d'un comportement anormal.

Outre les blocages, la préemption est une autre forme d'attente qui peut contribuer à la latence d'une application. Déterminer la cause de la préemption sur un ordinateur local est simple, car elle est causée par toutes les tâches qui s'exécutent sur le processeur courant pendant l'intervalle de préemption. Lorsqu'un programme s'exécute dans une machine virtuelle, la cause de la préemption peut être une tâche s'exécutant sur l'hôte ou dans une autre machine virtuelle. Nous avons proposé une approche pour analyser globalement les causes de préemption, dont le fonctionnement est analogue à celui du chemin actif distribué [96]. La machine hôte et les machines virtuelles sont tracées. Comme chaque noyau maintient une référence temporelle distincte, les traces sont synchronisées pour être analysées. L'algorithme retrouve la cause racine de chaque préemption des processus représentant des processeurs virtuels.

En dernier lieu, une contribution a été apportée aux travaux de portant sur l'analyse de système temps réel [97]. L'étude vise à réduire le délai absolu du traçage, mais aussi à réduire la variance des exécutions.

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS

Cette recherche a permis de faire avancer l'état de la connaissance dans le domaine de l'analyse de la performance d'applications distribuées et hétérogènes. Nos contributions incluent :

- La conception et l'implémentation de l'instrumentation noyau requise pour l'analyse,
- Une méthode pour produire un graphe d'exécution à partir d'une trace noyau,
- Un algorithme d'extraction du chemin actif depuis un graphe d'exécution,
- Une méthode d'échantillonnage de compteur de performance matériel pour les interpréteurs,
- Une suite de bancs d'essai pour la conception, la validation et la mesure du surcout.

En conclusion, l'approche préconisée basée sur le traçage noyau permet de comprendre l'attente se produisant dans un traitement ayant lieu sur plusieurs ordinateurs simultanément. Elle repose sur des principes fondamentaux du système d'exploitation, ce qui permet de prendre en compte les tâches noyau elles-mêmes ainsi que tous les appels systèmes, sans en connaitre leur fonctionnement propre. Les bancs d'essai ont démontré qu'en utilisant une infrastructure de traçage efficiente, en sélectionnant minutieusement l'instrumentation et en ajustant l'environnement, il est possible d'effectuer cette analyse tout en altérant faiblement les caractéristiques temporelles des programmes observés. Les cas d'utilisation ont démontré que cette analyse s'applique de manière universelle, indépendamment des moteurs d'exécution, du langage de programmation, des librairies et des intergiciels. Le résultat permet de non seulement comprendre la latence d'une application complexe, révélant des inefficacités autrement difficilement identifiables, mais aussi d'exposer son fonctionnement interne et ses interactions avec d'autres tâches du point de vue du système d'exploitation.

8.1 Limitations de la solution proposée

La principale limitation de l'analyse du chemin actif distribué concerne les programmes basés sur une boucle d'évènement asynchrone. La fenêtre de blocage peut être scindée en plusieurs périodes plus courtes, qui retournent après l'expiration d'un minuteur. Le chemin actif retourné est correct du point de vue du système, mais moins utile pour le programmeur, car le minuteur masque la cause réelle de l'attente d'un autre évènement. Une euristique est nécessaire pour prendre en compte cette situation. La difficulté est de trouver une approche avec des hypothèses sous-jacentes minimales sur le fonctionnement interne du programme, pour éviter de réduire la généralité de l'analyse.

Une limitation technique concerne la consommation de mémoire de l'algorithme d'analyse, qui est linéaire en fonction de la taille de la trace. Pour traiter des traces de grande taille, deux solutions sont possibles. On pourrait traiter le graphe en continu pour en extraire le chemin actif, puis supprimer les noeuds qui ne seront assurément plus accédés. Le désavantage est que la trace doit être relue pour le calcul du chemin actif d'une autre tâche. L'autre solution possible serait de créer un index sur disque. Par exemple, chaque intervalle du graphe pourrait être inscrit dans un arbre d'historique [22]. Il faudrait par contre étendre une telle structure de données pour prendre en considération les arcs verticaux.

8.2 Améliorations futures

Le chemin critique d'une communication TCP/IP a être utilisé pour déterminer lequel du client ou du serveur contribue au délai d'exécution [64]. Cette technique pourrait servir à déterminer quel ordinateur est impliqué dans le chemin critique, puis la méthode du chemin actif pourrait être appliquée localement. Comme la fenêtre de transfert de TCP pour les données volumineuses forme un pipeline, il serait intéressant de comparer les résultats obtenus à ceux se limitant à l'intervalle de blocage.

Le traçage des verrous permettrait de déterminer le début de l'accès à une section critique. Leur instrumentation représente un défi, car il s'agit d'une source d'évènements à débit très élevé, ce qui pourrait être significativement couteux, alors que seulement la période de contention est requise aux fins de l'analyse. De manière générale, pour une mise à l'échelle efficace, la probabilité de contention sur un verrou doit demeurer faible donc, selon cette présomption, le critère de la contention permettrait de réduire substantiellement le surcout. Une méthode à évaluer serait de conserver le temps de la prise du verrou et d'enregistrer un évènement seulement si, lors de la sortie de la section critique, une autre tâche est en attente. Pour réduire davantage le surcout, seules les contentions dépassant un certain seuil pourraient être enregistrées. En combinant ces techniques, l'approximation du chemin actif aurait le potentiel d'être plus précise.

L'analyse se produit actuellement hors-ligne, à l'aide d'une trace enregistrée. Une amélioration pertinente consisterait à effectuer l'analyse en ligne, sous forme de statistiques. Les métriques du chemin actif d'une tâche spécifique seraient affichées en continu, sans nécessiter l'écriture ou le transfert de la trace.

RÉFÉRENCES

[1] A. S Tanenbaum et M. Van Steen, *Distributed systems : principles and paradigms*, Second Edition. Pearson Prentice Hall, 2006, isbn : 0-13-239227-5.

[2] N. Tolia, D. G. Andersen et M. Satyanarayanan, "Quantifying interactive user experience on thin clients", *Computer*, t. 39, n° 3, p. 46–52, 2006. adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1607949 (visité le 29 avr. 2013).

[3] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson et B. Shneiderman, "Determining causes and severity of end-user frustration", *International journal of human-computer interaction*, t. 17, n° 3, p. 333–356, 2004. adresse : http://www.tandfonline.com/doi/abs/10.1207/s15327590ijhc1703_3 (visité le 11 fév. 2014).

[4] J. L. Hennessy et D. A. Patterson, *Computer architecture : a quantitative approach*. Elsevier, 2012. (visité le 20 mai 2015).

[5] M. Desnoyers, "Low-impact operating system tracing", Thesis, École Polytechnique de Montréal, Montréal, 2009.

[6] K. G. Lockyer, "An introduction to critical path analysis", 1969. adresse : http://cds.cern.ch/record/103617 (visité le 21 mai 2015).

[7] R. Sedgewick et K. Wayne, *Algorithms*, en. Addison-Wesley Professional, fév. 2011, isbn : 9780132762564.

[8] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang et Z. Zhang, "Precise, scalable, and online request tracing for multi-tier services of black boxes", *IEEE Transactions on Parallel and Distributed Systems*, n° 99, p. 1–1, 2010.

[9] M. Dagenais, K. Yaghmour, C. Levert et M. Pourzandi, "Software Performance Analysis", *Arxiv preprint cs/0507073*, 2005.

[10] M. Desnoyers et M. Dagenais, "The lttng tracer : A low impact performance and behavior monitor for gnu/linux", in *Proceedings of the Ottawa Linux Symposium*, t. 2006, 2006.

[11] M. Desnoyers et M. Dagenais, "Low disturbance embedded system tracing with linux trace toolkit next generation", in *ELC (Embedded Linux Conference)*, 2006.

[12] B. Poirier, R. Roy et M. Dagenais, “Unified Kernel and User Space Distributed Tracing for Message Passing Analysis”, in *Proceedings of the First International Conference on Parallel, Distributed and Grid Computing for Engineering*, B. Topping et P Ivanyi, éds., sér. Civil Comp Proceedings, 1st International Conference on Parallel, Distributed and Grid Computing for Engineering, Univ Pecs, Pollack Mihaly Fac Engn, Pecs, HUNGARY, APR 06-08, 2009, CIVIL COMP PRESS, 2009, p. 218–234, isbn : 978-1-905088-29-4.

[13] P. Fournier, M. Desnoyers et M. Dagenais, “Combined tracing of the kernel and applications with LTTng”, in *Proceedings of the 2009 Linux Symposium*, 2009.

[14] E. Clément et M. Dagenais, “Traces synchronization in distributed networks”, *Journal of Computer Systems, Networks, and Communications*, p. 5, 2009.

[15] B. Poirier, R. Roy et M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events”, *ACM SIGOPS Operating Systems Review*, t. 44, nº 3, p. 75–87, 2010.

[16] H. Marouani et M. Dagenais, “Comparing high resolution timestamps in computer clusters”, in *Canadian Conference on Electrical and Computer Engineering, 2005*, mai 2005, p. 400–403. doi : `10.1109/CCECE.2005.1556956`.

[17] D. Toupin, “Using tracing to diagnose or monitor systems”, *IEEE Software*, t. 28, nº 1, p. 87–91, 2011.

[18] A. S. Sendi, M. Jabbarifar, M. Shajari et M. Dagenais, “FEMRA : Fuzzy Expert Model for Risk Assessment”, IEEE, 2010, p. 48–53, isbn : 978-1-4244-6726-6. doi : `10.1109/ICIMP.2010.15`. (visité le 18 mar. 2012).

[19] G. Matni et M. Dagenais, “Automata-based approach for kernel trace analysis”, in *Canadian Conference on Electrical and Computer Engineering, 2009. CCECE ’09*, mai 2009, p. 970–973. doi : `10.1109/CCECE.2009.5090273`.

[20] P. Fournier et M. Dagenais, “Analyzing blocking to debug performance problems on multi-core systems”, *ACM SIGOPS Operating Systems Review*, t. 44, nº 2, p. 77–87, 2010.

[21] M. Desnoyers et M. Dagenais, “OS tracing for hardware, driver and binary reverse engineering in Linux”, *CodeBreakers Journal*, t. 1, nº 2, 2006.

[22] A. Montplaisir-Goncaves, “Stockage sur disque pour accès rapide d’attributs avec intervalles de temps”, Mémoire, École Polytechnique de Montréal, Montréal, 2011.

[23] E. Koskinen et J. Jannotti, “Borderpatrol : isolating events for black-box tracing”, in *ACM SIGOPS Operating Systems Review*, t. 42, 2008, p. 191–203.

[24] P. Reynolds, J. Wiener, J. Mogul, M. Aguilera et A. Vahdat, “WAP5 : black-box performance debugging for wide-area systems”, in *Proceedings of the 15th international conference on World Wide Web*, 2006, p. 347–356.

[25] S. Agarwala, F. Alegre, K. Schwan et J. Mehalingham, “E2eprof : Automated end-to-end performance management for enterprise systems”, in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, p. 749–758.

[26] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier et A. Simma, “Constellation : automated discovery of service and host dependencies in networked systems”, *TechReport, MSR-TR-2008-67*, 2008.

[27] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds et A. Muthitacharoen, “Performance debugging for distributed systems of black boxes”, in *ACM SIGOPS Operating Systems Review*, t. 37, 2003, p. 74–89.

[28] D. Reed, P. Roth, R. Aydt, K. Shields, L. Tavera, R. Noe et B. Schwartz, “Scalable performance analysis : The Pablo performance analysis environment”, in *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, 1993, p. 104–113.

[29] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan et C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure”, *Google Research*, 2010.

[30] M. Casas, R. Badia et J. Labarta, “Automatic analysis of speedup of MPI applications”, in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, p. 349–358.

[31] W. Nagel, A. Arnold, M. Weber, H. Hoppe et K. Solchenbach, *VAMPIR : Visualization and analysis of MPI resources*. Citeseer, 1996.

[32] D. Gunter, B. Tierney, B. Crowley, M. Holding et J. Lee, “Netlogger : A toolkit for distributed system performance analysis”, in *Proceedings of 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000, p. 267–273.

[33] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez et G. Ganger, “Stardust : tracking activity in a distributed storage system”, in *ACM SIGMETRICS Performance Evaluation Review*, t. 34, 2006, p. 3–14.

[34] K. Hori et K. Yoshihara, “Pinpointing patch impact test targets using kernel tracing”, in *Network and Service Management (CNSM), 2010 International Conference on*, 2010, p. 326–329.

[35] R. Isaacs, P. Barham, J. Bulpin, R. Mortier et D. Narayanan, “Request extraction in Magpie : events, schemas and temporal joins”, in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004, p. 17.

[36] P. Barham, R. Isaacs, R. Mortier et D. Narayanan, “Magpie : Online modelling and performance-aware systems”, in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 15–15.

[37] M. Chen, E. Kiciman, A. Accardi, A. Fox et E. Brewer, “Using runtime paths for macroanalysis”, in *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, 2003, p. 14–14.

[38] C. K Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J Reddi et K. Hazelwood, “Pin : building customized program analysis tools with dynamic instrumentation”, in *ACM SIGPLAN Notices*, t. 40, 2005, p. 190–200.

[39] S. Chiba, “Javassist : Java bytecode engineering made simple”, *Java Developer's Journal*, t. 9, nº 1, 2004.

[40] C. C. Williams et J. K. Hollingsworth, “Interactive binary instrumentation”, in *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Citeseer, 2004. adresse : `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.9623&rep=rep1&type=pdf` (visité le 22 mai 2015).

[41] O. Zaki, E. Lusk, W. Gropp et D. Swider, “Toward scalable performance visualization with Jumpshot”, *International Journal of High Performance Computing Applications*, t. 13, nº 3, p. 277–288, 1999.

[42] S. Mysore, B. Mazloom, B. Agrawal et T. Sherwood, “Understanding and visualizing full systems with data flow tomography”, *ACM SIGPLAN Notices*, t. 43, nº 3, p. 211–221, 2008.

[43] R. Fonseca, G. Porter, R. Katz, S. Shenker et I. Stoica, “X-trace : A pervasive network tracing framework”, in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007, p. 20–20.

[44] B. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar et R. Chang, “vPath : precise discovery of request processing paths from black-box observations of thread and network activities”, in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009, p. 19–19.

[45] A. Chanda, A. Cox et W. Zwaenepoel, “Whodunit : Transactional profiling for multi-tier applications”, in *ACM SIGOPS Operating Systems Review*, t. 41, 2007, p. 17–30.

[46] N. Froyd, J. Mellor-Crummey et R. Fowler, “Low-overhead call path profiling of unmodified, optimized code”, in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, p. 81–90.

[47] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng et B. Sang, “Precise request tracing and performance debugging for multi-tier services of black boxes”, in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, IEEE, 2009, p. 337–346. (visité le 16 juin 2015).

[48] A. Mirgorodskiy et B. Miller, “Diagnosing distributed systems with self-propelled instrumentation”, *Middleware 2008*, p. 82–103, 2008.

[49] C. Yang et B. Miller, “Critical path analysis for the execution of parallel and distributed programs”, in *8th International Conference on Distributed Computing Systems*, 1988, p. 366–373.

[50] J. Hollingsworth, “An online computation of critical path profiling”, in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1996, p. 11–20.

[51] A. Saidi, N. Binkert, S. Reinhardt et T. Mudge, “Full-system critical path analysis”, in *IEEE International Symposium on Performance Analysis of Systems and software (ISPASS)*, 2008, p. 63–74.

[52] G. Franks, T. Al-Omari, M. Woodside, O. Das et S. Derisavi, “Enhanced modeling and solution of layered queueing networks”, *IEEE Transactions on Software Engineering*, t. 35, n° 2, p. 148–161, 2009.

[53] G. Franks, D. Lau et C. Hrischuk, “Performance measurements and modeling of a java-based session initiation protocol (SIP) application server”, in *Proceedings of International Symposium on Architecting Critical Systems (ISARCS)*, 2011, p. 63–72.

[54] T. Israr, D. Lau, G. Franks et M. Woodside, “Automatic generation of layered queuing software performance models from commonly available traces”, in *Proceedings of the 5th international workshop on Software and performance*, 2005, p. 147–158.

[55] F. Ricciato et W. Fleischer, “Bottleneck detection via aggregate rate analysis : a real case in a 3g Network”, in *10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006, p. 1–4.

[56] T. Soulami, *Inside Windows Debugging*. Pearson Education, 2012. (visité le 16 juin 2015).

[57] B. Gregg et J. Mauro, *DTRACE : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011. (visité le 16 juin 2015).

[58] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston et B. Chen, *Architecture of systemtap : a Linux trace/probe tool*. 2005.

[59] I. M. et. al., *Perf Wiki*, juil. 2014. adresse : `https://perf.wiki.kernel.org/`.

[60] R. Inc., *Red Hat Enterprise Linux Developer Guide*. adresse : `http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Developer_Guide` (visité le 30 mar. 2012).

[61] M. Bligh, M. Desnoyers et R. Schultz, “Linux kernel debugging on google-sized clusters”, in *Proceedings of the Linux Symposium*, 2007, p. 29–40.

[62] N. Nethercote et J. Seward, “How to Shadow Every Byte of Memory Used by a Program”, in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, sér. VEE ’07, New York, NY, USA : ACM, 2007, p. 65–74, isbn : 978-1-59593-630-1. doi : `10.1145/1254810.1254820`. adresse : `http://doi.acm.org/10.1145/1254810.1254820` (visité le 21 nov. 2014).

[63] W. E. Cohen, “Tuning programs with OProfile”, *Wide Open Magazine*, t. 1, p. 53–62, 2004. adresse : `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.182.7491&rep=rep1&type=pdf` (visité le 11 fév. 2014).

[64] P. Barford et M. Crovella, “Critical path analysis of TCP transactions”, *ACM SIGCOMM Computer Communication Review*, t. 30, nº 4, p. 127–138, 2000.

[65] G. A Fink, V. Duggirala, R. Correa et C. North, “Bridging the host-network divide : survey, taxonomy, and solution”, in *Proceedings of the 20th Conference on Large Installation System Administration Conference*, 2006, p. 247–262.

[66] N. Joukov, A. Traeger, R. Iyer, C. P. Wright et E. Zadok, “Operating system profiling via latency analysis”, in *Proceedings of the 7th symposium on Operating systems design and implementation*, sér. OSDI ’06, Berkeley, CA, USA : USENIX Association, 2006, p. 89–102, isbn : 1-931971-47-1. adresse : `http://dl.acm.org/citation.cfm?id=1298455.1298465` (visité le 8 avr. 2012).

[67] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao et P. Dinda, “Panappticon : event-based tracing to measure mobile application and platform performance”, in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, IEEE, 2013, p. 1–10. adresse : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6659020` (visité le 21 juil. 2014).

[68] M. Broberg, L. Lundberg et H. Grahn, “Performance optimization using extended critical path analysis in multithreaded programs on multiprocessors”, *Journal of Parallel and Distributed Computing*, t. 61, nº 1, p. 115–136, 2001.

[69] R. Fowler, T. LeBlanc et J. Mellor-Crummey, “An integrated approach to parallel program debugging and performance analysis onlarge-scale multiprocessors”, in *ACM SIGPLAN Notices*, t. 24, 1988, p. 163–173.

[70] R. Love, *Linux kernel development*. Pearson Education, 2010.

[71] R. Sambasivan, A. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu et G. Ganger, “Diagnosing performance changes by comparing request flows”, in *Symposium on Networked Systems Design and Implementation*, 2011, p. 43–56.

[72] N. Nethercote et J. Seward, “Valgrind : A program supervision framework”, *Electronic notes in theoretical computer science*, t. 89, nº 2, p. 44–66, 2003. adresse : http://www.sciencedirect.com/science/article/pii/S1571066104810429 (visité le 11 fév. 2014).

[73] O. A. Inc., *Trail : rmi (the javatm tutorial)*. adresse : https://docs.oracle.com/javase/tutorial/rmi/ (visité le 10 nov. 2015).

[74] D. S. Foundation, *Django project*. adresse : https://www.djangoproject.com (visité le 10 nov. 2015).

[75] K. Sedgewick Robert and Wayne, *Algorithms*, English, 4th Edition. Addison-Wesley Professional, mar. 2011, isbn : 032157351X.

[76] S. McConnell, *Code complete, 2nd Edition*. Microsoft press, 2004. (visité le 24 avr. 2015).

[77] Intel, *Intel Architecture Software Developer’s Manual, Volume 3 : System Programming Guide*. Intel, 2015, t. 3. adresse : http://developer.intel.com/.

[78] S. V. Moore, “A comparison of counting and sampling modes of using performance monitoring hardware”, in *Computational Science—ICCS 2002*, Springer, 2002, p. 904–912. (visité le 24 avr. 2015).

[79] F. Nybäck, “Improving the support for ARM in the IgProf profiler”, thèse de doct., Aalto University, Espoo, Finland, 2014. (visité le 24 avr. 2015).

[80] A. V. Aho, M. S. Lam, R. Sethi et J. D. Ullman, *Compilers : principles, Techniques, and tools, 2nd ed.* Addison Wesley, 2007.

[81] I. Spectrum, *2014 ranking top 10 programming languages*. adresse : http://spectrum.ieee.org/computing/software/top-10-programming-languages (visité le 10 nov. 2015).

[82] *Systemtap*, avr. 2015. adresse : https://sourceware.org/systemtap/.

[83] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandevoorde, C. Waldspurger et W. Weihl, “Continuous profiling : where have all the cycles gone ?”, English, *ACM SIGOPS Operating Systems Review*, t. 31, nº 5, p. 1 –14, déc. 1997, issn : 0163-5980. doi : 10.1145/269005.266637. adresse : http://portal.acm.org/citation.cfm?id=266637 (visité le 1er mai 2015).

[84] J. Dongarra, K. London, S. Moore, P. Mucci et D. Terpstra, “Using PAPI for hardware performance monitoring on Linux systems”, in *Conference on Linux Clusters : The HPC Revolution*, t. 5, 2001.

[85] M. M. Tikir et J. K. Hollingsworth, “Using hardware counters to automatically improve memory performance”, in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, IEEE, 2004, p. 46–46. (visité le 24 avr. 2015).

[86] R. Azimi, M. Stumm et R. W. Wisniewski, “Online performance analysis by statistical sampling of microprocessor performance counters”, in *Proceedings of the 19th annual international conference on Supercomputing*, ACM, 2005, p. 101–110. (visité le 24 avr. 2015).

[87] G. Ammons, T. Ball et J. Larus, “Exploiting hardware performance counters with flow and context sensitive profiling”, *ACM Sigplan Notices*, t. 32, nº 5, p. 85–96, 1997.

[88] P. Hofer et H. Mössenböck, “Fast Java profiling with scheduling-aware stack fragment sampling and asynchronous analysis”, in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform : Virtual machines, Languages, and Tools*, ACM, 2014, p. 145–156. (visité le 6 mai 2015).

[89] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais et J. Walpole, “User-level implementations of read-copy update”, *Parallel and Distributed Systems, IEEE Transactions on*, t. 23, nº 2, p. 375–382, 2012.

[90] D. M. Diez, C. D. Barr et M. Cetinkaya-Rundel, *OPENINTRO statistics*, 2nd Edition. CreateSpace independent publishing platform, 2012. (visité le 16 mai 2015).

[91] J. L. Bentley, *Writing efficient programs*. Prentice-Hall, Inc., 1982. (visité le 24 avr. 2015).

[92] SGI, *REACT Real-Time for Linux Programmer Guide*. Silicon Graphics International Corp., 2014.

[93] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais et M. Desnoyers, “Recovering System Metrics from Kernel Trace”, in *Linux Symposium*, 2011, p. 109.

[94] F. Giraldeau et M. R. Dagenais, “Teaching Operating Systems Concepts with Execution Visualization”, 2014. adresse : `http://www.asee.org/file_server/papers/attachment/file/0004/4481/paper-asee13-v5.pdf` (visité le 26 juin 2015).

[95] F. Pierre Doray, “Analyse de variations de performance par comparaison de traces d’exécution”, Master thesis, École Polytechnique de Montréal, 2015.

[96] M. Gebai, F. Giraldeau et M. R. Dagenais, “Fine-grained preemption analysis for latency investigation across virtual machines”, *Journal of Cloud Computing*, t. 3, nº 1, p. 1–15, 2014. adresse : `http://link.springer.com/article/10.1186/s13677-014-0023-3` (visité le 26 juin 2015).

[97] R. Beamonte, F. Giraldeau et M. Dagenais, “High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems”, in *Proceedings of the 14th Real-Time Linux Workshop. OSADL*, 2012. adresse : `http://step.polymtl.ca/~fgiraldeau/resources/papers/rtlws14_paper.pdf` (visité le 26 juin 2015).

[98] Z. Zhou, “Teaching an Operating System Course to CET/EET Students”, in *Proceedings of the 2009 American Society for Engineering Education Annual Conerence*, 2009.

[99] M. D. Filsinger, “Writing Simulation Programs as a Tool for Understanding Internal Computer Processes”, in *Proceedings of the 2004 American Society for Engineering Education Annual Conerence*, 2004.

[100] S. F. Barrett, D. J. Pack et C. Straley, “Real-Time operating Systems : A Visual Simulator”, in *Proceedings of the 2004 American Society for Engineering Education Annual Conerence*, 2004.

[101] L. P. Maia, F. B. Machado et A. C. Pacheco Jr., “A Constructivist Framework for Operating Systems Education : A Pedagogic Proposal Using the SOsim”, *SIGCSE Bull.*, t. 37, nº 3, p. 218–222, juin 2005, issn : 0097-8418. doi : `10.1145/1151954.1067505`.

[102] W. A. Christopher, S. J. Procter et T. E. Anderson, “The Nachos instructional operating system”, in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, 1993, p. 4–4. adresse : `http://dl.acm.org/citation.cfm?id=1267307` (visité le 1er jan. 2014).

[103] C. L. Anderson et M. Nguyen, “A survey of contemporary instructional operating systems for use in undergraduate courses”, *Journal of Computing Sciences in Colleges*, t. 21, nº 1, p. 183–190, 2005. adresse : `http://dl.acm.org/citation.cfm?id=1088822` (visité le 23 juin 2015).

[104] T. Bower, “Using Linux Kernel Modules for Operating Systems Class Projects”, in *Proceedings of the 2006 American Society for Engineering Education Annual Conerence*, 2006.

[105] M. Desnoyers et M. Dagenais, “Teaching real operating systems with the lttng kernel tracer”, in *ASEE Annual Conference and Exposition, Conference Proceedings*, Pittsburg, PA, United states, 2008.

[106] C. T. Fosnot et R. S. Perry, “Constructivism : A psychological theory of learning”, *Constructivism : Theory, perspectives, and practice*, p. 8–33, 1996. adresse : `http://rsperry.com/fosnotandperry.pdf` (visité le 27 déc. 2013).