UNIVERSITÉ DE MONTRÉAL

MÉTHODES EFFICACES DE PARALLÉLISATION DE L'ANALYSE DE TRACES NOYAU

FABIEN REUMONT-LOCKE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2015

© Fabien Reumont-Locke, 2015.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODES EFFICACES DE PARALLÉLISATION DE L'ANALYSE DE TRACES NOYAU

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

Mme NICOLESCU Gabriela, Doctorat, présidente
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BOYER François-Raymond, Ph. D., membre

REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur de recherche, Michel Dagenais, qui m'a permis de réussir dans mes objectifs et de me surpasser. C'est grâce à son expérience et à ses conseils que j'ai pu développer mes connaissances et grâce à son encadrement que j'ai pu m'y retrouver dans le domaine de la recherche.

Je souhaite remercier mes collègues du laboratoire DORSAL, non seulement pour leur appui, mais aussi pour les bons moments passés ensemble. Un grand merci à Francis pour m'avoir tant appris et aidé. Merci à Geneviève et Julien pour vos réponses à mes problèmes techniques. Merci aussi à Suchakra, mon voisin de bureau, nos conversations vont me manquer !

Merci à Ericsson et au Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour leur soutien financier, sans lequel cette recherche n'aurait pas pu voir le jour.

Je tiens à remercier tout particulièrement ma grande famille pour toute l'aide et le soutien qu'ils m'ont donnés. Merci à mon père, Vincent, qui a été le premier à me donner la passion de la science, et à ma mère, Mary Ann, qui m'a toujours appuyé dans mes projets. Merci à ma sœur Florence, qui m'a convaincu de faire ma maîtrise lorsque j'avais des doutes. Finalement, merci à Félix, pour ton soutien et ta patience ; je n'aurais pas réussi sans toi.

RÉSUMÉ

Les architectures hautement parallèles sont de plus en plus répandues, non seulement dans les systèmes haute-performance, mais aussi dans les ordinateurs grand public. La détection et la résolution de problèmes lors de l'exécution parallèle de logiciels sur ces types de systèmes sont des tâches complexes auxquelles les outils classiques de débogage ne sont pas adaptés. Des études précédentes ont démontré que le traçage s'avère être une solution efficace à la résolution de problèmes dans des systèmes hautement parallèles. Cependant, l'augmentation du nombre d'unités parallèles dans les systèmes tracés cause aussi une augmentation de la quantité de données générées par le traçage. Les architectures distribuées ne font qu'exacerber ce problème puisque chaque nœud peut contenir plusieurs processeurs multicœurs.

Les données de trace doivent être analysées par un outil d'analyse de traces afin de pouvoir extraire les métriques importantes qui permettront de résoudre les problèmes. Or, les outils d'analyse de traces disponibles sont conçus de manière à s'exécuter séquentiellement, sans tirer avantage des capacités d'exécution parallèle. Nous nous retrouvons donc face à une différence de plus en plus grande entre la quantité de données produite par le traçage et la vitesse à laquelle ces données peuvent être analysées.

La présente recherche a pour but d'explorer l'utilisation du calcul parallèle afin d'accélérer l'analyse de traces. Nous proposons une méthode efficace de parallélisation de l'analyse de traces qui supporte la mise à l'échelle. Nous nous concentrons sur les traces en format CTF générées par le traceur LTTng sur Linux.

La solution présentée prend en compte des facteurs clés de la parallélisation efficace, notamment un bon équilibrage de la charge, un minimum de synchronisation et une résolution efficace des dépendances de données. Notre solution se base sur des aspects clés du format de trace CTF afin de créer des charges de travail équilibrées et facilement parallélisables. Nous proposons aussi un algorithme permettant la détection et la résolution de dépendances de données, pendant l'analyse de traces, qui utilise au minimum le verrouillage et la synchronisation entre les fils d'exécution. Nous implémentons trois analyses de traces parallèles à l'aide de cette solution : la première permet de compter les événements d'une trace, la seconde de mesurer le temps CPU utilisé par processus et la troisième de mesurer la quantité de données lues et écrites par processus. Nous utilisons ces analyses afin de mesurer la mise à l'échelle possible de notre solution, en utilisant le concept d'efficacité parallèle.

Puisque les traces peuvent être potentiellement très volumineuses, elles ne peuvent être gardées en mémoire et sont donc lues à partir du disque. Afin d'évaluer l'impact de la performance des

périphériques de stockages sur notre implémentation parallèle, nous utilisons un programme simulant des charges de travail sur le CPU et sur le disque, typiques de l'analyse de traces. Nous évaluons ensuite la performance de ce programme sur plusieurs types de périphériques de stockage, tels que des disques durs et des disques SSD, afin de démontrer que la performance de l'analyse parallèle de traces n'est pas gravement limitée par les accès au disque, surtout avec des périphériques de stockage modernes. Nous utilisons aussi ce programme afin d'évaluer l'effet d'améliorations futures au décodage de la trace, sur la mise à l'échelle de l'analyse parallèle.

Notre solution offre une efficacité parallèle au-dessus de 56% jusqu'à 32 cœurs, lors de l'exécution de l'analyse de traces parallèle, ce qui signifie une accélération de 18 fois par rapport au temps de traitement séquentiel. De plus, les résultats de performance obtenus à partir du programme de simulation confirment que l'efficacité parallèle n'est pas sérieusement affectée par les accès au disque lorsque des périphériques de type SSD sont utilisés. Cette observation tient d'ailleurs même lorsque le décodage de la trace est plus rapide. Certains facteurs qui nuisent à la mise à l'échelle sont dus au modèle séquentiel de la bibliothèque de lecture de traces et peuvent être réglés avec une refonte de celle-ci, tandis que d'autres proviennent de goulots d'étranglement au sein du module de gestion de la mémoire du noyau et pourraient être améliorés ou contournés.

ABSTRACT

Highly parallel computer architectures are now increasingly commonplace, whether in commercial or consumer-grade systems. Detecting and solving runtime problems, in software running in a parallel environment, is a complicated task, where classic debugging tools are of little help. Previous research has shown that tracing offers an efficient and scalable way to resolve these problems. However, as the number of parallel units in the traced system increases, so does the amount of data generated in the trace. This problem also compounds when tracing distributed systems, where each individual node may have many-core processors.

Trace data has to be analyzed by a trace analysis tool, in order to extract significant metrics which can be used to resolve problems. However, the current trace analysis tools are designed for serial analysis on a single thread. We therefore have an ever widening gap between the amount of data produced in the trace and the speed at which we can analyse this data.

This research explores the use of parallel processing in order to accelerate trace analysis. The aim is to develop an efficient and scalable parallel method for analyzing traces. We focus on traces in the CTF format, generated by the LTTng tracer on Linux.

We present a solution which takes into account key factors of parallelization, such as good load balancing, low synchronization overhead and an efficient resolution of data dependencies. Our solution uses key aspects of the CTF trace format to create balanced, parallelizable workloads. We also propose an algorithm to detect and resolve data dependencies during trace analysis, with minimal locking and synchronization. Using this solution, we implement three trace analyses (counting events; measuring CPU time per-process; measuring amount of data read and written per-process) which we use in order to assess the scalability in terms of parallel efficiency.

Traces, being potentially very large, are not kept entirely in memory and must be read from disk. In order to assess the effect of the speed of storage devices on the parallel implementation of trace analysis, we create a program that simulates the CPU and I/O workloads typical of trace analysis. We then benchmark this program on various storage devices (e.g. HDD, SSD, etc.) in order to show that parallel trace analysis is not seriously hindered by I/O-boundedness problems, especially with modern storage hardware. We also use this program in order to assess the effect of future improvements in trace decoding on the analysis.

Our solution shows parallel efficiency above 56% up to 32 cores, when running the parallel trace analyses, which translates to a speedup of 18 times the serial speed. Furthermore,

benchmarks on the simulation program confirm that these efficiencies are not seriously affected by disk I/O on solid state devices, even in the case of faster trace decoding. Some factors affecting scalability are found within the serial design of the tracing library and can be fixed by a re-design, while others come from bottlenecks within the memory management unit of the kernel which could be improved or worked around.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX xi

LISTE DES FIGURES xii

LISTE DES SIGLES ET ABRÉVIATIONS xiii

CHAPITRE 1 INTRODUCTION 1
- 1.1 Définitions et concepts de base 2
  - 1.1.1 Le traçage 2
  - 1.1.2 Analyses de traces 2
  - 1.1.3 Calcul parallèle 3
- 1.2 Éléments de la problématique 5
- 1.3 Objectifs de recherche 6
- 1.4 Plan du mémoire 6

CHAPITRE 2 REVUE DE LITTÉRATURE 7
- 2.1 Le traçage sous Linux 7
  - 2.1.1 Concepts du traçage 7
  - 2.1.2 Traceurs sous Linux 10
- 2.2 L'analyse de traces 12
  - 2.2.1 Concepts de l'analyse de traces 13
  - 2.2.2 Machine à états 13
  - 2.2.3 Système d'états 15
  - 2.2.4 Métriques système 15
  - 2.2.5 Formats de trace 16
- 2.3 Les modèles et outils de parallélisation 17
  - 2.3.1 Pthreads 18

2.3.2 OpenMP . . . 18
2.3.3 TBB . . . 19
2.3.4 OpenCL . . . 20
2.3.5 MPI . . . 21
2.3.6 MapReduce . . . 21
2.4 L'analyse de traces en parallèle . . . 22
2.4.1 Uniparallélisme . . . 22
2.4.2 Réexécution . . . 25
2.4.3 Exécution parallèle de machines à états . . . 26
2.4.4 Détection d'intrusion parallèle . . . 27
2.5 Conclusion de la revue de littérature . . . 29

CHAPITRE 3 MÉTHODOLOGIE . . . 31
3.1 Environnement . . . 31
3.1.1 Matériel . . . 31
3.1.2 Logiciel . . . 31
3.2 Analyses . . . 32
3.2.1 Comptage d'événements . . . 32
3.2.2 Analyse de temps CPU . . . 32
3.2.3 Analyse d'entrées/sorties . . . 33
3.2.4 Traces analysées . . . 33
3.3 Programme de simulation . . . 33
3.4 Tests d'efficacité . . . 33

CHAPITRE 4 ARTICLE 1: EFFICIENT METHODS FOR TRACE ANALYSIS PARALLELIZATION . . . 35
4.1 Abstract . . . 35
4.2 Introduction . . . 36
4.3 Related Work . . . 37
4.3.1 Tracing tools . . . 37
4.3.2 Parallel tools . . . 37
4.3.3 Parallel trace analysis . . . 38
4.4 Background . . . 40
4.4.1 CTF trace format . . . 40
4.4.2 Babeltrace trace reader . . . 41
4.4.3 Trace analysis model . . . 41
4.5 Proposed Solution . . . 42

4.5.1 Trace data partitioning 43
4.5.2 Resolving data dependencies 45
4.6 Experimental Results 48
4.6.1 Experimental methodology 48
4.6.2 Simulation program 49
4.6.3 Parallel trace analysis efficiency 54
4.7 Conclusion and future work 57

CHAPITRE 5 DISCUSSION GÉNÉRALE 59
5.1 Retour sur les résultats 59
5.2 Opérations concurrentes sur la mémoire 60
5.2.1 Résultats et traces 60
5.2.2 Source du problème 61
5.3 Autres solutions explorées pour l'analyse parallèle 62

CHAPITRE 6 CONCLUSION 64
6.1 Synthèse des travaux 64
6.2 Limitations de la solution proposée 65
6.3 Améliorations futures 65

RÉFÉRENCES 67

LISTE DES TABLEAUX

Tableau 3.1 Détails de la trace test 33
Table 4.1 Test storage devices specifications 52
Table 4.2 Benchmark results for trace analyses on PCIe SSD 57

LISTE DES FIGURES

Figure 2.1 Une machine à état pour l'état d'un processus 14
Figure 2.2 Paquets d'un flux d'événements CTF avec index de paquets 17
Figure 2.3 Exécution simplifiée d'une tâche MapReduce 22
Figure 2.4 Illustration du principe d'uniparallélisme 24
Figure 2.5 Calcul parallèle de machine à états 27
Figure 2.6 Parallélisation au niveau du flux 28
Figure 4.1 Event stream, packets and packet index 40
Figure 4.2 Current state updating as trace events are read 42
Figure 4.3 (a) Per-time range, (b) per-stream and (c) hybrid partitioning schemes, with gradient showing event density 44
Figure 4.4 Data dependencies between executions across streams and chunks . . . 48
Figure 4.5 Speedup of concurrent memory operations for increasing CPU workloads (number of iterations) 51
Figure 4.6 Parallel efficiency of simulation for various storage devices 52
Figure 4.7 Parallel efficiency as simulated trace decoding speed increases 54
Figure 4.8 Parallel efficiency of trace analyses for various storage devices 56
Figure 5.1 Effet de l'accélération du décodage de trace simulé sur l'accélération globale pour différents nombres de fils d'exécution 59
Figure 5.2 Trace d'exécution des opérations concurrentes sur la mémoire 61
Figure 5.3 Détails de la trace d'exécution montrant la sérialisation des appels à `munmap` 61

LISTE DES SIGLES ET ABRÉVIATIONS

LTTng	Linux Tracing Toolkit Next Generation
CTF	Common Trace Format
API	Application Programming Interface
OpenMP	Open Multi-Processing
TBB	Threading Building Blocks
MPI	Message Passing Interface
TSDL	Trace Stream Description Language
RCU	Read-Copy-Update
HDD	Hard Disk Drive
SSD	Solid State Drive
CPU	Central Processing Unit
GPU	Graphics Processing Unit

CHAPITRE 1 INTRODUCTION

Une trace noyau est un enregistrement de tous les événements produits à l'intérieur du noyau d'un système d'exploitation, comme les appels système, les accès au disque et au réseau, l'allocation de mémoire et la gestion des processus. Le traçage n'est pas limité qu'au noyau : les applications s'exécutant en mode utilisateur peuvent aussi être utilisées comme source d'événements de traces. Cependant, une trace détaillée est difficile à analyser sans traitement, puisqu'elle peut facilement contenir plusieurs centaines de millions d'événements.

L'analyse de traces permet d'obtenir d'importantes informations sur le système à partir des fichiers de trace. La plupart des analyses sont basées sur la lecture séquentielle d'une trace, tout en accumulant l'état du système à mesure que le temps avance. On peut, par exemple, obtenir un aperçu extrêmement détaillé de l'état de chaque processus (en exécution, bloqué, préempté, interrompu, etc.), avec une précision du degré de la nanoseconde, en lisant et en analysant les événements d'ordonnancement.

Cependant, l'arrivée de processeurs possédant un grand nombre de cœurs, ainsi que les architectures infonuagiques, ne fait qu'augmenter la quantité de données à traiter lors du traçage de ces systèmes. Il devient donc important, voire nécessaire, de s'assurer que l'analyse de traces peut s'effectuer de manière rapide et efficace. Il est possible d'améliorer la performance des différentes parties de l'analyse de la trace, que ce soit le décodage des événements ou l'analyse en soi, mais ces améliorations ne pourront pas suivre le rythme d'augmentation de la quantité de données à traiter. Il est donc nécessaire de se tourner vers des solutions permettant une plus grande mise à l'échelle.

De même que l'augmentation du nombre d'unités de calcul parallèles cause une augmentation des données à traiter, elle pourrait aussi être une partie de la solution face à ce problème : si l'on arrive à exploiter la puissance de calcul parallèle de ces architectures, peut-être serait-on capable de mettre à l'échelle de manière efficace des analyses qui, face à des traces de très grandes tailles, deviennent trop longues à traiter en temps raisonnable.

La présente recherche a pour but d'accélérer l'analyse de traces noyau en explorant différentes techniques de parallélisation. Le but est d'arriver à des méthodes permettant la mise à l'échelle horizontale, c'est-à-dire une mise à l'échelle en ajoutant des unités de calcul parallèles plutôt qu'en utilisant des unités plus performantes.

1.1 Définitions et concepts de base

1.1.1 Le traçage

Le traçage est un procédé visant l'enregistrement détaillé des événements d'un système ou d'une application. Il est apparenté à la journalisation, mais vise un but différent : alors que la journalisation sert à enregistrer des événements de haut niveau et relativement sporadiques (p. ex. erreurs ou avertissements), le traçage vise l'enregistrement détaillé de toutes les opérations du système tracé, et ce avec un surcoût minimum. Le résultat du traçage est une trace contenant une liste d'événements ordonnée chronologiquement qui peut ensuite être lue ou analysée par un programme d'analyse de traces.

Événement

Un événement de trace est identifié par un type et peut contenir une certaine quantité d'informations associées. Par exemple, un événement d'ordonnancement des tâches peut contenir l'identifiant du fil d'exécution qui a été choisi pour être exécuté. Comme le nombre d'événements d'une trace peut s'élever à plusieurs centaines de millions, la quantité d'information liée à un événement doit être soigneusement choisie afin de ne pas surcharger la trace. Par exemple, l'identifiant du processus ayant généré un événement n'a pas besoin d'être inclus dans l'événement, puisque celui-ci peut être récupéré grâce aux événements d'ordonnancement.

Flux d'événements

Une trace est souvent séparée en plusieurs flux d'événements, qui représentent des ensembles d'événements ayant une source commune. Le traceur LTTng, qui sera utilisé dans la présente recherche, sépare chaque événement d'après le processeur l'ayant généré. Une trace LTTng, dans le format CTF, contient donc autant de flux d'événements que de processeurs, séparés en différents fichiers.

1.1.2 Analyses de traces

Comme il est difficile de comprendre le contenu d'une trace en parcourant une liste d'événements, une trace est souvent traitée à l'aide d'un programme exécutant des analyses de traces. Ces analyses permettent de générer des rapports graphiques ou textuels sur l'activité du système pendant la trace : état des processus, quantité de données lues ou écrites, latences des interruptions, etc.

1.1.3 Calcul parallèle

Traditionnellement, l'exécution d'un programme sur un ordinateur se fait à travers un seul processeur exécutant les instructions de manière séquentielle. Cependant, il est maintenant commun dans les systèmes modernes d'avoir accès à plusieurs unités de calcul parallèles : processeurs multicœurs, multiprocesseurs, GPU, systèmes distribués, etc. Ces architectures permettent l'exécution parallèle et simultanée de plusieurs instructions. Elles apportent aussi leur lot de défis de programmation, que nous verrons plus loin.

Fils d'exécution

Une des primitives de base du calcul parallèle sur systèmes multicœurs ou multiprocesseurs est le fil d'exécution (*thread*). Chaque processus du système peut créer un certain nombre de fils d'exécution qui partageront toutes les ressources du programme, à l'exception des registres et de la pile, qui sont uniques à chaque fil. Ces fils peuvent ensuite s'exécuter sur une unité de calcul (processeur ou cœur) de manière simultanée. Puisqu'ils partagent le même espace mémoire, il est facile et relativement peu coûteux de créer de nouveaux fils.

Primitives de synchronisation

Afin de gérer les communications et les accès aux données entre les différents fils, il est nécessaire d'avoir accès à des primitives de synchronisation. Celles-ci sont d'habitude fournies par le système d'exploitation, et permettent, par exemple, d'assurer l'exclusion mutuelle (*mutex*) pour une section d'instructions. On évite ainsi les conditions de concurrence (*race condition*) et la corruption des données. Le sémaphore peut être vu comme une généralisation du mutex se comportant comme un compteur, permettant aux fils d'accéder à un ensemble de ressources. La barrière, quant à elle, permet de synchroniser l'exécution des fils en s'assurant que tous les fils soient rendus à un endroit précis de l'exécution avant de pouvoir continuer.

Défis du calcul parallèle

La parallélisation d'un programme séquentiel comporte certains défis. Les principaux défis rencontrés dans la présente recherche sont décrits ici :

- Équilibrage de la charge : cet aspect réfère à la répartition des calculs sur les unités parallèles. Il faut s'assurer que la charge de travail est répartie de manière égale afin que toutes les unités soient occupées en tout temps. Dans le cas d'une charge déséquilibrée, certaines unités seront inoccupées pendant que d'autres finissent leur travail.

- Partition des données : cet aspect consiste à déterminer la manière dont les données sont divisées entre les unités de calcul. Par exemple, dans le cas de données en 2 dimensions (p. ex. les pixels d'une image), il est possible de partitionner les données de plusieurs manières : par colonnes, par rangées, par régions carrées, etc. Il est important de choisir une division de données qui est logique pour le traitement à effectuer en plus d'être efficace en termes d'équilibrage de la charge et de communication entre les unités de calcul.
- Verrouillage et synchronisation : afin de préserver la cohérence des données et de gérer les communications entre les unités de calcul, il est nécessaire d'utiliser des verrous et des mécanismes de synchronisation. Ceux-ci permettent, par exemple, de protéger des données partagées par les fils d'exécutions ou de créer une attente entre des fils. Cependant, le verrouillage et la synchronisation sont à éviter le plus possible si l'on veut obtenir de hautes performances, puisqu'ils introduisent des périodes d'inactivité dans l'exécution.
- Dépendances des données : lorsque le traitement d'une tâche est séparé sur plusieurs unités de calcul, il est possible que ces calculs soient dépendants de certaines données auxquelles l'unité n'a pas accès. C'est le cas, par exemple, si cette donnée est calculée par une autre unité. Ces dépendances de données sont un point majeur de la parallélisation, puisqu'elles peuvent limiter le degré de parallélisme atteignable. Il est donc important de concevoir une solution minimisant l'impact de ces dépendances.

Ces concepts seront abordés à plusieurs reprises lors de la description de la solution proposée pour l'analyse de traces noyau en parallèle.

1.2 Éléments de la problématique

Les traceurs noyau sont maintenant en mesure de générer des traces sur des systèmes hautement parallèles, ainsi que sur des systèmes distribués, de manière hautement efficace. Cet aspect du traçage est très important, car il permet la détection et la résolution de problèmes dans des infrastructures à grande échelle, par exemple dans le domaine de l'infonuagique. En effet, il est difficile, voire impossible de détecter certains problèmes en utilisant des outils de débogage "classiques", tels qu'un débogueur. Cependant, l'utilisation du traçage sur de telles infrastructures veut aussi dire que la quantité de données tracées augmente au même rythme qu'augmente la quantité d'unités de calcul parallèles.

Cet aspect est surtout problématique lorsque l'on considère que les analyses de traces, indispensables afin de détecter et comprendre les problèmes enregistrés dans une trace, ne font pas usage de la puissance de calcul parallèle à notre disposition. Puisque l'exécution de ces analyses est séquentielle, pour le moment, on se retrouve face à une différence grandissante entre la puissance de calcul sérielle des analyses et la quantité de données à traiter. Cela veut dire qu'il faudra de plus en plus de temps pour traiter les traces de nos systèmes.

La parallélisation apparaît donc comme un moyen efficace afin d'améliorer la performance des analyses. Cependant, cette parallélisation n'est pas simple, et doit prendre en compte un nombre de problèmes (décrit à la section 1.1.3). En effet, l'analyse de traces est une tâche intrinsèquement séquentielle : les événements sont lus en ordre chronologique, et le traitement d'un événement a des répercussions sur l'analyse des événements subséquents. En d'autres mots, l'analyse de traces est une tâche se basant sur un système d'états global (*state system*), ce qui implique un grand nombre de dépendances de données.

De plus, puisque les traces d'un système peuvent être potentiellement très volumineuses, celles-ci sont stockées sur disque et ne sont pas entièrement lues et contenues dans la mémoire principale. Or, les accès au disque sont beaucoup plus lents que des accès à la mémoire, et ce par plusieurs ordres de magnitude. Puisque la parallélisation opère sur la partie du programme s'exécutant sur le processeur, ces accès mémoire peuvent être vus comme des sources de traitement séquentiel, puisque le disque ne peut exécuter qu'une requête à la fois. Cependant, de récentes avancées dans le domaine des disques SSD (*Solid State Drive*) pourraient changer la donne pour un certain nombre de programmes dont la performance repose sur les accès aux disques.

Il est donc nécessaire, afin de démontrer que la parallélisation est une solution efficace au problème de performance des analyses de traces, de non seulement mettre en place une méthode d'analyse parallèle, mais aussi de prouver que les problèmes connexes tels que la

lecture de données d'un disque ne mettent pas en péril la mise à l'échelle de notre solution.

1.3 Objectifs de recherche

La présente recherche a pour but de répondre à la question de recherche suivante :

> Est-il possible d'utiliser le calcul parallèle sur systèmes multiprocesseurs afin d'obtenir une meilleure performance ainsi qu'une bonne mise à l'échelle de l'analyse de traces noyau ?

Les objectifs de cette recherche sont les suivants :

— identifier un modèle d'analyse se prêtant à une parallélisation efficace, tout en restant applicable à plusieurs types d'analyses ;

— développer une méthode de parallélisation de l'exécution de l'analyse utilisant ce modèle, en tenant compte des contraintes d'équilibrage de la charge, de synchronisation et de dépendances de données ;

— tester l'efficacité parallèle de différentes analyses implémentées avec la solution développée ;

— valider les capacités de mise à l'échelle des méthodes sur différents périphériques de stockages ;

— valider les capacités de mise à l'échelle des méthodes en tenant compte d'éventuelles améliorations au décodage de la trace.

1.4 Plan du mémoire

Le chapitre 2 présentera une revue de littérature portant sur les domaines du traçage, de l'analyse de traces, des modèles et outils de parallélisation ainsi que de l'analyse de traces en parallèle. Le chapitre 3 portera sur la méthodologie utilisée afin d'atteindre les objectifs cités ci-haut. L'article de journal "Efficient Methods for Trace Analysis Parallelization" est inclus au chapitre 4, et contient le cœur des résultats de la présente recherche. Le chapitre 5 contient une discussion générale sur les résultats obtenus, ainsi que des résultats complémentaires. Enfin, le chapitre 6 contiendra une conclusion résumant et synthétisant l'ensemble de la recherche et de ses résultats, avant d'offrir un aperçu des possibles travaux futurs.

CHAPITRE 2 REVUE DE LITTÉRATURE

La présente revue de littérature a pour but d'explorer les différents domaines reliés à la parallélisation de l'analyse de traces. Les domaines explorés seront le traçage, l'analyse de traces, les techniques et modèles de parallélisation ainsi que des solutions à l'analyse de traces en parallèle.

2.1 Le traçage sous Linux

Le traçage est un moyen efficace d'obtenir de l'information précise sur un système en générant un enregistrement détaillé des événements s'étant produits sur celui-ci. Ces événements doivent être définis par des points de trace statiques ou dynamiques, en espace utilisateur ou au niveau du noyau du système d'exploitation. Ces caractéristiques seront abordées en premier dans les sections qui suivent, suivi d'une vue d'ensemble des traceurs disponibles sous le système d'exploitation Linux.

2.1.1 Concepts du traçage

Le traçage est défini comme l'enregistrement détaillé des événements survenus lors de l'exécution d'un système. Il se différencie de la journalisation (*logging*) par la quantité de données enregistrées ainsi que par le niveau de détail des événements : la journalisation vise des événements haut-niveau, tels qu'une connexion ou une erreur système, tandis que le traçage vise des événements bas-niveau, tels que les allocations mémoires ou les appels systèmes. Afin de pouvoir tracer l'exécution d'un système, il faut spécifier à quels moments de l'exécution un événement doit être enregistré. Ces moments peuvent être spécifiés de deux manières : soit dynamiquement, lors de l'exécution du système, ou bien statiquement, au moment de la compilation. Chaque méthode a ses avantages et ses inconvénients, et les différents traceurs peuvent faire usage de l'une ou l'autre (ou d'une combinaison) des deux techniques.

Instrumentation statique

L'instrumentation statique permet la définition de points de trace au moment de la compilation du système. Ainsi, ces points de trace feront partie du code source de l'exécutable final. Cette approche a principalement l'avantage d'être plus performante, puisque le point de trace est directement inclus dans le programme binaire et n'a pas besoin d'être ajouté à la volée. De plus, il est possible d'obtenir un surcoût quasi nul lorsque le système n'est pas tracé grâce à

l'utilisation de code automodifiant. Ce mécanisme permet la création d'un embranchement inconditionnel en remplaçant l'instruction d'embranchement par une instruction fictive (*no-op*). Lorsque le traçage est par la suite activé, il suffit de remplacer cette instruction par un saut inconditionnel vers le code de traçage, opération coûteuse, mais qui n'est effectuée qu'une seule fois.

Cependant, il devient plus difficile d'ajouter de nouveaux points de trace, puisqu'un accès au code source et une recompilation du programme sont nécessaires. Ainsi, l'instrumentation statique est adaptée à des modules utilisés à grande échelle (noyau du système d'exploitation, base de données, serveur web, etc.) et à des modules dont le code source est disponible, mais ne permet pas de tracer des applications que l'on ne peut pas recompiler.

Instrumentation dynamique

L'instrumentation dynamique permet d'insérer des points arbitraires dans l'exécution de programmes où l'on voudrait émettre un événement. Cette instrumentation peut être effectuée au moment de l'exécution du module, et ce sans avoir besoin d'un accès au code source et sans avoir à le recompiler. On obtient ainsi une plus grande flexibilité qu'avec l'instrumentation statique.

Cette flexibilité a toutefois un coût en terme de performance, principalement dû à l'ajout d'instructions de point d'arrêt (*breakpoint*), technique utilisée par la plupart des mécanismes d'instrumentation dynamique. Cependant, il est intéressant de noter certaines optimisations possibles, telles que l'optimisation par instruction de saut (*jump optimization*), qui remplace les instructions de point d'arrêt par des instructions de sauts, lorsque certaines conditions sont présentes, améliorant ainsi la performance de l'instrumentation dynamique (Hiramatsu, 2010).

Espaces de traçage

Lors de l'exécution d'un programme dans un environnement contrôlé par un système d'exploitation, il est utile de départager les deux espaces d'exécution que sont l'espace utilisateur et l'espace noyau. Cette ségrégation permet de stabiliser et de protéger le système en offrant un espace privilégié au système d'exploitation (ainsi qu'aux modules noyaux et certains pilotes de périphériques) et un espace protégé aux programmes de l'utilisateur. La plupart des traceurs offrent des fonctionnalités en fonction de l'espace visé.

Traçage utilisateur

Le traçage en espace utilisateur permet de recueillir des traces de l'exécution de programmes tels qu'une base de données, un navigateur web ou n'importe quel autre exécutable supervisé par le système d'exploitation. Ce type de traçage est utile lorsque l'on veut analyser en détail l'exécution d'une application, mais nécessite que celle-ci soit instrumentée, soit de manière statique ou dynamique. Certaines applications utilisateurs populaires, comme certains serveurs web ou bibliothèques standards, sont distribuées avec de l'instrumentation statique afin de faciliter leur traçage, mais une application bâtie à partir de zéro doit nécessairement être instrumentée avant de pouvoir être tracée en mode utilisateur.

Le système d'exploitation Linux n'offre pas de mécanisme standard pour l'insertion d'instrumentation statique dans les programmes utilisateurs. Certains traceurs en espace utilisateur offrent leurs propres outils, tels que le traceur LTTng. Pour ce qui est de l'instrumentation dynamique, le noyau Linux offre le mécanisme `uprobes` (Keniston et al., 2007) afin d'insérer de l'instrumentation dynamique, appelé sondes, à des endroits arbitraires dans les programmes et bibliothèques en espace utilisateur. Le mécanisme `uprobes` permet l'exécution de code dans le noyau Linux lorsqu'une sonde est rencontrée lors de l'exécution, permettant par exemple de générer un événement.

Traçage noyau

Le traçage en espace noyau permet d'obtenir des traces de l'exécution des différentes parties du système d'exploitation, telles que le système de gestion de la mémoire, l'ordonnanceur et les modules d'entrées-sorties. Ces informations ne sont pas seulement utiles pour analyser le noyau lui-même, mais aussi pour analyser les programmes utilisateurs, et ce sans avoir à instrumenter ceux-ci. En effet, une fois le noyau instrumenté, comme c'est le cas du noyau Linux, une trace noyau peut contenir une foule d'informations concernant le comportement des applications utilisateurs supervisées par le noyau, comme nous le verrons dans la section sur les analyses de traces.

Dans le noyau Linux, l'instrumentation nécessaire au traçage est fournie par les mécanismes suivants. La macro `TRACE_EVENT()` (Rostedt, 2010) permet d'ajouter des points de trace statiques à même le code source du noyau. Ces points de trace sont ensuite compilés avec le noyau, et permettent à des modules d'enregistrer des fonctions de rappel (*callback*) lors de leur exécution. L'avantage de cette infrastructure est que ces points de traces peuvent être activés et désactivés pendant l'exécution du système, avec un surcoût nul lorsqu'ils sont désactivés. Ce surcoût nul est en grande partie dû au mécanisme de code automodifiant décrit précédemment.

Quant à l'instrumentation dynamique, le mécanisme `kprobes` (Mavinakayanahalli et al., 2006) (sur lequel est basé le mécanisme `uprobes` mentionné ci-haut) permet l'insertion de sondes à des endroits arbitraires du noyau Linux, afin de déboguer ou de tracer celui-ci.

2.1.2 Traceurs sous Linux

ftrace

Le traceur ftrace, distribué avec le système d'exploitation Linux, permet le traçage statique et dynamique du noyau Linux (Rostedt, 2009). Il utilise entre autres les points de trace statiques `TRACE_EVENT()` ainsi que l'instrumentation dynamique offerte par `kprobes` afin de recueillir des traces du noyau. Son utilisation passe par le pseudo système de fichier *debugfs*, à travers duquel l'utilisateur peut envoyer des commandes afin d'activer des événements, démarrer le traçage et obtenir les traces. Comme cette interface n'est pas très conviviale, on peut aussi utiliser le programme `trace-cmd` afin de contrôler ftrace. La performance de ftrace repose sur l'utilisation de tampons circulaires très efficaces. Chaque processeur enregistre ses événements dans son propre tampon, ce qui permet de réduire la synchronisation nécessaire entre ceux-ci.

ftrace offre aussi une variété de modules d'analyses permettant de traiter les données tracées et de présenter une analyse à l'utilisateur. Par exemple, l'analyse `wakeup` montre les latences dans l'ordonnancement des processus. Le traceur ftrace est un traceur purement noyau : il n'est donc pas capable de tracer les applications en espace utilisateur. Ceci limite grandement les possibilités de tracer le comportement d'applications utilisateurs complexes.

Le traceur ftrace peut générer des traces soit dans un format lisible pour l'utilisateur, soit dans un format binaire correspondant au contenu des tampons circulaires utilisés.

SystemTap

Le traceur SystemTap est aussi distribué avec Linux, et permet le traçage statique et dynamique du noyau Linux (avec `TRACE_EVENT()` et `kprobes`) ainsi que le traçage dynamique en espace utilisateur avec `uprobes` (Eigler and Hat, 2006). Sa particularité réside dans l'utilisation de scripts, écrits dans un langage propre à SystemTap, qui permettent de rapidement accumuler et présenter l'information recueillie lors du traçage alors même que celui-ci s'exécute. Ces scripts sont d'abord compilés en modules qui sont ensuite insérés et exécutés dans le noyau Linux.

Les traces obtenues par SystemTap prennent la forme de texte envoyé sur la sortie standard. Ce texte est entièrement dépendant de l'analyse exécutée et, bien qu'il soit techniquement

possible de générer une trace en format binaire compact à partir d'un script, SystemTap n'offre aucun support pour ce genre de format.

SystemTap offre une grande flexibilité qui, quoiqu'utile pour un administrateur système cherchant rapidement la cause d'un problème dans un système, vient avec un coût en terme de performance non négligeable. De plus, la mise à l'échelle sur des machines multicœurs est connue pour être problématique en terme de performance lors du traçage avec SystemTap, à cause des mécanismes de synchronisation coûteux utilisés par le traceur (Desnoyers and Dagenais, 2012).

perf

Disponible avec Linux depuis la version 2.6.31, l'outil perf permet de récupérer de l'information sur les compteurs de performance du système pendant son exécution (de Melo, 2010). Ces compteurs peuvent représenter, par exemple, le nombre de fautes de page, de changements de contexte ou de défauts de cache. De plus, perf peut aussi accumuler des événements provenant des points de trace `TRACE_EVENT()`, des `kprobes` ou des `uprobes` afin de générer des traces en espace noyau et utilisateur.

Le fonctionnement des compteurs de performance de perf est basé sur l'échantillonnage : les compteurs sont accumulés jusqu'à ce qu'ils dépassent une certaine valeur, après quoi une interruption est générée afin de récupérer leur valeur. Pour ce qui est du traçage, perf utilise aussi des tampons circulaires, comme ftrace et LTTng, sauf que ceux-ci ne sont pas aussi optimisés et souffrent de problèmes de mise à l'échelle.

Le traceur perf génère des traces dans un format binaire optimisé pour la rétrocompatibilité. Ces traces peuvent être lues grâce à l'outil `perf-report(1)`, ou bien analysées avec l'aide d'une bibliothèque spécialisée.

LTTng

Le traceur LTTng permet l'acquisition de traces en espace noyau ainsi qu'en espace utilisateur sur la plateforme Linux (Desnoyers and Dagenais, 2006). Comme les traceurs vus précédemment, LTTng utilise l'instrumentation statique présente dans le noyau Linux (`TRACE_EVENT()`) ainsi que les kprobes afin de recueillir des événements noyau. Du côté utilisateur, dont la fonctionnalité est offerte par la composante LTTng-UST, des points de trace statiques propres à LTTng peuvent être insérés dans les exécutables utilisateurs. LTTng-UST ne supporte pas les `uprobes` pour le moment.

LTTng est optimisé afin de réduire au maximum l'impact sur les systèmes, en particulier ceux

fortement multicœurs, lorsque le traçage est activé. Pour ce faire, plusieurs optimisations sont utilisées, telles que l'enregistrement des événements dans des tampons circulaires propres à chaque processeur afin d'éviter le verrouillage, et l'utilisation du mécanisme Read-Copy-Update (RCU), qui permet une meilleure mise à l'échelle des opérations nécessitant une synchronisation (Desnoyers and Dagenais, 2012). C'est pour ces raisons qu'il est spécifiquement adapté au traçage de systèmes temps réels et embarqués (Beamonte et al., 2012).

LTTng utilise le concept de flux d'événements (*stream*), où les événements sont enregistrés dans un flux par processeur, similairement aux tampons circulaires décrits ci-haut. Une trace est donc constituée d'une ensemble de fichiers correspondants aux différents flux. Cet aspect est intéressant d'un point de vue de parallélisation, puisque chaque flux peut être vu comme un ensemble plus ou moins indépendant de données à traiter. Nous verrons plus tard comment cette partition des données est intéressante pour la parallélisation.

Un autre aspect intéressant de LTTng est sa capacité à créer des traces corrélant des événements à la fois provenant de l'espace noyau et de l'espace utilisateur. Ceci permet de faire des analyses précises sur des comportements complexes du système tracé (Fournier et al., 2009).

LTTng permet aussi plusieurs autres types d'utilisations, tels que le mode en ligne, qui permet de lire et analyser la trace à mesure que celle-ci est créée, que ce soit sur la même machine ou à travers le réseau, ou le mode "instantané" (*snapshot*), qui permet d'obtenir le contenu des tampons circulaires lorsque le système rencontre une certaine condition (p. ex. utilisation CPU à 100%). Ces différentes fonctionnalités, associées à son très faible impact sur le système, en font une solution intéressante pour des systèmes en production, par exemple.

2.2 L'analyse de traces

Le traçage génère une quantité potentiellement très importante d'événements. Ces événements peuvent être consultés à la manière d'un journal, sous la forme d'une liste chronologique. Bien qu'une telle liste puisse être utile dans certaines situations, il est plus souvent utile d'effectuer une analyse sur la trace afin d'en extraire des informations utiles et facilement consommables. Certains traceurs offrent des méthodes d'analyse intégrées, tandis que d'autres s'appuient sur des outils externes afin d'effectuer les analyses voulues. Les prochaines sections discuteront des différentes méthodes d'analyse ainsi que certains modèles récurrents.

2.2.1 Concepts de l'analyse de traces

Analyse en ligne

L'analyse en ligne consiste à effectuer l'analyse à mesure que le système est tracé. Par exemple, le traceur SystemTap offre un langage de script permettant l'exécution de code d'analyse lorsque certaines conditions sont remplies (p. ex. point de trace, prédicat, etc.). Cette approche a l'avantage de simplifier l'utilisation du traceur : les analyses peuvent être distribuées sous forme de scripts, permettant ainsi à l'utilisateur de tracer le système, d'analyser la trace et d'obtenir les résultats de manière transparente. Cependant, une analyse en ligne peut causer des latences difficiles à prédire, du fait de la surcharge changeante d'une analyse à une autre. Ces latences peuvent occasionner des changements dans le comportement du système tracé, surtout lors de l'exécution de systèmes temps-réel. Par exemple, une analyse complexe, telle qu'une analyse de la pile d'appels à chaque événement, peut occasionner de fortes latences et perturber l'exécution du système.

Analyse hors-ligne

L'analyse hors-ligne permet l'exécution de l'analyse après le traçage du système. Le traceur LTTng utilise cette approche : les événements sont enregistrés sans traitement dans une trace, puis l'utilisateur peut analyser cette trace en utilisant divers outils d'analyse. Cette approche a l'avantage de causer une surcharge mineure et prévisible sur le système : en effet, la surcharge par événement est plus facile à prédire lorsque l'on n'exécute pas du code quasi arbitraire à chaque événement.

2.2.2 Machine à états

L'analyse hors-ligne de traces noyau peut servir à détecter des problèmes complexes dans le système, sans affecter la performance lors de son exécution, comme c'est le cas lorsque l'on utilise des analyses en ligne. Ces analyses peuvent être représentées efficacement sous forme de machines à états (Matni and Dagenais, 2009). Ces machines à états sont définies par un ensemble d'états, un ensemble de transitions d'un état à un autre ainsi qu'une ou plusieurs actions par transition. Les transitions sont déclenchées par certains événements, et les actions s'y rattachant peuvent servir à mettre à jour certaines informations ou émettre des informations à l'utilisateur.

Par exemple, on pourrait chercher à effectuer une analyse visant à vérifier une contrainte de temps réel : est-ce qu'un certain processus, s'exécutant périodiquement, dépasse un certain

temps d'exécution pour certaines périodes ? Cette machine à état comporterait deux états : soit le processus s'exécute, soit il est inactif. Les transitions seraient définies par les événements d'ordonnancement, et s'accompagneraient d'une sauvegarde de l'estampille de temps au moment de l'événement. On pourrait ainsi comparer ces estampilles de temps et émettre un avertissement si le seuil de temps a été dépassé pour une exécution périodique.

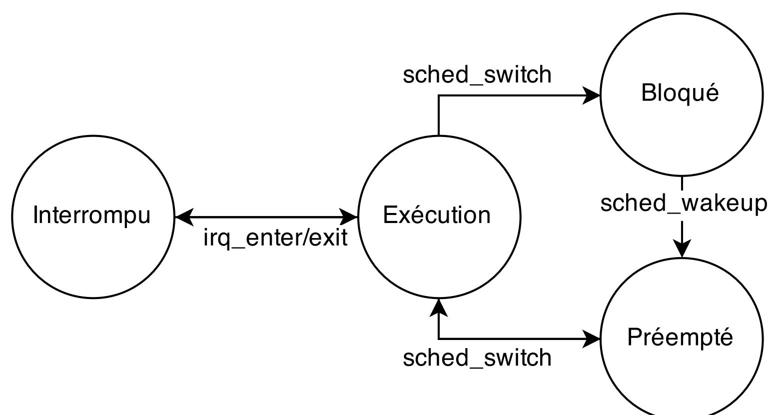

Figure 2.1 Une machine à état pour l'état d'un processus

Cette approche à l'avantage d'être générique, s'adaptant à une grande variété d'analyses. L'utilisateur souhaitant créer sa propre analyse n'aurait qu'à la définir en tant que machine à état, possiblement en utilisant un langage adapté ou une interface graphique, puis l'exécuter sur des traces acquises.

Une caractéristique intéressante de l'analyse par machines à états est qu'une seule lecture de la trace peut servir à exécuter un grand nombre d'analyses distinctes. Comme la plupart des analyses décrites dans l'article ne partagent pas d'information, il est donc possible d'exécuter plusieurs machines indépendamment les unes des autres. D'un point de vue de parallélisation, cela signifie que chaque machine pourrait être exécutée parallèlement, à mesure que les événements de la trace sont lus. Cependant, cette approche souffre du fait qu'une grande partie de l'analyse de traces consiste à décoder les événements, comme nous le verrons plus loin. Le gain en performance lié à la parallélisation des exécutions des machines à états serait donc minime, puisqu'une grande proportion du programme resterait sérielle. De plus, les travaux plus récents en terme d'analyses utilisent un état global partagé pour une trace, sur lequel se basent ensuite les différentes analyses. C'est ce système d'état qui sera abordé dans la prochaine section.

2.2.3 Système d'états

Une trace d'un système permet de voir le comportement de celui-ci dans le temps. Il est donc possible de modéliser celui-ci sous la forme d'un système d'états, avec un état global qui peut être récupéré pour chaque instant dans le temps. Montplaisir et al. (2013) proposent une méthode permettant de bâtir ce système d'état à partir d'une trace.

Cette méthode est basée sur la présence d'attributs, qui contiennent par exemple l'état d'un processus, l'identifiant du processus s'exécutant sur un processeur ou bien la quantité de données lues d'un fichier. La valeur de ces attributs est représentée sous forme d'intervalles de temps : par exemple, un processus peut entrer dans l'état "bloqué" au temps `t1` et en sortir au temps `t2`. Ces intervalles sont ensuite stockés dans une structure de données en arbre enregistrée sur le disque, nommée arbre d'historique d'états. On peut ainsi effectuer des requêtes sur cette structure afin de récupérer l'état global ou l'état d'un certain attribut à un moment donné. D'autres approches, telles que stocker ces informations à même la trace, comme le font Chan et al. (2008), sont aussi possibles. Cependant, comme n'importe quel traitement en ligne, elles encourent des latences lors du traçage.

Un fois l'arbre d'historique d'états construit, on peut effectuer un grand nombre d'analyses, par exemple présenter visuellement l'état des processus comme le fait le logiciel Trace Compass, ou bien retirer des métriques en faisant des requêtes sur l'état du système entre deux points dans le temps. La construction en parallèle de l'arbre d'historique ne sera pas abordée dans la présente recherche, mais le concept d'état courant sera à la base de nos analyses.

2.2.4 Métriques système

Une trace noyau contient un grand nombre d'informations sur la totalité du système, pas seulement sur le noyau lui-même. Il est possible, à travers une analyse hors-ligne de traces, de récupérer d'importantes métriques par rapport à l'utilisation du CPU, du réseau ou du disque, entre autres (Giraldeau et al., 2011).

Par exemple, les événements d'ordonnancement peuvent donner un calcul précis du temps CPU utilisé par chaque processus pour une période de temps donnée. Dans le même ordre d'idée, les appels systèmes `open`, `read`, `write` et `close` permettent de récupérer des statistiques quant à l'utilisation des fichiers pour chaque processus. Récupérer ces événements avec un traceur performant ne cause qu'une faible surcharge sur le système, ce qui rend cette approche intéressante pour diagnostiquer des problèmes dans des systèmes où la performance est critique (c.-à-d. systèmes temps réel, en production, etc.).

Ces métriques sont récupérées à travers une lecture de la trace, de manière en ligne ou hors-

ligne. De plus, l'analyse peut même être effectuée sur un système différent de celui tracé. Elles peuvent notamment être extraites efficacement grâce à l'approche par système d'états décrit ci-haut : par exemple, seulement deux requêtes sur l'arbre d'historique d'états permettent d'obtenir la quantité de données lues d'un fichier entre deux estampilles temporelles.

2.2.5 Formats de trace

Le format dans lequel les données de trace sont enregistrées est très important lorsque l'on discute de l'analyse. En effet, celui-ci peut grandement affecter les performances de l'analyse, et doit être conçu en conséquence.

Fichier journal

Un des formats les plus simples pour emmagasiner les données de trace, le fichier journal (*log file*) contient en texte lisible les événements de la trace ainsi que leurs données (estampille de temps, source, etc.) Ce fichier peut ensuite être analysé ligne par ligne soit par un utilisateur, soit par un programme d'analyse. Bien que cette représentation soit pratique, il est difficile pour un utilisateur de détecter des problèmes complexes en inspectant manuellement un fichier texte. De plus, ce format se prête difficilement à l'analyse automatique, du fait du temps de traitement afin d'analyser les lignes, et génère souvent une quantité importante de données à cause de son format non compact.

CTF

Le format CTF est un format binaire de trace axé sur la performance (Desnoyers). Une trace CTF comprend non seulement les données binaires de la trace, mais aussi des métadonnées décrivant le format binaire de la trace. On obtient ainsi une très grande flexibilité lorsque l'on écrit la trace, ce qui permet d'obtenir des gains en performance, par exemple, en écrivant les événements directement dans un fichier, sans les traduire dans un autre format a priori. Le traceur LTTng exploite cette particularité afin d'obtenir de très faibles latences lors du traçage.

Une trace CTF est constituée d'un ensemble de flux de données (*streams*). Ces flux peuvent représenter, comme c'est le cas pour les traces LTTng, les événements générés par chaque processeur, par exemple. Ces flux sont enregistrés sous la forme de fichiers distincts, permettant ainsi la lecture et l'analyse indépendante de chaque flux. Cette caractéristique est intéressante pour la parallélisation, puisque l'on peut lire et traiter chaque flux de manière quasi distincte en parallèle.

Les flux de données sont composés de paquets de taille variable, contenant un en-tête de paquet ainsi que son contenu. Dans le cas de LTTng, chaque paquet correspond à l'enregistrement des événements d'un sous-tampon du tampon circulaire. L'en-tête contient entre autres le temps de début, le temps de fin et la taille du paquet. Au moment de l'enregistrement de la trace, un index de paquets est généré pour chaque flux. Cet index contient l'en-tête de chaque paquet du flux et permet de rapidement trouver certains points dans la trace en utilisant le temps de début et la taille. Ainsi, un analyseur de trace peut facilement, à l'aide d'une recherche dichotomique, trouver dans quel morceau de la trace se trouvent les événements correspondant à une certaine estampille de temps. Ceci s'avèrera être un aspect important lors de la parallélisation de l'analyse de traces, surtout utile lors de l'équilibrage de la charge, comme nous le verrons plus loin.

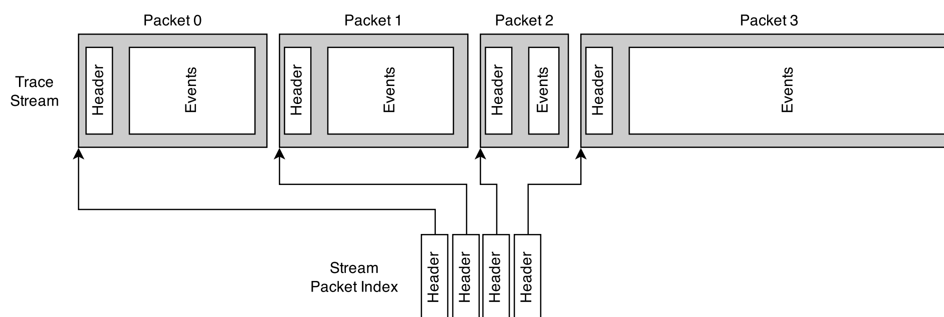

Figure 2.2 Paquets d'un flux d'événements CTF avec index de paquets

Au moment de l'analyse de la trace, les métadonnées sont d'abord lues afin de connaître le format exact de la trace et de ses événements. Ces métadonnées prennent la forme d'une description du format dans un langage adapté appelé Trace Stream Description Language (TSDL), qui décrit entre autres la taille des données, leur organisation (petit ou gros-boutien) ainsi que les structures utilisées. Une fois ces métadonnées analysées, il est possible de lire les données binaires de la trace et les traduire en événements. L'outil Babeltrace permet de lire les traces CTF et de les convertir en texte, en plus d'offrir une biblioth C pour la lecture de traces CTF.

2.3 Les modèles et outils de parallélisation

Les architectures multicœurs sont maintenant omniprésentes, et la quantité d'unités de calcul sur les processeurs symétriques augmente à grande vitesse. De plus, les systèmes distribués offrent une puissance de calcul parallèle inégalée, mais difficile à exploiter sans les outils appropriés. Cette section a pour but de cerner les différents modèles et outils facilitant la

programmation parallèle.

2.3.1 Pthreads

Pthreads (ou POSIX Threads) est le standard POSIX concernant la manipulation de fils d'exécution (Mueller, 1993). Ce standard est implémenté sous la forme d'une Application Programming Interface (API) par la plupart des systèmes d'exploitation tels que Linux, Solaris et FreeBSD. À travers cette API, un programme peut créer des fils, leur assigner une tâche et attendre la fin de leur exécution. Pthreads offre aussi d'autres primitives utiles à la programmation parallèle, telles que des mutexes, des sémaphores et des méthodes de synchronisation.

L'avantage de Pthreads est son omniprésence à travers les différentes plateformes et sa simplicité d'utilisation. Cependant, l'API offre des fonctions à un niveau relativement bas : aucun modèle haut-niveau n'est exposé, que ce soit par rapport à la répartition de la charge sur les fils ou la communication entre les fils, par exemple. Il n'est donc pas toujours aisé de développer une application avec un comportement parallèle complexe en n'utilisant que Pthreads, car l'utilisateur doit implémenter lui-même tous ces aspects cruciaux de la programmation parallèle. Ainsi, l'utilisation de Pthreads est plutôt appropriée pour la conception de bibliothèques parallèles de haut niveau, comme celles que nous verrons ici.

2.3.2 OpenMP

L'API Open Multi-Processing (OpenMP) offre un modèle de programmation parallèle plus haut-niveau que Pthreads (Dagum and Menon, 1998). Elle permet la parallélisation d'un programme C, C++ ou Fortran à travers un ensemble d'instructions données directement au compilateur par des directives "#pragma" dans le code source. Par exemple, on peut facilement, avec l'aide de cette API, paralléliser l'exécution d'une boucle, de telle sorte que chaque fil calcule la partie du résultat liée à sa partie du domaine d'entrées.

Listing 2.1 Exemple de programme OpenMP

```
static const int N = 10000;
/* Programme calculant les 10000 premiers carres */
int main() {
    int arr[N];
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
```

```
        arr[i] = i*i;
    }
}
```

OpenMP permet aussi de contrôler le partage de données entre les fils, de facilement unifier les résultats des fils à l'aide d'opérations de réduction, et d'automatiquement répartir la charge sur les fils en utilisant un ordonnanceur dynamique. L'usage d'OpenMP est très utile pour des calculs itératifs qui bénéficieront grandement de la parallélisation, comme un calcul utilisant la méthode des éléments finis. En effet, OpenMP fonctionne à une granularité très fine, permettant une parallélisation au niveau d'un ensemble d'instructions relativement petit. Cet outil n'est donc pas nécessairement adapté à des programmes demandant une parallélisation de tâches plus vastes.

2.3.3 TBB

Threading Building Blocks (TBB), développé par Intel, est une bibliothèque C++ qui offre une interface permettant la création de programmes parallèles en faisant abstraction des détails concernant la parallélisation (Reinders, 2007). La différence entre TBB et OpenMP est que TBB expose un modèle où l'utilisateur décrit son programme sous la forme d'algorithmes, plutôt que de spécifier quelles sections du code doivent être parallélisées. De plus, TBB utilise le concept de tâche comme unité de base de la parallélisation. Ces tâches sont organisées dans un graphe de dépendances, de telle sorte que les tâches s'exécutent en prenant compte des dépendances entre celles-ci.

Un autre aspect important de TBB est l'utilisation du vol de tâche (*task stealing* en anglais) afin de régler le problème de l'équilibrage de la charge entre les fils. Les données à traiter sont d'abord séparées entre les fils de manière égale, mais lorsque l'un de ces fils a fini de traiter ses données, il peut "voler" une partie des données d'un fil ayant une charge plus élevée. On obtient ainsi un rebalancement automatique de la charge sans que le programmeur ait à le rendre explicite.

TBB supporte aussi un ensemble d'algorithmes parallèles plus avancés, tels que le pipeline parallèle, qui est très utile lorsque l'on cherche à paralléliser un programme composé d'étapes pouvant s'exécuter en parallèle, mais nécessitant les résultats des étapes précédentes, comme une chaîne de montage dans une usine. Ces algorithmes avancés font de TBB un outil extrêmement intéressant lorsque l'on cherche à modéliser des interactions parallèles plus complexes et hétérogènes.

2.3.4 OpenCL

Le standard OpenCL définit une API permettant l'exécution de programmes sur des plateformes hétérogènes consistant, entre autres, de CPUs, GPUs et autres processeurs (Stone et al., 2010). Nous nous concentrerons ici principalement sur l'exécution sur GPU, qui permet d'effectuer des calculs sur un grand nombre d'unités de calcul parallèles.

OpenCL fonctionne à travers l'exécution de programmes écrits dans un langage spécifique (OpenCL C) sur le GPU. Ce programme, appelé *kernel*, permet d'effectuer les mêmes opérations sur un grand nombre de données en parallèle. Cependant, certaines contraintes existent : par exemple, sur la plupart des architectures, les embranchements sont tous exécutés, peu importe les données traitées. Il faut donc éviter le plus possible le traitement hautement dépendant du type de données, et tenter d'obtenir un traitement le plus homogène possible. De plus, l'unité de calcul principal (le CPU) est responsable des transferts de données vers et à partir du GPU. Il est donc important de prendre en compte la surcharge liée à ces transferts, en prenant soin de ne pas se retrouver avec un goulot d'étranglement au niveau des transferts de données.

Listing 2.2 Exemple de programme OpenCL

```
/* Calcule une addition vectorielle (result = a + b) */
__kernel void my_kernel(__global const float *a,
                        __global const float *b,
                        __global float *result, const int nb) {
    /* Recupere l'index courrant */
    const int id = get_global_id(0);
    /* Verifier les limites du tableau d'entree */
    if (id < nb) result[id] = a[id] + b[id];
}
```

L'analyse de traces se trouve à être une tâche nécessitant un traitement hétérogène, hautement dépendant du type d'événement à traiter. OpenCL, n'étant pas adapté à ce type de traitement, n'offre donc pas une solution efficace pour la parallélisation de l'analyse de traces, du moins lorsque l'on parle de l'analyse en général. Il est cependant possible que certaines parties de l'analyse soient transférables sur GPU, comme le font certaines solutions que nous verrons dans les prochaines sections. Nous n'aborderons pas directement cette approche dans la présente recherche, préférant nous concentrer sur la parallélisation globale de l'analyse.

2.3.5 MPI

Le système Message Passing Interface (MPI) est un standard définissant un protocole de communication entre des unités de calcul parallèle (Gropp et al., 1999). Il offre une interface indépendante du langage de programmation permettant la création de programmes hautement performants, tout en assurant la mise à l'échelle et la portabilité. Les implémentations MPI offrent entre autres une API dans certains langages de programmation (typiquement C ou FORTRAN) ainsi que l'infrastructure de communication.

L'API MPI permet au programme d'envoyer des messages et de synchroniser différents processus se trouvant typiquement sur des unités de calcul faisant partie d'une grappe de calcul. Ces fonctions permettent d'envoyer et de recevoir des messages points à point (`MPI_SEND`, `MPI_RECV`) ou d'effectuer des communications globales (`MPI_SCATTER`, `MPI_GATHER`). De plus, le programmeur peut aussi assigner une certaine topologie, telle qu'une topologie cartésienne ou en graphe, aux différents nœuds (`MPI_CART_CREATE`, `MPI_GRAPH_CREATE`). Ceci permet de faire correspondre facilement le domaine d'un problème à celui de la grappe de calcul. Par exemple, si l'on veut effectuer un calcul de dispersion de chaleur sur une surface, MPI nous permettrait de séparer celle-ci en régions bidimensionnelles, puis d'appliquer une topologie cartésienne à notre grappe de calcul afin de faire correspondre chaque région à un nœud. Ainsi, MPI est capable d'optimiser les communications entre les noeuds en prenant compte de la configuration matérielle de notre grappe.

MPI offre une API à un niveau relativement bas, s'occupant principalement des communications entre les nœuds. Ceci est à la fois un avantage, puisqu'elle offre ainsi une grande flexibilité en terme de type de programme supporté, ainsi qu'un inconvénient, puisque le programmeur est obligé d'écrire une grande quantité de code bas niveau afin de programmer même le plus simple des programmes.

Comme il est possible de tracer et corréler les traces de systèmes distribués, MPI pourrait offrir une solution intéressante au traitement parallèle et réparti des traces d'un système multinœuds. Par exemple, on pourrait avoir des machines dédiées au monitorage auxquelles seraient transférées les données de traces des machines surveillées à travers MPI. Ces machines pourraient ensuite effectuer l'analyse de leurs traces, tout en communiquant entre elles avec MPI afin d'échanger les données et recueillir les résultats finaux des analyses.

2.3.6 MapReduce

MapReduce, développé par Google, est un système permettant le calcul distribué sur un grand nombre d'unités de calcul en mémoire non partagée (Dean and Ghemawat, 2008). Le principe

de MapReduce, comme l'indique son nom, repose sur la spécification de deux fonctions par le programmeur, une fonction *map* et une fonction *reduce*. Empruntées du vocabulaire de la programmation fonctionnelle, c'est deux fonctions représentent les deux étapes principales d'une tâche MapReduce. La fonction *map* permet d'appliquer un traitement à chaque élément des données à traiter, et sauvegarde le résultat. La fonction *reduce* opère sur l'ensemble des résultats obtenus à la dernière étape afin d'obtenir un résultat final.

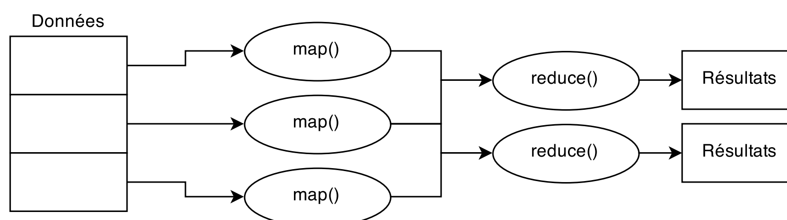

Figure 2.3 Exécution simplifiée d'une tâche MapReduce

Les principales qualités d'un système MapReduce sont non seulement sa capacité à paralléliser automatiquement une tâche, mais aussi sa capacité à gérer les défaillances et à optimiser les communications entre les nœuds. C'est pourquoi ce genre de système est surtout utile afin d'effectuer des calculs sur une très grande quantité de données réparties sur une grande quantité de machines.

Le système MapReduce a aussi été adapté à une exécution sur des systèmes en mémoire partagée. Ranger et al. (2007) proposent une implémentation, nommée Phoenix, des principes de MapReduce pour systèmes en mémoire partagée. L'implémentation consiste en une API ainsi qu'un *runtime* ayant comme but de permettre au programmeur de définir ses tâches, gérer les processus *map* et *reduce* et récupérer les calculs en cas de fautes, entre autres. Cette implémentation fût par la suite améliorée par Yoo et al. (2009) afin d'améliorer les performances sur des systèmes comportant un grand nombre d'unités de calcul (c.-à-d. système à quatre puces de 32 coeurs). Un autre projet, nommé Mars et détaillé par He et al. (2008), adapte les mêmes concepts aux GPU, qui contiennent généralement des centaines de processeurs.

2.4 L'analyse de traces en parallèle

2.4.1 Uniparallélisme

Un défi important lié à l'analyse de traces et à plusieurs types d'analyses que l'on voudrait paralléliser est la gestion des dépendances de données. Par exemple, un fil d'exécution voulant traiter un extrait de la trace commençant au milieu de celle-ci n'est pas en mesure de connaître

l'état courant, puisque celui-ci est dépendant des événements ayant eux lieu précédemment.

Pour régler ce problème, certains ont choisi d'utiliser une stratégie de parallélisation nommée uniparallélisme. Cette technique permet d'outrepasser plusieurs problèmes liés à l'analyse en parallèle de données contenant de fortes dépendances sérielles. L'uniparallélisme a été d'abord proposé par Veeraraghavan et al. (2012) afin d'offrir une méthode parallèle efficace au problème de l'enregistrement et de la réexécution de programmes. Dans leur article, Wester et al. (2013) utilisent cette méthode afin de paralléliser la détection de conditions critiques lors de l'exécution d'un programme parallèle.

Les algorithmes de détection de conditions critiques ont de fortes dépendances entre les instructions : l'analyse d'instructions dépend de l'analyse des instructions précédentes. Afin de permettre une analyse en parallèle, les auteurs proposent une solution basée sur la segmentation du programme en *époques*, où chaque époque consiste en l'exécution de tous les fils du programme entre deux instants dans le temps. Ces époques peuvent être exécutées de deux manières. La première exécute les époques séquentiellement et les fils en parallèle. C'est en quelque sorte une exécution normale du programme. Sur la figure 2.4, cette exécution correspond à l'exécution des fils A, B et C par les CPU 0 à 2. La deuxième exécution exécute les époques en parallèle et les fils de manière séquentielle. Sur la figure 2.4, cette exécution correspond à l'exécution des époques 0 à 4 sur les CPU 3 à 7.

L'exécution parallèle des époques (où les fils sont exécutés séquentiellement) permet d'effectuer une analyse efficace, puisque le fait que tous les fils du programme soient exécutés sur le même processeur évite l'utilisation de verrous dans l'analyse. Pour ce qui est des dépendances de données (car l'analyse doit connaître l'état au début d'une époque, qui est dépendant de l'état des époques précédentes), l'article suggère d'exécuter les époques non seulement parallèlement, mais aussi séquentiellement. Ainsi, l'exécution séquentielle des époques (où les fils sont exécutés parallèlement) pourrait offrir une prédiction de l'état de départ des époques, qui peut ensuite être utilisée pour effectuer l'analyse parallèle.

Puisqu'une époque n'a pas accès aux données de l'époque précédente, il est impossible que certaines parties de l'analyse puissent quand même être faites (par exemple, si une condition critique commence dans l'époque précédente et se termine dans l'époque courante). C'est pourquoi une dernière passe est effectuée à la fin de manière séquentielle entre les époques afin de fusionner les morceaux d'analyse manquants.

L'une des conditions pour que cette méthode fonctionne est que l'analyse effectuée lors de l'exécution séquentielle doit être très rapide : en effet, puisque l'exécution séquentielle permet de prédire l'état initial pour l'exécution parallèle, il faut que celle-ci s'exécute plus rapidement. La grande partie du travail d'analyse doit ensuite s'effectuer lors de l'exécution parallèle.

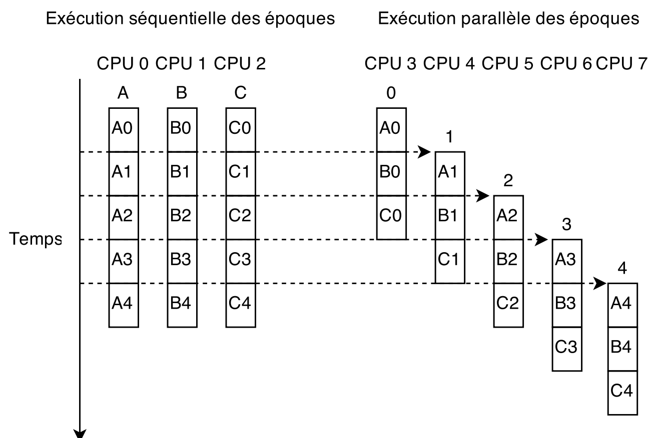

Figure 2.4 Illustration du principe d'uniparallélisme

L'autre condition est de minimiser la quantité d'analyse qui doit être faite séquentiellement à la fin de l'exécution parallèle, puisque nous voulons détecter la condition critique le plus rapidement possible.

La parallélisation par époques a aussi été utilisée par Nightingale et al. (2008) afin d'effectuer des vérifications de sécurité sur des programmes en parallèle avec l'exécution du programme ; par Süßkraut et al. (2010), sous le nom de "*Predictor/Executor*", afin de créer un compilateur supportant la parallélisation de programmes séquentiels ; ainsi que par Zilles and Sohi (2002), sous le nom de "*Master/Slave Speculative Parallelization*", comme technique de parallélisation de programmes séquentiels.

La méthode de l'uniparallélisme et de l'exécution parallèle des époques offre des pistes de solution intéressantes, mais a été développée avec des contraintes qui ne sont pas présentes lors de l'analyse de traces. Premièrement, l'analyse est de type dynamique, c'est-à-dire qu'elle cherche à détecter des conditions critiques lors de l'exécution du programme, et non a priori ou a posteriori. Dans le cas de l'analyse de traces qui nous intéresse ici, nous effectuerons des analyses a posteriori, ce qui nous permet de déférer l'obtention de résultats jusqu'à la fin de l'exécution de l'analyse.

Deuxièmement, les analyses adaptées dans l'article (*happens-before* et *lockset*) sont très sensibles aux dépendances, alors que les analyses de traces, comme nous le verrons, ne restent pas longtemps dans un état indéterminé. Il nous est donc possible d'outrepasser la prédiction de l'état initial des époques, tant que cet état peut être retrouvé à la fin de l'analyse.

Finalement, la majeure partie du travail de l'analyse de traces est souvent le décodage des événements, ce qui restreint beaucoup l'accélération possible avec l'uniparallélisme. En effet,

si l'on effectuait deux "exécutions", l'une avec des époques séquentielles et l'autre avec des époques parallèles, il faudrait lire au moins deux fois chaque événement. Or, ceci restreint de beaucoup notre capacité d'accélération. La présente recherche se basera sur les mêmes genres de problèmes que ceux de l'uniparallélisme, mais tout en relâchant les contraintes mentionnées.

2.4.2 Réexécution

Plusieurs outils permettent l'analyse de la performance et du comportement d'applications MPI, par exemple l'outil visuel Vampir. L'outil KOJAK EXPERT, développé par le Jülich Supercomputing Centre, facilite cette analyse manuelle en la remplaçant par une analyse automatisée des données de trace afin de trouver certaines caractéristiques propres à des problèmes de performance. Par exemple, on pourrait vouloir trouver toutes les instances d'attentes causées par des communications inefficaces. C'est exactement ce que démontrent Wolf et al. dans leur travail (Wolf and Mohr, 2003). Leur système fonctionne en enregistrant puis en fusionnant les traces d'événements de chaque processus, puis en parcourant cette trace globale à l'aide d'un algorithme utilisant une fenêtre glissante (sliding window) dans laquelle les séquences d'événements occasionnant une attente sont détectées. Cependant, à mesure que le nombre d'unités de calcul et de processus augmente, il devient vite impossible d'effectuer cette analyse de manière séquentielle, faute de puissance de traitement et d'espace mémoire. C'est pourquoi une nouvelle approche exploitant la puissance de calcul parallèle a dû être mise en place.

Dans leur travail, Geimer et al. ont adapté le programme séquentiel d'analyse des états d'attentes afin de le rendre exécutable en parallèle sur un grand nombre de nœuds (Geimer et al., 2009). Plutôt que de fusionner plusieurs traces locales en une grande trace globale, opération coûteuse en temps et en mémoire, les traces locales sont analysées séparément par les nœuds les ayant engendrées. Encore une fois, on se retrouve face à un problème de dépendance de données : la séquence d'événements recherchée se retrouve dans plusieurs fichiers de trace. Afin de régler ce problème sans de coûteuses communications entre les nœuds analysants, leur nouvel outil, Scalasca, utilise une approche qui se base sur la réexécution des événements de communication qui ont été enregistrés dans les traces sur le même système multinœuds sur lequel l'exécution originelle s'est déroulée. Cela permet donc d'obtenir automatiquement la correspondance entre les événements des traces à mesure que les communications sont réexécutées.

Ce système a besoin de plusieurs prérequis afin de fonctionner. Premièrement, il doit y avoir une correspondance de 1-à-1 entre les processus analysés et les processus d'analyse

(c.-à-d. autant de processus faisant l'analyse que de processus analysés.) Deuxièmement, l'infrastructure d'analyse doit être la même que celle qui est analysée. De plus, la solution proposée ne peut traiter en ce moment que des traces pouvant être chargées complètement en mémoire.

Cependant, cette approche reste intéressante, surtout dans le cas de l'analyse de traces distribuées. En effet, l'analyse peut être effectuée sur chaque nœud, en réexécutant les communications entre les nœuds afin de synchroniser ceux-ci et partager les données.

2.4.3 Exécution parallèle de machines à états

Comme nous l'avons vu ci-haut, l'analyse de traces peut être modélisée sous la forme d'une machine à états. Les machines à états sont un outil essentiel de l'informatique, permettant la modélisation d'une foule de problèmes, tels que la validation d'expressions régulières ou bien le décodage de Huffman. Plusieurs méthodes de parallélisation de l'exécution de machines à états ont été proposées dans la littérature. La plupart se basent sur la méthode du calcul parallèle de la somme préfixe afin de paralléliser l'exécution de la machine à état (Ladner and Fischer, 1980).

La somme préfixe représente la somme courante à chaque point dans une liste de valeurs. Par exemple, soit la liste composée des chiffres 1, 2, 3 et 4, la somme préfixe serait 1, 3, 6 et 10. L'algorithme de somme préfixe parallèle vise à effectuer ce calcul sur plusieurs unités de calcul parallèles en effectuant un nombre plus élevé d'opérations qui, lorsqu'effectuées en parallèle, permettent un gain en performance avec $\mathcal{O}(n/\log n)$ processeurs. L'algorithme est bien illustré dans Hillis and Steele Jr (1986). Cet algorithme peut être généralisé pour le calcul de n'importe quelle séquence d'opérations binaires associatives.

Dans le cas de machines à états, il est possible d'appliquer cette stratégie de parallélisation de plusieurs manières. La première, proposée par Ladner and Fischer (1980), fonctionne grâce au calcul de la somme préfixe de matrices M_s avec $M_s[i,j] = 1$ si et seulement s'il existe une transition de l'état i à j lorsque le symbole s est lu. Ainsi, on peut calculer $M = M_{sm} \times ... \times M_{s2} \times ... \times M_{s1}$, ce qui nous donne $M[i,j] = 1$ si et seulement si l'on obtient l'état j en commençant à l'état initial i lorsqu'on lit la séquence $s1, s2, ..., sm$. Une autre manière, nécessitant moins de calcul, est proposée par Hillis and Steele Jr (1986). Elle consiste à effectuer un calcul de somme préfixe sur les fonctions de transitions en utilisant comme opérateur la composition de fonction (qui est une opération associative). On obtient donc $T = T_{sm} \circ ... \circ T_{s2} \circ T_{s1}$, où T_s est la fonction de transition pour le symbole s.

Mytkowicz et al. (2014) se basent sur les solutions précédentes afin de proposer une implé-

mentation efficace de la parallélisation de machines à états. Cette implémentation utilise une approche qui consiste à garder un vecteur d'état courant plutôt qu'un seul état lors de l'exécution de l'automate. Ce vecteur représente l'état courant pour chaque état initial possible. Pour un automate de n états, on aurait donc un vecteur de n éléments. Une fois que chaque processeur a traité ses symboles, il suffit que le processeur ayant traité la première partie communique l'état à la fin de son traitement au second processeur, et ainsi de suite jusqu'au dernier processeur (voir Figure 2.5).

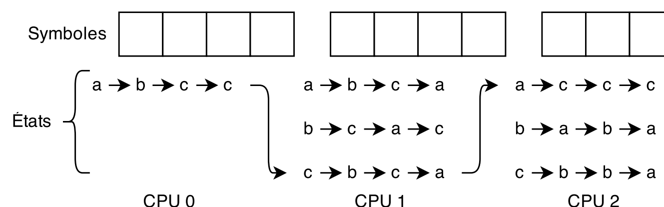

Figure 2.5 Calcul parallèle de machine à états

La contribution principale provient de la possibilité d'exploiter certaines caractéristiques de la majorité des machines à états afin de réduire le nombre de calculs à effectuer lors de la lecture des symboles, ainsi que de l'utilisation des différents niveaux de parallélisme disponible sur les architectures modernes (processeurs multicœurs, vectoriels, etc.).

Bien que ces approches soient intéressantes, elles nécessitent deux passes de lecture des symboles. Or, dans le cas de traces, ces symboles sont des événements sauvegardés dans des fichiers de traces. Ces événements doivent être lus du disque et désérialisés, ce qui prend un temps considérable. On perdrait donc une grande partie de la performance en utilisant ces méthodes. Un prototype utilisant cette méthode a d'ailleurs été implémenté dans le cadre de la présente recherche, mais la perte en performance due à la double lecture de la trace en faisait une solution moins intéressante que celle qui sera présentée plus loin.

2.4.4 Détection d'intrusion parallèle

Les réseaux d'entreprises sont une cible fréquente d'attaques informatiques. Afin de se défendre contre de telles attaques, des systèmes de détection d'intrusions réseau (NIDS) peuvent être mis en place au sein du réseau afin d'en analyser le trafic et de détecter des intrusions. Cependant, à mesure que le trafic réseau augmente, il devient difficile pour le NIDS de traiter les paquets avec un débit de traitement raisonnable.

Plusieurs ont exploré la parallélisation comme solution à ce problème. Une partie importante de la tâche d'un NIDS étant la recherche de chaînes de caractères, la parallélisation de cette

recherche est en mesure d'offrir une hausse de performance considérable. Une grande quantité de recherche a été faite sur ce problème, que ce soit sur un système multicœur à mémoire partagé, un système distribué ou bien sur un GPU (Tumeo et al., 2012). Il en va de même avec la validation d'expressions régulières. Cependant, la recherche de chaînes de caractères n'est pas la seule partie parallélisable d'un NIDS.

Une avenue intéressante de parallélisation est celle de la parallélisation au niveau du flux (*flow level parallelization*). Dans ce contexte, un flux représente une communication entre deux points et est composé de messages transitant dans des paquets réseau. Schuff et al. offrent deux solutions basées sur la parallélisation au niveau du flux, l'une conservatrice et l'autre optimiste (Schuff et al., 2008).

Dans les deux cas, il s'agit de séparer le trafic réseau de telle sorte que chaque unité de calcul traite les données liées à un flux. Pour ce faire, un "producteur" est d'abord chargé d'effectuer une analyse sommaire du paquet reçu afin de déterminer le flux auquel il appartient, puis l'envoie en traitement au "consommateur" visé. Chaque consommateur s'exécutant sur son propre fil d'exécution, on obtient ainsi un traitement parallèle des flux. Puisque les algorithmes de détection d'intrusion n'opèrent que sur un seul flux, on obtient la garantie que toutes les données nécessaires seront disponibles au processeur traitant le flux et qu'aucune synchronisation ne soit nécessaire avec d'autres processeurs. Cependant, ceci nécessite que chaque flux ne soit traité que par un seul processeur.

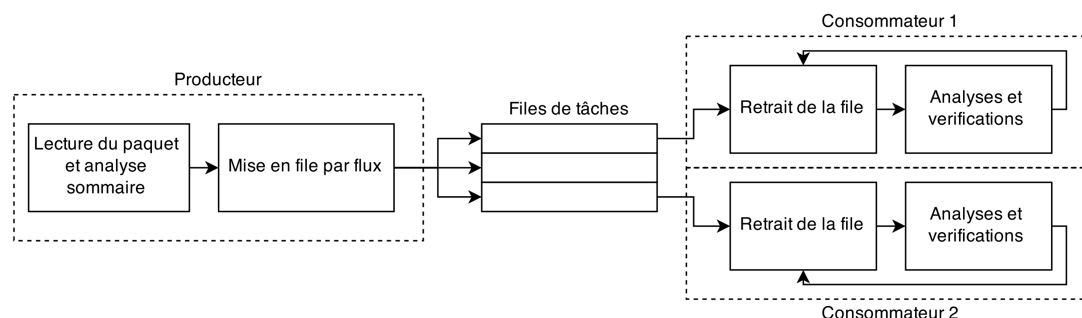

Figure 2.6 Parallélisation au niveau du flux

L'analyse conservatrice respecte cette contrainte, mais permet quand même de réassigner un flux à un autre processeur libre, si celui-ci continue d'être le seul à traiter le flux. Cependant, ceci peut causer un problème d'équilibre de la charge : s'il n'y a souvent pas autant de flux que de processeurs, notamment. L'analyse optimiste relâche cette contrainte, en se basant sur le fait que la plupart des paquets n'ont pas besoin d'accéder à l'état partagé du flux. En effet, d'après une analyse statistique de différentes analyses du logiciel Snort, il apparaît que

seulement 3% des paquets ont besoin d'avoir accès à l'état partagé. Ainsi, les paquets d'un même flux peuvent être traités par plusieurs processeurs simultanément, à condition que ce traitement bloque si le paquet analysé a besoin de l'état et qu'il n'est pas le paquet le plus vieux chronologiquement.

Il est intéressant de noter que les travaux de Vasiliadis et al. utilisent le concept de parallélisation au niveau des flux et ajoutent une stratégie de parallélisation à l'intérieur d'un même flux à travers l'utilisation de GPUs (Vasiliadis et al., 2011). Cette approche permet d'obtenir un niveau de parallélisation supérieur, au coût d'une latence ajoutée due à la mise en lots (*batching*) des requêtes vers le GPU.

On peut tirer un parallèle ici avec l'analyse de traces : plutôt que d'avoir un flux composé de paquets provenant d'une communication, nous avons un flux composé d'événements provenant d'un même processeur ou d'un même processus. Il serait donc possible d'appliquer le même genre de stratégie de parallélisation au niveau du flux d'événements. Cependant, cette approche est problématique dans le cas d'une trace dont les flux sont par CPU. Dans le cas de l'approche conservatrice, on se retrouverait à être limité dans le nombre de processeurs utiles à l'analyse parallèle. Par exemple, si l'on analyse une trace comportant huit flux (provenant d'une machine à huit processeurs) sur une machine ayant accès à soixante-quatre processeurs, cinquante-six de ceux-ci seront inutilisés.

Pour ce qui est de l'approche optimiste, les hypothèses permettant son emploi dans un NIDS ne tiennent pas nécessairement lors de l'analyse de traces. En effet, la plupart des analyses fonctionnent non pas au niveau des CPUs, mais plutôt au niveau des processus. Par exemple, il est plus utile de connaître la quantité de données écrites sur disque par processus plutôt que par CPU. Or, à cause de contraintes d'espace, il n'est pas toujours possible ou désirable d'enregistrer le PID pour chaque événement. Il faut donc garder dans un état global le processus s'exécutant présentement sur chaque CPU (facilement récupérable grâce aux événements d'ordonnancement) et consulter cet état à chaque fois qu'un événement est traité.

2.5 Conclusion de la revue de littérature

Le survol des outils de traçage sous Linux nous a permis d'évaluer les capacités de chaque traceur. Nous avons aussi vu que le traçage est un moyen efficace d'obtenir de l'information sur l'exécution d'un système avec un minimum de perturbation. En effet, il est possible, en particulier avec le traceur LTTng, d'obtenir des traces de systèmes hautement parallèles et distribués avec un surcoût minimum. Nous avons ensuite exploré les différents aspects liés à l'analyse de trace, tels que les modèles d'analyses, ce qui nous a permis d'identifier de

potentiels candidats à la parallélisation au sein de ceux-ci. Puis, nous avons exploré les outils de parallélisation disponible, évaluant au passage leur pertinence dans notre contexte de recherche. Finalement, nous avons fait un tour des solutions liées à l'analyse de traces en parallèle, dressant des parallèles et soulignant les différences avec notre contexte.

La prochaine section décrit la méthodologie utilisée afin de mettre en place et tester notre solution à l'analyse de traces en parallèle. Comme nous le verrons plus loin, les concepts décrits ci-haut formeront la base de nos méthodes proposées.

CHAPITRE 3 MÉTHODOLOGIE

Nous décrirons ici la méthodologie utilisée afin d'évaluer la parallélisation comme méthode efficace d'accélération de l'analyse de traces. Plusieurs facteurs doivent être pris en compte afin d'évaluer une méthode parallèle : outre l'accélération atteinte, il faut aussi prendre en compte l'efficacité de la solution, sa possibilité de mise à l'échelle, ainsi que l'impact des ressources pouvant limiter la performance (telles que les accès au périphérique de stockage).

L'environnement de travail sera décrit dans la section suivante. Nous décrirons aussi les analyses sélectionnées pour la parallélisation, ainsi que les traces qui seront analysées. Puis, nous présenterons le programme utilisé afin de simuler la lecture de traces et de mesurer l'impact des accès au disque et de l'amélioration du décodage des événements.

3.1 Environnement

3.1.1 Matériel

Afin de mettre à l'épreuve les capacités de mise à l'échelle de notre solution, nous avons utilisé un système à mémoire partagée comprenant un grand nombre de cœurs. Ce système est pourvu de quatre processeurs ayant chacun seize cœurs, pour un total de soixante-quatre cœurs.

Il est à noter que le système utilise l'architecture Bulldozer de AMD, ce qui signifie que les processeurs sont en fait constitués de huit *modules* constitués chacun de deux unités de calcul entier partageant une seule unité de calcul à virgule flottante ainsi que certaines étapes du pipeline [1]. On peut donc considérer, pour les calculs sur entiers, que le processeur contient seize cœurs, mais il faut prendre en compte que ces ressources partagées causent un coût en terme de performance lorsque les seize cœurs sont utilisés.

- Carte mère : Supermicro H8QGL
- Processeurs : 4 × AMD Opteron 6272 avec 16 cœurs à 2.1 GHz
- Mémoire vive : 16 × 8 Go Kingston DDR3 SDRAM 1600 MHz

3.1.2 Logiciel

Le système d'exploitation Linux nous permet d'utiliser le traceur haute performance LTTng afin d'obtenir nos traces. De plus, puisque le code source est libre, nous pouvons facilement

1. http ://en.wikipedia.org/wiki/Bulldozer_(microarchitecture)

analyser certaines parties du système d'exploitation en lisant le code source. Cet aspect s'est révélé être important dans notre recherche, comme nous le verrons plus loin.

Pour ce qui est de l'implémentation de la solution, plusieurs bibliothèques ont été utilisées, notamment les bibliothèques Qt Concurrent, OpenMP ainsi que TBB. Qt Concurrent offre une implémentation simple du principe de *map-reduce* présenté dans la revue de littérature, sans les aspects liés au calcul distribué. Les deux autres technologies sont décrites aux sections 2.3.2 et 2.3.3 respectivement.

- Système d'exploitation : Fedora 20 64-bit
- Version du noyau Linux : 3.17
- Version de Qt : 5.3
- Version de TBB : 4.1
- Version de GCC (pour support OpenMP) : 4.8.3

3.2 Analyses

Les analyses sélectionnées ci-dessous permettent l'évaluation de l'efficacité parallèle avec des analyses de différents niveaux de complexité. Ces analyses, sauf pour le comptage d'événements, sont adaptées des analyses en Python créées par Julien Desfossez [2]. Elles ont été implémentées en utilisant la bibliothèque Qt Concurrent, dans le langage C++. Le code source de ces analyses est disponible à l'adresse suivante : `https://github.com/TheZorg/lttng-parallel-analyses`.

3.2.1 Comptage d'événements

Cette analyse constitue l'analyse la plus simple que l'on puisse effectuer sur une trace, c'est-à-dire compter les événements. Les performances parallèles de cette analyse constitueront une borne inférieure sur l'accélération possible, puisque la surcharge sur le CPU est minimale : la majorité du travail proviendra du décodage des événements, et l'effet des accès au disque sera le plus haut.

3.2.2 Analyse de temps CPU

Cette analyse permet d'afficher la proportion de temps CPU utilisé par chacun des processus du système, ainsi que la proportion de temps d'activité pour chacun des CPU du système. L'analyse CPU est légèrement plus complexe, mais beaucoup plus utile que le comptage d'événements.

2. https ://github.com/lttng/lttng-analyses

3.2.3 Analyse d'entrées/sorties

L'analyse d'entrées/sorties permet d'obtenir la quantité de données lues et écrites par chaque processus grâce aux appels système de type *read* et *write*. Cette analyse est plus complexe que l'analyse CPU, nécessitant de garder une certaine quantité d'information pour chaque processus.

3.2.4 Traces analysées

Les traces analysées ont été enregistrées à l'aide du traceur LTTng sous Linux. La principale trace de test est une trace d'un banc d'essai de la base de données Redis lancé en parallèle, de manière à générer plusieurs événements d'entrées/sorties sur plusieurs cœurs. Les informations sur la trace sont contenues dans le tableau 3.1.

Tableau 3.1 Détails de la trace test

Système tracé	AMD FX-9370 avec 8 cœurs à 4.4 GHz
Version LTTng	2.6.0
Nombre d'événements	44 897 970
Événements activés	Tous

3.3 Programme de simulation

Afin d'évaluer l'impact de certains facteurs sur l'analyse de trace parallèle, il nous a été utile d'utiliser un programme simulant celle-ci. Ce programme, très simple de nature, a pour but de simuler la charge sur le CPU ainsi que sur le disque d'une analyse simple, ne visant qu'à décoder les événements. Ce programme sera plus amplement décrit à la section 4.6.2.

Le code source du programme de simulation est disponible à l'adresse suivante : `https://github.com/TheZorg/experiments`. Le programme `io-test` correspond à la version sans pipeline, et le programme `pipelined-io-test` correspond à la version avec pipeline utilisée pour les bancs d'essai.

3.4 Tests d'efficacité

La métrique principale qui sera présentée afin d'évaluer notre solution sera celle de l'efficacité parallèle. L'efficacité parallèle est définie comme étant le pourcentage d'accélération linéaire atteint avec un certain nombre de tâches parallèles. Par exemple, si un programme est 16

fois plus rapide lorsqu'exécuté avec 16 fils d'exécution, l'efficacité parallèle est de 100%. Si le programme n'avait été que 8 fois plus rapide, l'efficacité parallèle aurait été de 50%.

L'efficacité parallèle est calculée avec la formule $E = S/N$, où S correspond à l'accélération (*speedup*) obtenue avec N tâches parallèles. L'accélération, quant à elle, est donnée par la formule $S = t_1/t_N$, où t_1 est le temps d'exécution séquentiel et t_N est le temps d'exécution parallèle avec N unités de calcul.

La raison pour laquelle nous utiliserons l'efficacité plutôt que l'accélération est que, en général, l'accélération est une métrique parfois trop optimiste et difficile à mettre en contexte et à comparer. En effet, il est souvent possible d'obtenir des accélérations toujours plus grandes en augmentant la quantité d'unités de calcul parallèle. Or, ceci implique qu'une quantité potentiellement importante de cycles soit gaspillée. L'efficacité parallèle est aussi facile à évaluer, puisqu'il suffit de comparer à une efficacité parfaite de 100%.

Les différents aspects de la méthodologie décrite ici sont utilisés dans le prochain chapitre, qui contient l'article de journal "Efficient Methods for Trace Analysis Parallelization". C'est dans cet article que nous abordons le cœur des résultats de la présente recherche.

CHAPITRE 4 ARTICLE 1: EFFICIENT METHODS FOR TRACE ANALYSIS PARALLELIZATION

Authors
Fabien Reumont-Locke
École Polytechnique de Montréal
fabien.reumont-locke@polymtl.ca

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

Keywords: Tracing, Trace analysis, Parallel computing

Submitted to: IEEE Transactions on Parallel and Distributed Systems, July 2 2015

4.1 Abstract

Tracing provides a low-impact, high-resolution way to observe the execution of a system. As the amount of parallelism in traced systems increases, so does the data generated by the trace. Most trace analysis tools work in a single thread, which hinders their performance as the scale of data increases.

In this paper, we explore parallelization as an efficient approach to speedup system trace analysis. We propose a solution which uses key aspects of the CTF trace format to create balanced, parallelizable workloads. We also propose an algorithm to detect and resolve data dependencies during trace analysis, with minimal locking and synchronization. Using this approach, we implement three different trace analysis programs: event counting, CPU usage analysis and I/O usage analysis. The parallel implementations achieve speedups of up to 18 times with 32 cores using trace data stored on consumer-grade solid state storage devices. We also show the scalability and potential of our approach by measuring the effect of future improvements to trace decoding on parallel efficiency.

4.2 Introduction

As multi-core and distributed systems become ubiquitous, detecting and solving runtime problems becomes increasingly difficult. Tracing, by providing a detailed log of low-level events across the system, allows one to gather enough information to solve a variety of behavioral and performance problems, with a very low impact on system performance. The LTTng tracer (Desnoyers and Dagenais, 2006) allows for very low overhead when tracing and is optimized for scalability as the number of parallel processors increases (Desnoyers and Dagenais, 2012).

It is possible, with these tools, to efficiently trace very large-scale systems. The amount of trace data for such systems can be staggering, with traces of a few seconds of execution containing several gigabytes of data. Since manual inspection of these traces is not viable, a trace analysis program is often used to extract meaningful information from the traces. The analysis may be graphical or text-based, and can extract system metrics, run verifications on the system or present graphical information about the execution.

Trace analysis tools for LTTng include Trace Compass [1] (formerly TMF), which provides graphical analyses of traces, and the babeltrace library [2], which is used to implement command-line analyses. Both of these solutions use a single-threaded approach to trace analysis. This is problematic: as the architecture of the traced systems becomes more and more parallel, the gap between the amount of data produced and the single-threaded analysis speed will widen. This is why the present paper explores trace analysis parallelization as an efficient method to overcome this ever widening gap.

The goal of this research is threefold. First, we will propose a method to parallelize the seemingly serial trace analysis model. Second, we will show, using a program that simulates the CPU and I/O workloads of trace analysis, that the memory and I/O intensive nature of trace analysis does not make parallelization an unreasonable solution. We will also prove that parallel trace analysis is future-proof, even in the event of more efficient trace decoding. Finally, we will present and discuss benchmarks of three parallel trace analyses that were implemented using the aforementioned parallel methods, focusing on scaling and efficiency.

1. http://projects.eclipse.org/projects/tools.tracecompass
2. https://www.efficios.com/babeltrace

4.3 Related Work

4.3.1 Tracing tools

A variety of tracing tools is available for the Linux operating system. The ftrace tracer (Rostedt, 2009) allows for kernel-space only tracing using static and dynamic instrumentation provided by the Linux kernel. SystemTap (Eigler and Hat, 2006) relies on user-provided scripts that are compiled into kernel modules before being executed as the system is traced, a very flexible approach that comes at a cost in performance, especially when scaling to many-core systems (Desnoyers and Dagenais, 2012). The perf tool (de Melo, 2010), once used mainly to retrieve performance counter values through a sampling-based approach, can also provide tracing capabilities, though with lower performance than ftrace due in part to a less efficient ring buffer implementation. LTTng (Desnoyers and Dagenais, 2006) provides correlated traces of user and kernel-space execution (Fournier et al., 2009) with minimal overhead and in a highly scalable manner (Desnoyers and Dagenais, 2012).

Tracing as a diagnostic tool has been used in a variety of use cases, such as system verification and system metrics extraction. Trace analysis can be modeled using a finite-state machine approach (Matni and Dagenais, 2009), where each analysis is executed as an automaton reading events and executing analysis code on transitions to states. These automatons could be good candidates for parallelization (i.e. one automaton per thread) but, since many analyses depend on similar data (process scheduling data, for example), the finite-state machine approach has been generalized into a global state system, which does not lend itself as easily to parallelization. The state system works by storing the current state of the whole system, which can be accessed for any time stamp within the trace. Montplaisir et al. (2013) propose a method for building this state system by reading a trace sequentially and storing the state system's data into an efficient disk-based interval tree called a State History Tree.

4.3.2 Parallel tools

With the advent of highly parallel architectures and distributed systems, parallel programming tools, frameworks and models become an important part of the development of high performance programs. Lower level libraries, such as Pthreads (Mueller, 1993), allow for easy manipulation of threads and synchronization primitives. However, as the number of parallel computing units increases, it becomes hard to develop complex parallel behavior using these low-level functionalities. OpenMP (Dagum and Menon, 1998) provides a higher-level parallel programming model based on compiler directives. It allows for the fine-grained parallelization of loops and blocks of code, taking care of load balancing and basic synchronisation, but is

not well suited to the parallelization of complex, larger scale tasks. TBB (Reinders, 2007) provides an object-oriented C++ library allowing for task-based parallelization, whereas the user provides tasks to be parallelized, instead of having to manipulate parallelization primitives. It also implements work stealing as a mean to balance the workload at run time.

Parallelization is not limited to CPUs. The OpenCL standard (Stone et al., 2010) specifies an API and language (OpenCL C) allowing the execution of programs across heterogeneous, highly parallel devices such as GPUs, ASICs and others. The MPI standard (Gropp et al., 1999) defines a communication infrastructure for parallel computing units. It is based on the passing of messages from one node to another, in order to exchange data and synchronize processing. Finally, the MapReduce system (Dean and Ghemawat, 2008) has seen a lot of traction in recent years due to its simplification of running high performance calculations on distributed systems, taking into account fault tolerance and the optimization of node communications. It has since then also been adapted to shared memory systems (Ranger et al., 2007; Yoo et al., 2009) and GPUs (He et al., 2008), amongst others.

4.3.3 Parallel trace analysis

A major problem with the parallel analysis of kernel traces comes from the strong sequential dependencies within the trace data. This is due to most analyses being stateful, meaning that analyzing certain events requires information contained in previous events. A method called uniparallelism, proposed by Veeraraghavan et al. (2012), aims at solving this problem through "epoch-parallel execution" (Nightingale et al., 2008), also called "Predictor/Executor" (Süßkraut et al., 2010) and "Master/Slave Speculative Parallelization" (Zilles and Sohi, 2002). Uniparallelism works by splitting the program's execution into *epochs*, which are sequences of chronologically executed instructions, and by running two versions of the program in parallel, one epoch-sequential (i.e. a regular execution) and one epoch-parallel. The epoch-sequential execution then runs a light-weight analysis which will provide the speculated starting state for the epoch-parallel execution. The epoch-parallel executions may then be executed in a parallel pipeline along the epoch-sequential execution. While this solution is interesting to solve data dependencies, it ignores possible optimizations inherent to trace analysis, which removes the need for the epoch-sequential execution (i.e. the "Predictor").

The KOJAK EXPERT tool enables the automatic analysis of MPI application traces, for example to find wait states caused by inefficient communications (Wolf and Mohr, 2003). However, these types of analyses become impossible to run on systems with a high number of parallel computing nodes, due to a lack of processing power and memory space. Geimer et al. (2006) propose a scalable parallel approach to wait state diagnosis, which was improved by

Geimer et al. (2009). Their approach is based on replaying run-time communications between nodes at trace analysis time. This allows not only the synchronization of the trace analysis, but also the communication of data dependencies between the analysis nodes. This approach is interesting for the analysis of distributed traces, but is not efficient in a shared memory trace analysis context, where communications between "nodes" (CPUs) are frequent.

Though not directly related to trace analysis, parallel execution of state machines is an interesting avenue to consider, since trace analysis is often expressed in terms of finite automatons (Matni and Dagenais, 2009). Since state machines are an essential part of computing, their parallel execution was studied for many years. Most methods are based on the parallel prefix sum algorithm. A prefix sum calculation returns the running total (prefix sum) at every point in an array of values. Though seemingly inherently serial, due to calculations needing access to previous results, this problem has an efficient parallel implementation (often called *parallel scan*), detailed by Ladner and Fischer (1980), and can be generalized for the calculation of any sequence of associative binary operations. By encoding transitions as functions and using function composition as the operator for a parallel prefix sum, Hillis and Steele Jr (1986) offer a method for the parallelization of finite-state machine execution. Using these concepts, Mytkowicz et al. (2014) propose an implementation that executes a FSM using an enumerative approach, where the current state is kept for each possible starting state. It exploits modern computer architectures as well as properties of most FSMs to efficiently perform this enumerative execution. However, it requires two passes through the input data, which greatly hinders scalability in a trace analysis context: indeed, decoding trace data, as we will see, is a major part of the CPU work for trace analysis.

Network Intrusion Detection Systems, or NIDS, analyse network traffic in order to detect certain patterns suggesting unusual network activity. However, the NIDS throughput must be able to keep up as network traffic increases. Some have explored parallelization as a solution to scaling NIDS processing power. String matching being a large part of a NIDS's work, it is an obvious candidate for parallelization (Tumeo et al., 2012). However, it is not the only place where parallelization can come into play. Flow-level parallelization, for example, adds a level of parallelism by assigning the analysis of one flow per execution thread, where a flow is defined as a communication between two endpoints comprised of network packets. Schuff et al. (2008) propose two methods, one conservative and the other optimistic, in order to allow for load balancing between the flows. A similar flow-parallel method was used by Vasiliadis et al. (2011) and enhanced by adding intra-flow parallelism by offloading part of the work within a flow to the GPU. In the context of trace analysis, a similar method could be used, with the concept of "flow" being applied to the events generated by one processor of the traced system (i.e. the content of a specific per-CPU ring buffer). This flow-based approach will be revisited

in section 4.5.1.

4.4 Background

In order to identify potential parallelization solutions, we must understand how serial trace analysis functions. This section will focus on the current trace analysis design, including the trace data format, the parsing of trace files and the analysis model. Since we are using traces generated by LTTng, we will be using the Common Trace Format (CTF), in which LTTng traces are generated, and the babeltrace library, which allows reading and writing CTF traces.

4.4.1 CTF trace format

CTF is a flexible, compact binary trace format which caters to the needs of the embedded, telecom, high-performance and kernel communities (Desnoyers). Its particularity resides in its ability to specify the exact structure of the trace and its events through the TSDL. The trace definition, written in TSDL, is stored in a metadata file which is then parsed in order to create the binary parser used to read the trace. This allows a large amount of flexibility in terms of the data stored in the trace, letting the user define the size, alignment and endianness of basic types, as well as compound types such as structures and unions.

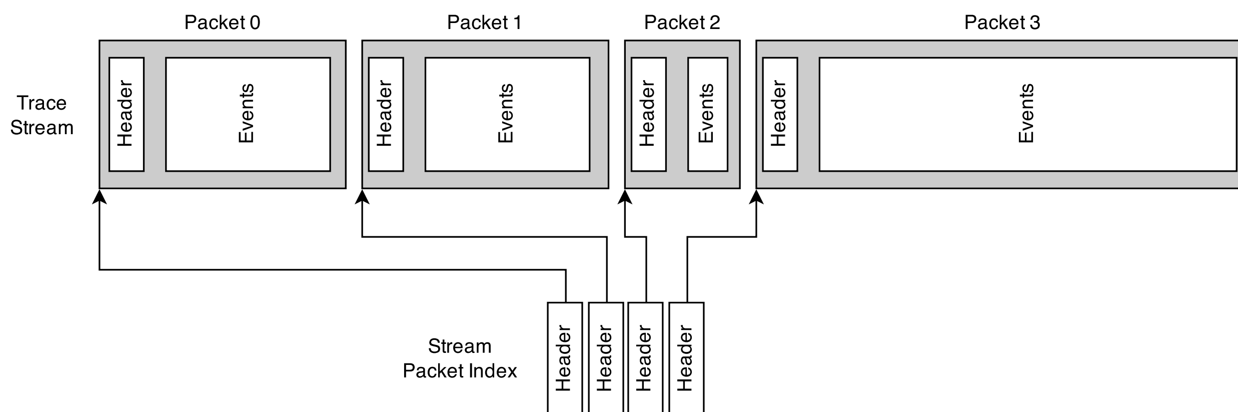

Figure 4.1 Event stream, packets and packet index

The CTF trace format uses the concept of *streams* of events. Each stream represents a subset of the events of the trace, often coming from the same source. In the case of LTTng kernel traces, each stream contains the events of one CPU on the traced system. A stream is further divided into variable-sized *packets*, which represent contiguous sets of events. The LTTng

kernel tracer creates a packet every time a sub-buffer from the trace ring buffer is committed (e.g., ready to write to disk or send through the network).

Each packet within a stream contains a *packet header*. This header contains information on the packet, such as its size and start time. Whenever a packet is added to a stream, its packet header is also written in the stream's *packet index* (see figure 4.1). This packet index therefore contains all the packet headers for the stream, thus allowing for fast seeking in the trace using the packet's start time and size to determine the offset of a specific time position in the trace.

As we will see, the trace format already hints at potential parallelization candidates: streams, for example, do not share information other than the read-only metadata and are good candidates for processing in parallel. Also, the concept of packet and the packet index will prove useful when discussing trace data partitioning and load balancing, in section 4.5.1.

4.4.2 Babeltrace trace reader

In order to read CTF traces, we will be using babeltrace. Babeltrace includes a basic CTF-to-text converting program in order to quickly read events from a CTF trace, as well as a C library in order to parse CTF traces from our program. In order to do so, the library first parses the TSDL metadata file, creating a binary parser. This parser is then used to decode the event data within the trace. The program may then query the library in order to obtain the event data.

Babeltrace works iteratively: the user creates an iterator at a specific timestamp in the trace, and the events are then parsed one-by-one and returned as the iterator advances. The concept of stream packets is also used here: babeltrace maps entire packets into memory one-by-one, unmapping the last one after all its events have been read. That way, the amount of memory used is small and nearly constant, even for very large traces.

4.4.3 Trace analysis model

Trace analysis is based on the concept of a *state system*. This state system holds what is called the trace's *current state*, which represents the exact state of the system at the latest point in time during the analysis. It holds information such as a map of all processes and their state (e.g. running, preempted or interrupted), the amount of data read from and written to every file, or the average latencies for every interrupt. The current state keeps both the information that will be presented at the end of the analysis and allows the analysis to do stateful processing, for example by allowing to only consider preemption during system calls.

To illustrate this, we can look at an analysis of the latency of I/O system calls per process. A

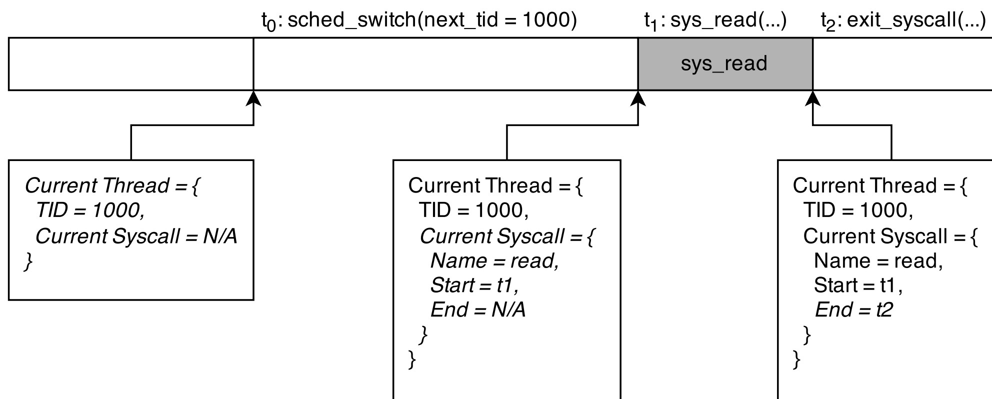

Figure 4.2 Current state updating as trace events are read

part of its execution is shown in figure 4.2. It's current state is represented as a map of the system's processes, as well as the currently running process for each CPU. When a process is scheduled to run by a CPU, the current state updates the currently running process for this CPU. If an I/O system call entry event is read, it records the currently running system call along with its start time for the currently running process. Then, when the system call exit event is read, it retrieves the ongoing system call for the currently running process and calculates the difference between the end time and start time. It then adds this latency to a list of system call latencies for the currently running thread. As we can see, the current state serves both as input and output to the analysis.

This processing, at first sight, seems to be inherently serial: the fact that events provide feedback to the current state, which in turn drives large parts of the analysis, means that there are strong serial dependencies between the events. However, as we will see in section 4.5.2, some features of traces and trace analysis may be exploited in order to break these dependencies, or more precisely to reduce their impact and defer their resolution without the need for costly synchronization.

4.5 Proposed Solution

The proposed solution hinges on the concepts described above in order to obtain an efficient parallelization of trace analysis. The design of a parallel implementation must take into account a number of aspects in order to achieve acceptable scalability. The following sections will detail the proposed solution with regards to its handling of data partitioning, load balancing, locking and synchronization, and data dependencies.

4.5.1 Trace data partitioning

Data partitioning plays an important role in parallelization: how the workload is divided among the processing nodes has an impact on the communications between nodes, the synchronization strategy as well as the load balancing. An improper partition of the data may lead to highly inefficient behavior and poor scalability.

Trace data may be organized along two "dimensions": the time dimension, with events arranged chronologically along a timeline, and the stream dimension, with events arranged by their source. In the case of kernel traces, these sources correspond to the CPUs of the traced system. Data partitioning therefore arises from these two dimensions: the trace data can be partitioned per stream or per time range.

Per-stream partitioning

Per-stream partitioning provides a natural way of assigning data to processing nodes. Its design is simple: assign each stream in the trace to an analysing thread, as can be seen in figure 4.3b. If there are more streams than processors, then the streams are enqueued and analyzed when a processor is free. Since each stream represents the events of a single traced processor, the analysing processor may be seen as simply replaying the events. Furthermore, this data partitioning lends itself well to CPU-based analyses. For example, an analysis that aims to extract the percentage of active CPU time for each processor may do so without the need of any communication or synchronization, since all the data needed for the analysis of a single CPU is held withing the stream (in this case, scheduling events.)

However, this approach presents a number of problems. The main problem is that it puts an upper bound on the number of parallel processing units that can be used for analysis. Indeed, the analyzing system will not be able to use more processors than the number of processors on the analyzed system, since a stream cannot be assigned to more than one processor. For example, if a 64-core machine is used to analyse a trace coming from an 8-core machine, 56 of its cores will be unused. It is also possible that some streams contain more events than others (i.e. some processors were more busy), thus creating load balancing problems. Furthermore, most trace analyses take place at process level, rather than at CPU level, such as analysing the state of processes over time, querying the amount of I/O per process or computing the critical path of a process' execution. These analyses rely on inter-stream dependencies, such as process migrations, that must be resolved.

Per-time range partitioning

Another solution would be to partition the trace along the time axis, as in figure 4.3a. This means that a trace could be split into as many chunks of data as necessary, which would then be dispatched for analysis on the processors. Each processor would then parse and analyse the events from all streams between two specific timestamps. The merging of events from different streams can be done using a minimum heap data structure, with the event's timestamp used as the sorting key. This ensures that each processor will parse events in a totally ordered manner (since all streams use the same monotonic clock source.)

This solution solves the main problem of the per-stream partitioning by removing the upper bound on the number of parallel units that can be used. Indeed, a trace may be split into as many time ranges as desired, thus allowing, theoretically, for an arbitrarily large number of processors to analyse the trace. Furthermore, it solves the data dependency problems of the per-stream analysis: each segment of trace data may be treated as an individual trace, since each processor has access to the data of every stream.

But this approach is also problematic, mainly due to the fact that trace events are unevenly distributed through time. In figure 4.3, this is shown by the grey-scale heat-maps. A typical system trace will see periods of high event densities, for example during a lock contention that generates a high number of system calls and scheduling events, as well as periods of low event densities, for example while a program waits for user input or network activity. This means that some processors will have more events to parse and analyse than others, leading to load balancing problems and hindering scalability. Another problem is that even though inter-stream dependencies are no longer a problem, temporal data-dependencies arise: events that occurred in a previous time range may have an effect on the analysis of the current time range, but are not accessible to the processor. These data dependencies must also be solved in order to obtain an accurate analysis.

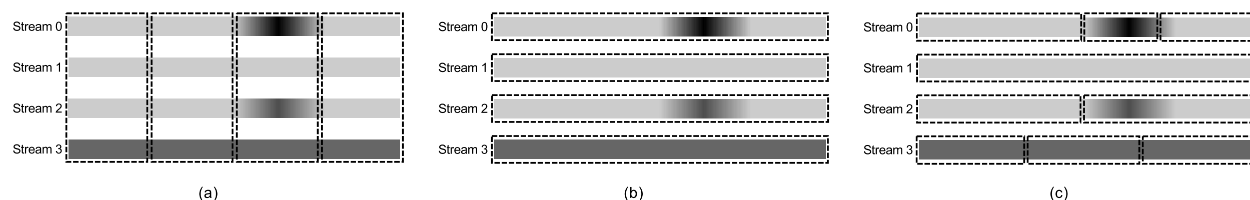

Figure 4.3 (a) Per-time range, (b) per-stream and (c) hybrid partitioning schemes, with gradient showing event density

Hybrid packet-based partitioning

In order to solve the problem of load balancing, we use an element of the CTF trace format called the stream packet index. As described in section 4.4.1, CTF traces define trace streams as a series of *packets*, which represent contiguous trace data. Each packet contains a packet header followed by the events' data. The packet header includes metadata such as the starting timestamp of the packet and its size, which is used when parsing the trace. Furthermore, every packet header is also written to an index file called the stream packet index. This index allows for fast timestamp seeking in a trace stream by enabling a binary search to be done on packet start times, and getting an offset in the stream file where to start reading.

Our solution to the data partitioning problem is to use this packet index to create balanced workloads. If we assume that each event has more or less the same size, we can use the packet size in order to estimate the number of events in a packet. By iterating through the stream packet index, we can accumulate packets until a certain threshold size is met and thus create *chunks* of data from a single stream between two timestamps, as in figure 4.3c. These chunks can then be added to a shared queue for analysis by the processing nodes. We therefore use a hybrid data partitioning scheme, splitting workloads both per-stream and per-time range, while assuring that the number of events in each partition is approximately constant. This solves the problem of load balancing by creating equal workloads, whose size can be parameterized using a smaller or larger threshold.

4.5.2 Resolving data dependencies

The aforementioned problems of data dependencies are closely linked to the design of trace analyses. Most trace analyses proceed using the concept of a current state, which is both read and written to as the analyser reads the trace events (see section 4.4.3). This current state holds, for example, the state of each process running on the system (e.g. running, blocked, preempted) or the amount of data read from and written to each file. Analyses can write to the current state to update the values that will be presented at the end of the analysis, but can also read from it in order to know, for example, the starting time of the current system call in order to calculate its latency.

Temporal dependencies

In our solution, each thread holds its own independent current state, which is empty when the thread starts. Problems arise when some parts of the current state are unknown due to some events being in a previous chunk of trace data. Since the trace is partitioned in time ranges,

the current state at the beginning of each time range is unknown to the thread processing it. For example, it is possible that a system call spans two time ranges, with its entry event in one time range and its exit in the next time range. The thread processing the second time range does not have access to the start time and input parameters of the system call, while the thread processing the first time range does not have access to its end time and return value.

Fortunately, most analyses are not deeply dependant on the system's initial state: unknown state does not radically affect the results of the analysis by propagating erroneous values. Indeed, this unknown state happens invariably at the beginning of any trace, even if analysed sequentially. Some events, implemented by the LTTng tracer, provide complementary information (such as the file names linked to file descriptors that were open before tracing started) and are triggered only at the beginning of the trace, but these are usually not a hard requirement for analyses. We can therefore tolerate this unknown state for certain values, provided that the next event affecting this value will determine unambiguously its new state. In more formal terms, our analysis must have a 1-to-1 relationship between transitions and state, such that each transition points to one and only one state.

The algorithm for resolving data dependencies goes as such: each worker thread keeps a thread-local current state initialised to default values. It then iterates through the events of the trace data that was assigned to it, updating the current state as events are read. If an event causes processing that depends on an unknown current state value, the worker thread saves the necessary information separately from the current state. Once all the events are read, the worker thread returns its current state which holds the results of the analysis on its chunk of data, plus all the unknown state values that need further processing.

In order to resolve the unknown values, a merging phase is executed on the results of the worker threads. The merging routine does two things: first, it gathers the content of both states in order to obtain a single current state, for example by summing the total amount of data read from a file during both time ranges. Second, it resolves unknown values by propagating values from the earlier chunks' current state to the later chunks. For example, an interrupt that spanned two chunks will have the interrupt's start event in a first chunk and the end event in a second chunk. When merging the chunks, we detect both an unfinished interrupt in the first chunk and an unknown interrupt ending in the second. By matching these two, we can reconstruct the whole information for that specific interrupt and add its latency, for example, to a global list of interrupt latencies.

The merging phase operates in a sum-like fashion, whereas the current states of the first two chunks are merged into a resulting current state, which is then merged with the third chunk's

current state, and so on. The final current state will therefore hold all the merged information from all the chunks. This means that merging must occur in chronological order. While this seems to introduce strongly serial processing to the algorithm, two aspects can be exploited in order to exploit parallelism during the merge phase. First, merging is a relatively small task executed once for every chunk, and can easily be pipelined into the general processing described before: as soon as the chunks are processed, they may be merged in parallel while the rest of the trace is analysed. This is hampered by the fact that chunks must be treated in chronological order, but assuming that chunks are balanced and processed in chronological order, this fact should not cause bad pipeline stalling. Second, merging is an associative binary operation, meaning that the parallel scan algorithm described in section 4.3.3 could possibly be used to introduce parallelism into the merge phase.

Inter-stream dependencies

The above data dependency resolving strategy works well for temporal dependencies. However, the hybrid packet-based trace data partitioning scheme detailed above introduces another data dependency problem: inter-stream dependencies, where some of the dependent information is located in another stream. This happens, for example, when a process is migrated from one CPU to another during tracing, such that its events are on different streams.

In order to solve these dependencies, we can use a similar approach as to the one described above. The difference is that dependent events may now occur both in a prior chunk as well as in a "concurrent" chunk, meaning a chunk who's time range overlaps the time range of the current chunk. Since inter-stream dependencies happen on process migrations, we can detect these migrations by reading the migration event, and treat all subsequent analysis of the migrated thread as dependant on its execution in another stream. We therefore need to keep multiple current states, one for each "execution", where an execution is a series of events for a process separated by a migration. We may then apply the same merging algorithm as before on these executions, and thus propagate unknown values throughout our analysis.

In figure 4.4, the analysis would work as follow:

1. As chunk 0 is analyzed, a migration event occurs. The current state is saved for execution A. When the process migrates back to chunk 0, its current state is saved.
2. In parallel, the other chunks are processed. They treat migration events similarly, and we get a number of executions (A, B, C, D, E).
3. Once analysis is done, the merging begins. Merging is done in chronological order of start time, such that chunks 0 and 1 are merged first.

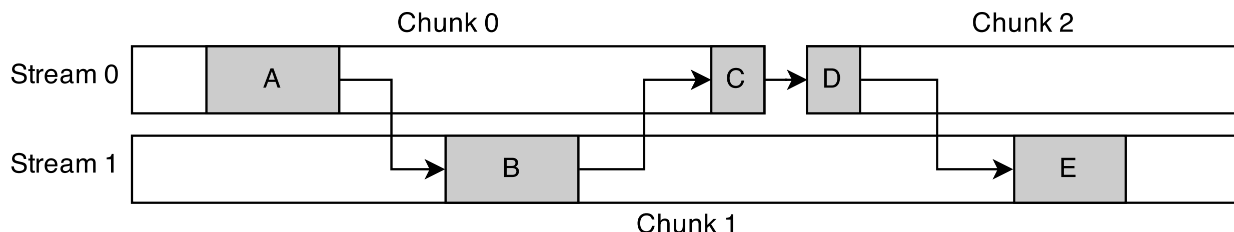

Figure 4.4 Data dependencies between executions across streams and chunks

4. The executions in chunks 0 and 1 are sorted by start time. Executions A, B and C are merged. Execution E is kept unmerged, since it occurs after the end of chunk 0 and is dependent on execution D, which is not yet available.
5. The merged results of chunks 0 and 1 are then merged with chunk 2. The temporal dependency at execution D is merged, then the inter-stream dependency at E. All the chunks have been merged and the analysis is done.

This solution adds additional work in two ways: first, it requires chunks to be merged in order of starting time, meaning that the chunks from different streams must be sorted after iterating through the packet indexes. Second, it requires the execution of the merging routine for every "sub-chunk" created when a migration occurs. These two factors do not affect the overall performance greatly, since the number of chunks is relatively low and therefore fast to sort, and since migrations are unfrequent and therefore do not add a significant amount of merging.

4.6 Experimental Results

4.6.1 Experimental methodology

In order to assess whether parallelization is a viable strategy to enhance the performance of trace analysis, we must not only show that our method provides speedup in the analysis of trace data, but also that this speedup is *efficient*: it is sometimes possible to obtain large speedups given larger amounts of computing power, but that may come at the cost of wasted computing power due to low efficiency.

Furthermore, trace analysis suffers from relying on storage device speed, since trace data resides on disk and may not be able to fit into main memory. Since disk accesses are orders of magnitude slower than accesses to memory and cache, they can become a bottleneck if the amount of work on the CPU is less than the time spent retrieving data from disk.

Finally, the CPU workload of trace analysis is strongly dependant on the speed of trace decoding (in our case provided by the babeltrace library). Testing our parallel trace analyses using this library ignores possible future improvements to trace decoding which could hinder the efficiency of parallelization by lowering the CPU workload and increasing the effect of the I/O bottleneck.

For our suggested method to be deemed efficient, we must explore these possibilities and weigh their effect on trace analysis parallelization. To do this, we created a simulation program, described in the following section, which allows us to run experiments that will help calculate bounds on the parallel efficiency on various storage devices and with (simulated) faster trace decoding.

The following results were all obtained on a server equipped with a quad-socket Supermicro H8QGL motherboard, with 4 AMD Opteron 6272 16-core 2.1 GHz processors (for a total of 64 cores) and 128 GB of DDR3 SDRAM.

4.6.2 Simulation program

In order to better assess the effect of memory and I/O operations on parallel trace analysis, our simulation program simulates the CPU, memory and disk workload of a trace analysis program. The simulation program works as follows: a single file is given as input to the program, which separates the reading of the file to multiple threads. Each thread maps a part of the file into memory, then proceeds to read every first byte aligned on page boundaries. This way, we ensure that each byte read will trigger a page fault with minimum accompanying CPU work. The program then does an arbitrary amount of work for every byte in the form of a loop by simply incrementing a counter. Once all the threads are finished, the program returns the sum of all the thread results. Listing 4.1 shows a simplified version of the code used. The actual program was implemented using the OpenMP compiler instructions.

This program allows us to tweak certain parameters in order to observe the program's scalability under different conditions. We can change the amount of CPU work done per page fault by setting a higher or lower number of iterations. This allows us to simulate a more or less complex trace analysis, or to simulate a more optimized trace decoding. We can also experiment with different sizes for the memory mapped regions in order to simulate trace packets of various sizes. Finally, we can also tweak the parameters sent to the I/O functions, allowing us to activate or deactivate features such as read-ahead or pre-faulting of pages.

By tweaking the parameters of the program, we can simulate a throughput similar to that of reading a trace file sequentially, without doing any analysis. This will give us an upper bound

Listing 4.1 Simulation program code

```
threadRoutine(size_t chunk_size, void *chunk_offset, int file) {
  char *buffer = mmap(NULL, chunk_size, PROT_READ, MAP_PRIVATE, file,
      chunk_offset);
  unsigned long sum = 0;
  for (size_t i = 0; i < chunk_size; i += PAGE_SIZE) {
    sum += buffer[i];
    /* burn CPU */
    for (int j = 0; j < ITERATIONS; j++) {
      sum++;
      asm(""); // Prevents compiler from optimizing the loop
    }
  }
  munmap(buffer);
  return sum;
}
```

in terms of how much trace analysis can benefit from parallelization on various I/O hardware.

Concurrent memory operations

When running benchmarks on the simulation program, significant slowdowns were observed as the number of concurrent threads increased. This effect can be seen in figure 4.5. However, these slowdowns were unrelated to I/O operations and were mainly observable during cache hot benchmarks, meaning that the trace data was contained entirely within the system's page cache (i.e. the disk-backed pages kept in memory by the operating system). This slowdown was not created by a contention in our program either, nor by cache thrashing, since no data was shared between the threads.

Upon further inspection, it appeared that the bottleneck was within the kernel itself, more specifically within the memory management module of the kernel. Under the Linux operating system, each process holds a data structure describing its virtual memory, as well as a single read/write lock for the whole address space. These components are shared by all the threads within the process, since threads operate on the same address space. Whenever a memory operation (i.e. `mmap` and `munmap` system calls) or a page fault occurs, the global lock is taken in order to protect the global structures from concurrent accesses. This means that the memory operations and page faults of our simulation program are being serialized, thus creating a strong contention on the kernel lock and causing the observed slowdown.

A workaround to this problem would be to use processes instead of threads, thus separating the address spaces of our workers. However, this approach increases the complexity of the code,

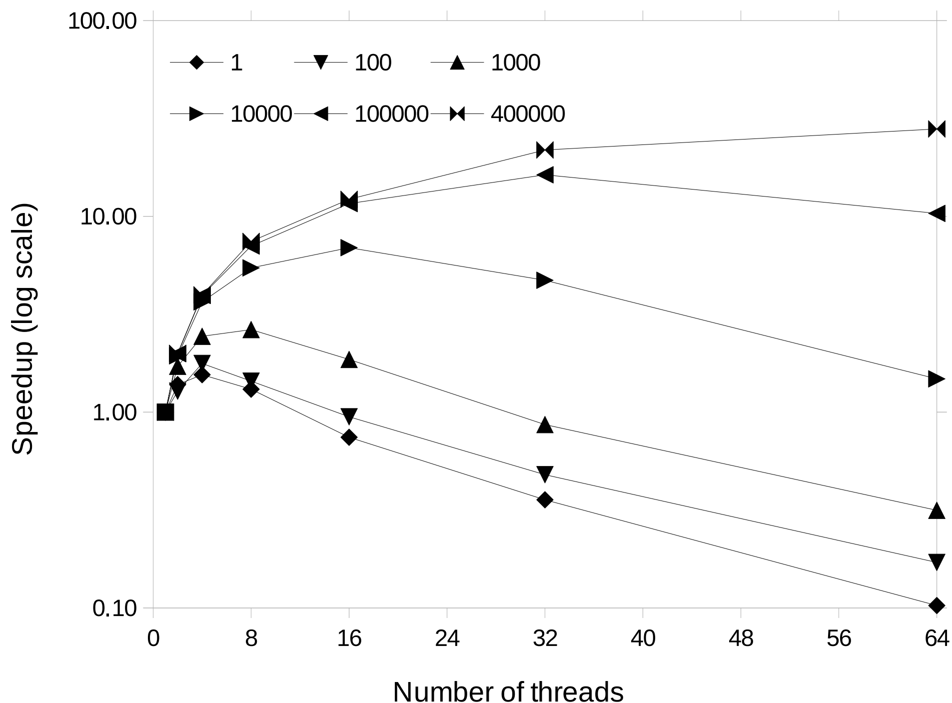

Figure 4.5 Speedup of concurrent memory operations for increasing CPU workloads (number of iterations)

since inter-process communications must be handled instead of relying on shared memory. Another approach would be to modify the kernel code in order to improve the scalability of memory operations, as proposed by Clements et al. (2012).

In order to work around this issue without modifying the kernel or making our application too complex, we decided to simply use a pipelined approach in order to remove the overhead due to lock contention. The pipelined program works by doing all the memory operations and page faults serially: one thread map the chunks in a serial manner and hands them off to be processed in parallel, with the final stage of the pipeline doing a serial unmapping of the chunks. This pipelined program was implemented using the Intel Threading Building Block (TBB) library, which allows the creation of pipelined tasks using a C++ API.

Parallel efficiency on various storage devices

The following results were obtained by running our pipelined simulation program on different storage devices. The program's parameters were selected such that the sequential throughput in terms of data processed per second would be on par with the throughput of a sequential

reading of a trace using the babeltrace reader. The devices used are summarized in table 4.1. The sequential read speed was acquired experimentally using the command `hdparm -t`, which gives sequential read speed without file system overhead.

Table 4.1 Test storage devices specifications

Model	Interface	Sequential Read Speed (MB/s)
WD RE4 HDD	SATA II	135
Intel SSD 520	SATA II	250
Intel DC SSD P3700	PCIe 2.0	1145

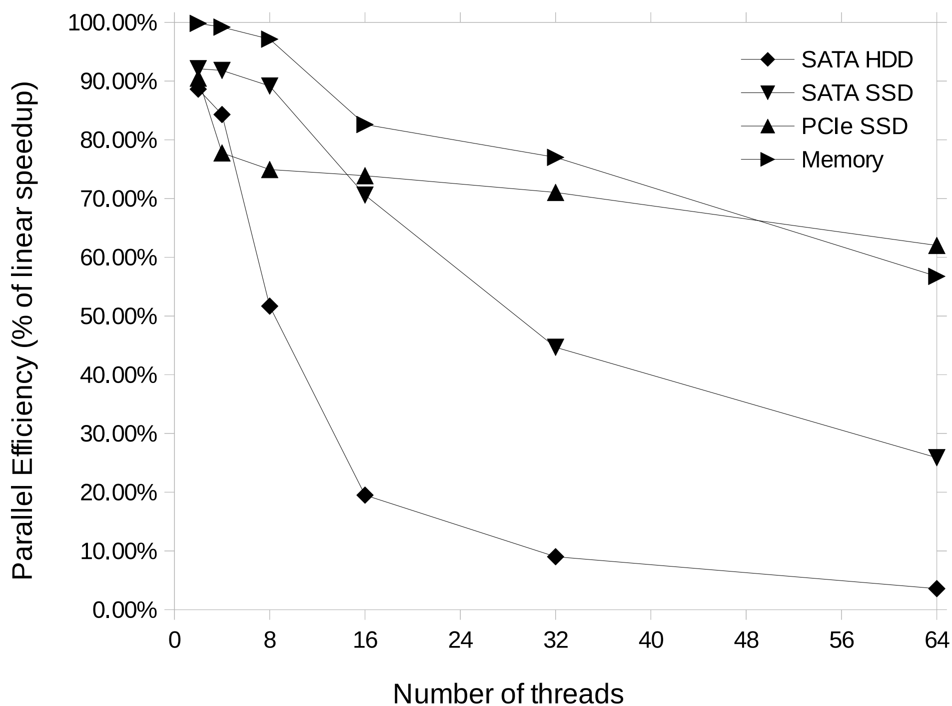

Figure 4.6 Parallel efficiency of simulation for various storage devices

Figure 4.6 shows the parallel efficiency for our simulation program reading trace data from various storage hardware as well as directly from memory (i.e. from the page cache). Parallel efficiency is measured as the percentage of linear speedup achieved, using the formula $E = S/N$ where S is the speedup obtained using N threads. Speedup is defined as $S = t_1/t_N$ where t_1 is the serial time and t_N the time with N threads. An optimal solution would have a

constant 100% parallel efficiency for any N and an efficiency of $1/N$ means no speedup over the sequential solution.

The results indicate that the efficiency of processing data from a hard disk drive quickly drops as the number of threads increases. However, we still maintain a 60% parallel efficiency at 8 threads, which translates to a speedup of 5. In other words, a parallel trace analysis could expect up to 5 times speedup using 8 threads, or process 5 times as much data in the same time period.

If better storage devices are used, such as an SSD, we get a higher than 70% parallel efficiency up to 16 threads, which translates to a speedup of 11. For a high-performance PCIe SSD, the parallel efficiency is above 63% up to 64 threads, which translates to a speedup of 40. At the moment of this writing, such high-performance SSDs are becoming readily available in consumer- and server-grade markets, pushing back the boundary on I/O-boundedness for many types of programs.

Parallel efficiency with optimized trace decoding

In order for trace analysis to be efficiently parallelizable, it is important to consider the impact of possible future optimizations in the CPU efficiency of trace decoding. In other words, if trace decoding becomes more efficient in its CPU usage, will trace analysis become bounded by I/O operations, and will therefore cease to gain from parallelization?

In order to test this scenario, we changed our benchmarks such that the simulation program would run not only with parameters that simulated throughput similar to current sequential trace decoding, but also with parameters that simulated the cases where decoding would be twice, 4 times, 8 times and 16 times as fast. These benchmarks were again run on the storage devices from table 4.1 and on memory. The results are shown in figure 4.7.

The parallel efficiency of analyzing data from a hard drive drops dramatically as simulated trace decoding improves. This is due to the data being analyzed faster than it is read from disk, because of the slow accesses to the drive. We could therefore argue that parallelization is not a future-proof solution, since its efficiency is greatly hindered when using better trace decoding. However, hard disk drives are being supplanted by more efficient storage devices, namely high performance solid state drives. When using these types of drives, the drop in efficiency is not as severe as before: using a regular SATA SSD, we keep decent parallel efficiency even with twice and 4 times faster trace decoding. Using a high performance PCIe SSD, we stay above 50% efficiency up to 32 threads with a 4 times faster trace decoding, and decent parallel efficiency for 8 times faster trace decoding.

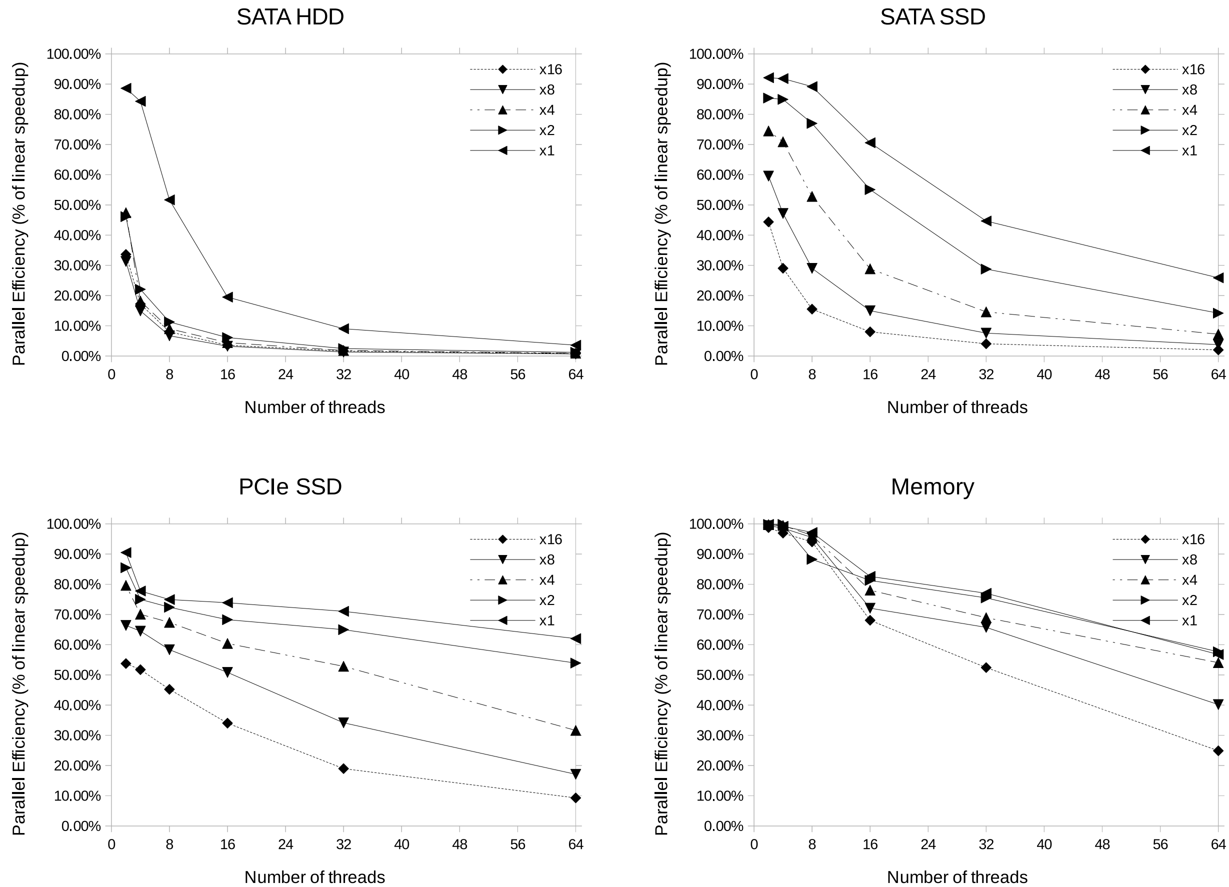

Figure 4.7 Parallel efficiency as simulated trace decoding speed increases

An aspect to keep in mind is that the results shown above represent a lower bound on parallel efficiency: they represent the worst case scenario for trace analysis, since the workload simulates events that are only decoded, without analysis. Normal trace analyses execute possibly CPU heavy operations when events are read, meaning that the actual CPU workload (and therefore parallelization opportunity) would in fact be greater, thus improving parallel efficiency. Furthermore, by freeing up CPU time from trace decoding, we would possibly be able to run more complex analyses or a higher number of simultaneous analyses, which would in turn benefit from the parallel execution.

4.6.3 Parallel trace analysis efficiency

We have shown in the previous section that parallel trace analysis would not be seriously hindered by I/O-boundedness problems, whether with the current trace decoding speed or with better trace decoding. We can now look at the parallel efficiency for actual trace analyses,

using an implementation of the proposed solution.

The solution was implemented using the Qt Concurrent [3] framework, which offers a simple MapReduce-like API for parallelization. The framework allows for the creation of map and reduce tasks, which are used for the chunk analysis and merge phases, respectively. The trace on which the analyses were run is from an 8-core machine (thus contains 8 streams of events) and contains 44,897,970 events.

Test analyses

The following three analyses were tested:

- **count**: the *count* analysis represents the simplest possible "useful" analysis. It simply counts the number of events in the trace and returns the total to the user. This analysis does little more than parsing the events, and can be treated as a lower bound in terms of parallelization potential since so little CPU work is done.
- **cpu**: the *cpu* analysis is a more useful, but still very simple analysis. It gathers the percentage of active CPU time per-CPU and per-thread using scheduling events. This analysis is slightly more complex than *count*, but only looks at scheduling events.
- **io**: the *io* analysis gathers the amount of data read and written by every process. It does so by parsing read- and write-related system calls. This analysis is more complex than the last two: it requires stateful processing (in order to know which system call exited, since system call exit events do not specify the system call that has exited) and slightly more complex merging. It therefore requires more CPU work, and should benefit more from parallelization.

The results in table 4.2 show the elapsed time, speedup and parallel efficiency for the three different analyses, with trace data stored on a PCIe SSD, while figure 4.8 shows the efficiency for the analyses with trace data stored on the storage devices from table 4.1 and in memory. It is important to note that the current parallel trace analyses suffer from the concurrent memory operation problem mentioned in section 4.6.2 since the memory operations are not yet pipelined.

We can see, in figure 4.8, that parallel efficiency is lower than with our simulation program. This is due to various factors: the bottleneck in the kernel due to the non-pipelined memory operations, inefficiencies within the trace decoding library and less efficient I/O operations (e.g. no read-ahead or pre-faulting), for example. There is also a fixed amount of serial work at the beginning of the analysis due to the parsing of the trace metadata, which becomes proportionally smaller as trace data size increases. As for the serial work done in the merge

3. http://doc.qt.io/qt-5/qtconcurrent-index.html

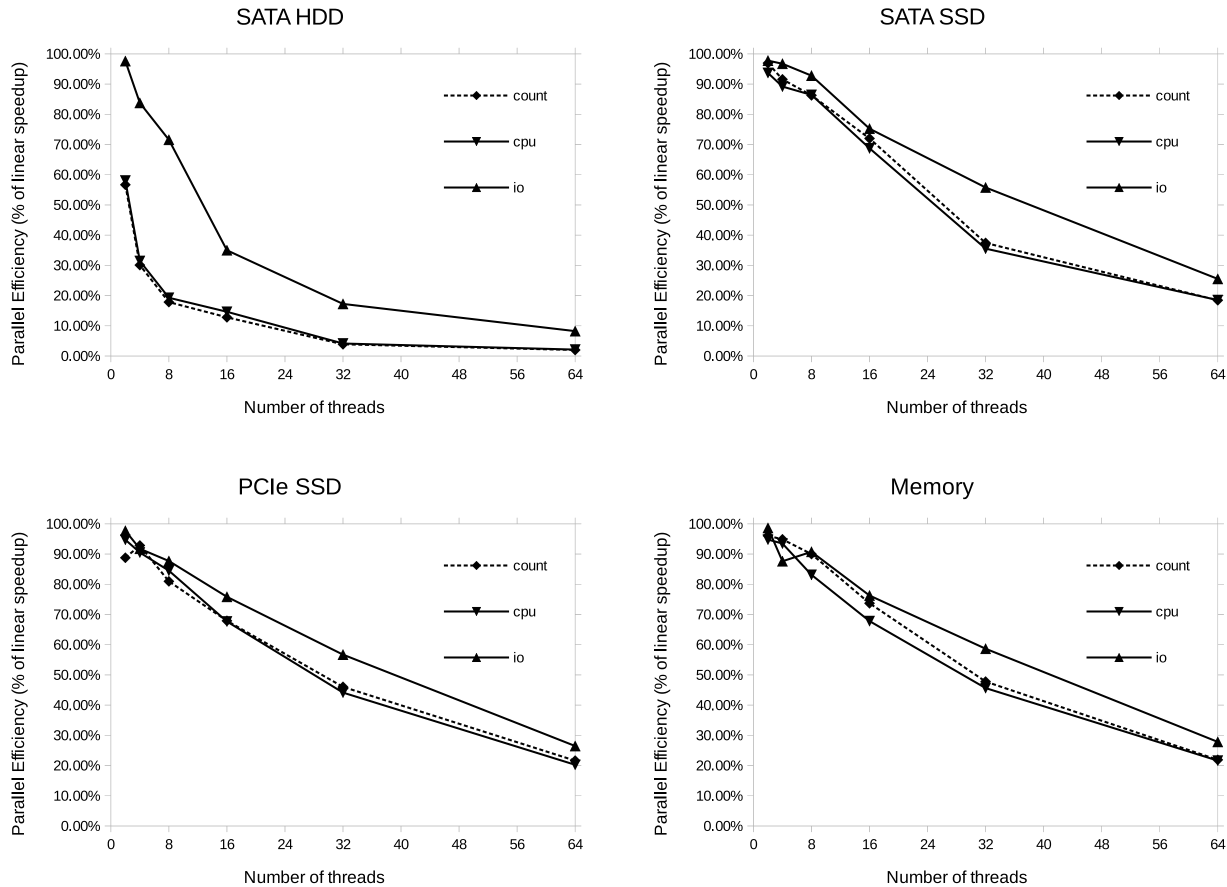

Figure 4.8 Parallel efficiency of trace analyses for various storage devices

phase, it should not affect the total run time, since this phase can be pipelined into the rest of the program, meaning that every time a chunk is ready to be merged, the merge is done in parallel with the other chunks being processed.

The babeltrace trace reading library was the source of various bottlenecks: its inability to create multiple iterators on a single trace lead to a work-around which functioned by copying a certain amount of shared data when creating a new iterator. However, this method brought its own set of inefficiencies, mainly in the form of a certain amount of unwanted and unnecessary locking during the copy. We can therefore attribute a part of the loss of parallel efficiency to the serial design of the trace decoding library, which had nothing to do with problems in the detailed solution.

Still, the benchmarks show appreciable efficiency up to 16 threads, and efficiency above 80% up to 8 threads, which are typical levels of parallelism found in regular workstations. These efficiencies translate to speedups of one order of magnitude ($> \times 10$) on consumer-level

Table 4.2 Benchmark results for trace analyses on PCIe SSD

		Threads						
Analysis	Metric	1	2	4	8	16	32	64
count	*Time (s)*	108.21	60.95	29.14	16.70	9.96	7.35	7.82
	Speedup	-	1.78	3.71	6.48	10.87	14.73	13.84
	Efficiency	-	.888	.928	.810	.679	.460	.216
cpu	*Time (s)*	116.94	61.72	32.31	17.30	10.80	8.28	9.02
	Speedup	-	1.89	3.62	6.76	10.83	14.12	12.96
	Efficiency	-	.947	.905	.845	.677	.441	.202
io	*Time (s)*	191.48	97.99	52.17	27.29	15.78	10.55	11.31
	Speedup	-	1.95	3.67	7.02	12.13	18.15	16.93
	Efficiency	-	.977	.918	.877	.758	.567	.265

hardware. This is a significant speedup, allowing for time-consuming analyses to be processed in seconds rather than minutes.

As we can see from the results, the parallel efficiency is higher when running more complex analyses, visible here in the *io* analysis. This is expected, since more complex analyses rely on higher CPU workloads which are better suited to parallelization. However, it is interesting to note that a slight increase in complexity (in this case mainly due to the usage of a hash-map in the *io* analysis) translates to significantly better parallel efficiency: the efficiency with 32 threads for the *io* analysis is 28.5% higher than for the *cpu* analysis (from 44.1% up to 56.7%). More complex analyses could potentially benefit even more from parallelization.

We also see that, apart from the results where trace data was stored on a hard-drive, the analysis is not bound by the accesses to the storage devices. We can therefore conclude that trace decoding in its current implementation is not I/O bound when reading trace data from solid state drives. Furthermore, using the results from the previous section, we can extrapolate that this efficiency will hold even as trace decoding improves.

4.7 Conclusion and future work

This research has shown that parallelization provides an efficient way to enhance the performance of trace analysis. We developed a solution which provides balanced workloads using data available directly from the trace format, and which keeps locking and synchronization

to a minimum by deferring the resolution of data dependencies. Using this solution, we implemented parallel trace analyses that yielded speedups of up to 18 times, with good parallel efficiency up to 32 threads. We also showed, using our simulation program, that trace analysis is no longer I/O-bound when using modern but readily-available solid state storage devices, and that this holds even as trace decoding improves.

By allowing for greater amounts of data to be processed, parallelization brings a viable solution to the problem of the increasing amount of trace data. Moreover, it allows us to use the same highly parallel systems that we trace in order to analyse the resulting traces, without confining ourselves to single-threaded analysis.

In order to achieve better scaling, improvements must be made to the trace decoding library. These improvements will allow for more efficient parallelization by reducing unnecessary copying of data and lock contention. Since only read-only data is shared between the threads, the potential speedup could be brought up at least to the level of the simulation program. Furthermore, the problem of concurrent memory operations in the Linux kernel could be resolved, or worked around, for interesting scalability gains. We know that these improvements will enhance scalability, since we showed in section 4.6.2 that potentially better speedups are possible with good enough storage devices.

Trace analysis results can be stored in the State History Tree database in order to allow fast access to the current state at any moment in the trace. For the moment, the construction of the State History Tree relies on intervals being added in sorted time order, to avoid rebalancing. A possible future work would be to allow the history tree to be rebalanced, so that we may add unsorted intervals, therefore allowing for parallel analysis to output into a State History Tree.

Distributed trace analysis would also be an interesting solution, especially in the context of analyzing distributed traces: the concepts detailed in the present research could possibly be applied to analyzing trace data on multiple nodes, allowing for dedicated analysis nodes within a computing infrastructure. Furthermore, some current trace analyses allow for live processing of traces streamed over the network. Future work could adapt parallel trace analysis to the context of live tracing.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Cette section contient de plus amples détails sur la présente recherche. En plus de faire un bref retour sur les résultats présentés ci-haut, nous approfondirons les observations par rapport aux opérations concurrentes sur la mémoire. Nous verrons aussi d'autres pistes de solutions envisagées puis abandonnées lors de la recherche.

5.1 Retour sur les résultats

Les résultats présentés au chapitre précédent démontrent que la parallélisation de l'analyse de traces est une méthode efficace et viable, même lorsque le décodage d'événements est amélioré. De plus, il est important de noter que le gain de performance grâce à la parallélisation amplifie les améliorations au décodeur de traces : en effet, en se confinant à une analyse de traces séquentielle, on limite le degré d'accélération obtenu par un meilleur décodage.

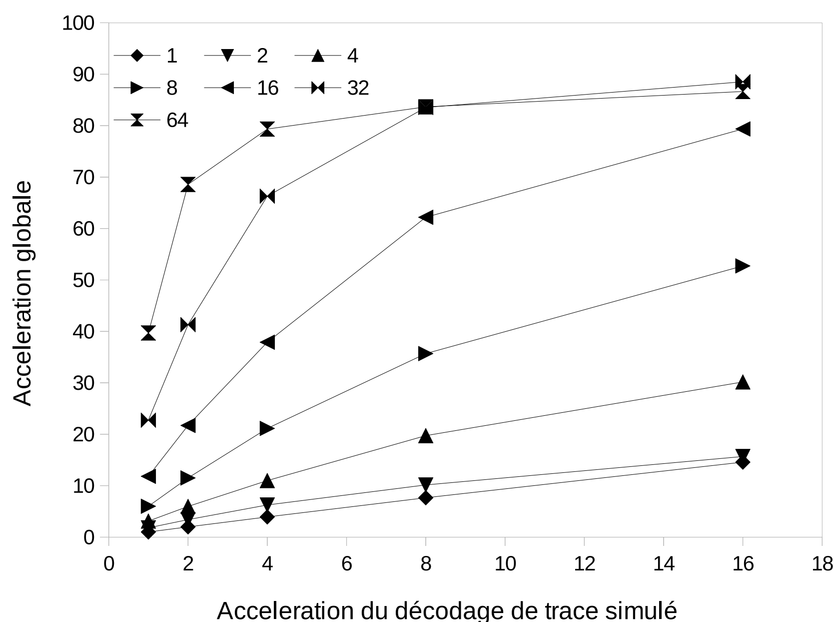

Figure 5.1 Effet de l'accélération du décodage de trace simulé sur l'accélération globale pour différents nombres de fils d'exécution

Par exemple, si le décodage est amélioré de sorte que celui-ci est 8 fois plus rapide, on n'obtient qu'une analyse 8 fois plus rapide. Or, avec une analyse parallèle sur 16 fils d'exécution, cette amélioration ce traduit par une exécution 62 fois plus rapide que l'exécution séquentielle avant

l'amélioration. En d'autres mots, si l'on n'exploite pas les capacités de calcul parallèle de notre système d'analyse, on se retrouve à faire une croix sur d'importants gains potentiels en performance. On peut voir ces gains potentiels sur la figure 5.1 : en se limitant à l'exécution séquentielle, les gains en terme d'accélération du décodage de la trace se limitent à 16 fois. Or, un système possédant 16 ou 32 cœurs peut potentiellement obtenir des gains beaucoup plus haut en exploitant la parallélisation.

5.2 Opérations concurrentes sur la mémoire

Une des embûches majeures à la parallélisation de l'analyse de traces s'est avérée provenir non pas de notre application, mais de la gestion de la mémoire dans le noyau. En effet, l'exécution de programmes s'appuyant sur un grand nombre d'opérations sur la mémoire provoque un goulot d'étranglement dans le système de gestion de la mémoire de Linux. Nous verrons ici cette problématique plus en détail, de la découverte à la résolution.

5.2.1 Résultats et traces

Le problème des opérations concurrentes sur la mémoire n'était pas visible lors de tests préliminaires sur un système de huit cœurs. Cependant, lors des tests sur notre système de 64 cœurs, l'accélération devenait de plus en plus faible et même devenait fortement négative à mesure que le nombre de fils d'exécutions augmentait (voir figure 4.5).

Afin d'évaluer le problème, une trace du système fût enregistrée avec le traceur LTTng et analysée grâce à l'outil Trace Compass. Les résultats de l'analyse, visibles à la figure 5.2, montrent l'état des différents fils d'exécution dans le temps. Le vert signifie que le fil exécute du code utilisateur, le bleu montre les appels système, le jaune signifie que le fil est bloqué et l'orange montre une préemption.

Puisque le programme est simple, on peut aisément identifier les différentes sections de l'exécution dans la trace. La première partie (avant la ligne verticale bleue) correspond au mappage des données en mémoire (`mmap`) puis de la lecture. La ligne bleue verticale correspond au "dé-mappage" (`munmap`) simultané des données lorsque les fils ont terminé leur travail. Les fils sont ensuite synchronisés sur une barrière et terminent simultanément dans la dernière section verte.

Le programme tracé ne faisait que lire les données sans faire de travail CPU. Cependant, la quantité de temps passé dans l'état "bloqué" est étrange et ne correspond pas au comportement habituel. La figure 5.3 montre un zoom sur la section de la trace où les données sont enlevées de la mémoire avec `munmap`. Ces appels sont tous commencés en même temps, mais terminent

les uns après les autres, de manière séquentielle. Il est clair que le système d'exploitation sérialise l'exécution de ces opérations. Le même phénomène est visible dans la première section de la trace, avec les appels à `mmap` et `madvise` ainsi qu'avec les fautes de pages. En somme, toutes les opérations sur l'espace mémoire virtuel du programme sont sérialisées.

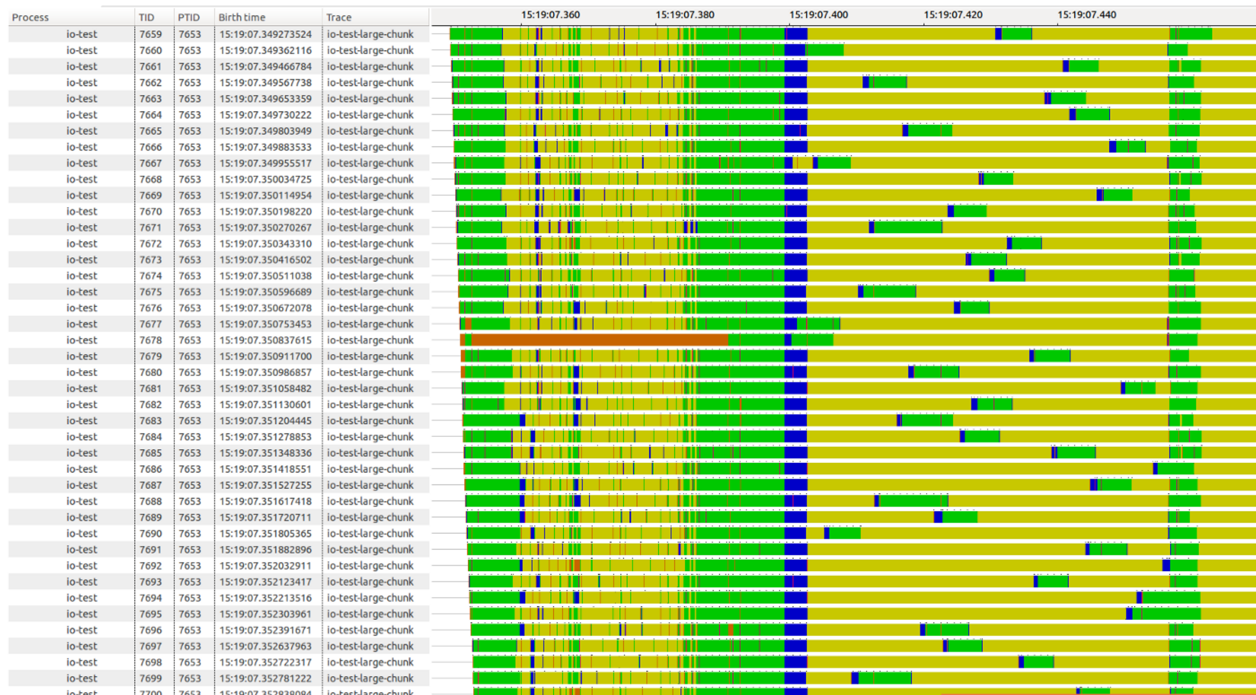

Figure 5.2 Trace d'exécution des opérations concurrentes sur la mémoire

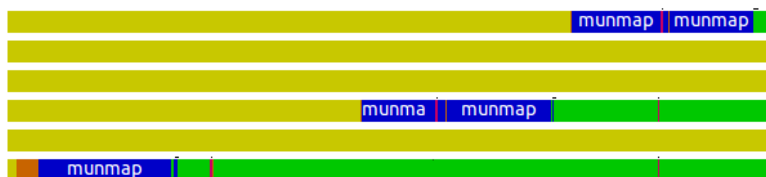

Figure 5.3 Détails de la trace d'exécution montrant la sérialisation des appels à `munmap`

5.2.2 Source du problème

L'avantage d'utiliser le système d'exploitation Linux est que nous avons accès à son code source. On peut donc chercher le code source de l'appel système `mmap`, dans l'espoir de retrouver la source de notre problème. Le code source de la fonction au cœur de l'appel système `mmap` est donné dans le listage 5.1.

Listing 5.1 Code source de mmap

```
/* Dans mm/util.c */
unsigned long vm_mmap_pgoff(struct file *file, unsigned long addr,
  unsigned long len, unsigned long prot,
  unsigned long flag, unsigned long pgoff)
{
  unsigned long ret;
  struct mm_struct *mm = current->mm;
  unsigned long populate;

  ret = security_mmap_file(file, prot, flag);
  if (!ret) {
    down_write(&mm->mmap_sem);
    ret = do_mmap_pgoff(file, addr, len, prot, flag, pgoff,
            &populate);
    up_write(&mm->mmap_sem);
    if (populate)
      mm_populate(ret, populate);
  }
  return ret;
}
```

On peut voir, à la ligne 12, qu'un verrou (plus précisément un sémaphore) est pris. Ce verrou est contenu dans la structure `mm` de type `mm_struct`, qui contient la description de l'espace d'adressage virtuel du programme. Comme les fils d'exécution partagent le même espace d'adressage, cette structure est partagée par tous les fils. Et puisque ceux-ci effectuent des opérations concurrentes sur la mémoire, on obtient une forte contention sur le verrou de cette structure et une sérialisation des opérations.

5.3 Autres solutions explorées pour l'analyse parallèle

Outre la solution présentée ici, d'autres pistes de solutions avaient été explorées pendant la recherche. L'une de ces pistes, prometteuse en théorie, se basait sur l'exécution parallèle de machines à états (voir section 2.4.3). Cette méthode semblait tout à fait appropriée pour l'analyse de trace, puisque l'analyse est modélisée comme un système d'états, qui est en quelque sorte un ensemble de machines à états dépendantes.

La solution explorée était en majeure partie basée sur l'exécution énumérative retrouvée dans

les travaux de Mytkowicz et al. (2014). Cette solution consiste à garder, pour chaque machine à états, un vecteur d'états courants plutôt qu'un seul état. Ce vecteur représente tous les états courants possibles pour chaque état initial. On peut ainsi savoir dans quel état notre machine serait selon l'état au début du traitement. Cela nous permet donc de traiter chaque partie de la trace indépendamment, puis de propager l'état initial dans chaque partie à partir du début.

Cette exécution demandait donc deux phases : la première phase permet d'obtenir le vecteur d'états pour la fin de chaque morceau de trace, pour ensuite propager l'état initial à partir du début. La deuxième phase permet d'effectuer l'analyse pour de bon, en prenant les bonnes actions à chaque événement lu (voir figure 2.5).

Cette approche à l'avantage de permettre une analyse sans avoir à gérer de potentiels états inconnus. Cependant, elle contient un désavantage majeur dans le cas de l'analyse de trace : puisque deux phases sont nécessaires, cela veut dire que la trace doit être lue à deux reprises. Et comme la majeure partie du travail dans l'analyse de trace est le décodage des événements, et que les événements décodés ne peuvent pas être tous gardés en mémoire, on se retrouve avec un surcoût énorme qui limite grandement l'accélération de cette solution. Nous avons implémenté un prototype de cette solution, mais les performances n'étaient pas acceptables pour qu'elle soit retenue.

CHAPITRE 6 CONCLUSION

Nous conclurons notre recherche dans ce dernier chapitre en présentant une synthèse des travaux, puis en discutant des limites de la solution proposée et des améliorations futures dont elle pourrait profiter.

6.1 Synthèse des travaux

Dans la présente recherche, nous avons évalué la parallélisation comme solution au problème de mise à l'échelle de l'analyse de traces. Cette solution devait être efficace en terme de mise à l'échelle.

Nous avons présenté une solution permettant la lecture et l'analyse parallèle de traces, en tenant compte des contraintes du calcul parallèle efficace. En effet, cette solution devait prendre en compte l'équilibrage de la charge et la résolution des dépendances de données, tout en minimisant le verrouillage et la synchronisation. Nous avons donc exploité les caractéristiques non seulement de l'analyse de traces, mais aussi du format de trace CTF afin créer une solution permettant une haute mise à l'échelle.

Notre solution se base sur la création de charges de travail équilibrées grâce à l'utilisation de l'index de paquets du format CTF. La résolution des dépendances de données se base sur la résolution tardive des dépendances après la lecture de la trace, et est rendue possible grâce au faible impact des dépendances sur l'analyse.

Puisque l'analyse de traces est une tâche nécessitant un grand nombre d'accès au disque, nous avons validé la mise à l'échelle possible de l'analyse parallèle en mettant en place un programme simulant les charges de travail, sur le CPU et le disque, de l'analyse de traces. En mesurant les performances de ce programme avec différents périphériques de stockage, nous avons démontré les possibilités de mise à l'échelle lorsque des périphériques de type SSD sont utilisés. Nous avons aussi démontré que la mise à l'échelle n'est pas gravement affectée, même lorsque le décodage d'événements est plus efficace.

Nous avons implémenté et testé trois types d'analyses parallèles : comptage d'événement, analyse du temps CPU par processus et analyse des données lues et écrites par processus. Les résultats d'efficacité parallèle se sont avérés positifs, avec une efficacité parallèle au-dessus de 56% jusqu'à 32 cœurs, ce qui signifie une accélération de 18 fois par rapport au temps de traitement séquentiel. De plus, les résultats de performance obtenus à partir du programme de simulation confirment que l'efficacité parallèle n'est pas sérieusement affectée par les accès

au disque, lorsque des périphériques de type SSD sont utilisés

L'un des points intéressants de la recherche est lié à l'amélioration des performances des périphériques de stockages qu'amènent les récentes avancées technologiques dans ce domaine. En effet, les disques SSD PCIe offrent des performances de presqu'un ordre de magnitude plus élevées que les disques durs traditionnels. Ce type de performances permet la parallélisation de nouvelles classes de programmes qui autrefois n'auraient pas pu profiter de la parallélisation, à cause du goulot d'étranglement créé par les accès au disque. L'analyse de trace fait justement partie de ce type de programmes.

En résumé, cette recherche a démontré la pertinence de la parallélisation comme solution à l'amélioration de la performance de l'analyse de traces, et qu'elle restera pertinente dans le futur, même si des améliorations sont apportées au décodage de la trace.

6.2 Limitations de la solution proposée

La solution proposée ne fonctionne pour l'instant que pour des analyses en ligne de commande. Afin de supporter les analyses graphiques, il faudrait que les analyses parallèles puissent créer un arbre à historique d'états, qui est utilisé par le logiciel d'analyse graphique Trace Compass. Il faudrait apporter quelques modifications au modèle ainsi qu'à la méthode de construction de l'arbre afin que la solution proposée puisse générer des analyses graphiques.

Certaines analyses peuvent potentiellement ne pas pouvoir être adaptées au modèle proposé : si une analyse ne peut pas se passer des données générées au début de la trace par les événements *lttng_statedump*, par exemple. Il est toujours possible de bloquer jusqu'à ce que ces informations soient disponibles, en les stockant dans une structure globale, mais ces modifications auraient un certain coût en terme de performance.

Finalement, nous supposons que seules les migrations entre les processeurs peuvent occasionner des dépendances entre les flux d'événements. Bien que cette approche soit valable pour la plupart des analyses, il est possible que d'autres analyses, comme l'analyse du chemin critique, aient d'autres genres de dépendances.

6.3 Améliorations futures

Tel que mentionné plus haut, une première amélioration possible serait de rendre possible la génération parallèle d'un arbre d'historique d'état, afin de pouvoir effectuer des analyses graphiques. Comme les intervalles de temps générés par l'analyse parallèle ne sont pas triés (puisque la trace est divisée en plusieurs segments), il serait nécessaire de soit rebalancer

l'arbre d'historique ou modifier notre modèle d'analyse.

Il serait aussi bénéfique d'effectuer une refonte de la bibliothèque de décodage de traces afin de mieux gérer les accès concurrents. Cela permettrait de réduire les copies inutiles de données et les contentions sur les verrous, améliorant ainsi la mise à l'échelle de la solution. De même, le goulot d'étranglement causé par les accès mémoire concurrents pourrait être résolu ou contourné pour des gains intéressants en terme d'efficacité.

Comme le traçage de systèmes distribués est possible, il serait aussi fort intéressant d'adapter notre solution à l'analyse de traces distribuées. Certains aspects abordés dans la présente étude donnent des indices quant à une possible solution distribuée, qui pourrait être incorporée au sein d'un système distribué sous la forme de nœuds dédiés à l'analyse.

Finalement, une amélioration permettant l'analyse parallèle du traçage "en direct" (*live tracing*) permettrait d'améliorer la performance d'analyses de traces à mesure que le système est tracé. Ce dernier aspect s'accorde d'ailleurs avec le précédent, permettant ainsi l'analyse en continu d'un système, à partir de nœuds dédiés, avec une performance maximale.

RÉFÉRENCES

R. Beamonte, F. Giraldeau, et M. Dagenais, “High performance tracing tools for multicore linux hard real-time systems”, dans *Proceedings of the 14th Real-Time Linux Workshop. OSADL*, 2012.

A. Chan, W. Gropp, et E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files”, *Scientific Programming*, vol. 16, no. 2, pp. 155–165, 2008.

A. T. Clements, M. F. Kaashoek, et N. Zeldovich, “Scalable address spaces using rcu balanced trees”, dans *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 199–210.

L. Dagum et R. Menon, “Openmp : an industry standard api for shared-memory programming”, *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

A. C. de Melo, “The new linux’perf’tools”, dans *Slides from Linux Kongress*, 2010.

J. Dean et S. Ghemawat, “Mapreduce : simplified data processing on large clusters”, *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

M. Desnoyers et M. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux”, dans *Proceedings of the Ottawa Linux Symposium*, vol. 2006, 2006.

M. Desnoyers. Common trace format (ctf) specification. En ligne : `http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master`

M. Desnoyers et M. R. Dagenais, “Lockless multi-core high-throughput buffering scheme for kernel tracing”, *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 65–81, 2012.

F. C. Eigler et R. Hat, “Problem solving with systemtap”, dans *Proc. of the Ottawa Linux Symposium.* Citeseer, 2006, pp. 261–268.

P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, “Combined tracing of the kernel and applications with lttng”, dans *Proceedings of the 2009 linux symposium*, 2009.

M. Geimer, F. Wolf, B. J. Wylie, et B. Mohr, “Scalable parallel trace-based performance analysis”, dans *Recent Advances in Parallel Virtual Machine and Message Passing Interface.*

Springer, 2006, pp. 303–312.

M. Geimer, F. Wolf, B. J. N. Wylie, et B. Mohr, “A scalable tool architecture for diagnosing wait states in massively parallel applications”, *Parallel Computing*, vol. 35, no. 7, pp. 375 – 388, 2009. DOI : `http://dx.doi.org/10.1016/j.parco.2009.02.003`. En ligne : `http://www.sciencedirect.com/science/article/pii/S0167819109000398`

F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, et M. Desnoyers, “Recovering system metrics from kernel trace”, dans *Linux Symposium*, vol. 109, 2011.

W. Gropp, E. Lusk, et A. Skjellum, *Using MPI : portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

B. He, W. Fang, Q. Luo, N. K. Govindaraju, et T. Wang, “Mars : a mapreduce framework on graphics processors”, dans *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

W. D. Hillis et G. L. Steele Jr, “Data parallel algorithms”, *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

M. Hiramatsu. (2010) kprobes : Kprobes jump optimization support. En ligne : `http://lwn.net/Articles/375232/`

J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, et V. Prasad, “Ptrace, utrace, uprobes : Lightweight, dynamic tracing of user apps”, dans *Proceedings of the 2007 Linux Symposium*, 2007, pp. 215–224.

R. E. Ladner et M. J. Fischer, “Parallel prefix computation”, *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

G. Matni et M. Dagenais, “Automata-based approach for kernel trace analysis”, dans *Electrical and Computer Engineering, 2009. CCECE’09. Canadian Conference on*. IEEE, 2009, pp. 970–973.

A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, et M. Hiramatsu, “Probing the guts of kprobes”, dans *Linux Symposium*, vol. 6, 2006.

A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data”, dans *Runtime Verification*. Springer, 2013, pp. 219–234.

F. Mueller, “Pthreads library interface”, *Florida State University*, 1993.

T. Mytkowicz, M. Musuvathi, et W. Schulte, “Data-parallel finite-state machines”, dans *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems.* ACM, 2014, pp. 529–542.

E. B. Nightingale, D. Peek, P. M. Chen, et J. Flinn, “Parallelizing security checks on commodity hardware”, dans *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 308–318.

C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, et C. Kozyrakis, “Evaluating mapreduce for multi-core and multiprocessor systems”, dans *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on.* Ieee, 2007, pp. 13–24.

J. Reinders, *Intel threading building blocks : outfitting C++ for multi-core processor parallelism.* " O’Reilly Media, Inc.", 2007.

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

——. (2010) Using the TRACE_EVENT() macro (part 1). En ligne : `http://lwn.net/Articles/379903/`

D. L. Schuff, Y. R. Choe, et V. S. Pai, “Conservative vs. optimistic parallelization of stateful network intrusion detection”, dans *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on.* IEEE, 2008, pp. 32–43.

J. E. Stone, D. Gohara, et G. Shi, “Opencl : A parallel programming standard for heterogeneous computing systems”, *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, et C. Fetzer, “Prospect : A compiler framework for speculative parallelization”, dans *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2010, pp. 131–140.

A. Tumeo, O. Villa, et D. G. Chavarria-Miranda, “Aho-corasick string matching on shared and distributed-memory parallel architectures”, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 3, pp. 436–443, 2012.

G. Vasiliadis, M. Polychronakis, et S. Ioannidis, “Midea : a multi-parallel intrusion detection architecture”, dans *Proceedings of the 18th ACM conference on Computer and communications*

security. ACM, 2011, pp. 297–308.

K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, et S. Narayanasamy, "Doubleplay : Parallelizing sequential logging and replay", *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 3, 2012.

B. Wester, D. Devecsery, P. M. Chen, J. Flinn, et S. Narayanasamy, "Parallelizing Data Race Detection", *SIGPLAN Not.*, vol. 48, no. 4, pp. 27–38, Mars 2013. DOI : 10.1145/2499368.2451120. En ligne : http://doi.acm.org/10.1145/2499368.2451120

F. Wolf et B. Mohr, "Automatic performance analysis of hybrid mpi/openmp applications", *Journal of Systems Architecture*, vol. 49, no. 10, pp. 421–439, 2003.

R. M. Yoo, A. Romano, et C. Kozyrakis, "Phoenix rebirth : Scalable mapreduce on a large-scale shared-memory system", dans *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* IEEE, 2009, pp. 198–207.

C. Zilles et G. Sohi, "Master/slave speculative parallelization", dans *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on.* IEEE, 2002, pp. 85–96.