

Infrastructure pour la gestion générique et optimisée des traces d'exécution pour les systèmes embarqués.

Alexis Martin

▶ **To cite this version:**

Alexis Martin. Infrastructure pour la gestion générique et optimisée des traces d'exécution pour les systèmes embarqués. . Calcul parallèle, distribué et partagé [cs.DC]. Université Grenoble Alpes, 2017. Français. NNT : . tel-01492474v1

HAL Id: tel-01492474

https://hal.science/tel-01492474v1

Submitted on 20 Mar 2017 (v1), last revised 10 Apr 2017 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Alexis MARTIN

Thèse dirigée par **Vania MARANGOZOVA-MARTIN**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**Ecole Doctorale des Mathématiques, des Sciences et Technologies de l'Information et de l'Informatique**

Infrastructure pour la gestion générique et optimisée des traces d'exécution pour les systèmes embarqués.

Thèse soutenue publiquement le **Vendredi 13 Janvier 2017**,
devant le jury composé de :

M. Noel DE PALMA
Professeur, Université Grenoble Alpes, Président
M. Jean-Marc MENAUD
Professeur, Mines de Nantes, Rapporteur
M. Julien IGUCHI-CARTIGNY
Maître de conférences, Université de Lille 1, Rapporteur
Mme Sara BOUCHENAK
Professeur, INSA Lyon, Examinatrice
M. Miguel SANTANA
STMicroelectronics, Examinateur
Mme Vania MARANGOZOVA-MARTIN
Maître de conférences HDR, Université Grenoble Alpes, Directrice de thèse

Table des matières

Table des matières **I**

1 Introduction **1**

1.1 Les défis de l'analyse de traces 2
1.2 Le projet SoC-TRACE 3
1.3 Contributions 4
1.4 Organisation du manuscrit 4

I POSITIONNEMENT **7**

2 État de l'art **9**

2.1 La validation des systèmes embarqués 9
2.1.1 Tests unitaires 10
2.1.2 Cas d'usage 10
2.1.3 Benchmarks 10
2.1.4 Monitoring 11
2.1.5 Débogage 11
2.1.6 Profilage 12
2.1.7 Traces d'exécution 12
2.1.8 Synthèse 13
2.2 Exploitation de traces d'exécution 13
2.2.1 Critères de classification des outils de trace 14
2.2.2 Outils d'exploitation de trace existants 16
2.2.3 Synthèse 19
2.3 Analyse structurée par *workflow* 19
2.3.1 Critères de classification des outils de workflow 21

2.3.2 Outils de workflow existants . . . 23
2.3.3 Synthèse des outils de workflow existants . . . 25
2.4 Synthèse . . . 25

3 Propositions pour l'analyse de traces 29

3.1 Représentation de traces d'exécution . . . 30
3.2 Système de *workflow* . . . 34
3.2.1 Modèle de programmation . . . 35
3.2.2 Modules prédéfinis . . . 37
3.2.3 Modèle d'exécution . . . 38
3.3 Analyse générique d'un système Linux . . . 38
3.3.1 Les métriques observées . . . 38
3.3.2 Collecte de traces génériques . . . 39
3.3.3 Modules d'analyse . . . 42

II VALIDATION 43

4 Analyse de trace par structuration sémantique 45

4.1 Détection d'anomalies dans les traces . . . 45
4.1.1 Anomalies dans une trace . . . 46
4.1.2 Causes d'une anomalie . . . 46
4.2 Distance entre deux séries d'événements . . . 47
4.2.1 Cas d'usage . . . 47
4.2.2 Corrélation visuelle . . . 49
4.2.3 Corrélation par découpe de temps égales . . . 50
4.2.4 Corrélation par découpe de temps irréguliers . . . 51
4.2.5 Algorithmes . . . 51
4.3 Corrélation temporelle : implémentation dans Framesoc . . . 54
4.3.1 Place de l'outil au sein de Framesoc . . . 54
4.3.2 Interface fonctionnelle . . . 55
4.3.3 Interface graphique . . . 56
4.3.4 Évaluation . . . 57
4.4 Conclusion . . . 58

5 Contribution au projet Vistrails 59

5.1 Présentation de VisTrails . . . 59
5.2 VisTrails et l'analyse de traces . . . 62
5.2.1 Définition de modules spécifiques dans VisTrails . . . 62
5.2.2 Analyse utilisant une base de données . . . 63
5.3 Introduction d'un mécanisme de *streaming* . . . 65
5.3.1 Le mécanisme d'exécution de *workflow* dans VisTrails . . . 66
5.3.2 Exécution *multi-thread* de *workflows* . . . 66
5.3.3 Vers une exécution par tâches . . . 67
5.4 Validation du nouveau modèle d'exécution . . . 71
5.4.1 Analyse d'une trace d'exécution en streaming . . . 72

5.4.2 Performances 73
5.5 Conclusion 74

6 SWAT : système de *workflow* pour l'analyse de traces 77

6.1 Implémentation 77
6.1.1 Modules 78
6.1.2 Ports 78
6.1.3 Construction et exécution d'un *workflow* 79
6.2 Analyse de benchmarks de la suite Phoronix 80
6.2.1 Présentation de *Phoronix Test Suite* (PTS) 81
6.2.2 *Workflow* d'analyse de PTS 82
6.2.3 Modules spécialisés du workflow d'analyse 85
6.2.4 Configuration expérimentale 87
6.3 Résultats 88
6.3.1 Résultats pour le poste de travail (x86) 88
6.3.2 Résultats pour la carte Juno (ARM) 94
6.3.3 Validation des performances de SWAT 100
6.3.4 Validation de l'analyse générique de système 100
6.3.5 Retour d'expérience 101
6.4 Conclusion 104

7 Conclusion et perspectives 105

7.1 Objectifs de la thèse 105
7.2 Proposition, réalisation et validation 106
7.3 Perspectives 107

Table des figures 109

Liste des tableaux 111

Bibliographie 113

Chapitre 1

Introduction

A l'origine, les systèmes embarqués étaient définis comme ayant une fonctionnalité principale et utilisant un nombre limité de ressources. Des exemples sont les premières montres à affichage digital ou encore les systèmes d'alarmes de surveillance qui sont contraints par leur puissance de calcul et leur quantité de mémoire.

Depuis quelques années, les systèmes embarqués poursuivent une évolution spectaculaire. Les progrès techniques de miniaturisation, ainsi que la réduction des coûts de production des microcontrôleurs ont permis l'expansion et la complexification des systèmes embarqués. Ceci a eu pour effet de multiplier le nombre de systèmes embarqués mais aussi leur domaines d'application. Nous pouvons le voir avec la généralisation des objets dit "connectés", aussi appelés objets de l'*Internet of Things* (IoT). Parmi ces objets nous pouvons citer les décodeurs vidéo (ou *set-top box*), ordinateurs de bord de voitures, drones, *wearable devices* (montres et vêtements connectés), etc.

Du point de vue de l'architecture, nous observons une convergence entre des systèmes de domaines considérés initialement distincts. En effet, les stations de bureau ont évolué vers des postes mobiles, intégrant des contraintes issues du monde de l'embarqué telles que la consommation énergétique ou les communications réseau. Les systèmes parallèles ont également aujourd'hui de grandes préoccupations énergétiques et se tournent de plus en plus vers des processeurs issus du domaine embarqué. Les systèmes embarqués, quant à eux, ont vu leurs capacités s'étendre et atteindre un niveau d'utilisation aussi large qu'un ordinateur classique. Ils intègrent aujourd'hui une panoplie de processeurs hétérogènes, intégrant des processeurs puissants et des caractéristiques multi-cœur. Le smartphone que l'on connaît aujourd'hui en est l'exemple le plus criant.

Les systèmes embarqués d'aujourd'hui ne se contentent plus d'une fonction unique. Si l'on reprend l'exemple des *set-top boxes*, prévues à l'origine pour décoder un flux TV analogique, il est courant aujourd'hui de voir dans ces décodeurs des fonctions

de stockage et d'accès réseau, du décodage multimédia numérique voir même de jeux vidéos. L'exemple des ordinateurs de bord est lui aussi parlant, ces systèmes gèrent maintenant non seulement les systèmes de freinage (type ABS) mais aussi une partie divertissement (*infotainement*), ainsi que le contrôle de direction grâce à des caméras, accéléromètres, sonars, etc. La diversification de fonctionnalités appelle l'utilisation de plus de ressources, logicielles ou matérielles. Ce besoin accru de ressources est néanmoins toujours accompagné par les contraintes sur le fonctionnement, notamment le mode autonome et la réaction en temps réel. La garantie de ces contraintes, avec la complexification des systèmes embarqués, devient donc un vrai défi.

La construction de systèmes embarqués *corrects* passe par leur validation et évaluation. En effet, il est nécessaire de garantir le respect des spécifications (validation) et de s'assurer de bonnes performances à l'exécution (évaluation de performances). Pour cela, plusieurs méthodes sont disponibles, à différentes étapes du développement des systèmes. Une première approche est de produire des systèmes corrects par construction, en intégrant des contraintes directement dans le système. Une deuxième approche est d'utiliser la simulation, en prenant en compte les différents paramètres des systèmes afin d'évaluer les interactions entre ces différents composants. Enfin, une dernière approche consiste à observer le comportement du système, une fois celui-ci conçu, via des expérimentations en fonctionnement réel.

Traiter tous les problèmes d'exécution et de performances lors de la conception est une méthode coûteuse qui est difficile à appliquer dans beaucoup de cas de systèmes complexes et non critiques. Le problème est similaire pour une validation par simulation qui dépend fortement du niveau d'abstraction et des modèles utilisés. Toute validation sur un modèle qui n'est pas représentatif du système réel ne permet aucune garantie. La méthode majoritairement utilisée aujourd'hui reste l'observation du système final en exécution réelle. La méthode capture typiquement les *événements* se produisant dans le système afin de produire une *trace d'exécution*. Cette trace d'exécution fournit un historique de l'exécution qui est utilisée afin de comprendre et d'analyser le comportement du système. Toutefois, l'analyse de traces fait face à plusieurs défis majeurs.

1.1 Les défis de l'analyse de traces

L'utilisation de traces d'exécution pour l'analyse de systèmes n'est pas sans difficultés [53].

Avec l'évolution des systèmes embarqués et le complexification de leurs parties matérielles et logicielles, les informations que les analystes voudraient capturer deviennent elles aussi complexes et variées. Les outils de traces doivent, en général, fournir un maximum d'informations sur le système tracé tout en le perturbant le moins possible. En effet, comme toute mesure physique, une observation induit forcément une perturbation du phénomène observé. Il est donc nécessaire de mesurer l'impact de cette observation afin d'être sûr que ce que l'on observe comme résultat n'est pas seulement dû à l'observation elle-même.

Pour fournir des outils de traces efficaces, les producteurs de systèmes embarqués proposent des outils *ad-hoc*, optimisés pour des plates-formes particulières. Ceci vient en contradiction avec leurs besoins d'outils réutilisables et maintenables, ainsi que de temps de production et coût limités.

Les informations capturées dans une trace de système embarqué sont souvent de bas niveau. En d'autres termes, ce sont des informations qui ne reflètent pas directement le comportement global du système, mais qui montrent des parties matérielles ou logicielles en particulier. D'une part, ces traces deviennent facilement volumineuses : plusieurs giga-octets de données peuvent être générés en seulement quelques secondes d'exécution. Se pose alors des problèmes de stockage, de manipulation et de performance d'analyse de ces traces de grande taille. D'autre part, les informations bas niveau sont difficiles à appréhender par un analyste qui a besoin de vue plus globale, à un niveau d'abstraction plus élevé. Il existe, en effet, un fort besoin en termes d'outils qui permettent de traiter de grands volumes de données et d'extraire des informations sémantiques à présenter aux concepteurs de systèmes.

Il existe de nombreux outils effectuant différentes analyses sur les traces d'exécution. Cependant, l'analyse d'une trace d'exécution volumineuse suppose souvent différentes étapes avec différents traitements qui sont rarement tous proposés par le même outil. Les analystes se trouvent obligés soit de limiter leurs analyses, soit d'utiliser plusieurs outils, soit d'écrire des scripts spécifiques. Aucune de ces possibilités n'est satisfaisante. En effet, limiter l'analyse pourrait faire manquer la détection de phénomènes intéressants ou à problèmes. Écrire son propre outil de manipulation de données demande une expertise technique avancée et permet rarement de réutiliser les traitements. Enfin, utiliser plusieurs outils est difficile de par leur diversité, la hétérogénéité des formats utilisés et encore, l'expertise technique nécessaire pour une utilisation pertinente.

1.2 Le projet SoC-TRACE

Ce travail de thèse s'inscrit dans le cadre du projet SoC-TRACE [13, 14, 56], faisant partie du programme Fond Unique Interministériel (FUI). Ce programme a pour but d'aider le développement de projets R&D collaboratifs, associant de grandes entreprises, des PME et des instituts de recherche. Dans le cadre de SoC-TRACE, les acteurs sont STMicroelectronics, ProBayes et Magillem Design Services, pour la partie industrielle ; ainsi que l'Université Joseph Fourier et Inria Grenoble pour la partie académique. Le projet SoC-TRACE a pour objectif de faire avancer l'état de l'art concernant l'exploitation de traces d'exécution dans le domaine des systèmes embarqués. Le projet s'intéresse au développement de nouvelles méthodes d'analyse de traces, d'une part, et à la conception d'infrastructure logicielle standardisant les traitements de stockage, d'analyse et de visualisation de traces, d'autre part.

1.3 Contributions

Nos contributions sont les suivantes :

Un modèle enrichi des traces d'exécution.
Nous proposons un modèle de traces d'exécution générique et enrichi en sémantique. La généricité du modèle nous permet de définir des traitements d'abalyse de traces qui peuvent s'appliquer dans differents contextes et sur des traces provenant de différents systèmes. L'enrichissement de la trace permet d'y intégrer des notions sémantiques liées au domaine d'application. Nous capturons ces notions à l'aide d'*événements ponctuels*, des *états du système*, des *liens entre deux entités* du système ou des *valeurs numériques*. La structuration de la trace à l'aide de ces quatre types d'événements est faite explicitement dès le départ dans nos outils, alors qu'elle est implicite, calculée à la demande ou non présente dans les outils existant. Cet enrichissement de la trace est nécessaire afin de fournir des traitements d'analyse utiles de plus haut niveau.

Système de *workflow* pour l'analyse de traces.
Notre deuxième proposition consiste à utiliser les principes des systèmes de *workflow* pour la gestion de traces. Nous définissons des briques de base d'analyse de traces que nous pouvons interconnecter pour former une analyse complexe. Les connexions entre les briques d'analyse définissent des canaux d'échange de données. Pour ces échanges nous utilisons des flux continus permettant de supporter le traitement de gros volumes de données. L'analyse est représentée par un *workflow* qui de fait décrit un plan d'analyse, qui peut être facilement compris, partagé, et réutilisé. L'analyse est construite de manière itérative ce qui permet de faire évoluer l'analyse en fonction des résultats intermédiaires. Les propriétés du *workflow* permettent une exécution déterministe ce qui est nécessaire pour reproduire et comprendre l'analyse et les résultats obtenus. Enfin, l'analyse peut être réutilisée dasn des contextes différents et avec des traces différentes.

Analyse générique des performances des systèmes.
Notre troisième proposition porte sur l'évaluation de performances d'exécution dans un environnement Linux. Nous choisissons des métriques clés et montrons comment tracer et analyser une application afin de les obtenir. Notre méthode produit, sans connaissances *a priori* et sans modifier les application, des profils de performances unifiés permettant de les comparer. Le profil reflète l'utilisation des CPU, l'utilisation de la mémoire et l'activité du noyau Linux. Nous instancions la méthode à travers un workflow d'analyse utilisant notre infrastructure. Nous l'appliquons de manière approfondie sur la suite de benchmarks Phoronix.

1.4 Organisation du manuscrit

La thèse est organisée en deux parties. Nous présentons dans un premier temps notre positionnement. Le Chapitre 2 introduit l'état de l'art sur les méthodes et les

outils d'analyse, ainsi que les méthodes de workflow. Le Chapitre 3 détaille nos propositions sur le modèle enrichi de traces, le système de workflow, ainsi que l'analyse générique de performances systèmes. Dans la seconde partie, nous présentons la validation de notre proposition. Le Chapitre 4 présente l'application de notre modèle enrichi de traces. Le Chapitre 5 expose l'implémentation de notre modèle de système de workflow au sein du projet VisTrails. Le Chapitre 6 discute de l'implémentation et valide notre modèle d'analyse générique des performances. Le Chapitre 7 conclut et ouvre les perspectives de nos travaux.

Première partie
POSITIONNEMENT

Dans cette première partie nous dédions un chapitre (Chapitre 2) à l'état de l'art sur les outils et méthodes de gestion de traces. En particulier, nous discutons les caractéristiques et les limites des outils de trace existant. Nous présentons également le domaine des système de workflow qui présentent des aspects très intéressants pour la gestion de traces.

Le deuxième chapitre de cette partie définit nos objectifs et détaille notre proposition. Nous présentons nos trois contributions portant sur la modélisation des traces d'exécution, sur un système de workflow générique pour l'analyse de traces et sur une méthode générique d'analyse des performances d'un système.

Chapitre 2

État de l'art

Dans ce chapitre, nous présentons les travaux relatifs à l'exploitation de traces des systèmes embarqués. Nous parlons dans un premier temps des défis de validation de systèmes et de l'importance croissante des traces d'exécution. Nous dressons ensuite un état de l'art des outils d'exploitation de traces existant. Comme ces outils manquent des aspects d'enchaînement d'analyses et d'automatisation, nous regardons comment ces aspects sont gérés par les approches de workflow.

2.1 La validation des systèmes embarqués

La validation de tout système informatique, y compris les systèmes embarqués, est une phase critique du développement. En effet, sans validation, aucune garantie de fonctionnement n'est assurée. La validation des systèmes porte sur deux aspects : l'aspect fonctionnel, et l'aspect performances. La validation fonctionnelle permet de garantir que le système s'exécute selon ses spécifications. La validation des performances porte typiquement sur l'utilisation de ressources à l'exécution.

La validation d'un système peut être faite en utilisant des méthodes formelles [22, 25] qui garantissent la production de code correct. Habituellement, ce type de validation est destiné aux systèmes dits critiques, comme par exemple dans l'aérospatiale ou l'automobile. Néanmoins, la complexité de la plupart des systèmes rendent les méthodes formelles difficilement applicables. Par conséquent, dans la plupart des cas, la validation des systèmes se fait pendant et après le développement, en testant les fonctionnalités déjà développées. Dans la suite, nous présentons brièvement les méthodes concernées.

2.1.1 Tests unitaires

Un test unitaire permet de vérifier le fonctionnement d'une partie précise d'un système. L'idée est de valider les fonctionnalités à grain fin, comme par exemple considérer directement une fonction ou un objet. La production de tests unitaires est devenue une phase obligatoire du processus de développement.

Le développement et l'exécution des tests unitaires accompagnent toute la phase de développement du système. Ils permettent ainsi de mettre en évidence les problèmes d'implémentation le plus tôt possible.

Idéalement, le test unitaire doit être exhaustif sur les données testées. Cependant, ces tests étant écrits par le développeur, ils ne peuvent tester que ce à quoi le développeur a pensé. Certains problèmes peuvent donc passer sous silence pendant ces tests, et se manifester ensuite.

2.1.2 Cas d'usage

Un scénario, ou cas d'usage, décrit une séquence correcte d'actions et interactions lors d'une utilisation réelle du système. À l'inverse des tests unitaires testant des composantes précises du système, les cas d'usage testent le système de manière plus macroscopique.

Ces cas d'usage définissent une utilisation particulière du système, permettant sa validation dans le cadre d'une utilisation réelle.

De manière analogue aux tests unitaires, la validation par scénario ne permet de valider que les cas d'utilisations qui ont été prévus.

2.1.3 Benchmarks

Un benchmark est un programme dédié au test d'un système. Son but est de collecter, d'un point de vue quantitatif, des mesures de performances sur des aspects clés du système testé. Il définit un processus de test standard, reconnu de ses utilisateurs, permettant de comparer les systèmes les uns aux autres en fonction des résultats obtenus. Un benchmark peut aussi servir à tester les limites d'un système, ou sa robustesse, en mettant le système sous pression (*stress*).

On peut distinguer deux grandes catégories de benchmarks. Les *micro-benchmarks*, testant une petite partie précise d'un système tel qu'une routine du noyau ou une fonction d'un programme. Il est courant d'utiliser des micro-benchmarks lors d'un développement logiciel, afin d'évaluer certaines parties critiques du code. Les *macro-benchmarks*, testant un système de manière plus complète, en prenant en compte tout les composants de ce système. Un benchmark teste donc de manière exhaustive différents aspects du fonctionnement d'un système, d'une partie spécifique ou de son ensemble.

Il existe des programmes de benchmark connus, qui se sont imposés comme standards *de facto* dans certains domaines. Ils sont donc utilisés pour situer un système par rapport à un autre en fonction des performances testées par le benchmark. Ils

permettent aussi d'évaluer le gain, ou la perte, de performance lorsque des modifications sont apportées à un système. Cependant, un benchmark ne reflète en général que des performances "brutes" du système. Les performances lors d'une utilisation réelle seront moindres.

2.1.4 Monitoring

Le monitoring désigne la surveillance de l'activité d'un système. On parle aussi de supervision ou de *logging* [87, 88].

Le monitoring est utilisé pour surveiller un système en fonctionnement. Il permet d'obtenir des informations en suivant l'évolution de l'exécution. Ces informations peuvent être quantitatives, comme par exemple des métriques de performances, ou qualitatives, pour le cas d'un journal système enregistrant les activités des composants systèmes.

Le monitoring permet de mettre en place des mécanismes de réaction en fonction des événements observés. Il est ainsi possible de créer des alertes en temps réel, par exemple l'envoi d'un mail dans le cas d'un dysfonctionnement. Il permet aussi d'auto modifier le système, par exemple l'activation d'un processeur en plus lors d'une charge de travail trop importante. Le mécanisme de monitoring est cependant intrusif pour le système car il nécessite de s'exécuter en parallèle sur le système, pouvant altérer ses performances.

2.1.5 Débogage

Le débogage consiste à observer un programme en cours d'exécution. Contrairement au monitoring, l'approche typique des méthodes de débogage consiste à entrelacer les phases d'exécution et celles d'analyse. Les outils de débogage permettent de définir des points d'arrêts dans le programme (*breakpoint*). Lors de l'exécution, le programme se met en pause à la rencontre d'un point d'arrêt, laissant la main au développeur pour la partie analyse.

Le débogage peut être fait de manière sommaire, par exemple en affichant des valeurs de variables à certains instants du programme (`printf, printk`). Cette méthode sommaire peut ne pas suffire dans le cas d'un programme impliquant un code de taille conséquente. Le débogage plus avancé se fait en utilisant des outils plus complets, typiquement l'outil GDB pour un code C/C++. Il permet de regarder les valeurs associées à certaines variables, voire même d'en modifier leurs valeurs, ou encore forcer l'exécution de certaines fonctions. Le développeur peut aussi contrôler les fils d'exécution d'un programme.

Afin de pouvoir contrôler l'exécution des programmes, les outils de débogage tels que GDB émulent certaines parties du système. Ces émulations permettent aux développeurs de contrôler ces parties du système, par exemple la mémoire ou l'ordonnancement. Par ailleurs, l'émulation biaise l'exécution du programme et le comportement analysé peut ne pas être représentatif d'une exécution réelle du programme. En effet, les méthodes de débogage biaisent le déroulement naturel d'un

système, notamment à cause des points d'arrêts. Certains problèmes peuvent donc ne jamais se manifester lors de cette phase de débogage.

2.1.6 Profilage

Le profilage consiste à observer une exécution d'un système afin de produire des données statistiques sur les fonctions de ce système. Les mesures typiques observées sont le nombre d'appel par fonctions, le temps passé à exécuter celles-ci, ou encore la répartition de ces appels dans le temps. Ces mesures sont obtenues généralement en observant la pile d'exécution du programme. Le profilage peut se faire de deux manières selon que le programme observé soit compilé ou interprété.

Pour les programmes interprétés, s'exécutant dans un environnement d'exécution virtuel, tel que Python ou Java, les données sont collectées directement via cet environnement. En effet, ces environnements d'exécution possèdent des informations du fait qu'ils soient en charge de l'exécution, et permettent d'accéder à ces données. Typiquement, l'outil JvisualVM permet d'observer l'exécution d'un code Java, tel que le nombre de classes instanciées, l'utilisation de la mémoire, des threads et du temps CPU utilisé. Pour les programmes compilés, type C/C++, les informations d'exécution ne sont pas disponibles directement. Il est cependant possible d'utiliser des outils comme gprof, qui permet d'insérer du code supplémentaire au moment de la compilation, pour obtenir ces informations.

Le profilage est utilisé principalement en phase de tests et n'a pas pour but d'être utilisé lors de l'exécution réelle du système, contrairement au monitoring. Généralement, les fonctions permettant le profilage sont désactivées une fois les phases de tests finies. Le profilage est utile pour connaître des informations globales sur un système, cependant les données récoltées sont souvent agrégées, sous forme de statistiques. Pour les programmes compilés, on utilise parfois du *sampling*, c'est-à-dire une observation par échantillonnage, afin d'obtenir les informations à intervalles réguliers durant l'exécution. Le *sampling* permet d'enregistrer les informations, tout en limitant l'impact sur le programme observé. Cependant, les résultats ainsi obtenus ne sont pas exacts mais expriment une approximation statistique de l'échantillon observé.

2.1.7 Traces d'exécution

Une trace d'exécution est un enregistrement des événements qui se sont produits sur le système. Elle représente un processus évolutif dans le temps. Les événements qui composent la trace possèdent une information temporelle (*timestamp*) permettant de situer chaque événement dans le temps. Il est possible que les événements ne contiennent aucune information temporelle, c'est alors l'ordre de ces événements qui est important et qui définit une relation de précédence entre ceux-ci.

Il est possible de tracer des événements *macroscopiques* d'un système, tels que le login/logout de chaque utilisateur ou des événements *microscopiques* comme, par exemple, tous les accès disques de l'ensemble du système. De plus, une trace peut

contenir d'autres d'informations en plus du *timestamp*, par exemple un type d'événement, la partie du système responsable de cet événement, des détails sur l'événement, etc.

Plus la trace contient d'événements, plus l'exploitation de celle-ci pourra être poussée car elle contient plus d'informations. Cependant, il faut faire un compromis entre la quantité de données capturées et la perturbation du système. En effet, le fait de tracer un système est intrusif et plus on capture de données, plus cette intrusion est grande.

2.1.8 Synthèse

L'utilisation de cas d'usages et de tests unitaires pour la validation n'est pas suffisante. En effet, ces méthodes permettent de tester les systèmes seulement dans les configurations considérées. L'oubli d'un cas limite ou d'une manière d'utilisation peuvent quand même permettre de valider le système car les tests n'ont pas été exhaustifs.

Les méthodes de benchmarks et de débogage ne permettent pas d'avoir une validation sur l'exécution réelle d'un système. Les benchmarks permettent seulement d'obtenir des performances "brutes" ou des limites maximums. Le débogage biaise le fil d'exécution et donc les interactions entre les différentes parties du système observé. Ces méthodes ne sont donc pas applicables dans le cas où les métriques considérées sont les performances du système en utilisation réelle.

Le monitoring et le profilage proposent la plupart du temps des résultats statistiques calculés sur des échantillonnages. Il est alors possible que certains problèmes passent sous silence et ne soient pas visibles dans les résultats. Ces méthodes ne permettent donc pas toujours une observation et une validation du système de manière complète.

L'utilisation des traces d'exécution pour la validation est donc un choix pertinent pour permettre une analyse complète d'un système reflétant une utilisation réelle de celui-ci.

2.2 Exploitation de traces d'exécution

L'exploitation des traces d'exécution fait référence aux différents processus opérés jusqu'à l'obtention des résultats finaux. Nous pouvons citer quatre grandes phases indépendantes qui constituent l'exploitation de traces :

- La capture, faisant référence aux mécanismes d'enregistrement d'une trace d'exécution ;
- Le stockage, qui correspond à la manière dont est stockée et manipulée la trace ;
- L'analyse, qui représente la transformation des informations contenues dans la trace en résultat d'analyse, typiquement des métriques de performance ;
- La visualisation des résultats, c'est-à-dire les différentes manières et méthodes de présenter les résultats d'analyse.

Beaucoup d'outils d'exploitation de traces sont disponibles, certain se focalisent seulement sur un sous-ensemble des quatre phases de l'exploitation. Nous en avons sélectionné une partie représentative de ce qu'il se fait, puis les avons classés selon des critères relatifs à ces phases.

2.2.1 Critères de classification des outils de trace

Nous présentons dans un premier temps les différents critères, caractérisant les outils d'exploitation de trace. Ces critères permettent d'identifier le comportement et les fonctionnalités des différents outils selon les phases de l'exploitation de trace.

Domaine d'application

Un outil d'analyse ne devrait pas être trop spécialisé dans un domaine en particulier. En effet, les outils spécialisés dans un domaine d'application auront l'avantage de proposer, par exemple, des analyses ou visualisations adaptées au domaine auquel ils se destinent. Cependant, ils ne seront pas utilisables dans un autre domaine du fait de cette spécialisation. Dans notre état de l'art, nous nous sommes intéressés aux outils système, HPC, et embarqués.

Capture de la trace (Capture)

Un outil d'analyse permettant la phase de capture de la trace possède une simplicité d'utilisation du fait de l'interaction forte entre la phase de traçage et les autres phases. Cela évite aussi des actions supplémentaires type exportation puis importation. Certains outils proposent cette fonctionnalité, mais d'autres laisse cette phase à des outils plus spécialisés.

Contenu de la trace (Capture)

La trace capturée doit être la plus complète possible. On peut distinguer les outils de capture par échantillonnage et ceux par capture totale des événements. La capture de trace complète permet, par la suite, une analyse représentative de l'application tracée. La capture par échantillonnage ne permettra qu'une vue partielle et une approximation statistique. La capture par échantillonnage combine en fait la phase de capture mais aussi un morceau de la phase d'analyse, en proposant de capturer des informations déjà pré-traitées sommairement.

Format de trace utilisé (Stockage)

Le format de trace désigne la façon dont est représenté les données de la trace. Il devrait être compatible avec le plus d'outils possibles. En effet, les différentes phases étant indépendantes et les outils se concentrant généralement sur un sous-ensemble de celles-ci, on veut pouvoir utiliser, par exemple, n'importe quel outil de visualisation quel que soit l'outil de capture utilisé. Il est typiquement possible d'enregistrer les données de la trace dans un fichier binaire, une base de données, ou encore un simple fichier texte. Il existe différents formats de trace orientés pour des besoins spécifiques [21]. Dans tous les cas, l'aspect important est le standard

utilisé. En effet, un format commun à plusieurs outils permet d'utiliser la même trace entre ces outils. A l'inverse, un format de fichier interne et propriétaire contraint l'utilisation des seuls outils pouvant lire ce format.

Type d'analyses (Analyse)

Les traitements faits sur les traces doivent apporter de l'information, ou sémantique, en plus lors des analyses. Un exemple d'analyse apportant de la sémantique est typiquement une opération de détection de motifs. On cherche dans la trace une séquence d'événements particuliers, qui représente une information de plus haut niveau. Par exemple, pour le cas d'une trace sur les manipulations de fichiers, l'ouverture, la modification, puis la sauvegarde, correspond à la modification du fichier considéré. La plupart du temps les outils se contentent cependant de proposer des analyses simples, dites directes, sans ajout de sémantique. Ces analyses sont typiquement des filtres, agrégations, calculs statistiques, etc.

Manipulation de différentes traces (Analyse)

La manipulation de plusieurs traces de manière simultanée permet par exemple de pouvoir comparer les résultats et comportements d'une même exécution. Cela permet aussi, par exemple, de combiner des résultats provenant des deux différentes sources de données. Ne travailler avec qu'une seule trace à la fois peut donc s'avérer limitatif pour certaines analyses.

Automatisation de l'analyse (Analyse)

L'automatisation de l'analyse est la possibilité de créer une séquence de plusieurs analyses, et de l'exécuter sans se préoccuper du lancement de ces différentes analyses. Cette automatisation n'est généralement pas possible car la plupart des outils se présentent sous la forme d'outils graphiques nécessitant des interactions manuelles (sélection, zoom, etc.), à la sourie ou au clavier par exemple. L'automatisation implique que l'outil propose des mécanismes pour enchaîner différentes analyses, c'est-à-dire la réutilisation d'un résultat comme donnée d'entrée d'une seconde analyse. Un enchaînement d'analyse classique est le filtrage des éléments d'une trace, suivie d'une analyse statistique sur le sous-ensemble ainsi filtré.

Visualisation des données (Visualisation)

La visualisation de données est importante pour comprendre les résultats calculés par les outils. Des visualisations avancées permettent de mieux appréhender les différentes valeurs obtenues [34]. Les outils proposant des visualisations avancées permettent typiquement de visualiser les résultats sous forme de graphiques, courbes, gantt, ou histogrammes. L'outil Score-P par exemple propose des visualisations 3D représentant des communications, avec un code couleur symbolisant l'intensité d'utilisation. Certains outils ne proposent par contre que des visualisations brutes de la trace, sous forme de tables, comme par exemple Ftrace.

Extensibilité

L'aspect évolutif des outils est important, en effet, il est intéressant de pouvoir modifier l'outil ou d'apporter des méthodes d'analyses supplémentaires. Ces extensions peuvent se présenter sous forme d'API ou de modules. Ces ajouts, correspondant à des besoins plus spécifiques, permettent une analyse plus avancée, en ne se limitant pas seulement aux bornes imposées par l'outil utilisé.

Traitement de traces de taille conséquente

Les outils doivent pouvoir produire les résultats d'analyse dans un temps raisonnable. Lorsque les traces atteignent de grande quantité d'événements, le temps nécessaire à l'analyse sera augmenté si les outils ne tiennent pas en compte ce paramètre. Les outils issus du HPC ont su prendre en compte cette contrainte afin de produire des outils efficaces, même lorsque les traces sont de taille conséquente. L'outil Tracecompass a lui aussi su intégrer des mécanismes pour traiter les traces système efficacement, car ces traces ont une densité d'événement très élevée.

2.2.2 Outils d'exploitation de trace existants

Nous avons considéré plusieurs outils d'analyse, parmi les plus répandus. Dans la suite, ils sont énumérés par ordre alphabétique. Pour chaque outil, nous présentons une description ainsi que leur classification dans les critères cités précédemment.

Nous avons aussi bien étudié les outils d'analyses de traces que certains outils de profilage. En effet, si de prime abord ces outils ne permettent pas l'importation d'une trace pour son analyse, la plupart des outils de profilage utilisent des traces de manière interne.

Ftrace [5]

Ftrace est un outil simple permettant d'analyser un système en utilisant les mécanismes du noyau. Ftrace intègre le mécanisme de traçage et d'analyse, ce qui en facilite son utilisation. Il permet d'observer les appels système, les fonctions de traitement d'interruption, les fonctions d'ordonnancement et les piles réseau.

LTTng [33, 32]

LTTng est un outil de capture de trace système utilisant un format de trace ouvert et performant pour l'écriture de données intensives, le format CTF [73]. Il utilise les interfaces du noyau Linux afin d'enregistrer les événements système. Il permet aussi de tracer les programmes s'exécutant en espace utilisateur en implémentant ce code. LTTng propose aussi des scripts offrant la possibilité d'enregistrer la trace de manière automatique puis d'analyser celle-ci. Un des scripts permet par exemple de calculer sur la trace des statistiques sur les appels système. Ces statistiques concernent le nombre d'appels, le temps moyen, minimum et maximum de leur exécution. Ces scriptes présentent les résultats sous forme textuelle.

Pajé [76, 48]

Pajé est un outil de visualisation de traces permettant une vision riche en sémantique d'une exécution d'un programme parallèle. Il permet typiquement d'observer le comportement d'un programme MPI. Pour proposer cette visualisation riche en sémantique, l'outil utilise un format de trace éponyme. Ce format permet d'ajouter de la sémantique à la trace, pouvant être utilisé par la suite pour la visualisation.

Paraver [9]

Paraver est un outil de visualisation spécialisé dans l'analyse de programmes utilisant les interfaces MPI, OpenMP, pthreads, OmpSs ou CUDA. Il peut utiliser une trace enregistrée depuis un autre outil, ou alors tracer lui-même l'application observée. Il permet de travailler sur la trace à plusieurs niveau (application, tâche, thread). Il peut aussi travailler avec plusieurs traces à des fins de comparaisons.

Perf [10]

Perf est à la base un outil de profilage s'intégrant au noyau Linux et permettant d'en observer le comportement. Il permet l'enregistrement d'une trace, mais aussi son analyse. Il permet d'observer les points de traçage disponibles dans le noyau et les compteurs de performances.

Scalasca [40]

Scalasca est un outil de profilage et d'analyse de traces. Il cible les applications parallèles utilisant les modèles de programmation OpenMP et MPI. En s'appuyant sur les modèles de programmation et les interfaces bien connus d'OpenMP [29] et de MPI [39], l'outil intègre des aspects de calcul de statistiques, de filtrage, d'agrégation et de reconnaissance de motifs. La structure des traces reflète la structure en processus et threads de l'application. La reconnaissance de motifs porte majoritairement sur les schémas de communication et de synchronisation correspondant à des problèmes connus par la communauté HPC.

ScoreP [23]

La complexité d'analyse des performances des systèmes HPC et la prolifération d'outils de traçage, de profilage et de visualisation sont au coeur des motivations du projet Score-P. Score-P vise la définition d'une infrastructure qui permet l'intégration et la collaboration entre outils existants. Dans ce sens, Score-P se rapproche aux motivations du projet SoC-TRACE. Score-P intègre plusieurs outils HPC déjà largement connus : Periscope, Scalasca, Vampir et Tau. Score-P travaille avec le format OTF2 [50, 51] pour le traçage et le format CUBE4 [72] pour les profiles.

Strace [15]

Strace est un outil permettant l'analyse des appels système. Il utilise l'interface permettant de capturer les interactions des programmes avec le noyau Linux via les

appels système. Strace enregistre une trace au format ASCII sans sémantique et ne propose que des analyse basiques, type statistique, sur celle-ci.

STWorkbench [16]

STWorkbench est un outil complet de traçage, d'analyse et de visualisation. L'outil KPTrace [65] s'adresse au logiciel embarqué sur les systèmes d'exploitation ARM et les distributions STLinux. L'outil active dynamiquement les points de traçage qui concernent principalement des unités de calcul, des flots d'exécution et des interruptions. L'outil peut également observer les symboles du noyau (interruptions, appels système, changements de contexte, etc.), les opérations sur la mémoire (allocation et libération), etc. La visualisation des traces et des profils se fait dans des vues Eclipse qui incluent surtout des diagrammes de Gantt et des vues tabulaires.

TAU [71]

Ciblant l'évaluation de performances d'applications parallèles à travers de profils et de traces, TAU met un accent fort sur les mécanismes efficaces d'instrumentation et de mesure de systèmes. En mettant en place des mécanismes d'instrumentation à la source, au niveau pré-processeur ou compilation, TAU donne accès à un ensemble riche et configurable d'informations. En font partie les événements ponctuels de début ou de fin d'appels, des changements de valeurs, des informations d'intervalle (calcul d'état), des statistiques sur le temps d'exécution de régions de code, etc. TAU fournit une infrastructure complète de stockage et d'analyse de profils. Les informations communes contenues dans les profils sont représentées en utilisant un format commun, sont stockées dans une base de données et peuvent être manipulées à travers une interface d'accès. Le but étant de pouvoir brancher différentes analyses (fouille de données, analyses statistiques, etc.) sur les profils.

Tracecompass [17]

Tracecompass est un outil d'analyse graphique de traces d'exécution. Il permet de lire des traces au format CTF, produites par LTTng par exemple, afin de les analyser. Les analyses qu'il propose sont des analyses statistiques sur les événements de la trace. Si la trace contient les informations nécessaires, il permet de visualiser via un diagramme de Gantt l'utilisation du matériel, ainsi qu'une vue sur les états des processus exécutés. L'outil propose des mécanismes de synchronisation de vue, par exemple lors d'une sélection dans la table des événements, les vues statistiques relatives s'adaptent en montrant les résultats sur cette sélection seulement. Tracecompass offre la possibilité de travailler sur plusieurs traces en même temps. Il permet de mettre en lien visuellement deux traces dans son interface en affichant des vues similaires sur des données différentes. Cependant, il n'est pas possible de faire interagir l'outil sur les deux traces en même temps. Tracecompass permet l'extension de ses fonctions via des plugins, pour des analyses ou l'importation de traces dans d'autres formats que ceux nativement supportés.

Vampir [60, 52]

Vampir est un outil d'analyse de programmes OpenMP et/ou MPI qui se compose de deux parties. VampirTrace pour le profilage et la collecte de traces. Vampir pour la visualisation. VampirTrace permet une instrumentation à différents niveaux et capture des informations standard incluant les graphes d'appels, les historiques d'exécution, les compteurs matériels, l'usage de la mémoire et les E/S. Vampir fournit des vues qui peuvent être considérées standard : vue temporelle, vue de synthèse, vue sur les communications, vue statistiques, etc. Le point qui distingue Vampir des autres outils dans le domaine HPC est son objectif explicite de supporter l'échelle de très grands systèmes parallèles.

VTune Amplifier [6]

VTune Amplifier est un outil de profilage et d'analyse proposé par Intel. L'outil propose des fonctionnalités d'analyse du système concernant l'échantillonnage statistique de l'utilisation des processeurs, calcul des occurrences d'appels de fonctions, graphe d'appels, calcul des états d'attente, etc. L'outil utilise les spécificités des processeurs Intel afin de donner des informations sur le mapping des threads sur les processeurs. Il peut accéder aux compteur matériels spécifiques et donner des informations sur la consommation énergétique.

2.2.3 Synthèse

Il existe de nombreux outils qui adressent différentes phases de l'exploitation de traces. Nous pouvons dire que la phase de capture est bien maîtrisée par les différents outils. Toutefois, en ce qui concerne le stockage, les outils utilisent de nombreux formats spécifiques et pas forcément compatibles. Par conséquent, ils sont utilisés de manière indépendante et demandent aux analystes de traces un temps d'apprentissage conséquent. La phase d'analyse reste critique et doit, en plus des analyses simples déjà proposées par une majorité d'outils, inventer des analyses plus complexes, utilisant la sémantique du système analysé et manipulant une grande quantité de données. La phase de visualisation est déjà bien considérée dans de nombreux outils mais est également face aux défis de grandes données et de diversification des domaines nécessitant des outils. De manière générale, les outils restent confinés dans un domaine d'application et ne proposent qu'un ensemble limité de traitements, difficilement personnalisable et automatisable.

La Table 2.1 représente le positionnement des outils d'exploitation cités par rapport aux critères de classification choisis.

2.3 Analyse structurée par *workflow*

Un *workflow* représente les étapes successives d'un processus. Par exemple, les systèmes de workflow sont largement utilisés pour gérer les procédures administratives et les échanges au sein de l'entreprise [41, 77]. Une représentation typique d'un workflow est un graphe orienté acyclique. Les nœuds du graphe représentent des

	Domaine	Capture	Contenu	Format	Analyses	Traces mult.	Automatisation	Visu.	Extensible	Taille
Ftrace	système	oui	complet	spécifique	directes	non	enchaînement	brute	non	oui
LTTng	système	oui	complet	compatible	complexes	non	enchaînement	brute	oui	oui
Pajé	HPC	non	complet	compatible	complexes	non	non	avancée	oui	oui
Paraver	HPC	oui	complet	compatible	complexes	non	non	avancée	oui	oui
Perf	système	oui	complet	spécifique	directes	non	enchaînement	brute	non	oui
Scalasca	HPC	non	complet	compatible	complexes	non	non	avancée	non	oui
ScoreP	HPC	oui	complet	compatible	complexes	non	non	non	oui	oui
Strace	système	oui	complet	spécifique	directes	non	enchaînement	non	non	oui
STWorkbench	embarqué	oui	complet	spécifique	directes	partiel	non	avancée	non	oui
TAU	HPC	non	complet	compatible	complexes	non	non	avancée	non	oui
Tracecompass	Système	non	complet	compatible	complexes	oui	enchaînement	avancée	non	oui
Vampir	HPC	non	complet	compatible	complexes	partiel	enchaînement	brute	non	oui
VTune	système	oui	partiel	spécifique	complexes	partiel	enchaînement	avancée	non	oui

Table 2.1 – Synthèse des outils d'exploitation de traces.

opérations à effectuer et les liens entre ces noeuds modélisent les dépendances entre opérations.

La Figure 2.1 montre un exemple de workflow. Il décrit le processus d'évaluation des papiers soumis à une conférence. Les liens entres les étapes indiquent qu'une tâche ne peut commencer avant que la précédente soit terminée : dans cet exemple, les liens indiquent une *dépendance de tâches*. Dans ce cas présent, on parle de flux de contrôle (*control-flow*).

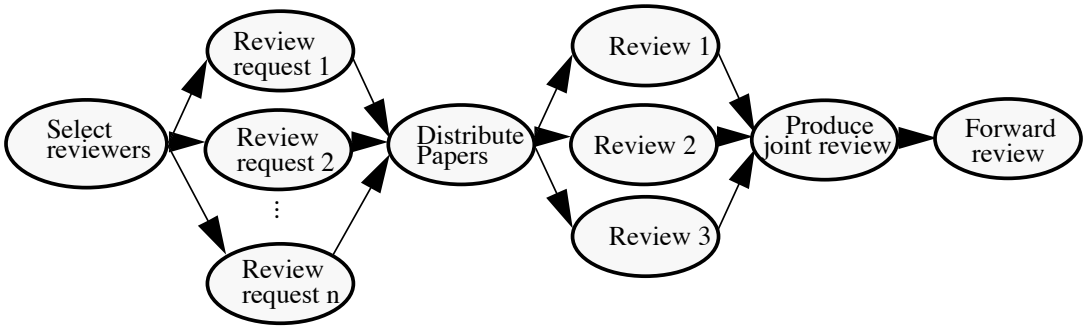


FIGURE 2.1 – Control-flow capturant le processus d'évaluation de papiers (source : [41]).

De manière alternative, les dépendances entre étapes peuvent indiquer des *dépendances de données*. Par exemple, il peut modéliser le processus de gestion de commande de repas (Figure 2.2). Dans ce cas, une tâche ne peut commencer que si la précédente a produit une donnée nécessaire (la commande dans l'exemple cité). Le lien exprime donc une notion de consommation et de production de données. On parle de flux de données (*data flow).*

De manière générale, les outils de workflow ont été proposés afin de gérer de manière *automatique* des processus avec de acteurs et interactions multiples [69, 42, 24]. Par conséquent, ils fournissent les moyens de définir des schémas complexes qu'ils peuvent exécuter [68]. Une fois un schéma modélisé [82], il peut être réutilisé et donc exécuté dans différents contextes ou avec des données différentes. Ce modèle peut aussi être vérifié via des méthodes de validation [49].

2.3.1 Critères de classification des outils de workflow

Pour évaluer les outils de workflow existant, nous choisissons les critères suivants.

Domaine d'application

Comme pour les outils de gestion de traces, nous faisons la différence entre les outils de workflow conçus pour un domaine d'application particulier et les outils qui peuvent être utilisés dans des contextes différents.

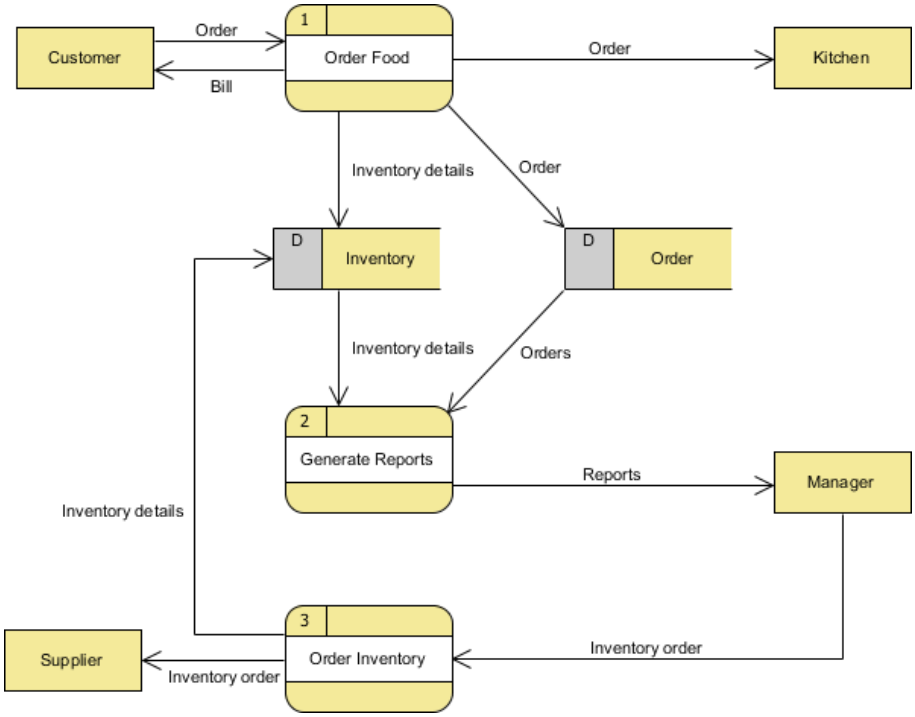


FIGURE 2.2 – Dataflow capturant le processus de commande et gestion de repas (source : visual-paradigm.com).

Type de workflow

Le type de workflow définit la manière de concevoir un workflow. Nous faisons la différence entre les workflows mettant l'accent sur les données (*data-flow*), ceux mettant l'accent sur l'ordonnancement des actions (*control-flow*) et les outils permettant une approche hybride.

Construction du workflow

Il existe des outils qui ne considèrent pas le processus de construction d'un workflow mais permettent de travailler avec des workflows déjà définis par ailleurs [31]. Parmi ceux qui permettent la description de workflows, nous distinguons trois types : les outils qui utilisent des langages spécifiques (DSL), ceux qui fournissent des API de programmation et les outils graphiques.

Gestion d'historique

La gestion d'historique consiste à suivre l'évolution du schéma du workflow en termes de rajouts et suppressions d'actions, ainsi qu'en termes de versions de ces actions. Par exemple, le workflow de gestion de commandes de repas peut rajouter une action *livraison* ou avoir besoin de mettre à jour les actions de gestion de commande afin de prendre en compte de nouvelles informations. Dans les outils de workflow, nous distinguons donc ceux qui permettent d'avoir la trace de l'évolution des schémas et gèrent un historique et ceux qui ne le font pas.

Provenance de données

La provenance de données est un sujet important dans toute science expérimentale [28]. Il s'agit d'établir et de sauvegarder l'information sur le lien entre données expérimentales d'entrée, outils qui manipulent ces données et résultats [70]. Pour les outils de workflow, nous distinguons les outils qui ne considèrent pas cet aspect de leur fonctionnement, ceux qui proposent certains éléments de solution et les outils complets.

Passage à l'échelle

Selon leur domaine d'application, les outils de workflow proposent des mécanismes variés pour gérer le nombre important de tâches ou le volume de données. Dans notre contexte de gestion de traces d'exploitation, ce qui nous intéresse ce sont les mécanismes mis en œuvre pour la gestion de grandes quantités de données. Pour cela, nous nous intéressons aux mécanismes utilisés.

2.3.2 Outils de workflow existants

Chimera [38]

Chimera est un outil qui permet de décrire des workflows, il ne permet cependant pas de les exécuter. Chimera ne propose pas d'interface graphique pour la construction d'un workflow mais propose un langage pour pour décrire le workflow. Le langage permet de décrire des *transformations*, représentant les actions à exécuter. Ces transformations contiennent des informations telles que : l'auteur, la version, le nom du programme, la localisation du programme, les arguments nécessaires, etc. Le langage permet aussi de décrire des *objets de données*, qui sont en fait des fichiers servant au stockage des données produites ou consommées par les actions. L'instanciation du workflow se fait en créant une *dérivation*, qui est l'association d'un objet de données à une transformation.

Kepler [54]

Kepler et un outil de création et d'exécution de workflow, utilisé principalement dans le domaine de la biologie et du biomédical. Kepler offre une interface graphique pour la construction du workflow, qui se fait en piochant dans un catalogue d'actions prédéfinies. Le workflow se construit en connectant les modules, appelés *actors*, via des canaux de communication pour les données. Kepler permet d'utiliser des données locales, mais aussi d'interagir avec des services web. Du point de vue de l'exécution du workflow, Kepler permet une exécution locale mais aussi un déploiement sur des machines parallèles et sur une grille d'exécution. Il sépare donc l'aspect modèle d'analyse des données de son exécution. Kepler intègre un dépôt facilitant le partage des modules. Par exemple, des modules R et Matlab sont disponibles et permettent d'utiliser au sein du workflow des analyses statistiques.

Pegasus [31]

Pegasus est un système qui permet l'exécution d'un workflow en local ou sur une grille de calcul. Il ne permet pas de construire de workflow à proprement parlé, mais utilise en fait les workflows abstraits créés par Chimera. Il permet de transformer un workflow abstrait, en un workflow concret, pour lequel il se charge de placer les tâches à exécuter ainsi que les données en fonction de la machine qui l'exécute. Pegasus se concentre sur les aspects d'exécution du workflow, permettant d'exécuter jusqu'à plusieurs millions de tâches, tout en garantissant des mécanismes de fiabilité et de récupération d'erreurs. Les résultats intermédiaires peuvent être conservés et enregistrés pour conserver la provenance des données. Il propose des outils de monitoring pendant l'exécution du workflow.

Storm [1]

Storm est un système de calcul distribué open-source, dont le modèle de programmation se rapproche d'un système de workflow. Storm permet de définir un workflow, appelé *topologie*, qu'il est ensuite possible de déployer sur une grille de calcul. L'implémentation de Storm est faite en Java et la construction du workflow se fait aussi de manière programmatique, en Java. Les actions du workflow sont en fait des classes Java que Storm va ensuite instancier et exécuter. Les données sont transférées entre les modules sous forme de tuples fortement typés, et sous forme de flux continu de données. La topologie est adaptative en fonction de la charge de données que chaque module doit traiter. Un module est capable de se dupliquer afin de traiter des données en parallèle. L'outil Storm se concentre sur les aspects de fiabilité dans l'environnement distribué, s'assurant que les données envoyées soient correctement transmises, ainsi que des mécanismes de passage à l'échelle permettant d'optimiser le traitement des données.

Taverna [86]

Taverna est un système de gestion de workflow open-source, permettant la création de workflows hybrides, exprimant à la fois des dépendances de données et des dépendances de tâches. Il prend en charge l'exécution du workflow, en local ou sur une grille de calcul. Taverna est orienté pour le domaine de la bio-informatique, dans lequel les interactions avec services web sont très présentes. Taverna se concentre principalement sur les aspects de provenance. Les informations enregistrées sont sur la définition des workflows (auteur, date), l'exécution de ceux-ci (exécutions précédentes et résultats), et sur les données d'entrée (d'où proviennent les données).

Vistrails [27]

Vistrails permet de définir via une interface graphique des workflows hybrides. Il permet principalement de définir des data-flows, cependant, il est aussi possible d'y intégrer des modules de contrôle comme par exemple des boucles ou des exécutions conditionnelles. Vistrails intègre aussi un gestionnaire de dépôts, permettant d'utiliser des modules issus de la communauté. Par exemple, un package de module définit

tout une interface pour la bibliothèque VTK, permettant la représentation de données 2D/3D, ou encore une interface pour l'utilisation de la bibliothèque Matplotlib. Il est aussi possible d'écrire son propre package, afin de définir de nouveaux modules personnalisés. Il n'est donc pas limité à un seul domaine d'application. Vistrails propose des outils aidant au développement du workflow. Par exemple, toutes les modifications du workflow sont enregistrées, un arbre des modifications est construit. Il est alors possible de revenir vers une version précédente, ajouter des modifications, comparer deux versions d'un workflow, etc. Vistrails permet aussi l'exploration des résultats en proposant d'exécuter le workflow plusieurs fois, en faisant varier les paramètres des modules. Des mécanismes de cache sont utilisés afin d'éviter les calculs inutiles d'une exécution à l'autre. Nous présentons l'outil Vistrails plus en détail dans le Chapitre 5.

2.3.3 Synthèse des outils de workflow existants

L'analyse des outils de workflow nous a permis de mettre en évidence les critères nécessaires pour les environnements d'analyse à base de workflows. Il en ressort qu'un outil de workflow pour l'analyse de données doit avoir les critères suivants : il doit proposer une interface pour la construction simple du workflow, permettant de réutiliser des sous-parties de workflow ; utiliser un dataflow, permettant de mettre en valeur les échanges de données entre les modules du workflow ; gérer des méta-données concernant le workflow, notamment la provenance des données utilisées/produites, mais aussi l'historique de construction du workflow ; utiliser des mécanismes permettant le passage à l'échelle concernant le traitement de grandes quantités de données. Actuellement, aucun des outils ne possède tous ces critères à la fois. Le Table 2.2 présente une synthèse des critères observés pour les outils de workflow.

2.4 Synthèse

La validation des systèmes est un aspect critique qui permet de garantir que leur fonctionnement est correct. L'application de méthodes formelles étant complexe à mettre en oeuvre ou trop coûteuse, la validation de systèmes est faite majoritairement en considérant leur comportement à l'exécution. Dans ce contexte-là, l'utilisation de traces d'exécution devient de plus en plus courante. En effet, les traces d'exécution contiennent des historiques détaillés qui permettent de raisonner sur les états de systèmes, détecter des erreurs et évaluer les performances.

Il existe de nombreux outils de manipulation de traces qui se différencient selon leur domaine d'application et selon le type des traitements effectués. Cependant, la plupart restent confinés à un domaine d'application et leurs traitements pour analyser un système sont prédéfinis et limités. Les analystes peuvent difficilement créer leur propre traitement ou faire interopérer les outils existants.

Avec l'évolution des systèmes, en particulier dans le cadre des systèmes embarqués, la gestion de traces fait face à de nouveaux défis. En effet, il devient nécessaire

	Domaine d'application	Type de Workflow	Construction	Historique	Provenance	Passage à l'échelle
Chimera	Générique	data-flow	Textuelle	Non	Oui	Non
Flink	Générique	data-flow	Programmatique	Non	Non	Oui
Kepler	Biologie	hybride	Graphique	Non	Non	Oui
Pegasus	HPC	control-flow	Non	Non	Oui	Oui
Storm	HPC	data-flow	Programmatique	Non	Non	Oui
Taverna	Biologie	hybride	Graphique	Non	Non	Non
Vistrails	Générique	hybride	Graphique	Oui	Oui	Non

TABLE 2.2 – Synthèse des outils de workflow.

de pouvoir facilement réutiliser des analyses, d'adapter des analyses à des cas particuliers ou d'en créer des nouvelles. L'analyse de traces se complexifie et devient composée de plusieurs étapes. Or, les aspects liés au séquencement et à l'automatisation des analyses sont très peu abordés par les outils de gestion de trace existants.

Les outils de workflow sont de plus en plus présents dans le domaine de l'analyse de données. Non seulement ils mettent l'accent sur l'enchaînement de traitements et leur automatisation mais également permettent leur réutilisation. Toutefois, les outils de workflow ne considèrent pas l'analyse de traces d'exécution. Ils pourraient cependant faire bénéficier au domaine de la gestion de traces leurs aspects liés à l'automatisation, mais aussi à la mise en commun de traitements similaires. Ces traitement pourrait être, par exemple, des calculs statistiques, des filtres ou des visualisations.

Notre proposition, présentée en détail dans le chapitre suivant, porte donc sur le rapprochement de ces deux mondes : créer des environnements de gestion de traces bénéficiant des aspects d'automatisation des systèmes de workflow.

Chapitre 3

Propositions pour l'analyse de traces

Les traces d'exécution sont de plus en plus utilisées lors du processus de développement et de mise au point des systèmes. Néanmoins, comme nous l'avons montré dans l'état de l'art, les outils existants restent limités dans leurs analyses et surtout sont cantonnés à des domaines d'application spécifiques. Dans le contexte informatique actuel, marqué par une forte croissance du marché et une diversification impressionnante des produits, de nouvelles propositions sont nécessaires pour une analyse avancée des traces d'exécution.

Les objectifs qui nous ont guidé pour notre proposition de gestion de traces sont les suivants :

- **Généricité.** Notre solution doit proposer des traitements d'analyse qui sont réutilisables et qui ne dépendent pas du domaine d'application, ni de la représentation (le format) des traces d'exécution.
- **Sémantique des traces** Notre solution doit pouvoir intégrer et exploiter des informations provenant du contexte applicatif qui a généré les traces d'exécution. En d'autres termes, nous voulons pouvoir capturer et utiliser la sémantique *métier* des traces, qui souvent n'est pas représentée mais utilisée de manière implicite par les développeurs.
- **Workflow** Notre solution doit permettre l'enchaînement et l'exécution automatique de plusieurs traitements d'analyse de traces.
- **Grandes traces** Notre solution doit rendre possible le travail avec des traces de l'ordre de plusieurs dizaines de giga-octets.

Notre travail se place dans le contexte du projet SoC-Trace [13] qui vise le développement d'outils et de méthodes pour l'analyse des traces d'exécution d'applications embarquées multi-cœur. Ce projet s'adresse aux besoins croissants d'observabilité, de débogabilité et de diminution des coûts de production des systèmes embarqués. Le projet fait collaborer plusieurs équipes de recherche de l'Université Joseph Fourier et de l'Inria, ainsi que les sociétés STMicroelectronics, ProbaYes et

Magillem Services. L'objectif est de fournir un système prenant en charge le processus complet d'exploitation de traces d'exécution : collecte et analyse de traces, ainsi que production et visualisation de résultats.

Ce travail de thèse s'inscrit dans le sous-projet "Infrastructure de gestion de traces" qui fournit le socle commun pour les travaux du projet SoC-TRACE. Ce socle commun a pris la forme d'un outil appelé Framesoc [61, 4]. Framesoc s'occupe du stockage de traces et fournit des traitements pour la manipulation de traces. Il définit le cadre et les règles de programmation pour une intégration et une collaboration entre des traitements d'analyse développés par des parties tierces.

Dans Framesoc, les traces sont importées au sein d'une base de données en utilisant une représentation pivot. Notre première contribution porte sur l'enrichissement de cette représentation (Section 3.1). Cette contribution a été intégrée au sein de Framesoc et a fait l'objet d'une publication dans une conférence nationale [59].

Framesoc permet un enchaînement *manuel* de différents traitements d'analyse. En effet, les résultats d'un traitement d'analyse peuvent être enregistrés et ensuite utilisés en entrée pour un autre traitement. Cette fonctionnalité a été utilisée dans le cadre de SoC-TRACE pour réaliser des scénarios prédéfinis. Notre deuxième contribution porte sur la définition et l'exécution automatique d'enchaînements de traitements d'analyse (système de *workflow*, Section 3.2). Cette contribution a été définie de manière indépendante de Framesoc et définit donc des perspectives d'évolution de l'outil. En particulier, notre solution facilite le travail avec de grandes traces d'exécution de l'ordre de dizaines voire centaines de giga octets de données.

Notre troisième contribution instancie notre système de workflow et définit une analyse permettant d'établir le profil de performances d'un système (Section 3.3). Ce travail a fait l'objet d'une publication dans une conférence internationale [58].

3.1 Représentation de traces d'exécution

Les traces d'exécution provenant de systèmes embarqués sont capturées de manière à maximiser la quantité d'informations, pour ne pas limiter les possibilités d'analyses par la suite. Elles contiennent de nombreux événements de *bas niveau* reflétant le fonctionnement du matériel et du système d'exploitation. Ces événements contiennent beaucoup d'informations, comme illustré sur la Figure 3.1.

Dans cet extrait de trace LTTng, les événements observés sont des événements du noyau Linux. Par exemple, la première ligne indique qu'au temps 501710119, un changement de contexte (`sched_switch`) de processus a été effectué par l'ordonnanceur sur le processeur 0. L'ordonnanceur a arrêté le processus "lttng-consumerd" (`prev_comm`) et a exécuté le processus "swapper/0" (`next_comm`). Cet exemple expose parfaitement le problème des traces d'exécution : elles contiennent une quantité importante d'informations avec une structure non standardisée.

L'outil Framesoc propose de représenter les données de traces en utilisant un format auto-décrit (*Self Defining Pattern*, Figure 3.2). Ce format permet de définir un type d'événement en spécifiant son nom et en définissant le nombre et le type des

```
[501710119] sched_switch: { cpu_id=0 }, { prev_comm="lttng-consumerd", prev_tid=2208, next_comm="swapper/0", next_tid=0 }
[501952796] sched_switch: { cpu_id=0 }, { prev_comm="swapper/0", prev_tid=0, next_comm="lttng-consumerd", next_tid=2208 }
[501953442] syscall_exit_sync_file: { cpu_id=0 }, { ret=0 }
[501953688] syscall_entry_fadvise64: { cpu_id=0 }, { fd=78, offset=65536, len=0, advice=4 }
[501954634] sched_switch: { cpu_id=2 }, { prev_comm="swapper/2", prev_tid=0, next_comm="kworker/2:1H", next_tid=129 }
[501954825] syscall_exit_fadvise64: { cpu_id=0 }, { ret=0 }
[501955088] syscall_entry_ioctl: { cpu_id=0 }, { fd=75, cmd=62982, arg=1 }
[501955296] sched_switch: { cpu_id=2 }, { prev_comm="kworker/2:1H", prev_tid=129, next_comm="swapper/2", next_tid=0 }
[501955465] syscall_exit_ioctl: { cpu_id=0 }, { ret=0, arg=1 }
```

FIGURE 3.1 – Extrait de trace au format LTTng

informations le caractérisant (les paramètres). Ainsi, un événement est représenté par l'entité `Event` et des entités `EventParam`. Les types sont respectivement définis par un `EventType` et des `EventParamType`. Si on reprend l'exemple de trace LTTng, l'événement de la première ligne sera de type `sched_switch`, alors que les informations complémentaires sur les processus concernés seront dans les paramètres de l'événement.

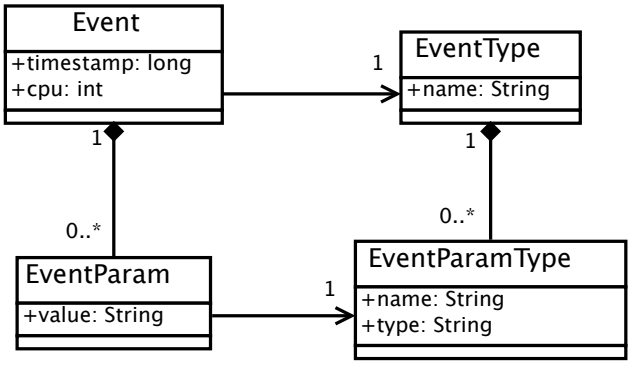


FIGURE 3.2 – Représentation des événements dans Framesoc

Le format Framesoc permet de séparer les informations commune à tous les événements des informations spécifiques à chaque événement. Les informations communes sont définies comme des attributs de l'entité `Event` alors que les informations spécifiques, que nous appelons *informations métier*, sont représentées par des `EventParam`. Ces dernières ne sont utilisables dans une analyse que par des personnes connaissant leur signification. Par exemple, sortie de son contexte, l'information "fd=78" n'apporte aucune information en l'état pour l'analyse d'une trace.

L'entité `Event` capture un ensemble d'informations que l'on retrouve dans une majorité de traces d'exécution, quel que soit leur domaine de provenance ou leur format. Notamment, il s'agit de :

— *Estampille.* L'estampille indique quand est survenu l'événement sur l'échelle temporelle.
— *Type.* Le type de l'événement reflète sa nature. Il permet donc de distinguer les différents événements d'un système. Les événements du même type sont caractérisés par le même type d'informations.
— *Producteur.* Le producteur définit un contexte déclenchant un événement. Cette entité peut être logicielle, comme, par exemple, un processus. Elle peut indiquer également une entité matérielle comme une machine en particulier ou un processeur. En spécifiant un producteur parent, il est possible de définir des hiérarchies entre producteurs.

Nous proposons d'enrichir l'ensemble d'informations communes afin de distinguer entre *événement ponctuel*, *événement état*, *événement lien*, et *événement valeur*. Ces notions ne sont pas nouvelles et sont déjà utilisées par des outils tels que Tracecompass [17], STWorkbench [16] ou VTunes [6]. Néanmoins, elles ne font pas partie de

la représentation initiale des traces. Elles sont calculées à chaque fois que les traces sont manipulées, en interne, et dans des représentations spécifiques. Nous proposons d'intégrer ces informations dès le départ à la représentation de la trace et ainsi capturer un premier niveau de sémantique utile pour l'analyse [44].

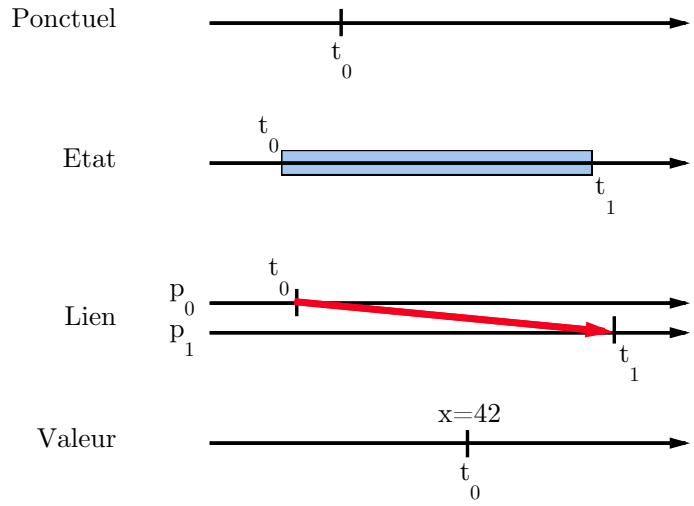


FIGURE 3.3 – Représentation des quatre natures d'événements

Les notions d'*événement ponctuel*, *événement état*, *événement lien*, et *événement valeur* sont imagées dans la Figure 3.3. Nous les définissons comme suit :

— *Événement ponctuel.* Les événements ponctuels sont les événements de base, utilisés dans tous les cas de traces, que nous avons décrit précédemment. Ils servent à représenter un événement survenu à un moment précis dans le temps.
— *Événement état.* Les événements *état* permettent de décrire un événement qui dure dans le temps et qui modélise un état du système. Un état peut représenter, par exemple, un mode de fonctionnement (actif, inactif, etc.) ou un contexte d'exécution (appel de fonction). Un événement état est décrit par un *timestamp* de début (déjà présent dans un événement ponctuel), un *timestamp* de fin et une valeur qui caractérise l'état du système.
— *Événement lien.* Un événement *lien* permet de décrire une relation de causalité entre deux événements. Ces événements sont typiquement utilisés pour représenter des communications et relient un envoi et la réception d'un message sur le réseau. Ils sont décrits, en plus des informations de l'événement ponctuel, par une valeur renseignant un *estampille de fin*, et un *producteur de destination*.
— *Événement valeur.* Les événements *valeur* permettent de décrire l'évolution d'une valeur au cours du temps. Ils permettent de représenter, par exemple, l'évolution d'une consommation énergétique, d'une valeur d'un capteur ou tout autre quantité. Ils contiennent, en plus des informations de l'événement ponctuel, la valeur de cette quantité tracée au moment de l'événement.

Si l'on reprend l'exemple de la trace LTTng de la Figure 3.1, on voit que les deux premiers événements représentent un changement de contexte d'un processus `lttng-consumerd` au processus `swapper/0`, puis inversement. Si l'on extrait l'information métier de ces événements, en utilisant le fait que `prev_comm`, `prev_pid`, `next_comm`, et `next_pid` correspondent aux noms et PIDs des processus, on peut représenter la trace différemment. En effet, nous pouvons représenter les changements de contexte par des événements liens et utiliser un événement état pour le temps passé à exécuter le processus `swapper/0` (Figure 3.4).

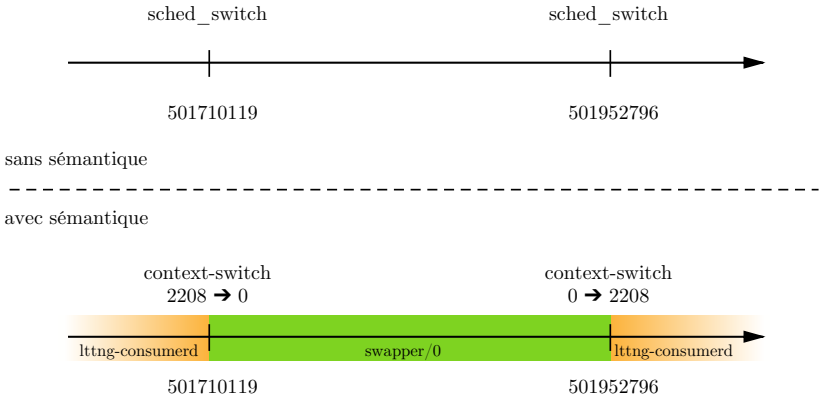


FIGURE 3.4 – Représentation sans et avec sémantique

La validation de notre proposition est discutée au Chapitre 4.

3.2 Système de *workflow*

Pour la gestion de traces, nous proposons une infrastructure logicielle de *workflow*. afin de fournir les fonctionnalités suivantes :

— **Construction d'analyses complexes.** Nous proposons un système où les différents traitements d'analyse de traces d'exécution peuvent être définis et encapsulés dans des modules indépendants. Une analyse complète de traces d'exécution est construite pas à pas, en définissant les différents modules et en les inter-connectant. Le workflow qui en résulte capture le processus complet d'analyse et définit dans ce sens *un plan d'analyse*. Ce plan fournit une vision globale des traitements d'analyse et est indispensable pour les points suivants.

— **Automatisation du processus d'analyse.** Une description globale et explicite de toutes les étapes de traitement des traces d'exécution permet leur lancement et exécution automatiques. Cela facilite le travail et évite des erreurs à l'analyste qui n'est plus obligé de lancer à la main les différents traitements, ni de s'occuper de la sauvegarde et de l'éventuelle transformation de données.

— **Réutilisation du plan d'analyse.** Une fois défini, le plan d'analyse peut être exécuté plusieurs fois, sur des données d'entrée différentes. Ceci peut être

utile quand la même analyse doit être appliquée sur différents ensembles de données, ou quand les analystes doivent établir un lien entre les variations de résultats et données d'entrée. Le fait que l'exécution soit automatique évite un travail fastidieux et répétitif à l'analyste qui peut donc se concentrer uniquement sur les résultats. Il est aussi possible de réutiliser une sous-partie du plan d'analyse afin de faciliter la construction de nouvelles analyses.

— **Résultats reproductibles.** Le workflow capture la séquence d'analyse des traces d'exécution. Appliqué sur les mêmes données et configuré de la même manière, il permet de reproduire des résultats d'analyse [75, 30, 28]. Ceci est utile pour comprendre comment les traces sont manipulées, pour vérifier certaines conclusions ou pour faciliter le travail d'analystes qui ne sont pas à l'origine les concepteurs du plan d'analyse.

— **Optimisation des performances de l'analyse.** Le fait d'avoir une description globale du plan d'analyse permet de gérer la manière dont s'effectuent les différents traitements afin d'optimiser la manipulation de grandes quantité de données ou accélérer les calculs. Ainsi, il est possible, par exemple, de distribuer les étapes de l'analyse, de mettre en place des mécanismes de transferts de données efficaces ou de gérer des caches de données. Dans notre proposition nous nous intéressons à la production de résultats au fur et à mesure de l'analyse afin de fournir un retour à l'analyste lors de traitements longs.

Notre proposition porte sur trois volets : la manière de construire un *workflow* (Modèle de programmation, décrit dans Section 3.2.1), un ensemble de modules d'analyse de traces pouvant participer dans un *workflow* (Section 3.2.2) et, enfin, l'exécution d'un *workflow* (Section 3.2.3).

La validation de cette proposition est discutée lors de deux implémentations du modèle, dans les Chapitre 5 et Chapitre 6.

3.2.1 Modèle de programmation

Nous pensons que les systèmes de *workflow* peuvent réutiliser naturellement les concepts de la programmation par composants [55]. Dans ce modèle, la création d'une application se fait en assemblant des *composants*, qui représentent des *services* et qui communiquent entre eux via des interfaces prédéfinies. Dans les systèmes de *workflow* chaque *module* peut être représenté par un composant. Les liens entre les modules du *workflow* peuvent être représentés par des communications entre les composants via leurs interfaces prédéfinies. Ainsi, la programmation par composants permet de séparer [36] toute la partie *non-fonctionnelle* (exécution, transfert de données, synchronisation, etc.) gérée par l'infrastructure de la partie *fonctionnelle* qui représente l'analyse de traces en elle-même.

Dans notre modèle de programmation, chaque traitement d'analyse de données de traces est encapsulé dans un module séparé. Chaque module définit explicitement quels types de données il utilise en entrée et quels types de données il produit en sortie. Ceci est fait en spécifiant un ensemble de *ports* d'entrée et de sortie pour chaque module.

La connexion entre modules se fait en inter-connectant des ports d'entrée à des ports de sortie du même type. La Figure 3.5 montre un exemple de connexion entre deux modules.

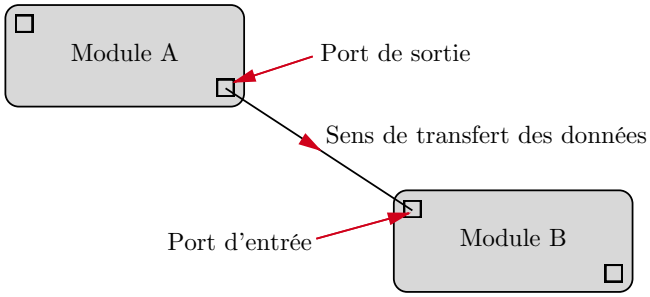


FIGURE 3.5 – Exemple de connexion entre deux modules

Nous proposons d'échanger les données entre les modules sous forme de flux (*streaming*). Les modules produisent et consomment les données dès que celles-ci sont disponibles [63]. Les données sont donc transmises dans le *workflow* de manière continue.

Comme montré dans la Figure 3.6, les données sont d'abord produites par un module (éléments en production). Elles forment un flux qui transite entre les deux modules (éléments en attente), typiquement via un *buffer*. Les données sont enfin traitées par le module suivant (éléments en consommation).

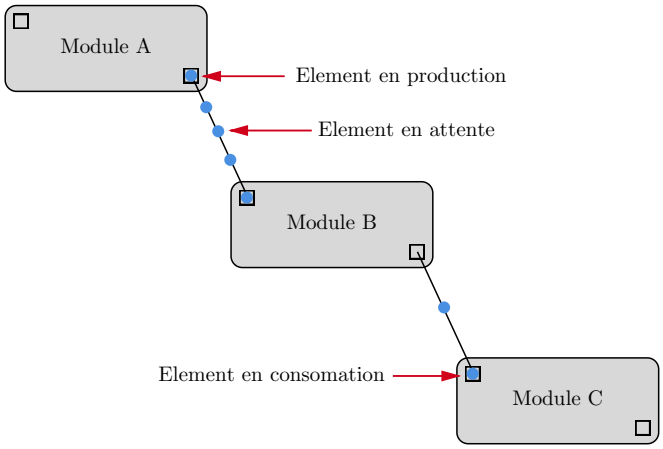


FIGURE 3.6 – Mécanismes de flux de données

Nous proposons plusieurs possibilités de connexion entre ports. La plus simple est la connexion $1-1$ que nous avons illustrée juste avant. Une autre possibilité est de connecter un port de sortie d'un module à plusieurs ports d'entrée de plusieurs autres modules. C'est un mécanisme de *broadcast* permettant de transférer l'intégralité des données à tous les modules de destination.

Trois mécanismes supplémentaires, implémentés par des modules spécifiques, permettent de contrôler les flux. Le premier mécanisme distribue les éléments d'un

flux en *round robin* sur les ports d'entrée. Le module prends en entrée un flux et envoie chaque élément de ce flux vers un des N port de sortie différent. Le second mécanisme rassemble N éléments provenant de N flux différents (ports de sorties) dans un même flux de données (port d'entrée). La Figure 3.7 montre un exemple de ces deux modules avec $N = 3$.

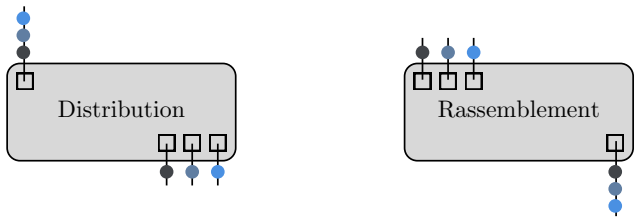


FIGURE 3.7 – Modes de transfert des données

Le troisième mécanisme permet de limiter le nombre d'éléments d'un flux à considérer. On définit un nombre d'éléments à ignorer en début de flux, et un nombre d'éléments à garder. Par exemple dans une trace d'exécution, on peut vouloir ne s'intéresser qu'au régime stationnaire de l'application, on souhaite donc éliminer le début de la trace que l'on pense correspondre au régime transitoire.

3.2.2 Modules prédéfinis

Nous proposons plusieurs modules de traitements de données qui encapsulent des opérations communes et souvent utilisées. Ces modules peuvent être vus comme une bibliothèque de modules réutilisables dans des workflow d'analyse différents. Les modules que nous proposons fournissent des opérations de filtrage, de calculs statistiques, et de visualisation de résultats.

Concernant les opérations de filtrage, plusieurs modules sont disponibles pour filtrer les événements de la trace selon les différents attributs de notre modèle. Ainsi, il est possible, par exemple, de filtrer les événements d'un certain type, de filtrer les événements provenant d'un certain producteur ou les événements *état*. Les modules de filtre possèdent tous deux ports de sortie, un pour les données qui satisfont la condition de filtrage et un pour celles qui ne la satisfont pas.

Concernant les calculs statistiques, un premier module permet de compter le nombre d'éléments. Un second module, plus complexe, permet de calculer des statistiques sur les événements comprenant la moyenne, l'écart-type, les valeurs minimum et maximum, et la somme. Un des ports d'entrée de ce module permet de spécifier sur quelle donnée calculer les statistiques : la durée des événements *état*, l'écart entre l'estampille de début et de fin des événements *lien*, ou encore les valeurs des événements *valeur*.

Concernant la manipulation de résultats d'analyse, nous proposons des modules permettant de créer des graphiques. Notamment, il est possible de créer à partir de statistiques un graphique en boîte à moustache. Les événements *valeur* peuvent être utilisés pour créer des courbes, en utilisant le timestamp en abscisse et la valeur en

ordonnée. Sachant que tous les modules peuvent représenter les données manipulées de manière textuelle, un module dédié permet d'enregistrer des données dans un fichier.

L'ensemble des modules proposé a été défini au cours de nos expériences d'analyses de traces et a vocation à être enrichi.

3.2.3 Modèle d'exécution

Pour définir notre modèle d'exécution, nous avons été guidés par l'objectif de pouvoir traiter les grandes quantités de données contenues dans les traces d'exécution. Pour cela, il faut que les différentes parties de l'analyse puissent avancer de manière parallèle, aidées par les transferts de données en *streaming*. Nous proposons donc d'avoir un fil d'exécution par module d'analyse. La communication des messages entre ces modules en utilisant des tampons, en ayant un tampon par connexion entre modules.

3.3 Analyse générique d'un système Linux

Notre troisième proposition porte sur une méthodologie pour analyser les performances de logiciels s'exécutant sur un système Linux. Nous avons choisi Linux puisque ce système où des systèmes d'exploitation très semblables sont largement utilisés, en particulier dans le domaine des systèmes embarqués. L'utilisation d'un système *open source* amène également les avantages d'utiliser des interfaces standardisées et nous permet de bénéficier de nombreux outils de travail sous Linux.

Nous analysons les performances de logiciels s'exécutant sous Linux, sans utiliser des informations spécifiques. Nous collectons une trace en utilisant seulement des informations provenant des interfaces du système Linux. Ensuite, notre workflow d'analyse utilise les informations de cette traces afin de caractériser les performances des logiciels observés. Notre méthode d'analyse n'est donc dépendante ni d'un contexte, d'un programme, ou d'un modèle de programmation, de plus il n'est pas nécessaire de modifier (ou recompiler) les programmes observés. Les expériences de validation de cette proposition sont discutées dans le Chapitre 6.

Dans la suite, nous détaillons les métriques que nous utilisons pour caractériser les performances d'un logiciel, les traces que nous proposons de capturer, ainsi que les modules d'analyse de ces traces. La Figure 3.8 détaille le workflow abstrait du processus d'analyse générique, de la capture à l'obtention des résultats.

3.3.1 Les métriques observées

Nous considérons plusieurs métriques caractérisant de manière générique l'exécution d'un système :

— La durée d'exécution, afin de donner une indication sur le temps passé.
— L'occupation CPU, donnant une indication sur l'utilisation des différents processeurs, et ainsi sur l'aspect parallèle de l'exécution.

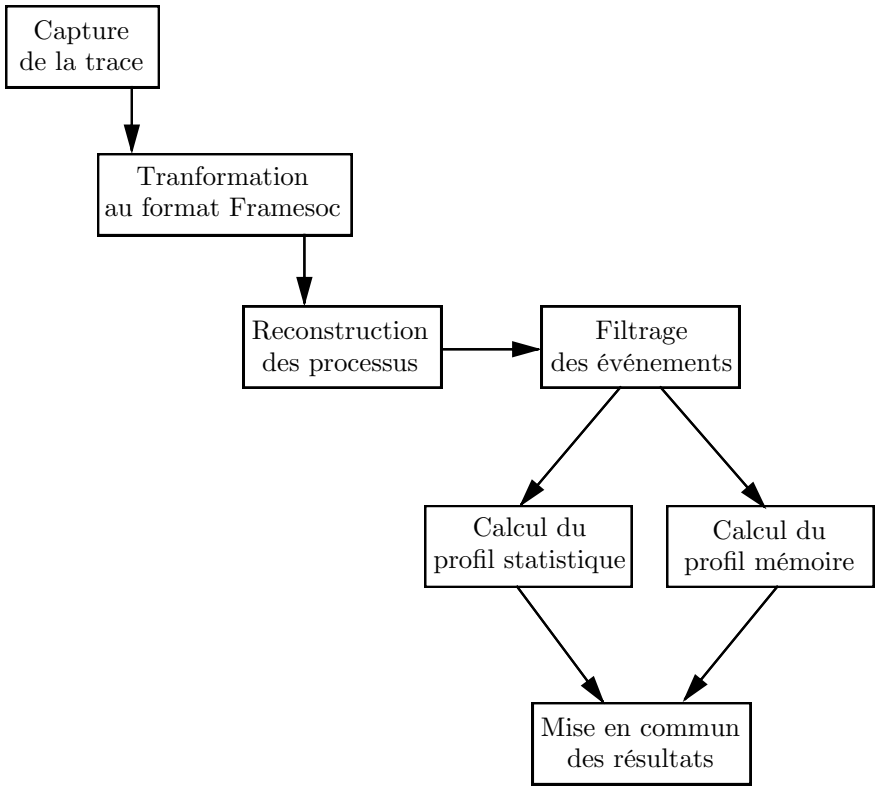


FIGURE 3.8 – Workflow abstrait de l'analyse générique d'un système Linux

— La répartition temps noyau, temps utilisateur et temps inexploité (*idle*), afin de connaître les proportions de temps passé dans le noyau et suspendu.
— La répartition de l'utilisation des différentes parties du noyau, renseignant quelles ressources matérielles sont les plus sollicitées (mémoire, réseau, disque, graphique, processeur).
— L'utilisation de la mémoire, en termes de quantités allouées mais également la manière d'allouer (séquences de fonctions, nombre d'appels).
— La répartition des opérations exécutées entre celles liées a du calcul pur et celles en lien avec l'accès aux données en mémoire.

3.3.2 Collecte de traces génériques

Afin de capturer une trace contenant ces informations, sans dépendre du type d'application s'exécutant sur le système, nous proposons d'utiliser des outils utilisant les interfaces du noyau Linux [43]. Cette méthode ne nécessite aucune instrumentation du code des applications observées.

La Figure 3.9 présente les interfaces utilisées pour la collecte de trace. Nous proposons d'utiliser LTTng [33, 32, 7] qui est le standard *de facto* pour la capture de traces. LTTng permet de capturer une trace noyau mais aussi des événements de l'espace utilisateur.

Pour ce qui est du traçage de l'espace noyau, LTTng implémente les *tracepoints*

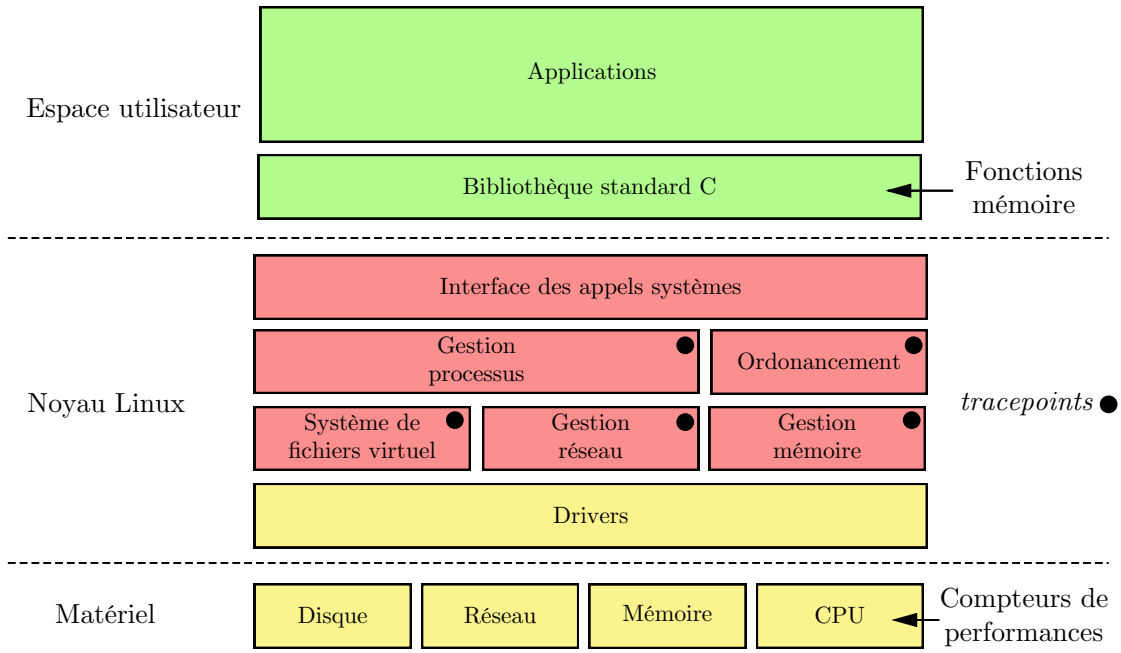


FIGURE 3.9 – La collecte de trace pour la caractérisation générique

qui sont situés à des endroits stratégiques du noyau, afin d'enregistrer des informations sur l'exécution du système. Cela permet typiquement de récolter des informations sur l'ordonnancement des processus, les changements de contexte, les allocation mémoire du noyau, etc. LTTng permet aussi de tracer les appels système. Ces événements renseignent du début et de la fin de l'appel, ainsi que son nom et les valeurs retournées. (cf. événements `syscall_entry` et `syscall_exit` Figure 3.1).

Pour ce qui est du traçage de l'espace utilisateur, LTTng fournit une bibliothèque spécialisée, nommée `liblttng-ust`, permettant de définir des *tracepoints* dans du code utilisateur.

LTTng permet d'ajouter pour chaque événement des informations de *contexte*. Ces informations peuvent être lié aux processus qui a créé l'événement (PID, PPID, TID, pthread_id, nom, *niceness*, etc.), ou en lien avec la machine (nom, adresse ip, compteur de performance).

Trace noyau

Le traçage des *tracepoints* peut être activé ou désactivé, selon les besoins d'analyse de système, de manière indépendante. Pour notre analyse, nous avons décidé de tracer le passage dans tous les *tracepoints*, afin d'obtenir une information complète de l'utilisation des fonctions du noyau.

Nous avons décidé de ne rajouter aucune information de contexte aux événements. En effet, le fait de capturer plus d'information augmente la perturbation du système tracé, ainsi que la taille de la trace obtenue. Comme la trace contient déjà les informations d'ordonnancement des processus, nous pouvons recalculer quel

événement a été produit par quel processus *a posteriori*. En effet, l'événement indiquant un changement de contexte contient les informations des noms et PIDs des processus exécutés.

Compteurs de performances

La plupart des processeurs actuels possèdent des compteurs de performances permettant de compter certaines opérations du processeur. Le noyau Linux permet d'accéder facilement à ces valeurs en définissant une interface commune faisant abstraction de l'hétérogénéité matérielle sous-jacente. Les principaux compteurs de performances renseignent le nombre de cycles CPU, le nombre d'instructions processeur exécutées, les changements de contexte, les migrations de processus, les défauts de pages mémoire et les accès de cache.

LTTng permet d'accéder à ces compteurs de performances, via l'interface du noyau Linux, en ajoutant ces valeurs dans un contexte pour un type d'événements. Nous avons décidé de regarder les compteurs d'instructions exécutés par le processeur. En effet, c'est un compteur classique disponible sur quasiment tous les systèmes. Certains compteurs plus particuliers, comme par exemple ceux en lien avec les performances énergétiques, ne seront pas disponibles tout le temps, c'est pourquoi nous ne les avons pas considérés. De plus, le nombre d'instructions exécutées est un bon indicateur de performance.

En regardant périodiquement ces compteurs, il est possible d'enregistrer dans la trace l'évolution de ces valeurs. Nous avons choisi de relever la valeur des compteurs à chaque changement de contexte. De ce fait, on obtient une granularité suffisamment petite permettant de suivre l'évolution. De plus, cela permet d'isoler les valeurs pour chaque processus.

Bibliothèque standard C

LTTng fournit une bibliothèque, nommée `liblttng-ust-libc-wrapper.so`, qui permet de tracer les fonctions liées à la mémoire. Cette bibliothèque modifie l'appel de plusieurs fonctions pour qu'un événement soit enregistré chaque fois qu'elle est appelée. Les fonctions implémentées sont les fonctions `malloc`, `calloc`, `realloc`, `free`. Il suffit ensuite d'utiliser la fonction de pré-chargement de bibliothèque, *via* la variable `LD_PRELOAD`, afin d'activer le traçage des fonctions liées à la mémoire.

En utilisant les informations enregistrées dans les événements, il est possible de suivre l'évolution de la mémoire allouée [85]. En effet, dans la trace, l'événement lié à un appel à la fonction `malloc` contient aussi la taille allouée, de même pour les autres fonctions. Il est alors possible de connaître le nombre, la fréquence, des allocations. De manière analogue aux compteurs de performance, le fait d'avoir les informations d'ordonnancement des processus nous permet d'avoir le suivi des allocations pour chaque processus.

3.3.3 Modules d'analyse

Nous proposons plusieurs modules qui permettent de transformer et d'analyser une trace obtenue à l'aide de LTTng.

Transcription du format CTF au format générique

Ce module sert à transformer la trace binaire obtenue par LTTng afin de la représenter dans notre format générique. Les événements LTTng sont représentés par des événements ponctuels avec leur *timestamp*, type, producteur matériel et leurs paramètres spécifiques respectifs.

Reconstruction de la hiérarchie logicielle des processus

En utilisant les informations d'ordonnancement, ce module calcule l'appartenance des différents événements à des processus logiciels. En effet, en utilisant les événements de changement de contexte, il est possible de connaître le processus qui a été actif jusque là. On peut donc déduire que les événements suivant le dernier changement de contexte et précédent celui-ci sont liés à l'exécution de ce processus.

En utilisant les événements de début et de fin d'appels système, le module distingue les périodes d'exécution de code utilisateur, celles d'exécution de code système, et celles où le système est en pause (*idle*). Ces informations sont représentées en utilisant les événements *état*.

Utilisation mémoire

L'évolution de la mémoire est construite en utilisant des événements *valeur* contenant les informations provenant de la trace utilisateur de la librairie standard C. Pour chaque événement correspondant à la fonction d'allocation mémoire (`malloc`, `calloc`, `realloc`), le module calcule la nouvelle utilisation mémoire en ajoutant la taille allouée à un compteur global, puis crée un événement contenant la valeur de ce compteur global. Pour une opération de libération (`free`), le module calcule la nouvelle utilisation mémoire en soustrayant la taille allouée précédemment au compteur global, puis crée un événement contenant la valeur de ce compteur global.

Deuxième partie
VALIDATION

Cette partie porte sur la validation de nos propositions. Notre première proposition, la modélisation des traces d'exécution, a été intégrée au sein du projet SoC-TRACE par les ingénieurs de développement. L'utilité de cette fonctionnalité est décrite dans le Chapitre 4.

Notre deuxième proposition, portant sur un système générique de workflow, a été validée de deux manières. Tout d'abord, il a fait l'objet d'une implémentation au sein de l'outil de workflow Vistrails [27]. L'implémentation et la validation de ce travail sont présentés dans le Chapitre 5.

Les limitations de Vistrails ont motivé l'implémentation d'un prototype de workflow dédié à l'analyse de traces. Les détails d'implémentation et la validation sont donnés dans le Chapitre 6. Ce chapitre présente également l'utilisation du prototype pour l'analyse des performances des systèmes Linux. De ce fait, il décrit également la validation de notre troisième proposition.

Chapitre 4

Analyse de trace par structuration sémantique

Ce chapitre présente la validation de la proposition concernant le format et la modélisation des traces d'exécution. Dans le contexte du projet SoC-Trace, en collaboration avec la société STMicroelectronics, nous avons pu travailler sur des traces issues de cas d'usages définis. En s'appuyant sur ces cas d'usage, nous avons pu valider l'utilité pour l'analyse de traces de notre modèle enrichi avec les notions d'événements *ponctuels*, *états*, *liens* et *valeurs*. Ce chapitre correspond aux travaux de recherche publiés sous forme d'un rapport technique [57] et d'un article dans une conférence nationale [59].

Le chapitre est organisé comme suit. Après avoir présenté la notion d'anomalie au niveau des événements de la trace (Section 4.1), nous montrons comment des outils statistiques peuvent être utilisés pour détecter de telles anomalies (Section 4.2). Nous utilisons les valeurs moyennes et le calcul de corrélation pour établir des distances entre des séries d'événements. Les calculs sont effectués sur les informations enrichies que nous avons rajoutées pour représenter les événements *ponctuels*, *états*, *liens* et *valeurs*. La dernière section (Section 4.3) discute de notre intégration de ce travail au sein de l'outil Framesoc développé dans le cadre du projet SoC-Trace.

4.1 Détection d'anomalies dans les traces

Dans ce travail nous nous intéressons à la détection d'anomalies dans les traces d'exécution. Nous définissons une anomalie comme étant tout comportement non attendu d'un système[19].

4.1.1 Anomalies dans une trace

Nous considérons les cas d'analyse de traces d'exécution où il n'y a pas de spécifications préalables disponibles sur le comportement attendu du système. Par conséquent, la détection automatique d'anomalies est faite en deux phases : la définition de comportements corrects (assimilés à des comportements attendus) du système d'abord et la détection des comportements divergents (anomalies) ensuite. Pour que l'identification d'un comportement correct soit possible, la trace doit contenir plusieurs enregistrements s'y référant. C'est le cas, par exemple, des applications multimédia qui ont un comportement périodique.

Dans le cadre de SoC-Trace, nous considérons les comportements attendus à granularité d'événement. En d'autres termes, nous nous intéressons à définir les événements attendus et à détecter les événements représentant des anomalies. Les comportements normaux et divergents sont exprimés en fonction des valeurs des attributs communs des événements dans notre modèle générique de trace.

Dans ce travail, nous utilisons l'intervalle de confiance [67] pour estimer les comportements moyens, considérés normaux. L'intervalle de confiance à 99% pour une série de n éléments et de variable X est définie par :

$$\left[\bar{x} - 3 \times \frac{\sigma(X)}{\sqrt{n}}; \bar{x} + 3 \times \frac{\sigma(X)}{\sqrt{n}}\right]$$

Dans la formule, $\bar{x}$ est la moyenne des valeurs de la variable X et $\sigma(X)$ est l'écart type. Nous définissons comme anomalies les événements dont les valeurs des paramètres estimés sont en dehors de l'intervalle de confiance.

4.1.2 Causes d'une anomalie

Une anomalie peut se produire une seule fois pendant l'exécution ou apparaître plusieurs fois. Dans le cas d'une seule occurrence, nous laissons l'analyse au développeur. Dans le cas de plusieurs occurrences, nous nous intéressons à établir des liens de cause à effet qu'il pourrait y avoir entre ces occurrences et les autres événements de la trace. Pour cela, nous comparons l'ensemble d'occurrences d'événements anormaux du même type et des ensembles formés par les autres événements de la trace (regroupés également par type). Les questions auxquelles nous essayons de répondre sont les suivantes : *L'apparition d'une anomalie coïncide-t-elle avec l'apparition d'un autre événement ? L'anomalie apparaît-elle uniquement quand cet autre événement se produit ?*

Pour répondre à ces questions, nous utilisons le fait que les ensembles d'événements sont ordonnés dans le temps (série temporelle) et visons à détecter une *corrélation temporelle* entre deux séries d'événements. La première série contient les anomalies, alors que la deuxième contient l'ensemble des événements d'un autre type. Si une corrélation est établie, nous considérons que l'anomalie peut avoir comme cause probable les actions reflétées par la deuxième série d'événements. Pour considérer deux séries d'événements, nous utilisons les *histogrammes*[64] et le *coefficient de corrélation linéaire*.

Histogramme Un histogramme est un outil statistique qui permet une visualisation par agrégation de valeurs sur un intervalle de temps. En coupant cet intervalle en plusieurs segments, on obtient une représentation plus synthétique de ces valeurs. Dans notre cas, l'histogramme est utilisé pour représenter la densité (nombre d'occurrences) d'événements par intervalle de temps et par type d'événement. L'objectif est de pouvoir mettre en évidence les événements se produisant dans les mêmes intervalles de temps. La difficulté ici est de déterminer en combien de segments nous allons découper notre espace de temps. En effet, un trop petit nombre de segments ne permettrait pas de faire ressortir une concentration local d'événements. Il en est de même pour un nombre trop grand où l'on aurait une vision sur un trop petit intervalle de temps. Il est usuel de prendre comme nombre de segments : $\sqrt{N}$ où N est le nombre d'éléments.

Corrélation La corrélation entre deux variables aléatoires est le degré de dépendance de ces deux variables [3]. Elle se mesure grâce au coefficient de corrélation linéaire qui est une mesure numérique comprise entre -1 et 1. Si la valeur absolue de cette métrique est comprise entre 0.5 et 1, on dit qu'il y a une corrélation forte. Sinon, il y a une faible corrélation. Le coefficient de corrélation est le rapport entre la covariance des deux variables aléatoires et le produit de leur écart-type :

$$corr(X, Y) = \frac{cov(X, Y)}{\sigma_X \times \sigma_Y}$$

En prenant comme variables les temps d'occurrence de deux types d'événements différents, l'utilisation de la corrélation nous permet d'établir si les événements se produisent en même temps. En effet, dans notre cas, nous nous intéresserons aux événements se produisant dans les mêmes intervalles de temps. De manière plus générale, il s'agit d'établir le degré de proximité temporelle entre les deux séries d'événements, ce qui peut être considéré comme une mesure de distance.

4.2 Distance entre deux séries d'événements

Dans cette partie, nous présentons notre approche expérimentale avec le logiciel R[74, 12]. Nous présentons, dans un premier temps, les cas d'usages fournis par STMicroelectronics sur lesquels nous avons travaillé.

La Section 4.2.2 présente l'intuition de corrélation entre deux ensembles d'événements en explorant la représentation visuelle. La Section 4.2.3 considère un premier calcul du coefficient de corrélation en découpant la trace en tranches de temps égales. La Section 4.2.4 considère le calcul du coefficient de corrélation dans le cas d'ensembles à forte disparité dans leur cardinalité. Le calcul est fait en découpant la trace en tranches de temps irréguliers. La Section 4.2.5 donne les schémas algorithmiques pour effectuer les deux calculs selon les méthodes présentées précédemment.

4.2.1 Cas d'usage

Nous disposons de deux traces fournies par STMicroelectronics. Les deux cas d'usage, nommés *Unicast* et *TSRecord*, concernent le fonctionnement d'un décodeur

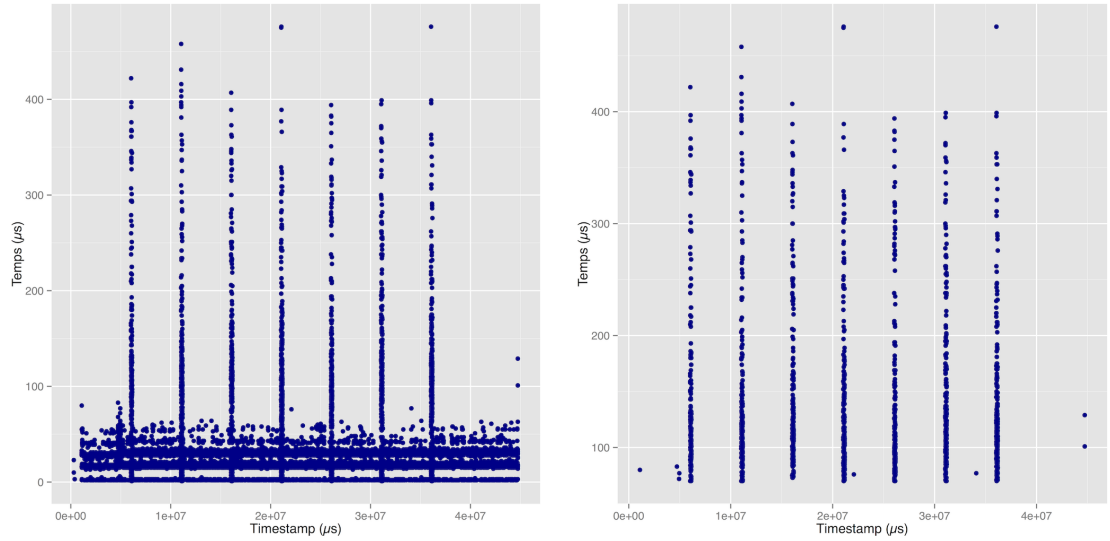
vidéo. Les traces brutes sont produites en traçant les différentes couches logicielles (e.g. noyau Linux, intergiciel ST, ...). Les informations tracées reflètent des appels de fonctions, des interruptions logicielles et matérielles, des communications réseau, etc.

Unicast

Dans le cas *Unicast*, le décodeur lit un fichier depuis le réseau et le diffuse sur sa sortie vidéo. Durant l'exécution surviennent des sauts d'images et des craquements audio qui indiquent un problème de décodage. Le problème de décodage se traduit dans la trace par un temps d'exécution trop long de certains appels à `SoftIRQ`. Ces durées anormalement longues sont en fait un effet de bord d'une autre tâche : `Flush`.

Dans notre modèle enrichi de traces, les événements `SoftIRQ` sont des événements *états* et disposent de l'information de durée. Sont considérés comme des anomalies les appels à `SoftIRQ` qui ont une durée *trop* longue. Cette durée est estimée par rapport à la durée moyenne des traitements `SoftIRQ`. Nous calculons, en effet, la moyenne $\bar{x}$ et l'écart type σ et définissons comme anormaux les éléments en dehors de l'intervalle de confiance.

La figure 4.1 représente les événements `SoftIRQ` de la trace. Chaque point représente un événement avec en ordonnée son estampille et en abscisse sa durée. La partie (a) montre tous les événements, alors que la partie (b) contient uniquement les événements anormaux.



(a) Ensemble des événements `SoftIRQ` (b) Événements `SoftIRQ` anormaux

FIGURE 4.1 – Filtrage des événements `SoftIRQ` selon leur durée.

TSrecord

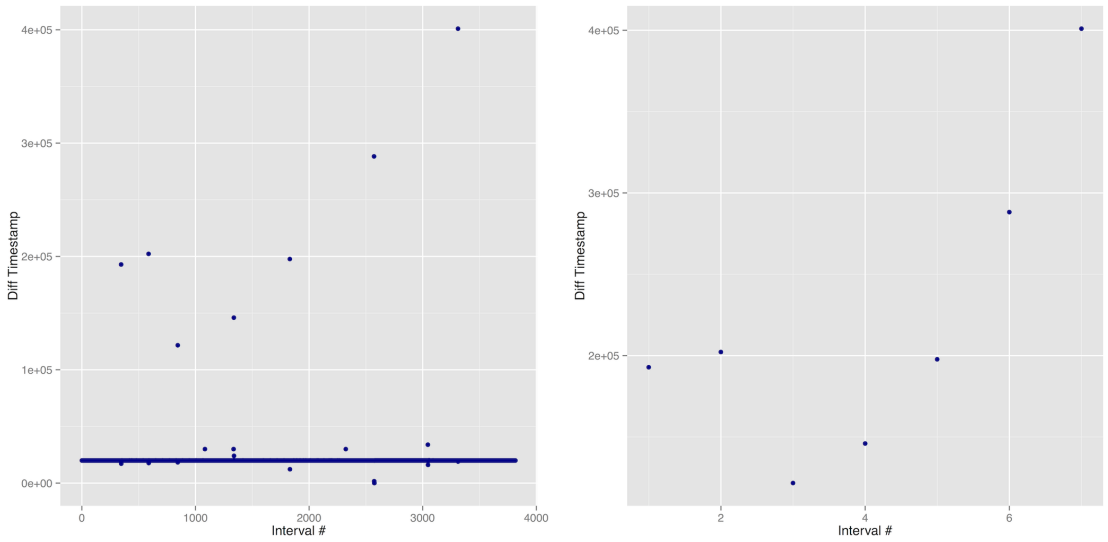
Dans le cas TSrecord, le décodeur lit un flux vidéo depuis le réseau et l'enregistre sur un disque dur USB. La vidéo enregistrée présente des sauts d'images et des craquements audio qui indiquent une perte d'information.

Le problème est le suivant. Toutes les $100ms$, les buffers IP qui stockent le flux vidéo sont vidés dans la zone mémoire correspondant au cache disque. Or, toutes les $5000ms$, le cache est verrouillé lors de son écriture sur disque. Par conséquent, le flux vidéo qui arrive entre temps est perdu.

Dans la trace, ceci peut être vu au niveau des appels à `TSrecord` qui est responsable de la lecture du flux vidéo depuis le réseau et plus particulièrement du vidage des buffers IP vers le cache disque. `TSrecord` ne s'exécute pas lorsque survient la tâche `USB-storage`, responsable de l'écriture effective sur le disque.

Dans ce cas d'usage, nous nous intéressons à la périodicité des appels `TSrecord`. Pour cela, nous considérons les intervalles de temps entre deux appels consécutifs. Nous considérons comme anormaux les éléments qui ne rentrent pas dans l'intervalle de confiance de la moyenne des durées des intervalles.

La figure 4.2 représente la périodicité des événements `TSrecord`. Chaque point représente la durée d'une intervalle entre deux appels consécutifs, avec en ordonnée le numéro de l'intervalle et en abscisse sa durée. La partie (a) montre tous les intervalles, alors que la partie (b) contient uniquement les intervalles anormalement longs.



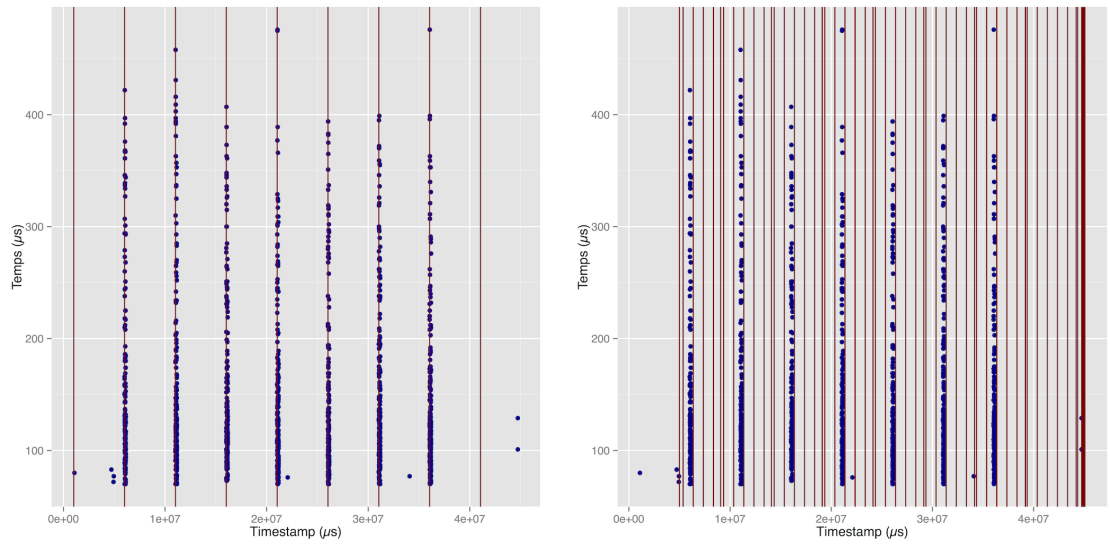
(a) Durée entre deux appels consécutifs (b) Intervalles à périodicité cassée

FIGURE 4.2 – Filtrage des événements `TSrecord` selon la durée entre deux appels consécutifs.

4.2.2 Corrélation visuelle

Dans le cas d'*Unicast*, nous superposons visuellement les événements `SoftIRQ` anormaux et les événements d'un autre type. Dans la figure 4.3, les événements `SoftIRQ` sont représentés par des points, alors que les occurrences des autres événements sont indiqués par des traits rouges verticaux. Dans la partie (a) nous observons

une superposition parfaite, alors que dans la partie (b) ce n'est pas le cas. En conclusion, les événements `Flush` sont intéressants à considérer en tant que cause probable des anomalies.



(a) *Unicast* : anomalies `SoftIRQ` superposées avec `Flush`

(b) *Unicast* : anomalies `SoftIRQ` superposées avec `Cyclesoak`

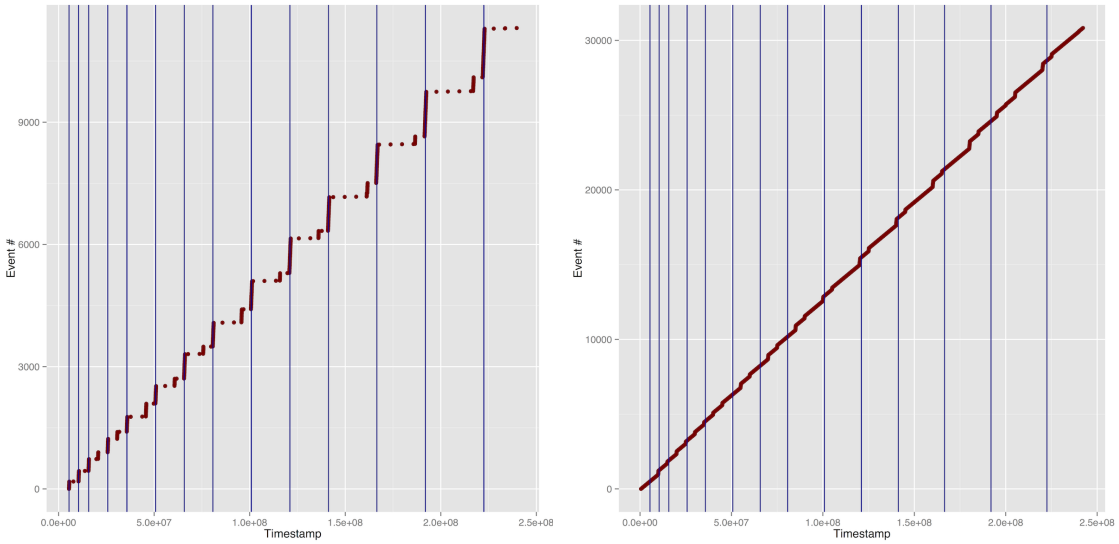
FIGURE 4.3 – Corrélation visuelle pour *Unicast*

De manière analogue, nous pouvons établir visuellement un lien entre les événements `TSrecord` bloqués et les événements d'un autre type dans le cas d'usage *TSrecord*. Dans la figure 4.4, les événements `TSrecord` sont représentés par des traits bleu verticaux, alors que les occurrences des autres événements sont indiqués par des points rouges. Dans la partie (a) nous observons une superposition majoritaire, alors que dans la partie (b) ce n'est pas le cas. En conclusion, les événements `USB-storage` sont intéressants à considérer en tant que cause probable des anomalies.

4.2.3 Corrélation par découpe de temps égales

Notre objectif étant d'établir la corrélation entre deux ensembles d'événements de manière numérique et non visuelle, nous avons utilisé la mesure de corrélation linéaire appliquée à la densité d'événements des deux ensembles. Pour chaque ensemble, nous avons partagé l'exécution en tranches de temps égales et considérons le nombres d'occurrences par tranche. La densité pour chaque ensemble est donc représentée par une suite numérique dont chaque élément représente une tranche de temps. Nous utilisons la fonction de corrélation sur ces deux suites pour obtenir la valeur du coefficient de corrélation linéaire.

Pour le cas d'usage *Unicast* et les types d'événements représentés dans Figure 4.3, nous obtenons $0,85$ pour la corrélation entre `SoftIRQ` et `Flush` et $0,15$ pour la corrélation entre `SoftIRQ` et `Cyclesoak`. La mesure reflète donc bien ce que nous avons constaté de manière visuelle. De manière analogue, pour le cas d'usage *TSrecord* et



(a) Anomalies `TSrecord` superposées avec `USB-storage`

(b) Anomalies `TSrecord` superposées avec `Kworker`

FIGURE 4.4 – Corrélation visuelle pour *TSrecord*

les types d'événements représentés dans Figure 4.4, nous obtenons $0,83$ pour la corrélation entre `TSrecord` et `USB-storage` et $0,11$ pour la corrélation entre `TSrecord` et `Kworker`.

4.2.4 Corrélation par découpe de temps irréguliers

Si nous explorons la corrélation entre un ensemble contenant quelques anomalies et un ensemble à plusieurs milliers d'événements, nous obtenons un graphique de corrélation très peu lisible (c.f Figure 4.5).

Pour effectuer une analyse plus représentative, nous utilisons les données sur les anomalies comme point de départ pour l'analyse et considérons les intervalles de temps aux alentours. Nous considérons les densités de points du deuxième ensemble dans les intervalles ainsi obtenus. La figure 4.6 montre un exemple de ce découpage avec en bleu les zones aux alentours des anomalies détectées. Ces intervalles sont définis selon un paramètre delta : $anomalie \pm delta$. La valeur de *delta* est déterminée par l'utilisateur.

4.2.5 Algorithmes

L'algorithme 1 présente la méthode de calcul de corrélation avec découpage en temps régulier.

L'algorithme 2 présente la méthode de calcul de corrélation avec découpage en temps non-régulier.

La complexité globale de l'algorithme est en $O(n \times m)$, ou n est le nombre d'éléments de la plus grande liste et m le nombre d'éléments de l'autre liste. Le filtrage d'une liste se fait en $O(n)$. Pour l'algorithme 1 le découpage de l'espace se

Data:
L1 : liste des événements de type 1 ;
L2 : liste des événements de type 2 ;
Result:
Cor : Coefficient de corrélation linéaire ;

forall *l in L1, L2* **do**
  if *on doit considérer les événements anormaux* **then**
    if *l contient des états* **then**
      Calculer la durée moyenne;
      Filtrer les événements anormaux selon durée;
    else
      Calculer la durée moyenne entre deux occurrences;
      Filtrer les événements anormaux selon cette durée;
    end
  end
end
Prendre comme espace de temps l'union des espaces temps des deux listes;
Découper l'espace de temps en N tranches égales avec $N = sqrt(L1.size + L2.size)$;
Compter le nombre d'éléments de L1 par tranche de temps et sauver cette série dans S1;
Compter le nombre d'éléments de L2 par tranche de temps et sauver cette série dans S2;
Retourner le coefficient de corrélation linéaire Cor entre S1 et S2;

Algorithm 1: Algorithme de corrélation avec découpage régulier.

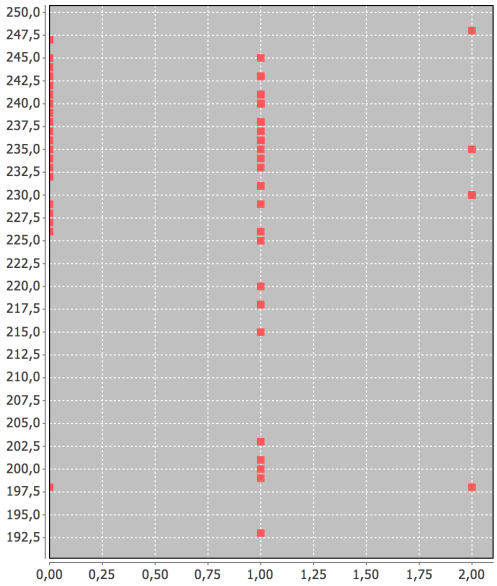


FIGURE 4.5 – Corrélation peu lisible due à une grande différence du nombre d'événements entre les deux ensembles.

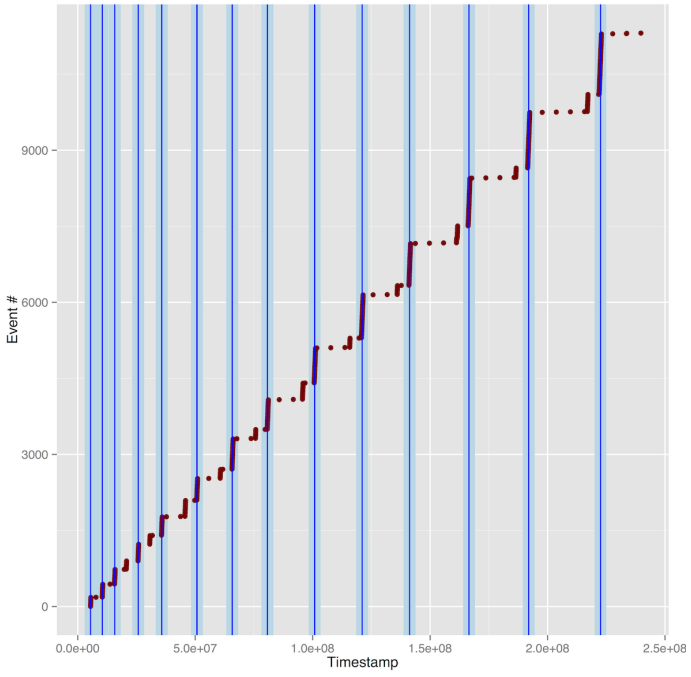


FIGURE 4.6 – En bleu, zones aux alentours des anomalies, elles-mêmes représentées par des barres verticales

fait en $O(1)$, et pour l'l'algorithme 2 en $O(n)$. Compter le nombre d'éléments par liste est en $O(n)$ et $O(m)$. Le calcul du coefficient de corrélation est le produit des variances par la covariance, sa complexité dépend donc du calcul de la covariance ayant une complexité de $O(n \times m)$.

Data:
L1 : liste des éléments de type 1 ;
L2 : liste des éléments de type 2 ;
Delta : intervalle définissant la proximité pour un événement à problème ;
Result:
Cor : Coefficient de corrélation linéaire ;

forall *l in L1, L2* **do**
  if *on doit considérer les événements anormaux* **then**
    if *l contient des états* **then**
      Calculer la durée moyenne;
      Filtrer les événements anormaux selon durée;
    else
      Calculer la durée moyenne entre deux occurrences;
      Filtrer les événements anormaux selon cette durée;
    end
  end
end
Prendre comme espace de temps l'union des espaces temps des deux listes;
E1 = la plus petite série entre L1 et L2;
E2 = la plus grande série entre L1 et L2;
Utiliser E1 et Delta pour découper l'espace de temps en deux sous ensembles disjoints :
- tous les points à une distance inférieure à Delta des points de E1,
- tous les points à une distance supérieure à Delta des points de E1;
Compter le nombre d'éléments de L1 par tranche de temps et sauver cette série dans S1;
Compter le nombre d'éléments de L2 par tranche de temps et sauver cette série dans S2;
Retourner le coefficient de corrélation linéaire Cor entre S1 et S2;

Algorithm 2: Algorithme de corrélation avec histogramme non-régulier.

4.3 Corrélation temporelle : implémentation dans Framesoc

Nous avons implémenté les deux méthodes présentées dans la section précédente au sein de l'infrastructure Framesoc, développées dans le cadre du projet SoC-TRACE. Nous exposons succinctement l'infrastructure Framesoc dans une première partie. Nous présenterons ensuite les fonctionnalités de l'outil puis son interface graphique.

4.3.1 Place de l'outil au sein de Framesoc

La figure 4.7 représente l'architecture logicielle de Framesoc. Les trois couches de l'architecture de Framesoc concernent respectivement le stockage des traces (`MultiTrace`

DB), l'accès aux données (`SoC-TRACE Library`) et les outils d'analyse (`GUI & Tools`). La couche de stockage implémente le modèle de représentation des traces d'exécution. Notre proposition d'enrichissement du modèle et d'intégration des notions d'*état*, *lien*, *événement ponctuel* et *valeur* a été intégrée à ce niveau-là. L'implantation actuelle est faite au sein d'une base de données relationnelle (SQLite [1]) mais pourrait être basée sur un autre support de stockage comme, par exemple, un système de fichiers. L'accès aux données se fait via une bibliothèque fourni par Framesoc qui propose les fonctions de base de manipulation de données comme la lecture, l'écriture ou le filtrage. Les outils Framesoc se placent au sommet de cette architecture.

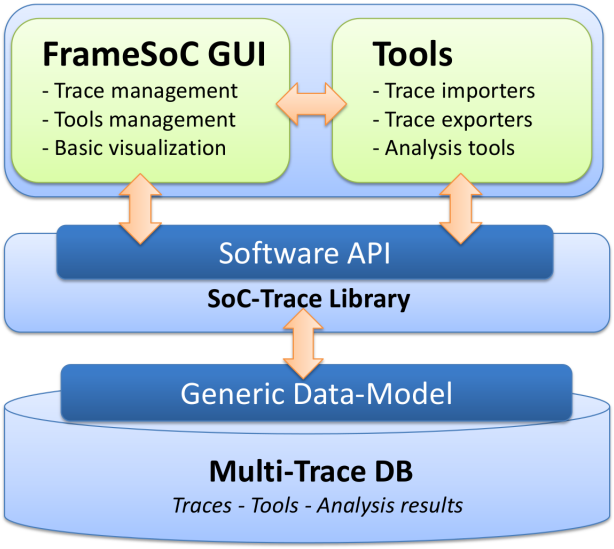


FIGURE 4.7 – Architecture logicielle de Framesoc

Framesoc ainsi que les outils sont implémentés en Java et sont des plugins Eclipse. En termes de règles de programmation, les outils Framesoc doivent étendre une classe abstraite qui sert de lien entre l'infrastructure et les outils. De cette manière, l'infrastructure connaît les outils lancés et leur fournit des facilités d'accès à la base des traces et de visualisation. Plus de détails sur l'implémentation sont disponibles dans [62].

Pour intégrer nos traitements de détection d'anomalies et de calcul de corrélation temporelle entre événements, nous avons développé un outil Framesoc spécifique. Notre outil d'analyse représente 8 classes Java pour un peu plus de 1000 lignes de code.

4.3.2 Interface fonctionnelle

L'outil que nous proposons permet d'obtenir de deux façons une corrélation entre deux types d'événements : la première de manière visuelle à l'aide de graphiques, la seconde en donnant le coefficient de corrélation linéaire. Les données à fournir en

1. www.sqlite.org

entrée sont la trace que l'on souhaite analyser, ainsi que les deux séries d'événements que l'on veut corréler. Ces dernières sont définis par leur type et leur producteur. Il est possible de demander à filtrer les éléments anormaux d'une série d'événements. Il faut aussi sélectionner laquelle des deux méthodes présentées en section 4.2 (nuage de points ou histogramme) il faut utiliser pour la représentation. Après le calcul, l'outil affichera en résultat la répartition des événements selon le temps sur un premier graphique. Sur un second, l'histogramme de densité de ces événements selon les tranches de temps dans lequel l'espace a été découpé. Un résultat numérique est aussi affiché et correspond au coefficient de corrélation linéaire présenté en section 4.1.

4.3.3 Interface graphique

La figure 4.8 montre un exemple d'entrées d'analyse.

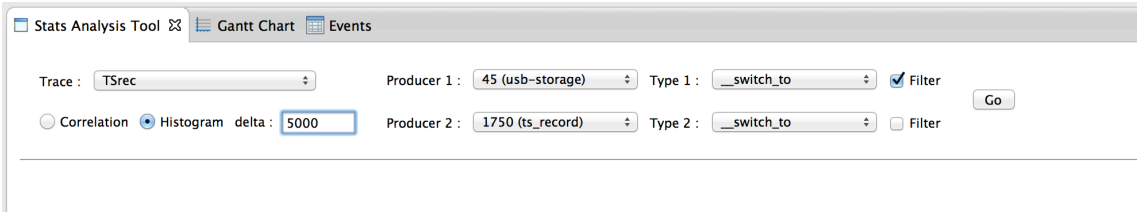


FIGURE 4.8 – Exemple de paramètres d'entrée

Il faut dans un premier temps sélectionner la trace que l'on veut analyser. L'outil propose de choisir entre toutes les traces présentent dans la base de donnée de Framesoc. L'outil propose ensuite les types d'événements correspondant à la trace sélectionnée. Il est possible de choisir de filtrer ou non les événements anormaux de l'une ou des deux séries. Un bouton radio permet de définir l'une ou l'autre méthode de calcul. Il est aussi nécessaire de rentrer une valeur dans la case "delta" (en μs), celle-ci sera utilisée lors du calcul avec l'histogramme non régulier pour définir les éléments "aux alentours" des points de la plus petite série. Le bouton "GO" permet de lancer l'analyse avec en paramètre les données sélectionnées.

Une fois les calculs effectués, l'outil affiche deux graphiques comme montrés sur les Figures 4.9 et 4.10. Le graphique de gauche représente les événements pour chaque série. Un point représente un événement. En abscisse on a l'estampille de l'événement et en ordonnée le numéro de l'événement. Cette représentation permet de palier le manque de visibilité si plusieurs événements sont intervenus de manière trop proche dans le temps. On retrouve le même graphique que dans la Figure 4.4 qui représente le cas *TSrecord*.

Le graphique de droite peu représenter, comme montré sur la Figure 4.9 le nuage de points de l'histogramme régulier. On voit que, même si les événements sont corrélés, cet affichage n'est pas toujours le plus réussi pour donner une intuition à l'analyste. Il est donc possible d'utiliser un autre type d'affichage (Figure 4.10) pour montrer l'histogramme des densités des événements pour la méthode avec histogramme non régulier. Dans cet exemple, on retrouve dans la figure de l'histogramme

les intervalles déterminés comme dans la figure 4.6. En bleu est représenté le nombre d'éléments de la série dans les intervalles aux alentours des anomalies, en rouge le nombre d'éléments entre ces intervalles.

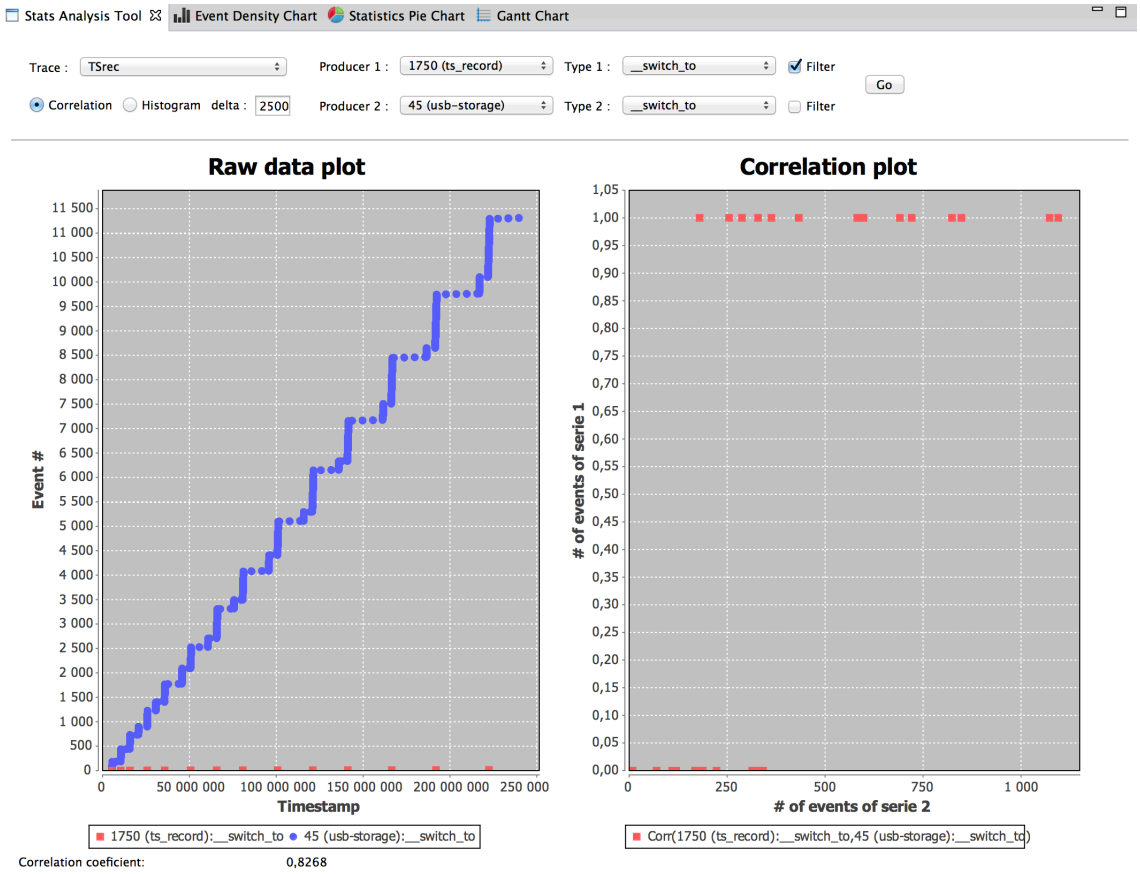


FIGURE 4.9 – Exemple de résultat par corrélation avec découpage régulier

4.3.4 Évaluation

Nous avons évalué notre outil en utilisant le cas TSrecord présenté en section 4.2.1. Nous avons utilisé Eclipse en version 4.3 et java 1.7. La machine que nous avons utilisée est un MacBook Pro avec un CPU Intel i7 quadri-core 2.6 GHz, 16Go de RAM, et utilisant macOS 10.9. La trace considérée contient au total 814 289 événements. Les deux séries que nous comparons contiennent 23 408 et 11 312 événements. Il faut environs 9,2 secondes pour récupérer ces deux séries d'événements. Nous filtrons la première série, cela prend environ 3 secondes. Le calcul du coefficient de corrélation prend environ 300 ms. Sur cet exemple, le temps total de l'analyse est d'environ 12 à 14 secondes selon la méthode utilisée.

Dans les deux cas, la mesure de corrélation obtenue nous indique une forte corrélation entre ces deux séries. Dans le cas de l'histogramme régulier, la mesure est unique car elle ne dépend que des deux séries données en entrée. Pour l'histogramme non-régulier, l'utilisation d'un delta variable définit par l'utilisateur nécessite plusieurs calculs afin de trouver une valeur faisant ressortir cette corrélation.

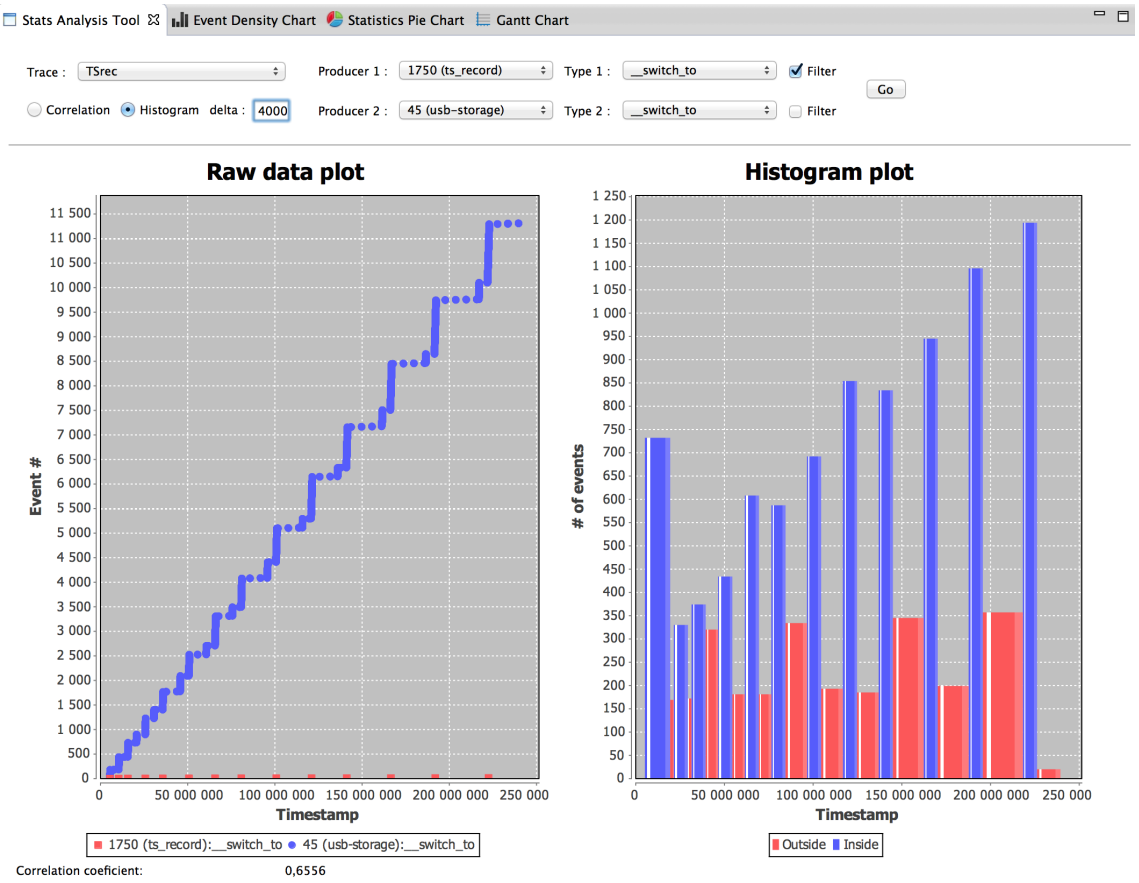


FIGURE 4.10 – Exemple de résultat par corrélation avec découpage non régulier

4.4 Conclusion

Dans ce chapitre, nous avons validé l'intérêt d'utiliser de notre format de trace enrichi en sémantique pour l'analyse. Cette modélisation de la trace contenant la sémantique nous a permis de représenter de nouveaux concepts dans la trace comme, par exemple, les transitions entre processus via des événements de liens, ou la latence de certains appels de fonctions via des événements de durée. Cette représentation enrichie nous a permis aussi de concevoir des analyses plus évoluées. Nous avons défini des traitements pour détecter les événements anormaux, utilisant des traitements statistiques comme, par exemple, l'intervalle de confiance. Nous avons aussi utilisé un traitement par corrélation entre événements via des histogrammes et des corrélations afin de retrouver les causes de ces anomalies détectées. Cette validation a été faite sur des cas d'usages réels, fournis par la société STMicroelectronics. Les problèmes observés sur les traces, ainsi que leurs causes, ont été validés avec les équipes de développement de STMicroelectronics.

Chapitre 5

Contribution au projet Vistrails

Dans ce chapitre nous présentons notre contribution à l'outil Vistrails [27, 83]. Cette contribution est issue d'une collaboration avec le laboratoire ViDA (*Visualization and Data Analysis*) de l'Université de New-York (NYU). Durant un séjour de trois mois (de Novembre 2015 à Janvier 2016), nous avons collaboré avec Prof. Juliana Freire, responsable du laboratoire et Rémi Rampin, développeur principal de Vistrails. L'objectif était d'intégrer dans VisTrails des mécanismes facilitant l'analyse de traces d'exécution.

Après une brève présentation de VisTrails, nous abordons son utilisation dans un contexte d'analyse de traces. Nous discutons les limitations rencontrées et introduisons les modifications faites sur l'outil. Nous concluons avec un exemple montrant les apports et l'utilité de nos modifications.

5.1 Présentation de VisTrails

VisTrails [83] est un outil de workflow développé au sein de l'Université de New-York depuis 2007. VisTrails compte de nombreux utilisateurs, son développement est toujours actif et supporté par une forte communauté. VisTrails ne considère pas uniquement l'aspect d'automatisation d'un processus mais porte un intérêt particulier aux aspects d'analyse de données. Notamment, l'outil se spécialise dans la simulation, l'exploration et la visualisation de données. Il est utilisé dans des domaines tels que la bio-informatique, la météorologie ou la physique.

Nous avons choisi VisTrails comme outil d'expérimentation pour deux raisons principales. Tout d'abord, c'est un outil connu et utilisé dans différentes communautés scientifiques. Il respecte donc notre recherche de *généricité*. D'autre part, le modèle de programmation des workflows VisTrails est très proche de celui que nous proposons. Nous avons donc considéré que cette base commune faciliterait l'implémentation des nos propositions de gestion et d'analyse de traces d'exécution.

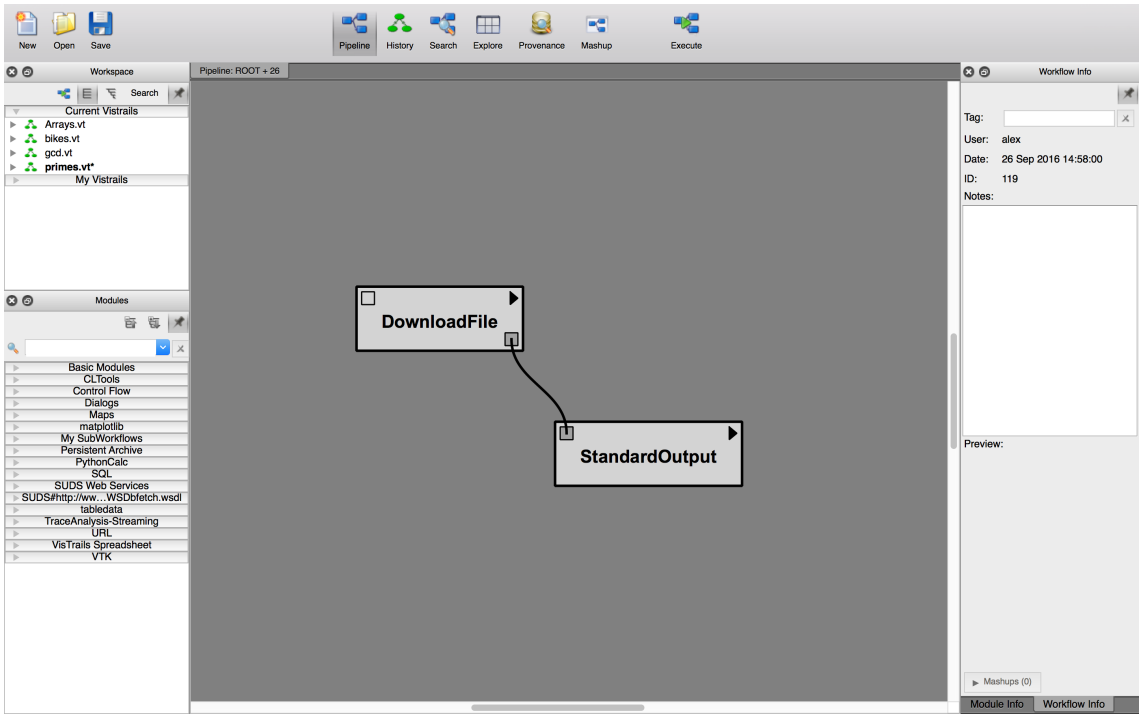


FIGURE 5.1 – Interface graphique de l'outil VisTrails.

VisTrails facilite la construction de workflow en fournissant aux utilisateurs une interface graphique. Comme montré sur la Figure 5.1, le workflow est construit en manipulant des modules et des liens de dépendance. Dans l'exemple, les modules correspondent à des traitements et le lien indique l'échange de données. Le premier module télécharge un fichier et le transmet au deuxième module qui l'affiche sur la sortie standard.

Les principales fonctionnalités de VisTrails sont présentées dans ce qui suit.

Gestion de modules

VisTrails propose des modules prédéfinis, ainsi que des mécanismes pour concevoir des modules personnalisés.

Parmi les modules prédéfinis, il existe des modules VisTrails pour représenter des données comme les entiers ou les chaînes de caractères. Dans ces cas-là, les modules fournissent les opérations de base sur ces données (la conversion d'un entier en chaîne de caractères, concaténation de chaînes de caractères, etc.). D'autres modules capturent les structures de contrôle d'exécution comme les tests (*if*) ou les boucles (*for* et *while*). Il existe des modules de travail avec les fichiers ou avec les bases de données. Il existe également des modules pour l'interaction avec l'utilisateur (boîtes de dialogue) ou des modules permettant l'utilisation de bibliothèques externes (par exemple Google Maps, Matplotlib, ou VTK).

VisTrails permet la création de modules personnalisés. Pour cela, il fournit une API qui respecte un modèle de programmation très proche des modèles orientés composant, et donc de notre proposition (Chapitre 3). Les modules VisTrails doivent

spécifier leurs ports d'entrée et de sortie, ainsi qu'implémenter une fonction spéciale `compute`. Les ports d'entrée et de sortie représentent respectivement les données dont le module a besoin et les données qu'il produit. La fonction `compute` définit le comportement du module. La programmation d'un module se fait en Python, le langage dans lequel VisTrails est écrit, comme le montre la Figure 5.2. Dans cet exemple, le module calcule le successeur d'un entier.

```
1 class Successor(Module):
2     _input_ports = [IPort("number", "basic:Integer")]
3     _output_ports = [OPort("successor", "basic:Integer")]
4
5     def compute(self):
6         n = self.get_input('number')
7         succ = n + 1
8         self.set_output('successor', succ)
```

FIGURE 5.2 – Exemple de module VisTrails

Création du workflow et exécution

Un workflow est construit en interconnectant différents modules à travers leurs ports. L'interface graphique de l'outil permet à l'utilisateur de choisir dans une liste de modules disponibles, de glisser un module dans la zone graphique du workflow et de relier deux modules pour établir les liens. Une fois le workflow entièrement construit, un bouton de l'interface graphique permet de lancer son exécution.

L'exécution du workflow est assurée par un seul fil d'exécution (un processus Python) et est par conséquent séquentielle. Un module qui est démarré, est exécuté sans interruption jusqu'à sa terminaison, si il n'a pas besoin de données en entrée. Si il a besoin de données, l'exécution est interrompue et l'exécution du module dont dépend la production de données est lancée. Le fil d'exécution passe donc du premier module au deuxième, un peu comme un appel de fonction. Ce procédé peut être répété plusieurs fois si d'autres dépendances de données doivent être satisfaites. Quand les données sont produites, le premier module résume son exécution et termine. Ce mécanisme est présenté plus en détail dans la suite du chapitre.

Gestion d'historique

VisTrails intègre un système de gestion de versions de workflow similaires aux systèmes de *versioning* de fichiers (type *git*). A chaque modification du workflow l'outil enregistre ces modifications : l'ajout ou la suppression d'un module ou d'une connexion, le fait de définir une valeur constante pour un port, et même le fait de déplacer un module dans l'interface pour améliorer sa lisibilité. Ces modifications sont présentées sous forme d'arbre (Figure 5.3) et il est possible de revenir à une version précédente du workflow. Il est donc possible de travailler avec plusieurs versions d'un même workflow, tout en passant d'une version à une autre facilement.

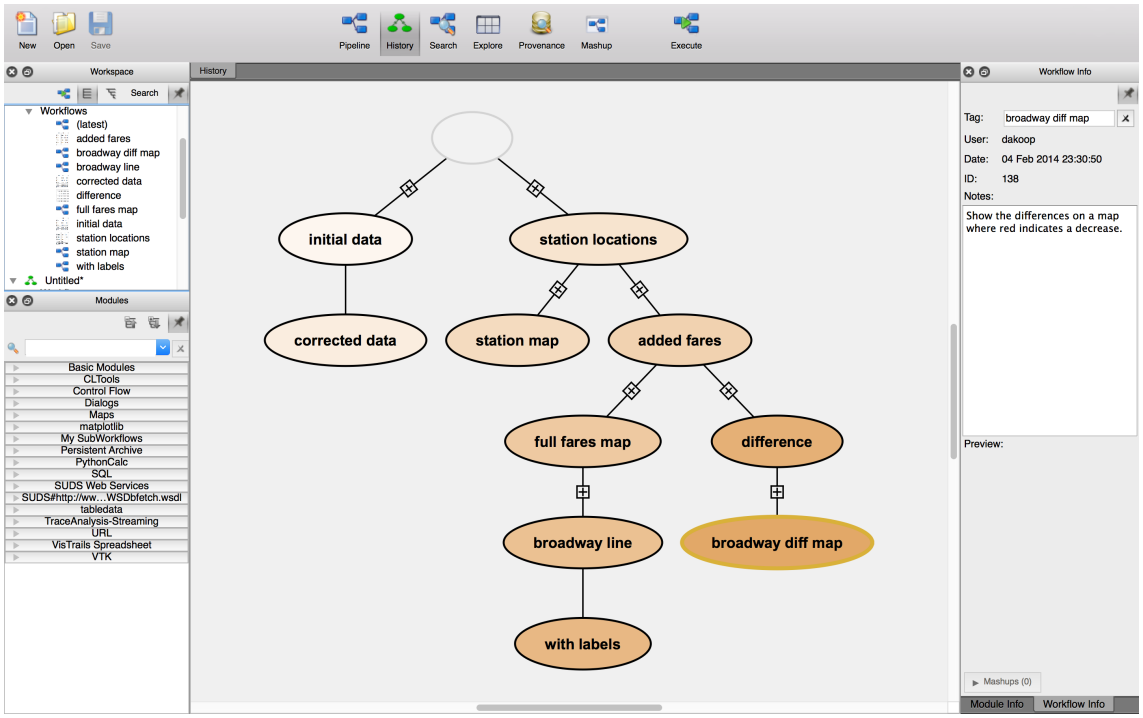


FIGURE 5.3 – Historique d'un workflow dans VisTrails.

Provenance et cache de données

VisTrails optimise l'exécution des workflows en intégrant un mécanisme de cache qui permet de réutiliser un résultat déjà calculé. Ce cache permet de ne pas exécuter à nouveau un module si ses données d'entrée n'ont pas changé par rapport à la dernière exécution. VisTrails offre aussi des informations sur la provenance des données. Il est possible de savoir si un module a été exécuté ou si les données produites sont récupérées depuis le cache, ainsi que la date de sa dernière exécution. Les informations de provenance pour le workflow complet renseignent sur l'auteur, la date de création, et les dates d'exécutions. Les informations de provenance peuvent être assimilées à des méta-données sur le workflow et les modules, permettant la traçabilité des données produites.

5.2 VisTrails et l'analyse de traces

Nous avons expérimenté avec VisTrails pour construire des workflows d'analyse de traces d'exécution. Dans la suite, nous décrivons notre cheminement qui a identifié les limitations de l'outil, et nous a amené à proposer une nouvelle conception intégrant le mécanisme de *streaming* de données.

5.2.1 Définition de modules spécifiques dans VisTrails

Nous avons, dans un premier temps, essayé d'utiliser l'API VisTrails pour définir des modules pour l'analyse de traces d'exécution. Nous avons donc défini des modules qui prennent en donnée d'entrée une trace d'exécution, effectuent un calcul et

produisent un résultat. Pour modéliser une trace d'exécution en termes de structures de données VisTrails, nous avons procédé comme suit.

Un événement est modélisé par un tuple (type `Event`) contenant tous les champs définis dans le modèle Framesoc (Section 3.1). Le tuple contient, entre autres, l'estampille de l'événement, son type et son producteur. Une trace étant une séquence d'événements, nous l'avons modélisée comme un tableau d'événements. Pour ce faire, nous avons utilisé la notion de *profondeur de port* de VisTrails. La *profondeur* d'un port représente la dimension des données manipulées. Ainsi, un port de type entier et de *profondeur zéro* indique que la donnée sur ce port est de type entier. Un port de type entier et de *profondeur un* indique une donnée de type liste d'entiers. Pour nos besoins, la trace est donc manipulée à travers des ports d'entrée et de sortie de type `Event` de *profondeur un*.

La Figure 5.4 montre un exemple de module VisTrails qui filtre les événements d'une trace selon leur type. Le module est défini comme une classe Python, qui hérite d'une classe prédéfinie `Module`. La liste des ports d'entrée est définie dans la variable `_input_ports`. Chaque port d'entrée est construit en utilisant l'interface `IPort`. De manière analogue, la définition des ports de sortie se fait via la variable `_output_ports` en définissant des `OPort`. La lecture et l'écriture de données sur les ports se fait via les fonctions `get_input` et `set_output`.

Le corps de l'exécution du module est défini dans la fonction `compute` qui décrit les calculs faits par le module. Dans cet exemple, le module va récupérer sur un port une liste d'événements (port `events`) et sur un autre un type d'événements (port `type_name`). L'exécution de la fonction `compute` consiste à séparer les événements selon le type donné dans deux tableaux, respectivement `matched_events` et `excluded_events`. Cette opération se fait dans la boucle `for` du module. Une fois ce tri fait sur tous les événements, les données sont envoyées sur les deux ports de sortie, un pour chaque nouvelle liste.

Le problème avec cette solution est que la donnée passée en entrée du module (ici le tableau d'événements) doit être chargée en mémoire. Or, dans notre contexte, cela veut dire charger une trace entière en mémoire ce qui est impossible avec des traces volumineuses. En effet, nos traces pouvant atteindre plusieurs dizaines de Gigaoctets, la mémoire sature et l'utilisation de l'outil devient impossible. Nous avons donc dû utiliser une autre méthode, afin de supporter les grosses traces.

5.2.2 Analyse utilisant une base de données

Pour nous affranchir de la limitation des données en mémoire, lors de notre deuxième approche nous avons utilisé une base de données pour stocker la trace. Le choix de la base de données à également été influencé par le fait que Framesoc utilise une base de données pour stocker les traces d'exécution. Nous avons pu, par conséquent, utiliser la même base avec les mêmes schémas et travailler sur les traces déjà traitées (importées) pour travailler au sein de Framesoc. L'analyse se fait en utilisant les modules de requêtes SQL présents dans VisTrails.

La Figure 5.5 montre le workflow décrivant un processus d'analyse via une base de données. Par soucis de clarté, nous avons omis les modules de représentations de

```
class EventFilter(Module):
    _input_ports = [IPort("events", "Event", depth=1),
                    IPort("type_name", "basic:String")]
    _output_ports = [OPort("matched_events", "Event", depth=1),
                     OPort("excluded_events", "Event", depth=1)]

    def compute(self):
        events = self.get_input('events')
        type_name = self.get_input('type_name')

        matched_events = []
        excluded_events = []

        for event in events:
            if event.e_type == type_name:
                matched_events.append(event)
            else:
                excluded_events.append(event)

        self.set_output('matched_events', matched_events)
        self.set_output('excluded_events', excluded_events)
```

FIGURE 5.4 – Module de manipulation de trace dans VisTrails

résultats. Le premier module du workflow (en haut à gauche sur l'image) est un module représentant une donnée, ici le fichier de base de données. Il est connecté à un second module servant à ouvrir une connexion sur cette base de données (*DBConnection*). La référence de connexion est ensuite transférée aux différents modules de requêtes SQL. L'analyse de la trace se compose de plusieurs requêtes, toutes ayant besoin de la référence de connexion, ce qui explique la structure en "étoile" que l'on observe sur la figure.

L'utilisation de l'outil VisTrails pour l'analyse de traces de cette manière est très vite limitative pour les raisons suivantes. Tout d'abord, l'utilisation d'une base de données nous contraint, lors du premier accès, à lire entièrement la trace afin de l'importer au sein de la base. Bien que cette phase soit utile, notamment pour l'ajout de sémantique, cette première phase d'importation est coûteuse en temps, surtout dans le cas de grosses traces.

Ensuite, la performance de l'analyse dépend fortement des performances des requêtes SQL. Si celles-ci ne sont pas optimisées, l'analyse va avoir des performances dégradées. Optimiser les requêtes SQL demande une expertise technique qui n'est pas à la portée de tout analyste de traces. En plus, devoir faire de telles optimisations est en contradiction avec l'idée de proposer une infrastructure d'analyse permettant de se concentrer seulement sur l'analyse en s'abstrayant des contraintes techniques sous-jacentes.

Enfin, la structure en "étoile" autour du module de connexion à la base de données ne permet pas de définir un ordre de précédence. Pour imposer un ordre particulier, il faut alors utiliser des astuces. Seulement, ces moyens détournés complexifient inutilement le workflow et utilisent des mécanismes de VisTrails qui ne

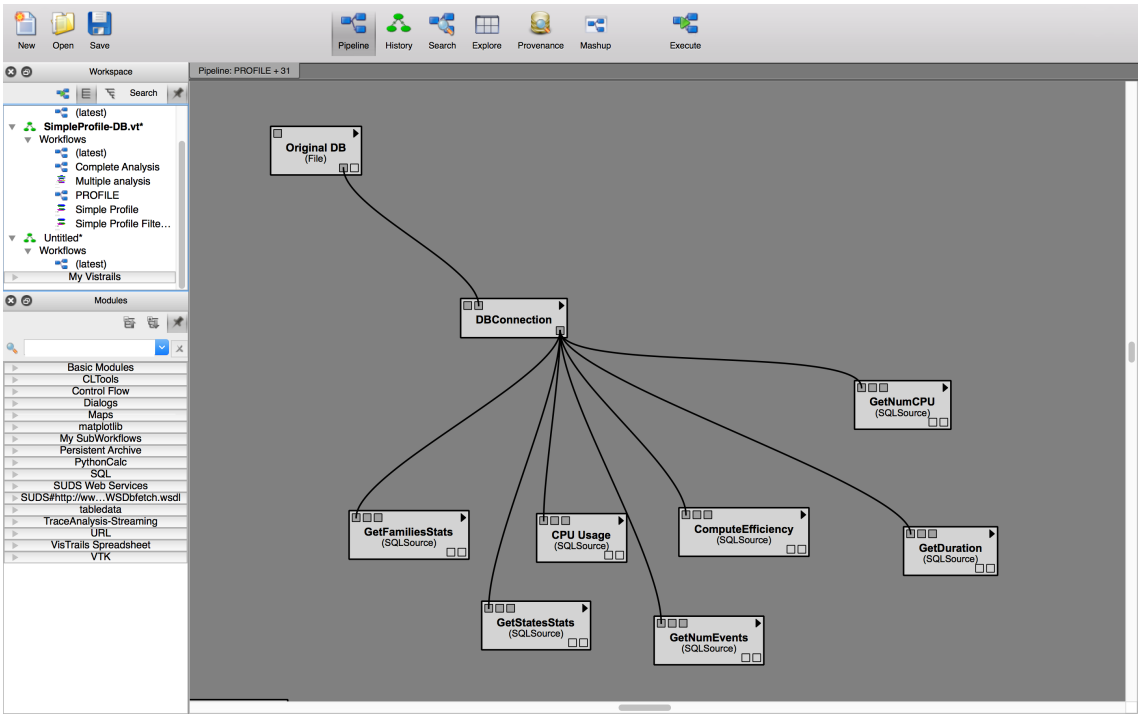


FIGURE 5.5 – Workflow décrivant l'analyse via une base de données dans VisTrails.

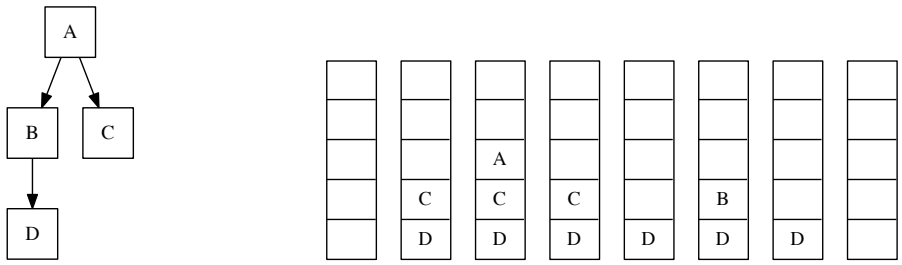
sont pas prévus à cet effet.

5.3 Introduction d'un mécanisme de *streaming*

Notre expérimentation avec VisTrails nous a permis de constater que l'outil n'est pas adapté à l'analyse de traces. En effet, VisTrails a été conçu pour des workflows où les modules font beaucoup de calculs sans transférer de gros volumes de données. Le modèle d'exécution que nous recherchons pour l'analyse de traces est plutôt organisé en termes de gros transferts de données traitées par des opérations de granularité fine (filtres, agrégations, statistiques, etc.).

La solution que nous proposons consiste à découper les traces et à envoyer leurs données (les événements) en continu sur les liens de dépendance entre modules. Par exemple, au lieu de transférer plusieurs milliers d'événements en une fois d'un module à un autre, les événements pourraient être envoyés par paquets de mille. L'analyse, dans ce cas, se fait en itérant sur les paquets de données et les modules se retrouvent à traiter des données partielles. Ceci permet de proposer en continu des résultats partiels et fournit ainsi un retour à l'utilisateur, comme préconisé par les méthodes de *visual analytics* [37, 45].

Pour rendre possible ce genre de traitement au sein de VisTrails, nous avons proposé d'y intégrer un mécanisme de *streaming*. Nous présentons dans la suite la manière dont nous avons procédé, sachant que l'ajout de ce mécanisme nécessite la réécriture complète du moteur d'exécution de workflows de VisTrails.



(a) Workflow à exécuter (b) Évolution de la pile d'exécution des modules

FIGURE 5.6 – Exemple d'exécution d'un workflow

5.3.1 Le mécanisme d'exécution de *workflow* dans VisTrails

Comme mentionné précédemment, le moteur d'exécution actuel de VisTrails exécute le *workflow* module par module. Le moteur isole et commence l'exécution par les modules terminaux d'un workflow. Ce sont les modules *puits* (*sinks*) du graphe associé au workflow i.e les modules qui ne produisent pas de données. Lorsqu'un module utilise une donnée depuis un port d'entrée, deux cas de figure peuvent se présenter. Si la donnée est déjà disponible, le module continue normalement son exécution. Sinon l'exécution du module dont dépend la production de la donnée est exécuté. Une fois que celui-ci est fini, l'exécution du premier module reprend au point où il s'était arrêté. Ce processus est effectué pour l'exécution de chaque module.

La Figure 5.6 montre un exemple d'exécution d'un workflow fictif de quatre modules A, B, C, et D. Pour ordonnancer l'exécution des modules, VisTrails utilise une pile. Dans cette pile sont ajoutés les modules C et D qui sont les modules *puits* de ce workflow. L'exécution commence par le premier module dans la pile i.e. le module C. C a besoin d'une donnée venant de A, qui n'est pas encore disponible. Le moteur empile donc A et lance son exécution. Une fois l'exécution de A fini, A est dépilé et le moteur continue avec l'exécution de C qui se termine normalement. La pile contenant encore un module à exécuter, D, le moteur continue avec son exécution. Suivant le même principe, D est interrompu, B est empilé et exécuté. Le module B s'exécute normalement car la donnée produite par A est déjà disponible. Lorsque B est fini, D continue et se termine.

Ce modèle d'exécution ne permet pas de parallélisme et exige la production "en une fois" d'une donnée. Il est donc impossible de l'utiliser pour créer une exécution dans laquelle deux modules travaillent en parallèle et s'échangent des données de manière *continue* en suivant le modèle producteur/consommateur.

5.3.2 Exécution *multi-thread* de *workflows*

Pour rendre possible le transfert et le traitement de données en continu, nous avons au départ mis en place un modèle d'exécution parallèle classique. Notamment,

il s'agit d'attribuer un fil d'exécution indépendant à chaque module, d'exécuter les modules de manière indépendante et de les faire communiquent à travers de files de messages. La synchronisation entre modules et la régulation des flux (des vitesses de production et de consommation) sont à gérer au niveau des files et de mécanismes bloquants d'envoi/réception.

Pour réaliser ceci, nous avons utilisé la possibilité de multi-threading de Python. Nous avons utilisé un thread par module et avons implémenté les liens de connexions à l'aide de tampons partagés. L'accès aux tampons se fait de manière synchronisée classique en utilisant les verrous et les conditions.

Le problème avec cette implémentation vient de contraintes propres au langage Python. En effet, Python utilise du *pseudo-parallélisme* [66] où le parallélisme physique de la machine utilisée ne peut être exploité. L'exécution des différents fils d'exécution est donc séquentielle.

Pour introduire du parallélisme avec la version actuelle de Python, il faut donc utiliser plusieurs processus. Or, un processus par module de workflow combiné avec un mécanisme de *streaming* transférant plusieurs giga-octets de données n'est pas envisageable à cause du coup des mécanismes de communication inter processus (IPC).

Nous avons exploré une autre solution qui consiste à utiliser plusieurs processus qui servent de support d'exécution d'un *workflow* afin de permettre du parallélisme. Nous avons dû dépasser les limitations venant des *threads* qui nous cantonnent à un seul espace d'adressage. Nous avons décidé de ne pas considérer les modules en tant qu'unités d'exécution et d'ordonnancement mais de considérer des unités plus fines. Nous avons donc mis en place un modèle d'exécution par tâches qui a deux aspects majeurs. D'une part, les tâches sont conçues de manière à décrire les différents étapes de traitements de données. Cette approche permet de réaliser, même dans un environnement séquentiel, un modèle de communication producteur/consommateur et donc du *streaming* de données. D'autre part, dans le contexte où plusieurs processus s'exécutent en parallèle, le modèle à base de tâches permet de faire de la répartition de charge entre les processus. Les détails de réalisation sont décrits dans ce qui suit.

5.3.3 Vers une exécution par tâches

Le modèle de tâches que nous proposons consiste à faire correspondre plusieurs tâches à un module de workflow. Les différentes tâches correspondent à des séquences de traitement de données (calcul) ou s'occupent des activités de transfert de données. Ainsi, l'exécution d'un module se traduit par l'instanciation et l'exécution de plusieurs tâches. Ces actions sont contrôlées par un ordonnanceur de tâches. Ceci est schématiquement représenté dans la Figure 5.7.

Modèle de programmation des modules

Dans notre modèle par tâches, le comportement d'un module n'est plus défini par une seule fonction `compute`, mais est découpé en plusieurs parties en utilisant cinq fonctions. Ce sont notamment :

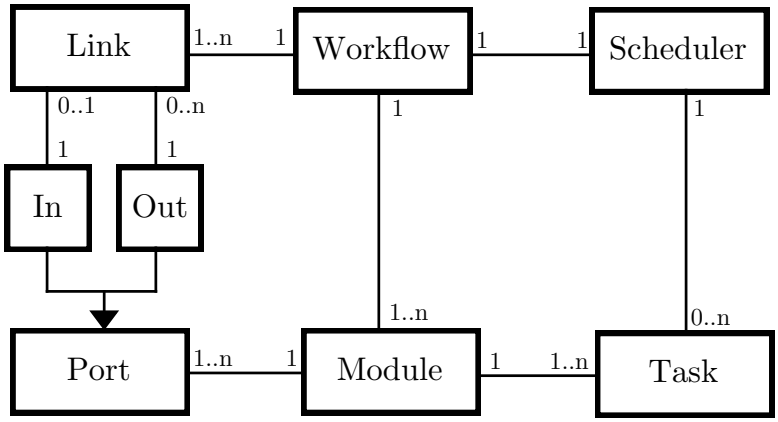


FIGURE 5.7 – Diagramme UML de la partie ordonnancement

— `start()` : Cette fonction est appelée par l'ordonnanceur de tâches lors du lancement d'un module et s'occupe typiquement des initialisations.
— `input(port, value)` : Cette fonction est appelée par l'ordonnanceur de tâches lorsqu'une donnée devient disponible sur un port d'entrée du module. Les paramètres *port* et *value* contiennent respectivement le port sur lequel la donnée arrive, et la valeur de celle-ci.
— `step()` : Cette fonction est utilisée pour les modules produisant des données de manière continue et sans dépendances d'autres modules (les modules initiaux du workflow). Cette fonction permet de définir une action qui sera exécutée de manière périodique tout au long de l'exécution du module. Par exemple, "lire X lignes du fichier et les envoyer sur le port de sortie".
— `input_end(port)` : Cette fonction est appelée lorsqu'un flux de données sur un des ports se termine, typiquement, lorsque le module produisant le flux de données se termine. Le paramètre permet de savoir de quel port il s'agit, et ainsi exécuter le traitant correspondant.
— `finish()` : Cette fonction est appelée avant la terminaison d'un module et peut, par exemple, servir à produire un résultat final.

Il est important de noter qu'un module n'est pas obligé d'implémenter toutes les cinq fonctions mais peut ne fournir qu'un sous-ensemble.

Les modules définissent l'ordre d'exécution des différentes fonctions et donc leur comportement en appelant les fonctions suivantes :

— `request_step()` : Cette fonction demande à ce que la fonction `step()` soit exécutée. L'appel est adressé à l'ordonnanceur qui déclenchera l'exécution de `step()` ultérieurement.
— `request_finish()` : De manière identique, cette fonction sert à demander l'exécution de la fonction `finish()` ultérieurement.
— `request_input(port)` : Cette fonction sert à demander la réception d'une donnée de manière asynchrone sur un port.
— `produce_output(port, value)` : Cette fonction produit de manière asynchrone une donnée sur un port.

La Figure 5.8 montre l'implémentation d'un filtre, similaire à celui présenté en

Figure 5.4, mais utilisant la nouvelle interface. Dans la fonction `start`, le module demande une donnée sur le port *producer_name*. L'ordonnanceur va donc exécuter le module qui doit produire cette donnée et, une fois celle-ci disponible, il va exécuter la fonction *input* de notre module (ligne 10). Le port concerné étant le port `type_name`, seule la partie correspondante du test `if` va être exécutée, à savoir l'enregistrement du type pour le filtre et la requête de données sur le port `events` (lignes 18 à 20). Si des données sont produites pour le port `events`, l'ordonnanceur va de nouveau exécuter la fonction `input`. Seulement cette fois, ce sera la partie correspondante au port `events` qui va être exécutée (lignes 11 à 17). L'exécution consiste à envoyer l'événement reçu par un des deux ports de sortie, puis demander à nouveau la réception d'un événement (ligne 17). Dans cet exemple, les fonctions `input_end` et `finish` ne sont pas implémentées et le module s'arrête une fois le flux d'événements terminé.

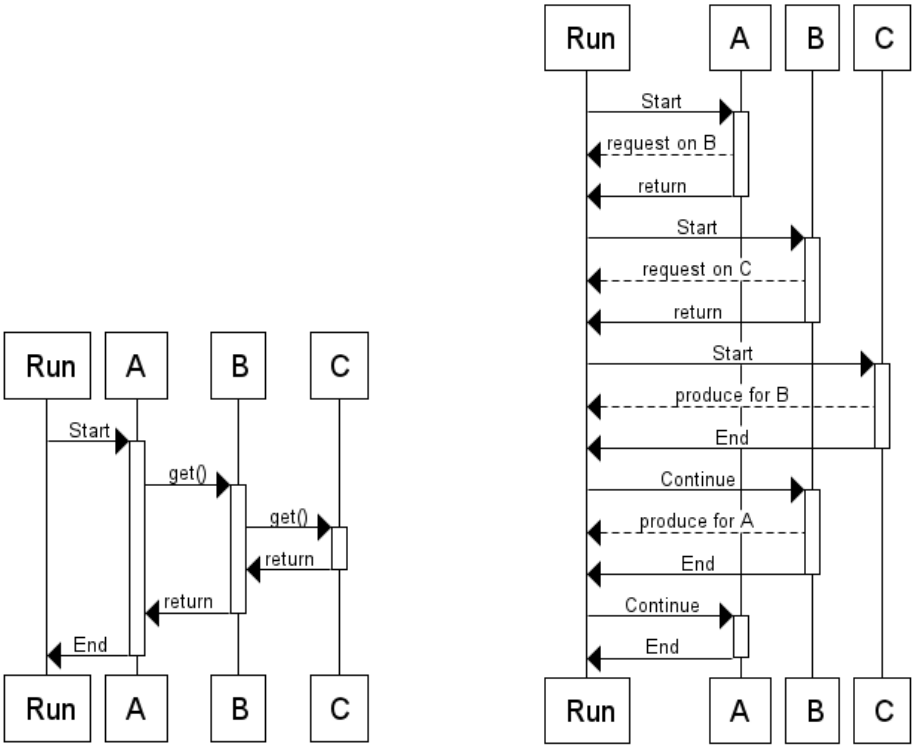
```
1  class EventFilter(Module):
2      input_ports = [IPort("events", "Event"),
3                     IPort("type_name")]
4      output_ports = [OPort("matched_events"),
5                      OPort("excluded_events")]
6
7      def start(self):
8          self.request_input("producer_name")
9
10     def input(self, port, value):
11         if port == "events":
12             event = value
13             if type_matcher == event.e_type:
14                 self.produce_output("matched_events", event)
15             else:
16                 self.produce_output("excluded_events", event)
17             self.request_input("events")
18         elif port == "type_name":
19             self.type_matcher = value
20             self.request_input("events")
```

FIGURE 5.8 – Exemple de module avec la nouvelle interface de programmation

Mécanisme de push/pull

Du point de vue conceptuel, notre moteur d'exécution utilise un mécanisme de *pull/push* [84]. Un module n'accède pas directement aux données mais demande au moteur d'exécution une donnée sur un port via la fonction `request_input(port)` (*pull*). Plusieurs données peuvent être demandées en appelant cette fonction sur plusieurs ports. Ensuite, le moteur lance l'exécution des modules en charge de produire les données demandées. Les modules produisent les données et notifient le moteur via la fonction `produce_output(port, value)` (*push*). Enfin le moteur exécute la fonction `input(port, value)` du premier module ayant demandé cette donnée.

La Figure 5.9 montre les différences entre les mécanismes de *pull* et *push/pull* sur un exemple fictif avec trois modules A, B et C en chaîne, où C produit une



(a) Modèle d'exécution pull (b) Modèle d'exécution pull/push

FIGURE 5.9 – Exemple d'exécution d'un workflow

donnée pour B, qui produit à son tour une donnée pour A. Sur cette figure "Run" correspond au moteur d'exécution. Dans le modèle *pull*, on remarque que l'exécution des modules se fait de manière imbriquée lors de l'appel aux fonction `get` pour récupérer des données. Dans le modèle *pull/push*, le moteur pousse (*push*) l'exécution des modules lorsque les données sont demandées (*pull*). Dans le cas où plusieurs données sont demandées, les modules peuvent produire les données en parallèle. Le modèle *pull/push* permet donc aussi d'éviter une pile d'appels de fonction trop conséquente.

Tâches et ordonnanceur

Les cinq fonctions `start`, `input`, `step`, `input_end`, et `finish` sont représentées par cinq différentes tâches : `StartTask`, `InputTask`, `StepTask`, `InputEndTask`, `FinishTask` (Figure 5.10). Quand un module effectue un appel à une des fonctions `request_step`, `request_finish`, `request_input` et `produce_output`), l'ordonnanceur crée une tâche correspondante et l'ajoute à la liste des tâches à ordonnancer.

Par exemple, lorsqu'un module appelle la fonction `produce_output`, l'ordonnanceur est notifié de la production d'une donnée. Il utilise la représentation du workflow pour déterminer les modules qui vont consommer cette donnée et créer une tâche `InputTask` par module. Une fois une de ces tâches ordonnancée, c'est la fonction `input` du module associé qui va être exécutée.

L'ordonnancement consiste, de manière analogue à l'ancien mécanisme de VisTrails, à considérer dans un premier temps les modules *puits* du workflow. Une tâche

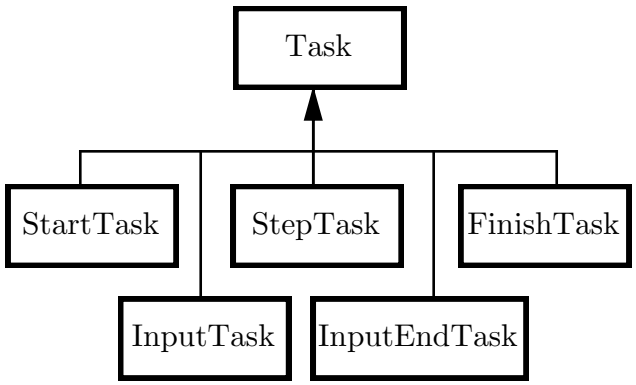


FIGURE 5.10 – Diagramme UML des tâches

`StartTask` est alors créée pour chaque module. Les tâches suivantes sont créées selon les notifications reçues des modules grâces aux fonctions d'interactions.

Les modules suivent l'automate à états présenté en Figure 5.11.

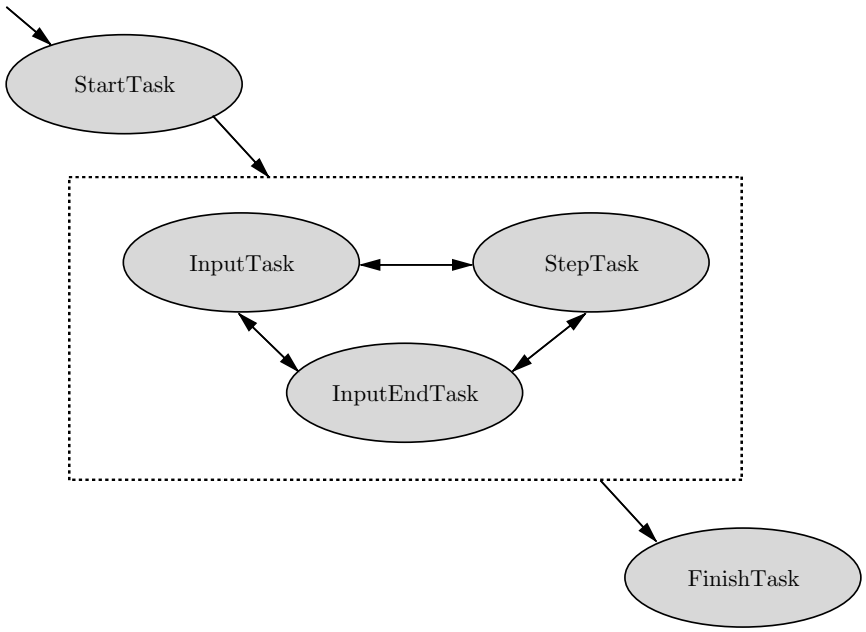


FIGURE 5.11 – Automate à états représentant l'exécution d'un module

5.4 Validation du nouveau modèle d'exécution

Le moteur d'exécution des modules est au cœur de l'outil VisTrails et est étroitement intégré avec ses autres mécanismes, à savoir la gestion de l'historique, l'exploration des données, le gestion de cache et de provenance. Afin de valider notre approche, nous avons implémenté le nouveau moteur d'exécution de manière auto-

nome. Ceci nous permet de valider ce nouveau modèle d'exécution avec les mécanismes de streaming et son utilisation pour l'analyse de traces.

5.4.1 Analyse d'une trace d'exécution en streaming

Nous avons utilisé un workflow simple composé de 3 modules (Figure 5.12).

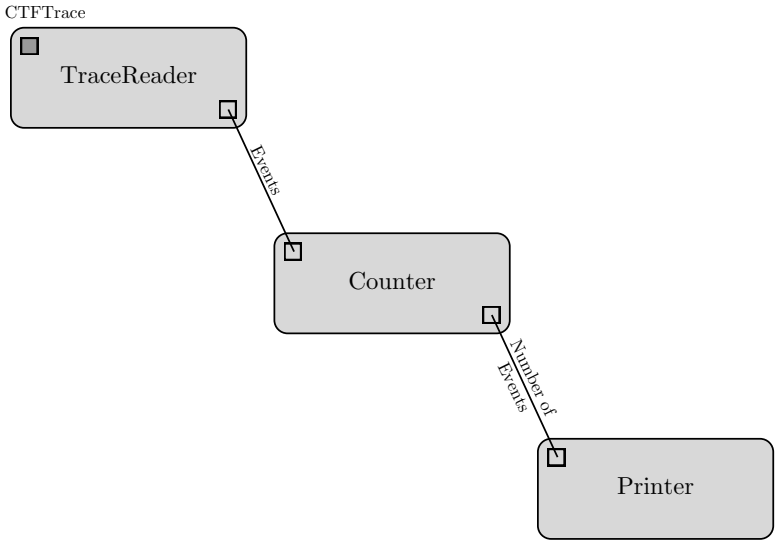


FIGURE 5.12 – Analyse d'une trace d'exécution en utilisant le streaming

Un premier module est en charge de lire une trace et de créer les événements dans notre modèle de données. Ce module envoie ensuite les événements à un second module qui compte les éléments reçus. Une fois tous les événements transmis par le premier module, le module compteur envoie le nombre total d'événements à un troisième module qui l'affiche sur la sortie standard. Ici, le module comptant les événements attend la fin du flux de données du module de lecture de trace avant de produire son résultat. Il aurait été possible d'implémenter le module "Counter" de sorte à ce qu'il "publie" la valeur du compteur toutes les n secondes afin de suivre l'évolution de l'analyse.

Nous avons utilisé une trace au format CTF [73] dont la taille est de 2 Go, et contenant environ 80 millions événements.

Cet exemple, même simple, montre que notre modèle par tâches nous permet de traiter une trace de plusieurs millions d'événements aisément. En plus, la structuration explicite du workflow nous permet d'appréhender plus facilement l'analyse des données. Enfin, le processus de création de l'analyse se voit lui aussi simplifié. La Figure 5.13 montre le code correspondant à cette analyse, qui consiste donc simplement à utiliser des modules existants, implémentés au préalable, et les connecter pour créer le workflow d'analyse.

```
1  from workflow_exec.workflow import Workflow
2  from workflow_exec.basic_modules import StandardOutput, Count
3  from workflow_exec.trace_analysis import CTFTraceReader
4
5  def main():
6
7      w = Workflow()
8
9      reader = w.add(CTFTraceReader)
10     w.set_value(reader, 'trace_path', '~/lttng/network-loopback/')
11
12     count = w.add(Count)
13     printer = w.add(StandardOutput)
14
15     w.connect(reader, 'event', count, 'data')
16     w.connect(count, 'length', printer, 'data')
17
18     w.run()
```

FIGURE 5.13 – Création d'un workflow d'analyse via une interface programmatique

5.4.2 Performances

L'exécution de l'exemple précédent a duré environ une heure. En effet, nous nous heurtons aux problèmes suivants.

Tout d'abord, l'utilisation massive d'événements au sein du workflow entraîne la création de nombreux objets, créant un surcoût pour le mécanisme de *garbage collection*. De plus, la gestion de la mémoire interne de Python ne permet pas d'optimiser le placement des données en mémoire pour tirer parti des mécanismes de cache. Typiquement, le placement non contigu en mémoire des données d'un flux qui devraient être manipulées séquentiellement pourrait déclencher de nombreux défauts de cache. Le fait que Python soit un langage interprété et non compilé rajoute aussi un surcoût non négligeable lors de l'exécution d'une application.

La mise en place du nouveau modèle d'exécution à base de tâches est coûteux. En effet, si dans le cadre de VisTrails le nombre de tâches à exécuter correspond au nombre de modules, dans notre cas ce nombre est bien plus important car chaque module est représenté par plusieurs tâches. Typiquement, si un module a été programmé à recevoir un flux de données à granularité d'événement, notre implémentation créera une tâche `InputTask` par événement. Pour une trace de 80 millions d'événements, cela fait 80 millions de tâches à ordonnancer.

Une dernière cause de ralentissement de notre analyse se situe au niveau du module lisant la trace CTF. La bibliothèque Python `babeltrace` qui permet de lire le format CTF est bien moins performante que la bibliothèque native C.

Nous avons étudié les performances de l'implémentation par tâches en variant deux facteurs. D'une part, nous avons considéré des configurations différentes où les événements sont transférés de manière différentes. Nous avons travaillé avec la configuration transmettant les événements un par un et avons considéré une configuration où les événements sont transférés par groupes de 1000. D'autre part, nous

Temps d'analyse		Tâches (1 évt)	Tâches avec groupes (1000 évts)	*Multi-thread*
Taille du *buffer*	1	87m	41m	71m
	1 000	55m	42m	63m
	10 000	113m	43m	61m

TABLE 5.1 – Temps d'exécution du workflow selon plusieurs configurations

avons agi sur la structure interne utilisée pour le transfert de données et avons fait varié la taille du *buffer*. Nous avons considéré trois cas : un buffer pouvant contenir 1 événement, 1000 événements et 10000 événements.

Nous avons également comparé l'implémentation par tâches à l'implémentation *multi-thread* en Python. La Table 5.1 résume les résultats des analyses que nous avons conduites.

En considérant la première colonne qui donne les temps d'analyse et en considérant un granularité d'un événement, nous voyons que cette implémentation peut être qualifiée de "naïve" puisque les performances sont les plus mauvaises. En effet, comparée à l'implémentation *multi-thread*, dans le pire des cas considérés, cette configuration est 2 fois plus lente. On remarque, toutefois, que la taille du buffer influence fortement les performances de cette configuration naïve, la taille intermédiaire étant la plus efficace. Avec un buffer intermédiaire pouvant stocker 1000 événements lors des transferts, la configuration naïve devient meilleure que la meilleure configuration *multi-thread*.

Si on considère la configuration à 1000 événements (deuxième colonne), elle montre une nette amélioration par rapport à la configuration naïve. En effet, le temps d'analyse est de 2 à 3 fois plus court. Plus important, le temps d'analyse de cette deuxième configuration est meilleur que l'implémentation *multi-thread* tout en restant stable par rapport aux différents tailles de buffers internes.

Même si cela dépend d'une paramétrisation des mécanismes internes, nous pouvons conclure que notre modèle d'exécution par tâches peut amener un vrai gain en termes de performances comparé à la configuration classique *multi-thread*.

5.5 Conclusion

L'outil VisTrails est celui qui se rapproche le plus du modèle d'infrastructure que nous proposons pour l'analyse de traces. La possibilité de créer des modules personnalisés nous a permis d'implémenter des fonctions pour l'analyse de trace dans VisTrails, bien que l'outil n'ait pas été prévu pour cela initialement. Nous avons pu valider l'utilité d'utiliser VisTrails pour l'analyse de traces, notamment pour la possibilité de créer un workflow de manière itérative au fil de l'analyse. Les contraintes relatives au transfert de données ont pu être supprimées en intégrant un nouveau mécanisme de transfert par flux permettant le traitement de traces de plusieurs giga-octets. La validation de la méthode nous a convaincu que l'utilisation des mécanismes de *streaming* est la direction à suivre. Cependant, les performances limitées de l'exécution du workflow nous ont amenées à repenser l'implémentation, en

s'affranchissant notamment des contraintes techniques de Python. Nous présentons dans le chapitre suivant le prototype que nous avons conçu.

Chapitre 6

SWAT : Un système de *workflow* pour l'analyse de traces d'exécution

Dans ce chapitre, nous présentons SWAT (*Streamed Workflow Analysis Tool*) : un prototype que nous avons développé pour valider notre proposition de système de *workflow* pour l'analyse de traces d'exécution (cf. Section 3.2). *Streamed Workflow Analysis Tool* (SWAT) tire les enseignements de notre expérience avec Vistrails et propose une implémentation efficace des fonctionnalités indispensables à un système de *workflow* devant gérer de grandes quantités de données en appliquant le principe de traitement en continu (*streaming*).

Ce chapitre est organisé en deux parties. La première partie présente l'implémentations de SWAT. Elle expose nos choix techniques pour proposer un outil mettant l'accent sur la généricité et la modularité.

La deuxième partie du chapitre traite de la validation de SWAT en présentant son utilisation pour la mise en place de la méthode d'analyse générique de systèmes qui fait l'objet de notre troisième proposition (cf. Section 3.3). Nous validons cette proposition en détaillant comment ce *workflow* peut être utilisé afin d'évaluer les performances de la suite de benchmarks Phoronix [11].

6.1 Implémentation

SWAT est implémenté en C++ ce qui nous permet de nous affranchir des limitations du langage Python rencontrées précédemment. En effet, le langage C++ nous permet de mieux contrôler la mémoire et d'utiliser des mécanismes de *multi-threading* exploitant le parallélisme physique des machines. De plus, le mécanisme de *templates* de C++ [78] nous permet de fournir des briques de base génériques pour la construction de *workflows*. L'implémentation de SWAT représente environ 1500 lignes de code.

Dans la suite, nous présentons les aspects liés à la définition des modules de workflow, de leur interconnexion, de la construction globale d'un workflow et de son exécution.

6.1.1 Modules

Un module SWAT est implémenté par une classe abstraite `Module`. Tout module d'un *workflow* doit être implémenté sous forme de classe C++ qui hérite de celle-ci. La classe `Module` définit les fonctions `start()`, `compute()` et `join()`. La fonction `start()` est appelée lors du lancement du *workflow* global. La fonction `compute()` définit le traitement qui doit être effectué par le module. La fonction `join()` est utilisée pour attendre la fin de l'exécution d'un module.

6.1.2 Ports

Pour la définition des ports d'un module, nous définissons deux classes génériques `InputPort` et `OutputPort`. `InputPort` est utilisée pour définir un port d'entrée (réception de données), alors que `OutputPort` est utilisée pour définir un port de sortie (envoi de données). Au sein d'un module (classe C++ héritant de `Module`), des attributs de classes héritant d'`InputPort` vont définir les ports d'entrée du module. D'autres attributs de classes héritant d'`OutputPort` vont définir les ports de sortie.

En interne, la connexion entre modules se traduit par la création d'une file de communication qui sert à la gestion du flux de données. La file est gérée par une structure de données qui gère un tampon circulaire auquel ont accès un (module) producteur et plusieurs (modules) consommateurs. L'interface fournie par la structure est constituée des fonctions `send()`, `recv()` et `close`. La fonction `send()` est utilisée sur un port de sortie pour envoyer des données. La fonction `recv()` est utilisée sur un port d'entrée pour l réception de données. Enfin, la fonction `close` est utilisée sur un port de sortie pour indiquer la fin d'un flux de données côté module producteur.

Notre structure de tampon circulaire reprends les principes de base d'une implémentation. Elle se distingue néanmoins par le fait qu'elle fonctionne avec un producteur mais peut avoir plusieurs consommateurs. Le cas échéant, tous les consommateurs lisent toutes les données écrites par le producteur. Nous avons utilisé le principe des templates C++ et pouvons donc manipuler des données de tout type.

Nous avons un pointeur de début et un pointeur de fin pour distinguer la zone contenant des éléments de celle étant vide. Ce sont respectivement `Beg` et `End` représentés par des points bleus sur la Figure 6.1. La structure maintient également un tableau de pointeurs. Un pointer est lié à un consommateur et indique quelles sont les données lues par ce consommateur. Sur la figure, `R1` et `R2` sont les pointeurs des consommateurs. Sur cet exemple, le producteur à écrit 10 éléments dans la structure, le premier consommateur a lu 2, et le second 5.

Pour implémenter une connexion entre deux modules où il y a un module produisant des données et un module les consommant, le tampon en interne travaille avec un producteur et un consommateur. Pour les connexions où un module envoie

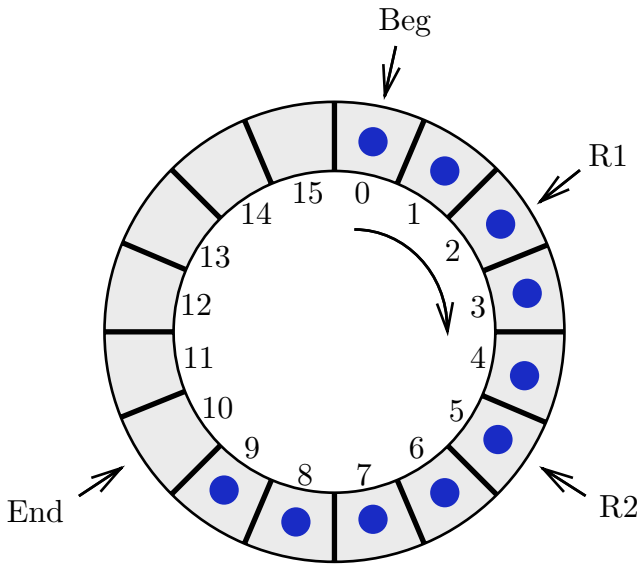


FIGURE 6.1 – Représentation du buffer circulaire

des données à plusieurs modules $(1 - N)$, le tampon travaille avec un producteur et plusieurs consommateurs.

6.1.3 Construction et exécution d'un *workflow*

Un workflow est représenté par une classe `Workflow` qui contient la liste des modules participant au *workflow*. La construction du workflow se fait de manière programmatique. Tout d'abord les modules sont instanciés, ajoutés au *workflow* et inter-connectés. La Figure 6.2 montre un exemple de création de workflow simple avec deux modules.

```
1  // creation of the workflow
2  Workflow w;
3
4  // instanciate the two modules
5  Producer p;
6  Consumer c;
7
8  // add modules into the workflow
9  w.add_module(&p);
10 w.add_module(&c);
11
12 // link the two ports of the modules
13 w.link(p.oport, c.iport);
14
15 // run the workflow
16 w.start();
```

FIGURE 6.2 – Exemple d'instanciation d'un workflow

Il est également possible de créer des sous-workflows, en encapsulant un workflow dans un module. Pour cela, nous proposons les fonctions `link_outputs(OutputPort&, OutputPort&)` et `link_inputs(InputPort&, InputPort&)` qui font correspondre des ports du module englobant à des ports de modules du workflow (Figure 6.3).

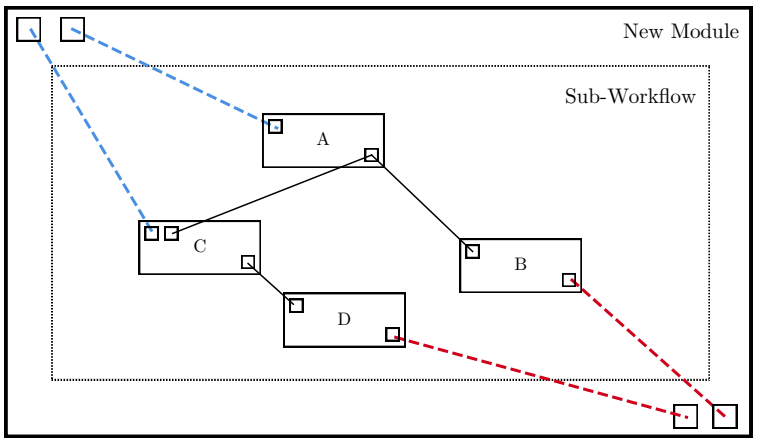


FIGURE 6.3 – Création d'un sous-workflow dans un module

Pour l'exécution d'un workflow, nous utilisons un modèle classique où un module est exécuté par un fil d'exécution. Nous permettons ainsi l'exécution en parallèle de plusieurs modules du workflow, la difficulté d'ordonnancement étant prise en charge par l'ordonnanceur système.

6.2 Analyse de benchmarks de la suite Phoronix

L'utilisation de benchmarks est une des solutions les plus utilisées pour analyser le comportement et les performances d'un système. La construction d'un programme de benchmark étant compliquée [20], les développeurs se tournent en général vers des solutions déjà existantes. Les benchmarks sont habituellement regroupés (*suites*) par domaine d'application, car ils sont représentatifs d'une charge de travail (*workload*) particulière. Par exemple, pour le domaine des systèmes parallèles, les suites PARSEC et SPLASH-2 [26] proposent des *benchmarks* dont le *workload* est représentatifs des applications pour les multiprocesseurs.

Comme il existe beaucoup de benchmarks et de suites disponibles, les développeurs sont confrontés au choix desquels utiliser [79, 47]. Leur choix se porte naturellement vers les benchmarks les plus populaires. Or, ces derniers ne sont pas toujours adaptés aux systèmes et situations de test et peuvent donc produire des résultats non représentatifs[80]. Il est donc important de comprendre le fonctionnement des programmes de benchmarks.

Nous proposons d'utiliser notre modèle d'analyse générique de système afin d'étudier les *benchmarks* de la suite PTS [11]. Nous avons implémenté le workflow pour l'analyse générique de systèmes dans notre infrastructure SWAT, afin d'obtenir pour les benchmarks les profils unifiés proposés dans la Section 3.3. Ces profils, générés

automatiquement grâce à notre chaîne d'analyse générique, permettent de comparer les benchmarks, de mieux comprendre leur comportement et ce qu'ils testent, et donc de pouvoir interpréter les résultats de performances obtenus. Cette implémentation représente environ 3000 lignes de code C++ et comprend la définition des modules spécialisés pour l'analyse de trace, ainsi que la définition du workflow. Cette analyse permet aussi de valider notre proposition quant à la simplification du procédé d'analyse par construction de workflow, le traitement de grosses quantité de données, et enfin la généricité du modèle.

6.2.1 Présentation de PTS

Nous avons choisi d'étudier la suite de benchmarks PTS car elle est l'une des suites les plus utilisées lorsqu'il s'agit de tester les performances système. En effet, elle est disponible sur plusieurs plates-formes, notamment Linux, macOS, Windows, Solaris, et BSD, et elle propose pas loin de deux cents benchmarks open-source. De plus, la suite inclut un système performant pour le téléchargement, le lancement, et l'enregistrement des résultats des différents benchmarks. Il est aussi possible de proposer de nouveaux benchmarks afin d'enrichir la suite.

Certains tests disponibles dans la suite PTS se concentrent sur un aspect en particulier, comme par exemple, la consommation énergétique, la puissance de calcul, le système de stockage ou encore la mémoire. D'autres se concentrent sur une technologie particulière, comme, par exemple, OpenGL, Java, ou PHP. La seule information donnée sur le comportement d'un benchmark est une description sommaire fournie par son développeur.

PTS catégorise les *benchmarks*en leur associant une *famille* : le *disque*, la *mémoire*, le *processeur*, le *réseau*, le *système*ou la partie *graphique*.

Intuitivement, la famille donne une indication sur la partie du système qui est testée. Toutefois, PTS ne donne pas d'explications sur cette assignation des *benchmarks*. De plus, on observe que la répartition des différents benchmarks dans les familles est très irrégulière (Table 6.1). Un seul benchmark teste le *réseau*, alors que quatre-vingt-dix benchmarks sont disponibles pour tester le *processeur*.

Famille	*système*	*processeur*	*réseau*	*mémoire*	*graphique*	*disque*
Benchmarks	31	90	1	2	75	13

TABLE 6.1 – Répartition des benchmarks par familles dans PTS

Bien qu'une analyse exhaustive des 200 benchmarks de la suite PTS est possible grâce à notre approche, nous avons décidé de nous concentrer sur une dizaine de benchmarks. Ceux-ci on été choisis en utilisant une fonction de PTS permettant de connaître les benchmarks les plus populaires. Nous avons fait en sorte d'obtenir un ensemble représentatif par rapport aux classement dans les différentes *familles* fait par PTS. Les benchmarks, leur version ainsi que la famille PTS associée sont listés Table 6.2. Nous avons donc pour objectif d'étudier et de comprendre cette classification.

Benchmark	Version	Famille PTS
compress-gzip	1.1.0	*processeur*
ffmpeg	2.5.0	*processeur*
scimark2	1.2.0	*processeur*
stream	1.3.0	*mémoire*
ramspeed	1.4.0	*mémoire*
phpbench	1.1.0	*système*
pybench	1.0.0	*système*
iozone	1.8.0	*disque*
unpack-linux	1.0.0	*disque*
network-loopback	1.0.1	*réseau*

TABLE 6.2 – Liste des benchmarks utilisés

Chaque exécution d'un *benchmark* calcule un *score*. Ce *score* représente la métrique de performance associée à l'exécution du programme sur la plate-forme. Il peut représenter un temps d'exécution, un débit de données, ou encore un nombre sans unité. La Table 6.3 présente un exemple de scores obtenues pour les *benchmarks* que nous avons sélectionnés.

Benchmark	Score	Unité
compress-gzip	11.27	Secondes
ffmpeg	10.93	Secondes
iozone	363.82	Mo/Secondes
network-loopback	9.43	Secondes
phpbench	141858	Score
pybench	1623	Millisecondes
ramspeed	14633.43	Mo/Secondes
scimark2	1392.42	Mflops
stream	16795.10	Mo/Secondes
unpack-linux	8.07	Secondes

TABLE 6.3 – Exemples de scores PTS

6.2.2 *Workflow* d'analyse de PTS

Afin de produire des profils de performances pour les benchmarks PTS, nous traçons leur exécution, analysons les traces à l'aide d'un workflow SWAT, et validons nos résultats. Pour rappel, les métriques que nous observons sont les suivantes :

- La durée d'exécution du benchmark.
- L'occupation CPU et l'aspect parallèle de l'exécution.
- La répartition d'exécution en espace noyau et utilisateur.
- La répartition de l'utilisation des différentes parties du noyau via les *tracepoints*.
- L'utilisation de la mémoire.

— La répartition des opérations liés au calcul pur, et à la mémoire.

Pour produire les traces dont nous avons besoin pour l'analyse, nous exécutons chaque benchmark selon les trois configurations suivantes :

— *bench-all-event* : Le benchmark est tracé avec LTTng avec tous les *tracepoints* et les appels systèmes activés. Cette trace nous permet de calculer la durée d'exécution, d'investiguer l'activité des différentes parties du noyau, et d'établir la répartition du temps passé en espace noyau et utilisateur.
— *bench-libc* : Lors de l'exécution du benchmark, nous traçons seulement les événements d'ordonnancement et l'utilisation des fonction mémoire.
— *bench-perf-trace* : Ce que nous traçons lors de cette configuration sont les compteurs de performances.

Pour valider nos résultats, nous avons utilisé les trois configurations suivantes :

— *bench-only* : Le benchmark seul est exécuté sans traçage, le score ainsi obtenu nous sert de référence.
— *bench-time* : Le benchmark est exécuté avec le programme `time`, qui calcule le temps passé en espace utilisateur et noyau. Il nous sert a comparer les valeurs obtenues avec notre analyse par trace.
— *bench-perf* : Le benchmark est exécuté avec l'outil `perf`, pour observer les compteurs de performance. Il nous sert également a comparer les valeurs obtenues avec notre analyse par trace.

En plus des métriques considérées dans le workflow d'analyse générique, nous avons également considéré la *stabilité* des métriques observés. En effet, afin d'obtenir une analyse statistiquement correcte pour la compréhension des benchmarks, nous avons voulu étudier les écarts de comportements lors de l'exécution successive du même benchmark. Nous avons donc, pour chaque configuration, lancé le benchmark 32 fois, afin d'observer la cohérence statistique des résultats obtenus [46, 81].

Le workflow est défini en utilisant des sous-workflows que nous avons implémentés. Chaque sous-workflow se spécialise dans l'analyse d'une des six configurations d'exécution des benchmarks. Nous avons donc six sous-workflows dont voici le détail :

— `ProfileAllRuns` : Analyse la durée, le temps utilisateur/noyau, l'occupation CPU, et la répartition de l'utilisation du noyau pour les 32 traces issues de la configuration *bench-all-events*.
— `MemoryAllRuns` : Analyse l'utilisation mémoire pour les 32 traces issues de la configuration *bench-libc*.
— `PerfAllRuns` : Analyse la répartition des opérations de calcul et de mémoire à partir des 32 traces issues de la configuration *bench-perf-trace*.
— `TimeAllAnalyses` : Analyse le temps d'exécution pour les six configurations.
— `ScoreAllAnalyses` : Analyse le score obtenue pour les six configurations.
— `TraceAllAnalyses` : Analyse le nombre d'événements et la taille des traces, pour les trois configurations produisant une trace.

La Figure 6.4 présente le workflow principal utilisant les sous-workflow que nous venons de décrire. Chaque sous-workflow prend en donnée d'entrée le chemin où les traces et les résultats des six configurations sont stockés.

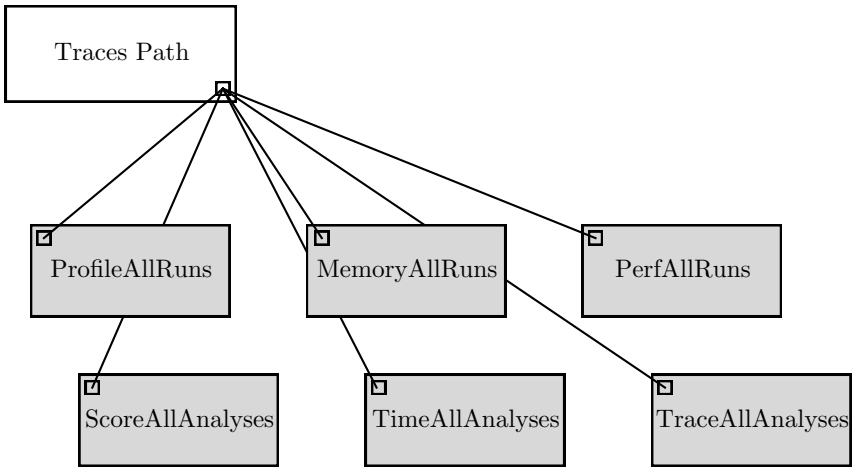


FIGURE 6.4 – Workflow principal de l'analyse

La Figure 6.5 montre la partie du workflow associé à la configuration *bench-perf-trace*, qui calcule à partir des compteurs de performance collectés le nombre d'opérations de calcul, ainsi que celui lié aux accès mémoire. Un premier module `CTFReader` est en charge de lire la trace. Le flux d'événements créé est envoyé au module `Processes`, en charge de reconstruire les processus et d'associer à chaque événement le processus qui a créé cet événement. Le module `Filter` va ensuite filtrer les événement en fonction du nom du programme dont on veux conserver les événements. Cela nous permet d'isoler les événements produits par le benchmark seulement. Ce flux est envoyé au module `PerfCounterAnalysis` en charge de calculer le nombre d'opérations de calcul, et d'opérations liées aux accès mémoire.

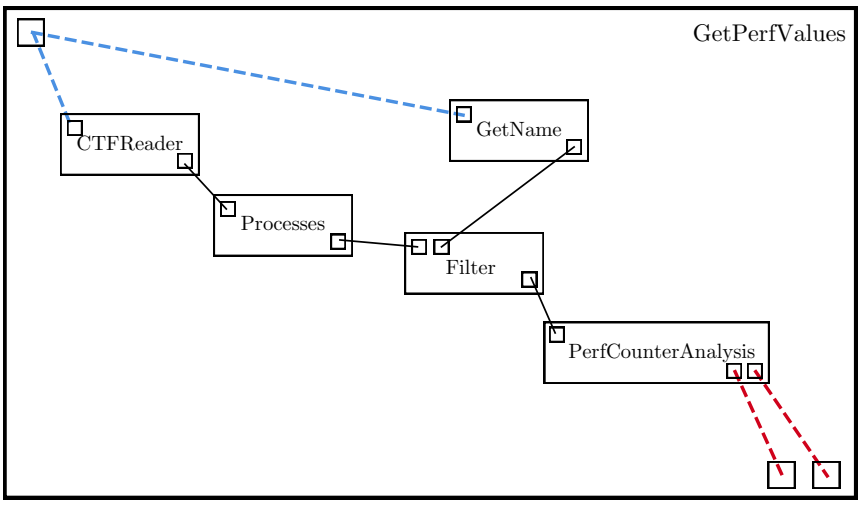


FIGURE 6.5 – Détail du sous-workflow `GetPerfValues`

Le sous workflow que nous venons de décrire est en fait encapsulé dans un module appelé `GetPerfValues`. Ce module est utilisé 32 fois dans le module `PerfAllRuns`,

un pour chaque trace. Le module `PerfAllRuns` est en charge de collecter les données produites par les 32 modules `GetPerfValues`. Pour chacune des valeurs retournées par le module, un module `ValueToStream` est utilisé pour regrouper dans un seul flux de données, puis envoyer à un module calculant des statistiques, notamment la moyenne et l'écart type. Les résultats sont ensuite écrits dans un fichier pour une interprétation ultérieure. La figure Figure 6.6 détaille ce workflow pour le module `PerfAllRuns`.

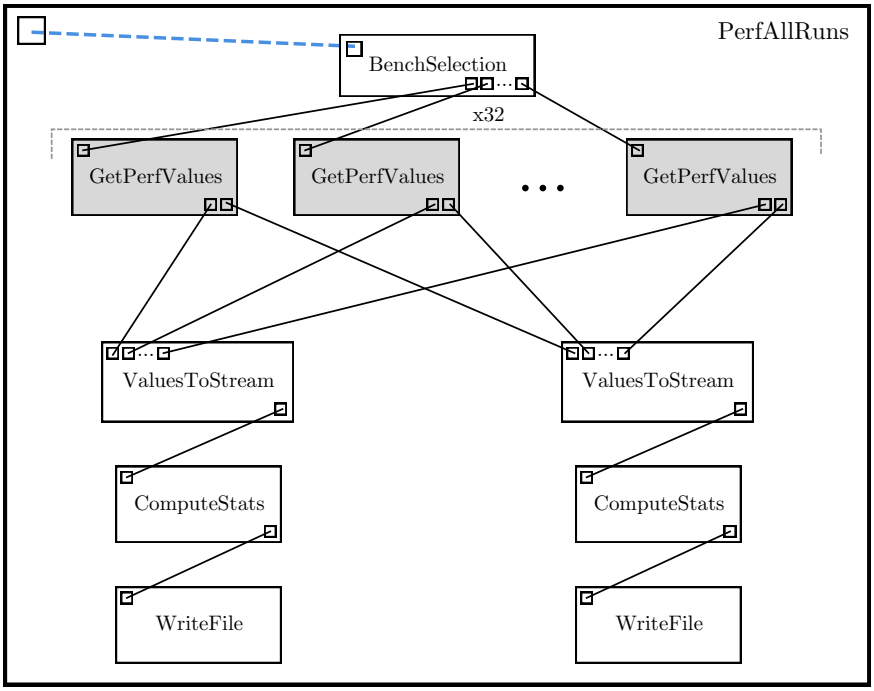


FIGURE 6.6 – Détail du sous-workflow `PerfAllRuns`

6.2.3 Modules spécialisés du workflow d'analyse

Dans ce qui suit, nous présentons les aspects intéressants ou techniques de nos modules d'analyse.

Reconstruction et filtrage des processus

Le module exploite les événements générés par l'ordonnanceur système afin de reconstruire l'arbre des processus. Ces événements sont : `sched_process_fork` et `sched_process_exec` pour la création de nouveaux processus, et `sched_switch` pour les changements de contexte d'exécution (processus). Lors de la lecture de la trace, lorsque le module rencontre un événement `sched_switch`, le module utilise les informations métier de cet événement contenant le nom des processus précédent et suivant exécutés.

Pour le CPU sur lequel l'événement `sched_switch` est survenu, on connaît donc le nom et le PID du processus qui vient d'être ordonnancé. Tous les événements

jusqu'au prochain `sched_switch` sont donc produits par ce processus. Les événements `syscall_entry` et `syscall_exit` sont utilisés pour différencier le mode utilisateur ou noyau pour chaque processus. Les événements `sched_process_fork` et `sched_process_exec` sont utilisés pour renseigner les producteurs parents lors de la création d'un nouveau processus. Au final, le module produit des événements contenant toutes les informations sur les producteurs, mais aussi les événements états donnant les informations sur le mode d'exécution.

Pour le filtrage des processus, on utilise les informations de producteur logiciel et producteur logiciel parent. Tous les processus dont le nom correspond à celui passé en paramètre sont donc conservés, les autres sont défaussés. Au final, on ne conserve que les événements associés au sous-arbre de processus de celui considéré, typiquement, le benchmark que l'on veux analyser.

Catégorisation des événements (utilisé par `ProfileAllRuns`)

Pour catégoriser les événements noyau selon les familles PTS, nous utilisons le nom des *tracepoints* Ce nom est stocké comme étant le *type* de l'événement correspondant. Pour chaque *tracepoint* disponibles dans le noyau, nous avons investigué à quelle partie du système il était lié, afin de l'associer à une des *famille* PTS. En effet, intuitivement, si le benchmark est catégorisé comme étant un benchmark *processeur*, on s'attend à voir beaucoup d'événements liés au processeur. Par exemple, l'événement `mm_page_alloc` et `mm_page_free` sont des événements liés à la gestion de la mémoire, ils sont donc associés à la famille *mémoire*. Les événements `power_cpu_idle` et `htimer_expire` sont quant à eux associés à la *famille processeur*.

Notre répartition de ces *tracepoints* dans différentes *familles* est présentée en Table 6.4. En utilisant le préfix des noms, nous avons obtenu la répartition des *tracepoints* donnée. La classification se fait donc en faisant correspondre le *type* de l'événement avec la *famille* qui lui est associée.

Famille	Tracepoint
processeur	`timer_*; hrtimer_*; itimer_*; power_*; irq_*; softirq_*;`
mémoire	`kemem_*; mm_*`
système	`workqueue_*; signal_*; sched_*; module_*;rpm_*; lttng_*; rcu_*; regulator_*; regmap_*;regcache_*; random_*; console_*;gpio_*;`
graphique	`v4l2_*; snd_*;`
disque	`scsi_*; jbd2_*; block_*;`
réseau	`udp_*; rpc_*; sock_*; skb_*; net_*; netif_*;napi_*;`

TABLE 6.4 – Catégorisation des tracepoints du noyau

Analyse des allocations mémoire

Le module en charge de l'analyse mémoire utilise les événements générés par la bibliothèque `liblttng-ust-libc-wrapper.so` de LTTng. Pour chaque appel à l'une des fonctions liée à la mémoire, un événement est généré. Pour un événement

Plates-formes de tests

Nos expérimentations on été faites sur plusieurs plates-formes : une carte UDOO [18], une carte Nvidia Jetson TK1 [8], une carte de développement Juno [2] et un poste de travail "classique". Dans la suite, nous présentons les résultats sur deux de ces plates-formes, à savoir la carte Juno et le poste de travail. Ces deux plates-formes ayant des caractéristiques différentes, il est intéressant de comparer les profils des benchmarks que nous obtenons.

La carte Juno possède un processeur big.LITTLE 64-bit composé d'un Cortex bi-cœur A57 et d'un Cortex quad-cœur A53, 8GB de RAM. La machine de bureau possède un processeur x86-64 Xeon E3-1225@3.20GHz, et de 32GB de RAM. Les deux plates-formes utilisent une connexion réseau Ethernet gigabit, et un disque SSD pour le stockage.

Logiciels

Afin d'obtenir des résultats comparables, nous avons voulu avoir le système le plus similaire sur les deux plates-formes utilisés pour nos tests. Ces deux plates-formes utilisent la même version de Debian avec le noyau 4.3.0-1-arm64 pour la carte Juno, et 4.4.0-1-amd64 pour la machine de bureau. Nous avons utilisé la version de LTTng 2.8-stable pour la partie trace. Phoronix est utilisé dans sa version 6.2.2. GCC, utilisé pour compiler les benchmarks lors de l'installation de ceux-ci via PTS, est en version 5.3.1.

6.3 Résultats

Dans cette section, nous présentons tout d'abord les résultats obtenues sur le poste de travail. Dans un second temps nous comparons les résultats du poste de travail avec ceux de la carte Juno afin d'exposer les éventuelles différences de comportements des benchmarks d'une plate-forme à l'autre.

6.3.1 Résultats pour le poste de travail (x86)

Investigation sur la *famille* du benchmark

Pour comprendre notre analyse, considérons le cas du benchmark `compress-gzip`. Utilisant la trace LTTng et le module qui filtre et catégorise les événements noyau, nous obtenons la répartition montrée Figure 6.7. Pour ce benchmark, 80% des événements sont liés à la gestion de la *mémoire*, environ 20% au *système*, et 2% pour le *processeur*. Or, ce benchmark est sensé tester les performances du *processeur*, une grande partie de l'utilisation noyau est lié à la *mémoire*. Notre intuition de départ, qui consiste à dire qu'un benchmark classé dans une famille PTS aura une majorité d'événements lié à cette famille, n'est donc pas vérifiée dans ce cas.

La Table 6.5 montre les résultats pour tous les *benchmarks* considérés. Nous prenons les événements majoritaires à chaque fois pour établir la *familles calculées*,

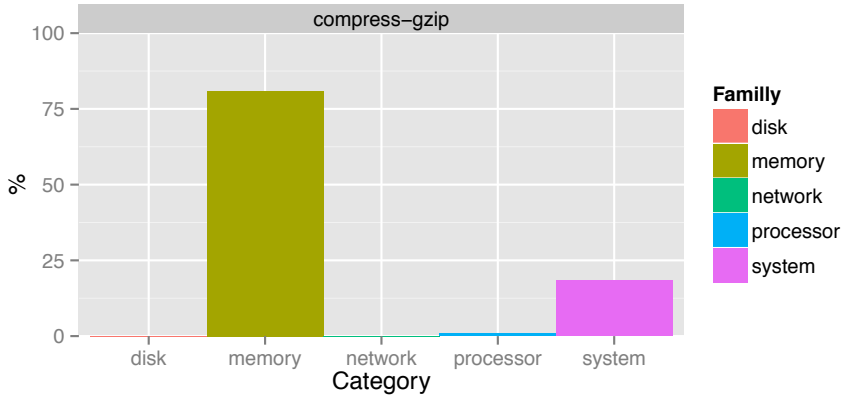


FIGURE 6.7 – Répartition de l'utilisation du noyau du benchmark `compress-gzip`

et comparons avec la famille annoncée par PTS. On observe que seulement les familles *mémoire* et *système* sont représentées. De plus, pour ces deux familles il y a correspondance entre la *famille* PTS et la *famille* observée.

Benchmark	Famille PTS	Famille observée
`compress-gzip`	*processeur*	*mémoire*
`ffmpeg`	*processeur*	*système*
`iozone`	*disque*	*mémoire*
`network-loopback`	*réseau*	*système*
`phpbench`	*système*	*système*
`pybench`	*système*	*système*
`ramspeed`	*mémoire*	*mémoire*
`scimark2`	*processeur*	*système*
`stream`	*mémoire*	*mémoire*
`unpack-linux`	*disque*	*mémoire*

TABLE 6.5 – Présentation des familles observées

La Figure 6.8 présente les résultats détaillés pour les 10 benchmarks. Ce que l'on peut observer, tout d'abord, est la présence d'événements spécifiques selon les benchmarks. En effet, le benchmark `network-loopback`, de la famille *réseau*, est le seul produisant des événements liés au réseau (à hauteur de 30%). Ces événements ne sont pas majoritaires, mais indiquent tout de même une activité réseau lors de l'exécution du benchmark. De manière analogue, les benchmarks de la famille `iozone` et `unpack-linux` sont les seuls à contenir des événements liés à l'utilisation du *disque*. Les benchmarks `ramspeed` et `stream` présentent une majorité d'événements liés à la mémoire avec respectivement 92% et 83%, ce qui est cohérent avec le fait que ces benchmarks testent la *mémoire*.

Sur les 10 benchmarks, seuls 4 d'entre eux sont associés à une *famille* correspondant à ce que nous observons au niveau du noyau. Pour les benchmarks *disque* et *réseau*, on observe tout de même la présence d'événements spécifiques à ces *familles*. Ces observations étant faites seulement sur l'utilisation du noyau, tout ce

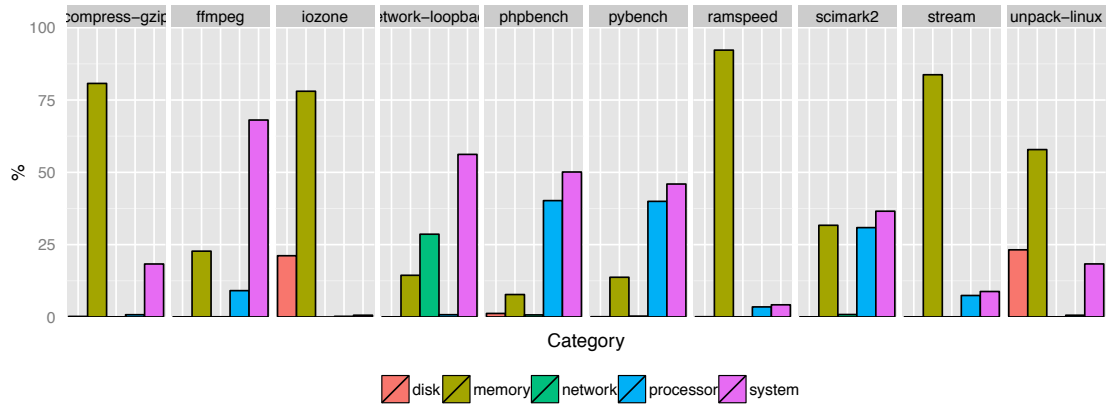


FIGURE 6.8 – Catégorisation des événements noyau pour les 10 benchmarks

qui se passe en espace utilisateur n'est donc pas observé. De plus, l'utilisation noyau peut être très variable d'un programme à l'autre, et ce que l'on observe peut être représentatif seulement d'une petite partie de l'exécution du benchmark. Nous avons donc observé le temps passé en espace noyau et utilisateur afin de vérifier cela.

Temps noyau et utilisateur

La Figure 6.9 montre la répartition entre le temps passé en mode noyau et celui en mode utilisateur. Nous observons que la plupart des *benchmarks* exécutent en majorité du code utilisateur. On constate que les benchmarks utilisant des entrées/-sorties, comme le disque ou le réseau, passent plus de temps à exécuter du code noyau. Ces benchmarks sont principalement `iozone` et `network-loopback`, et de manière moins marquée `unpack-linux`.

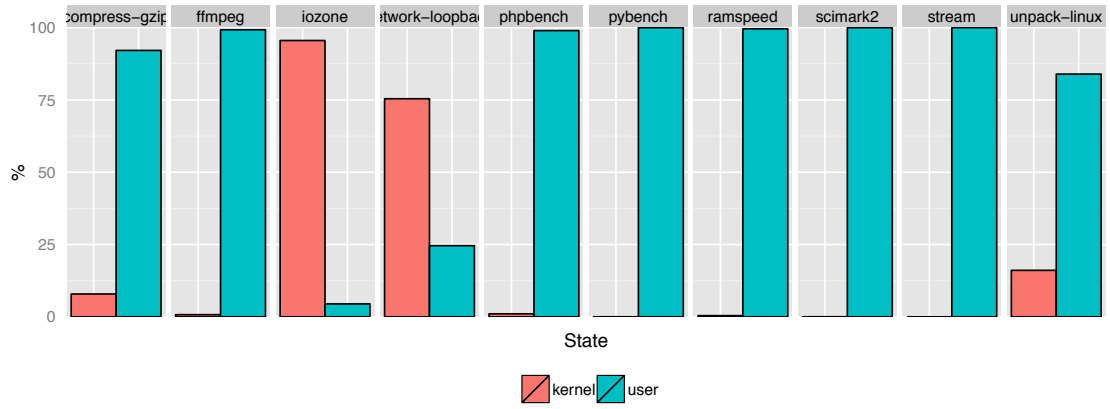


FIGURE 6.9 – Ratio du temps passé en mode utilisateur et noyau

Du point de vue caractérisation de *benchmark*, nous identifions donc des *benchmarks* dont le comportement du noyau. Du point de vue de l'investigation des familles PTS, les profils des *benchmarks* s'exécutant majoritairement en mode utili-

sateur doivent être complétés par une analyse coté utilisateur. Dans tous les cas, le profil noyau seul n'est pas représentatif.

Utilisation des CPUs et parallélisme

La Figure 6.10 présente les résultats pour l'utilisation des différents CPUs. Pour chacun des CPUs, on regarde le temps où les processus liés au benchmark sont actifs. Le benchmark `stream` par exemple est pratiquement à 100% d'utilisation pour les 4 processeurs de la machine. Pour le benchmark `iozone`, les processeur 0 et 1 sont utilisés à environ 10% et les processeurs 2 et 3 entre 20% et 25%. Le reste (la partie grise) correspond donc au temps *idle*. Pour ce benchmark, beaucoup de ressources processeur ne sont pas utilisés, cela peut s'expliquer par le temps passé à attendre les synchronisation des entrées/sorties du disque.

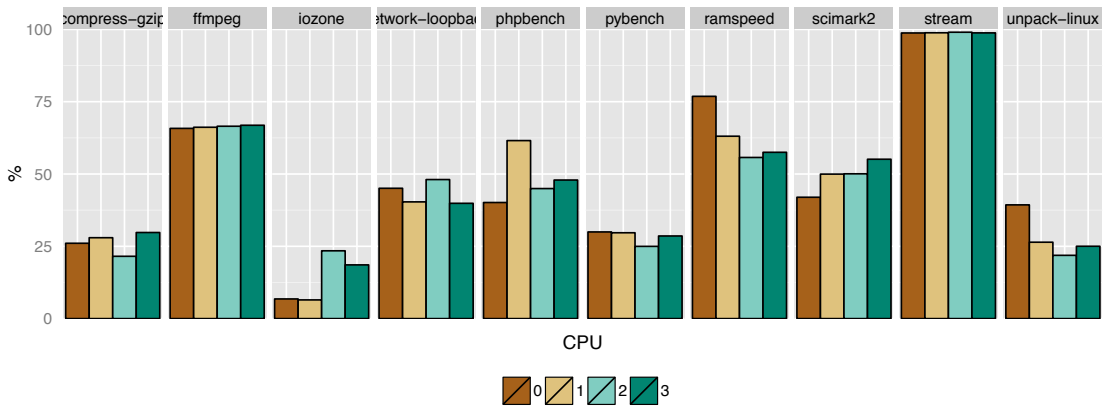


FIGURE 6.10 – Utilisation des processeurs en moyenne sur 32 exécutions

Les valeurs présentées étant la moyenne sur les 32 exécution de chaque benchmark, la Figure 6.10 ne montre pas l'utilisation réelle des processeurs. En effet, l'ordonnancement et le placement des processus liés aux benchmarks peuvent varier d'une exécution à l'autre. la Figure 6.11 montre l'utilisation des processeurs pour une itération par benchmark.

Sur cette figure, on peut aisément distinguer les benchmarks utilisant le parallélisme comme par exemple `ffmpeg`, `network-loopback`, `ramspeed`, `stream` et `unpack-linux`. Pour les benchmarks séquentiels comme `compress-gzip`, `phpbench`, `pybench`et `scimark2`, on remarque que leur exécution sur plusieurs itérations (Figure 6.10) est répartie sur les différents processeurs. Cela montre que les benchmarks n'ont pas de placements CPU déterminés.

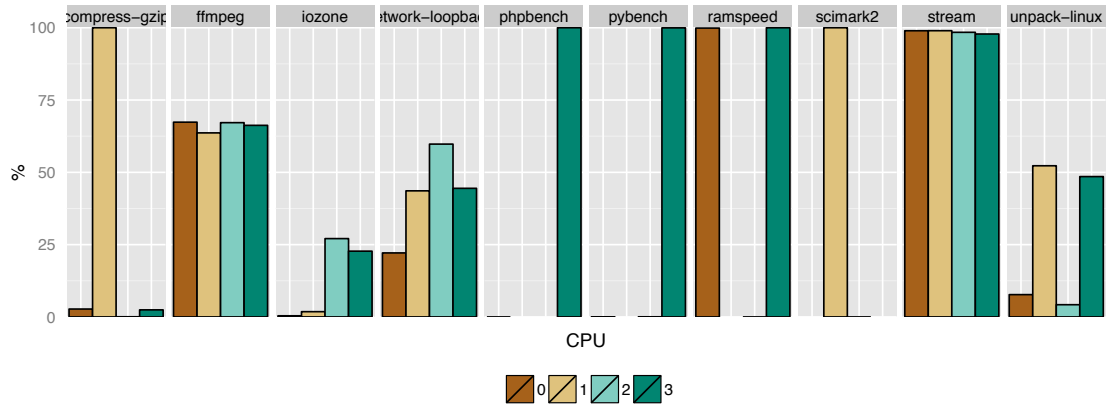


FIGURE 6.11 – Utilisation des processeurs pour une itération

Analyse mémoire et calcul

La Table 6.6 présente les résultats pour la mémoire. Les deux benchmarks testant la *mémoire*, `stream` et `ramspeed`, ont une utilisation des fonctions d'allocations différentes. Le benchmark `ramspeed` alloue jusqu'à 3.5 Go de mémoire, alors que `stream` seulement 10 Ko.

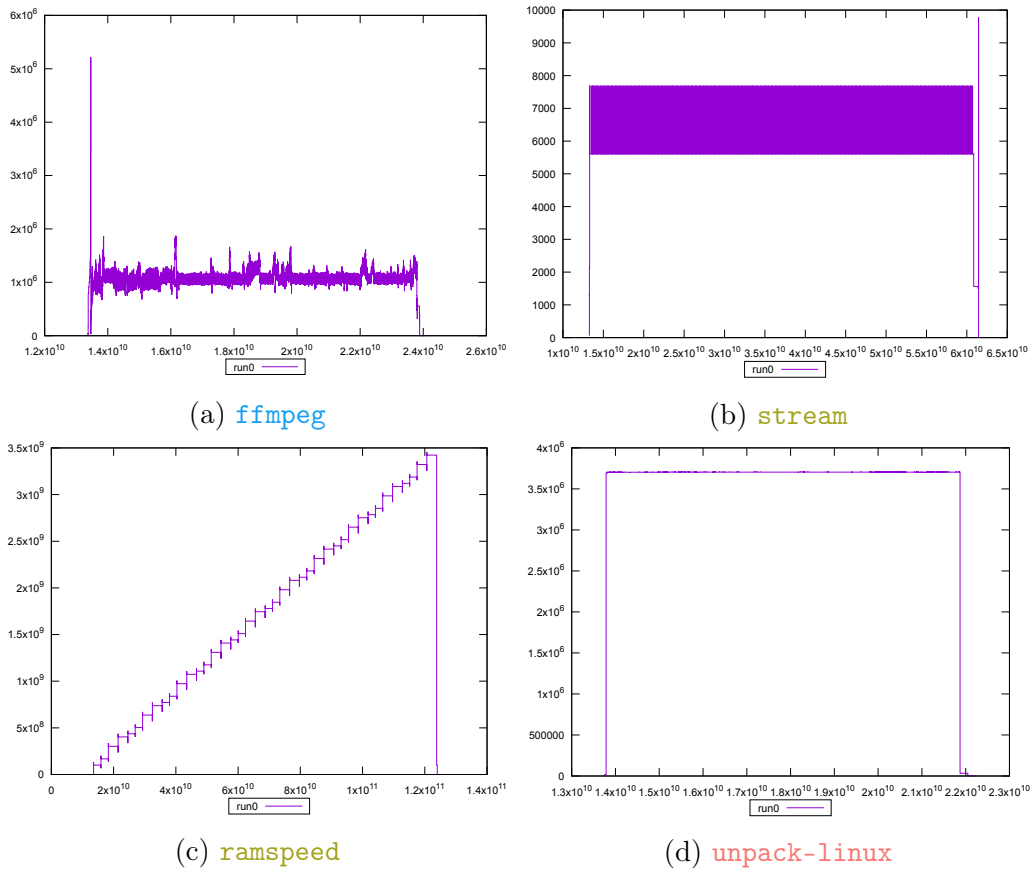
Benchmark	Mémoire maximum allouée (en Ko)	Appels à `free` inutiles
`compress-gzip`	134	5
`ffmpeg`	5 215	215 270
`iozone`	37 753	3
`network-loopback`	1 033	95
`phpbench`	4 577	45
`pybench`	2 864	42
`ramspeed`	3 456 110	4
`scimark2`	16 780	0
`stream`	10	404
`unpack-linux`	3 705	494 910

TABLE 6.6 – Mémoire maximum et appels inutiles à la fonction free

Nous avons aussi observé le nombre d'appels à la fonction sur le pointeur `null`. Nous pouvons noter que les benchmarks `ffmpeg` et `unpack-linux` en présentent une quantité importante. L'appel de la fonction `free` sur un pointeur `null` n'est pas incorrect, cependant ceci peut mener à un surcoût inutile et impacter les performances mesurées par le benchmark, notamment pour `ffmpeg` dont le score calculé est un temps d'exécution en secondes.

La reconstruction des allocations mémoire au fil du temps nous permet de dresser un profil mémoire pour les benchmarks. La Figure 6.12 présente ces profils pour 4 benchmarks. Ce profil permet de discerner des comportements particuliers. Par exemple, le benchmark `ramspeed` procède à des allocations continues. Le benchmark

`ffmpeg`, quant à lui, montre un profil mémoire complexe et non régulier. Le benchmark `unpack-linux` semble allouer une grosse quantité de mémoire dès le début de son exécution, et ne la libère qu'à la fin.



(a) `ffmpeg` (b) `stream`

(c) `ramspeed` (d) `unpack-linux`

FIGURE 6.12 – Profile mémoire construit à partir des allocations

La Figure 6.13 présente le ratio entre les instructions liés au calcul et les instructions liés à la mémoire. Les benchmarks `ffmpeg` et `scimark2` présentent légèrement plus d'instructions liés au calcul, ce qui est en accord avec le fait qu'ils appartiennent à la famille *processeur*. Cependant, on observe globalement un rapport de deux à trois fois plus d'instructions de calcul que d'instructions d'accès mémoire.

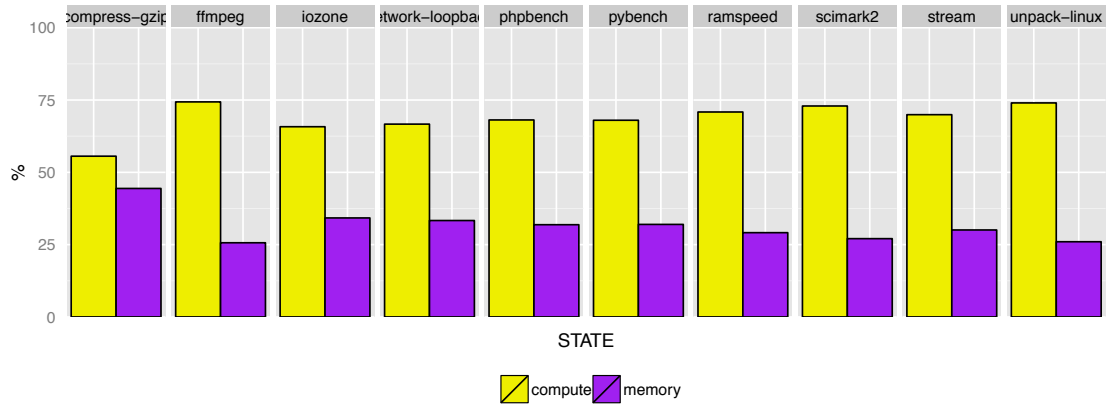


FIGURE 6.13 – Ratio des instructions de calcul et de mémoire

Famille	Moyenne	%	Écart-type
disque	4 130	0.18	1.73
mémoire	1 842 820	80.74	0.5
réseau	133	0.01	1.46
processeur	17 489	0.77	3.7
système	417 915	18.31	1.4

TABLE 6.7 – Résultats pour l'utilisation noyau du benchmark `compress-gzip`

Stabilité

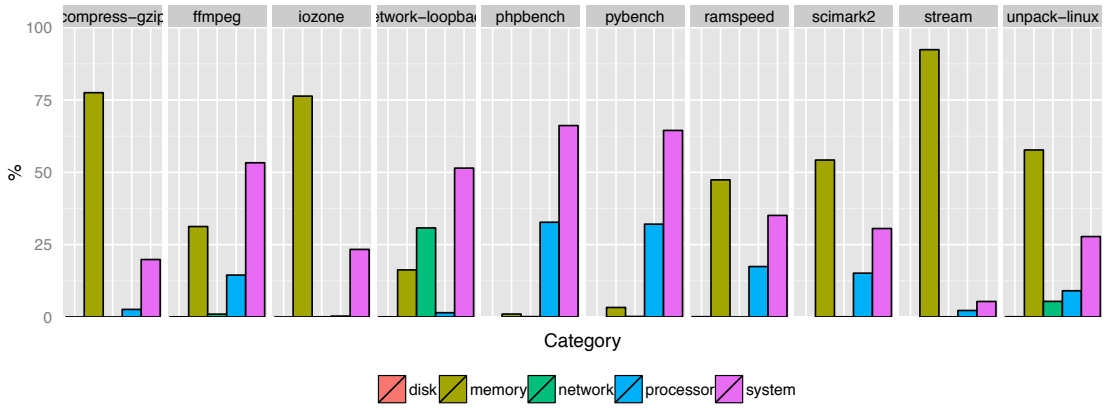
Pour chaque métrique présentée précédemment, nous avons présenté une moyenne sur les 32 mesures prises. L'écart-type des valeurs présentées est toujours inférieur à 1%. Ceci nous indique un comportement régulier des *benchmarks* observés et donc une exécution déterministe. La Table 6.7 montre un exemple de résultats pour l'utilisation du noyau du benchmark `compress-gzip`. Les valeurs moyennes obtenues sont donc représentatives du comportement réel sur une seule exécution des *benchmarks*. Une analyse sur un nombre moins élevé de traces peut se faire afin d'en alléger l'exécution pour des investigations ultérieures.

6.3.2 Résultats pour la carte Juno (ARM)

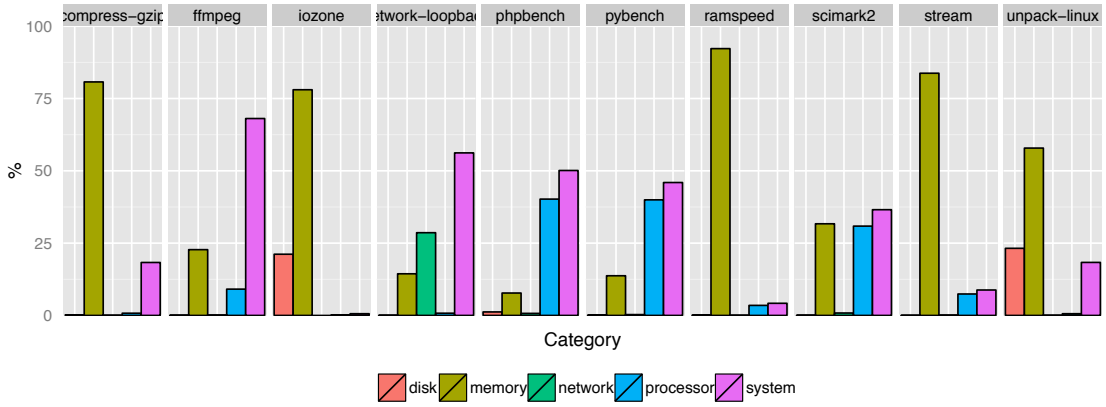
Utilisation noyau

Sur la Figure 6.14, nous présentons la répartition des événements noyau pour les deux plates-formes afin de pouvoir les comparer.

Les résultats sont similaires, on constate cependant l'absence d'événements liés aux *disque* pour la carte Juno. Les événements sont pourtant bien présents dans la trace originale, mais absents lorsque l'on filtre seulement les événements du benchmark. C'est en fait le processus `kworker`, processus noyau en charge d'alléger les processus générants trop d'interruptions, sur cette plate-forme qui s'est chargé des entrée/sorties liés au disque. Ceci montre une gestion du disque différente par rapport à la machine de bureau.



(a) Juno (ARM)



(b) Poste de travail (x86)

FIGURE 6.14 – Utilisation du noyau pour les 10 benchmarks

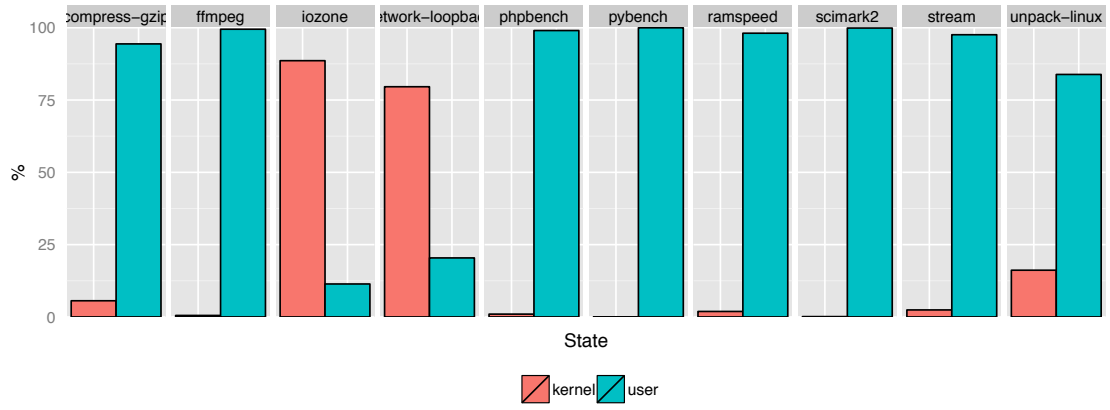
Temps noyau et utilisateur

Si on compare la répartition des du temps passé en mode utilisateur et en mode noyau entre les deux plates-formes (Figure 6.15), nous observons encore un comportement similaire.

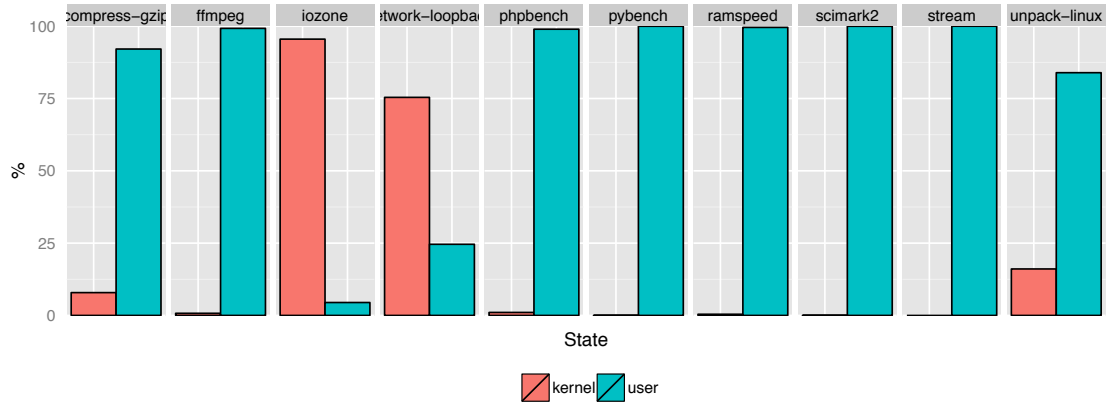
Utilisation des CPUs et parallélisme

La Figure 6.16 montre l'utilisation des processeurs pour une exécution par benchmark. Sur la carte Juno, les CPUs 0 et 1 sont les CPUs du SoC A57, qui a le rôle du processeur pour les performances de calcul dans l'architecture big.LITTLE. les CPUs 2 à 5 sont ceux du SoC A53, moins puissant et utilisé pour les performances énergétiques dans l'architecture big.LITTLE.

Les benchmarks utilisant le parallélisme utilisent toutes les ressources disponibles, malgré une différence entre les processeurs utilisés. Les CPUs 0-1 et 2-5 étant physiquement séparés, ils ne partagent pas de cache et donc des problèmes de performance peuvent être détectés par ces *benchmarks*.



(a) Juno (ARM)



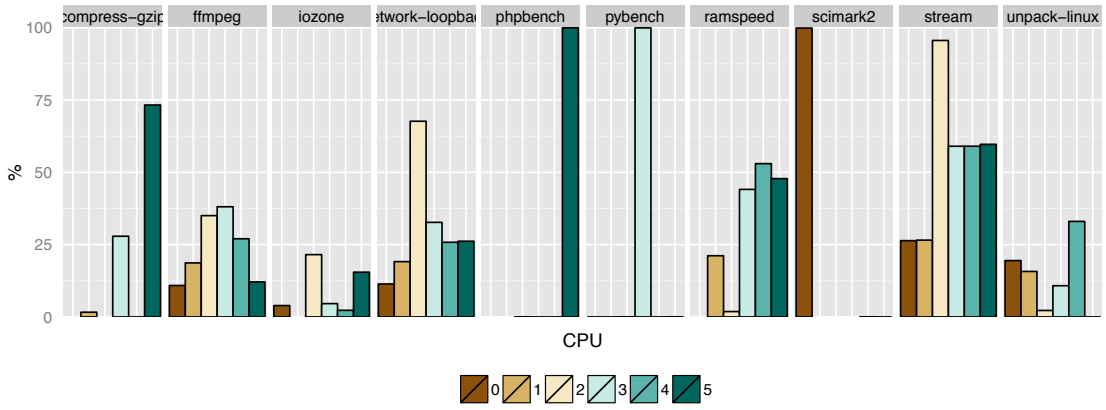
(b) Poste de travail (x86)

FIGURE 6.15 – Ratio du temps passé en mode utilisateur et noyau

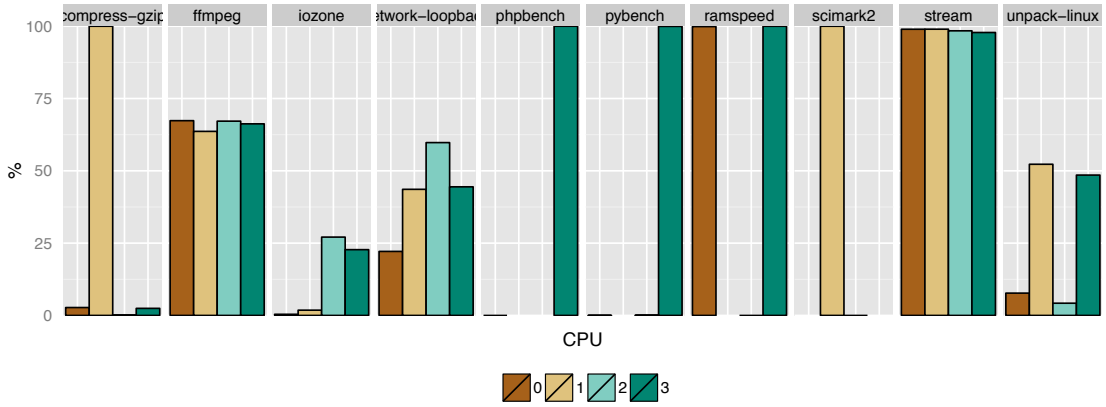
Sur cette exécution, les *benchmarks* `phpbench` et `pybench` utilisent un seul cœur, celui du SoC A53, moins puissant que les processeurs du A57. Sur une autre itération, les benchmarks pourraient utiliser un des coeurs dédiés aux performances de calcul du A57. Les résultats du benchmark seraient donc différents. L'hétérogénéité de la plate-forme n'est donc pas prise en compte lors de l'exécution de ces benchmarks.

Analyse mémoire et calcul

La Figure 6.17 montre le ratio entre les instructions de calcul et d'accès à la mémoire. Comparé à la machine x86, les benchmarks utilisent ici des instructions liés à la mémoire et au calcul de manière plus équilibrée. On note cependant que les benchmarks conservent de manière générale un plus grand nombre d'instructions de calcul sur les 2 plates-formes. Ceci s'explique par le fait que les instructions pour la machine de bureau (x86) permettent des accès mémoire lors d'instructions de calcul. Sur la carte Juno (ARM), le jeu d'instructions utilisé ne permet pas cette optimisation et nécessite des chargements explicites des données depuis la mémoire. Il en résulte donc plus d'opérations liées à la mémoire, et donc un rapport plus



(a) Juno (ARM)



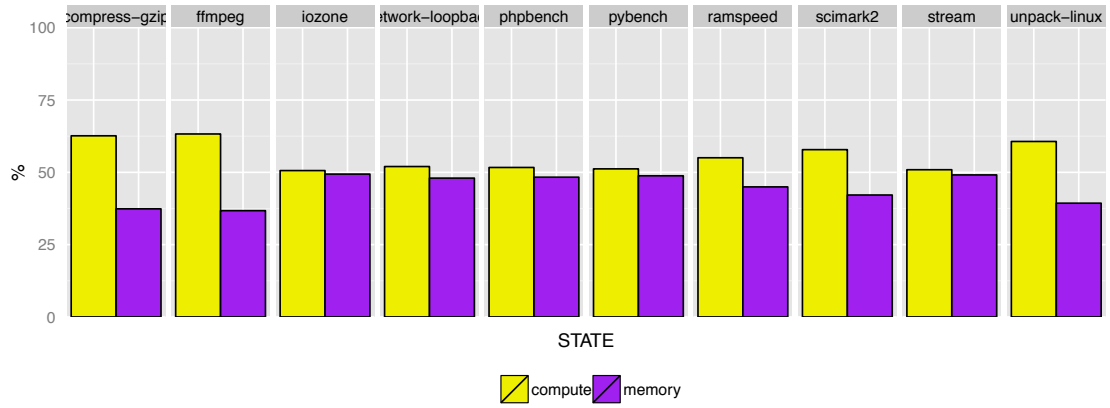
(b) Poste de travail (x86)

FIGURE 6.16 – Utilisation des différents processeurs pour une itération

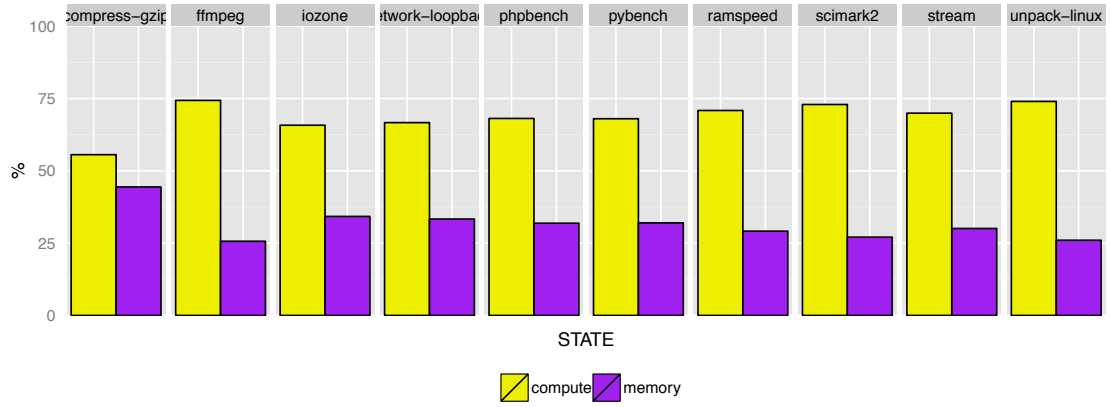
équilibré comparé aux résultats de la plate-forme de bureau. On note cependant que pour les *benchmarks* `compress-gzip`, `ffmpeg` et `scimark2`, de la *famille processeur*, ont un rapport moins équilibré en faveur des opérations de calcul que les autres *benchmarks* sur la carte Juno.

La Table 6.8 présente les résultats pour l'analyse mémoire des benchmarks sur la carte Juno. Les résultats du poste de travail sont mis entre parenthèses. On remarque un comportement similaire sur les deux plates-formes, que ce soit en terme d'allocations mémoire, mais aussi des appels inutiles à la fonction `free`. La différence de plate-forme ou encore de quantité de mémoire disponible n'a donc pas d'impact sur les benchmarks exécutés.

La Figure 6.18 présente les profils mémoire de 4 des benchmarks pour les 2 plates-formes. Pour chaque benchmark, on remarque une forme similaire entre les 2 plates-formes. Cela confirme une fois de plus que le changement de plate-forme d'exécution n'a qu'un faible impact sur le fonctionnement interne des *benchmarks*. On note cependant une petite différence notamment sur le nombre d'allocations pour le benchmark `stream`.



(a) Juno (ARM)

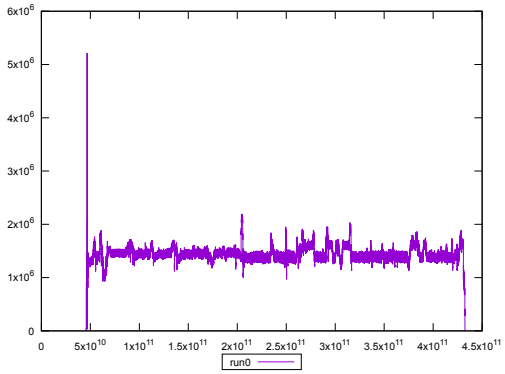


(b) Poste de travail (x86)

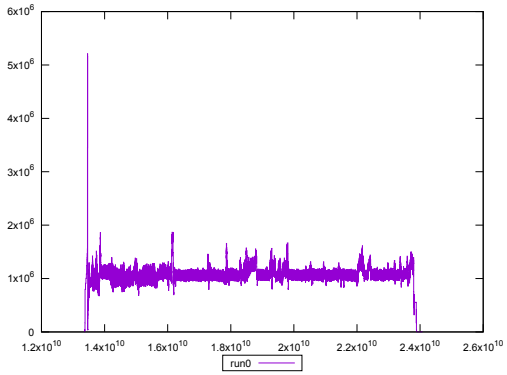
FIGURE 6.17 – Ratio des instructions de calcul et de mémoire

Benchmark	Mémoire maximum allouée (en Ko)		Appels à free inutiles	
compress-gzip	138	(134)	4	(5)
ffmpeg	5 215	(5 215)	216 544	(215 270)
iozone	37 751	(37 753)	3	(3)
network-loopback	1 063	(1 033)	317	(95)
phpbench	4 572	(4 577)	45	(45)
pybench	2 863	(2 864)	158	(42)
ramspeed	3 456 110	(3 456 110)	0	(4)
scimark2	16 780	(16 780)	0	(0)
stream	10	(10)	44	(404)
unpack-linux	3 697	(3 705)	495 306	(494 910)

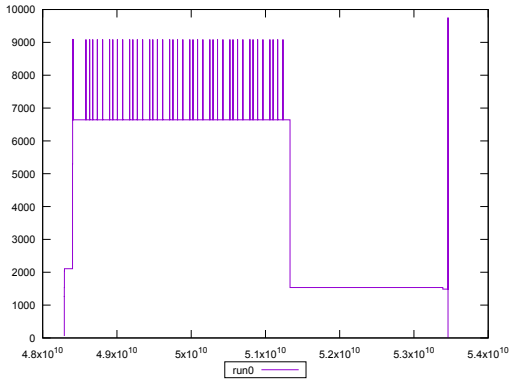
TABLE 6.8 – Analyse mémoire pour la carte Juno (ARM)



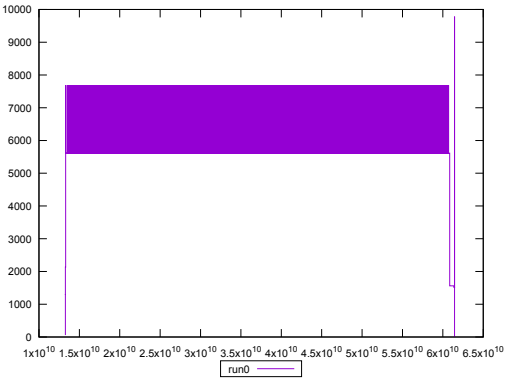
(a) `ffmpeg` Juno (ARM)



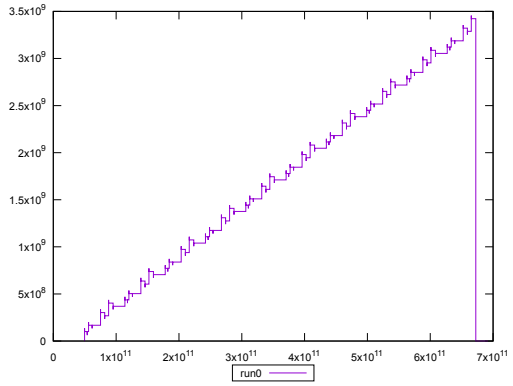
(b) `ffmpeg` Poste de travail (x86)



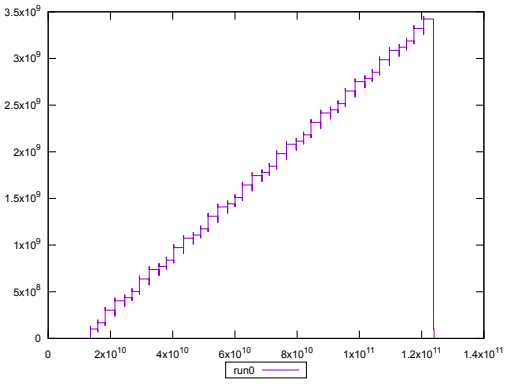
(c) `stream` Juno (ARM)



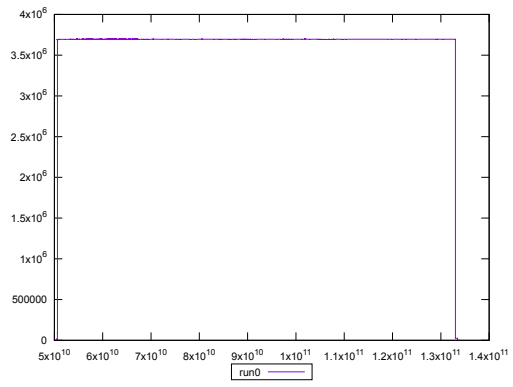
(d) `stream` Poste de travail (x86)



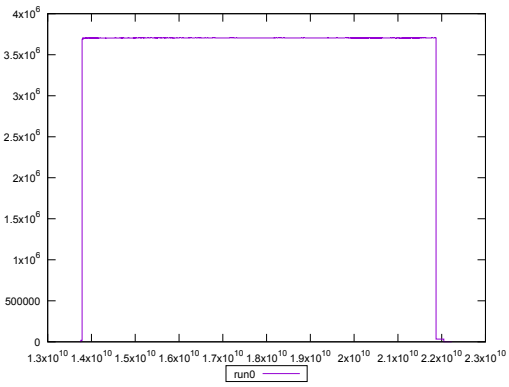
(e) `ramspeed` Juno (ARM)



(f) `ramspeed` Poste de travail (x86)



(g) `unpack-linux` Juno (ARM)



(h) `unpack-linux` Poste de travail (x86)

FIGURE 6.18 – Comparaison des profils mémoire Juno et poste de travail

6.3.3 Validation des performances de SWAT

La Table 6.9 présente les détails pour les traces de la machine de bureau. Pour chaque benchmark nous donnons la taille des traces obtenues en Mégaoctets, le nombre d'événements, le temps de l'analyse des traces ainsi que le temps d'exécution du benchmark. Ces valeurs sont la somme pour toutes les traces de chaque benchmarks, c'est à dire, pour les 32 itérations, et les 3 configurations ayant généré des traces.

Benchmark	Taille (Mo)	événements	Analyse (s)	Exécution (s)
compress-gzip	43 166	1 899 $\times 10^6$	6 626	2 431
ffmpeg	4 720	221 $\times 10^6$	1 018	1 927
iozone	17 227	770 $\times 10^6$	3 231	2 494
network-loopback	266 652	13 346 $\times 10^6$	69 369	2 386
phpbench	47 788	2 790 $\times 10^6$	14 595	9 927
pybench	45 222	2 732 $\times 10^6$	12 347	5 629
ramspeed	32 094	1540 $\times 10^6$	6 074	29 640
scimark2	1 921	111 $\times 10^6$	352	4 272
stream	6 289	330 $\times 10^6$	1 580	5 162
unpack-linux	14 139	643 $\times 10^6$	2 930	1 683

TABLE 6.9 – Informations sur les traces pour le poste de travail (x86)

Pour la machine de bureau, nous avons donc collecté environ 500 Gigaoctets de traces, qui représentent 25 milliards d'événements. Le temps d'analyse a représenté en moyenne le double du temps de l'exécution. Pour les benchmarks très intensifs en code noyau, comme network-loopback, il est normal que le nombre d'événements enregistré soit plus grand. Le temps d'analyse est donc beaucoup plus important que le temps d'exécution du benchmark. On constate cependant que le temps d'analyse et proportionnel au nombre d'événements de la trace. Notre workflow d'analyse à en moyenne traité environ 200 000 événements par seconde.

A titre de comparaison, nous avons aussi implémenté le même workflow que celui utilisé pour la validation du mécanisme d'exécution par tâches, présenté dans le chapitre précédent (Chapitre 5). Ce workflow lisais une trace et envoyé les événements dans un module qui comptait le nombre d'éléments de celui-ci. Pour la même analyse donc, et en utilisant la même trace, le temps d'exécution de SWAT est d'environ 10 minutes, soit 4 à 6 fois plus rapide que pour les implémentations du système de workflow en Python. De plus, la variation de taille du *buffer* servant à stocker les éléments, ne fait pas varier le temps d'exécution, contrairement à ce que nous avions constaté pour les différentes versions du mécanisme d'exécution de workflow en Python.

6.3.4 Validation de l'analyse générique de système

Grâce à notre modèle d'analyse générique, nous avons pu dresser un profil unifié pour différents benchmarks de la suite PTS. Ces benchmarks étant pourtant très

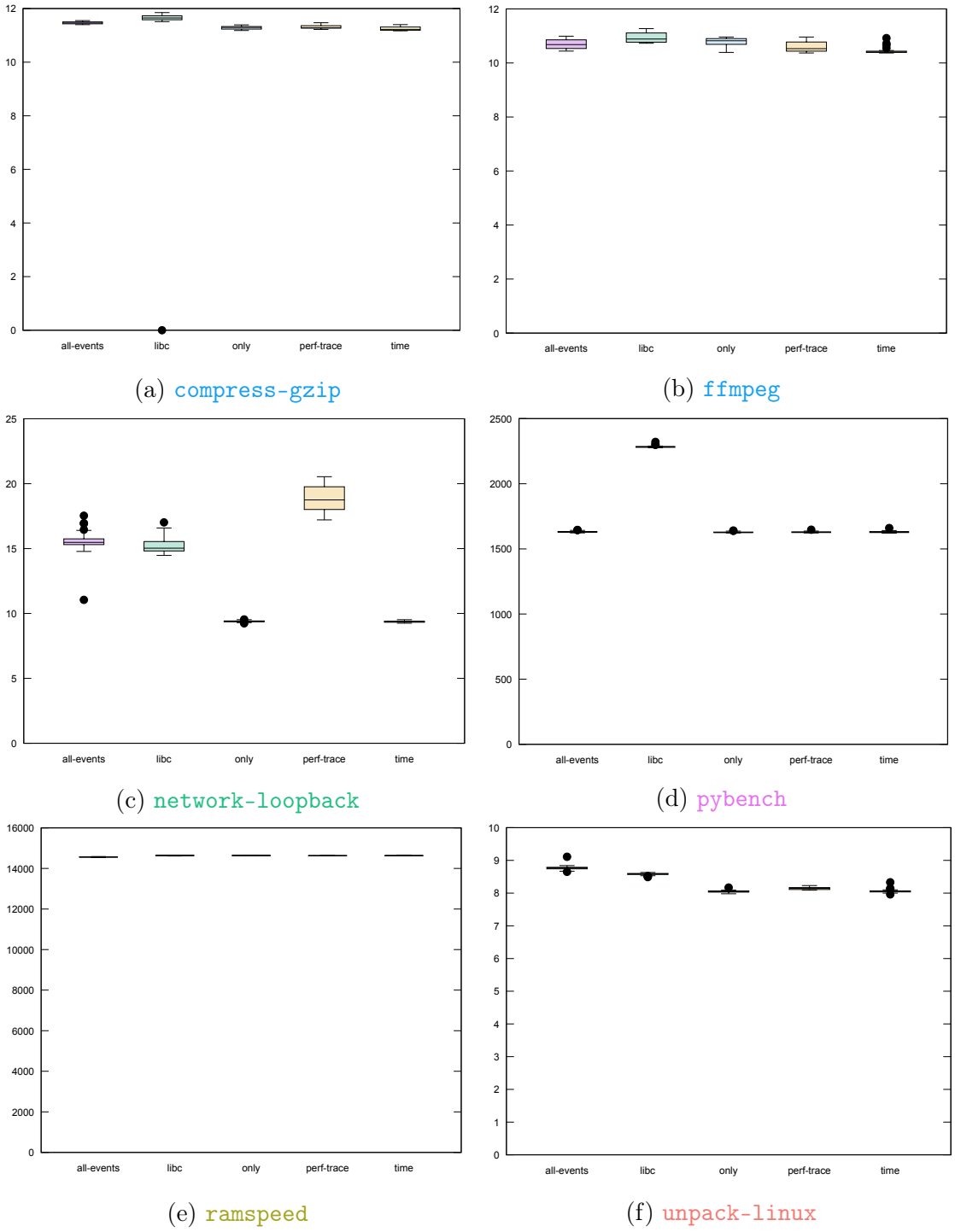
différents, notamment en terme de technologie utilisé (Python, PHP, etc.) nous avons pu, de manière automatique, récolter des données afin de comparer leur exécutions sur des critères similaires. De plus, ces expérimentations on pu être faites sur des plates-formes et des architectures différentes, et nous permettent de comparer le comportements du même programme sur des systèmes différents.

Ces profils nous ont aidé à mettre en évidence des différences entre certains benchmarks, pourtant appartenant à la même famille PTS. Ces différences concernent entre autres l'utilisation de la mémoire pour les benchmarks `ramspeed` et `stream`. Pour les benchmarks testant le processeur, on note aussi une différence entre les *benchmarks* `compress-gzip` et `scimark2` n'utilisant pas de parallélisme et `ffmpeg` qui utilise tous les processeurs à sa disposition. Pour les *benchmarks*testant le *disque*, `iozone` le temps passé à exécuter du code noyau est très différent. Les *benchmarks* `pybench`et `phpbench`, pourtant de familles différentes, ont montré quand à eux des comportement similaires quant à l'utilisation du noyau, la répartition du temps de calcule selon le mode utilisateur ou noyau et le parallélisme. Les observations faites sur la carte Juno montrent aussi que les aspects particulier des nouvelles plates-formes, tels que les processeurs hétérogènes, ne sont pas pris en compte lors de l'exécution des *benchmarks*, ce qui peut conduire à une mauvaise interprétation des performances du système. Au final, malgré un comportement déterministe sur plusieurs exécutions, et similaire sur des plates-formes différentes, l'étude des *benchmarks* de la suite PTS nous a montré des différences importantes au sein des *benchmarks* appartenant aux mêmes familles, et des similarités pour certains de familles différentes. La classification faite par PTS est une première approche permettant de cibler partiellement les tests, mais l'ajout de profils comme ceux que nous proposons aiderait les développeurs dans leurs choix.

En comparant les scores obtenus pour les différentes configurations, nous avons observé un faible surcoût généré par LTTng. Ceci ce retrouve en observant l'impact du traçage au niveau du score calculé par le benchmark. La Figure 6.19 montre des diagrammes en boîte pour les résultats des benchmarks sur le poste de travail. Seul le traçage des fonctions liés à la mémoire influe parfois sur l'exécution comme pour le benchmark `pybench`. Ceci peut s'expliquer par le fait que les événements interviennent au niveau utilisateur. Ces événements sont enregistrés dans la trace en utilisant des mécanismes d'encapsulation de fonctions qui au final sont plus coûteux que l'implémentation des tracepoints du noyau. Pour le benchmark `network-loopback`, on remarque aussi que le traçage induit une perte de performance. Pour ce benchmark, le nombre excessif d'événements collecté, environ 8 milliards par traces explique cette dégradation du score obtenue.

6.3.5 Retour d'expérience

Nous avons travaillé sur plusieurs plates-formes embarquées différentes, trois au total. Nous avons tout d'abord travaillé sur la carte UDOO [18], puis une carte Nvidia Jetson TK1 [8], et enfin sur la carte Juno ARM64 présentée dans ce chapitre. Chaque carte intègre du matériel spécifique, lié aux besoins du système. Par exemple, les cartes réseaux utilisées ne sont pas toujours "standard", de même pour



(a) `compress-gzip` (b) `ffmpeg`

(c) `network-loopback` (d) `pybench`

(e) `ramspeed` (f) `unpack-linux`

FIGURE 6.19 – Variation du score des benchmarks selon les configurations (x86)

les contrôleurs vidéos. Contrairement aux système x86 classiques, les noyaux Linux utilisés sont compilés spécifiquement pour chaque plate-forme, afin d'intégrer les pilotes nécessaires au fonctionnement des différents composants. Ces différentes cartes sont donc fournies avec un système Linux spécifique. Les outils que nous voulions utiliser n'étaient pas toujours compatibles avec ces spécificités. Par exemple, l'utilisation des compteurs de performance requière une configuration particulière

du noyau Linux, qui doit entièrement être compilé à nouveau si celle-ci n'est pas activée. Nous avons donc utilisé un noyau générique "multi-plates-formes" : le noyau `armmp` disponible dans les dépôts Debian. Ce noyau utilise un fichier *Device Tree* décrivant la plate-forme afin d'utiliser les bon pilotes selon la plate-forme sur laquelle il s'exécute. L'utilisation de ce noyau à tout de même nécessité des modifications sur les cartes, notamment la mise à jour et la configuration du *bootloader*, ou encore la désactivation manuelle de certain pilotes qui rendaient une des plate-forme instable.

La taille des traces générées fut un réel défi tant en terme de stockage, que de transfert et de traitement. En effet nous avons présenté ici des résultats sur un ensemble de traces, représentant environ un Téraoctet de données, mais plusieurs itérations on dû être faites tout au long du développement de notre processus d'analyse. Par exemple, pour certains benchmarks très intensifs en production d'événements, une partie de ces événements étaient perdue au moment du traçage à cause d'une mauvaise configuration de LTTng et de la taille des *buffers* utilisés. Différentes traces ont dû être générées aussi lors des changements des versions de noyau afin d'obtenir des traces comparables sur les différentes architectures. C'est donc au final beaucoup de données qui ont été générées, dont certaines sur des plates-formes distantes (la carte Juno étant sur une grille de calcul) incluant un temps de transfert de ces données non négligeable. Les premières analyses ont été faites sur des implémentations en Python présentés dans le chapitre précédent, moins performant que notre outil, le temps d'analyse était lui aussi conséquent et se comptait en jours. Malgré les performances de notre infrastructure, les analyses nécessitent tout de même plusieurs heures de calcul.

Au vu du nombre de plates-formes, de benchmarks, de configurations et du nombre d'exécutions, nous avons automatisé la phase de traçage, qui lui aussi nous a posé problème. En effet, malgré une attention particulière sur le fait d'utiliser des systèmes similaires sur les différentes plates-formes, leur configuration peut différer. Le benchmark `network-loopback`par exemple utilise en interne le programme `netcat`. Or plusieurs versions de ce programme sont disponibles et selon les machines, la version "classique" de `netcat` peut être installée ou alors la version BSD. Le benchmark `network-loopback`utilise un paramètre du programme disponible que dans la version BSD, pour les plates-formes ou l'autre version est installée, il en résulte donc une exécution erronée, qui de plus n'est pas notifiée par PTS.

Nous pouvons enfin citer un dernier problème que nous avons rencontré, concernant les compteurs de performance mal configurés. En effet, l'accès aux compteurs de performance est spécifique à chaque processeur, c'est au noyau de faire le lien entre pour pouvoir y accéder. Sur la plate-forme ARM, nous avions constamment un nombre égal pour les compteurs `L1-dcache-loads` et `L1-dcache-stores`, devant compter les accès en lecture et en écriture au cache L1. Après avoir investigué les compteurs matériels nous nous sommes rendus compte que cette valeur était en fait la valeur du compteur `L1-dcache-access` qui représente déjà la somme des accès en lecture et en écriture. Nous avions donc le double d'événements du nombre d'instruction mémoire dans nos analyse lorsque l'on faisait la somme des deux compteurs.

6.4 Conclusion

Nous avons proposé une implémentation en C++ de notre modèle d'analyse par workflow. Cette implémentation propose les interfaces de programmation nécessaires afin de facilement pouvoir l'adapter à tout types d'analyse. Nous avons conçu des modules dédiés à l'analyse de traces. Nous avons ensuite construit en utilisant cette infrastructure un workflow complexe, composé de ces modules d'analyse, afin de calculer des profils d'utilisation du système. Ce profil renseigne des informations telles que l'occupation CPU, le parallélisme, l'utilisation détaillée des ressources noyau ainsi que de la mémoire.

Notre infrastructure d'analyse nous a permis de traiter environ 50 milliards d'événements contenus dans nos traces, ce qui représente près d'un Téraoctet de données. Cela à été possible grâce aux mécanismes de transfert par flux des données mis en place dans l'implémentation du workflow. Au final nous avons pu analyser un sous-ensemble de *benchmarks* issus de la suite PTS. Nous avons pu comprendre leur fonctionnement en détail sans aucune information préalable. Le fait que l'infrastructure puisse supporter le traitement de grandes quantités de données nous a enfin permis de faire des analyses sur de nombreuses exécutions des benchmarks, permettant de mettre en évidence une stabilité quant au comportement de ceux-ci.

L'outil que nous proposons actuellement se présente sous la forme du mécanisme d'exécution de workflow. L'interface de construction du workflow se fait de manière programmatique. A terme, il pourrait être intégré dans Vistrails afin de remplacer le mécanisme d'exécution de workflow actuel. Ce remplacement permettrait de tirer parti des avantages de Vistrails tels que la méthode graphique de constructions de workflow, la gestion d'historique, de cache, ou encore la gestion de provenance des données. Ainsi, Vistrails permettrait l'exécution de workflows basés sur les méthodes de streaming, tout en proposant une exécution performante.

Chapitre 7

Conclusion et perspectives

7.1 Objectifs de la thèse

L'objectif de cette thèse est de proposer des méthodes et outils génériques pour la validation des systèmes embarqués via des traces d'exécution.

Avec la présence croissante des systèmes embarqués dans notre quotidien, et leur complexification au fil du temps, la validation de ces systèmes est devenue une étape capitale. L'utilisation des traces d'exécution pour la validation a prouvé son efficacité face aux validations par méthodes formelles, trop coûteuses à mettre en place pour des systèmes aussi complexes. Cependant, les outils actuellement utilisés pour analyser les traces d'exécution ne sont pas suffisants. Souvent trop spécifiques, ou restreints en termes de fonctionnalités, les développeurs sont confrontés aux difficultés de l'utilisation de plusieurs outils, non prévus pour un fonctionnement collaboratif. Dès lors, la mise en place de mécanismes de conversion de formats de traces est nécessaire, ainsi qu'un traitement des données produites par ces outils.

Dans cette thèse, nous proposons un environnement d'analyse dont le but est d'unifier et d'automatiser les traitements et analyses des traces d'exécution. En proposant un format de trace générique et enrichi en sémantique, nous permettons des analyses poussées prenant en compte des informations spécifiques et permettons un premier niveau d'analyse automatique. Notre environnement d'analyse étant basé sur un mécanisme de *workflow*, nous permettons une construction d'analyses simplifiée, ainsi qu'une exécution automatisée et collaborative, utilisant plusieurs composants d'analyse. Enfin, nous proposons une méthodologie d'analyse générique de systèmes Linux, fondée sur notre modèle de données et d'analyse par workflow, permettant la compréhension du fonctionnement d'applications sans aucune information *a priori*, et s'appliquant au cas des systèmes embarqués.

7.2 Proposition, réalisation et validation

L'étude des outils d'analyse de traces nous a permis de mettre en évidence les lacunes de ces outils, notamment concernant les aspects de collaboration ainsi que la personnalisation des traitements appliqués aux données. L'étude des outils d'analyse basés sur les workflows, nous a permis de mettre en évidence les critères nécessaires pour la mise en place d'un environnement d'analyse de traces à base de workflows. Ces critères regroupent notamment les aspects de gestion de grandes quantités de données, ou encore la modularité relative à la construction du workflow.

Nous avons proposé un modèle de représentation des traces d'exécution enrichi en sémantique. Ce modèle de trace permet de structurer les informations relatives aux événements, et ainsi d'exprimer des notions plus complexes. Ces notions sont basées sur ce qui se fait déjà dans les outils, en interne, sur les représentations intermédiaires de la trace et de manière non standardisée. Le modèle que nous proposons permet de représenter des événements ponctuels, des événements représentant un état du système borné dans le temps, des événements de causalité entre deux entités, ainsi que des événements représentant l'évolution d'une valeur au cours du temps. Nous avons conçu un outil d'analyse exploitant les notions que nous proposons dans le modèle. Cet outil, que nous avons développé et intégré au sein de l'outil Framesoc du projet SoC-TRACE, permet de retrouver des corrélations entre événements. Nous avons montré, sur des traces issues de cas d'usages proposés par la société STMicroelectronics, partenaire du projet SoC-TRACE, que l'utilisation de notre modèle de traces est utile lors de l'analyse des systèmes. Les traces, que nous avons enrichies de manière semi-automatique, puis analysées *via* l'outil de corrélation nous ont permis de retrouver la cause des anomalies présentes dans la trace. Ces résultats ont été validés avec les équipes de développement de STMicroelectronics.

Suite à l'étude des outils d'analyse de traces d'exécution, et de outils d'analyse de données par workflow, nous avons proposé un modèle d'analyse de traces à base de workflows. Utilisant des composants à connecter les uns aux autres, ce modèle permet de définir des analyses complexes. Les connexions, représentant des flux de données, utilisent des mécanismes de *streaming* permettant de traiter de grandes quantités de données en continu. Nous avons réalisé une première implémentation de ce modèle en collaboration avec l'équipe ViDA de l'Université de New-York. Cette équipe est en charge du développement de VisTrails, un outil de workflow pour l'analyse spécialisé dans la simulation, l'exploration, et la visualisation de données. Notre implémentation a permis le traitement de grosses traces d'exécution, chose impossible dans la version officielle actuelle de l'outil VisTrails. Nous avons validé notre proposition et cette première implémentation sur un exemple synthétique, montrant la simplicité de construction d'analyse par workflow, ainsi que l'efficacité de traitement des données par mécanisme de *streaming*. Nous sommes actuellement en discussion avec l'équipe de développement de VisTrails sur l'évolution de son moteur d'exécution afin d'intégrer les mécanismes de streaming proposés.

Nous avons proposé, enfin, un modèle d'analyse générique dédié à l'observation des programmes sur un système Linux. En utilisant les interfaces du noyau Linux, ainsi que de la bibliothèque standard C, nous avons proposé une méthode de traçage

d'applications. La méthode n'a pas besoin de connaissances préalables sur les applications et n'impose pas de modifications sur celles-ci. Les traces obtenues permettent une compréhension des aspects d'utilisation des ressources de calcul, de la mémoire, et de l'utilisation des fonctions du noyau. Nous avons réalisé le prototype *Streamed Workflow Analysis Tool* (SWAT) fournissant une seconde implémentation de notre modèle de workflow en tant qu'outil indépendant. Forts de l'expérience avec l'outil VisTrails, ce prototype synthétise notre compréhension de ce que devrait être un workflow pour l'analyse de traces et bénéficie de nombreuses optimisations d'implémentation. Nous avons utilisé SWAT pour mettre en place notre workflow d'analyse de performances et l'avons instancié dans le cadre du benchmark *Phoronix Test Suite* (PTS). Nous avons pu collecter et analyser des traces de manière automatique sur les différents benchmarks. Cette étude a confirmé la nécessité de compréhension du fonctionnement de benchmarks, car ils adoptent des usages très différents des ressources systèmes. La généricité de cette analyse a permis d'observer des programmes utilisant des technologies différentes, mais aussi l'observation d'un même programme sur des architectures différentes. Nous avons réussi à traiter près de 1TB de traces et avons expérimenté sur quatre différentes plateformes comprenant des machines de travail standard et des dispositifs embarqués.

7.3 Perspectives

Plusieurs améliorations sont directement envisageables suite à nos travaux.

Concernant le modèle de données, dans notre travail, la structuration de la trace est faite une seule fois en utilisant une interprétation possible des données brutes. Or, ces données brutes peuvent être structurées de différentes manières reflétant différentes vues de l'exécution. Par exemple, nous avons utilisé les informations d'ordonnancement pour représenter l'exécution des processus. Il serait possible de structurer la trace en prenant en compte les utilisateurs ou des aspects énergétiques. Il serait donc intéressant de voir comment faire co-exister plusieurs structurations de la même trace.

Concernant notre outil SWAT, les performances obtenues sont prometteuses, et une intégration dans l'outil VisTrails est prévue. Ceci nécessite néanmoins une analyse sur l'adaptation des des différentes fonctionnalités de VisTrails, comme par exemple la gestion d'un cache de calcul, au transfert de données par *streaming*.

Nous avons proposé un ensemble de modules d'analyse que nous avons exploité dans nos différentes expérimentations. L'enrichissment de cet ensemble avec des modules diverses de calcul statistique, de fouille de données ou de visualisation est un développement naturel de ce travail.

Concernant notre modèle d'analyse générique, celle-ci pourrait exploiter d'autres métriques de performances. Par exemple, l'utilisation plus large des fonctions de la librairie standard C, les interruptions matérielles ou encore le détail des appels systèmes. De plus, LTTng permet d'envoyer une sur le réseau, permettant le traitement sur une autre machine par exemple. Nous avons montré que le temps d'analyse était en moyenne deux fois le temps d'exécution, cependant la taille des traces (plusieurs centaines de gigaoctets) et la quantité d'événements (plusieurs dizaines de millions)

explique cela. Il serait possible de créer un workflow d'analyse plus simple, sur des traces moins conséquentes, utilisant une trace générée en continu, afin de proposer un outil d'analyse en temps réel.

Enfin, une analyse exhaustive de tous les benchmarks de la suite PTS serait possible afin de proposer les profils d'analyse aux utilisateurs. Ceux-ci pourrait alors choisir quels benchmarks et quelles configurations utiliser en comprenant pleinement le fonctionnement de ces benchmarks.

Nous sommes convaincus de l'utilité des systèmes de workflow pour les outils d'analyse de données et nous avons exploré une première voie d'application dans le domaine d'analyse des traces d'exécution de systèmes embarqués. Néanmoins, une généralisation de l'approche nécessiterait une investigation sur plusieurs volets.

Du point de vue du domaine d'application, nous pensons que les méthodes d'analyse doivent explicitement permettre la capture de la sémantique métier. Dans notre cas, pour que les worklflows d'analyse soient largement diffusés et utilisés dans le contexte des systèmes embarqués, un travail important devrait être effectué sur les types de nouvelles analyses de traces à mettre en place. Or, comment représenter la sémantique des système embarqués tout en incluant des asepcts d'architecture, ainsi que des asepcts applicatifs ?

De point de vue de l'analyse de traces, nous ne sommes qu'aux phases exploratoires d'observation de metriques d'exécution et d'analyse approfondie. En effet, dans une majorité d'outils les metriques reflètent directement l'exécution bas niveau d'un système et les analyses proposées n'elèvent pas le niveau d'abstraction. La question de quels métriques pour quelles analyses est toujours d'actualité.

De point de vue d'un système de workflow pour l'analyse de traces, il faut avancer vers une compréhension plus poussée des mécanismes et techniques permettant de bonnes performances tout en assurant les propriétés de reproductibilité et de généricité.

Table des figures

2.1 Control-flow capturant le processus d'évaluation de papiers 21
2.2 Dataflow capturant le processus de commande et gestion de repas . . 22

3.1 Extrait de trace au format LTTng 31
3.2 Représentation des événements dans Framesoc 32
3.3 Représentation des quatre natures d'événements 33
3.4 Représentation sans et avec sémantique 34
3.5 Exemple de connexion entre deux modules 36
3.6 Mécanismes de flux de données 36
3.7 Modes de transfert des données 37
3.8 Workflow abstrait de l'analyse générique d'un système Linux 39
3.9 La collecte de trace pour la caractérisation générique 40

4.1 Filtrage des événements `SoftIRQ` selon leur durée. 48
4.2 Filtrage des événements `TSrecord` selon la durée entre deux appels consécutifs. 49
4.3 Corrélation visuelle pour *Unicast* 50
4.4 Corrélation visuelle pour *TSrecord* 51
4.5 Corrélation peu lisible due à une grande différence du nombre d'événements entre les deux ensembles. 53
4.6 En bleu, zones aux alentours des anomalies, elles-mêmes représentées par des barres verticales 53
4.7 Architecture logicielle de Framesoc 55
4.8 Exemple de paramètres d'entrée 56
4.9 Exemple de résultat par corrélation avec découpage régulier 57
4.10 Exemple de résultat par corrélation avec découpage non régulier . . . 58

5.1 Interface graphique de l'outil VisTrails. 60

5.2 Exemple de module VisTrails . . . 61
5.3 Historique d'un workflow dans VisTrails. . . . 62
5.4 Module de manipulation de trace dans VisTrails . . . 64
5.5 Workflow décrivant l'analyse via une base de données dans VisTrails. 65
5.6 Exemple d'exécution d'un workflow . . . 66
5.7 Diagramme UML de la partie ordonnancement . . . 68
5.8 Exemple de module avec la nouvelle interface de programmation . . . 69
5.9 Exemple d'exécution d'un workflow . . . 70
5.10 Diagramme UML des tâches . . . 71
5.11 Automate à états représentant l'exécution d'un module . . . 71
5.12 Analyse d'une trace d'exécution en utilisant le streaming . . . 72
5.13 Création d'un workflow d'analyse via une interface programmatique . 73

6.1 Représentation du buffer circulaire . . . 79
6.2 Exemple d'instanciation d'un workflow . . . 79
6.3 Création d'un sous-workflow dans un module . . . 80
6.4 Workflow principal de l'analyse . . . 84
6.5 Détail du sous-workflow `GetPerfValues` . . . 84
6.6 Détail du sous-workflow `PerfAllRuns` . . . 85
6.7 Répartition de l'utilisation du noyau du benchmark `compress-gzip` . 89
6.8 Catégorisation des événements noyau pour les 10 benchmarks . . . 90
6.9 Ratio du temps passé en mode utilisateur et noyau . . . 90
6.10 Utilisation des processeurs en moyenne sur 32 exécutions . . . 91
6.11 Utilisation des processeurs pour une itération . . . 92
6.12 Profile mémoire construit à partir des allocations . . . 93
6.13 Ratio des instructions de calcul et de mémoire . . . 94
6.14 Utilisation du noyau pour les 10 benchmarks . . . 95
6.15 Ratio du temps passé en mode utilisateur et noyau . . . 96
6.16 Utilisation des différents processeurs pour une itération . . . 97
6.17 Ratio des instructions de calcul et de mémoire . . . 98
6.18 Comparaison des profils mémoire Juno et poste de travail . . . 99
6.19 Variation du score des benchmarks selon les configurations (x86) . . . 102

Liste des tableaux

2.1 Synthèse des outils d'exploitation de traces. 20
2.2 Synthèse des outils de workflow. 26

5.1 Temps d'exécution du workflow selon plusieurs configurations 74

6.1 Répartition des benchmarks par familles dans PTS 81
6.2 Liste des benchmarks utilisés 82
6.3 Exemples de scores PTS 82
6.4 Catégorisation des tracepoints du noyau 86
6.5 Présentation des familles observées 89
6.6 Mémoire maximum et appels inutiles à la fonction free 92
6.7 Résultats pour l'utilisation noyau du benchmark `compress-gzip` . . . 94
6.8 Analyse mémoire pour la carte Juno (ARM) 98
6.9 Informations sur les traces pour le poste de travail (x86) 100

Bibliographie

[1] Apache Storm. http://storm.apache.org.

[2] ARM64 Juno. http://www.arm.com/products/tools/development-boards/.

[3] Correlation (in statistics). a.v. prokhorov (originator), encyclopedia of mathematics. http://www.encyclopediaofmath.org/index.php?title=Correlation_(in_statistics)&oldid=11629.

[4] Framesoc. https://github.com/soctrace-inria/framesoc.

[5] Ftrace. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[6] Intel VTune. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[7] LTTng. http://lttng.org.

[8] Nvidia TegraK1. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html.

[9] Paraver. http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview.

[10] Perf. https://perf.wiki.kernel.org/index.php/Main_Page.

[11] Phoronix Test Suite. http://www.phoronix-test-suite.com/.

[12] Projet R. http://www.r-project.org/.

[13] Projet SoC-TRACE. http://www.minalogic.com/fr/projet/soc-trace.

[14] Projet SoC-Trace. http://soc-trace.minalogic.net.

[15] Strace. https://en.wikipedia.org/wiki/Strace.

[16] STWorkbench. http://stlinux.com/stworkbench/.

[17] TraceCompass. http://tracecompass.org.

[18] UDOO. http://udoo.org.

[19] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[20] *Measuring Computer Performance : A Practitioner's Guide*. Cambridge University Press, New York, NY, USA, 2000.

[21] Luay Alawneh and Abdelwahab Hamou-Lhadj. MTF : A Scalable Exchange Format for Traces of High Performance Computing Systems. *2011 IEEE 19th International Conference on Program Comprehension*, pages 181–184, jun 2011.

[22] JoseBacelar Almeida, MariaJoao Frade, JorgeSousa Pinto, and Simao Melo de Sousa. An overview of formal methods tools and techniques. In *Rigorous Software Development*, Undergraduate Topics in Computer Science, pages 15–44. Springer London, 2011.

[23] Dieter An Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A Unified Performance Measurement System for Petascale Applications. In *Proc. of the CiHPC : Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, number June 2010, pages 85–97. Springer, 2012.

[24] Adam Barker and Jano Van Hemert. Scientific Workflow : A Survey and Research Directions. *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, pages 746–753, 2008.

[25] Friedrich L. Bauer. From specifications to machine code : Program construction through formal reasoning. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 84–91, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[26] Christian Bienia and Sanjeev Kumar. PARSEC vs. SPLASH-2 : A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. *2008 IEEE International Symposium on Workload Characterization*, pages 47–56, oct 2008.

[27] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos Scheidegger, Claudio T Silva, and Huy T Vo. VisTrails : Visualization meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 745–747, 2006.

[28] Fernando Chirigati, Dennis Shasha, and Juliana Freire. ReproZip : Using Provenance to Support Computational Reproducibility. In *USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[29] Leonaxdo Dagum and Ramesh Menon. OpenMP : An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5 :46–55, 1998.

[30] Andrew Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science and Engineering*, 14(4) :48–56, 2012.

[31] Ewa Deelman, Gurmeet Singh, Mei-hui Su, James Blythe, Yolanda Gil, Carl Kesselman, G Bruce Berriman, John Good, Anastasia Laity, Joseph C Jacob, and Daniel S Katz. Pegasus : a Framework for Mapping Complex Scientific Workflows onto Distributed Systems Decisions that need to take place in Workflow Mapping. *Scientific Programming*, 13(3), 2005.

[32] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, 2009.

[33] Mathieu Desnoyers and Michel R Dagenais. The LTTng tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux. In *Ottawa Linux Symposium*, 2006.

[34] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-temam. Aftermath : A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *7th workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, number 1, pages 1–13, 2014.

[35] Ulrich Drepper and Red Hat. What Every Programmer Should Know About Memory. 2007.

[36] Anchor Fastener. VISMASHUP : Streamlining the Creation of Custom Visualization Applications. *Online*, (October), 2009.

[37] Jean-daniel Fekete. Software and Hardware Infrastructures for Visual Analytics. *IEEE Computer*, 43(8) :1–7, 2013.

[38] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera : a virtual data system for representing, querying, and automating data derivation. *Proceedings 14th International Conference on Scientific and Statistical Database Management*, 2002.

[39] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. In *11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[40] Markus Geimer, Felix Wolf, Brian J N Wylie, Erika Abraham, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. In *In International Workshop on Scalable Tools for High-End Computing (STHEC*, number 01, pages 702–719, 2008.

[41] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An Overview of Workfow Management : From Process Modeling to Work ow Automation Infrastructure. *Work*, 153 :119–152, 1995.

[42] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *Computer*, 40(12) :24–32, 2007.

[43] Francis Giraldeau, Julien Desfossez, David Goulet, Michel Dagenais, and Mathieu Desnoyers. Recovering system metrics from kernel trace. *OLS (Ottawa Linux symposium)*, pages 109–116, 2011.

[44] Abdelwahab Hamou-lhadj, Syed Shariyar Murtaza, Waseem Fadel, Ali Mehrabian, Mario Couture, and Raphael Khoury. Software Behaviour Correlation in a Redundant and Diverse Environment Using the Concept of Trace Abstraction. pages 328–335, 2013.

[45] IEEE. Computer July 2013. (July 2013), 2013.

[46] R Jain. *The Art Of Computer Systems Performance Analysis*. Wiley India Pvt. Limited, 1991.

[47] Ajay Joshi, Student Member, Aashish Phansalkar, and Student Member. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE TRANSACTIONS ON COMPUTERS*, 55(6) :769–782, 2006.

[48] Jacques Chassin De Kergommeaux and Benhur De Oliveira Stein. Pajé : An Extensible Environment for Visualizing Multi-threaded Programs Executions. In Roland Bode, Arndt and 0002, Thomas Ludwig and Karl, Wolfgang and Wismüller, editor, *Euro-Par*, pages 133–140. Springer, 2000.

[49] Jihie Kim, Yolanda Gil, and Marc Spraragen. A Knowledge-Based Approach to Interactive Workflow Composition. *Icaps - 04*, (Icaps 04), 2004.

[50] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. Introducing the Open Trace Format (OTF). In Jack Alexandrov, Vassil N. and van Albada, G. Dick and Sloot, Peter M. A. and Dongarra, editor, *International Conference on Computational Science (2)*, pages 526–533, 2006.

[51] Andreas Knüpfer, Holger Brunst, and Ronny Brendel. Open Trace Format Specification. 2009.

[52] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Holger Mickler, S M Matthias, and Wolfgang E Nagel. The Vampir Performance Analysis Tool-Set. In Alexander Resch, Michael M. and Keller, Rainer and Himmler, Valentin and Krammer, Bettina and Schulz, editor, *Parallel Tools Workshop*, pages 139–155. Springer, 2008.

[53] Johan Kraft, Anders Wall, and Holger Kienle. Trace Recording for Embedded Systems : Lessons Learned from Five Industrial Projects. In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*, 2010.

[54] Bertram Ludäscher, Chad Berkley, Matthew Jones, and Edward A Lee. Scientific Workflow Management and the Kepler System. *Concurrency and Computation : Practice and Experience*, 0078296 :1–19, 2005.

[55] Vania Marangozova-Martin. *Duplication et cohérence configurables dans les applications réparties à base de composants*. PhD thesis, 2003.

[56] Vania Marangozova-martin. SoC-TRACE : Handling the Challenge of Embedded Software Design and Optimization. 2012.

[57] Alexis Martin and Vania Marangozova-Martin. Analyse de traces d'exécutions pour les systèmes embarqués : détection d'anomalies par corrélation temporelle. Technical Report RT-0450, Inria, October 2014.

[58] Alexis Martin and Vania Marangozova-Martin. Automatic Benchmark Profiling Through Advanced Trace Analysis. In *Euro-Par*, volume 9833, pages 659–671, 2016.

[59] Alexis Martin, Generoso Pagano, Jérôme Correnoz, and Vania Marangozova-Martin. Analyse de systèmes embarqués par structuration de traces d'exécution. In *ComPAS'2014*, Neuchâtel, Suisse, apr 2014.

[60] Wolfgang E Nagel, A Arnold, and M Weber. VAMPIR : Visualization and Analysis of MPI Resources. *Supercomputer*, 12 :69–80, 1996.

[61] Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. Trace Management and Analysis for Embedded Systems. *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pages 119–122, sep 2013.

[62] Generoso Pagano and Vania Marangozova-Martin. SoC-Trace Infrastructure. Technical Report July, 2012.

[63] Cesare Pautasso and Gustavo Alonso. Parallel computing patterns for grid workflows. *2006 Workshop on Workflows in Support of Large-Scale Science, WORKS '06*, 2006.

[64] K Pearson. Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material. 1896.

[65] Carlos Prada-Rojas, Frederic Riss, Xavier Raynaud, Serge De Paoli, and Miguel Santana. Observation Tools for Debugging and Performance Analysis of Embedded Linux Applications. In *Conference on System Software, SoC and Silicon Debug 2009*, 2009.

[66] Thread state and the global interpreter lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[67] Sheldon Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, 2009.

[68] Nick Russell, Arthur H M Ter Hofstede, David Edmond, and Wil M P Van Der Aalst. Workflow data patterns : Identification, representation and tool support. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3716 LNCS :353–368, 2005.

[69] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems : An overview. *European Journal of Operational Research*, 134 :664–676, 2001.

[70] Idafen Santana-perez, Rafael Ferreira, Mats Rynge, Ewa Deelman, and S P. Leveraging Semantics to Improve Reproducibility in Scientific Workflows. *The reproducibility at XSEDE workshop*, 2014.

[71] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2) :287–311, may 2006.

[72] F. Song, Felix Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 63–72 vol.1, 2004.

[73] Aaron Spear, Markus Levy, and Mathieu Desnoyers. Using Tracing to Solve the Multicore Problem. *Computer*, pages 60–64, 2012.

[74] Phil Spector. *Data Manipulation with R*. 2008.

[75] Luka Stanisic, Arnaud Legrand, and Vincent Danjean. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *ACM SIGOPS Operating Systems Review*, 49 :61–70, 2015.

[76] B De Oliveira Stein. Pajé trace file format. 2003.

[77] Edward a. Stohr and J. Leon Zhao. Workflow Automation : Overview and Research Issues. *Information Systems Frontiers*, 3 :281–296, 2001.

[78] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[79] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and M Seltzer. Benchmarking file system benchmarking : It* is* rocket science. *HotOS XIII*, (HotOS XIII) :1–5, 2011.

[80] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2) :1–56, 2008.

[81] Jessica M. Utts and Robert F. Heckard. *Mind on Statistics*. 2007.

[82] W. M. P. van der Aalst, K. M. van Hee, a. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets : classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3) :333–363, aug 2010.

[83] Vistrails.
https://www.vistrails.org.

[84] Huy T. Vo, Daniel K. Osmari, Brian Summa, João L D Comba, Valerio Pascucci, and Cláudio T. Silva. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29(3) :1073–1082, 2010.

[85] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation : A Survey and Critical Review. In *Memory Management*. 1995.

[86] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite : designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1) :W557–W561, 2013.

[87] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112, jun 2012.

[88] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems*, 30(1) :1–28, feb 2012.

Résumé La validation des systèmes est un des aspects critiques dans les phases de développement. Cette validation est d'autant plus importante pour les systèmes embarqués, dont le fonctionnement doit être autonome, mais aussi contraint par des limitations physiques et techniques. Avec la complexification des systèmes embarqués ces dernières années, l'application de méthodes de validation durant le développement devient trop coûteux, et la mise en place de mécanismes de vérification post-conception est nécessaire. L'utilisation de traces d'exécution, permettant de capturer le comportement du système lors de son exécution, se révèle efficace pour la compréhension et la validation des systèmes observés. Cependant, les outils d'exploitation de traces actuels se confrontent à deux défis majeurs, à savoir, la gestion de traces pouvant atteindre des tailles considérables, et l'extraction de mesures pertinentes à partir des informations bas-niveau contenues dans ces traces.

Dans cette thèse, faite dans le cadre du projet FUI SoC-TRACE, nous présentons trois contributions. La première concerne la définition d'un format générique pour la représentation des traces d'exécution, enrichi en sémantique. La seconde concerne une infrastructure d'analyse utilisant des mécanismes de workflow permettant l'analyse générique et automatique de traces d'exécution. Cette infrastructure répond au problème de gestion des traces de tailles considérables *via* des mécanismes de streaming, permet la création d'analyses modulaires et configurables, ainsi qu'un enchaînement automatique des traitements. Notre troisième contribution propose une méthode générique pour l'analyse de performances de systèmes Linux. Cette contribution propose à la fois la méthode et les outils de collecte de traces, mais aussi le workflow permettant d'obtenir des profils unifiés pour les traces capturées.

La validation de nos propositions ont été faites d'une part sur des traces issues de cas d'usages proposés par STMicroelectronics, partenaire du projet, et d'autre part sur des traces issues de programmes de benchmarks. L'utilisation d'un format enrichi en sémantique a permis de mettre en évidence des anomalies d'exécutions, et ce de manière semi-automatique. L'utilisation de mécanismes de streaming au sein de notre infrastructure nous a permis de traiter des traces de plusieurs centaines de gigaoctets. Enfin, notre méthode d'analyse générique nous a permis mettre en évidence, de manière automatique et sans connaissances *a priori* des programmes, le fonctionnement interne de ces différents benchmarks. La généricité de nos solutions a permis d'observer le comportement de programmes similaires sur des plates-formes et des architectures différentes, et d'en montrer leur impact sur les exécutions.

Abstract Validation is a critical aspect of system development. This is especially true in the context of embedded systems which come with various technical and physical constraints. As embedded systems have enormously grown in complexity in recent years, formal method validation is too complex or too costly to apply. Validation must therefore use post-conception methods, the major method being trace analysis. Indeed, execution traces capture many details about systems' execution and provide a means for advanced system analysis. However, trace analysis faces two major challenges. On one hand, trace analysis has to deal and support huge execution traces. On the other hand, analysis methods should be able to retrieve relevant metrics from the low-level information traces contain.

This thesis has been done as part of the SoC-TRACE projet and presents three contributions. Our first contribution is a definition of a generic execution trace format that expresses semantics. Our second contribution is a workflow-based infrastructure for generic and automatic trace analysis. This infrastructure addresses the problem of huge traces management using streaming mechanisms. It allows modular and configurable analyses, as well as automatic analyses execution. Our third contribution concerns generic performance analysis for Linux systems. This contribution defines the means for trace recording, as well as an analysis workflow producing unified performance profiles.

We validate our contributions with use cases given by STMicroelectronics and traces obtained from benchmark executions. Our trace format with semantics allowed us to automatically bring out execution problems. Using streaming mechanisms, we have been able to analyze traces that can reach several hundreds of gigabytes. Our generic analysis method for systems let us to automatically highlight, without any prior knowledge, internal behavior of benchmark programs. Our generic solutions point out a similar execution behavior of benchmarks on different machines and architectures, and showed their impact on the execution.