UNIVERSITÉ DE MONTRÉAL

DÉBOGUAGE À LARGE ÉCHELLE SUR DES SYSTÈMES HÉTÉROGÈNES PARALLÈLES

DIDIER NADEAU
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUILLET 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DÉBOGUAGE À LARGE ÉCHELLE SUR DES SYSTÈMES HÉTÉROGÈNES PARALLÈLES

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

M. OZELL Benoît, Ph. D, président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BOIS Guy, Ph. D., membre

REMERCIEMENTS

Je désire tout d'abord remercier mon directeur de recherche, Michel Dagenais, pour m'avoir donné l'opportunité de participer à ce projet de recherche. Ses conseils, son soutien ainsi que son expertise ont été grandement utiles pour me guider dans ce projet de recherche.

De plus, je tiens à souligner le soutien financier offert par Ericsson, EfficiOS, Prompt ainsi que le Conseil de recherche en sciences naturelles et en génie du Canada pour ce projet de recherche.

Par ailleurs, je remercie Simon Marchi, Marc Khouzam, Marc Dumais et plusieurs de leurs collègues chez Ericsson pour m'avoir grandement aidé grâce à leur expertise dans le domaine du déboguage.

Mes collègues au laboratoire Dorsal ont aussi contribué à mon projet de recherche à travers leur soutien, leurs connaissances variées ainsi que leur bonne humeur. Je tiens à souligner l'aide fournie par Françis Giraldeau, Suchakra Sharma, Naser Ezzati et Geneviève Bastien, qui m'a été très utile pour avancer mon travail.

Finalement, j'aimerais aussi remercier ma famille, ma compagne ainsi que mes amis pour m'avoir supporté tout au long de ma maîtrise.

RÉSUMÉ

Le traitement parallèle est devenu une nécessité afin de bénéficier du plein potentiel des processeurs centraux modernes. En effet, l'ajout de coeurs de calcul est un des facteurs ayant eu le plus d'impact sur le gain de performance des processeurs dans les dernières années. Pour cette raison, l'usage des processeurs multi-coeurs est de plus en plus répandu en industrie ainsi que chez les particuliers, et le nombre de coeurs continue à augmenter.

Cependant, le développement de programmes adaptés aux processeurs parallèles apporte de nouveaux défis. En effet, le paradigme de développement séquentiel ne suffit plus, et le programmeur doit prendre en compte le parallélisme. Il doit s'assurer que les interactions entre les fils d'exécution ne créent pas de problèmes. Cela est difficile, notamment lorsque le programme utilise un grand nombre de fils d'exécution, et des problèmes dus au parallélisme peuvent survenir. Cela pose un défi important en industrie, car ces problèmes compliquent le développement logiciel, et peuvent augmenter les coûts de développement.

Il est donc nécessaire d'avoir des outils adaptés au développement de programmes parallèles pour permettre aux programmeurs d'identifier et de trouver rapidement ces problèmes. Un outil vital pour le processus de développement logiciel est le débogueur. Cet outil permet à l'utilisateur de contrôler le fonctionnement d'un programme, de l'arrêter à certains endroits et d'utiliser plusieurs autres techniques pour trouver l'origine de problèmes. Or, la majorité des débogueurs et des techniques de déboguage utilisés existent depuis plusieurs années, et ont été développés alors que les processeurs parallèles étaient peu répandus.

Plusieurs problèmes liés au parallélisme découlent d'une mauvaise synchronisation entre les fils d'exécution. La course de donnée en est un example. Elle survient lorsque plusieurs fils d'exécution modifient une variable en mémoire sans synchronisation. La valeur finale de la variable dépend alors de l'ordre d'exécution des fils et peut varier entre différentes exécutions du programme. Ce type de problème n'est donc pas déterministe, et n'arrive pas nécessairement même si on ré-exécute un programme avec les mêmes valeurs de départ. Or, la plupart des techniques de déboguage sont intrusives. En effet, stopper un programme et examiner ses variables a un effet important sur la synchronisation. Il arrive donc qu'un problème disparaisse ou change lorsqu'un programme est débogué en raison de l'impact du débogueur. Il est évident que des techniques de déboguage minimalement intrusives sont nécessaires afin d'aider les développeurs à déceler ce type d'erreur.

Un autre défi présent avec le déboguage de programmes parallèles est l'interface utilisateur. En effet, les programmes parallèles contiennent un très grand nombre de fils d'exécution, et peuvent générer un grand débit d'évènements qui doivent être traités par le développeur. Par example, il

peut être nécessaire d'inspecter l'état de plusieurs dizaines de fils d'exécution lors d'un arrêt du programme pour bien comprendre le contexte. Cela est une tâche pénible, et peut mener le développeur à manquer certains problèmes en raison d'une surcharge d'information. Il peut arriver qu'il soit difficile de trouver l'information pertinente à travers toutes les informations affichées. Il faut donc que l'utilisateur ait accès à une interface qui lui permette de facilement visualiser et contrôler l'état d'un programme.

Ce projet de recherche a abordé ces deux problèmatiques du déboguage de systèmes parallèles. Le travail de recherche a été effectué sur des ordinateurs conventionnels parallèles possédant un très grand nombre de coeurs de calculs, ainsi que sur des systèmes hétérogènes parallèles utilisant une carte graphique pour le calcul générique. Les travaux ont été effectués en modifiant deux programmes de déboguage populaires, soit le débogueur GDB ainsi que l'environnement de développement Eclipse CDT.

Le débogueur GDB a été amélioré afin de permettre de tracer dynamiquement un programme parallèle avec un impact minimal. Cela permet de capturer une série d'évènements survenus lors de l'exécution du programme, tout en affectant minimalement son exécution, de manière à trouver les problèmes sensibles à la synchronisation. Cette technique de traçage dynamique utilise GDB pour insérer le point de trace, et LTTng pour collecter les informations recueillies. De plus, des tests ont été faits avec GDB pour évaluer l'impact des points d'arrêt conditionnels, et une technique pour diminuer cet impact a été proposée.

Finalement, le travail effectué avec Eclipse CDT a permis de proposer des techniques pour représenter efficacement un programme parallèle. Une vue regroupant automatiquement des fils d'exécution en utilisant leur pile d'appel a été proposée. Une autre une vue, dédiée aux cartes graphiques, groupe les fils d'exécution fonctionnant sur celle-ci grâce à leur position dans la grille de donnée. De plus, des techniques pour permettre à l'utilisateur de filtrer efficacement l'information ont été proposées, afin de permettre à celui-ci de limiter la quantité de données qu'il a à traiter.

ABSTRACT

A significant proportion of performance gains in processors in recent years has come from parallel processing. Indeed, adding cores in a processor has been heavily used by most manufacturers to gain performance improvements. For this reason, multi-core and many-core processors are increasingly common in personal computers and industrial systems.

However, designing software for parallel processors is not an easy task. Indeed, the sequential programming paradigm traditionally used is not sufficient anymore, as multiple tasks can be processed simultaneously. The software designer must take interactions between threads into account, and write code accordingly. This is a complicated task, especially when a program uses a large number of threads. This can lead to multiple issues, thus increasing development time and cost.

Therefore, using development tools suitable for parallel development is essential to allow developpers to quickly identify and locate these issues. One of the most important tools is the debugger, a tool to control a program execution and inspect its state. Unfortunately, most debugging tools have been in use for a long time, and were designed to debug single-threaded applications.

Many problems can be created by a mistake in thread synchronization. One example is a data race, that occurs when multiple threads try to access a shared variable without concurrency control. In this case, its final value depends on the order in which each thread accesses it, and the result may change between different executions as the threads scheduling varies. Inspecting this type of problem with a debugger can be problematic, as a debugger can be very intrusive and disturb the execution of the program. Therefore, a multi-threading issue may be masked or modified during debugging, greatly diminishing the usefulness of the debugger. Thus, adapted debugging techniques are needed for parallel software.

Another challenge is the large amount of data to be handled by the user. Indeed, as parallel software may contain a great number of threads, there is a lot of information to display, and a large number of events can be generated. This can confuse the developer by overloading him with information. Therefore, appropriate techniques to visualize and control multi-threaded software are needed.

These two challenges have been addressed during this research project. Two different platforms have been used for tests. The first one is a server with four multi-core processors that has a total of 64 physical cores. The other one is a general purpose computer with a multi-core processor and a graphic card used for calculations.

The GDB debugger has been enhanced to allow dynamic scalable tracing in order to trace parallel programs with minimal overhead. This enhancement has been achieved by modyfing GDB to

address limitations for parallel tracing, and using LTTng for trace collection. LTTng has also been enhanced to allow dynamic event registration. Finally, the impact of conditional breakpoints has been measured, and a technique to minimize this impact has been proposed.

Furthermore, Eclipse CDT has been used to propose techniques for the user to interface with the debuggers. A view that automatically groups threads using their call stack has been created, and provides an efficient way for the user to visualize a large number of threads. Another view shows the threads currently executing on the graphic cards, and groups them using their position in the data grid sent to the GPU. Finally, a technique to filter threads executing on the CPU and the GPU has been proposed. This allows the developer to remove unneeded information from the debugging context and concentrate on relevant data.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX xi

LISTE DES FIGURES xii

LISTE DES SIGLES ET ABRÉVIATIONS xiii

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.1.1 Système hétérogène parallèle 1
1.1.2 Débogage 2
1.1.3 Instrumentation 2
1.1.4 Traçage 3
1.2 Éléments de la problématique 4
1.3 Objectifs de recherche 5
1.4 Environnement de travail 5
1.5 Plan du mémoire 6

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE 7
2.1 Architecture des processeurs 7
2.1.1 Processeurs à un seul cœur 7
2.1.2 Processeurs parallèles 8
2.1.3 Architectures hétérogènes 8
2.1.4 Processeurs graphiques 12
2.2 Débogage 16
2.2.1 Débogage intrusif 18
2.2.2 Débogage réversible 18
2.2.3 Débogage asynchrone 20

2.2.4 Débogueurs disponibles 20
2.2.5 Interface graphique pour le débogage 21
2.3 Instrumentation 24
2.3.1 Instrumentation statique 24
2.3.2 Instrumentation dynamique 25
2.4 Traçage 27
2.4.1 Mise à l'échelle 27
2.4.2 Traceurs du noyau d'exploitation 28
2.4.3 Traceurs pour les programmes utilisateurs 30
2.5 Analyse de données de traçage 32
2.5.1 Analyses automatisées 32
2.5.2 Interfaces graphiques 34
2.6 Conclusion de la revue de littérature 36

CHAPITRE 3 ARTICLE 1 : EFFICIENT LARGE SCALE HETEROGENEOUS DEBUGGING USING DYNAMIC TRACING 38
3.1 Abstract 38
3.2 Introduction 38
3.3 Related work 40
3.4 Motivation 45
3.5 Performance evaluation of current tools 46
3.6 Proposed contributions 51
3.6.1 Dynamic tracing 51
3.6.2 Visual debugging 55
3.7 Discussion 58
3.8 Conclusion 65
3.9 Acknowledgement 66

CHAPITRE 4 RÉSULTATS COMPLÉMENTAIRES 67
4.1 Multicore Visualizer pour les cartes graphiques 67
4.2 Points d'arrêt conditionnels 69

CHAPITRE 5 DISCUSSION GÉNÉRALE 72
5.1 Retour sur les résultats 72
5.1.1 GDB 72
5.1.2 Interfaces graphiques 73
5.2 Limitations 74

5.3 Présentation aux partenaires industriels 74

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS 76
6.1 Synthèse des travaux 76
6.2 Améliorations futures 77

LISTE DES RÉFÉRENCES 79

LISTE DES TABLEAUX

Tableau 1.1 Ordinateur de bureau 5
Tableau 1.2 Serveur « Manchot » 6
Tableau 4.1 Durée d'exécution totale moyenne d'un programme de test lorsque la condition d'un point d'arrêt est évaluée 1 000 000 fois, et marge d'erreur associée avec un intervalle de confiance de 95% 70

LISTE DES FIGURES

Figure 2.1 The overall Epiphany architecture. Tiré de (Olofsson et al., 2014). ©2011 IEEE. Reproduit avec permission. 10
Figure 2.2 1st Generation Cell components. Tiré de (Hofstee, 2005). ©2005 IEEE. Reproduit avec permission. 12
Figure 2.3 Fenêtre affichant les fils d'exécution et leur état dans Eclipse Neon CDT . . 22
Figure 2.4 Multicore Visualizer disponible avec Eclipse CDT 24
Figure 2.5 Control and trace data paths between LTTng components. ©The LTTng Project 2014–2017. Reproduit avec permission. 30
Figure 2.6 Analyse des appels systèmes avec LTTng analyses 33
Figure 2.7 Capture d'écran de la vue « Control Flow » de Trace Compass 35
Figure 2.8 Capture d'écran de la vue des ressources de Trace Compass 35
Figure 3.1 Total execution time for a test program as a function of the number of threads when a fast tracepoint is inserted 47
Figure 3.2 Representation of program state during normal execution 47
Figure 3.3 Representation of program state while tracing it with the default GDB fast tracing architecture 48
Figure 3.4 Timeline of threads state when a tracepoint has been inserted 49
Figure 3.5 Main Eclipse CDT view to shows threads in the Debug perspective 50
Figure 3.6 Proposed fast tracing architecture using LTTng 53
Figure 3.7 Standard fast tracepoint workflow 53
Figure 3.8 Proposed LTTng tracepoint workflow 55
Figure 3.9 Algorithm to build the tree structure for the call stack view. 56
Figure 3.10 Total execution time to compress a 100 MB file using pbzip2 60
Figure 3.11 Speed up of pbzip2 debugging using GDB standard and proposed tracing . 61
Figure 3.12 Trace data visualization with the proposed GDB fast tracing architecture . . 61
Figure 3.13 Stack Aggregation view implemented within Eclipse CDT to display active threads 62
Figure 3.14 Debug view and Stack Aggregation view when the filter has been applied. . 62
Figure 3.15 Eclipse CDT main debug view with GPU waves shown 64
Figure 3.16 GPU focus view to filter HSA waves 64
Figure 3.17 Eclipse CDT main debug view with GPU waves and an active GPU focus . 65
Figure 4.1 Vue du « Multicore Visualizer » montrant les vagues s'exécutant sur une carte graphique. 68

LISTE DES SIGLES ET ABRÉVIATIONS

GNU	GNU's Not Unix
GDB	GNU Debugger
CDT	C/C++ Development Tools
OpenMP	Open Multi-Processing
HSA	Heterogeneous System Architecture
OpenCL	Open Computing Language
MPI	Message Passing Interface

CHAPITRE 1 INTRODUCTION

Le développement d'un programme informatique est un processus complexe qui peut poser des difficultés. En effet, il est rare qu'un programme fonctionne parfaitement dès sa première version, surtout s'il s'agit d'un logiciel de plus grande envergure. Le débogage est alors employé pour découvrir et régler les problèmes qui font en sorte qu'un programme ne fonctionne pas comme prévu. Le débogage est donc un aspect majeur du développement logiciel, et il peut occuper une fraction importante du temps de développement. L'arrivée de systèmes complexes avec des processeurs ayant un grand nombre de coeurs de calcul, ainsi que des accélérateurs de calculs, amène des problèmes compliqués. En effet, ce type de système modifie le paradigme de programmation, et le développeur doit prendre la concurrence en compte. Cela n'est pas toujours aisé, et peut mener à différents problèmes dus au parallélisme. Il est donc nécessaire d'avoir des outils adaptés aux systèmes hétérogènes parallèles afin de bénéficier de leurs capacités et faciliter le développement.

1.1 Définitions et concepts de base

Cette section introduit et définit certains concepts utilisés dans ce mémoire.

1.1.1 Système hétérogène parallèle

Système parallèle

Un système parallèle est un système informatique qui contient plus d'une unité de calcul, de manière à ce que plusieurs processus puissent être exécutés simultanément. Un système parallèle peut être simplement composé de plusieurs ordinateurs ou processeurs mis en réseau. Il peut aussi s'agir d'un ordinateur ayant un processeur multi-coeur, c'est-à-dire un processeur avec plus d'un coeur de calcul. Des accélérateurs de calcul peuvent aussi être ajoutés à un ordinateur pour aider le processeur central. Finalement, un système parallèle peut être composé d'une combinaison des types de systèmes précédemment expliqués, par exemple il pourrait s'agir de plusieurs ordinateurs en réseau, chacun ayant un processeur multi-coeurs et plusieurs accélérateurs de calcul.

Système hétérogène

Un système est défini comme hétérogène s'il utilise plus qu'un seul type de processeur. Il peut s'agir d'un ordinateur avec un processeur central et une carte graphique utilisée pour le traitement générique. Puisque ces deux types de processeurs ont des architectures différentes, l'ordinateur est

donc un système hétérogène. Il faut noter qu'étant donné qu'un système hétérogène a nécessairement plus d'un processeur, un système hétérogène est intrinsèquement parallèle. Cependant, un système parallèle n'est pas obligatoirement hétérogène.

1.1.2 Débogage

Le débogage est le processus d'identification, de localisation et de correction des erreurs dans le code. L'outil principal utilisé durant le débogage est le débogueur, qui sert à contrôler l'exécution d'un programme. Étant donné qu'il y a plusieurs types d'erreurs, tels que les divisions par zéro ou un dépassement de pile, il faut qu'un débogueur soit versatile. D'autres outils spécialisés peuvent aussi être utilisés durant le débogage, tels qu'un traceur.

Évènement

Un évènement durant une session de débogage est lorsque le débogueur est notifié que quelque chose a eu lieu dans le programme débogué. Ainsi, cela peut simplement être que le programme a frappé un point d'arrêt, ou a chargé une nouvelle librairie dynamique. Cela peut aussi indiquer qu'un problème est survenu, et que le programme est arrêté. Un tel problème peut être une erreur de segmentation, une division par zéro, etc.

Point d'arrêt

Une des commandes les plus importantes avec un débogueur est d'insérer un point d'arrêt. Un point d'arrêt stoppe un programme lorsque ce dernier arrive à l'emplacement du point. Plusieurs méthodes existent pour implémenter un point d'arrêt. Deux méthodes très communes sont d'utiliser un registre spécial du processeur et y mettre l'adresse de l'instruction qui doit déclencher l'arrêt, ou d'insérer une instruction qui déclenche une interruption. Dans ce cas, l'interruption est reçue par le système d'exploitation qui se charge de réveiller le débogueur.

1.1.3 Instrumentation

L'instrumentation d'un programme consiste à modifier ce programme afin d'insérer des directives pour l'étudier et mesurer son fonctionnement. Une méthode élémentaire d'instrumentation consiste à imprimer des données à l'écran. Cela peut permettre de suivre l'évolution de la valeur de variables, et donc d'évaluer le fonctionnement du programme. Instrumenter un programme peut aussi être d'insérer des points d'attache dans le programme qui peuvent être activés ou non. Ainsi, un outil peut s'attacher à un de ces points afin d'être appelé à chaque fois que le programme arrive au point d'attache.

1.1.4 Traçage

Le traçage consiste à enregistrer une série d'information durant l'exécution d'un programme. L'enregistrement de données est effectué lorsqu'un évènement qui a été instrumenté avec un point de trace survient. Ces évènements peuvent venir d'un programme fonctionnant en mode utilisateur ou du système d'exploitation. Une trace venant d'un programme exécutant en mode utilisateur est appelée une trace utilisateur, ou usager, tandis qu'une trace venant du système d'exploitation est appelée une trace noyau.

Évènement

Un évènement correspond à un instant dans le système où le traceur est appelé et de l'information est enregistrée. Un évènement est constitué d'une date, de l'emplacement d'où il a été lancé ainsi que d'une série d'arguments de longueur variable. Cette série d'arguments peut être nulle, et dépend du type de l'évènement.

Point de trace

Un point de trace est un endroit dans le programme où un évènement est capturé. Il s'agit d'un point d'instrumentation qui appelle le traceur afin d'enregistrer un évènement.

1.2 Éléments de la problématique

En raison de la faible amélioration de performance des processeurs mono-coeur dans les dernières années, les processeurs multi-coeurs sont de plus en plus répandus. De plus, le nombre de coeurs de ces processeurs augmente. Or, ce type d'architecture nécessite un paradigme de programmation différent afin de profiter du parallélisme. Cela ouvre la porte à plusieurs types de problèmes absents avec un programme séquentiel.

Un usage typique d'un débogueur est de stopper un système à certains endroits, puis d'inspecter les valeurs et de contrôler manuellement l'exécution du programme. Cela fonctionne bien avec un programme séquentiel, puisque le comportement ne dépend normalement pas de la vitesse à laquelle le programme est exécuté. Or, plusieurs problèmes potentiels, sur les systèmes parallèles, émergents de problèmes de synchronisation, tels que les courses de données ou la famine. Un outil de débogage perturbant la synchronisation normale en stoppant des fils d'exécution risque donc de changer le comportement, et de masquer ou modifier des problèmes.

Une autre difficulté des systèmes parallèles et hétérogènes et la grande quantité d'information présente et le nombre d'évènements qui arrivent. En effet, les programmes développés pour ces plates-formes sont souvent complexes et utilisent un grand nombre de fils d'exécution. Cela leur permet de tirer profit de ces architectures. Cependant, un débogueur doit montrer l'ensemble des fils d'exécution de manière efficace. De plus, chaque fil peut engendrer des évènements lors du débogage. Le développeur, essayant de comprendre un problème, peut avoir de la difficulté à conceptualiser le fonctionnement, étant donné qu'il doit gérer tous ces fils et ces évènements.

Des outils modernes, spécialement adaptés aux systèmes hétérogènes parallèles, sont donc nécessaires. Ces outils doivent être capables de faire face à ces deux types de problèmes. Ainsi, il faut qu'ils puissent déboguer des problèmes tout en ayant un impact minimal sur sa synchronisation, de manière à ne pas affecter ces problèmes. De plus, une interface intuitive, permettant à un utilisateur de comprendre aisément l'état d'un programme, est nécessaire.

Le traçage est une solution potentielle au premier problème. En effet, cette technique vise à enregistrer des évènements afin de suivre l'évolution d'un programme, tout en minimisant l'impact sur son fonctionnement. Grâce aux évènements enregistrés, l'utilisateur peut essayer de trouver l'origine d'un problème. Cependant, le traçage nécessite un compromis entre la quantité de données enregistrées, et le ralentissement du système. De plus, tous les mécanismes de traçage ne sont pas aussi efficaces. Il est important de s'assurer que la technique choisie ait un impact satisfaisant les besoins.

Finalement, utiliser une interface graphique pour déboguer peut permettre d'afficher efficacement les informations à l'utilisateur. Cette interface doit être optimisée afin de montrer les informations

pertinentes de manière concise et utile. Étant donné que la plupart des interfaces ont été développées pour déboguer des programmes séquentiels, elles n'ont pas nécessairement des méthodes efficaces pour gérer les programmes parallèles.

1.3 Objectifs de recherche

Nous pouvons maintenant définir les objectifs de recherche :

1. Identifier un système hétérogène parallèle pertinent à utiliser pour les tests.
2. Étudier la performance d'un programme parallèle contrôlé par un débogueur, et identifier les facteurs limitant celle-ci.
3. Évaluer les capacités des interfaces graphiques pour permettre à l'utilisateur de déboguer facilement sur des systèmes hétérogènes parallèles.
4. Proposer des solutions pour résoudre des problèmes et limitations identifiés avec les débogueurs et les interfaces graphiques.
5. Intégrer ces solutions aux logiciels libres GDB et Eclipse CDT et tester leurs performances.
6. Présenter les avancés aux partenaires industriels travaillant dans le domaine.

1.4 Environnement de travail

Deux ordinateurs différents ont été utilisés pour les tests. Le premier est un ordinateur de bureau avec un processeur multi-coeur et une carte graphique sur lequel est installé une version d'Ubuntu 14.04 modifié par AMD. Cet ordinateur, décrit au Tableau 1.2, a été utilisé pour le travail effectué sur les cartes graphique avec le « Radeon Open Compute », les tests avec Eclipse CDT ainsi que le développement logiciel. Le deuxième ordinateur est un serveur doté de 4 processeurs multi-coeurs sur lequel est installé Fedora 24 et qui est décrit au Tableau 1.1. Il fut utilisé pour les tests sur les points de trace rapides et les points d'arrêt conditionnels.

Tableau 1.1 Ordinateur de bureau

Élément	Quantité	Description
Système d'exploitation	ND	Ubuntu 14.04.5
Processeur	1	Intel Core i7-4790 à 3.90 GHz
Carte Graphique	1	AMD Radeon R9 Nano
Mémoire vive	4	Kingston DIMM DDR3 1600 MHz de 8 GB

Tableau 1.2 Serveur « Manchot »

Élément	Quantité	Description
Système d'exploitation	ND	Fedora 24
Processeur	4	E7-8867 v3 à 2.50 GHz
Carte Graphique	0	ND
Mémoire vive	4	TBD

1.5 Plan du mémoire

La première partie de ce mémoire, la revue de littérature au chapitre 2, présente l'état actuel du domaine. Différentes architectures parallèles ainsi que des techniques de débogage et de traçage y sont présentées. Le chapitre 3 est constitué de l'article de journal écrit durant les travaux de maîtrise. Cet article a été soumis au « Journal of Systems and Software » publié par Elsevier. Ce chapitre explique des contributions proposées pour déboguer en traçant dynamiquement un programme, et en utilisant des interfaces graphiques optimisées. Le chapitre 4 présente d'autres résultats obtenus durant la maîtrise. Finalement, le chapitre 5 discute des résultats, et une courte synthèse est faite au chapitre 6.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

2.1 Architecture des processeurs

2.1.1 Processeurs à un seul cœur

La performance des ordinateurs électroniques multi-usage a constamment évolué depuis la création du premier prototype il y a près de 70 ans (Hennessy and Patterson, 2011). L'amélioration en performance a été très importante et a été affectée par de nombreuses avancées technologiques telles que le développement des circuits intégrés. Il y a eu un gain moyen de 25% de performance à chaque année durant les 25 premières années de l'informatique électronique. À partir de la fin des années 70, la performance a augmenté d'approximativement 35% annuellement en profitant de l'amélioration des circuits intégrés. Le gain en performance moyen atteint 52% lorsque l'on prend aussi en compte les améliorations architecturales dont ont profité les processeurs. Cependant, le début des années 2000 a mis fin à cette rapide évolution, et le gain moyen entre 2003 et 2010 était de seulement 22% (Hennessy and Patterson, 2011).

$$P \propto \frac{1}{2} C V^2 f \tag{2.1}$$

Deux facteurs importants, qui expliquent ces gains de performance des processeurs sur circuits intégrés, sont l'augmentation du nombre de transistors et l'augmentation de la fréquence de fonctionnement. Or, ces changements impactent la consommation énergétique du processeur. L'équation 2.1, tirée de (Hennessy and Patterson, 2011), indique que la relation entre la puissance consommée par un transistor et la fréquence de l'horloge du processeur est proportionnelle. De plus, la précision de gravure s'est grandement améliorée. Ainsi, la taille minimale des éléments d'un circuit intégré est passée de près de 10 microns en 1971 jusqu'à 32 nanomètres en 2011. Cela fait en sorte que la densité de transistor par millimètre carré est passée de 2851 pour le Intel 80286 en 1982 à 4 875 000 pour le Intel i7 en 2011 (Hennessy and Patterson, 2011). Les augmentations de densité de transistor et de fréquence de fonctionnement des transistors ont conduit à une hausse importante de la génération de chaleur, causant des problèmes de température. Ces problèmes de température ont imposé une limite à la performance des processeurs mono-cœur au début des années 2000. Intel a notamment abandonné ses projets de processeurs mono-cœur haute performance en 2004 (Hennessy and Patterson, 2011).

Face à cette limitation, les fabricants de processeurs ont innové afin de trouver d'autres moyens que l'augmentation de la fréquence afin leur performance. Les fabricants de processeurs se sont

tournés vers le calcul en parallèle afin de continuer à améliorer la vitesse de calcul (Hennessy and Patterson, 2011). Deux types de calculs en parallèle ont été exploités : le parallélisme de données et le parallélisme de tâche. Pour profiter de ces méthodes de calcul, de nouvelles instructions ainsi que de nouvelles architectures informatiques ont été développées.

2.1.2 Processeurs parallèles

Le parallélisme de tâche consiste à exécuter simultanément plusieurs tâches différentes, tel que le font les processeurs génériques multi-cœurs (Hennessy and Patterson, 2011). Dans ce cas, chaque tâche possède des valeurs privées et un pointeur d'instruction, puisqu'elles exécutent des instructions différentes. Ces tâches peuvent être des fils d'un même processus et partager le code exécutable ainsi que des variables en mémoire. Il peut aussi s'agir de deux processus distincts qui exécutent du code différent. Un cas classique de processus distincts est un ordinateur de bureau qui exécute deux programmes tels qu'un navigateur Internet et un éditeur de texte simultanément sur des cœurs de calcul différents.

Le parallélisme de données est un type de calcul en parallèle qui exploite la possibilité d'appliquer la même opération sur plusieurs emplacements mémoires simultanément (Hennessy and Patterson, 2011). Un grand nombre de processeurs génériques modernes possèdent des instructions qui font usage de ce type de parallélisme, appelées instructions *SIMD*. Ainsi, Intel et AMD manufacturent des processeurs avec des opérations du jeu d'instruction *SSE* qui permet de faire plusieurs instructions simultanément sur plusieurs variables contiguës. Ce type d'instruction permet d'augmenter le débit de calcul en faisant plusieurs opérations pour la même fréquence tout en augmentant peu la complexité des processeurs (Hennessy and Patterson, 2011). Le parallélisme de données peut être combiné avec le parallélisme de tâche, par exemple dans le cas d'un processeur générique multi-cœurs qui supporte le jeu d'instruction *SSE*.

2.1.3 Architectures hétérogènes

Avec la fin de l'augmentation de fréquence des processeurs au début des années 2000, d'autres avenues ont été explorées pour augmenter la performance des ordinateurs. Une possibilité est de bâtir un système distribué en combinant un grand nombre de processeurs conventionnels identiques. Cependant, la performance de ces systèmes homogènes n'est pas capable de suivre la performance de systèmes composés de plusieurs processeurs différents (Shan, 2006). En effet, la combinaison de processeurs conventionnels avec des accélérateurs spécialisés, des processeurs DSP, des processeurs vectoriels ou des FPGA permet d'atteindre de meilleures performances. Ainsi, le super ordinateur Cray XD1 est l'un des premiers systèmes à incorporer des FPGA disponibles aux usagers et le super ordinateur Cray X1E fait usage d'une combinaison de processeurs conventionnels

et vectoriels afin d'atteindre de meilleures performances.

Combiner des processeurs différents dans un même système permet de profiter des avantages de chacun. En effet, différents types de processeurs offrent des performances différentes en fonctions des tâches effectuées. Cela permet d'affecter une tâche au processeur le plus efficace pour celle-ci et ainsi d'améliorer la performance globale (Shan, 2006). Néanmoins, la complexité de développement sur des architectures hétérogènes est un facteur qui freine leur utilisation. En effet, distribuer un travail sur plusieurs processeurs complexifie le travail et l'adaptation à des architectures différentes ajoute un coût supplémentaire. De plus, les transferts mémoires entre les différents processeurs induisent un retard non négligeable qui font en sorte qu'il n'est pas nécessairement intéressant d'envoyer une tâche à un accélérateur approprié en dessous d'une certaine taille.

Le Xeon Phi est un accélérateur de calcul développé par Intel pour augmenter les performances de calcul scientifique. Il s'agit d'un processeur avec plus de 60 cœurs physiques de type x86-64 avec des instructions de calcul vectoriel ajouté (Chrysos, 2014). De plus, chaque cœur physique contient quatre cœurs virtuels, ce qui donne un total de plus de 240 cœurs. Les cœurs possèdent chacun une cache L2 qui leur est propre, et ils sont reliés ensemble par un bus circulaire. Étant donné que le Xeon Phi est conçu pour être connecté à un processeur conventionnel, le système final est hétérogène.

La compagnie Adapteva a développé l'Epiphany, un accélérateur de calcul multicœur qui peut être utilisé dans un système hétérogène. Ce processeur est un processeur homogène puisque ses cœurs sont identiques, mais il est conçu pour jouer le rôle d'accélérateur de calcul en combinaison avec un processeur conventionnel. On peut voir une image représentant la grille de nœud et la structure d'un nœud du processeur Epiphany à la Figure 2.1. Les cœurs de ce processeur sont disposés en deux dimensions et connectés par un réseau en grille. Cela permet de minimiser la longueur du réseau et donc de diminuer la longueur de propagation des signaux et l'énergie consommée (Gwennap, 2011). Cette puce est optimisée pour effectuer des tâches de traitement de signaux telles que la reconnaissance visuelle ou vocale avec une haute efficacité énergétique. Des choix d'architectures tels que l'absence de cache et l'utilisation d'une unité de calcul point flottant supportant un nombre limité d'opérations permettent de simplifier l'architecture et de réduire la consommation d'énergie. Cela permet au coprocesseur Epiphany d'atteindre une puissance de calcul par Watt plus de 5 fois supérieure au processeur ARM Cortex-A9. Cependant, les choix de design d'Adapteva font en sorte que le développeur logiciel doit explicitement gérer la mémoire SRAM et les transferts DMA. Il s'agit d'un compromis similaire à celui fait pour le processeur CELL (Hofstee, 2005) : optimiser l'efficacité de calcul au prix d'une plus grande complexité de développement.

S'il existe des systèmes hétérogènes constitués de plusieurs processeurs distincts, il existe aussi des systèmes hétérogènes sur une même puce. Ceux-ci peuvent prendre la forme de plusieurs cœurs dif-

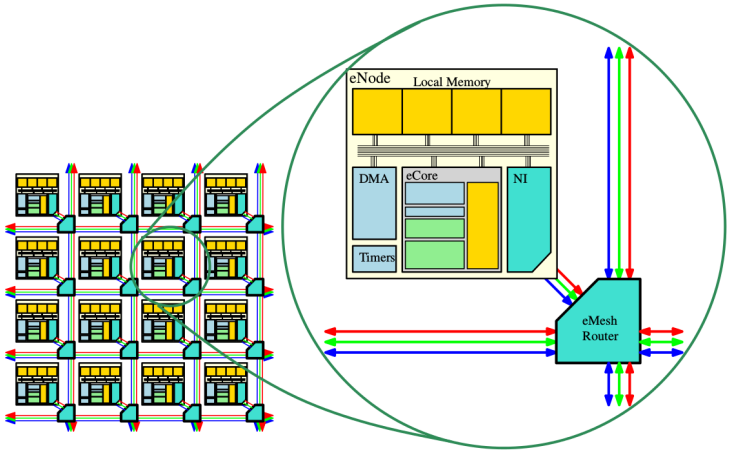

Figure 2.1 The overall Epiphany architecture. Tiré de (Olofsson et al., 2014). ©2011 IEEE. Reproduit avec permission.

férents dans un même processeur ou de plusieurs processeurs distincts gravés sur la même puce, ce qui diminue les problèmes de transfert mémoire. Un processeur hétérogène peut être constitué de plusieurs cœurs différents qui partagent le même jeu d'instructions. Ce type d'architecture permet aussi de maximiser la performance et l'efficacité énergétique d'un processeur puisque les cœurs ayant différentes puissances de calcul peuvent être affectés à des tâches appropriées (Kumar et al., 2004). Les cœurs diffèrent sur des caractéristiques telles que la taille de la cache, l'exécution des instructions en ordre ou en désordre, etc. Cette variation impacte l'espace sur la puce utilisé par chaque cœur et permet de maximiser l'utilisation de l'espace pour améliorer la performance. Kumar et al. présentent aussi un céduleur dynamique permettant d'optimiser l'assignation des tâches au cœur approprié pour optimiser la performance. En comparant un modèle d'un processeur hétérogène versus un processeur multicœur homogène de même taille, ils ont obtenu des gains en performance atteignant jusqu'à 69% sur un simulateur avec le processeur hétérogène. De plus, l'utilisation du même jeu d'instruction pour tous les cœurs permet de faciliter le développement puisque ces cœurs peuvent être considérés comme interchangeables par le programmeur et l'optimisation des tâches est effectuée dynamiquement par le céduleur. Cependant, un processeur hétérogène avec un seul jeu d'instruction ne profite pas de l'ensemble des bénéfices de l'hétérogénéité puisque différents types de tâches peuvent être mieux adaptées à différents jeux d'instructions.

Ce type de processeur hétérogène est répandu dans le domaine des téléphones cellulaires intelligents. En effet, la compagnie ARM a développé une architecture appelée *big.LITTLE* qui combine des cœurs ARM Cortex partageant un même jeu d'instruction, mais ayant des caractéristiques différentes (ARM, 2013). Samsung a utilisé l'architecture *big.LITTLE* avec *Exynos 5 Octa* qui combine 4 cœurs ARM haute performance avec 4 cœurs faible puissance (Chung et al., 2012). Le mode

d'opération le plus simple active soit le groupe de cœurs haute performance, ou le groupe faible puissance, en fonction de la charge de travail demandée. Le mode *HMP*, pour « Heterogeneous Multi-Processing », gère les 8 cœurs séparément en les activant, désactivant ou leur donnant un travail individuellement. Ce mode permet de maximiser la performance et l'économie d'énergie grâce à l'économie d'énergie offerte. Comme démontré par (Kumar et al., 2004), cela permet d'optimiser l'efficacité énergétique puisqu'il n'est pas nécessaire d'activer un cœur de haute performance pour une tâche légère, ce qui est très important dans le domaine des téléphones cellulaires où la capacité des batteries est limitée.

Il y a aussi eu des tentatives pour concevoir des processeurs hétérogènes constitués de cœurs ayant des jeux d'instructions distincts. Ainsi, une architecture composée de cœurs avec des jeux d'instructions Thumb, x86-64 et Alpha a été proposée par (Venkat and Tullsen, 2014). Les auteurs proposent un processeur supportant la migration automatique de tâche entre les cœurs. Cela simplifie le développement logiciel, mais amène des coûts lors des migrations puisque les tâches doivent être converties. Néanmoins, leur modèle prévoit un gain de performance jusqu'à 20,8% et une amélioration de 23% de l'efficacité énergétique comparativement à une architecture équivalente avec un seul jeu d'instructions. D'autres processeurs hétérogènes à plusieurs jeux d'instruction ne supportent pas la migration automatique des tâches, ce qui simplifie l'architecture, mais augmente la complexité de développement.

Le processeur CELL a été développé par un consortium formé de Toshiba, Sony et IBM et produit en grande quantité. Il s'agit du processeur utilisé par la console de jeu vidéo PlayStation 3. Son architecture hétérogène est constituée d'un cœur haute performance PowerPC couplé avec 8 SPE, des cœurs simples supportant des opérations SIMD avec un jeu d'instructions différent du cœur PowerPC. On peut voir un schéma de l'architecture du processeur CELL à la Figure 2.2. L'architecture mémoire est contrôlée par l'application plutôt que par le matériel afin de permettre une très bonne performance, mais nécessite de la programmation complexe de bas niveau (Crawford et al., 2008). Chaque SPE possède sa propre mémoire et ne peut accéder à la mémoire centrale. Les transferts entre les SPE et la mémoire centrale doivent être gérés explicitement par DMA car il n'y a pas de migration automatique des tâches. Le processeur CELL est populaire pour le calcul haute performance en raison de sa puissance de calcul. Ainsi, le super ordinateur RoadRunner qui était constitué de nœuds avec 4 processeurs de type CELL et 2 processeurs Opteron à deux cœurs obtenait une performance de 408 GigaFlops pour les processeurs CELL contre 15,3 GigaFlops pour les processeurs Opteron. Néanmoins, cette bonne performance est limitée par la difficulté de développement et le processeur CELL n'a jamais percé le marché des ordinateurs de bureau.

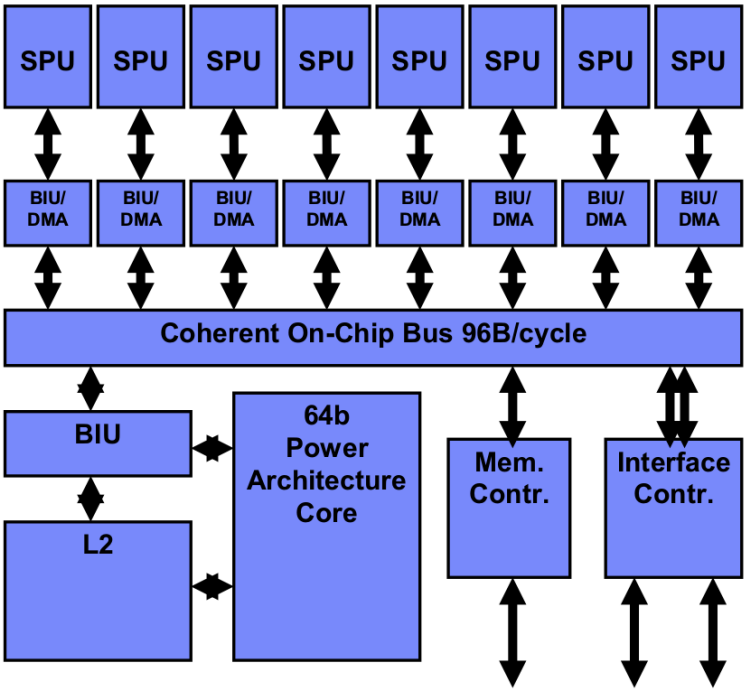

Figure 2.2 1st Generation Cell components. Tiré de (Hofstee, 2005). ©2005 IEEE. Reproduit avec permission.

2.1.4 Processeurs graphiques

Les processeurs graphiques sont un type de circuit spécialement développé pour exploiter le parallélisme de données. Les instructions sur un processeur graphique effectuent la même opération sur un grand nombre de données simultanément, de manière similaire aux instructions *SIMD* (Hennessy and Patterson, 2011). Ces processeurs ont initialement été développés pour servir d'accélérateur de traitement graphique, puisque ce type de tâche nécessite un grand nombre d'opérations identiques sur des données différentes. Les processeurs graphiques ont alors été développés pour libérer le processeur central de ce travail. Ils font aussi usage du parallélisme de tâche à un niveau supérieur puisqu'ils peuvent exécuter plusieurs tâches qui font elles-mêmes usage du parallélisme de donnée. Un ordinateur possédant une carte graphique constitue un système hétérogène puisque l'unité centrale de traitement et la carte graphique ne partagent pas la même architecture ni le même jeu d'instruction.

Ce type de processeurs a normalement une mémoire cache plus petite que celle des processeurs génériques et n'essaie pas de retenir en cache beaucoup de données simultanément. Ils programment plusieurs tâches une après les autres et en exécute une tout en allant chercher les données pour la prochaine. Les processeurs graphiques masquent ainsi le délai associé à la recherche en mémoire

centrale (Nickolls and Dally, 2010). Comparativement à un processeur générique, moins d'espace est utilisé sur le circuit par la cache puisque celle-ci est plus petite. Ainsi, cet espace est utilisé pour avoir un plus grand nombre de cœurs de calcul et permet aux processeurs graphiques d'avoir plusieurs centaines ou milliers de cœurs. De plus, un seul pointeur d'instruction est utilisé pour plusieurs, ce qui permet de simplifier l'architecture et d'augmenter la densité de cœurs de calcul. Cependant, les branchements conditionnels sont beaucoup plus coûteux sur les processeurs graphiques étant donné que l'unique pointeur d'instruction doit gérer toutes les instructions pour son groupe de données. Par exemple, deux conditions imbriquées avec des branches de longueurs égales sont évaluées avec une efficacité de 12,5% (Hennessy and Patterson, 2011).

Calcul générique sur une carte graphique

Les processeurs graphiques ont initialement été développés pour le traitement visuel, tel que leur nom l'indique, ils ont depuis été adaptés pour différentes tâches. Ainsi, dès 2001 des cartes graphiques ont été utilisées pour effectuer du calcul générique. Les premiers cas d'utilisation étaient des preuves de concept plutôt que des applications réelles, puisque les processeurs graphiques ne pouvaient fournir que des résultats numériques à virgules fixes de 8 bits, alors que des nombres à virgule flottante de 32 bits étaient le minimum nécessaire (Larsen and McAllister, 2001). Ces premiers tests ont démontré la possibilité d'utiliser un processeur graphique pour le calcul générique, bien que la performance n'ait pas égalé pas celle des processeurs génériques.

Suite à ces preuves de concepts, des outils ont été progressivement développés afin de faciliter le calcul générique sur processeur graphique. Initialement, les programmes sur ces engins étaient programmés en adaptant les outils pour disponibles pour les tâches graphiques. Par la suite, des standards pour le calcul générique ont été développés tels que CUDA, HAL ou CAL (Owens et al., 2008). De plus, les processeurs graphiques ont été adaptés afin de pouvoir générer des résultats à virgule flottante de 32 bits, ce qui a permis leur utilisation pour des applications réelles. Grâce à ces développements, les processeurs graphiques ont rapidement été utilisés pour le calcul scientifique. Ainsi, dès 2007 des simulations de dynamique moléculaire étaient effectuées 25 fois plus rapidement sur un processeur graphique que sur un processeur générique (Elsen et al., 2007). De plus, la performance atteinte par les processeurs graphiques pour ces simulations était similaire à celle de processeurs développés spécialement pour ce genre de simulation, tout en coûtant une fraction de leur prix (Elsen et al., 2007).

Les premiers processeurs graphiques avaient des architectures très spécialisées qui limitaient ou compliquaient leur utilité pour le calcul générique. Ainsi, ces processeurs possédaient des cœurs spécialisés pour le traitement des sommets, de la géométrie, ou des pixels. Ces cœurs de calculs possédaient des jeux d'instruction et des caractéristiques spécialisées pour leur tâche (Persson,

2007). Cela compliquait le calcul générique puisque le code devait être adapté à chacun des types de cœurs. De plus, cela peut limiter l'efficacité pour le travail graphique si une tâche utilise plus un certain type de cœur que prévu. En effet, les architectes conçoivent le processeur en considérant qu'une tâche normale utilise une certaine proportion de chaque type de cœur. Si une tâche ne respecte pas ces proportions, il y a un déséquilibre qui ralentit le traitement. La compagnie ATI a conçu une des premières cartes graphiques avec une architecture unifiée pour la console de jeux vidéos *Xbox 360* et a ensuite utilisé cette architecture pour les cartes de la génération *ATI HD 2000* (Luebke and Humphreys, 2007). Cela facilite le calcul générique puisque le code n'a plus à être adapté à chaque type de cœur. L'efficacité du traitement graphique peut aussi être optimisée puisque l'architecture peut s'adapter aux tâches demandant des proportions différentes de travaux sur des sommets, de la géométrie ou des pixels. La limitation de cette technique est la difficulté associée au partage dynamique des ressources entre les différents types de tâches.

PCI Express

Le standard PCI Express est le mode de communication le plus commun entre la carte graphique et la mémoire centrale. Cela fait en sorte qu'un coût non négligeable est associé aux transferts mémoires lors de l'exécution de code sur le processeur graphique. Le temps nécessaire aux transferts peut être plus important que le temps de traitement sur le processeur (Gregg and Hazelwood, 2011). Il est donc important de prendre ce facteur en considération lorsque le temps d'exécution sur la carte graphique est comparé à celui sur le processeur central qui a directement accès aux données. Par ailleurs, le délai imposé par les transferts mémoires peut limiter l'utilisation d'un accélérateur PCI Express pour des tâches qui ont des contraintes temps réel. Néanmoins, les cartes graphiques peuvent apporter un avantage important lorsqu'elles sont bien utilisées.

CUDA

Nvidia a introduit en 2006 l'architecture Tesla pour supporter le calcul de haute performance (Lindholm et al., 2008). Tesla offre une architecture avec des cœurs unifiés tels que les cartes *ATI HD 2000*. Tesla a été la première architecture supportant le modèle de programmation *CUDA*, conçu pour le calcul générique sur les cartes graphiques. *CUDA* est une extension des langages C et C++ pour exécuter du code sur un processeur graphique. Avec ce modèle, le développeur crée une fonction qui est appliquée sur chacun des éléments d'une grille de calcul. Cela permet aux programmes générés avec *CUDA* d'être intrinsèquement parallèles puisque les éléments peuvent être traités sur plusieurs cœurs. La capacité de programmer le code en C facilite grandement le travail des développeurs.

OpenCL

OpenCL est un modèle de programmation similaire à *CUDA*. *OpenCL* utilise aussi un modèle dans lequel une fonction est exécutée sur chaque élément d'une grille de calcul. Contrairement à *CUDA*, *OpenCL* n'a pas été développé seulement pour les cartes graphiques (Stone et al., 2010). En effet, *OpenCL* peut aussi être utilisé sur d'autres types d'accélérateurs de calcul tels que les FPGA. Ainsi, le processeur CELL peut exécuter du code *OpenCL* grâce à des outils développés par IBM. Cela illustre la portabilité du code *OpenCL* qui peut fonctionner sur plusieurs types d'accélérateurs différents.

Heterogeneous System Architecture

La fondation HSA, pour « Heterogeneous System Architecture », a été fondée en 2012 par les compagnies AMD, ARM, Imagination Technologies, MediaTek Inc et Texas Instruments (Stoner, 2012). Le but de cette fondation est de publier et populariser des standards ouverts pour faciliter le développement sur des systèmes hétérogènes composés de processeurs, de cartes graphiques et d'autres accélérateurs de calcul hétérogènes. L'architecture spécifiée par le standard est composée d'au moins un processeur central, qui gère le contexte d'exécution, et d'un agent ou plus qui reçoivent des tâches du contexte et les traitent (Foundation, 2016a). Le standard impose aussi d'avoir des sections mémoire partagées entre les agents et les processeurs qui sont utilisées pour transférer les données. Le modèle d'exécution consiste en une grille contenant les données, divisées en sections appelées groupes de travail. Un groupe de travail contient le nombre d'éléments de données pouvant être acceptés simultanément dans un même nœud de calcul d'un agent. Un groupe de travail contient plusieurs « waves », ou « vagues », qui s'apparentent à un fil d'exécution utilisant une instruction SIMD. Une vague est un concept représentant l'exécution d'une même instruction sur plusieurs éléments simultanément, il s'agit donc d'un fil d'exécution sur un agent HSA.

Circuit combinant les processeurs central et graphique

Il existe aussi des circuits intégrés qui contiennent un processeur central et un processeur graphique. Ainsi, les « APU », pour « Accelerated processing unit », contiennent un processeur et une unité de calcul, tels qu'un processeur graphique ou un FPGA, pour accélérer le calcul. Intel et AMD proposent des circuits « APU » contenant une carte graphique pour les processeurs d'ordinateurs de bureau. Ce type de circuit est aussi fréquent dans les téléphones intelligents (Yeap, 2013), et le circuit contient d'autres composants tels que la mémoire et un modem cellulaire. Ce type d'architecture regroupe tous les composants nécessaires pour former un ordinateur et est appelé un système sur une puce, ou « SoC » pour « System on Chip » en anglais. Une gamme de « SoC »

populaire dans le domaine des téléphones intelligents est la série de circuits Snapdragon offerts par Qualcomm (Qualcomm, 2011).

Les circuits intégrés contenant un processeur central et un processeur graphique ont l'avantage de diminuer les délais de communication. Avec une configuration traditionnelle, le processeur communique avec la carte graphique à travers un bus PCI Express qui est fréquemment un goulot d'étranglement, limitant la performance (Daga et al., 2011). En effet, le bus PCI Express peut transférer beaucoup moins d'information que le processeur graphique n'en traite à chaque seconde, ce qui force le développeur à fortement réutiliser les données en mémoire graphique. Cependant, cela n'est pas toujours possible et il n'est pas rare que le temps d'exécution d'une tâche sur la carte graphique soit moins important que le temps de transfert des données nécessaires pour ce travail. Combiner un processeur central et un processeur graphique sur un même circuit permet de réduire le coût de communication puisque le bus gravé sur le circuit peut être beaucoup plus rapide que le bus externe. AMD Fusion est un circuit intégrant un processeur x86 et 80 cœurs graphiques et a été comparé à un système avec une carte graphique ATI Radeon HD 5870 qui a 1600 cœurs de calcul. Les auteurs ont démontré qu'AMD Fusion peut être plus performant que le système avec une carte graphique séparée dans plusieurs cas (Daga et al., 2011). Néanmoins, AMD Fusion prend plus de temps pour transférer des ensembles de données de petite taille, ce qui semble indiquer un problème avec l'architecture.

Sandy Bridge a été le premier processeur commercialisé par Intel qui intégrait un processeur graphique sur le même circuit que le processeur central. Les communications se font à travers la cache de dernier niveau, à laquelle les deux processeurs ont accès afin d'accélérer les communications. Cependant, intégrer le processeur graphique sur le même circuit peut poser un problème de dissipation de chaleur (Rotem et al., 2012). En effet, la dissipation de chaleur est un défi sur les processeurs modernes et un processeur graphique peut consommer beaucoup d'énergie. L'architecture Sandy Bridge est conçue pour gérer dynamiquement l'attribution de puissance au processeur central et au processeur graphique afin de maximiser la performance, tout en respectant la limite de température.

2.2 Débogage

Le développement et la maintenance de logiciels sont des processus complexes qui nécessitent plusieurs outils afin d'optimiser la productivité des développeurs. Un des processus les plus importants du développement logiciel est le débogage. Cette étape arrive lorsqu'un programme n'a pas le comportement attendu. Lors du débogage, l'utilisateur pose premièrement une hypothèse pour expliquer le problème de fonctionnement (Zeller, 2009a). Il doit alors établir des prédictions sur le comportement du programme à partir de cette hypothèse. Ensuite, il vérifie si les prédictions sont respectées en testant le programme. Si les prédictions sont valides, le développeur peut raffiner et

améliorer son hypothèse et répéter le processus. Dans le cas où les prédictions ne se réalisent pas, il doit établir une nouvelle hypothèse. À la fin de ce processus itératif, le développeur devrait avoir identifié la cause du problème qu'il a observé. Une des méthodes les plus simples pour déboguer un programme est d'ajouter du code pour imprimer à l'écran des données telles que les valeurs de variables. Cela permet à l'utilisateur de suivre la progression du programme et éventuellement de comprendre le problème s'il ajoute du code au bon endroit. Cependant, cette méthode est lente et complexe puisqu'il faut continuellement modifier, recompiler et redémarrer un programme, à chaque fois que l'on veut avoir des informations supplémentaires. Pour améliorer ce processus, il est très fréquent d'utiliser un outil spécialisé appelé le débogueur.

Le débogueur est un outil qui permet de contrôler l'exécution d'un programme et d'inspecter les valeurs en mémoires lors de son fonctionnement. Les débogueurs les plus simples vont simplement montrer les instructions machines et les valeurs brutes enregistrées en mémoire au développeur. Or, les instructions machines ne sont pas faites pour être facilement interprétables par un humain. La plupart des débogueurs sont donc conçus pour utiliser des informations de débogage incluses avec le programme afin de montrer le code en langage de haut niveau à l'utilisateur (Zeller, 2009b). De plus, cela permet aussi de montrer les valeurs de variables plutôt que les valeurs brutes en mémoire.

Une méthode typique pour déboguer un programme est d'insérer des points d'arrêts à des endroits pertinents dans le code. Ces points d'arrêts sont des instructions qui lancent une interruption. Celle-ci est traitée par le système d'exploitation, et celui-ci informe le débogueur que le programme est stoppé sur un des points d'arrêt. Le débogueur donne alors le contrôle au développeur, et celui-ci peut inspecter les variables en mémoire et modifier l'état du programme (Metzger, 2004). Cette technique permet au développeur de stopper le programme à des endroits pertinents afin de comprendre ce qui se passe pour déterminer quelle est la cause du problème observé. Il peut faire avancer le programme une ligne de code à la fois afin de le ralentir pour mieux suivre l'exécution.

Les débogueurs supportent aussi des techniques plus avancées pour simplifier la tâche des développeurs. Une méthode utile est d'utiliser des points d'arrêts conditionnels (Metzger, 2004). Ceux-ci sont insérés dans le programme avec une condition qui doit être satisfaite. Si la condition est remplie, le point d'arrêt conditionnel se comporte comme un point d'arrêt normal et le contrôle est transféré à l'utilisateur. Cependant, si la condition n'est pas satisfaite, le programme continue sans que l'utilisateur ne soit alerté. Cela permet de n'arrêter le programme que dans des cas intéressants, pour réduire le nombre d'arrêts inutiles et faciliter le débogage. Cependant, les points d'arrêts conditionnels sont normalement implémentés avec des points d'arrêts typiques et le débogueur est responsable d'évaluer la condition. Il y a donc des changements de contexte entre le mode utilisateur et le mode noyau, même si la condition n'est pas satisfaite. L'exécution du programme est donc ralentie même si l'utilisateur ne reçoit pas d'alerte.

2.2.1 Débogage intrusif

Un aspect important à considérer lors du débogage est l'impact du débogueur. En effet, l'exécution lors du débogage diffère de l'exécution normale du programme, ne serait-ce que parce que le débogage ralentit le programme. Ainsi, cela peut créer des « Heisenbugs », dont le nom découle du principe d'incertitude d'Heisenberg (Zeller, 2009c). Les « Heisenbugs » sont des problèmes qui disparaissent, sont modifiés ou apparaissent en raison du caractère intrusif du débogage. Un cas typique est lorsqu'un programme avec plusieurs fils d'exécution est débogué. L'interaction entre les fils peut être perturbée par un débogueur, et cela peut modifier le comportement. Il est donc nécessaire d'avoir des outils appropriés, sans quoi on peut modifier un problème que l'on veut observer.

2.2.2 Débogage réversible

Lorsqu'un programme rencontre un problème, il n'est pas toujours facile de savoir comment il est arrivé à cet état. En effet, même en déboguant un programme il arrive que l'utilisateur n'ait pas placé de point d'arrêt au bon endroit ou anticipé un problème. Fréquemment, l'utilisateur doit alors redémarrer le programme, insérer de nouveaux points d'arrêt, et recommencer le débogage en espérant avoir placé les points aux bons endroits (Boothe, 2000). Il peut être nécessaire de recommencer plusieurs fois ces étapes. De plus, redémarrer un programme avec les mêmes valeurs de départ n'amène pas toujours au même résultat. Dans le cas d'un programme avec plusieurs fils d'exécution, il est possible que le résultat dépende de la synchronisation entre les fils (Engblom, 2012). Des outils plus sophistiqués peuvent alors être très utiles pour faciliter le débogage. Une technique possible est d'utiliser le débogage réversible.

Le débogage réversible a pour but de permettre non seulement d'avancer dans l'exécution d'un programme, mais aussi de revenir à un état précédent. Ainsi, cela signifie qu'il est possible de faire un pas vers l'arrière, de la même manière qu'un débogueur traditionnel peut faire un pas vers l'avant. Cela permet de revenir dans du code précédemment exécuté et de découvrir quel chemin a mené le programme à l'état actuel. Cela permet de ne pas avoir à redémarrer lorsque le programme rencontre un problème et de facilement en découvrir l'origine (Brook and Jacobowitz, 2007). Le débogage réversible peut être implémenté en enregistrant tous les changements de valeurs des variables. Cette technique est très coûteuse puisqu'elle nécessite beaucoup de mémoire pour l'historique des valeurs de variables, et que l'enregistrement ralentit fortement le programme. Une autre technique est d'enregistrer des états, et de collecter l'information nécessaire pour exécuter de nouveau le programme d'un état à l'autre. Il faut avoir suffisamment d'information pour s'assurer que l'exécution soit déterministe, afin de réellement avoir du débogage réversible (Engblom, 2012). Cette technique permet de limiter la quantité de données à enregistrer, et de diminuer

le ralentissement. Néanmoins, le ralentissement peut être important, et cela peut affecter l'exécution d'un programme. Ainsi, un programme à plusieurs cœurs peut être suffisamment ralenti par le débogage réversible pour que certains problèmes de type « Heisenbug » se présentent.

Undo

Plusieurs produits disponibles offrent un débogueur réversible. Undo est une compagnie proposant un logiciel propriétaire permettant le débogage réversible. Ce débogueur utilise des captures d'état et la compilation à la volée du code de l'application pour capturer les sources de comportement non déterministe (Undo, 2017). Les résultats non déterministes sont enregistrés dans un historique et sont utilisés avec les captures d'états pour restaurer n'importe quel état passé. Les développeurs affirment obtenir un ralentissement de moins de 5 fois dans la plupart des applications. Cependant, Undo sérialise les fils d'un programme à plusieurs fils afin d'enlever les sources d'incertitudes dues à la synchronisation. Cela permet d'obtenir un résultat déterministe utilisable pour le débogage réversible, mais va certainement fortement ralentir ces programmes. En effet, si on sérialise le travail normalement effectué en parallèle, le programme peut être beaucoup plus que 5 fois plus lent. De plus, on impacte fortement l'exécution du programme, et cela peut modifier des problèmes liés au parallélisme.

rr

Une autre solution disponible est le logiciel rr développé par Mozilla. Il s'agit d'un logiciel libre qui emploie une technique similaire à Undo et enregistre les résultats non déterministes, tels que les appels au système d'exploitation (Mozilla, 2017). Cependant, RR n'est pas un débogueur réversible. Le logiciel enregistre une trace de l'exécution avec un ralentissement pouvant être aussi bas que 20%. Ensuite, l'utilisateur peut faire rejouer l'exécution et la déboguer interactivement avec GDB en insérant des points d'arrêts. Cependant, rr sérialise aussi les fils d'exécution, ce qui peut poser un problème avec les programmes à plusieurs fils.

Support matériel

Il est parfois possible d'utiliser les capacités matérielles d'un processeur pour permettre le débogage réversible. Ainsi, certains processeurs ont des composantes dédiées à l'enregistrement de l'exécution d'un programme. Par exemple, certains processeurs Intel supportent la fonctionnalité « Intel Processor Trace ». Cette fonctionnalité permet d'enregistrer le minimum d'information nécessaire pour reconstituer l'exécution d'un programme (Kleen and Strong, 2015). Ainsi, un seul bit peut être nécessaire afin de déterminer quelle branche un programme a pris dans une condition.

De plus, le matériel dédié permet d'accélérer l'enregistrement de l'information. Cela permet de minimiser le ralentissement dû au débogage réversible. Néanmoins, même le support matériel n'est pas toujours suffisant, puisque de très grandes quantités de données peuvent être collectées et le ralentissement peut être non négligeable.

2.2.3 Débogage asynchrone

Lors du débogage d'un programme à plusieurs fils, il arrive que certains fils ne doivent pas être stoppés. C'est le cas si un fil est en charge un compteur « watchdog » qui redémarre le système s'il tombe à zéro. Il est aussi possible qu'un fil produise des données, et que l'arrêter bloque le programme. Finalement, il arrive aussi qu'on veuille minimiser l'impact, et donc qu'on ne veuille pas arrêter tous les fils. Un débogueur traditionnel agit normalement de manière synchrone, si un point d'arrêt est frappé, tous les fils sont stoppés simultanément. Dans le cas d'un compteur « watchdog », cela peut faire en sorte que le compteur n'est pas incrémenté et qu'il tombe à zéro lorsque tous les fils sont arrêtés. Il est alors intéressant d'utiliser un débogueur asynchrone permettant d'avoir des fils qui fonctionnent alors que d'autres sont stoppés (Sidwell et al., 2008). Ce type de débogage est aussi appelé « mode sans arrêt », ou « non-stop mode » en anglais. Cela permet de déboguer un système qui ne supporte pas que tous les fils soient arrêtés. De plus, l'impact sur le fonctionnement est limité puisqu'une partie des fils peuvent continuer leur travail, lorsqu'un fil est arrêté pour être inspecté.

2.2.4 Débogueurs disponibles

LLDB

Le débogueur LLDB fait partie de la suite d'outils LLVM, tels que le compilateur LLVM ou Clang. Le projet LLVM a été conçu pour être fait d'un ensemble de librairies qui peuvent facilement interagir. Ainsi, LLDB peut réutiliser des outils faisant partie de LLVM. Le débogueur supporte des opérations typiques telles que l'insertion de points d'arrêt, le débogage à distance, stopper, résumer ou avancer un programme une ligne à la fois, l'inspection des valeurs de variables, les programmes à plusieurs fils d'exécution, etc.

GDB

GDB est un débogueur libre de la « Free Software Foundation » dont la première version date de 1986. Il s'agit d'un débogueur qui est toujours en développement actif par plusieurs entreprises, et un très grand nombre d'architectures et de systèmes d'exploitation sont supportés. GDB supporte le C, C++ ainsi que d'autres langages tels que D, Go, Fortran, etc. Il supporte les opérations de

débogage de base, telles que les points d'arrêt et le contrôle manuel de l'exécution, ainsi qu'un grand nombre d'opérations plus avancées (Stallman et al., 2017). Ainsi, il permet de faire du débogage à distance, il offre une interface de communication pour les débogueurs graphique, permet de générer et utiliser des « core dump », etc. GDB supporte aussi les points de trace, et offre différents types de points de trace, tel qu'expliqué dans la section sur le traçage. Une librairie, appelée « In-Process Agent Library », servant à amener de la logique dans l'espace mémoire du programme débogué, permet d'accélérer certaines opérations. GDB offre aussi la possibilité d'intégrer des programmes Python écrits par l'utilisateur, pour créer facilement de nouvelles commandes ou modifier le comportement d'opérations existantes.

2.2.5 Interface graphique pour le débogage

La méthode la plus simple pour déboguer un programme est d'utiliser un débogueur à travers un terminal. Cela nécessite normalement une faible quantité de ressources et permet de démarrer rapidement une session de débogage. Cependant, utiliser un débogueur à travers un terminal n'est pas toujours intuitif, et la courbe d'apprentissage peut être longue. En effet, l'utilisateur doit entrer une commande pour obtenir des informations tels que l'emplacement du programme, le nombre de points d'arrêt, etc (Matloff and Salzman, 2008). Il faut donc que l'utilisateur se rappelle de ces commandes.

Une alternative qui permet souvent de faciliter l'utilisation d'un débogueur est d'utiliser une interface graphique. Il s'agit d'un programme composé d'au moins une fenêtre affichant des informations à l'utilisateur. Dans la plupart des cas, ce programme communique avec un débogueur et sert d'intermédiaire pour aider l'utilisateur (Zeller, 2001). La fenêtre principale dans la plupart des interfaces graphiques affiche le code source du programme, et indique où un fil est situé lorsque celui-ci est arrêté. L'utilisateur peut alors insérer des points d'arrêt simplement en cliquant sur une ligne du code source. Il est aussi possible de sélectionner des variables dans cette fenêtre afin de voir leur valeur courante. Ces opérations se font facilement avec la souris et l'utilisateur n'a pas à apprendre de commande.

De plus, les interfaces graphiques offrent normalement plusieurs autres vues pour faciliter le débogage (Microsoft, 2015a). Ainsi, il existe des fenêtres pour afficher les variables locales et leur valeur, afin que l'utilisateur puisse les voir rapidement sans avoir à cliquer sur chacune d'entre elles. Il est aussi possible d'afficher la pile d'appels d'un fil ou les valeurs des registres. Une vue très importante est la vue des fils d'exécution. Celle-ci affiche les différents fils ainsi que leur état pour permettre à l'utilisateur de choisir un fil et de déboguer un programme à plusieurs fils. Cette vue peut être combinée avec la vue de la pile d'appels, tel que montré dans la Figure 2.3. Les différentes fenêtres disponibles permettent de faciliter le travail de l'utilisateur en affichant l'information d'une

manière conviviale.

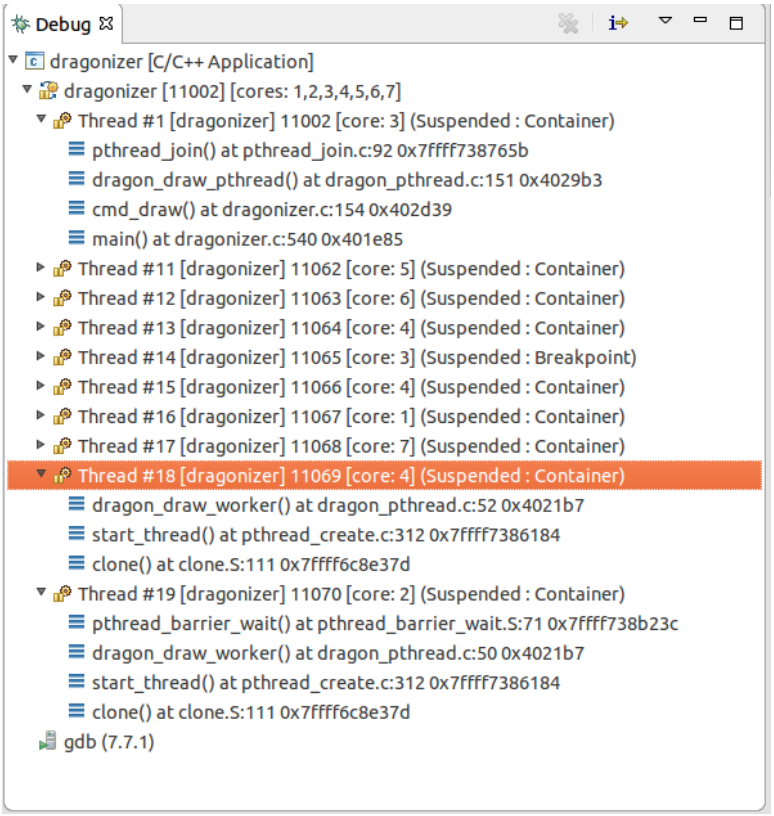

Figure 2.3 Fenêtre affichant les fils d'exécution et leur état dans Eclipse Neon CDT

Certains débogueurs graphiques sont inclus dans un environnement de développement, appelé « IDE » pour « Integrated Development Environment » en anglais. Un « IDE » est un programme avec une interface graphique qui vise à intégrer tous les outils et étapes nécessaires pour le développement logiciel (Wasserman and Pircher, 1987). Ainsi, ce type de programme permet normalement de gérer le code source et les ressources associées à plusieurs projets informatiques, écrire et compiler du code, utiliser un outil de gestion de version, etc. Un « IDE » intègre normalement un débogueur afin de permettre à l'utilisateur de tester le code écrit.

DDD

DDD est une interface graphique libre pour différents débogueurs tels que GDB, DBX, etc (Project, 2013). Ce programme fait partie de la suite d'outils offerts par l'organisation « Free Software Foundation ». Ce programme permet de naviguer à travers le code, insérer des points d'arrêt, inspecter des variables, déboguer des systèmes distants, etc (Zellers, 2004). Ce programme est toujours disponible, mais ne semble plus être activement développé. En effet, la dernière version, 3.3, date de

2009.

Eclipse CDT

Eclipse CDT est un environnement de développement libre pour les langages C et C++ basé sur la plateforme Eclipse. Le débogueur communément utilisé avec Eclipse CDT est GDB. Cet « IDE » supporte les opérations courantes telles que l'affichage des fils d'exécution, l'affichage et la manipulation des valeurs de variables, l'insertion de points d'arrêt, etc (Khouzam, 2016). Eclipse CDT permet aussi à l'utilisateur de faire des opérations plus avancées. Il permet notamment d'insérer dynamiquement des directives pour imprimer du texte à l'écran sans recompiler le programme, de modifier l'affichage par défaut d'un type, d'utiliser des filtres pour grouper des données ou d'utiliser le débogage asynchrone ou réversible.

Une vue d'Eclipse CDT développée spécifiquement pour les applications à plusieurs fils est le « Multicore Visualizer ». Les développeurs, en collaboration avec les fabricants du Parallella (un accélérateur de calcul largement multi-cœurs), ont réalisé que la vue traditionnelle de débogage n'affiche pas bien lorsqu'un grand nombre de fils d'exécution sont présents. Ils ont développé une vue qui donne un aperçu des cœurs d'un processeur, et des fils en exécution sur chacun de ces cœurs. Cela permet de donner une vision globale du système, plus intuitive qu'une simple liste de fils d'exécution. Il s'agit du « Multicore Visualizer », dont une capture d'écran durant une session de débogage est montrée à la Figure 2.4. On y voit les huit cœurs virtuels du processeur Intel i7-4790 utilisé, ainsi que les fils d'exécutions du programme débogué. L'état et l'identifiant de chaque fil est donné, et il est possible de sélectionner un fil à partir de cette vue.

Visual Studio

Visual Studio est un environnement de développement offert par Microsoft. Un débogueur propriétaire est inclus avec cette interface (Microsoft, 2015b). Ce débogueur supporte les opérations de base, et offre plusieurs capacités avancées. Ainsi, il est possible d'éditer du code sans redémarrer un programme, de débogueur un programme sur carte graphique, de déboguer à distance, etc. Visual Studio offre aussi des vues pour afficher efficacement des fils d'exécution. Un outil de Microsoft, IntelliTrace, est disponible pour les programmes .Net roulant en C# et Visual Basic. Cet outil enregistre automatiquement différents évènements tels que les exceptions, traitées ou non, les évènements reliés aux accès à des fichiers, aux interactions de l'utilisateur avec l'interface graphique, etc (Jones et al., 2016). Il est aussi possible d'enregistrer les appels de fonctions pour avoir des piles d'appels à n'importe quel moment. Il est ensuite possible d'inspecter cet historique avec Visual Studio.

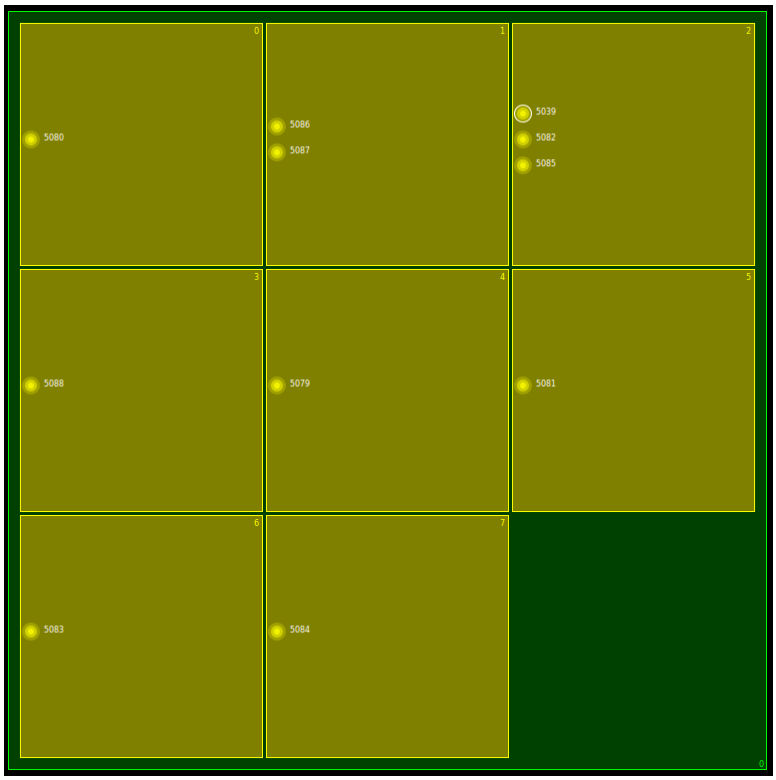

Figure 2.4 Multicore Visualizer disponible avec Eclipse CDT

Totalview

Totalview est un débogueur graphique propriétaire développé par Rogue Software. Il s'agit d'un outil pour le calcul haute performance, et permet de déboguer du code distribué sur un super-ordinateur (RogueWave). Il supporte aussi le débogage de programme utilisant des accélérateurs de calcul tels qu'une carte graphique ou un Xeon Phi. Ce débogueur peut aussi afficher de larges quantités d'information efficacement, pour le débogage réversible et pour la visualisation et le contrôle de larges quantités de fils d'exécution.

2.3 Instrumentation

2.3.1 Instrumentation statique

À des fins de débogage, d'évaluation de la performance ou simplement pour garder un historique de l'exécution d'un programme, il peut être nécessaire d'instrumenter un programme. La méthode la plus simple pour y parvenir est d'insérer des directives afin d'imprimer des informations pertinentes. Il s'agit d'une méthode élémentaire et peu efficace pour déboguer un programme (Binstock, 2013). Néanmoins, cette technique illustre ce qu'est l'instrumentation statique d'un programme.

Cela consiste à instrumenter certains endroits dans le code d'un programme. Il est possible d'insérer des directives pour enregistrer des données dans un historique, par exemple pour faire un journal de transactions.

Une technique plus avancée pour instrumenter un programme est d'insérer des points dans le code où un outil peut s'attacher. Cependant, si aucun outil n'active ces points, le programme fait simplement ignorer ces points d'instrumentation (Desnoyers, 2017). Un but de ce type de point d'instrumentation est d'être inséré dans des programmes utilisés en production. Ainsi, un outil peut s'attacher au programme durant son exécution afin de récolter des informations. Cependant, il est nécessaire de minimiser l'impact lorsqu'aucun outil n'est attaché. En effet, il faut que le ralentissement soit minimal, sinon il n'est pas intéressant d'utiliser ce type d'instrumentation. Les points d'instrumentations du noyau Linux ont un coût très faible lorsqu'ils ne sont pas activés. Ils rajoutent simplement l'évaluation d'une condition et occupent une faible quantité d'espace. Lorsque l'instrumentation est activée, le point de trace redirige l'exécution vers le code approprié.

2.3.2 Instrumentation dynamique

Uprobe, un traceur pour le mode utilisateur, est une fonctionnalité du noyau Linux disponible à partir de la version 3.5. Il s'agit d'un projet dérivé de utrace, venant de SystemTap. Uprobe permet d'insérer un point d'instrumentation en remplacement d'une instruction dans un fichier contenant du code, et de définir le comportement lorsque ce point est frappé (Corbet, 2012). Tous les programmes qui exécutent ce code frappent donc ce point. Lorsque cela arrive, la fonction spécifiée lors de l'instrumentation est appelée en mode noyau. L'instrumentation d'un programme avec Uprobe utilise une instruction provoquant une interruption, tel INT sur Intel x86. Cela provoque un changement de contexte à chaque fois que l'instruction originale est rencontrée. Étant donné qu'un changement de contexte est coûteux, le ralentissement associé à Uprobe peut être important. Cependant, cela permet de corréler des traces en mode usager avec des traces du système d'exploitation.

Une méthode plus sophistiquée pour instrumenter un programme est d'utiliser le code d'origine comme donnée. Dans ce cas, ce code peut être exécuté dans une machine virtuelle. Dans ce cas, cette machine virtuelle simule des registres, et émule le code original du programme. Cette technique est utilisée par la suite d'outils libres Valgrind (Nethercote and Seward, 2007). Cela permet de modifier le code réellement exécuté et d'y insérer des points d'instrumentation. Valgrind utilise aussi cette technique pour implémenter une mémoire fantôme. Cette mémoire fantôme permet de mieux observer les opérations mémoires et est notamment utilisée pour *memcheck*. Memcheck permet de tracer un programme et de vérifier qu'il n'accède pas à des valeurs non initialisées. L'utilisation d'une machine virtuelle pour émuler le code ainsi que d'une mémoire fantôme ralentit

fortement le programme. Cependant, cela permet d'implémenter une instrumentation très poussée.

Il est aussi possible d'utiliser le code initial en tant que donnée et de le recompiler pour l'instrumenter. L'outil Pin offert par Intel utilise cette technique (Luk et al., 2005). Pin utilise la compilation en temps réel pour y parvenir. Le programme initial est exécuté par une machine virtuelle dans Pin. La machine virtuelle lit la séquence d'instruction et utilise la compilation en temps réel pour générer un code instrumenté correspondant. Ce code est alors exécuté directement sur le processeur. Pin ne recompile une partie du programme que lorsque le code correspondant est appelé. Si un branchement n'est jamais atteint, il ne sera pas recompilé, afin de minimiser le ralentissement. L'utilisation de la compilation à la volée permet une instrumentation efficace, et Pin est jusqu'à 3 fois plus rapide que Valgrind.

Une autre technique possible pour l'instrumentation dynamique est l'utilisation d'une trampoline. Dans ce cas, une instruction est remplacée par un saut (Hazelwood, 2011). Dans ce cas, l'outil va insérer du code d'instrumentation dans la mémoire du programme. Ensuite, il va sélectionner une instruction à l'endroit où le point d'instrumentation doit être inséré. Cette instruction va être remplacée par un saut jusqu'au début de la section d'instrumentation. L'instruction originale est exécutée à la fin de la section d'instrumentation, avec des modifications si nécessaire pour gérer les changements d'adresse. Cette technique entraîne un ralentissement très faible, étant donné qu'elle ne fait que rajouter un saut et une série d'instructions à exécuter. Le ralentissement est donc similaire à un appel de fonction. Il n'y a pas d'interruption ou de changement de contexte. Cependant, les architectures avec des tailles d'instruction variables posent un problème puisqu'il faut sélectionner une instruction à remplacer assez large. En effet, il faut que cette instruction soit au moins aussi grande qu'un saut, sans quoi il n'est pas possible de sauter jusqu'au code d'instrumentation.

Il est possible d'éviter cette limitation sur la taille de l'instruction à remplacer. En effet, Dyninst permet d'instrumenter un programme à n'importe quel endroit dans le code (Bernat and Miller, 2011). Pour ce faire, il est possible de déplacer plusieurs instructions ou un bloc complet de code. Cela crée alors assez d'espace pour insérer un saut jusqu'à la section de code d'instrumentation. Cette section contient aussi toutes les instructions originales, qui peuvent avoir été modifiées pour prendre en compte le déplacement. Si un outil utilise cette technique, il doit s'assurer qu'il n'y ait pas de saut au milieu de la section de code déplacée. En effet, cela risque de créer un problème puisqu'une partie d'instruction peut avoir été réécrite. Pour éviter cela, l'outil peut lire le code pour vérifier qu'aucun saut ne pointe dans la section déplacée. Il peut aussi déplacer une fonction complète, si le programme saute toujours au début de la fonction.

2.4 Traçage

2.4.1 Mise à l'échelle

Pour faire usage d'un processeur multi-cœur efficacement, un programme doit être conçu attentivement. En effet, les fils peuvent essayer d'accéder à des ressources communes simultanément, et cela peut créer des problèmes. Ainsi, il est possible que l'opération qu'un fil fait sur une variable soit effacée si un autre fil essaie simultanément de l'affecter. Les régions de code contenant des opérations sur des ressources communes sont appelées des sections critiques (Tanenbaum and Bos, 2014). Une section critique contient du code qui ne doit pas être exécuté par plus d'un fil simultanément. Si plusieurs fils sont dans une section critique en même temps, il risque d'y avoir corruption de données et un résultat indéfini.

Lorsqu'une section critique n'est pas protégée, il peut arriver un phénomène de course de donnée. Une course de donnée survient lorsque plusieurs fils accèdent simultanément à une variable et que le résultat dépend de l'ordre d'accès. Il s'agit d'un problème trivial qui n'est pas nécessairement facile à identifier, et ce problème est connu depuis plus d'une décennie (Karam and Buhr, 1990). La course de donnée peut arriver lorsque deux fils d'exécution copient une variable globale en mémoire et l'utilisent dans des calculs. Ensuite, chaque fil recopie la valeur locale de la variable, qui a été modifiée par les calculs, dans la variable globale. Or, le deuxième fil peut alors écraser la valeur générée par le fil qui a recopié sa variable en premier. La valeur finale de la variable globale va donc dépendre de l'ordre d'exécution des fils. Puisque cet ordre dépend de multiples facteurs tels que l'ordonnancement, les fautes de caches, etc, le résultat est non-déterministe. De plus, les copies des valeurs des variables sont souvent implicites. En effet, le compilateur copie des valeurs dans les registres du processeur pour effectuer des calculs, et recopie le résultat dans la variable globale. Une ligne de code contenant une multiplication peut ainsi cacher une course de données.

L'accès à une section critique doit donc être contrôlé afin d'éviter les problèmes. Il faut faire usage d'exclusion mutuelle, ce qui signifie simplement empêcher que plus d'un fil soit simultanément dans une même région. L'exclusion mutuelle peut être faite avec un simple verrou qui doit être acquis par un fil avant d'accéder à la section critique. Cependant, la performance d'un programme à plusieurs fils peut fortement souffrir en raison de l'exclusion mutuelle. Pour cette raison, des techniques pour limiter l'impact de l'exclusion mutuelle sont développées depuis plusieurs décennies (Lamport, 1987). De plus, l'exclusion mutuelle peut mener à la famine de certains fils si elle n'est pas bien effectuée. La famine survient lorsqu'un fil ne peut accéder à une ressource durant longtemps, par exemple si cette ressource est continuellement utilisée par d'autres fils.

Le traçage est intéressant pour les systèmes multi-cœurs puisque son but est de capturer des évènements avec un ralentissement minime. Cela permet donc d'obtenir une trace représentant une

exécution typique du programme, ce qui permet de diagnostiquer des problèmes survenant lorsque le programme n'est pas tracé. Cependant, cela implique que le traceur lui-même soit assez efficace. Or, les architectures utilisées pour la collection de données peuvent être problématiques. Ainsi, certains traceurs utilisent un tampon global pour stocker les données. Par exemple, le traceur noyau « Linux Trace Toolkit », prédécesseur de LTTng, utilisait un verrou pour limiter l'accès à un tampon global (Torvalds, 2002).

K42 est un système d'exploitation libre développé à des fins de recherche par IBM. Un des buts de ce système d'exploitation est d'offrir une architecture de traçage intégrée, afin de pouvoir efficacement inspecter le fonctionnement du système. K42 a proposé un système utilisant des instructions atomiques, dont le coût est beaucoup plus faible qu'un verrou, afin de synchroniser l'accès à un tampon global (Wisniewski and Rosenburg, 2003). Lors de l'écriture d'une trace, un fil essaie de réserver de l'espace à l'aide d'une instruction atomique dans le tampon, répétant la demande jusqu'à obtention de l'espace. Ensuite, le fil écrit les données dans le tampon et continue. Cette architecture offre une bien meilleure performance qu'un tampon géré par un verrou, puisqu'une instruction atomique est traitée plus rapidement. De plus, le traceur de K42 permet de capturer des données de taille variable, ce qui donne plus de flexibilité au traceur. Néanmoins, l'usage d'un tampon unique va nécessairement amener de la contention, même si des instructions atomiques sont utilisées. De plus, il faut que l'instruction atomique verrouille le bus afin d'assurer qu'aucun autre cœur n'accède au compteur du tampon simultanément. Ainsi, cette technique peut tout de même être problématique pour un programme à plusieurs fils.

2.4.2 Traceurs du noyau d'exploitation

KProbes

Kprobes est une fonctionnalité du noyau Linux pour permettre l'instrumentation dynamique. Kprobes est un système pour insérer des points d'interruption dans le code noyau et enregistrer des fonctions de rappel associées. Ainsi, cela fonctionne de manière similaire à Uprobe (Keniston et al., 2017). L'instrumentation avec Kprobes provoque des interruptions, ce qui amène un ralentissement plus important que des points d'instrumentation statiquement définis à la compilation.

Ftrace

Le système d'exploitation Linux offre le traceur Ftrace, qui est inclus nativement. Ftrace peut s'attacher à des points d'instrumentation préalablement insérés dans le code du noyau Linux, avant la compilation. De plus, le traceur peut aussi s'attacher aux prologues de fonction qui ont été implicitement instrumentées à la compilation. En effet, lorsque le noyau est configuré pour sup-

porter le traçage, la compilation du code ajoute des points d'attache en prologue à chaque fonction. Ces points peuvent ensuite être utilisés par Ftrace ou d'autres traceurs (Bird, 2009). Les points de traces sont enregistrés dans des tampons circulaires. Chaque cœur possède son propre tampon afin d'être efficace sur un système multi-cœur. Finalement, des instructions atomiques sont utilisées pour synchroniser les accès aux tampons. Cela permet d'éviter d'utiliser un verrou et minimise le ralentissement.

SystemTap

SystemTap est un traceur noyau inspiré du traceur Dtrace. SystemTap permet d'écrire du code d'instrumentation dans un langage inspiré du C. Ce langage permet de définir des points d'instrumentation et de spécifier leur emplacement (Eigler and Hat, 2006). Il permet aussi aux usagers de définir des boucles, conditions, et d'autres opérations mathématiques. Ainsi, les utilisateurs peuvent écrire du code pour vérifier des conditions, modifier ou agréger des valeurs lorsque le point d'instrumentation est rencontré. Le code écrit pour SystemTap est compilé et est inséré dans le noyau du système d'exploitation. Cette technique pose certains risques puisque le code d'instrumentation est exécuté avec les privilèges du système d'exploitation. Les architectes de SystemTap ont intégré des étapes de vérification du code lors de la compilation. Néanmoins, il y a toujours une possibilité que cette technique cause des échecs du système d'exploitation (Gregg, 2011). De plus, la compilation des codes d'instrumentation impose un délai lors de l'instrumentation du système d'exploitation.

LTTng

LTTng est un traceur noyau pour Linux développé par Efficios. Ce traceur est développé spécifiquement pour des systèmes haute performance, multi-cœurs et temps réel. Il supporte la collection de traces de tailles différentes. Ce traceur utilise les points d'instrumentation statiques insérés dans le noyau Linux pour collecter des données. La Figure 2.5, tirée de The LTTng Project (2017), montre l'architecture de LTTng. LTTng utilise une architecture composée de plusieurs programmes, ou « daemon », ayant chacun un rôle spécifique. Le module noyau LTTng enregistre les données dans le tampon, et le « consumer daemon » accède aux tampons en mémoire partagée pour enregistrer les traces sur disque, ou les envoyer par le réseau. Le contrôle se fait dans un terminal grâce à « liblttng-ctl », et le « session daemon » est responsable de gérer les sessions de traçage, ainsi que de démarrer et stopper le traçage.

LTTng utilise une architecture avec un tampon circulaire, formé de plusieurs sous-tampons, par cœur. De plus, la synchronisation pour gérer l'accès aux tampons est effectuée à l'aide d'instructions atomiques, de manière similaire à K42. Cependant, l'implémentation diffère de K42, notamment parce que LTTng est destiné aux systèmes de productions et que K42 utilise une architecture

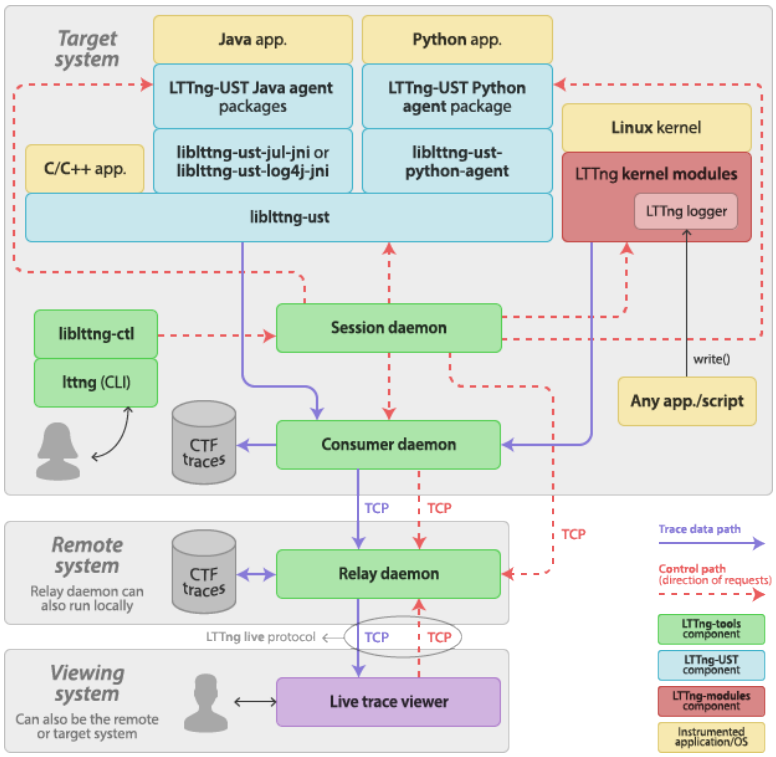

Figure 2.5 Control and trace data paths between LTTng components. ©The LTTng Project 2014–2017. Reproduit avec permission.

insuffisante d'un point de vue sécuritaire. De plus, l'architecture utilisée par K42 ouvre la porte à des corruptions de données dans le cas de course de données, même si cela est peu probable (Desnoyers and Dagenais, 2012). LTTng utilise des instructions atomiques locales, qui n'affectent pas l'ensemble des cœurs, et qui ont donc un impact plus faible que les instructions atomiques globales, pour l'écriture des traces. LTTng n'utilise des instructions atomiques globales qu'en lecture des tampons, et cela survient plus rarement que l'écriture de trace. Il en découle que le ralentissement associé à LTTng est faible.

2.4.3 Traceurs pour les programmes utilisateurs

Strace

Strace est un outil pour tracer les appels système faits par un programme. Il intercepte les appels système du programme et imprime l'appel avec les paramètres à l'écran (Gregg, 2014). Ce traceur a l'avantage d'être très simple à utiliser et d'être inclus par défaut avec Linux. Il suffit de lancer strace et de lui donner le programme à tracer avec les bons arguments pour faire une expérience. Cependant, le ralentissement associé est très important et l'exécution peut prendre jusqu'à 100 fois

plus de temps. De plus, il ne trace que le programme donné en argument.

Valgrind

Valgrind est un outil pour instrumenter un programme qui peut permettre de tracer celui-ci. Valgrind propose une plateforme utilisant l'émulation du code original avec une machine virtuelle ainsi qu'une mémoire fantôme sur laquelle plusieurs outils sont bâtis. Ainsi, Callgrind enregistre les appels de fonction et des statistiques telles que le nombre d'instructions exécutées ou le nombre de fautes de cache (Weidendorfer, 2008). L'utilisateur peut ensuite utiliser Kcachegrind pour visualiser un résumé de ces données et identifier les problèmes de performance. Puisque Valgrind utilise l'émulation du code original, le ralentissement associé est très important. Ce n'est donc pas un outil destiné à être utilisé dans un environnement de production.

GDB

GDB est un débogueur qui permet de tracer un programme en mode utilisateur dynamiquement. Un avantage associé à la combinaison d'un débogueur et d'un traceur est d'utiliser le débogueur afin de présenter le code source. Cela permet à l'utilisateur de choisir l'emplacement des points de trace à insérer dynamiquement, plutôt que lors de l'écriture du code. GDB supporte un mode de traçage basé sur des points d'arrêt (Shebs, 2009). Ce mode de traçage ralentit fortement le programme, puisqu'il provoque un changement de contexte pour chaque point de trace enregistré. Cependant, GDB supporte aussi un mode de traçage rapide. Ces points de trace rapides sont insérés avec l'aide d'une trampoline. Ainsi, une instruction est remplacée par un saut vers le code d'instrumentation. Le ralentissement associé aux points de trace rapides est donc très faible. Cependant, le traceur intégré dans GDB a des problèmes avec la mise à l'échelle sur des processeurs multi-cœurs, puisqu'il utilise un seul tampon pour stocker les traces. L'accès à ce tampon est contrôlé par un verrou actif, ce qui peut fortement pénaliser les programmes avec plusieurs fils d'exécution.

LTTng-UST

LTTng-UST est le traceur pour les programmes en mode utilisateur associé au traceur noyau LTTng. Il est basé sur l'architecture du traceur LTTng noyau. Lors de l'écriture du programme, le développeur insère de points de trace à des endroits clés et spécifie les variables à enregistrer. Lors de l'exécution, le programme est dynamiquement lié à la librairie « liblttng-ust » qui définit un tampon circulaire pour chaque cœur (Fournier et al., 2009). Chaque tampon est composé de plusieurs sous-tampons, et la gestion de ceux-ci est effectuée grâce à un usage minimal d'instructions atomiques. Cela permet d'éviter d'utiliser des verrous, et diminue fortement le ralentissement dû

au traçage. Les tampons sont placés dans des pages mémoire partagées avec un « daemon », un processus d'arrière-plan. Ce « daemon » est responsable de vider les sous-tampons au fur et à mesure qu'ils sont remplis, et de transférer les traces vers le disque ou le réseau. Si le « daemon » n'a pas le temps de vider un tampon, des évènements sont perdus afin de ne pas ralentir le programme.

Uftrace

Uftrace est un traceur de fonctions libre, inspiré du traceur noyau ftrace. Ce traceur peut s'attacher aux débuts de fonctions lorsqu'un programme a été compilé pour supporter l'instrumentation (Kim, 2017). Cela peut être fait en activant l'option « -finstruments-function » du compilateur. Il produit un historique de l'entrée de chaque fonction. Cela peut être utilisé pour créer une vue de la pile d'appels en fonction du temps.

Extrae

Extrae est une suite d'outils développés par le centre de calcul de haute performance de Barcelone (Barcelona Supercomputing Center, 2017a). Il s'agit d'une suite d'outils qui offre différentes manières d'instrumenter un programme pour capturer différents types de traces. Ainsi, Extrae permet d'enregistrer des évènements reliés à OpenMP, Pthread, CUDA et MPI en utilisant des libraries dynamiques d'instrumentation. De plus, Extrae peut faire usage de Dyninst afin d'ajouter dynamiquement des points de traces dans un programme. Finalement, Extrae offre aussi d'autres manières d'instrumenter des programmes utilisant MPI, et permet d'analyser le fonctionnement d'un programme en utilisant l'échantillonnage.

2.5 Analyse de données de traçage

Lorsqu'une expérience de traçage est terminée, ou même durant l'expérience, il est nécessaire d'analyser les données récoltées. En effet, un point de trace peut être frappé extrêmement fréquemment, ce qui fait en sorte que la trace produite peut contenir un très grand nombre d'évènements. Or, il n'est ni facile ni efficace pour un humain de lire une longue série d'évènements afin de comprendre le fonctionnement d'un système. Il est donc nécessaire d'utiliser une méthode qui traite les données et en fait ressortir des informations pertinentes.

2.5.1 Analyses automatisées

Une méthode possible est d'utiliser un programme pour lire les données et analyser automatiquement certaines métriques. Le programme peut alors donner comme résultat un résumé créé à partir

des évènements.

LTTng analyses

LTTng offre des programmes pour analyser automatiquement certains aspects d'un programme à partir d'une trace. Ainsi, il est possible de créer un graphique de la distribution de la latence lors d'un transfert entrée/sortie (Desfossez, 2017). Une autre analyse donne des statistiques sur les appels système effectués par les différents fils d'exécution, tel que montré à la Figure 2.6. On y voit les appels système effectués par chaque fil d'exécution, ainsi que des statistiques sur leur durée (en micro-secondes) et leur valeur de retour. D'autres types d'analyses sont aussi disponibles, telles qu'une analyse retournant les statistiques d'appels système ou des interruptions logicielles et matérielles. Finalement, un utilisateur peut aussi écrire ses propres scripts d'analyse en utilisant l'API Python de babeltrace (Efficios, 2016).

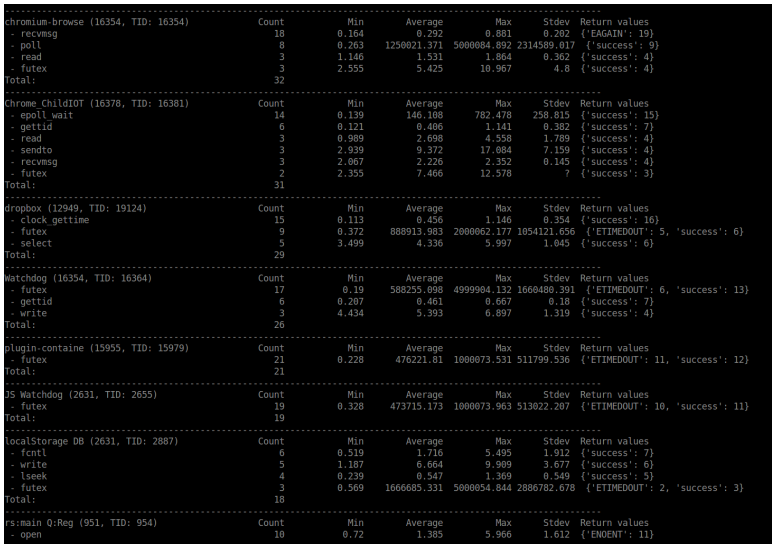

Figure 2.6 Analyse des appels systèmes avec LTTng analyses

CallGrind

Callgrind, un outil disponible avec Valgrind, utilise aussi une analyse automatisée. Ainsi, il enregistre les appels de fonction d'un programme utilisateur, et KCacheGrind peut utiliser cette trace pour afficher un résumé des fonctions appelées et les classer par le temps pris par chacune de ces fonctions (Weidendorfer, 2008).

DTrace

Certains traceurs offrent aussi la possibilité d'écrire des programmes afin de traiter les données lors de l'enregistrement des traces. Cela peut notamment permettre de réduire la quantité de données à enregistrer grâce à un prétraitement. DTrace est un traceur noyau disponible sur Solaris qui permet d'écrire des programmes appelés lorsqu'un point de trace est frappé. L'utilisateur peut alors définir des actions à prendre, effectuer des opérations sur différentes variables et enregistrer des informations (Beauchamp and Weston, 2008). Un désavantage de cette méthode est d'amener un plus grand ralentissement, puisque le traceur doit exécuter plus d'opérations.

2.5.2 Interfaces graphiques

Une technique possible, pour offrir à l'utilisateur une manière efficace pour comprendre les données, est d'utiliser une interface graphique. Cela permet d'afficher l'information d'une manière plus facilement compréhensible pour un humain.

Le développement d'interfaces graphiques, pour afficher efficacement les traces d'exécution d'un programme, existe depuis plus de 20 ans. Ainsi, ParaGraph est un outil proposé en 1991 pour visualiser des traces d'exécution d'un programme (Heath and Etheridge, 1991). Ce programme est principalement destiné à l'analyse de programmes parallèles roulant sur des systèmes distribués. En effet, la majorité des systèmes parallèles disponibles à ce moment étaient composés d'un grand nombre de nœuds, ayant chacun un processeur, connectés ensemble. Cet outil offre des vues pour afficher l'utilisation des processeurs, les communications entre les nœuds, l'état des tâches, etc.

Jumpshot est un visualisateur pour programmes parallèles développé en Java dans les années 90. Un des domaines ciblés par Jumpshot est l'analyse de programmes utilisant MPI, et le visualisateur est fourni avec une librairie d'instrumentation pour MPI. Jumpshot fournit plusieurs types de vues, telles qu'une vue de l'état d'attributs en fonction du temps ou des histogrammes (Zaki et al., 1999). Ces vues peuvent être utilisées pour visualiser différents types de données, telles que l'état d'un fil d'exécution utilisant MPI. On peut ainsi voir si le fil est en train de faire du calcul, en appel dans une fonction de communication MPI ou en attente d'autres fils. Les vues de Jumpshot peuvent aussi être utilisées pour visualiser d'autres informations, telles que l'accès à un disque par plusieurs fils.

Trace Compass est un programme basé sur Eclipse pour analyser des traces d'exécution d'un programme. Initialement développé pour analyser les traces du système d'exploitation créées par LTTng, il supporte toutes les traces de type CTF ainsi que d'autres formats. Trace Compass est basé sur une machine à états qui lit les traces d'exécution pour créer un modèle de l'état d'attributs en fonction du temps. Il est possible de créer une vue en définissant une machine à états, et les caractéristiques de la vue, à l'aide d'un fichier XML. Plusieurs vues sont fournies par défaut avec

Trace Compass pour analyser des traces noyau et usager. Ainsi, il est possible d'avoir une vue de la pile d'appels en fonction du temps, à partir d'une trace usager qui contient les évènements d'entrée de fonctions. La vue principale de Trace Compass pour les traces noyau est la vue « Control Flow », telle que montrée à la Figure 2.7. Cette vue montre une ligne pour chaque fil d'exécution, et colore chacun de ces fils en fonction de son état. Une autre vue disponible est la vue des ressources système, telle que montrée à la Figure 2.8. Cette vue montre l'état de différentes ressources du système, tel que montré à la Figure 2.8. Ces deux vues sont automatiquement créées par Trace Compass lors de l'importation d'une trace noyau avec les bons évènements. Il est possible de créer une telle trace en enregistrant tous les évènements noyaux avec LTTng, ce qui donne beaucoup d'information, mais risque de ralentir le système.

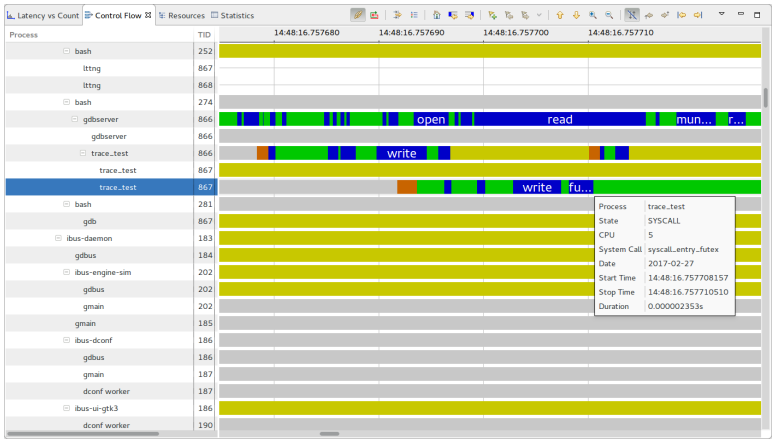

Figure 2.7 Capture d'écran de la vue « Control Flow » de Trace Compass

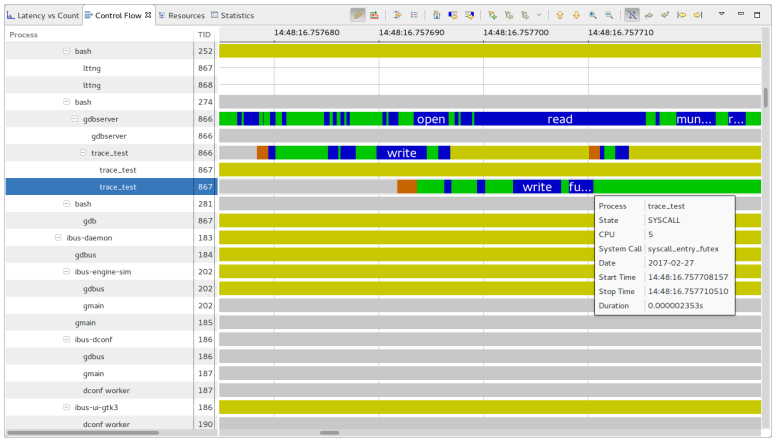

Figure 2.8 Capture d'écran de la vue des ressources de Trace Compass

Le navigateur Internet Chromium offre un outil pour visualiser des traces d'exécution. Il permet de visualiser des traces d'exécution de Chrome et Android. Il supporte aussi les traces de Ftrace,

ce qui permet d'utiliser cet outil pour les traces du noyau Linux (Google, 2016). Plusieurs vues sont offertes, telles qu'une vue représentant le travail effectué pour afficher une page Internet. Ce programme offre aussi des vues de la pile d'appels en fonction du temps. Finalement, l'utilisateur peut le modifier afin d'ouvrir différents types de traces.

Vampir est un ensemble d'outils développés pour l'analyse de performance d'applications parallèles dans le domaine du calcul scientifique. Vampir permet à l'utilisateur d'instrumentation des programmes manuellement, et collecte automatiquement des informations reliées à OpenMP, MPI, glibc, etc (Knüpfer et al., 2008). Vampir permet ensuite de visualiser les traces recueillies avec différentes vues. Ainsi, il y a une vue pour montrer les communications dans un système utilisant MPI, ou le temps d'exécution de chaque fonction. Il existe aussi d'autres vues spécialisées pour les entrées/sorties, l'utilisation de la mémoire, etc.

Paraver est un outil développé par le centre de calcul de haute performance de Barcelone, le « Centro Nacional de Supercomputación » (Barcelona Supercomputing Center, 2017b). Cet outil, développé pour analyser les applications sur des super ordinateurs distribués, offre de nombreuses vues et plusieurs métriques pour bien analyser un programme. Une vue est offerte pour afficher l'état d'une variable, telle qu'un fil d'exécution, un coeur de calcul ou une tâche, en fonction du temps. Paraver peut traiter des applications avec plusieurs niveaux de parallélisme, tels qu'une combinaison de MPI, OpenMP, pthread et CUDA. Il est aussi possible d'avoir des vues des communications MPI pour étudier un super-ordinateur. Paraver peut combiner plusieurs traces, notamment pour trouver les différences entre deux répétitions d'une tâche, afin de trouver les facteurs influençant la performance et le comportement. Finalement, cet outil permet aussi d'effectuer des analyses en se basant sur les traces et d'en obtenir des métriques telles que le nombre d'opération à virgule flottante à chaque milliseconde.

2.6 Conclusion de la revue de littérature

Les systèmes hétérogènes parallèles étant de plus en plus utilisés, il est important d'avoir des outils de développement appropriés. Différentes méthodes de débogage ainsi que leur utilité sur des systèmes parallèles ont été présentées. Les techniques d'instrumentation et de traçage ont aussi été abordées. Il a été constaté que peu de débogueurs permettent de tracer un programme lors d'une session de déboguage. De plus, les débogueurs offrant cette capacité ne sont pas nécessairement optimisés pour les systèmes parallèles. Par ailleurs, plusieurs techniques de déboguage ralentissent fortement un programme, ce qui pose problème avec les programmes parallèles affectés par la synchronisation entre les fils d'exécution. Finalement, il est nécessaire d'offrir des méthodes pour que l'utilisateur puisse faire abstraction de la complexité de ces systèmes afin de les déboguer efficacement.

Ainsi, le débogage de programmes sur des systèmes hétérogènes parallèles est un domaine d'intérêt en raison des défis présents. Ce mémoire présente des contributions visant à améliorer l'efficacité du débogage sur ces systèmes, et donc d'optimiser le développement logiciel.

CHAPITRE 3 ARTICLE 1 : EFFICIENT LARGE SCALE HETEROGENEOUS DEBUGGING USING DYNAMIC TRACING

Authors

Didier Nadeau
Polytechnique Montréal
didier.nadeau@polymtl.ca

Naser Ezzati-Jivan
Polytechnique Montréal
n.ezzati@polymtl.ca

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

3.1 Abstract

Heterogeneous multi-core and many-core processors are increasingly common in personal computers and industrial systems. Efficient software development on these platforms needs suitable debugging tools, beyond traditional interactive debuggers. An alternative, to interactively follow the execution flow of a program, is tracing within the debugging environment, as long as the tracer has a minimal overhead. In this paper, the dynamic tracing infrastructure of GDB was investigated to understand its performance limitations. Thereafter, we propose an improved architecture for dynamic tracing on many-core processors within GDB, and demonstrate its scalability on highly parallel platforms. In addition, the scalability of the thread data collection and presentation component was studied and new views were proposed within the Eclipse Debugging Service Framework and the Trace Compass visualization tool. With these scalability enhancements, debuggers such as GDB can more efficiently help debugging multi-threaded programs on heterogeneous many-core processors composed of multi-core CPUs, and GPUs containing thousands of cores.

3.2 Introduction

Software development is a complicated process that involves many steps such as programming, testing and maintenance, and uses many different tools. One of the most important tools is the debugger, used to detect, locate and correct faults in the code. As such, it is important to have efficient debuggers to facilitate software development and maximize efficiency. Debuggers have evolved in recent years to adequately support multi-threading and non-stop debugging. The software developer can thus select one thread among a few in order to inspect its call stack and local variables. However, as the number of processor cores increases, and with the addition of co-processors with thousands of cores, two significant challenges are brought up: the debugging infrastructure must efficiently cope with hundreds or thousands of parallel threads, and the user interface must properly

help the user to understand the general or detailed behavior of this very large number of threads.

Heterogeneous computers systems with multi-core processors are becoming more popular in many fields. It allows a computer system to contain different processor architectures to benefit from the specific strength of each architecture. Furthermore, multi-core processors allow programs to use parallel computing and improve their performance. However, debugging programs on heterogeneous multi-core systems is an important challenge. The developer can be confused by the large number of threads in the program and have problems keeping track of its execution Olofsson and Khouzam (2014). As the number of threads increases, it becomes harder for the developer to visualize and control the execution flow. Furthermore, the traditional debugging approach, consisting of pausing and stepping a thread, may not work well when the program relies on interactions between threads. Pausing a thread for debugging may cause a deadlock in the system or disturb its behavior Layman et al. (2013). Thus, debuggers for large multi-core systems must allow the user to easily follow and control the various threads execution while disturbing the program flow as little as possible. One interesting solution to this problem is to offer tracing features within debuggers.

There are debuggers that offer specialized features for debugging programs with multiple threads. TotalView and Allinea DDTAntypas (2007) propose different views to group a large number of threads for efficient visualization and control. However, these programs are closed-source and their internal algorithms unpublished. One debugging environment, Eclipse CDT (C/C++ Development Tooling), is open-source and has some views adapted for multi-thread debugging. However, the support is still limitedOlofsson and Khouzam (2014). Tracing support is also limited in debuggers, and the tracing algorithms used may involve an important overhead. To our knowledge, the only open-source debugger that supports tracing is GDB.

A complete debugging environment, adapted for heterogeneous multi-core systems, is needed to maximize developer efficiency. Optimized open-source debugging tools would allow users to not only achieve higher work performance but also to customize these tools for their specific needs and share these innovations. Modern debugging environments should offer optimized methods for grouping, visualizing and controlling a large number of threadsOlofsson and Khouzam (2014). Furthermore, the debugger should be able to look into the program behavior with minimal impact when needed. The aim of this paper is to provide a basis for an enhanced open-source debugging environment for heterogeneous multi-core systems, using GDB and Eclipse CDT as a vehicle for the new proposed algorithms. To this end, we propose contributions to two important aspects of the debugging algorithms and architecture.

One important feature, that large scale debugging tools should provide, is interactive views to help users easily find and follow relevant information. We propose techniques for both automatic and manual filtering, to improve interactive debugging. The first view automatically groups together

threads according to their call stacks, to provide logical groups. The GPU threads have been included into the main debug view, in groups automatically created from their coordinates in the task data grid. Finally, we propose a filter that allows the user to select a group of threads and focus on it by removing the rest of the threads from the debugger interface.

In cases where interactive debugging cannot be used, because it disturbs the system, an alternative option must be available. As the second important contribution, we propose a new technique to enable dynamic tracing with low overhead in GDB for large multi-core systems. Debugging can use tracing to get information on a program's execution, with minimal impact to preserve its normal behavior. The current tracer available in GDB does not scale well on multi-core processors. We propose a tracing architecture that uses components from the LTTng tracer, a kernel and user-space tracer developed by our research group. LTTng was extended to enable dynamic event registration, as it normally requires to define events during compilation. The combination of GDB and LTTng offers a scalable dynamic tracing architecture with correlated user-space and kernel tracing.

In this paper, in the related work section, several available techniques, to debug programs on large scale parallel systems, are discussed. An overview of available solutions for interactive debugging and for using tracing, as well as the limits of each method, are also presented. Then, the proposed contributions to improve interactive debugging in an open-source integrated development environment, and the proposed technique for dynamic tracing used in GDB, are described. Finally, the performance of the various contributions proposed, to enable large scale heterogeneous debugging using open-source tools, is evaluated.

3.3 Related work

There are several debugging tools available for different platforms. TotalView and Allinea DDT are the two graphical debuggers most commonly used for high-performance computer debugging Antypas (2007). They have specialized support for multi-threaded and distributed debugging. These features are relevant in HPC computing such as Cuda, OpenMP and MPI debugging. WinDbg and the Visual Studio Debugger are closed-source debuggers developed by Microsoft and available only for Microsoft Windows. LLDB LLVM Project (2017) is an open-source debugger developed as part of the LLVM project. It does not currently have as many features as GDB.

One common enhancement to facilitate large scale debugging is developing new views to conveniently show the user relevant information. These interfaces aim to show more essential information to the user without overwhelming him. This is extremely important on multi-core heterogeneous systems, since programs usually have very numerous threads to efficiently use their highly parallel architecture. Without specialized support, debugging can become inefficient, as the user will spend

too much time understanding the state of the program. The proprietary debuggers, TotalView and Allinea DDT, both offer views that group threads based on various information such as location in the source code, state, etc Antypas (2007). Eclipse CDT has a multi-core visualizer that shows threads and their state based on their location on a specific processor core Olofsson and Khouzam (2014). However, support for multi-core debugging in Eclipse is still limited, as there are few views designed for multi-threaded programs.

Non-stop debugging is a feature of GDB that was developed for multi-thread debugging. Traditionally, all the threads of a program are either running of stopped. When non-stop debugging is enabled, one or more threads can be stopped while other threads are still running. This allows the user to debug specialized systems where certain threads must continue to work, such as when time constraints must be satisfied or a watchdog timer must be handled Sidwell et al. (2008). Another important feature for debugging is conditional breakpoints. The thread stops when hitting the breakpoint only if a specific condition is met (e.g., a serious erroneous condition was met and the program can be stopped for diagnosing the issue). Otherwise, the debugger simply continues the thread execution and does not notify the user. This reduces the number of notifications received needlessly by the user and improves the debugging efficiency. It becomes even more useful for multi-thread debugging, when the user is more at risk of being overwhelmed by notifications due to the complexity of the program. However, the debugger must still handle the condition evaluation, when a breakpoint is hit, and context switches are involved. This can slow down the program if it is frequently hit.

A program that runs a large scale parallel system is, by design, more complex that a single threaded one, as there must be multiple threads that interact together to benefit from the architecture Spear et al. (2012). This brings problems that are not present in single threaded program, such as a race conditions. Furthermore, it complicates debugging as certain threads might need other threads to work to be able to continue their execution. Stopping the program during interactive debugging may also mask a bug and make it impossible to find it this way. Thus, simply stopping a program and stepping a thread does not always work anymore Shebs (2009).

Tracing is a technique that aims to record a sequence of events that have happened in a program. This technique records and saves each event that occurred, instead of aggregating them and giving a summary, as a code profiler would. Having this full sequence of events can help the developer to investigate in depth an issue, and discover not only the presence of a problem but also its causes The LTTng Project (2017); Vergé et al. (2017). Tracing works by inserting tracepoints in the software code. Tracepoints call the tracer to record data when they are hit during execution. In order to give a realistic view of the program, it must minimize its overhead to limit perturbations Desnoyers and Dagenais (2006). Thus, the goal of a tracer is to have a low impact and achieve good performance

for data collection.

There are two main instrumentation strategies for user-space software tracing. The first one is called static tracing, because it involves inserting code instrumentation into the program source code before compilation. This means that the tracing instrumentation is always present in the source code Hazelwood (2011). However, the instrumentation code is not necessarily active all the time and may simply be disabled during execution. This type of instrumentation is very efficient as the compiler can optimize the calls to the tracing functions. The main limitation of this method for debugging is the recompilation time required each time a tracepoint must be inserted. When the developer needs to insert a new tracepoint while debugging, he has to modify the code and recompile it before continuing, restarting the debugging process.

There are various tools available to statically insert tracepoints into a program. LTTng-UST is the user-space tracer of the *Linux Trace Toolkit : next generation* tool suite. It uses a lockless Read-Copy-Update synchronization to ensure scalability on multi-core systems and optimize data collection Desnoyers and Dagenais (2006). UFtrace is a user-space tracer inspired from the main Linux kernel tracer. It works by compiling a program with option *finstrument-functions* to generate profiling code, and uftrace hooks to this instrumentation to measure metrics such as function duration Kim (2017). Feather-Trace is a tracer with wait-free FIFO for multiple writers Brandenburg and Anderson (2007). There are also programming libraries available to insert logging code for debugging, with various integrated development environments such as Visual Studio. These logging API are not necessarily tracers but their usage is similar for debugging.

On the other hand, dynamic tracing aims to modify a program dynamically. It is called dynamic because instrumentation is added to the program loaded in memory, after compilation, and does not involve source code modification Hazelwood (2011). Bypassing source code modification allows this kind of tracing to quickly instrument a program. However, its performance is not always as good as static tracing. This category of tracing can be subdivided into three types of tracing.

The first type of dynamic tracing works by replacing an instruction by a trap that is caught by the kernel. The kernel then redirects the program flow to the appropriate instrumentation code for tracing. The main drawback of this technique is the cost associated with the interruption handling, which significantly slows down the program being traced Hazelwood (2011). There are multiple available tracers that use this technique. GDB uses this type of tracepoints for its regular tracepoints Stallman et al. (2017). UProbe is a feature of the Linux kernel that provides this kind of tooling and can be used directly, or as a backend for other tracing tools such as Systemtap Corbet (2012). The cost associated with the context switches, from user-space to kernel, limits its use for low impact tracing.

It is also possible to use a virtual machine to interpret the program code at runtime and add instru-

mentation Hazelwood (2011). This method has an important overhead since it adds another step for instruction decoding. However, it can be very flexible and is used in Valgrind, a very popular tracer and profiler Weidendorfer (2008). Valgrind has multiple functionalities such as call graph generation or memory leak detection. This technique is not used for the proposed architecture as its impact is too high.

The third dynamic tracing method replaces an instruction by a jump instruction to the instrumentation code, where the context is saved and the instrumentation is executed. Finally, the original instruction is executed and the program flow returns to the origin of the jump. This instrumentation technique has a very low overhead, similar to static instrumentation, as the cost of a jump is minimal Hazelwood (2011). However, it is limited since not every instruction is large enough to be replaced by a jump. For instance, on the Intel x86 architecture, the instruction must be at least 5 bytes long, or even longer if the jump target is beyond a certain distance. The fast tracing architecture in GDB uses this technique to insert tracepoints. This paper will use this technique as the proposed infrastructure is based on the fast tracepoint architecture in GDB. Furthermore, its cost is low enough to enable efficient tracing in multi-core systems.

Another important factor in tracing is the event collection infrastructure. The available tracers use various techniques to collect the data created by the tracepoints. It can simply involve a lock to access a buffer shared by multiple threads. More sophisticated synchronization methods, such as atomic instructions used by FtraceBird (2009), or lock-less FIFOs by Feather-TraceBrandenburg and Anderson (2007), for collection can offer a significant improvement in performance. However, these techniques are limited to fixed size events. The lock-less Read-Copy-Update synchronization mechanism used by LTTng provides both scalability on multi-core systems and flexible event sizes. Another aspect that impacts performance is trace data transfer. Many tracers save data in a buffer located inside the program memory and must empty them to allow further trace frames to be collected. A dedicated thread or program can be used to do this transfer in the background, or it can be done by a program also doing other tasks. These factors have an important impact on the performance and impact of the tracer.

To efficiently use tracing, the developer must have a way to understand the data. The simplest way to do so is simply to read the trace data using a text interpreter. This method is used by GDB to present the collected trace frames Stallman et al. (2017). Babeltrace is a tool to convert traces generated by LTTng, or other compatible tracers, into text format. However, when the data collected reaches a certain size, it is no longer feasible for a human to read it all. Thus, various tools exist to provide views to display trace data. Vampir is a visualization tool focused on massively parallel computer systems Ezzati-Jivan and Dagenais (2017). It provides multiple views to easily understand data such as a global timeline view, and a view that shows communication statistics.

KCachegrind reads traces generated by the callgrind tool in Valgrind and give a graphical view to facilitate its understanding Weidendorfer (2008). Chrome provides a trace viewer for trace data in its own format or for the ftrace data. Trace Compass is an Eclipse based visualization tool that provides various views such as a call graph or a timeline based view, for trace data generated by LTTng or other compatible tracers Project (2017).

Conventional general-purpose processors found in modern computers frequently have multiple cores that each have their own instruction pointer and data set. This means that each core can execute a thread independently of the others. Processors dedicated for graphics use a different kind of architecture and multiple cores share the same instruction pointer. The execution unit in a GPU is a wave, and consists of a group of data items processed simultaneously on a group of cores with the same instruction pointerHennessy and Patterson (2011). Thus, every core running the wave executes the same instruction concurrently on multiple data items. Due to this highly parallel architecture, graphics processors can achieve much higher calculation performance than general-purpose processors but are greatly impacted by conditional branches or concurrency control.

The HSA foundation is an organization involving academic and industry members that work on heterogeneous systems development. They promote a standard to provide a generic programming model for heterogeneous systems consisting of one or more generic host processors that dispatch work to agentsFoundation (2016b). These agents receive a function called a kernel and a three dimensions grid of data items. Each of these data items has an identifier in its work-group, called the local ID, and an absolute identifier, called the absolute ID. These identifiers are computed from the position of the data item in the grid. A work-group contains adjacent grid items dispatched together to a compute unit. Multiple compute units can be contained in a single agent. The compute unit executes the kernel function on multiple data items at the same time. The group of data items processed simultaneously by a compute unit is called a wave, and multiple waves can be part of the same work-group. A wave is similar to a thread using SIMD instructions, as an instruction is applied at the same time on every data item in the wave. This programming model is applicable for graphics processors, which can be used as agents in a system implementing the standard.

An open-source implementation of the HSA standard is currently being developed by AMD. This framework, called Radeon Open Compute, provides multiple components and allows shared memory space between the host processor and the graphics card. A version of GDB adapted for this software stack is also in development and currently able to interactively debug assembly code running on the graphics card Devices (2016). This debugger still uses a small closed source legacy library for low-level work, that will be replaced to provide a completely open debugger.

A survey of available debugging tools for heterogeneous systems has been presented with a focus

on open-source tools. Various methods and features to aggregate information and help the user to efficiently debug multi-threaded programs have been detailed. Using tracing in a debugger, where the impact on the program must be kept minimal, and different techniques to do so, have been discussed. The general programming model used for graphics processors in the HSA Foundation standard have been outlined, along with an open-source implementation of the standard. In the following sections, the motivation for this work are presented and the current performance and features of GDB and Eclipse are analyzed. Then, the contributions to debugging tools are explained and their impact is analyzed.

3.4 Motivation

Software debugging is a complex process that occurs during development and continues once the software has been deployed. It has been shown that the cost of debugging, testing and verification during software development can reach up to 50-75% of the total development cost in industry Hailpern and Santhanam (2002). This estimation shows how important these steps are, and it demonstrate that efficiency during debugging is vital in order to limit development costs. As the industry has shifted toward improving processor performance through parallelism, programs are increasingly multi-threaded to take advantage of this architecture. Therefore, debuggers must be enhanced for the specific challenges of multi-core processors.

The first issue with multi-core debugging is the large number of threads that need to be displayed and controlled. This can, up to a certain point, be done using specialized views and commands, as available in TotalView, Allinea DDT and Eclipse CDT, etc Olofsson and Khouzam (2014). However, the published algorithms for multi-thread debugging, or open-source reference implementations such as the Eclipse CDT Debugging Services, are limited. More work is needed to offer features that improve developer efficiency in large heterogeneous systems. Furthermore, support for graphics processors debugging is also limited and needs to be improved in order to allow developers to take full advantage of their capacities.

There are also cases where interactively debugging a program changes its behavior or prevents it from working. Traditional debugging, involving breakpoints and manually controlling execution, is not feasible in these circumstances. In these cases, using tracing within debuggers helps the developer to find a problem. However, the tracer overhead has to be sufficiently small to preserve the normal execution flow, and it must be scalable to be usable on multi-core systems.

As heterogeneous multi-core systems become increasingly used, debugging tools must be enhanced for these systems. In order to maximize productivity, tools must follow this trend. Many aspects of debugging need improvements to facilitate this process. In the following sections, we detail our

work on interactive debugging, using Eclipse, and dynamic tracing, using GDB.

3.5 Performance evaluation of current tools

The first step is the evaluation of current open-source debugging tools. The Eclipse CDT support for multi-threaded debugging, and the various views available, have been studied. The current support for fast tracing in GDB has been tested and its performance has been evaluated.

The performance of GDB fast tracing has been measured using benchmarks, and user-space and kernel tracing. GDB fast tracing was used on a multi-threaded test program where the tracepoint is very frequently hit, while changing the number of threads in the program. Three experiments were conducted, to measure the execution time, to capture kernel events, and to record access to the GDB tracing function.

The benchmark executes the test program repeatedly and the tracepoint is frequently hit in the main loop of the program by every thread. The script measures the total execution time as the number of threads is varied. Figure 3.1 shows the results of this test. The total execution time goes from 7.85 seconds for 1 thread to 131.68 seconds for 20 threads. The tracing impact is clear, as the program execution is approximately 17 times faster for 20 threads when there is no tracepoint. Furthermore, the program is significantly faster even for only one thread, going from 5.01 seconds without tracepoint to 7.85 seconds, for a 36 % overhead.

Kernel tracing records kernel events and system calls. It gives the state of each thread during execution and provides a good insight into issues such as resource starvation. Trace Compass, an open-source Eclipse plugin program, is used to visualize the kernel trace collected when GDB is tracing a program. Figure 3.2 represents the test program executing in GDB without tracing. Each line is a different kernel thread, and the one with TID 8677 is the test program executed with only one thread. This thread stays in the same state, executing, for most of its duration, as it simply does floating-point calculations in a loop. Figure 3.3 represents the test program when fast tracing is enabled in GDB. The thread doing the calculation is the one with TID 8088 and its state changes multiple times. It stays most of the time blocked, as shown by its grey color, while thread 8051 of GDBServer is transferring the data. These two figures show the high penalty incurred when the program is stopped to flush the trace buffer, which explains why the execution time is higher, even for only one thread, when GDB tracing is enabled.

The last test to analyze the behavior of the tracer is recording the access to the tracing code in GDB. The collection function code, in the In-Process Agent library, is instrumented using LTTng-UST. This collection function is called whenever a standard GDB fast tracepoint is hit. One event for function entry and another for function exit are defined. The test program is started with 10 threads

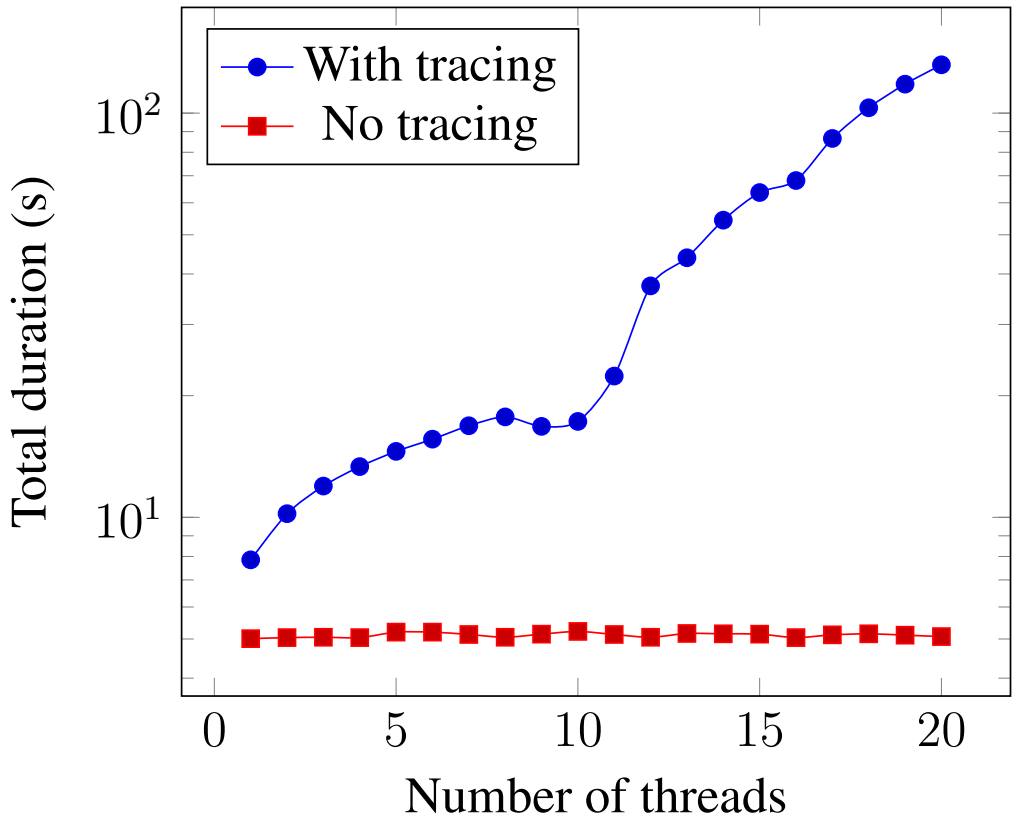

Figure 3.1 Total execution time for a test program as a function of the number of threads when a fast tracepoint is inserted

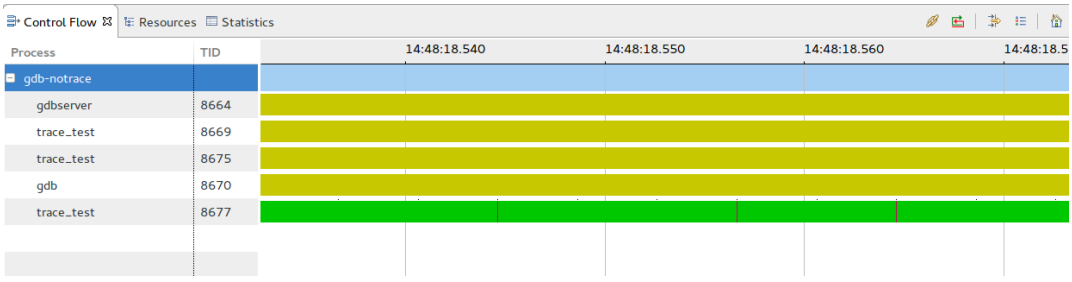

Figure 3.2 Representation of program state during normal execution

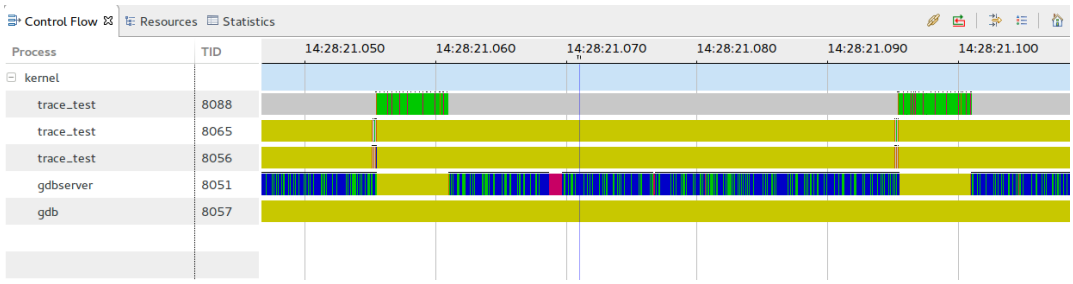

Figure 3.3 Representation of program state while tracing it with the default GDB fast tracing architecture

using GDB, and a fast tracepoint is inserted into its calculation loop. Then, access to the trace collection function in GDB is recorded for each thread using LTTng-UST.

Figure 3.4 shows the trace data in a Trace Compass view. A custom view was defined using the XML plugin. This view shows one line per thread. Each line is either in the *normal* state or in the *tracing* state. The mouse pointer in Figure 3.4 points to a thread in the *normal* state. Underneath the mouse pointer, we can see the notification window shown when the mouse points to a thread in the *normal* state, i.e., outside the tracepoint. The other state, in yellow, indicates that the thread is in the collection function. Figure 3.4 shows that there is only one thread in the *tracing* state at a time. This implies that there is a critical section in the tracepoint and mutual exclusion applies. A review of the jump pad code, that calls the collection function, reveals that there is a spin lock inside the jump pad. Therefore, only one thread can have this lock at a time, which explains why there is never more than one thread in the *tracing* state. Furthermore, we can see that there is almost always one thread that is in the tracing state. This implies that there is at least one thread waiting for the lock most of the time, and that it goes into the collection function as soon as possible. This lock is necessary, as there is a single buffer where every thread stores its trace data. However, it greatly limits scalability and is the most important contributor to the performance hit shown in Figure 3.1 when the number of threads is increased.

The support to debug code running on general purpose graphics processors in GDB is limited. Nvidia offers a debugger based on GDB to debug CUDA code on its graphics processors. This debugger is based on CUDA, which is closed-source and limited to Nvidia graphics processors. AMD is working on a version of GDB to support code running on its graphics processors using standards of the Heterogeneous Systems Architecture foundation. This version of GDB is based on the Radeon Open Compute environment and is called ROCm-GDB. The current version of ROCm-GDB still uses an old closed-source library to support GPU debugging but it will be replaced with an open-source version. Furthermore, the programming model used is an open standard and could be applied to devices from other vendors.

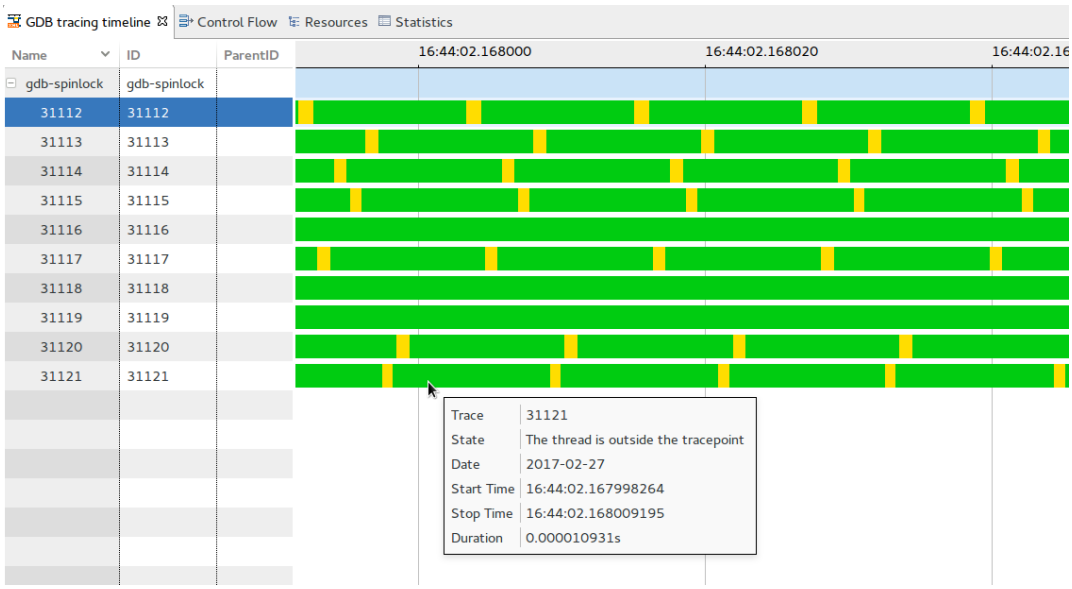

Figure 3.4 Timeline of threads state when a tracepoint has been inserted

Support for debugging code running on heterogeneous and multi-core platforms is limited in open-source development environments. Nvidia offers a version of Eclipse, Nvidia Nsight, to develop and debug code running on its graphics processors using CUDA. Nsight is open-source as it is based on Eclipse but it uses CUDA and has the same limits as CUDA-GDB. Both TotalView and Allinea DDT offer various methods and views to deal with a large number of threads, but both of these programs are closed-source.

Eclipse CDT offers support for multi-threaded debugging. The main debug view shows each thread of the program being debugged in an expandable tree, with the parent processes and the stack of each thread. Selecting a thread in this view allows the user to get more information on its context, such as registers and local variables values. Eclipse CDT also offers support for non-stop debugging, or reverse debugging, when used in combination with a version of GDB that allows it. The multi-core visualizer is a debugging view that shows CPU cores and the threads running on them. It gives an overview of the state of the program to the user, and can display more threads that the main debug view.

The scalability of the main debug view is limited. Indeed, as it represents a list of threads that can be expanded to see their stacks, it can be difficult for the debuggers to control a large number of threads. The stacks must be expanded to know where a thread is stopped, but this mean that there are multiple elements shown for each thread. When there are many threads, the view quickly becomes full, as shown in Figure 3.5. This forces the user to scroll or collapse elements and makes it harder to keep track of the program state.

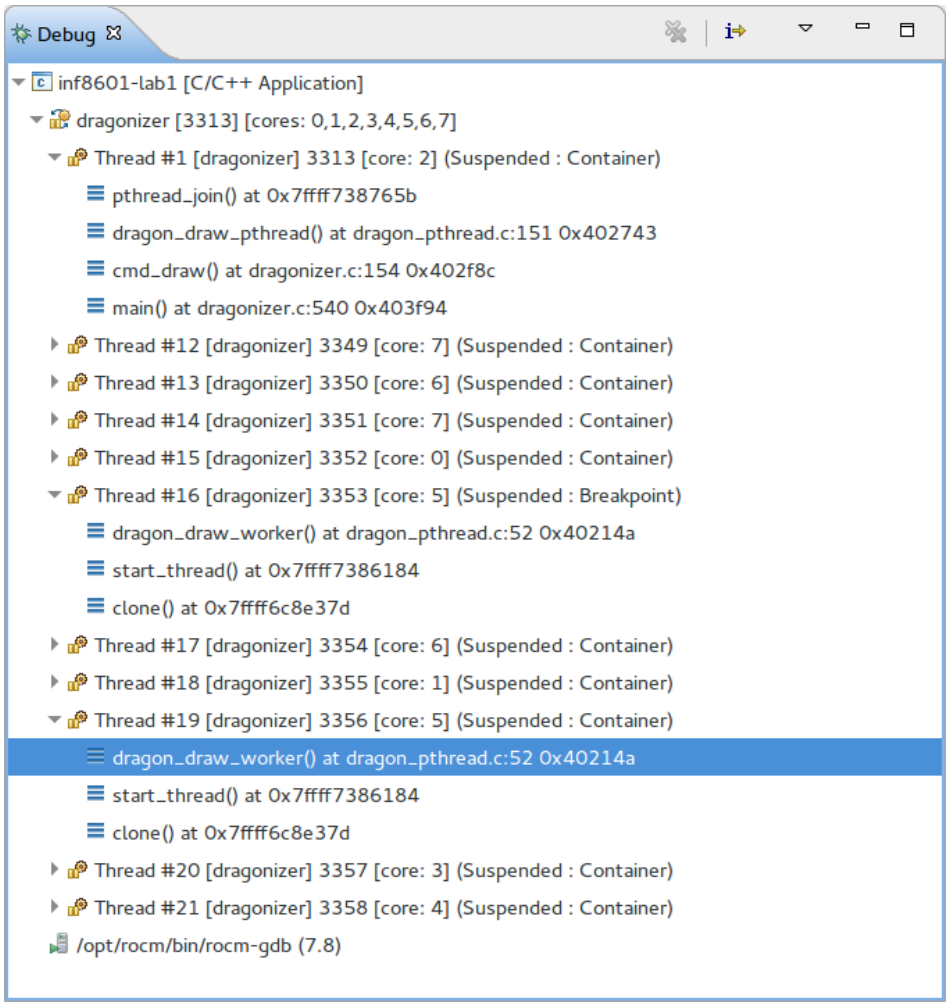

Figure 3.5 Main Eclipse CDT view to shows threads in the Debug perspective

Code executed on the graphics processors using the HSA specifications consists of a function applied to each element in a three dimensions grid. The GPU executes the function in waves, and each wave contains multiple units, that are at the same step in the function but are applied to different elements of the grid. A typical graphics processor can contain significantly more than a hundred waves executing simultaneously. Therefore, these waves can not realistically be displayed in the same way as CPU threads, because it would be inefficient for the user to deal with a list of hundreds of elements.

In conclusion, Eclipse CDT has support to allow the developer to debug multi-threaded programs on a processor. However, it needs significant enhancements and new features to be able to do so efficiently on large multi-core platforms. Support for graphics processors debugging is not present at the moment. Futhermore, the fast tracing architecture used in GDB has issues that strongly limit its performance on multi-threaded programs. In the next section, contributions to both Eclipse CDT and GDB will be proposed in order to optimize debugging on heterogeneous multi-core systems.

3.6 Proposed contributions

3.6.1 Dynamic tracing

Many tracers and loggers work by providing functions called in the source code and require recompilation to be included in the program. On the other hand, dynamic tracing works by modifying the program in memory, after compilation, in order to instrument it. As explained in the related work section, there are various techniques used to achieve this goal. A first technique inserts a breakpoint and the handler redirects the execution flow to the instrumentation code. It is also possible to interpret the code instead of executing it, thus allowing the interpreter to insert instrumentation on the fly. However, these techniques have an important overhead as the first method requires interruption handling by the operating system, while the second one needs to regenerate instructions before executing them Hazelwood (2011). Another option, called fast tracepoints in GDB, involves modyfing the program in memory to jump to the instrumentation code. This method has a significantly lower overhead than the previous techniques Shebs (2009).

In this paper, large-scale is used to describe systems that are strongly parallel and have a large number of cores. There is no exact limitation on what constitutes a large number of cores, but it implies that there are enough cores to bring concurrency challenges. Large multi-core platforms include multi-processor computers or specialized many-core processors such as the Xeon Phi, which can have more than 60 cores. For this paper, the computer used for testing has 4 processors with 16 cores each, for a total of 64 cores. The graphics processor used for GPU debugging is the AMD R9 Nano with 4096 processing units, using the Radeon Open Compute stack 1.1.0.

Previously, tracing only involved GDB, GDBServer and the In-Process Agent library. The user interacted with GDB that sent commands to GDBServer. The In-Process Agent library was preloaded into the debugged program and contained a buffer accessed by the threads, in mutual exclusion enforced by a spinlock. Once the buffer was full, the program was stopped and GDBServer transferred the data into its own buffer before restarting the experiment.

The proposed architecture offers performance improvements by using the LTTng tracer developed by our research group. It has a more sophisticated tracing infrastructure that avoids the penalty caused by locking and stopping. The tracer creates one ring buffer per processor core in a memory space shared between the liblttng-ust library and the consumer daemon. Each buffer contains multiple sub-buffers, and the size and number of these sub-buffers can be modified. Atomic instructions are used by the producer and the reader to keep track of the sub-buffer being currently read and the one being written toDesnoyers and Dagenais (2012). Furthermore, the producer uses local atomic instructions that only affect its core and not the whole processor to minimize overhead. The consumer uses a standard compare-and-swap instruction, although its overhead is limited because it only uses it once a whole sub-buffer has been read. This allows multiple threads to write trace data simultaneously, and does not require to stop the program to empty the buffer.

Furthermore, we have extended the LTTng tracer to enable dynamic tracing. One limitation of the standard LTTng-UST tracer is that events must be defined before compilation. We propose a method to circumvent this limitation, as we need to dynamically register tracepoints. We leverage the CTF trace format used by LTTng that store the trace definition in a single header file. We define multiple tracepoint at runtime that take char array of various size at runtime. When the user inserts a fast tracepoint in GDB, we calculate the total size needed for the desired tracepoint. Then, GDB selects a tracepoint that has a char array of sufficient size and links it by dynamically modifying the program. When the experiment is completed, we modify the trace definition in the header, replacing the char array argument by multiple arguments of the correct type. This method allows us to use the standard LTTng tracer packaged by many Linux distribution. Thus, usage is simplified as a developer can install the standard package from an available repository.

Tracepoint insertion is handled by GDBServer. It first checks if there is a large enough instruction, at the line where the user wants the tracepoint. On x86 processors, the instruction must be at least 5 bytes long. GDBServer replaces this instruction by a jump to a pad where the context is saved and the tracing function of the In-Process agent is called. The original instruction is executed at the end of the jump pad when the tracer returns.

Figure 3.7 shows the workflow for the original tracing technique. Once the thread is in the jump pad, it waits until it can acquire the spinlock. Then, it goes into the tracing function and reserves space in the buffer. If there is not enough space, it triggers a breakpoint to tell GDBServer to empty

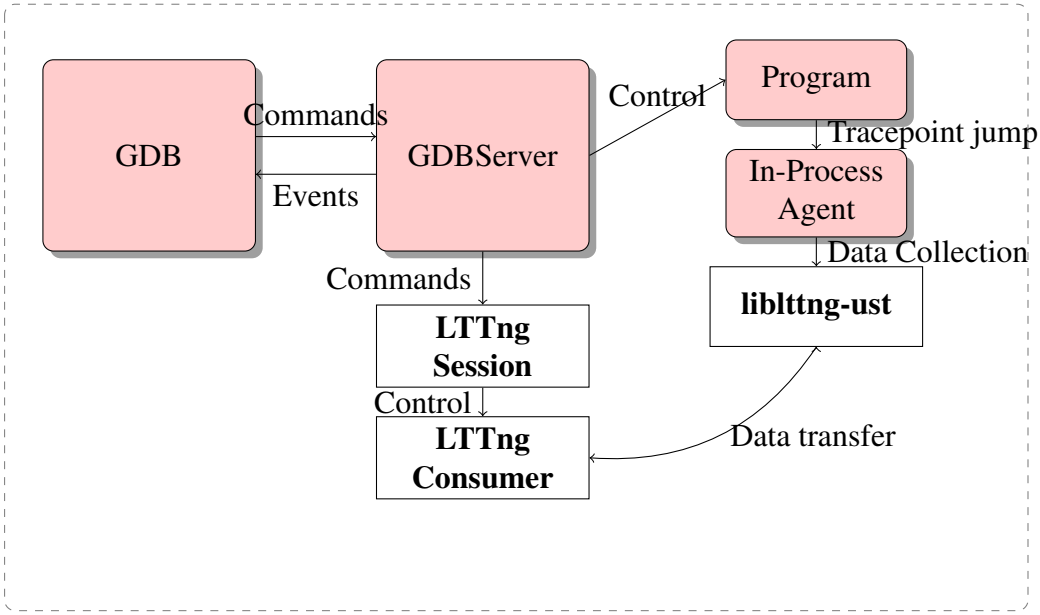

Figure 3.6 Proposed fast tracing architecture using LTTng

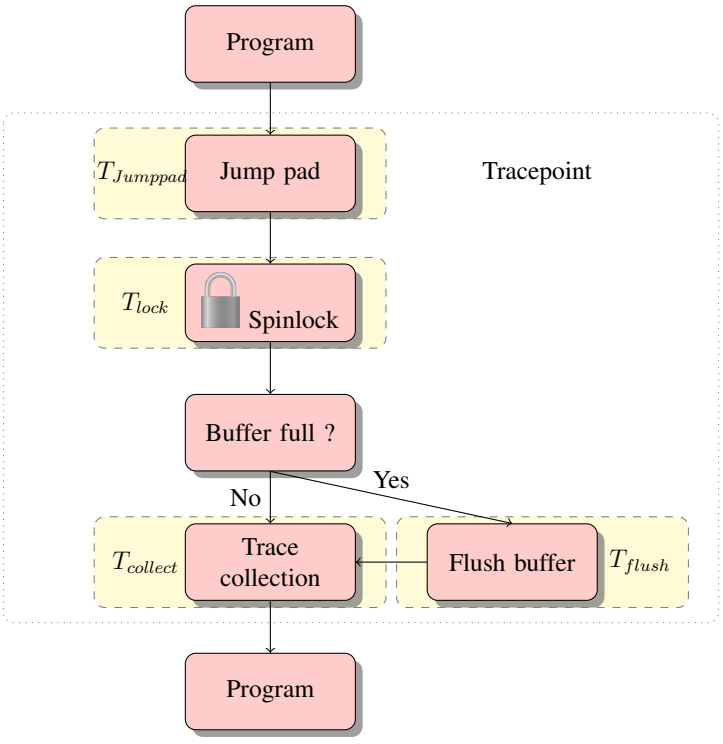

Figure 3.7 Standard fast tracepoint workflow

the buffer. After GDBServer has restarted the program, the thread saves the data, releases the lock and executes the original instruction before returning to the instruction following the inserted jump.

$$T_{tracepoint} = T_{jumppad} + T_{lock} + T_{collection} + T_{flushing} \tag{3.1}$$

The time penalty imposed by the default fast tracepoint implementation is given in equation 3.1. The values of $T_{jumppad}$ and $T_{collection}$ are constants with respect to the number of threads. The value of $T_{jumppad}$ comes from the time it takes to jump to the tracepoint instructions and save the context. The time taken by $T_{collection}$ comes from saving values into the buffer. The component $T_{flushing}$ is affected by the total number of trace frames collected, as the data transfer from the In-Process Agent buffer to the GDBServer buffer occurs only when enough trace frames have been collected for the In-Process Agent buffer to be full. T_{lock} is the component that strongly limits scalability. Indeed, it represents the time spent trying to acquire the spinlock. As the number of threads increases, the tracepoint is hit more frequently and an increasing number of threads try to acquire the lock at the same time. This causes a large penalty, as multiple threads are stuck trying to acquire the spinlock.

Figure 3.8 represents the workflow when a tracepoint is hit in the proposed architecture. First, the context is saved by the jump pad, as for standard fast tracepoints. Then, the data is collected and sent to an appropriate tracing function in the *liblttng-ust* library. Finally, the thread returns to the program.

$$T_{tracepoint} = T_{jumpad} + T_{collection} \tag{3.2}$$

Equation 3.2 gives the time penalty of a fast tracepoint using LTTng. The first component, T_{jumpad}, is the same as in the default implementation. The second component, $T_{collection}$, is extremely similar to the first, as it only uses a different function call to transfer data. It still is constant, unaffected by the number of threads. Both T_{lock} and $T_{flushing}$ are removed for LTTng tracepoints, as there is no lock needed for data flushing.

The proposed implementation brings performance enhancements by using a combination of the GDB dynamic tracing architecture and the LTTng tracer developed by our research group. GDB dynamic tracing enables the developer to insert tracepoints using a simple jump, a few instructions and a call to a function. The cost of this solution is significantly lower than for dynamic tracing using a breakpoint that triggers interruptions. However, the standard GDB fast tracing uses a single lock-protected buffer in the library to save the data. Furthermore, when the buffer is full, the program is stopped by GDBServer to transfer the data and empty the buffer. Both of these fac-

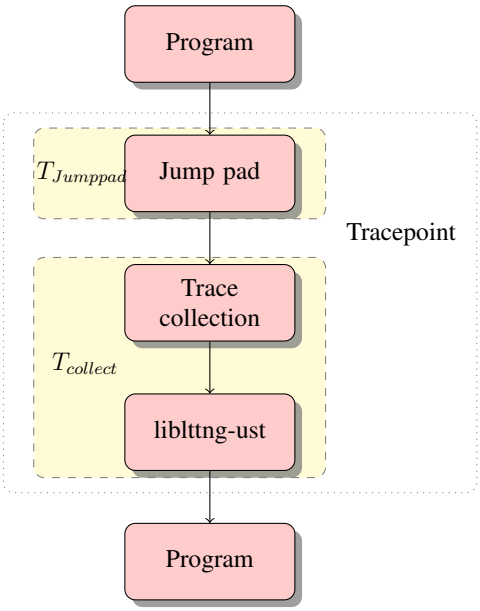

Figure 3.8 Proposed LTTng tracepoint workflow

tors contribute to significantly slow down a program traced on a large-scale parallel system. The proposed implementation enhances the existing fast tracing architecture by removing these two limitations. By using circular buffers in shared memory to store data from tracepoints, no lock is needed to access the buffers and the program is not stopped while the consumer daemon transfers the data. This allows tracing to scale significantly better when the number of threads is increased.

3.6.2 Visual debugging

The first view, implemented to help the user debug a program with a large number of threads, is a call stack aggregation view and is shown in Figure 3.13. This view automatically groups the threads by their call stacks to display important information in a limited space to the user. The call stacks of each thread are retrieved and merged together to create a tree. Therefore, two threads that have the same call stack will be shown in the same leaf of the three. If their call stacks have the same first functions but diverge at some point, this will be shown in the view. This user interface shows groups of thread that are likely to do related work, as they have a similar call stack. It helps the developer to focus on specific parts of the program, as he can collapse parts of the tree, if the threads in this section are doing work he is not interested in. Furthermore, the threads are logically grouped together, which should help the user to better understand the program work flow, as compared to the standard debug view that simply provides a very long, linear, list of threads.

The intended use of this call stack view is multi and many-core systems running programs with a large number of threads. Therefore, we need a scalable algorithm to build a tree structure for

```
function BUILDTREE(CallStacks)
    TreeNode
    i ← 0
    while i < CallStacks.size do
        APPENDTREE(TreeNode, CallStacks[i])
        i ← i + 1
    end while
end function

function APPENDTREE(TreeNode, head)
    if head == null then return
    end if
    child = TreeNode.find(head)
    if child ≠ null then
        APPENDTREE(head.next)
    else
        TreeNode.insert(head)
        APPENDTREE(head.next)
    end if
end function
```

Figure 3.9 Algorithm to build the tree structure for the call stack view.

the viewer from the call stacks. TotalView provides a similar interface, however the algorithms to build the data structure are unpublished. Figure 3.9 shows the algorithm to build the tree. It takes an array of pointers to the head of each thread call stack. The *AppendTree* function is called only once for each stack frame. Furthermore, we have chosen to use an hash table to contain a node children. It ensures *O(1)* complexity to insert or find a children. This gives the algorithm to build the tree a total running time of *O(kn)*, where k is the number of thread and n the average call stack depth. This ensures scalability on multi-threaded programs, as the running time is limited by the total number of stack frames read by the algorithm.

To further improve efficiency of multi-threaded debugging, we propose a filter to be applied on threads. Even with specialized views, overwhelming the developer is still a possibility. The large number of simultaneously active threads increases the frequency of events such as breakpoint hits. Multi-threaded programs running on large systems can contain multiple groups of threads that are doing separate work. Nonetheless, the user could be interested in debugging only one of of these groups. However, some functions can be shared between the groups to reuse code, and inserting a breakpoint can lead to unneeded notifications, complicating the debugging process. While it is possible to use conditional breakpoints in these cases, the user has to manually specify the list of threads concerned for each breakpoint. Using a filter, the user can select the threads

he needs to debug, and the filter removes the other threads from perspective. The developer is not notified of breakpoint hits for threads not included in the perspective, the threads are simply resumed. Furthermore, the filter removes unselected threads from the different debugging views. This enables the developer to focus completely on the problematic parts of the program, without seeing the other threads or receiving unneeded notifications.

Even with specialized views, overwhelming the developer is still a possibility. The large number of simultaneously active threads increases the frequency of events such as breakpoint hits. Multi-threaded programs running on large systems can contain multiple groups of threads that are doing separate work. Nonetheless, the user could be interested in debugging only one of these groups. However, some functions can be shared between the groups, to reuse code and inserting a breakpoint can lead to unneeded notifications, complicating the debugging process. To solve this issue, we proposed a filter concept inspired from conditional breakpoints. These breakpoints are only triggered if a specific condition is met, thus using a filter to reduce the number of notifications. We generalize this concept by filtering the threads themselves. The filtering not only applies to breakpoints but also to views, and threads that are not in focus would not be shown. This enables the developer to focus completely on the problematic parts of the program, without seeing the other threads or receiving notifications.

Displaying GPU waves is a difficult challenge. As previously outlined, a graphics processor can run more than a hundred waves simultaneously, and each of these waves executes a kernel function on multiple data items. The large number of elements to be displayed simply cannot be shown in a list, as it would be almost unusable. Therefore, a method to automatically group the waves and reduce the displayed information must be used. A call stack aggregation view, as proposed for CPU threads, is not possible. Indeed, functions calls are inlined on the GPU and there is no call stack. However, it is possible to take advantage of the programming model used. Each wave process data items contained in a range of the data grid. Furthermore, waves are contained in work-group that have an identifier along the x,y and z axis. We have used these identifiers to show ranges of work-groups in the debug view. The view uses a four level structure, with the first three levels respectively specifying ranges along the x,y, and z axis. The last level contains a list of waves.

For GPU waves, filtering could not be done in the same way, as every wave in a dispatch is executing the same function. Thus, the waves are not in different sections of the code and cannot be filtered as CPU threads. Due to the specific programming model, another type of filtering is possible. Indeed, as a grid of items is sent to the GPU, it is possible to filter the waves according to their position in the grid, and reduce the number of waves and events that the developer has to handle. The first goal of this filtering is simply to reduce information overloading, in case there is a generic problem in a function that occurs everywhere in the grid. If the problem occurs everywhere, the filter could show

only one wave to remove the parallel component from the perspective and facilitate debugging. On the other hand, some problems could occur only in specific sections of the grid, if the function gives an unexpected result for a certain input. In this case, filtering can be used to remove the sections of the grid where there is no problem. Thus, the developer would be able to insert breakpoints anywhere in the function but avoid useless notifications.

3.7 Discussion

In this chapter, we present and evaluate the contributions. In the firs part, the proposed tracing architecture has been tested and its performance measured. Then, the capabilities of the proposed Eclipse views and features are presented and discussed.

The tracing tests were performed on a Linux Fedora 24 computer using Linux kernel version 4.7.9-200.fc24.x86_64. The computer has four Intel Xeon E7-8867 processors at a frequency of 2.50 GHz. Each processor has 16 physical cores with hyperthreading, for a total of 64 physical cores and 128 logical ones. Each processor has a total of 45 MB of cache, and the computer has a total of 256 GB of physical random access memory installed. LTTng version 2.8.0-pre is used for tracing. The work on GDB tracing is based on the GDB 7.11 branch. The work on GPU debugging is based on the ROCm-GDB 1.0 version, on an Intel Core i7-4790 processor, with an AMD R9 Nano card using Eclipse Neon 4.6.

The proposed tracing architecture has been tested with an open-source parallel file compression program, pbzip2. This program is a parallel implementation of the serial file algorithm bzip2 that uses the Burrows-Wheeler algorithm to compress a single file. The benchmarks are done using version 1.1.13 of pbzip2. The program is compiled with debug symbols enabled, 01 optimisation level, and function inlining disabled to facilitate debugging. It is statically linked with the bzip2 library, version 1.0.6 compiled from source. A combination of bash and python scripting is used. There is a total of 15 426 444 tracepoint hits distributed among the different number of threads in each experiment. The same 100 MB randomly generated file is used in each experiment. Figure 3.10 shows the total execution time for the proposed GDB implementation using LTTng tracepoints, the default implementation using fast tracepoints and the baseline using GDB without tracepoints.

As expected, we can see in Figure 3.10 that the proposed implementation using LTTng tracepoints scales well as the tracer architecture is designed for parallel tracing. The default GDB implementation using fast tracepoints has a significantly higher cost than tracing using LTTng. In the case of a single thread, the total time using GDB with LTTng is 32.5 seconds, less than half the time it takes for the default fast tracepoints in GDB at 85.0 seconds. In this case, no time is lost while waiting to acquire the lock, as there is only a single thread. However, there is a penalty associated with the

stops for data transfers while the proposed architecture does not stop the program to transfer data.

When the number of threads is increased, the performance gain of the proposed architecture over the standard fast tracepoints becomes even more noticeable. The LTTng GDB tracing follows the baseline with a small time penalty and it achieves a very similar speedup to the baseline, as seen in Figure 3.11. On the other hand, the performance of the default tracer is slightly better for two threads than one, and starts to decreases for a larger number of threads. For instance, its execution time with 5 threads is longer than for a single thread. Contention for the spinlock, as multiple threads try to acquire it at a high frequency, is the reason for this performance issue on multi-threaded software. The performance improvement of the proposed architecture on large multi-core systems becomes clear, with the results in Figure 3.10.

The default GDB tracing implementation uses the tool Trace Visualizer in the console. This visualizer allows the developer to navigate between captured trace frames and look into the data contained by these frames. A similar interface has been proposed. This interface is implemented within GDB in Python, and uses Babeltrace to read trace data. The commands are very similar to what already exists in GDB. Figure 3.12 shows how this appears on the console. One advantage of using Python to create the commands for trace visualization is the flexibility. Indeed, it is easy for the end-user to modify how the trace data will appear on the terminal and does not necessitate recompilation to work.

Another advantage of the proposed tracer architecture is the ability to enable kernel tracing at the same time as user-space tracing while debugging. Kernel traces can be easily visualized in Trace Compass, as seen in Figure 3.3. This can help to find issues such as resource starvation or mutex contention. Furthermore, interpreting large amounts of trace data can be hard for a developer. The default fast tracer only provides a terminal based interface to read trace data. The proposed implementation also save the trace data to a file where it can be read by Trace Compass. This visualisation tool allows the developer to write XML analysis and easily create views that display the trace data. An example of an XML view in Trace Compass is shown in Figure 3.4.

Figure 3.13 shows the proposed stack aggregation view tested on the same program, used for the main debug view in Figure 3.5. The stack view displays the call stack and every thread in the program using significantly less space than the traditional debug view, where each thread must be expanded to get similar information. Using Figure 3.13, we can easily realize that there are two groups of threads. The first contains only one thread waiting in *pthread_join*, while the other contains nine threads doing work in *dragon_draw_worker*. A quick overview also shows that three threads of the second group are waiting on a barrier. When a program contains more threads, the developer can collapse a section of the tree, if threads are working on some parts of the code he is not interested in.

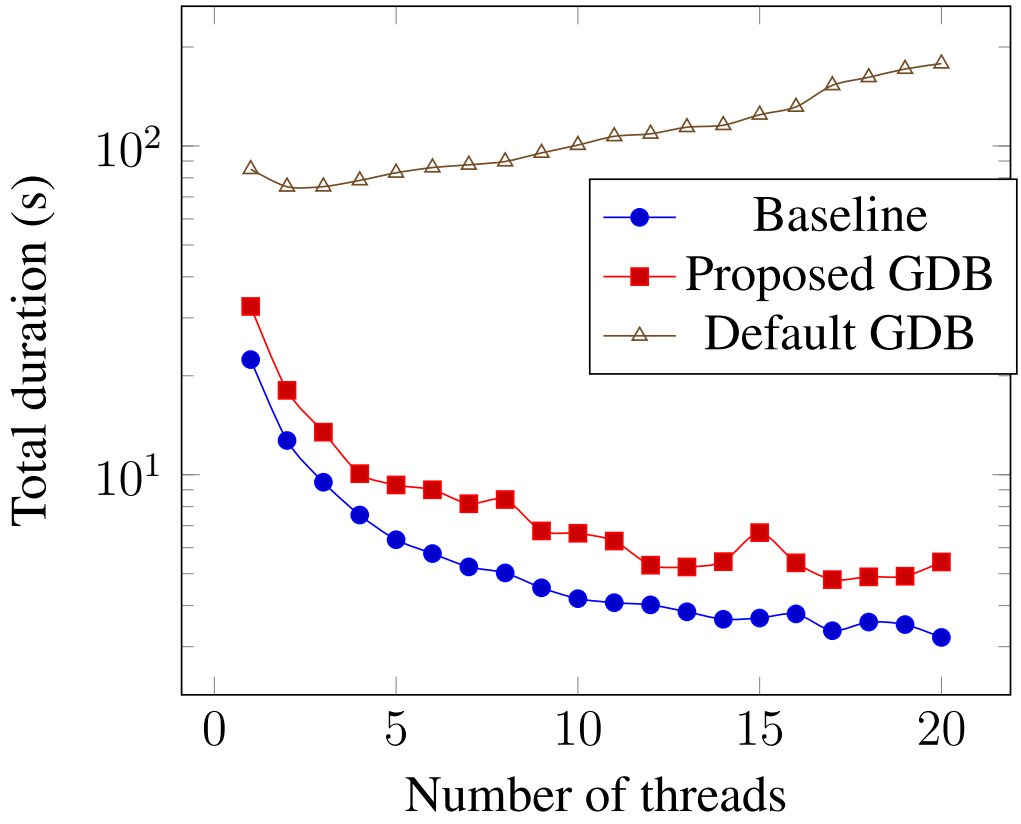

Figure 3.10 Total execution time to compress a 100 MB file using pbzip2

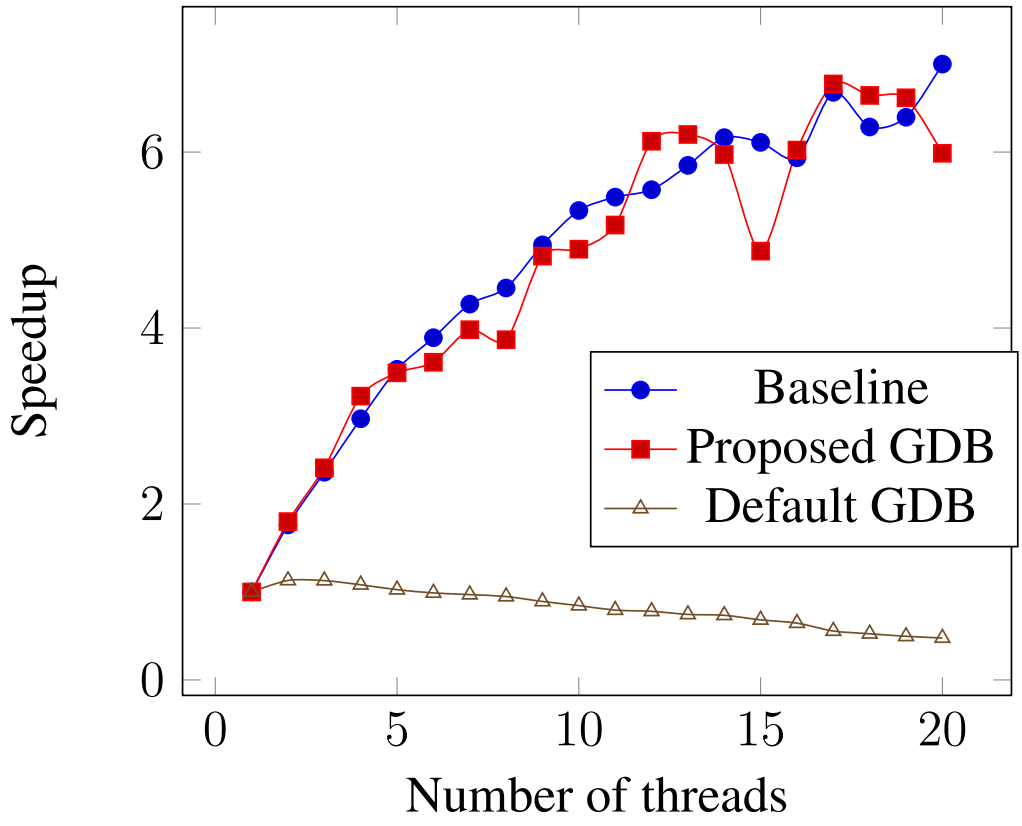

Figure 3.11 Speed up of pbzip2 debugging using GDB standard and proposed tracing

```
(gdb) lttng stop
Stopping tracing
Waiting for data availability.
Tracing stopped for session gdb-tracing
Session gdb-tracing destroyed
/home/didier/lttng-traces/gdb-tracing-20170228-114431
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 0:
x = 0.5
y = 0.0
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 1:
x = 1.0
y = 996.2153228505617
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 2:
x = 1.5
y = 996.7594107814588
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 3:
x = 2.0
y = 997.1210833682936
(gdb)
```

Figure 3.12 Trace data visualization with the proposed GDB fast tracing architecture

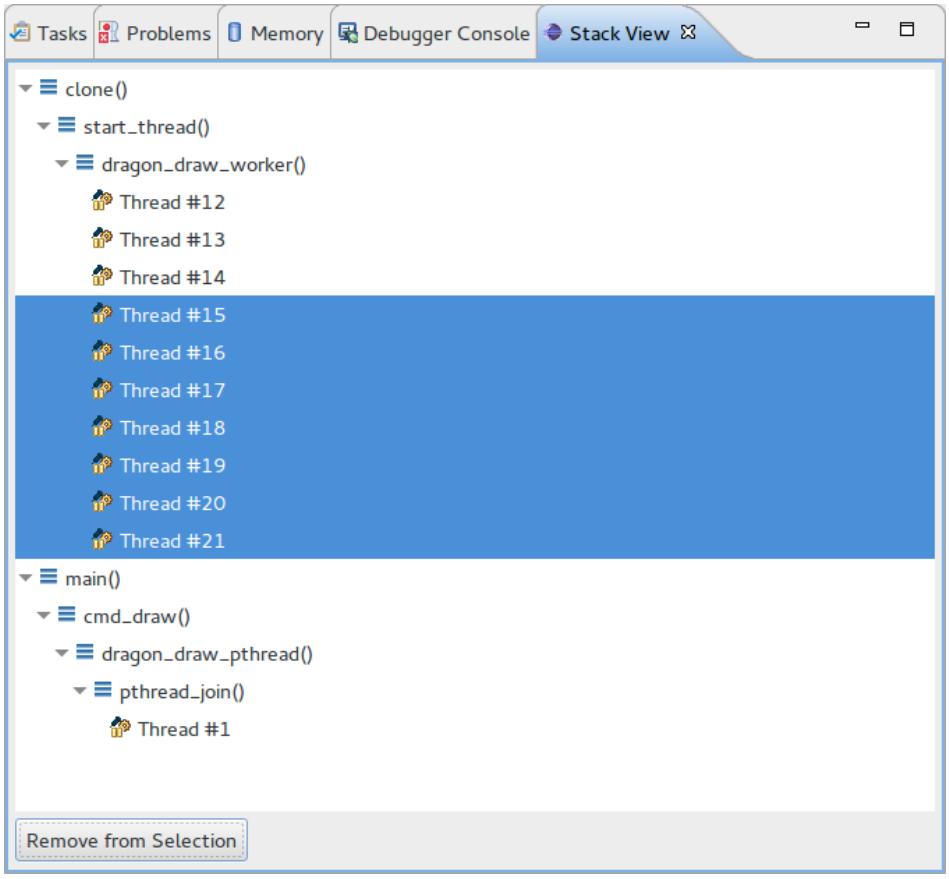

Figure 3.13 Stack Aggregation view implemented within Eclipse CDT to display active threads

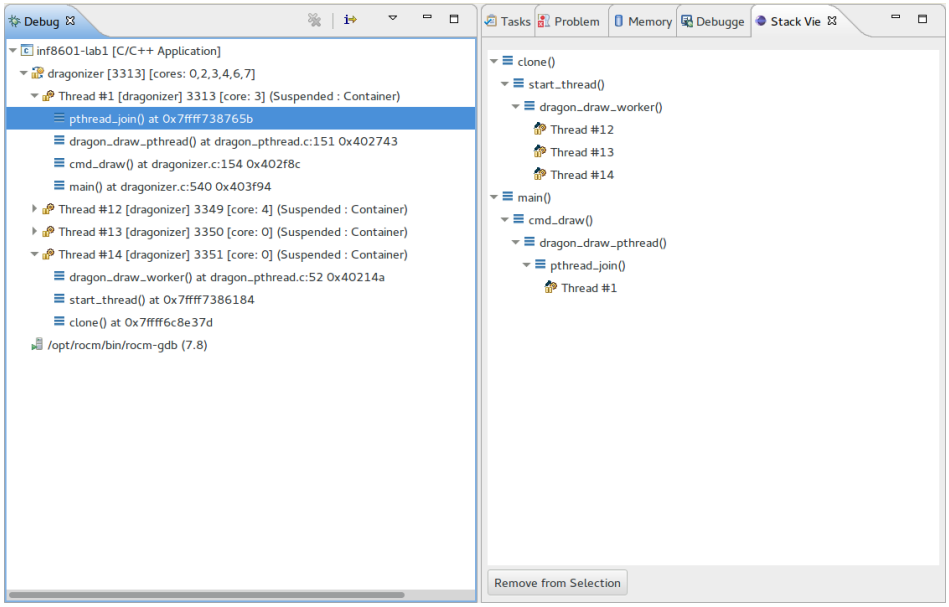

Figure 3.14 Debug view and Stack Aggregation view when the filter has been applied.

The proposed thread filter has been implemented within GDB. We have decided to use the stack aggregation view to set the filter for the threads, as it already shows logical groups of threads. To enable filtering, the user simply selects the threads he wants to remove, as shown in Figure 3.13 and clicks the *Remove from Selection* button. Eclipse CDT sends the information to GDB, and the filter is created. The selected threads are not shown the next time the program stops. As we can see in Figure 3.14, the threads selected in Figure 3.13 are not present because the filter has been applied. Furthermore, if one of the selected thread hits a breakpoint, GDB simply continues and does not notify the user. Filtering has been implemented within GDB to maximize performance. It avoids communication between Eclipse CDT and GDB to decide if a thread is in the selection, and accelerates breakpoint handling. Furthermore, it reduces the size and number of messages sent, as it does not need to send information related to threads excluded by the filter.

The GPU waves have been included in the main debug view using work-groups, as shown in Figure 3.15. There are four levels in the tree, one for each dimension of the grid, and the final one to display the waves in a work-group. With this view, the developer is able to have a better overview of the program than a long linear list of all the waves. It gives the necessary information to the developer, and allows him to select a specific rang in the data grid to inspect.

The GPU waves filter has also been implemented within GDB. A simple Eclipse view allows the developer to specify the range of position he wants to set the focus on. The user can select a combination of ranges along one or more axis. An example is shows in Figure 3.16. In this case, the ranges selected are 0 to 63 for the X axis, 0 to 47 for the Y axis and 0 to 1 for the Z axis. Then, the debug view, that previously displayed every GPU waves as shown in Figure 3.15, only display waves that are in the focus region. Figure 3.17 shows that the only waves displayed are the one that are inside the focus set in Figure 3.16. The reduced quantity of information helps the user focus on the problems. The user can also reduce the range of the filter as he narrows down on the origin of the problem. This helps the user stay focused on the problem. Furthermore, it allows him to abstract the parallel aspect, by choosing a specific wave of focus, on a problematic region of the grid.

In summary, we have demonstrated the advantages of the proposed architecture over the default tracer. It allows significant performance gains for single-threaded software and even more important gains on multi-core systems. The terminal reader for the traces is easily customizable by the user, without recompilation. Furthermore, it allows the developer to combine kernel and user-space tracing easily, and Trace Compass can be used to obtain helpful data visualization.

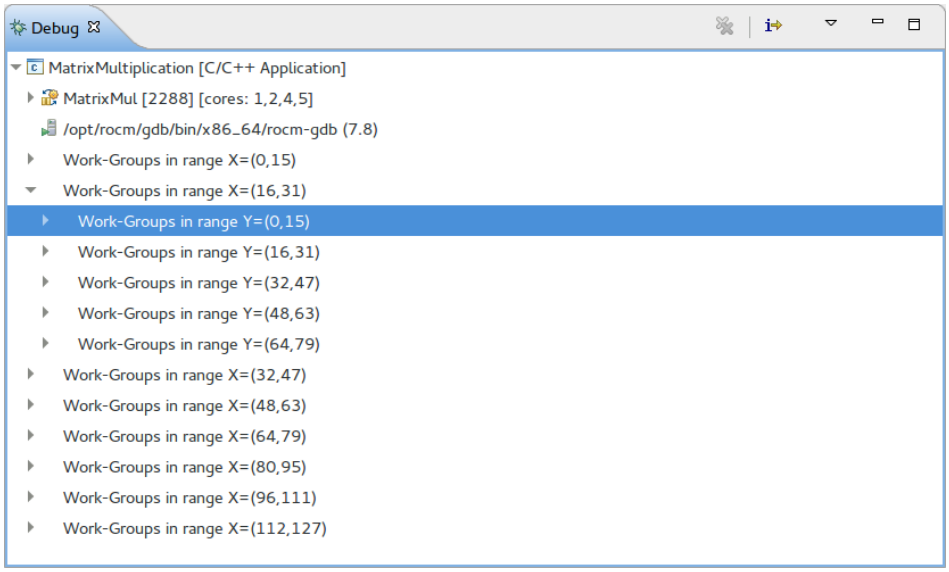

Figure 3.15 Eclipse CDT main debug view with GPU waves shown

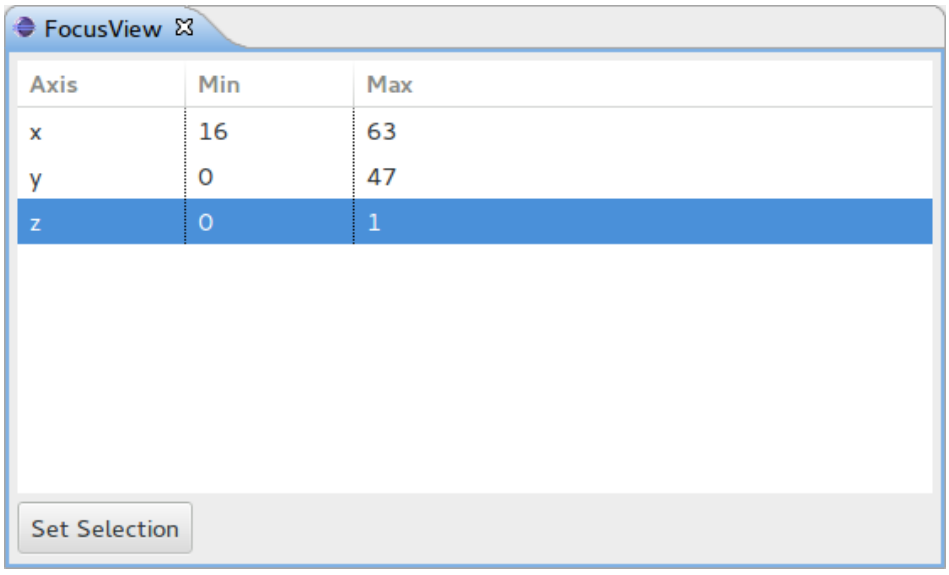

Figure 3.16 GPU focus view to filter HSA waves

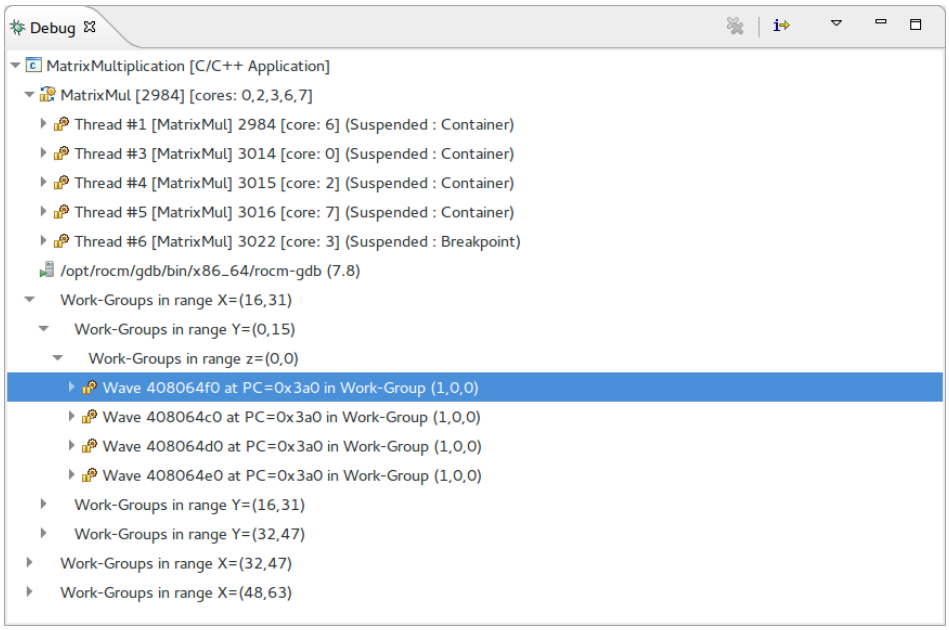

Figure 3.17 Eclipse CDT main debug view with GPU waves and an active GPU focus

3.8 Conclusion

In this paper, we have shown the limitations of current debuggers for large scale parallel systems. We have presented various methods currently available in both open and closed source software. We have analyzed the performance of the tracer available with GDB and the features offered in Eclipse CDT for multi-thread debugging.

An improved technique for dynamic tracing in GDB was proposed based on the results of the analysis. This architecture uses both GDB and LTTng-UST to ensure scalability of the dynamic user-space tracer. The performance of the proposed implementation has been compared to the performance of the default GDB fast tracer. The test results have shown that fast tracepoints using LTTng scale well, while fast tracepoints using the standard GDB implementation are greatly penalized by the lock protecting buffer access and data transfer. Tracing performance quickly deteriorates when multiple threads hit a breakpoint frequently, while the LTTng tracepoints performance is not affected by multi-threading.

A view to group threads by their call stack has been presented and evaluated. The algorithm to build this view has been developed to ensure scalability on large multi-threaded programs. The main Eclipse debug view has been enhanced to include GPU waves for heterogeneous debugging. A filter was developped to exclude irrelevant waves and threads from the debugging views to reduce unneeded information. These contributions help the developer by providing a better view of the debugging context to focus on the problems to solve.

These contributions are a first step to adapt debugging tools for modern large scale heterogeneous systems. The tracing technique efficiently provides data to help the user. However, it still requires an instruction large enough to replace it by a jump. This limitation could be removed by displacing a function frame and instrumenting it. The proposed views and the filter simplify and enhance the user experience. The handling of excluded threads works in the same way as for conditional breakpoints, and could be improved as it involves costly context switches that carry an important overhead.

3.9 Acknowledgement

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, and EfficiOS for funding this project.

CHAPITRE 4 RÉSULTATS COMPLÉMENTAIRES

4.1 Multicore Visualizer pour les cartes graphiques

Lors de la recherche de techniques pour afficher efficacement les vagues, ou fils d'exécutions, d'un agent HSA, diverses stratégies ont été envisagées. En effet, tel que discuté précédemment, une carte graphique peut supporter un très grand nombre de vagues s'exécutant simultanément. Cela pose un problème pour utiliser une interface graphique, puisque ceux-ci affichent traditionnellement les vagues dans une simple liste. Or, cela n'est pas approprié lorsqu'il y a plusieurs centaines d'éléments, puisque cela peut rendre l'utilisation du débogueur pénible. Il est donc nécessaire d'utiliser d'autres techniques pour afficher les vagues.

Le « Multicore Visualizer », tel que résumé au chapitre 2, est une vue d'Eclipse CDT qui a été développée lors d'une collaboration entre Ericsson et Adapteva. Il s'agit d'une vue conçue pour donner facilement une vision d'ensemble d'un système largement multi-coeur à un utilisateur. Cette vue offre un survol rapide de l'état global du système, mais fournit moins de détails qu'une vue des fils d'exécution en arbre présentée au chapitre 3, et affichée à la Figure 3.13.

Le processeur développé par Adapteva, l'Epiphany, est un processeur largement parallèle pouvant contenir jusqu'à 1024 coeurs sur un circuit. L'utilisation d'outils de débogage sur ce processeur peut donc mener à des problèmes similaires à ceux retrouvés sur une carte graphique. Le « Multicore Visualizer » a donc été adapté pour afficher les fils d'exécution sur une carte graphique.

Le « Multicore Visualizer » est implémenté avec SWT, le « Standard Widget Toolkit ». La gestion des éléments dans la vue est effectuée avec peu d'abstractions. En effet, le code calcule manuellement la taille de chaque élément, leur couleur, etc. Cela fait en sorte qu'il peut être difficile de l'adapter à une autre architecture. Ainsi, les coeurs des cartes graphiques sont regroupés dans une hiérarchie de groupes logiques et matériels. Un coeur de calcul fait partie d'une unité SIMD, qui elle-même fait partie d'un « Compute Unit » avec 3 autres SIMD. Les « Compute Unit » sont eux-même regroupés dans des « Streaming Elements », qui représentent le groupe de plus haut niveau dans le processeur. Comme ces différents groupes affectent le fonctionnement du processeur, par exemple pour le partage de mémoire, il est intéressant de le représenter dans le « Multicore Visualizer ». Or, il est difficile d'adapter le code actuel afin de montrer cette hiérarchie.

La vue a donc été modifiée pour utiliser la dernière version du « Graphical Eclipse Framework », ou GEF. Ce cadre d'application est conçu pour permettre de créer facilement des graphiques. Il est possible de spécifier la structure du graphique avec les noeuds adéquats pour représenter le processeur, et GEF gère les dimensions et le placement des noeuds. Cela permet donc d'intégrer

facilement une nouvelle architecture au « Multicore Visualizer ».

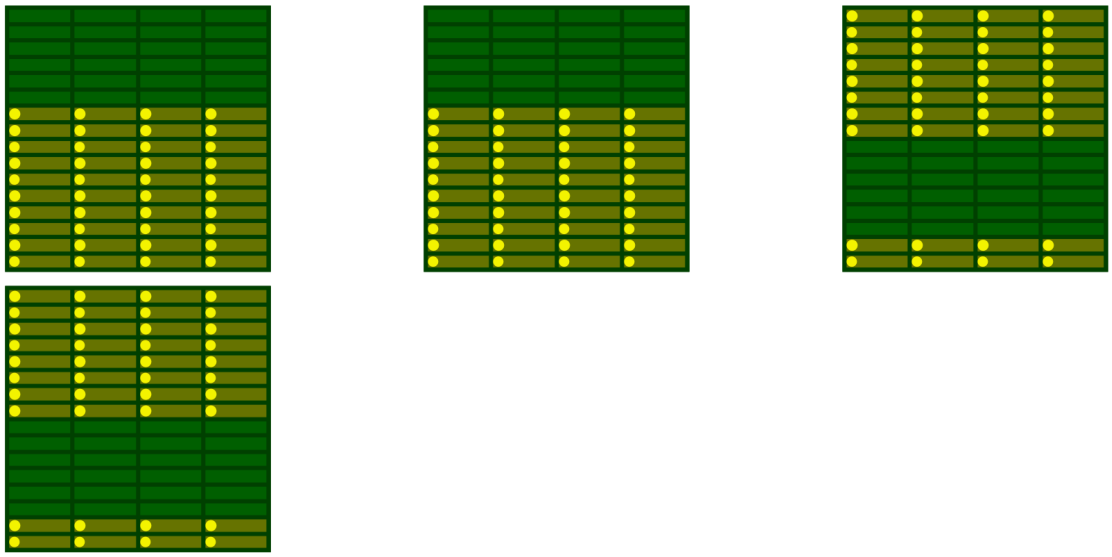

Figure 4.1 Vue du « Multicore Visualizer » montrant les vagues s'exécutant sur une carte graphique.

Une première version du « Multicore Visualizer » pour les processeurs graphiques a été développée avec le GEF. Cette version montre la structure des coeurs ainsi que les tâches arrêtées sur les coeurs où elles s'exécutaient en dernier. On peut voir cette vue à la Figure 4.1, représentant une session de débogage sur une carte AMD Radeon R9 Nano. Cette vue est une première version montrant l'architecture et un cercle pour chaque vague d'exécution. On voit que les vagues sont en jaune pour démontrer qu'elles sont arrêtées. Cette version a été présentée aux développeurs travaillant sur les outils pour le « Heterogeneous System Architecture » chez AMD.

Cependant, quelques problèmes avec cette vue ont été relevés lors de la présentation. Premièrement, les développeurs de AMD ont noté que, même avec cette vue, il peut être difficile voir et distinguer les différentes vagues. En effet, chaque unité SIMD, qui correspond aux plus petits rectangles à la Figure 4.1, peut supporter jusqu'à 10 vagues. Ils étaient d'avis que la vue ne serait plus alors très pertinente puisqu'il y aurait trop d'information. De plus, ils ont aussi noté que les vagues ont beaucoup moins d'affinité à un coeur que sur un processeur conventionnel. Étant donné qu'un processeur graphique possède proportionnellement moins de cache qu'un processeur conventionnel, il y a peu de chance qu'une vague recommence à s'exécuter sur la même unité SIMD après la préemption. En effet, lorsque la vague s'arrête durant le débogage, l'unité SIMD risque fortement d'enlever toutes les données associées de sa cache puisqu'il n'en a pas beaucoup. Lorsque la vague est repartie, elle n'a plus de préférence pour l'unité où elle s'exécutait, puisque celle-ci ne possède plus ses données. Il n'y a donc pas d'avantage à garder la vague sur la même unité SIMD. Dans ce cas, une vue telle que le « Multicore Visualizer » a beaucoup moins d'intérêt. En effet, il

est moins intéressant de représenter des vagues sur des coeurs physiques si elles n'arrêtent pas de changer d'emplacement. Suite à cette rencontre, le travail sur le « Multicore Visualizer » pour carte graphique a été abandonné.

4.2 Points d'arrêt conditionnels

Les points d'arrêt conditionnels peuvent fortement aider un utilisateur à déboguer un programme. En effet, il peut arriver qu'un problème survienne seulement lorsque certaines valeurs sont dans un intervalle précis. Par exemple, une fonction pourrait exhiber un comportement non désiré si des arguments ont des valeurs spécifiques. Grâce aux points d'arrêt conditionnels, on peut indiquer au débogueur qu'il doit arrêter le programme seulement lorsqu'un problème va survenir. Si on utilise un point d'arrêt normal, le débogueur va toujours stopper le programme à cet endroit, et on risque d'avoir un grand nombre d'arrêts inutile. Cela ralentit fortement le processus de débogage, et peut faire en sorte que le débogage échoue. En effet, l'utilisateur peut manquer l'arrêt pertinent à travers tous les autres arrêts inutiles, ou simplement décider que cela prend trop de temps et utiliser une autre technique. Il est donc évident que le point d'arrêt conventionnel est intéressant dans ce genre de situation.

Cependant, les points d'arrêt conditionnels sont généralement implémentés de la même manière qu'un point normal. Tel que décrit au chapire 2, une instruction est insérée dans le programme pour l'arrêter à un endroit. Lorsque cette instruction est rencontrée, le programme stoppe et le débogueur prend le contrôle. Celui-ci va alors inspecter les valeurs en mémoire, et va vérifier si la condition est respectée, basée sur ces valeurs. Si c'est le cas, le débogueur avertit l'utilisateur que le programme est arrêté et lui donne le contrôle. Autrement, le débogueur va simplement repartir le programme sans avertir l'utilisateur. Du point de vue de celui-ci, il ne se passe rien jusqu'à ce que la condition soit remplie.

Cependant, la situation est différente au niveau de la performance. En effet, l'impact d'un point d'arrêt conditionnel est non négligeable sur la performance d'un programme. Chaque fois que le point d'arrêt conditionnel est frappé, une interruption est déclenchée et doit être traitée par le système d'exploitation avant que le débogueur n'ait le contrôle. Lorsque le point d'arrêt est appelé fréquemment, le programme peut être fortement ralenti. La situation peut être encore pire pour un programme parallèle, puisque plusieurs fils d'exécution peuvent frapper le point d'arrêt alors que le débogueur peut n'avoir qu'un seul fil pour traiter ces évènements. Ainsi, lorsque la fréquence à laquelle le point d'arrêt est frappé est assez élevée, le débogueur peut avoir l'effet d'un goulot d'étranglement.

Une technique utilisée pour accélérer le traçage dynamique est d'insérer le code d'instrumentation

dans la mémoire du processus tracé, puis de remplacer une instruction par un saut jusqu'à cet endroit. Cela permet de réduire le ralentissement par rapport à du traçage avec point d'arrêt, tel que Uprobes le permet. Ainsi, le coût de l'interruption est remplacé par celui d'un appel de fonction, qui prend beaucoup moins de temps. Par ailleurs, la librairie « In-Process Agent » de GDB, utilisée pour la traçage dynamique rapide, permet d'évaluer une condition avant de capturer un évènement. Les points d'arrêt conditionnels étaient notamment cités comme un point pouvant être amélioré lors du développement du « In-Process Agent ».

La performance des points d'arrêt conditionnels normaux a été comparée à la performance de points d'arrêt conditionnels rapides implémentés avec la librairie IPA. Les résultats sont affichés au Tableau 4.1. Ces résultats découlent d'un test où la condition des points d'arrêts est évaluée extrêmement fréquemment. La condition n'est jamais remplie, donc le programme ne reste jamais arrêté. La durée totale d'exécution du programme est mesurée sans point d'arrêt, avec un point d'arrêt conditionel rapide, et avec un point d'arrêt conditionel normal. Le point d'arrêt conditionnel rapide allonge la durée d'exécution totale de près de 41%, un impact important en raison de la haute fréquence avec laquelle il est frappé. Cependant, cet impact est significativement plus petit que pour les points d'arrêts conventionnels. En effet, le fonctionnement du programme avec un point d'arrêt conditionnel normal prend près de 127 secondes. Cela est près de 500 fois plus lent que le programme sans point d'arrêt ou avec un point d'arrêt rapide. Cette situation est similaire à celle trouvée au chapitre 3 pour les points de traces, tel que montré à la Figure 3.10. Cela est attendu, étant donné que dans ces deux cas on compare une implémentation utilisant une instruction d'interruption à une implémentation utilisant un saut inséré dynamiquement.

Tableau 4.1 Durée d'exécution totale moyenne d'un programme de test lorsque la condition d'un point d'arrêt est évaluée 1 000 000 fois, et marge d'erreur associée avec un intervalle de confiance de 95%

Type	Durée totale (s)	Marge d'erreur (s)	Taille de l'échantillon
Sans point d'arrêt	0.24	0.02	20
Rapides	0.34	0.03	20
Conventionnels	127.9	0.4	20

L'implémentation utilisée pour les points d'arrêt conditionnels rapides contient le strict minimum. Lorsque le point d'arrêt est frappé, la condition est évaluée et le programme est arrêté si elle est remplie. Cependant, il reste à implémenter la logique pour gérer le programme si il s'arrête. En effet, il faut que l'utilisation de l'IPA soit invisible pour l'utilisateur. Or, le programme ne s'arrête pas à l'endroit où l'usager a demandé d'insérer le point d'arrêt. De plus, il faut restaurer le contexte, comme les valeurs de variables, registres et la pile d'appel, au moment où le programme est entré dans l'IPA. Sans cela, l'utilisateur va observer que le programme est arrêté à un endroit inattendu,

et les valeurs des registres ne seront pas pertinentes.

On constate donc que la performance des points d'arrêt conditionnels rapides est largement supérieure à celle des points d'arrêts conditionnels normaux. L'utilisation de ce type de point d'arrêt pourrait permettre d'améliorer l'expérience de débogage, et convaincre plus d'usagers de profiter de points d'arrêt conditionnels. Tel que mentionné, il reste à implémenter la gestion des arrêts pour que cet outil soit disponible.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Retour sur les résultats

5.1.1 GDB

Traçage

Tel que décrit au chapire 3, l'étude du traceur intégré dans GDB a révélé deux problèmes majeurs. On a constaté que cet outil fait usage d'un verrou pour contrôler l'enregistrement, et stoppe le programme pour transférer les données. Cela diminue sévèrement les performances d'un programme parallèle débogué, et peut modifier son comportement. GDB a été adapté afin d'utiliser le traceur LTTng-UST pour effectuer le traçage, ce qui a permis de fortement améliorer sa performance. De plus, le traceur LTTng a été amélioré puisqu'il ne supporte que les évènements définis lors de la compilation. Les modifications apportés permettent à l'utilisateur de définir dynamiquement un évènement avec GDB.

Cette amélioration permet de tracer efficacement des programmes parallèles, tel que montré à la Figure 3.10. Cette diminution du ralentissement permet d'utiliser le traçage dans un plus grand nombre de cas, et diminue le risque de perturber le système.

Points d'arrêt conditionnels

Les points d'arrêts conditionnels disponibles avec GDB ont aussi été étudiés, tel que décrit au Chapitre 3. Il a été montré que ceux-ci, en raison de leur implémentation avec des instructions d'arrêt, ralentissent fortement les programmes. Cela peut rendre une session de déboguage très lente, et est problématique pour les programmes parallèles où plusieurs fils peuvent frapper un point d'arrêt.

L'alternative présentée utilise la librairie « In-Process Agent » afin d'évaluer les conditions avant de déclancher une interruption. Le gain en performance à été montré au Tableau 4.1. Cette technique permet d'accélérer le déboguage, et pourrait inciter les utilisateurs à utiliser plus souvent cet outil. En effet, il serait possible pour l'usager d'insérer plusieurs points d'arrêt conditionnels afin de trouver un problème aisément. Cela permettrait de déboguer plus efficacement les programmes parallèles, et donc de diminuer le temps de développement.

5.1.2 Interfaces graphiques

Agrégation de piles d'appel

La vue d'aggrégation de la pile d'appel, présentée au Chapitre 3, groupe automatiquement les fils d'exécution par leur pile d'appel. Cela fait en sorte que cette vue est peuplée de plusieurs groupes logiques. En effet, tel qu'on peut le voir à la Figure 3.13, les fils regroupés ensemble exécutent des tâches similaires. Tel que discuté au Chapitre 3, cela permet de donner aisément l'information pertinente à l'utilisateur tout en réduisant la quantité d'information affichée. En effet, celui-ci n'a plus à agrandir chaque fil afin de savoir où il se trouve, et il peut facilement reconnaître les groupes de fils exécutant le même travail. Cette vue permettrait donc d'améliorer l'efficacité du déboguage d'un programme avec un grand nombre de fils d'exécution.

Affichage de vagues d'exécution

L'affichage et le contrôle efficace de fils d'exécution sur les cartes graphiques étaient un autre aspect du travail effectué durant la maîtrise. Une carte graphique peut supporter un très grand nombre de fils d'exécution simultanément, tel qu'expliqué au Chapitre 2. De plus, selon la nomenclature utilisée par la compagnie AMD pour son environnement libre « Radeon Open Compute », les fils d'exécutions sur GPU sont appelés des « waves », ou vagues d'exécution.

Le Chapitre 3 présente les travaux effectués pour afficher ces vagues. Tel qu'expliqué, il n'est pas possible d'utiliser la pile d'appel pour regrouper les vagues d'exécution, puisque celles-ci n'ont pas de pile d'appel. La Figure 3.15 montre la méthode utilisée pour afficher les vagues. Leurs position dans la grille de travail est utilisée afin de créer des groupes de vagues d'exécution. Tel que discuté au Chapitre 3, cette technique permet de limiter la quantité d'information affichée par défaut dans la vue de déboguage. De plus, il s'agit d'une manière logique de grouper les vagues, puisque certains problèmes peuvent être localisés dans la grille de donnée.

Utilisation de filtres

La dernière contribution présentée dans ce mémoire sont les filtres pour les fils et vagues d'exécution. Ces filtres ont été présentés au Chapitre 3, et sont applicables aux processeurs centraux ainsi qu'aux cartes graphiques. La Figure 3.14 montre l'effet lorsqu'un fil est appliqué à une application. Ce filtre permet à l'utilisateur d'enlever de l'espace de travail des informations qu'il juge non pertinentes. En réduisant le nombre de fils affichés ainsi que le nombre d'évènements reçus par l'utilisateur, on lui permet de mieux se concentrer sur le problème. Cela permet de faire abstraction de la complexité d'un système afin d'être plus productif.

5.2 Limitations

Traçage et débogage conditionnel

Une limitation importante du traçage dynamique rapide est qu'il y a une taille minimale pour remplacer une instruction. En effet, sur des architectures avec des instructions de taille variable, tel Intel x86-64, il faut que l'instruction ait au moins la taille d'un saut. Sur Intel, cela correspond à une instruction de 5 octets. On ne peut donc pas insérer des points de trace rapides n'importe où, et cela peut poser problème. Cette limitation est connue, tel que dénoté au Chapitre 3. La limitation affecte aussi les points d'arrêt conditionnels rapides puisqu'ils utilisent la même technique.

Affichage de vagues d'exécution

Les vagues d'exécutions sur un processeur graphique sont regroupées dans une structure en arbre basée sur leur position dans la grille de données. Cependant, seuls les nœuds du premier niveau de l'arbre sont affichés, puisqu'il y a un très grand nombre de vagues. Le développeur doit agrandir manuellement les groupes qu'il désire explorer. Cela peut compliquer l'utilisation de la vue, puisque l'utilisateur doit naviguer dans l'arbre.

Filtres

Les filtres permettent à l'usager de ne pas être averti d'évènements qui se produisent en dehors du focus. Cependant, le débogueur reçoit tout de même ces évènements, et doit les gérer. Cela fait en sorte que le programme peut être ralenti même si l'utilisateur ne reçoit pas d'évènement. Il s'agit du même problème que les points d'arrêt conditionnels normaux.

5.3 Présentation aux partenaires industriels

Les travaux effectués sur GDB et sur l'affichage de fils d'exécution sur des processeurs conventionnels ont été présentés chez Ericsson. Les développeurs d'outils de débogage ont donné plusieurs commentaires. Cela a notamment permis d'améliorer la vue d'agrégation de la pile d'appels, et a mené à utiliser JavaFX pour le « Multicore Visualizer » afin de le rendre plus flexible.

Une autre présentation a été faite aux développeurs d'outils de la compagnie AMD, qui développe un système basé sur le standard HSA. Les ingénieurs d'AMD ont donné leurs commentaires, ce qui a servi à orienter les travaux. Tel que mentionné au Chapitre 4, le « Multicore Visualizer » pour les cartes graphiques a été abandonné suite à cette présentation. En effet, ils ont expliqué que la faible affinité des vagues à un cœur particulier sur une carte graphique diminue grandement l'intérêt pour

cette vue.

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

6.1 Synthèse des travaux

Le domaine du déboguage sur les systèmes hétérogènes multi-coeurs a été le sujet de ce project de recherche. Il s'agit d'une problématique très importante en raison de la généralisation des processeurs multi-coeurs, de l'augmentation du nombre de coeurs de calcul dans ceux-ci ainsi que de l'utilisation grandissante d'accélérateurs de calcul. L'adoption de débogueurs adaptés à ces systèmes complexes est donc nécessaire pour plusieurs acteurs de l'industrie afin de contrôler la complexité du développement logiciel et les coûts associés. Les travaux présentés dans ce mémoire ont fait l'objet d'un article qui a été soumis au « Journal of systems and software ».

Nous avons premièrement commencé par faire une revue des avancées dans le domaine du débogage parallèle et hétérogène. Cela a inclus une présentation de plusieurs types d'architectures parallèles et de leurs caractéristiques. Puis, nous avons abordé les différents débogueurs et techniques de débogage disponibles. Finalement, les méthodes de traçage ont été expliquées, et plusieurs traceurs ont été présentés.

Par la suite, la performance de GDB a été évaluée. Des lacunes ont notamment été trouvées avec l'architecture de traçage sur des processeurs parallèles. En effet, celle-ci n'était pas adaptée aux programmes avec un grand nombre de fils d'exécution. De plus, des limitations ont été notées avec les points d'arrêt conditionnels, en raison du ralentissement associé à leur utilisation.

Eclipse CDT a aussi été testé avec des programmes parallèles. Il a été constaté que l'interface n'est pas bien adaptée à ce type de programme. En effet, la principale fenêtre utilisée, qui liste les processus et les fils d'exécution du programme, utilise simplement une liste pour afficher les éléments. Or, ce type de vue devient rapidement difficile à utiliser lorsque le nombre de fils augmente. De plus, cette vue doit être utilisée chaque fois que l'usager désire changer de fil. Il peut donc être amené à fermer ou ouvrir des éléments de la liste, ce qui complique l'utilisation.

Puis, différentes solutions pour pallier aux problèmes trouvés ont été envisagés. Utiliser LTTng a été envisagé afin de profiter de sa performance de traçage sur des systèmes parallèles. GDB a été modifié afin de permettre d'insérer un point de trace dans le programme et d'utiliser LTTng afin de collecter l'information. Cela a fortement amélioré les performances de traçage sur des systèmes multi-coeurs. De plus, une technique a été proposée afin d'adapter LTTng, puisque ce traceur ne supporte normalement que les évènements définis lors de la compilation. En profitant de la méthode proposée, il est possible d'insérer un point de trace dynamiquement avec GDB et de collecter les évènements avec LTTng. Cette combinaison qui offre la possibilité d'insérer dynamiquement des

points de trace avec une efficacité de traçage grandement accrue est une première contribution majeure.

De plus, une nouvelle technique pour afficher les fils d'exécutions ont été testé avec Eclipse CDT. La vue créée groupe automatiquement les fils d'exécution ensemble afin de facilement montrer l'état du système à l'utilisateur. Grâce à cette vue, le développeur peut rapidement comprendre où est-ce que les fils sont arrêtés, et repérer les groupes de fils qui exécutent des tâches similaires. La vue de débogage principale affichant les fils d'exécution a aussi été modifié afin d'inclure les vagues d'exécution des cartes graphiques. Étant donné qu'il peut y avoir un très grand nombre de vagues d'exécution traitées simultanément, celle-ci sont représentées dans une structure en arbre. Les vagues sont groupées automatiquement en utilisant la position de chaque vague dans la grille de données à trois dimensions. De cette manière, il est possible à l'utilisateur de voir chaque vague, mais la vue n'est pas surchargée par la quantité d'éléments qu'elle contient. Ceci constitue une seconde contribution importante.

Finalement, un filtre pour retirer des fils ou des vagues d'exécution a été ajouté au débogueur graphique. Ce filtre vise à réduire la difficulté de traiter un programme largement parallèle. En effet, même avec des commandes et des vues adapter, il n'est pas toujours facile de déboguer ce genre de problème. Avec le filtre, l'utilisateur peut sélectionner les fils ou vagues qui l'intéressent, par exemple ceux qui risquent de rencontrer un problème, et ne voir que ceux-ci. Cela permet d'enlever les informations superflues, pour aider le développeur à trouver les problèmes.

6.2 Améliorations futures

Tel que mentionné au Chapitre 5, l'infrastructure de traçage dynamique ne peut insérer un point de trace que lorsque l'instruction à remplacer est longue d'au moins 5 octets sur Intel x86-64. Cela limite le nombre d'endroits où un point de trace peut être inséré. Il pourrait être possible de contourner cette limitation en utilisant une autre technique d'instrumentation. Ainsi, il serait possible de déplacer une fonction complète en la modifiant, ce qui permettrait de rajouter l'espace nécessaire pour un ou plusieurs points de trace. Une autre technique possible serait de déplacer plus d'une instruction. Pour utiliser cette technique, il faudrait un outil qui est capable d'interpréter le code pour être relativement certain qu'il n'y a pas de saut vers la deuxième instruction déplacée. En effet, s'il y a un saut vers cette instruction, le programme arriverait au milieu de l'instruction dynamiquement ajouté, ce qui amènerait à un comportement indéfini. Modifier l'infrastructure de traçage permettrait donc d'améliorer la flexibilité du traçage rapide avec GDB et de faciliter son utilisation.

De plus, l'affichage des vagues d'exécution d'une carte pourrait être améliorée. En effet, la tech-

nique proposée groupe automatiquement les vagues en se basant sur leur position dans la grille de données. Cependant, cette technique crée un arbre avec un très grand nombre d'enfants. L'utilisateur doit donc fermer et ouvrir les différents éléments de cet arbre afin de voir les différentes vagues, ce qui peut prendre beaucoup de temps. Il serait intéressant d'avoir une vue qui montre l'état global des vagues et qui permettrait aussi à l'utilisateur de facilement choisir un sous-groupe pour avoir plus de détails.

Finalement, il serait intéressant d'intégrer un visualisateur de traces d'exécution (noyau et utilisateur) à un débogueur graphique. Cela permettrait à l'usager de commencer le traçage, et d'avoir des informations détaillées sur le fonctionnement du programme entre deux points d'arrêt. De cette manière, une section du programme qui ne peut pas facilement être arrêtée pourrait être déboguée. Actuellement, il est possible d'utiliser une combinaison de traçage et de déboguage, mais les outils de visualisation sont limités. Il serait donc très intéressant d'avoir une interface graphique qui permette aisément de déboguer et tracer un programme tout en affichant les traces.

LISTE DES RÉFÉRENCES

K. Antypas, “Allinea ddt as a parallel debugging alternative to totalview”, *Lawrence Berkeley National Laboratory*, 2007.

ARM, “big.little technology : The future of mobile”, ARM Limited, Rapp. tech., 2013.

Barcelona Supercomputing Center. (2017) Extrae. En ligne : https://tools.bsc.es/extrae

——. (2017) Paraver : a flexible performance analysis tool. En ligne : https://tools.bsc.es/paraver#coopanalysis

T. Beauchamp et D. Weston, “Dtrace : The reverse engineer’s unexpected swiss army knife”, *Blackhat Europe*, 2008.

A. R. Bernat et B. P. Miller, “Anywhere, any-time binary instrumentation”, dans *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, série PASTE ’11. New York, NY, USA : ACM, 2011, pp. 9–16. DOI : 10.1145/2024569.2024572. En ligne : http://doi.acm.org/10.1145/2024569.2024572

A. Binstock. (2013, Fevrier) Embedded print statements != debugging. En ligne : http://www.drdobbs.com/architecture-and-design/embedded-print-statements-debugging/240148855

T. Bird, “Measuring function duration with ftrace”, dans *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 47–54.

B. Boothe, “Efficient algorithms for bidirectional debugging”, *SIGPLAN Not.*, vol. 35, no. 5, pp. 299–310, Mai 2000. DOI : 10.1145/358438.349339. En ligne : http://doi.acm.org/10.1145/358438.349339

B. Brandenburg et J. Anderson, “Feather-trace : A lightweight event tracing toolkit”, dans *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, 2007, pp. 19–28.

P. Brook et D. Jacobowitz, “Reversible debugging”, dans *GCC Developers’ Summit*, 2007, p. 69.

G. Chrysos, “Intel® xeon phi™ coprocessor-the architecture”, Intel, Rapp. tech., 2014.

H. Chung, M. Kang, et C. H.-D., “Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology”, Samsung Electronics Co., Rapp. tech., 2012.

J. Corbet. (2012, may) Uprobes in 3.5. En ligne : https://lwn.net/Articles/499190/

C. H. Crawford, P. Henning, M. Kistler, et C. Wright, “Accelerating computing with the cell broadband engine processor”, dans *Proceedings of the 5th Conference on Computing Frontiers*, série CF ’08. New York, NY, USA : ACM, 2008, pp. 3–12. DOI : 10.1145/1366230.1366234. En ligne : http://doi.acm.org/10.1145/1366230.1366234

M. Daga, A. M. Aji, et W. c. Feng, “On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing”, dans *2011 Symposium on Application Accelerators in High-Performance Computing*, July 2011, pp. 141–149. DOI : 10.1109/SAAHPC.2011.29

J. Desfossez. (2017, février) Analyses scripts for lttng kernel and user-space traces (official repository). En ligne : https://github.com/lttng/lttng-analyses

M. Desnoyers. (2017, Mai) Using the linux kernel tracepoints. En ligne : https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

M. Desnoyers et M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux”, dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

——, “Lockless multi-core high-throughput buffering scheme for kernel tracing”, *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 3, pp. 65–81, Déc. 2012. DOI : 10.1145/2421648.2421659. En ligne : http://doi.acm.org/10.1145/2421648.2421659

A. M. Devices. (2016) Gpuopen. En ligne : http://http://gpuopen.com

Efficios. (2016, octobre) Babeltrace 2.0.0-pre python bindings. En ligne : http://diamon.org/babeltrace/docs/python/

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

E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, et E. Darve, “N-body simulations on gpus”, *arXiv preprint arXiv :0706.3060*, 2007.

J. Engblom, “A review of reverse debugging”, dans *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Sept 2012, pp. 1–6.

N. Ezzati-Jivan et M. Dagenais, “Multi-scale navigation of large trace data : A survey”, *Concurrency and Computation : Practice and Experience*, 2017. DOI : 10.1002/cpe.4068

H. Foundation, “Hsa platform system architecture specification 1.1”, HSA Foundation, Rapp. tech., January 2016.

——, “Hsa programmer’s reference manual : Hsail virtual isa and programming model, compiler writer, and object format (brig) 1.1”, HSA Foundation, Rapp. tech. 1.1, February 2016. En ligne : http://www.hsafoundation.com/standards/

P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, “Combined tracing of the kernel and applications with lttng”, dans *Proceedings of the 2009 linux symposium*. Citeseer, 2009, pp. 87–93.

Google. (2016) catapult - tracing. En ligne : https://github.com/catapult-project/catapult/blob/master/tracing/README.md

B. Gregg. (2011, october) Using systemtap. En ligne : http://dtrace.org/blogs/brendan/2011/10/15/using-systemtap/

——. (2014, may) strace wow much syscall. En ligne : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

C. Gregg et K. Hazelwood, “Where is the data? why you cannot debate cpu vs. gpu performance without the answer”, dans *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 134–144. DOI : 10.1109/ISPASS.2011.5762730

L. Gwennap, “Adapteva : More flops, less watts”, *Microprocessor Report*, vol. 6, no. 13, pp. 11–02, 2011.

B. Hailpern et P. Santhanam, “Software debugging, testing, and verification”, *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002. DOI : 10.1147/sj.411.0004

K. Hazelwood, “Dynamic binary modification : Tools, techniques, and applications”, *Synthesis Lectures on Computer Architecture*, vol. 6, no. 2, pp. 1–81, 2011. DOI : 10.2200/S00345ED1V01Y201104CAC015. En ligne : http://dx.doi.org/10.2200/S00345ED1V01Y201104CAC015

M. T. Heath et J. A. Etheridge, “Visualizing the performance of parallel programs”, *IEEE Software*, vol. 8, no. 5, pp. 29–39, Sept 1991. DOI : 10.1109/52.84214

J. L. Hennessy et D. A. Patterson, *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.

H. P. Hofstee, “Power efficient processor architecture and the cell processor”, dans *11th International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 258–262. DOI : 10.1109/HPCA.2005.26

M. Jones, C. Robertson, G. Hogenson, et S. Cai. (2016, novembre) Intellitrace features. En ligne : https://docs.microsoft.com/en-gb/visualstudio/debugger/intellitrace-features

G. M. Karam et R. J. A. Buhr, “Starvation and critical race analyzers for ada”, *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 829–843, Aug 1990. DOI : 10.1109/32.57622

J. Keniston, P. S. Panchamukhi, et M. Hiramatsu. (2017, may) Kernel probes (kprobes). En ligne : https://www.kernel.org/doc/Documentation/kprobes.txt

M. Khouzam, “Making your debugging efforts count : Best pratices with the cdt debugger”, dans *EclipseCon North America (Reston, Virginia, USA, March 2016)*, Reston, USA, 2016.

N. Kim. (2017, avril) Function (graph) tracer for user-space. En ligne : https://github.com/namhyung/uftrace

A. Kleen et B. Strong, “Intel® processor trace on linux”, *Tracing Summit*, vol. 2015, 2015.

A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, et W. E. Nagel, *The Vampir Performance Analysis Tool-Set*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 139–155. DOI : 10.1007/978-3-540-68564-7_9. En ligne : http://dx.doi.org/10.1007/978-3-540-68564-7_9

R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, et K. I. Farkas, “Single-isa heterogeneous multi-core architectures for multithreaded workload performance”, dans *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 64–75. DOI : 10.1109/ISCA.2004.1310764

L. Lamport, “A fast mutual exclusion algorithm”, *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 1–11, Jan. 1987. DOI : 10.1145/7351.7352. En ligne : http://doi.acm.org/10.1145/7351.7352

E. S. Larsen et D. McAllister, “Fast matrix multiplies using graphics hardware”, dans *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, série SC ’01. New York, NY, USA : ACM, 2001, pp. 55–55. DOI : 10.1145/582034.582089. En ligne : http://doi.acm.org/10.1145/582034.582089

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, et G. Venolia, “Debugging revisited : Toward understanding the debugging needs of contemporary software developers”, dans *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013, pp. 383–392. DOI : 10.1109/ESEM.2013.43

E. Lindholm, J. Nickolls, S. Oberman, et J. Montrym, “Nvidia tesla : A unified graphics and computing architecture”, *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008. DOI : 10.1109/MM.2008.31

LLVM Project. (2017) The **LLDB** debugger. En ligne : https://lldb.llvm.org/

D. Luebke et G. Humphreys, “How gpus work”, *Computer*, vol. 40, no. 2, 2007.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, et K. Hazelwood, “Pin : Building customized program analysis tools with dynamic instrumentation”, *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Juin 2005. DOI : 10.1145/1064978.1065034. En ligne : http://doi.acm.org/10.1145/1064978.1065034

N. Matloff et P. J. Salzman, “{CHAPTER} 1 - some preliminaries for beginners and pros”, dans *The Art of Debugging with GDB, DDD, and Eclipse*. San Francisco, CA, USA : No Starch Press, 2008, pp. 1 – 46.

R. C. Metzger, “9 - debugging tactics”, dans *Debugging by Thinking*, série HP Technologies, R. C. Metzger, éd. Burlington : Digital Press, 2004, pp. 221 – 256. DOI : https://doi.org/10.1016/B978-155558307-1/50009-4. En ligne : http://www.sciencedirect.com/science/article/pii/B9781555583071500094

Microsoft. (2015) Debugging in visual studio. En ligne : https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

——. (2015) Debugger windows - visual studio 2015. En ligne : https://msdn.microsoft.com/en-us/library/mt599640.aspx

Mozilla. (2017, may) How rr works. En ligne : http://rr-project.org/rr.html#1.0

N. Nethercote et J. Seward, “Valgrind : A framework for heavyweight dynamic binary instrumentation”, *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Juin 2007. DOI : 10.1145/1273442.1250746. En ligne : http://doi.acm.org/10.1145/1273442.1250746

J. Nickolls et W. J. Dally, “The gpu computing era”, *IEEE Micro*, vol. 30, no. 2, pp. 56–69, March 2010. DOI : 10.1109/MM.2010.41

A. Olofsson, T. Nordström, et Z. Ul-Abdin, “Kickstarting high-performance energy-efficient manycore architectures with epiphany”, dans *2014 48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 1719–1726. DOI : 10.1109/ACSSC.2014.7094761

A. Olofsson et M. Khouzam, “Cdt and parallella : Multicore debugging for the masses”, dans *EclipseCon North America (San Francisco, California, USA, March 2016)*, San Francisco, USA, 2014.

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, et J. C. Phillips, “Gpu computing”, *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. DOI : 10.1109/JPROC.2008.917757

E. Persson, “Ati radeon™ hd 2000 programming guide”, AMD Graphic Products Group, Rapp. tech., 2007.

G. Project. (2013, août) Ddd v3.3 - data display debugger. En ligne : https://www.gnu.org/software/ddd/#Doc

T. C. Project. (2017) Trace compass user guide. En ligne : http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/User-Guide.html

Qualcomm, “Snapdragon s4 processors : Sytem on chip solutions for a new mobile age”, Qualcomm Inc., Rapp. tech., 2011.

RogueWave. Totalview for hpc features. En ligne : https://www.roguewave.com/products-services/totalview/features

E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, et D. Rajwan, “Power-management architecture of the intel microarchitecture code-named sandy bridge”, *IEEE Micro*, vol. 32, no. 2, pp. 20–27, March 2012. DOI : 10.1109/MM.2012.12

A. Shan, “Heterogeneous processing : A strategy for augmenting moore’s law”, *Linux J.*, vol. 2006, no. 142, pp. 7–, Fév. 2006. En ligne : http://dl.acm.org/citation.cfm?id=1119128.1119135

S. Shebs, “Gdb tracepoints, redux”, *Proceedings of the GCC Developers’ Summit. GCC Summit. Montréal, Canada*, pp. 105–112, 2009.

N. Sidwell, V. Prus, P. Alves, S. Loosemore, et J. Blandy, “Non-stop multi-threaded debugging in gdb”, dans *GCC Developers’ Summit*, vol. 117, 2008.

A. Spear, M. Levy, et M. Desnoyers, “Using tracing to solve the multicore system debug problem”, *Computer*, vol. 45, no. 12, pp. 60–64, Dec 2012. DOI : 10.1109/MC.2012.191

R. Stallman, R. Pesch, S. Shebs *et al.*, *Debugging with GDB*, 10e éd. Free Software Foundation, 2017.

J. E. Stone, D. Gohara, et G. Shi, “Opencl : A parallel programming standard for heterogeneous computing systems”, *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010. DOI : 10.1109/MCSE.2010.69

G. Stoner, “Hsa foundation start of new era in computing”, http ://www.hsafoundation.com/hello-hsa-foundation/, June 2012.

A. S. Tanenbaum et H. Bos, *Modern Operating Systems*, 4e éd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2014.

The LTTng Project. (2017, march) The lttng documentation. En ligne : http://lttng.org/docs/v2.9/

L. Torvalds. (2002, Septembre) Re : [patch] ltt for 2.5.38 1/9 : Core infrastructure. En ligne : https://lkml.org/lkml/2002/9/22/102

Undo. (2017, may) Reversible debugging tools for c/c++ on linux and android. En ligne : http://undo.io/technology/

A. Venkat et D. M. Tullsen, “Harnessing isa diversity : Design of a heterogeneous-isa chip multiprocessor”, dans *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 121–132. DOI : 10.1109/ISCA.2014.6853218

A. Vergé, N. Ezzati-Jivan, et M. R. Dagenais, “Hardware-assisted software event tracing”, *Concurrency and Computation : Practice and Experience*, pp. e4069–n/a, 2017, e4069 cpe.4069.

DOI : 10.1002/cpe.4069. En ligne : http://dx.doi.org/10.1002/cpe.4069

A. I. Wasserman et P. A. Pircher, “A graphical, extensible integrated environment for software development”, *SIGPLAN Not.*, vol. 22, no. 1, pp. 131–142, Jan. 1987. DOI : 10.1145/390012.24224. En ligne : http://doi.acm.org/10.1145/390012.24224

J. Weidendorfer, *Sequential Performance Analysis with Callgrind and KCachegrind*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 93–113. DOI : 10.1007/978-3-540-68564-7_7. En ligne : http://dx.doi.org/10.1007/978-3-540-68564-7_7

R. W. Wisniewski et B. Rosenburg, “Efficient, unified, and scalable performance monitoring for multiprocessor operating systems”, dans *Supercomputing, 2003 ACM/IEEE Conference*, Nov 2003, pp. 3–3. DOI : 10.1145/1048935.1050154

G. Yeap, “Smart mobile socs driving the semiconductor industry : Technology trend, challenges and opportunities”, dans *2013 IEEE International Electron Devices Meeting*, Dec 2013, pp. 1.3.1–1.3.8. DOI : 10.1109/IEDM.2013.6724540

O. Zaki, E. Lusk, W. Gropp, et D. Swider, “Toward scalable performance visualization with jumpshot”, *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999. DOI : 10.1177/109434209901300310. En ligne : http://dx.doi.org/10.1177/109434209901300310

A. Zeller, “{CHAPTER} 6 - scientific debugging”, dans *Why Programs Fail (Second Edition)*, second edition éd., A. Zeller, éd. Boston : Morgan Kaufmann, 2009, pp. 129 – 146. DOI : https://doi.org/10.1016/B978-0-12-374515-6.00006-X. En ligne : http://www.sciencedirect.com/science/article/pii/B978012374515600006X

——, “{CHAPTER} 8 - observing facts”, dans *Why Programs Fail (Second Edition)*, second edition éd., A. Zeller, éd. Boston : Morgan Kaufmann, 2009, pp. 175 – 209. DOI : https://doi.org/10.1016/B978-0-12-374515-6.00008-3. En ligne : http://www.sciencedirect.com/science/article/pii/B9780123745156000083

——, “{CHAPTER} 4 - reproducing problems”, dans *Why Programs Fail (Second Edition)*, second edition éd., A. Zeller, éd. Boston : Morgan Kaufmann, 2009, pp. 75 – 103. DOI : https://doi.org/10.1016/B978-0-12-374515-6.00004-6. En ligne : http://www.sciencedirect.com/science/article/pii/B9780123745156000046

——, “Visual debugging with ddd”, *DOCTOR DOBBS JOURNAL*, vol. 26, no. 3, pp. 21–29, 2001.

A. Zellers, “Debugging with ddd”, Free Software Foundation, Rapp. tech., Janvier 2004.