Titre: Title:	Débogage et traçage de processeurs à grand nombre de coeurs
Auteur: Author:	Simon Marchi
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Marchi, S. (2014). Débogage et traçage de processeurs à grand nombre de coeurs [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1416/

UNIVERSITÉ DE MONTRÉAL

DÉBOGAGE ET TRAÇAGE DE PROCESSEURS À GRAND NOMBRE DE CŒURS

SIMON MARCHI
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)
AVRIL 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DÉBOGAGE ET TRAÇAGE DE PROCESSEURS À GRAND NOMBRE DE CŒURS

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

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

REMERCIEMENTS

J'aimerais d'abord remercier mon directeur de recherche, Michel Dagenais, pour m'avoir soutenu, autant lors du baccalauréat que de la maitrise. Lorsque je l'ai rencontré avant mon stage de première année dans le laboratoire, j'ai vaguement mentionné que, si j'en avais l'occasion, j'aimerais travailler dans le domaine du logiciel libre (sans trop savoir ce que ça impliquait, à l'époque). Si j'ai la chance de le faire aujourd'hui, c'est grandement grâce aux portes qui m'ont été ouvertes par le passage par son laboratoire.

Un grand merci, également, à nos associés de recherche, Yannick et Geneviève, pour le support technique et moral. Particulièrement, votre combat acharné pour obtenir un Xeon Phi fonctionnel ne sera pas oublié.

Évidemment, le temps passé au laboratoire n'aurait pas été le même sans la bonne compagnie de mes collègues. J'ai autant apprécié offrir que recevoir de l'aide de votre part. De plus, le temps passé à agrémenter l'ambiance du laboratoire aura certainement été aussi enrichissant qu'utile !

Finalement, merci aux membres du jury pour l'attention que vous porterez à ce mémoire, je suis très reconnaissant que vous accordiez de votre temps précieux à mon travail.

RÉSUMÉ

Les systèmes informatiques ont atteint un point où la performance d'un seul cœur rapide ou même d'un seul hôte puissant ne peut pas répondre à la demande. Les nouveaux systèmes sont donc de plus en plus bâtis selon des architectures distribuées, permettant une meilleure mise à l'échelle. On constate également l'arrivée de processeurs à grand nombre de cœurs, qui exploitent le parallélisme de façon extrême plutôt que de compter sur la puissance individuelle de chaque cœur. Ces changements compliquent l'utilisation des outils d'analyse du comportement des programmes. Nous nous sommes penchés sur deux problèmes en particulier, soit le débogage des appels à distance et le traçage sur processeurs à grand nombre de cœurs.

Au sein de systèmes distribués, un paradigme fréquemment utilisé est celui des appels de procédures à distance. Cette technique permet à un processus de faire exécuter une fonction dans un contexte différent, par exemple sur un hôte distant, plus adapté à accomplir la tâche voulue. Ainsi le fil d'exécution logique de l'application passe d'un hôte à l'autre, même si chaque hôte possède évidemment son propre fil d'exécution réel. Des bibliothèques spécialisées permettent même de cacher la complexité des appels distants et font en sorte qu'ils soient aussi simples à effectuer qu'un appel local. Toutefois, les débogueurs n'ont généralement pas connaissance de cette technique. Leur efficacité dans ce contexte en est donc réduite, puisqu'il est impossible de suivre le fil d'exécution logique lorsqu'il quitte l'hôte actuel.

Il existe quelques exemples de débogueurs auxquels la possibilité de suivre des appels à distance a été ajoutée. Par contre, il s'agissait toujours d'une solution spécifique à une bibliothèque particulière d'appels à distance ou à une seule plate-forme.

Nous proposons dans ce mémoire une méthode générique permettant d'ajouter aux débogueurs une connaissance des systèmes d'appels à distance. Nous avons effectué une implémentation de la solution à l'aide du débogueur GDB. GDB possède plusieurs fonctions intéressantes, comme la gestion de plusieurs processus débogués à la fois et est facilement extensible.

La solution consiste à placer des points d'arrêt aux endroits dans le programme client qui correspondent à l'initiation d'un appel à distance. Une fois un de ces points d'arrêt atteint, on peut déterminer à quel endroit l'exécution devrait passer dans le serveur pour servir la requête d'appel à distance du client. Il est alors possible d'y placer un point d'arrêt, afin d'interrompre le programme et permettre à l'utilisateur de continuer le débogage à partir de ce point. Une stratégie équivalente est utilisée pour le retour de l'appel à distance.

La solution doit bien sûr être adaptée à chaque bibliothèque spécifique d'appels à distance. Toutefois, comme le même modèle d'exécution se retrouve dans la plupart de ces bibliothèques, il est possible d'extraire une grande partie de la logique dans un module générique. La portion restante, spécifique à chaque bibliothèque, est donc très petite. Pour démontrer la flexibilité de la solution, le support de plusieurs bibliothèques été ajouté.

Le deuxième volet de notre travail porte sur le traçage des processeurs à grand nombre de cœurs. Dans des cas où le débogage interactif est trop invasif, le traçage peut être utilisé pour recueillir de l'information sur l'exécution du programme. Le traçage est semblable à la journalisation, mais met une emphase particulière à minimiser l'impact sur la performance de l'application étudiée.

Des points de traces sont préalablement insérés dans le programme. Lorsque l'exécution croise un de ces points, une information (nommée un évènement) est sauvegardée dans la trace, le fichier résultant. Dans un système multi-cœurs, chaque cœur possède son propre fil d'exécution et représente ainsi un générateur d'évènements. On comprend que plus le nombre de cœurs est élevé, plus le débit d'évènements généré a le potentiel d'être grand. Si ce débit est plus grand que ce que le traceur est capable d'enregistrer sur le médium supportant la trace, le traceur sera contraint de laisser tomber des évènements, ce qui donnera une trace incomplète.

Le traceur LTTng, développé à l'origine au laboratoire DORSAL de l'École Polytechnique de Montréal, est reconnu pour tracer efficacement aussi bien le noyau que les applications sur Linux. D'importants efforts ont été mis lors de son développement pour s'assurer que sa performance reste satisfaisante sur des systèmes multi-cœurs.

La première étape de ce volet a été le port de LTTng à deux nouvelles architectures, TILE-Gx de Tilera et Xeon Phi d'Intel. Dans le cas du processeur de Tilera, nos travaux ont été effectués sur un processeur comportant 36 cœurs alors que celui d'Intel en comportait 57. Étant une architecture jeune, le port à TILE-Gx a requis plusieurs correctifs autant au noyau de Linux qu'à LTTng. Le port au Xeon Phi a quant à lui été beaucoup plus direct.

Nous nous sommes ensuite intéressés à la performance de LTTng sur deux processeurs particuliers. Pour chacun, nous avons choisi une application représentative du segment de marché visé par le produit. Pour le processeur de Tilera, il s'agit d'un nœud de cache distribué au sein d'un système d'infonuagique. Pour celui d'Intel, il s'agit d'une application de calcul scientifique.

Plusieurs variations de traçage et d'exécution ont été étudiées. Notamment, nous comparons la sauvegarde locale et par réseau des données de traçage. En effet, comme l'espace disque est très restreint sur ce genre de plate-formes, l'enregistrement local des données est assez limité.

Les résultats montrent que l'impact en performance de LTTng sur les applications tracées ne change pas (en proportion), même à des nombres de cœurs très grands. Toutefois, la sauvegarde par réseau reste un problème, puisque le médium n'est pas capable de supporter le débit de traçage généré dès que la charge est importante sur un nombre élevé de cœurs.

ABSTRACT

Computer systems reached a point where the performance of a single fast core, or even a single powerful host can't reach the required performance for some large-scale applications. New systems are more and more built using distributed architectures, allowing better scaling. Many-core processors, which exploit massive parallelism, are also more and more common. These changes make standard software analysis tools harder to use. In this work, we tackle two particular problems, the debugging of remote procedure calls and application tracing on many-core processors.

Remote procedure calling is a paradigm often used when building distributed applications. It allows calling a function in a different context, for example on a distant host, which could be more apt to complete the given task. We can see it as if the logical execution flow of the program went from one host to another, even though each host has its own real execution flow. Some specialized libraries help hiding the complexity of remote calls and make them almost as easy to use as local calls. Unfortunately, debuggers are generally not aware of this technique. They are therefore less useful in this context, since it is impossible to follow the logical execution flow when it leaves the debugged host.

There are some examples of debuggers that have been taught how to debug remote procedure calls, but they are always targeting a particular library or platform.

In our work, we propose a generic method to add knowledge of remote procedure call libraries to debuggers. We implemented the solution using the GDB debugger, because it has many useful features, such as multiple process support, and is easily extensible through its Python API.

The solution consists of placing breakpoints at spots in the client program that correspond to the start of a remote call. When one of those breakpoints is hit, we can determine at which point in the server the execution should go to serve the remote call request. It is therefore possible to place a breakpoint there, which will interrupt the execution and allow the user to continue debugging from there. An equivalent strategy is used for the return portion of the call.

Obviously, the solution must be adapted to each specific remote procedure call library. However, since all these libraries are based on the same execution model, it is possible to extract most of the logic in a generic module. The remaining portion, specific to each library, is relatively small. To demonstrate the flexibility of the solution, support for a few of those libraries has been added.

The second part of our work is about tracing of many-core systems. In cases where

interactive debugging is too invasive, tracing can be used to collect information about the runtime of a program. Tracing is similar to logging, but its priority is to minimise its impact on the performance and behavior of the analyzed program.

Tracepoints are added beforehand in the studied program. When execution reaches one of those points, a piece of information, an event, is saved in the trace, the resulting file. In a multi-core system, each core has its own execution thread, and therefore is an event source. The higher the number of cores, the higher can be the rate of generated events. If the rate of tracing data is greater than what the rate the medium on which the trace is recorded is able to write to, the tracer has no choice but to start dropping events, which will result in an incomplete trace.

The LTTng tracer, developped initially at the DORSAL laboratory of the École Polytechnique de Montréal, is able to efficiently trace the Linux kernel as well as userspace applications. A lot of efforts have been made to ensure that it scales well on multi-core systems.

The first step of this work has been to port LTTng to two new architectures, TILE-Gx from Tilera and Xeon Phi from Intel. In the case of Tilera, we worked with a 36 cores processor. For the Xeon Phi platform, we worked with a 57 cores processor. Being a less mature architecture, the port to TILE-Gx required more fixes in the Linux kernel and LTTng than the Xeon Phi port.

We then looked at the performance of LTTng on those two processors. For each one of them, we chose a typical application of the market segment they target. For the Tilera processor, we chose a distributed cache node, as we can find in a cloud-computing application. For the Intel processor, we chose a scientific computing application.

Multiple variations in tracing and execution mode have been tested. For example, we compared saving the trace data locally and through the network. As disk space is very limited on this kind of platform, saving data locally is very restrictive.

Results show that the impact of LTTng on the traced applications stays constant (in proportion), even with high number of cores. However, saving trace data through the network is a problem, since the medium is not able to cope with the data flow generated from intense work on a high number of cores.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CDT	(Eclipse) C Developement Tools
GDB	GNU Debugger
GNU	GNU's Not Unix
IDB	Intel Debugger
LTTng	Linux Tracer Toolkit next generation
TMF	(Eclipse) Tracing and Monitoring Framework

CHAPITRE 1

INTRODUCTION

Le présent mémoire fait état des résultats de travaux de recherche effectués dans le champ du débogage et du traçage de plate-formes à grand nombre de cœurs.

Les processeurs à grand nombre de cœurs sont une évolution des processeurs multi-cœurs. Ils visent à pousser le parallélisme à un autre niveau en intégrant un grand nombre de cœurs moins puissants. Puisqu'ils sont efficaces pour certaines tâches particulières et non pas comme processeurs à usage général, ils sont destinés à être intégrés dans des systèmes répartis hétérogènes. Les applications s'étendant sur plusieurs nœuds sont particulièrement difficiles à déboguer avec les outils conventionnels. Nous proposons donc une technique permettant de déboguer de façon plus transparente les applications basées sur le paradigme des appels à distance, un paradigme que l'on retrouve souvent sous différentes formes dans ces appliations.

Sur un autre volet, nous étudions l'efficacité du traçage, une méthode d'analyse des systèmes informatiques alternative au débogage classique. Le fait qu'elle soit appliquée à des processeurs comportant un grand nombre de cœurs apporte des défis particuliers.

1.1 Définitions et concepts de base

1.1.1 Appel de procédure à distance

Le premier volet de ce mémoire porte sur les appels de procédures à distance. Dans un système réparti basé sur les appels à distance, certains intervenants (les serveurs) exposent une liste de procédures auxquelles d'autres intervenants (les clients) peuvent faire appel. Pour effectuer un appel, un client transmet au serveur une indication de la procédure à exécuter avec les arguments que la procédure requiert. Le serveur exécute la procédure demandée et envoie le résultat du calcul au client. Comme nous le verrons dans la revue de littérature, certains éléments peuvent varier (support de communication, protocole de sérialisation des données, interface de programmation), mais le concept tourne autour de ce patron.

Pour notre travail, la définition d'appel à distance se limitera aux bibliothèques d'appels à distance réseaux, qui permettent les appels entre deux processus sur le même hôte ou sur deux hôtes sur le même réseau.

1.1.2 Débogueur

Le débogage a existé aussi longtemps que la programmation elle-même. Toute étude d'un programme qui ne se comporte pas comme on le désire est une forme de débogage. De façon plus précise, un débogueur tel qu'on l'entend généralement est un programme assistant le programmeur dans son analyse du programme fautif. Un débogueur conventionnel est capable de pauser et reprendre l'exécution du programme à l'étude pour permettre au programmeur d'inspecter l'état de celui-ci. Il permet aussi généralement l'exécution d'une seule instruction ou d'une seule ligne de code à la fois.

Pour permettre que le programme s'arrête aux endroits voulus, le programmeur peut indiquer sa volonté en insérant des *points d'arrêt* à certains endroits du programme. Lorsque l'exécution rencontre un de ces points, elle s'arrête pour lui permettre d'inspecter l'état du programme.

1.1.3 Traceur

Le traçage est une technique alternative au débogage qui vise à enregistrer des informations sur l'exécution d'un programme en minimisant l'impact de l'observation sur celui-ci. La première étape du traçage consiste en l'insertion de *points de traces* à divers endroits d'intérêt, c'est-à-dire là ou le programme effectue des actions qui nous intéressent. Lorsque le programme est exécuté avec le traçage activé, à chaque fois que l'exécution croise un de ces points de trace, un *évènement* est généré et écrit dans un tampon, accompagné d'une estampille temporelle. Les données résultantes, la *trace*, consiste donc en un récit détaillé de l'exécution du programme.

La distinction à faire entre le traçage et une autre méthode plus commune, la journalisation, est que le traçage ne bloquera jamais l'application tracée. Pour satisfaire le requis d'intrusion minimale, si le programme génère plus d'évènements que le traceur est capable de consommer, on préfèrera perdre les données de traçage plutôt que de faire attendre l'application.

Une application particulière d'un traceur est l'étude du noyau du système d'exploitation. Dans le cadre de notre travail, nous nous intéressons aux traceurs spécifiques au noyau du système d'exploitation Linux.

1.2 Éléments de la problématique

Dans cette section, nous exposons les problèmes que les systèmes répartis intégrant des plate-formes à grand nombre de cœurs posent aux outils d'analyse existants.

1.2.1 Débogage d'appels à distance

Le débogage des plate-formes réparties et très parallèles pose plusieurs problèmes relatifs à l'utilisation des débogueurs classiques. Tout d'abord, ces outils sont généralement faits pour analyser un seul fil d'exécution de façon séquentielle. Lorsque le programmeur fait avancer l'exécution dans un fil en particulier, il perd le portrait global composé de tous les fils, qui peuvent se compter en dizaines et en centaines. Ensuite, comme ces outils sont faits pour étudier un fil d'exécution à la fois, les appels à distance, qui effectuent des appels dans un autre contexte d'exécution, représentent une muraille que les débogueurs ne peuvent franchir. C'est sur ce dernier problème que nous nous pencherons dans ce travail.

Dans les systèmes que l'on étudie, les processeurs à grand nombre de cœurs sont souvent très efficaces pour une ou quelques tâches précises. On les retrouvera donc comme coprocesseurs ou périphériques à des systèmes comportant des processeurs plus conventionnels. Les applications sont donc principalement exécutées sur les processeurs conventionnels, et ceux-ci délestent les tâches plus lourdes aux coprocesseurs. Souvent, ce délestage prendra la forme d'un appel à distance du processeur conventionnel vers le processeur spécialisé.

Plusieurs bibliothèques d'appels à distance tentent de faire en sorte que, du point de vue du programmeur, l'appel à une procédure distante ressemble autant que possible à appel local. Cela facilite énormément l'écriture du code, puisque cela permet au programmeur de structurer son programme comme si l'appel était local. Il passe donc moins de temps à se soucier de détails de bas niveau, comme la communication réseau ou la transmission des paramètres, et plus de temps à penser à la tâche réelle que le programme doit accomplir.

Toutefois, les débogueurs ne fournissent pas une aisance équivalente. Si le développeur tente de déboguer un programme faisant un appel à distance, il sera en mesure de voir la préparation de l'appel, l'envoi des paramètres et la réception de la réponse. Il ne pourra pas avancer pas-à-pas dans le code du serveur, qui effectue le travail réellement intéressant. Une option possible est de s'attacher d'avance au processus serveur et de placer un point d'arrêt à un endroit qui sera exécuté au début de l'appel à distance, ce qui permettra d'avancer pas-à-pas dans le code du serveur. Toutefois, cela demande davantage d'opérations dans un processus déjà laborieux, et n'est parfois pas suffisant. Par exemple, lorsqu'on place un point d'arrêt inconditionnel dans le serveur, il se peut que l'on interrompe d'autres invocations d'appels à distance, autres que celle qu'on voulait déboguer.

Comme nous le verrons dans la revue de littérature, certains outils permettent le débogage transparent du client vers le serveur. Toutefois, ces outils sont très spécifiques et ne peuvent être utilisés que dans le contexte pour lequel ils ont été prévus. Il n'existe pas de méthode générique pour déboguer de façon transparente un appel entre deux processus quelconques.

1.2.2 Traçage sur processeurs à grand nombre de cœurs

Tel que mentionné précédemment, le traçage est une technique permettant d'obtenir un portrait très précis de l'exécution d'un programme, en minimisant l'impact de l'observateur sur celle-ci. Cela est utile lorsqu'on désire observer le programme tout en le laissant fonctionner à pleine vitesse.

Les traceurs les plus efficaces utilisent plusieurs techniques leur permettant de minimiser leur impact en performance. Par exemple, ils utilisent généralement un système à plusieurs tampons utilisés en rotation. Pendant qu'un tampon reçoit les évènements générés par le programme, le traceur peut consommer le contenu du ou des autres en écrivant les données sur le disque. L'espace de tampon n'est donc jamais verrouillé. Une fois les données d'un tampon écrites sur le disque, le tampon est considéré vide et est prêt à être réutilisé.

Ces traceurs utilisent également des groupes de tampons différents pour chaque processeur. Comme chaque tampon est dédié à un seul cœur virtuel, l'utilisation de primitives de synchronisation n'est pas nécessaire. On peut donc présumer que la mise à l'échelle d'un tel traceur à un grand nombre de cœurs ne sera pas affectée par des problèmes de contention de verrous.

Par contre, comme chaque chaque cœur virtuel représente un générateur d'évènement indépendant, on comprend que la quantité d'évènements générée dans un certain laps de temps sur un processeur à plusieurs centaines de cœurs risque d'être assez importante. De plus, lors de l'exécution d'un programme sur un tel processeur, le facteur limitant est souvent le système de mémoire partagée, saturé en tentant de répondre aux demandes de tous les cœurs. En effet, pour retirer des performances satisfaisantes de ces processeurs, une attention toute particulière doit être portée à la façon dont chaque fil accède à la mémoire. Autrement, les cœurs passent leur temps à attendre que leurs accès mémoire soient complétés et sont sous-utilisés.

Le flux des données de traçage entre donc en compétition avec les accès mémoire du programme analysé. Si le bus vers la mémoire est déjà saturé sans le traçage, l'ajout de celui-ci pourrait causer d'importantes baisses de performance.

1.3 Objectifs de recherche

Considérant les éléments exposés dans la section 1.2, les objectifs de recherche sont les suivants :

1. Proposer et implémenter une méthode générique permettant le débogage transparent des appels de procédure à distance.

2. Évaluer et caractériser l'impact en performance, sur les applications, du traçage noyau et utilisateur sur des processeurs à grand nombre de cœurs.

1.4 Plan du mémoire

Dans le chapitre 2, nous présentons une revue de littérature autour des concepts reliés à nos objectifs : appels de procédures à distance, débogage, traçage, architecture à grand nombre de cœurs, etc.

Le chapitre 3 présente des concepts plus avancés, à propos du fonctionnement du débogueur GDB, celui que nous utilisons pour compléter notre premier objectif. Ces concepts sont à la base du fonctionnement de la méthode présentée dans le chapitre 4.

Le chapitre 5 traite quand à lui du second objectif. Il présente les résultats de divers tests effectués pour évaluer l'impact du traçage sur des applications types exécutées sur des processeurs à grand nombre de cœurs.

Finalement, une conclusion vient faire le point sur le travail, en discutant des forces et faiblesses de ce qui a été réalisé et en proposant des améliorations futures.

CHAPITRE 2

REVUE DE LITTÉRATURE

Cette section présente une revue de la littérature et de l'état actuel de la technologie en ce qui a trait aux trois facettes de ce travail. Nous aborderons d'abord le sujet des appels à distance, afin d'avoir un portrait des différences et similitudes entre les différents protocoles et implémentations. Nous passerons ensuite en revue des outils d'analyse du comportement des programmes, principalement ceux de débogage et de traçage.

2.1 Protocoles d'appels de procédures à distance

Le modèle de programmation basé sur les appels de procédures à distance (RPC) est une technique de communication utilisée pour concevoir des systèmes distribués. D'abord proposée en 1984 par Birrell et Nelson, cette technique permet de transférer le contrôle logique de l'exécution d'un programme, d'un système à un autre. À haut niveau, un programme dit client semble faire un appel à une fonction présente dans un espace d'adressage différent du sien. Cet espace différent, dans un programme dit serveur, peut aussi bien être sur le même système que sur un système différent. Le cadre d'application utilisé permet de cacher la complexité de la communication et donne l'impression au programmeur que l'appel est local (Comer, 2004). Les appels de procédures à distance peuvent être utilisés pour délester un traitement lourd sur un système plus puissant ou même quérir de l'information sur un serveur de bases de données.

Le principe de fonctionnement des bibliothèques RPC se base souvent sur l'utilisation de souches ou d'objets proxy (Comer, 2004). Pour chaque prodécure disponible sur le serveur, le client possède une procédure portant le même nom et la même signature, permettant au programme d'appeler la fonction exactement comme si elle était locale. La version de la fonction côté client s'occupe de convertir les paramètres en un format propice à leur retransmission sur le réseau (cette étape est parfois nommée sérialisation) et les envoit au serveur. Le client se met ensuite en attente d'une réponse. De l'autre côté, le serveur décode les paramètres reçus et appelle la version de la fonction contenant l'implémentation réelle. Une fois l'appel à cette fonction terminé, le serveur sérialise la valeur de retour et l'envoie au client. La version souche de la fonction désérialise cette valeur et la retourne au programme principal, terminant alors l'appel de procédude à distance.

L'avantage du fait que les fonctions sur le client et le serveur aient des signatures iden-

tiques est que le changement d'un appel local en appel distant ou vice-versa se fait très aisément. Le code source appelant cette fonction n'a pas à être modifié.

Pour assister le programmeur dans sa tâche, certaines bibliothèques permettent de générer un intergiciel qui facilite grandement l'appel de procédures distantes. Les différentes procédures offertes par le serveur sont d'abord décrites à l'aide d'un langage de définition d'interface (IDL). Un outil lit cette définition puis génère des bases de code pour le client et le serveur. Du côté client, une fonction est générée pour chaque procédure décrite dans le fichier de définition d'interface. Ces fonctions contiennent le code nécessaire à l'exécution d'un appel à distance, tel que décrit plus haut, et peuvent être appelées directement par le code client. Du côté serveur, une fonction vide est générée pour chacune des procédures, où le programmeur n'a qu'à placer le code effectuant le travail réel. La bibliothèque se chargera d'appeler automatiquement la bibliothèque à la réception d'une requête correspondante.

Ce mode de fonctionnement gène notamment lors de l'utilisation d'un débogueur interactif puisque celui-ci suit le fil d'exécution réel du programme et non le fil d'exécution logique, qui lui est transféré d'un système à un autre. L'utilisateur qui manipule le débogueur peut naviguer dans les appels de fonctions sans problème jusqu'à ce que le programme effectue un appel à distance. À ce moment, il pourra au mieux explorer le code du client s'occupant de la sérialisation des paramètres et de l'envoi au serveur, mais il ne pourra pas déboguer de façon transparente le fil d'exécution logique de programme qui fait un va-et-vient du client au serveur.

La définition d'un protocole clair, notamment en ce qui a trait au format de transmission des paramètres, permet d'avoir des clients et serveurs s'exécutant sur des architectures différentes. Par exemple, le client peut utiliser une représentation gros-boutiste alors que le serveur est en petit-boutiste. Le code de chaque côté est responsable de faire la traduction entre le format natif de la machine et la représentation définie par le protocole (et vice-versa).

Le reste de cette section donne un aperçu de plusieurs protocoles d'appel de procédure à distance disponibles aujourd'hui. Comme ces protocoles ne sont que des spécifications, il peut en exister plusieurs implémentations. Nous avons fait le choix d'étudier le fonctionnement et l'utilisation de l'implémentation de référence ou d'une implémentation répandue.

2.1.1 Open Network Computer Remote Procedure Call (ONC RPC)

Le protocole d'appels à distance SunRPC a été décrit par Sun Microsystems (1988) et a plus tard été renommé ONC (Srinivasan, 1995a). Malgré son âge, ce protocole est encore fréquemment utilisé puisque c'est sur celui-ci qu'est basé Network File System (NFS), un protocole de partage de systèmes de fichiers par réseau. L'implémentation que nous avons étudié est celle disponible dans la bibliothèque glibc, présente de base sur les systèmes Linux

conventionnels.

La conception d'un système basé sur le protocole ONC débute par la spécification d'un service via un langage de définition d'interface dont la syntaxe est inspirée de celle du C. Un serveur peut offrir plusieurs versions d'un même service, ce qui permet d'assurer une transition en douceur en supportant d'anciens et de nouveaux clients. Dans un service, plusieurs procédures peuvent être spécifiées, chacune étant associée à un numéro unique pour cette version du service. Finalement, chaque procédure spécifie le type de chaque argument qu'elle accepte ainsi que le type de la valeur qu'elle retourne. Une procédure peut donc être identifiée de façon unique à l'aide du tuple (service, version, procédure). L'outil `rpcgen` est utilisé pour générer une bibliothèque de fonctions pour les clients et le serveur, et optionnellement des exemples de code pour chacun.

L'appel d'une procédure à distance se fait grâce à un simple échange de messages `appel` et `réponse` entre le client et le serveur. Le client envoie d'abord un message de type `appel` qui débute avec un numéro de transaction unique. Sa valeur est déterminée par le client et n'a pas de signification, mais elle doit changer pour chaque appel. Les champs `prog` (le numéro du service), `vers` (le numéro de version) et `proc` (le numéro de procédure) suivent et permettent au serveur d'identifier la procédure à appeler. Finalement, les paramètres spécifiques à la fonction appelée sont empilés à la fin du message. Si les informations passées sont valides, le serveur exécute la fonction associée au tuple (service, version, procédure). Dans son message "réponse", le serveur copie le numéro de transaction envoyé par le client et y ajoute la valeur retournée par la fonction. Le numéro de transaction permet au client d'associer une réponse à une requête précédente et au serveur d'identifier les possibles retransmissions (Srinivasan (1995a)).

Toutes les valeurs passées dans les messages sont encodées dans le format eXternal Data Representation (XDR) décrit dans le RFC 1832 (Srinivasan (1995b)). Ce format permet la communication transparente entre des systèmes où l'ordre des octets dans un mot et la longueur des mots diffèrent.

2.1.2 XML-RPC

XML-RPC est un protocole d'appel de procédures à distance simple qui a été bâti sur des technologies déjà très répandues, telles que le format XML et le protocole d'application HTTP. Un document XML encode les informations sur la procédure à appeler du côté du serveur et les arguments passés (leurs formats et valeurs). Le résultat de l'appel est également transmis dans un document XML simple, qui peut d'ailleurs contenir un nombre variable de valeurs. L'implémentation étudiée ici est celle de la bibliothèque xmlrpc-c (Kidd, 2001).

L'utilisation de la bibliothèque ne nécessite pas l'invocation d'un préprocesseur pour gé-

nérer une quelconque base de code. Faute d'avoir un pré-traitement, tout se fait à l'exécution. Le serveur enregistre des fonctions de rappel associées à des noms de méthodes. La bibliothèque s'occupe de faire le décodage du format XML et d'appeler la fonction appropriée pour chaque requête. Elle fournit des fonctions, pour décoder et obtenir les valeurs des paramètres, auxquelles on doit passer une chaîne de formatage indiquant de quel type sont les paramètres attendus ainsi qu'une série de pointeurs correspondants. Cette méthode rappelle le fonctionnement de la famille de fonction scanf de la bibliothèque standard C. L'envoi de la réponse se fait grâce à une fonction dont l'utilisation rappelle quand à elle celle de printf, à qui on passe une chaîne de formatage et les valeurs correspondantes (Kidd, 2001).

L'avantage de réutiliser des technologies existantes est la facilité d'implémentation (par la réutilisation de bibliothèques XML et HTTP) et de déploiement (les infrastructures réseaux sont déjà configurées pour transmettre des flux HTTP). De plus, l'utilisation du protocole HTTP permet la l'utilisation des méthodes d'authentification et de chiffrement existantes. Par contre, l'utilisation de ces technologies vient avec son lot de désavantages, principalement en ce qui a trait à la performance. Les protocoles XML et HTTP contribuent tous deux à augmenter la proportion de la taille des métadonnées par rapport à la taille totale des messages, ce qui affecte l'efficacité du protocole en termes de données transmises (St. Laurent, 2001). Finalement, le choix d'une approche dynamique pour l'encodage et le décodage des paramètres vient avec son coût par rapport à une technique où des fonctions spécifiques sont pré-générées et compilées.

2.1.3 D-Bus

D-Bus est un système de communication permettant de faire interagir plusieurs applications via un démon intermédiaire (appelé bus) qui s'occupe de la transmission des messages. Une application s'enregistre sur un bus en fournissant un nom unique ainsi que la liste des éléments qu'il exporte. Ces éléments peuvent être des méthodes, des propriétés ou des signaux. Une méthode correspond à un appel à distance classique, possiblement avec plusieurs paramètres et valeurs de retour, que peut invoquer un programme tiers. Une propriété est une simple valeur rendue accessible aux client pouvant être en lecture/écriture ou en lecture seule. Un signal, quant à lui, est un message unidirectionnel émis par le serveur et distribué aux clients s'étant enregistrés à ce type de signal (Freedesktop.org, 2014a). D-Bus est couramment utilisé sur les systèmes Linux afin de permettre aux applications de communiquer pour toutes sortes de raisons, allant de l'apparition d'un nouveau périphérique à la requête de l'état d'un lecteur de musique.

Une implémentation de référence de bas niveau est disponible, mais il est indiqué dans sa documentation qu'il n'est pas conseillé de l'utiliser directement dans les programmes clients

(Freedesktop.org, 2014b). Il est plutôt conseillé d'utiliser un des nombreux API de plus haut niveau offerts par diverses bibliothèques (Freedesktop.org, 2013). Une implémentation relativement simple est GDBus, disponible dans la bibliothèquel GIO du projet GNOME (2012). L'interface du serveur est décrite en format XML et une base de code facilitant l'implémentation de clients et serveurs est générée à l'aide de l'outil gdbus-codegen. Du côté serveur, le programme doit effectuer manuellement son enregistrement au bus, puis peut exporter son interface à l'aide d'une des fonctions générées par l'outil. Il doit également enregistrer des fonctions de rappel qui implémentent les méthodes présentes dans son interface. De son côté, le client peut simplement utiliser une fonction générée pour effectuer un appel de méthode au serveur.

2.1.4 Microsoft COM

Le Component Object Model (COM) de Microsoft est un système permettant à différents composants logiciels sur un même ordinateur de communiquer entre eux (Microsoft, 2013h). Un programme peut appeler des méthodes sur des objets pouvant exister dans son processus ou dans un autre processus. La définition des méthodes exportées est faite par l'entremise du Microsoft Interface Definition Language (MIDL) (Microsoft, 2013d). Un compilateur est fourni pour générer les bases de code des clients et du serveur. Les fonctions disponibles aux clients ne sont que des façades chargées d'effectuer l'appel à distance sur le serveur. Distributed COM (DCOM) est une extension de COM permettant l'appel de méthodes sur des objets se trouvant sur d'autres hôtes (Microsoft, 2013a).

2.1.5 Microsoft WCF

Windows Communication Framework (WCF) est le composant de la plate-forme .NET offrant divers services de communication entre les programmes, dont des messages à sens unique et des messages requérant une réponse, essentiellement des appels à distance (Microsoft, 2012a). À l'aide d'annotations spéciales en C++, C# ou Visual Basic (Microsoft, 2010b), le programme serveur peut définir une interface. Plus spécifiquement, le serveur définit la signature des opérations exportées dans un contrat de service qui sera communiqué aux clients. Le serveur doit ensuite définir une classe implémentant cette interface avec le code désiré pour chaque méthode. La façon la plus simple de créer un client est d'utiliser l'utilitaire svcutil.exe (Microsoft, 2010a), qui télécharge le contrat de service existant pour connaitre son interface puis génère un modèle de code correspondant. Encore une fois, l'interface utilisée par le client pour faire les appels à distance imite celle du serveur, mais contient plutôt le code nécessaire pour effectuer la communication avec celui-ci.

2.2 Méthodes d'analyse du comportement des programmes

2.2.1 Débogage

Un débogueur interactif est un outil permettant au développeur logiciel d'interrompre l'exécution de son logiciel afin de pouvoir en inspecter son état à différents points (Agans, 2002). Certains débogueurs permettent l'inspection du programme au niveau des instructions machine, permettant à l'utilisateur d'exécuter une instruction assembleur à la fois et d'inspecter la valeur des registres et de la mémoire. À un niveau d'abstraction supérieur, on trouve la classe de débogueurs de niveau source qui permettent le débogage de programmes écrits en langages de plus haut niveau. Avec un tel outil, l'utilisateur peut exécuter son programme pas à pas, une ligne de code source à la fois, sans avoir besoin de comprendre ou de connaitre les détails de l'exécution au niveau du processeur. Lorsque le programme est interrompu, le débogueur peut lire la valeur des variables et produire d'autres informations, qui ont un sens uniquement dans le contexte du langage, telles que la pile d'appel. Puisque, pour offrir ces fonctions, le débogueur de niveau source doit connaitre les détails intimes de l'architecture sous-jacente, il offre généralement aussi les fonctions du débogueur de niveau machine.

Exécuter l'entièreté de son programme en mode pas-à-pas pour le déboguer devient rapidement impossible, dès que celui-ci prend moindrement de l'ampleur. La plupart des débogueurs permettent de définir des points d'arrêt à différents endroits d'intérêt dans le programme, par exemple juste avant un bogue suspecté. Le développeur peut alors lancer l'exécution normale de son programme, sachant qu'il s'interrompra si l'exécution croise un de ces points d'arrêt.

Le débogage est une méthode utile à la résolution d'une bonne partie des problèmes rencontrés par un développeur logiciel. Certaines circonstances restreignent cependant l'utilisation du débogage. Comme un processus arrêté sur un point d'arrêt est essentiellement en pause, les programmes qui intéragissent avec le monde extérieur, qui ont des contraintes temps réel avec des bogues qui ne se manifestent qu'en situation de charge intensive peuvent se comporter différemment lorsqu'ils sont sous la loupe d'un débogueur (Grotker *et al.*, 2008). Pour cette classe de programmes, le traçage peut s'avérer être un outil plus approprié.

Dans les sous-sections suivantes, nous recensons les débogueurs principaux permettant au minimum le débogage de programmes en C/C++. S'il y a lieu, nous décrivons les fonctions assistant au débogage de programmes utilisant les appels à distance.

GDB

Le GNU Debugger (GDB) est est débogueur de choix de la suite d'outils GNU, dont le compilateur GCC fait également partie (Free Software Foundation (2014d)). GDB supporte plusieurs langages, dont le C/C++ et une multitude de systèmes d'exploitation, architectures de processeurs et de formats d'exécutables.

GDB peut fonctionner en mode natif, où il supervise un ou plusieurs processus directement sur le système où il s'exécute lui-même, mais peut aussi fonctionner en mode distant, où il contrôle des processus par le biais d'un serveur de dégogage. En mode natif, GDB est disponible pour la plupart des systèmes descendants de Unix et sur Microsoft Windows. Sur Linux, la plate-forme étudiée, GDB utilise l'API ptrace pour contrôler et inspecter l'état des autres processus. Parmi les opérations rendues possibles via cet API, on retrouve la lecture des registres, de la mémoire, le redémarrage de l'exécution et l'exécution d'une seule instruction. Pour le débogage à distance, GDB utilise le Remote Serial Protocol (RSP), un protocole servant à la transmission des commandes (insertion ou retrait d'un point d'arrêt, interruption ou continuation d'un processus, etc) et des évènements liés à l'état des processus débogués. Le programme gdbserver peut être utilisé comme serveur de débogage sur les plate-formes supportées (sur Linux, par exemple), mais plusieurs plate-formes exemptes de système d'exploitation (comme les systèmes embarqués) doivent implémenter elles-mêmes la partie serveur. Des programmes de simulation, comme QEMU (2013), présentent une interface implémentant le protocole RSP pour interagir avec la plate-forme simulée.

GDB permet de déboguer plusieurs fils d'exécution et même plusieurs programmes simultanément. Dans le mode par défaut, all-stop, lorsqu'un fil frappe un point d'arrêt, tous les autres fils connus de GDB sont suspendus, permettant leur inspection. Lorsque le mode alternatif (non-stop) est activé, GDB suspend uniquement le fil qui frappe le point d'arrêt et laisse les autres s'exécuter (Free Software Foundation, 2014a).

L'interface utilisateur de base de GDB étant orientée vers la facilité d'utilisation pour un humain, il est difficile pour un autre programme (une interface graphique par exemple) de l'utiliser pour contrôler le débogueur. À cette fin, GDB possède une interface machine, via laquelle un autre programme peut envoyer des commandes à GDB et obtenir de l'information sur les processus débogués en un format plus approprié (Free Software Foundation, 2014c).

Une interface de programmation en Python permet de modifier le comportement de GDB ou d'y ajouter des fonctions sans avoir à modifier son code source. Il est par exemple possible de créer de nouvelles commandes, d'ajouter des point d'arrêt et de définir des fonctions de rappel lorsque ces points d'arrêt sont atteints (Free Software Foundation, 2014b).

LLDB

LLDB est le débogueur associé au projet LLVM (LLDB Project, 2014c). Il est à code source ouvert et est distribué sous la licence LLVM. Il supporte le débogage symbolique des progammes en C, C++ et Objective-C. En principe, LLDB fonctionne sur les principaux descendants d'Unix, dont Linux, mais l'ensemble des fonctions disponibles sur Mac OS X est plus grand et plus stable que celui sur les autres plate-formes (LLDB Project, 2014b). De façon analogue à gdbserver, le programme debugserver permet d'effectuer du débogage à distance.

LLDB possède aussi une interface Python permettant d'automatiser certaines tâches ou d'étendre sa fonctionnalité (LLDB Project, 2014a). D'abord, le débogueur peut être démarré et contrôlé de façon programmatique à partir d'un programme Python. Ensuite, comme pour GDB, son comportement peut être modifié à l'aide de cette interface. On peut ajouter de nouvelles commandes, ajouter des points d'arrêt et réagir lorsque l'exécution en croise un.

IDB

Le Intel Debugger (IDB) est le débogueur développé par Intel et distribué sous licence propriétaire au sein de plusieurs de ses produits (Intel, 2012). Le débogueur en soi est une application en ligne de commande, mais il vient avec une interface graphique basée sur Eclipse. Il est toutefois mentionné à plusieurs endroits dans la documentation d'Intel (Intel, 2013b) que le débogueur IDB est en voie d'être discontinué. Intel mise plutôt sur GDB auquel sont ajoutées quelques extensions (Intel, 2013a).

Microsoft Visual Studio Debugger

Le débogueur distribué par Microsoft avec l'environnement Visual Studio permet de déboguer des programmes dans tous les langages que Visual Studio supporte (Microsoft, 2013b).

Le débogueur de Visual Studio possède des fonctions assistant au débogage des appels de procédure à distance basés sur COM ou WCF. En déboguant pas à pas un programme faisant un appel à distance, il est possible de faire le lien entre le code du client et le code du serveur (Microsoft, 2013f,e). Du point de vue de l'utilisateur, il suffit de faire un pas au moment d'entrer dans la fonction constituant un appel à distance. Au lieu d'entrer dans le code implémentant l'appel à distance à bas niveau, le débogueur s'arrêtera au début de l'implémentation dans le serveur. Lorsque la fonction du serveur se termine, le débogueur revient au site initial de l'appel à distance, dans le client. Afin du montrer le fil d'exécution logique du programme, la vue de la pile d'appels montre les appels du client et du serveurs

de façon combinée (Figure 2.1). Dans certaines situations, le débogueur est capable de déterminer quel est le processus du serveur et de s'y attacher automatiquement, si nécessaire (Microsoft, 2013g).

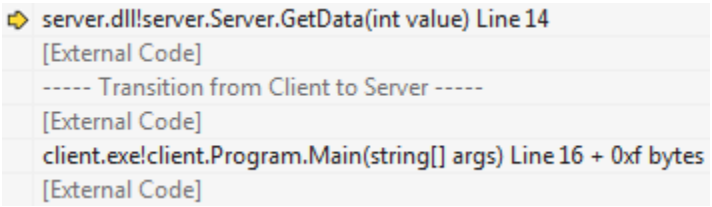

Figure 2.1 Visual Studio : vue de la pile d'appels distribuée lors d'un appel à distance

2.2.2 Traçage

Tel que mentionné précédemment, le déboguage interactif ne s'applique pas à toutes les situations. Le traçage est une méthode de collecte d'information similaire à la journalisation d'évènements, mais qui vise des cas d'utilisation différents. Un développeur peut insérer des points de trace directement dans la source d'un programme (traçage statique) ou en modifiant le binaire exécuté pendant l'exécution (traçage dynamique). Lorsque l'exécution atteint un point de trace, un évènement comprenant une estampille temporelle et quelques données jugées utiles est enregistré dans un fichier, appelé simplement une trace. Le traçage vise à minimiser la surcharge sur le système étudié, tout en permettant l'extraction d'un important débit d'information utile au déboguage du système. Contrairement au débogage, l'outil d'analyse ne bloque jamais l'exécution du programme. Si le programme génère plus de données que le traceur peut enregistrer pendant une certaine période, les données seront perdues. L'analyse se fait généralement a posteriori, mais il est aussi possible de la faire en temps réel.

Dans le cas d'un traceur statique, les instructions de traçage ajoutées dans le code résultent en des instructions présentes de façon permanente dans l'exécutable. Le traçage peut tout de même être activé ou désactivé, auquel cas l'exécution sautera par dessus les points de trace, mais ils ne peuvent pas être modifiés. Un traceur dynamique va quant à lui modifier le code de l'application pendant qu'elle est en marche pour injecter le code nécessaire au traçage. Les points de trace (leur emplacement et les informations récoltées) peuvent donc être modifiés sans recompilation et même sans redémarrer l'application.

LTTng

Le traceur LTTng prend ses origines au début des années 2000, alors que Karim J. Yaghmour crée la première version de LTT, le Linux Trace Toolkit (Opersys, 2012). Le développement a par la suite été repris et par Desnoyers et Dagenais (2006), qui ont continué d'apporter des améliorations au traceur, dorénavant connu sous le nom de LTTng 0.x. En 2012, une nouvelle version majeure, LTTng 2.x, a été rendue disponible (Desnoyers *et al.*, 2012). Elle offre notamment une interface unifiée pour le traçage noyau et utilisateur, la sauvegarde de traces sur le réseau et l'utilisation d'un format de trace flexible et extensible, le Common Trace Format (CTF).

Pour le traçage noyau, LTTng est comptible avec trois systèmes de points de trace présents dans le noyau.

tracepoints Ce sont des points de trace statiques insérés à des endroits intéressants par les développeurs noyaux. Ces points de trace peuvent être activés et désactivés pendant l'exécution du noyau (Desnoyers, 2008).

kprobes Il s'agit d'un système permettant d'insérer un point de trace pratiquement à n'importe quelle adresse exécutable du noyau. L'instruction à cet endroit est remplacée par un point d'arrêt ou un saut inconditionnel (selon la configuration) et un segment de code supplémentaire peut être exécuté. L'instruction originale est quant à elle exécutée hors-séquence (Keniston *et al.*, 2006).

function tracing Cette technique exploite l'infrastructure existante servant au profilage. Si le support pour le profilage est activé lors de la compilation du noyau, un espace de quelques octets est laissé libre au début de chaque fonction pour permettre l'installation d'un appel à une fonction de profilage. Les outils de traçage peuvent se servir de cet espace pour placer un point de trace au début de n'importe quelle fonction.

.

Pour le traçage en mode utilisateur, l'installation de points de trace statiques est pour l'instant requise (Desnoyers, 2012).

ftrace

ftrace est un traceur noyau inclus dans la distribution officielle de celui-ci depuis la version 2.6.27 (KernelNewbies, 2008). ftrace peut obtenir ses informations des mêmes sources que LTTng, soit des points de trace statiques, des kprobes et du traçage de fonction. Des modules permettent par exemple l'analyse de latence (le temps pendant lequel les interruptions sont

désactivées sur un processeur ou le temps requis pour exécuter un processus après qu'il se soit fait réveiller) ou la génération d'un graphe d'appel. Ces différentes analyses rendent ce traceur très intéressant pour les développeurs noyaux qui désirent observer une portion du noyau en particulier.

Systemtap

Systemtap est un traceur noyau offrant d'intéressantes fonctions de traçage dynamique (SystemTap, 2014). Il est fondé sur les même trois technologies que les autres traceurs. L'avantage de Systemtap est la possibilité de définir des scripts à exécuter lorsqu'un point de trace est rencontré. L'utilisateur peut donc effectuer pratiquement n'importe quelle action, vérifer des conditions et même modifier le contenu des variables dans le contexte du point de trace. Cependant, cette flexibilité vient à un coût en performance. Systemtap est reconnu pour avoir une mise à l'échelle peu optimale (Desfossez et Desnoyers, 2011).

2.3 Architectures à grand nombre de cœurs

Les technologies de fabrication actuelles des processeurs atteignent leur limite, ce qui ralentit la croissance de la performance que les fabricants sont capables d'aller chercher pour un seul cœur (Vajda, 2011b). Les processeurs multi-cœurs, apparus il y a déjà plusieurs années, sont une façon de pallier à ce problème en permettant plusieurs fils d'exécution simultanés. Les processeurs à usage général qu'on retrouve sur le marché des consommateurs et des serveurs ont de l'ordre d'une dizaine de cœurs physiques, au maximum. Toutefois, afin de répondre à un besoin dans certaines niches de marché (calcul de haute performance, serveurs pour certaines applications), des manufacturiers poussent le concept à l'extrême et développent des processeurs à grand nombre de cœurs, comprenant plusieurs dizaines de processeurs physiques, chacun pouvant être exposé comme plusieurs processeurs logiques.

Ces processeurs sont destinés aux applications et aux problèmes qui peuvent être parallélisés de façon extrême. Le nombre important de fils qui s'exécutent de façon simultanée pose des défis autant au niveau du débogage que du traçage.

Tel que mentionné plus haut, le débogage classique est une méthode parfois trop intrusive. Un problème dans une application comprenant plusieurs centaines de fils d'exécution peut être liée à une mauvaise interaction entre ces fils. En arrêtant complètement un fil avec le débogueur, il se peut que le comportement fautif soit éliminé (Vajda, 2011a). Un exemple de situation plausible est un programme qui utilise la technique du vol de tâches pour équilibrer la charge de travail entre les fils. Le temps que l'utilisateur inspecte l'état d'un fil arrêté, les autres complètent toutes les autres parts de travail, éliminant du coup l'interaction

problématique avec le fil observé.

Pour le traçage, le nombre élevé de cœurs peut compromettre l'atteinte d'un objectif principal du traçage, soit son caractère non-intrusif. D'un côté, si un traceur utilise des techniques de verrouillage et d'exclusion mutuelle pour l'enregistrement des évènements provenant de fils différents, on risque d'observer un phénomène de contention et une perte de performance importante. Pour éviter ce problème, si un traceur utilise des structure de données différentes pour chaque cœur pour les tampons et structures de contrôles, il risque d'augmenter l'utilisation de mémoire au delà d'un seuil acceptable. Autrement, comme chaque cœur est un producteur d'évènements, la quantité d'information générée par plusieurs centaines de fils s'exécutant de manière simultanée peut mettre une pression indue sur le système de mémoire où la trace doit être enregistrée.

Dans la section qui suit, nous nous intéressons à quelques architectures à grand nombre de cœurs.

2.3.1 Tilera Tile-Gx

La série de processeurs Tile-Gx de Tilera est principalement destinée à des applications réseau, multimédia et infonuagique diverses (Tilera, 2014). Il en existe actuellement des versions allant de 9 à 72 cœurs. Le processeur est disponible entre autres via la carte de développement TILEncore-Gx36 pouvant s'installer dans la fente PCI express d'une station de travail, permettant la communication entre la carte et l'hôte. Tel qu'illustré sur la figure 2.2, le processeur possède plusieurs sous-modules ou périphériques d'accélération matérielle (chiffrement et compression) et de communication (PCI, USB, Ethernet, ...) et deux contrôleurs mémoire DDR3. Physiquement, les cœurs, nommés tuiles, sont disposés en carrelage et sont reliés à leurs voisins par un réseau à faible latence et grand débit. Ce réseau est en fait constitué de plusieurs réseaux parallèles servant différentes fonctions. Parmi ceux-ci, deux réseau sont dédiés à la communication avec les contrôleurs de mémoire et les périphériques qui sont connectés en périphérie de l'échiquier. Un réseau, le User Data Network, est quant à lui disponible pour utilisation par le code qui s'exécute sur le processeur à l'aide d'instructions spécialisées (Tilera, 2012c).

Chaque cœur possède lui-même des caches de niveau 1 et 2. Afin d'implémenter une sorte de cache de niveau 3 efficace avec un grand nombre de processeurs, Tilera a intégré un système de cache distribué. Chaque adresse de mémoire physique possède une tuile maison. Si un cœur désire lire une valeur ne se trouvant pas dans sa propre mémoire cache, il peut envoyer une requête à la cache de la tuile à laquelle appartient l'adresse lue. Si celle-ci ne possède pas la valeur, une requète est dirigée vers la mémoire centrale (Tilera, 2012a).

Le système d'exploitation Linux peut être utilisé sur le processeur Tile-Gx. Des outils de

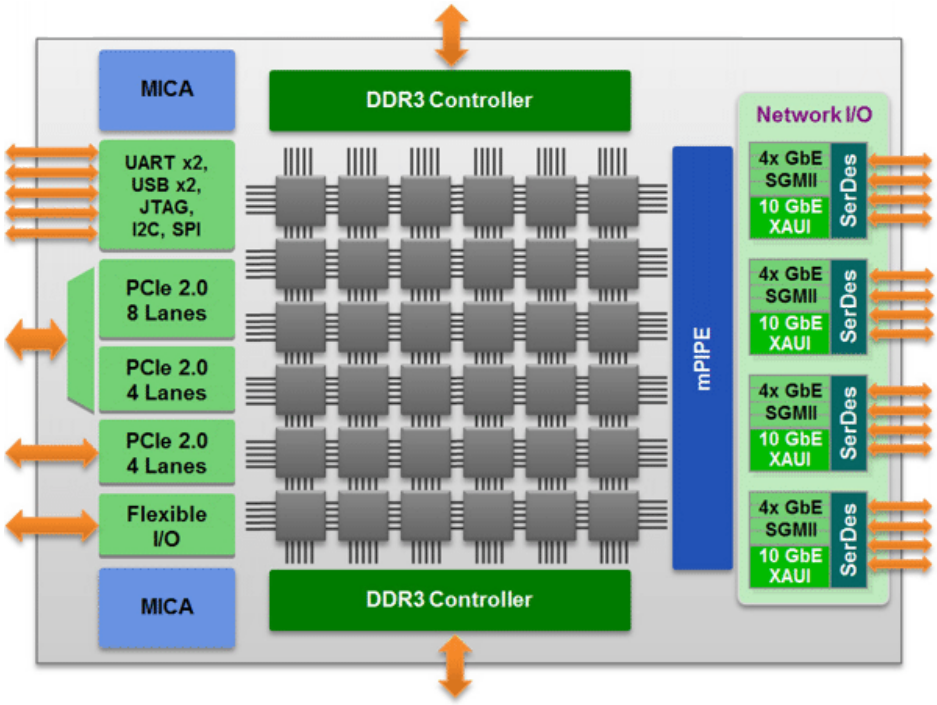

Figure 2.2 Architecture du Tilera Tile-Gx36 (Tilera, 2012b)

développement standards, tels que le compilateur GCC, le débogueur GDB et la bibliothèque GNU C ont été portées à cette architecture.

En 2011, Facebook a publié les résultats de tests démontrant l'efficacité du processeur TILEPro64 (le parent direct du TILE-Gx) exécutant memcached, une application de cache distribué. Le processeur de Tilera permettait de servir davantage de requêtes par watt que des plate-formes x86 de serveurs typiques (Berezecki *et al.*, 2011).

2.3.2 Intel Xeon Phi

Le Xeon Phi est un processeur d'Intel conçu pour les applications de calcul parallèle dont l'architecture est dérivée de l'architecture x86-64. Il ne possède pas de registres MMX, SSE ou AVX, mais possède plutôt son propre type de registres vectoriels de 512 bits. Selon le modèle, les processeurs peuvent compter de 57 à 61 cœurs, tous étant dotés de quatre cœurs logiques (technologie HyperThread), pour un total de 228 à 244 processeurs logiques (Intel, 2013).

Comme on peut le voir sur la figure 2.3, les cœurs du Xeon Phi sont reliés par un bus bi-directionnel auquel est également connecté plusieurs contrôleurs mémoire et un contrôleur PCI express (Intel, 2012). Comme pour le Tilera Tile-Gx, lors d'une faute de cache sur un cœur, celui-ci peut envoyer une requête de lecture à un autre cœur qui possède possiblement

cette donnée.

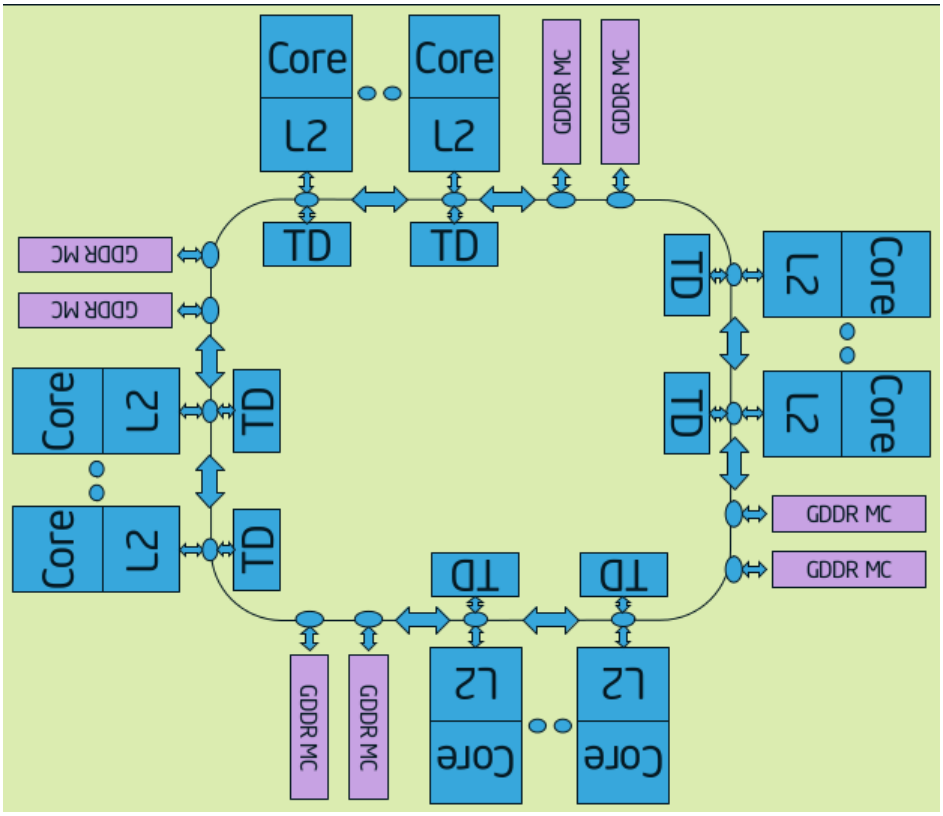

Figure 2.3 Architecture de l'Intel Xeon Phi (Intel, 2012)

Pour le moment, le processeur est uniquement disponible sous le format de cartes d'expansion PCI-express destinées à servir de coprocesseurs attachés à des systèmes conventionnels. La carte est elle-même un système complet sur lequel s'exécute un système Linux. Intel fournit une version du compilateur GCC qui est utilisé notamment pour compiler le noyau Linux. Toutefois, pour produire un code réellement optimisé pour le Xeon Phi (déroulage de boucles, utilisation des registres vectoriels), le compilateur ICC doit être utilisé. Intel fournit également les bibliothèque Math Kernel Library (MKL) et Vector Math Library (VML), qui contiennent des primitives écrites en assembleur extrêmement optimisées pour l'architecture du Xeon Phi. Les bibliothèques OpenMP, Thread Building Blocks (TBB) ou MPI peuvent être utilisées pour tirer profit du parallélisme offert par le Xeon Phi, comme sur les systèmes plus standards. Cependant, le fait que des applications parallèles existantes puissent être facilement adaptées au Xeon Phi ne garantit pas qu'elles seront performantes. Masci (2013) résume bien les points à surveiller lorsqu'on programme pour cette plate-forme. Les charges de travail destinées au Xeon Phi devraient contenir une part de calcul très demandante mais avec une utilisation de bande passante de mémoire faible. Une attention toute

particulière devait être apportée à l'utilisation de la mémoire cache et donc à la localité des accès mémoire. Bien que le compilateur soit capable de dérouler des boucles automatiquement, il se peut qu'il en omette à cause de certaines fausses dépendances apparentes entre les itérations. Le programmeur peut forcer la vectorisation de boucles à l'aide de pragmas. Mais surtout, Masci met une emphase particulière sur l'utilisation des bibliothèques MKL et VML fournies par Intel. Ces bibliothèques ayant fait l'objet d'énormément d'optimisation, le programmeur a tout intérêt à les utiliser, sachant qu'il est pratiquement impossible de les battre en performance.

Cramer *et al.* (2012) ont démontré que le Xeon Phi est une plate-forme tout à fait adaptée pour exécuter efficacement des applications de calcul mathématique très intense. La plupart des applications existantes pourront être exécutées sans grande modification, surtout si elles font appel à des bibliothèques telle que OpenMP pour la parallélisation.

De bons exemples de charges de travail adaptées au Xeon Phi sont les algorithmes de Monte Carlo, puisqu'ils incorporent généralement beaucoup de calcul pour peu de transfert de données (Li, 2011).

La seule façon dont le système peut communiquer avec le monde extérieur est avec l'hôte, via le lien PCI. Sur ce lien PCI, un lien réseau virtuel est établi entre l'hôte et la carte. Deux modes d'opération principaux sont donc proposés par Intel : natif et délestage. Le mode natif est similaire au fonctionnement d'une plate-forme embarquée classique : un compilateur croisé est utilisé pour générer un exécutable destiné directement au Xeon Phi. En démarrant un interpréteur de commandes, via SSH, il est possible de transférer le binaire et de l'exécuter comme sur un système normal. Le mode délestage, uniquement supporté par le compilateur ICC, consiste à choisir certaines fonctions qui seront spécifiquement exécutées sur le Phi. Le développeur désigne ces fonctions à l'aide de directives pragma. Le compilateur génère alors un exécutable hybride s'exécutant sur l'hôte, mais contenant des sections de code exécutable sur la carte. Au moment de l'exécution, une bibliothèque spécialisée se charge de transférer le code et les arguments des fonctions désignées sur le Xeon Phi (Newburn *et al.*, 2013). Un processus est lancé sur le Phi pour exécuter le travail délesté pendant que le programme principal est en attente. Une fois la fonction délestée terminée, la valeur de retour est transmise au programme sur l'hôte qui peut reprendre son exécution. Le mode délestage permet d'intégrer plus facilement le Xeon Phi dans un système de traitement complexe.

Initialement, Intel fournissait le débogueur IDB pour le débogage sur Xeon Phi. Intel supporte et conseille maintenant l'utilisation de GDB. Plusieurs extensions sont fournies, permettant d'améliorer l'expérience de débogage. Il est possible d'inspecter les registres vectoriels du Xeon Phi ainsi que d'analyser le travail fait avec OpenMP. De plus, GDB pour Xeon Phi est bien intégré dans les outils de développement C/C++ d'Eclipse (CDT),

et permet même le débogage des sections de code délestées de l'hôte vers la carte (Intel, 2014). En effet, en avançant pas-à-pas, le débogueur peut passer de façon transparente de l'exécution sur l'hôte à l'exécution sur la carte. En entrant dans une section délestée, en plus du processus de délestage lancé pour exécuter le code voulu, un agent de débogage est lancé. Le débogueur s'y attache automatiquement de façon distante et commence le débogage du processus sur la carte.

Le débogueur TotalView de RogueWave supporte également le débogage du Xeon Phi. Il permet lui aussi de déboguer les structures de délestage (Software, 2012).

CHAPITRE 3

CONCEPTS

Dans ce premier chapitre, nous décrivons les améliorations qui ont été apportées à GDB dans les quelques dernières années et sur lesquelles nous nous basons pour effectuer notre travail.

En quelques années, GDB est passé d'un débogueur supportant un seul processus, en mode "tout-arrêt", à un débogueur multi-processus comportant plusieurs modes d'interaction entre les différents processus. Si GDB était auparavant assez monolithique, sans réelle possibilité d'extension, il est aujourd'hui possible d'augmenter ou modifier facilement ses fonctions grâce à un API Python très complet.

3.1 Modes asynchrones et sans-arrêt

Cette section décrit les modes asynchrones (target-async) et sans-arrêt (non-stop) de GDB, qui sont des fondations essentielles à la mise en œuvre du débogage d'appels de procédure à distance. Ces modes et leur histoire est principalement décrite par Sidwell *et al.* (2008). Ces deux modes sont dits orthogonaux, mais sont souvent utilisés ensemble lors du débogage de systèmes parallèles.

Lorsque GDB ne supportait qu'un seul processus à la fois, on retrouvait la propriété suivante : soit GDB est actif et le processus débogué est en attente, soit l'inverse. GDB et le processus ne sont jamais actif en même temps. Les commandes qui lançaient l'exécution du processus bloquaient l'invite de commande jusqu'à la prochaine interruption du programme (point d'arrêt, interruption de l'utilisateur avec ctrl-C, etc.). Ceci empêchait l'utilisateur d'entrer une quelconque commande lors de l'exécution du processus débogué. Lorsque les développeurs de GDB ont ajouté le support multi-processus, il a également fallu ajouter un mode d'entrée utilisateur asynchrone. Autrement, si l'invite de commande est bloquée après avoir lancé le premier processus, il est impossible d'en lancer un second et d'exécuter les deux processus en même temps. Le mode asynchrone permet à l'utilisateur de lancer une exécution en arrière-plan en suffixant sa commande du caractère esperluette (&), un peu comme les interpréteurs de commandes supportent. Une fois le premier processus lancé, l'utilisateur peut entrer de nouvelles commandes pour démarrer d'autres processus. L'ensemble des commands que l'utilisateur peut utiliser pendant qu'un processus débogué s'exécute est toutefois restreint. Dans le cas de Linux, par exemple, il est impossible de consulter l'état de la mémoire

ou des registres d'un processus actif puisque cela n'est pas permis par ptrace, l'interface de programmation sous-jacente. Ce mode est utile pour les systèmes multi-processus où le développeur désire insérer des points d'arrêt dans plus d'un processus.

Les modifications apportées pour implémenter le mode asynchrone étaient principalement reliées à la boucle principale de gestion d'évènements au cœur de GDB. À l'origine, la boucle principale du débogueur attendait une commande de l'utilisateur sur le descripteur de fichier de l'entrée standard lorsque le processus était inactif, puis attendait un évènement du processus lorsqu'il était actif. La nouvelle boucle devait être capable de réagir aux évènements de l'entrée standard et des processus débogués en même temps. Cette boucle est implémentée à l'aide de l'appel système select, permettant d'attendre que la lecture soit possible sur un descripteur de fichier parmi plusieurs. Sur Linux, un problème vient du fait que la récupération des évènements des processus débogués ne vient pas sous forme d'une lecture sur un descripteur de fichier, mais du résultat d'un appel à une fonction de la famille wait. Étant un programme à un seul fil d'exécution, GDB ne peut évidemment pas bloquer sur les deux appels en même temps. Toutefois, lorsqu'un évènement survient dans un des processus enfants, le noyau envoie à GDB le signal SIGCHILD. Au démarrage, GDB crée donc un tuyau de communication dont il passe l'extrémité en lecture à l'appel à select dans sa boucle principale. À la réception d'un signal SIGCHILD, le gestionnaire du signal écrit un octet arbitraire à l'entrée du tuyau, permettant de débloquer la boucle principale et de réagir convenablement à l'évènement. Dans le cas du débogage à distance, la communication est effectuée par une interface de communication réseau (socket) ou une autre forme de descripteur de fichier, donc il n'y a pas de problème.

De base, avec le mode asynchrone, l'interruption d'un des fils d'exécution causait l'interruption de tous les fils d'exécution connus de GDB. Pour certaines situations, ce comportement est préférable, comme lorsque l'exécution de différents fils doit se faire de façon synchronisée. En les arrêtant et redémarrant tous simultanément, on s'assure de préserver leur temps d'exécution relatifs. Dans d'autres situations, on veut minimiser l'intrusion du débogage dans le système et éviter d'arrêter tous les fils d'exécution lorsque l'un d'eux est interrompu. Par exemple, en présence d'un fil responsable de répondre à une horloge de surveillance, l'arrêt de tous les fils rendrait le système inutilisable lors du débogage. C'est pour cette raison que le mode sans-arrêt a été développé. Par opposition, le mode original a été nommé tout-arrêt (all-stop).

L'implémentation du mode sans-arrêt a requis une refonte en profondeur de l'architecture de GDB. Lorsqu'un point d'arrêt est inséré, une instruction doit être écrasée par le code d'opération correspondant au point d'arrêt. Dans l'implémentation originale de GDB (le mode tout-arrêt), les points d'arrêt sont enlevés du processus lorsqu'il est interrompu. Cela

simplifie notamment les lectures de la mémoire du processus débogué, puisque GDB n'a pas besoin de considérer la possible présence de points d'arrêt dans la zone de mémoire lue. Pour le mode sans-arrêt, il faut toutefois que les points d'arrêt restent insérés en permanence, question que les autres fils du même processus puissent s'y arrêter. Pour chaque lecture de mémoire, GDB doit donc s'assurer de remplacer dans le tampon de lecture les instructions de points d'arrêt par le contenu original à ces endroits.

Une autre situation problématique est maintenant possible : un évènement correspondant à l'arrêt d'un fil sur un point d'arrêt est mis en file en vue d'être traité par la boucle principale. Avant d'être traité (par exemple, pendant le traitement d'un évènement précédent dans la file), le point d'arrêt correspondant est supprimé. Lorsque notre évènement original se fait finalement traiter, GDB ne détectera pas que le déroutement (trap) subi par le programme correspond à un point d'arrêt qu'il a lui même inséré. Cela causera un plantage du processus débogué, chose qui ne sera pas favorablement accueillie par l'utilisateur. Pour y remédier, GDB garde une liste des points d'arrêt défunts, c'est-à-dire ceux qui ont été supprimés mais pour lesquels un évènement non-traité pourrait toujours exister.

Le fait que les points d'arrêt doivent rester insérés en permanence amène un autre problème particulièrement difficile, celui de l'exécution en mode pas-à-pas d'un fil à partir de la position d'un point d'arrêt. En mode tout-arrêt, si un fil s'arrête à un point d'arrêt et que tous les points d'arrêt sont retirés du programme, GDB peut sans problème effectuer une exécution pas-à-pas de l'instruction se trouvant à la position du point ayant causé l'arrêt, puisque le contenu original de la mémoire a été momentanément restauré. Si l'utilisateur décide de continuer l'exécution normalement au lieu de le faire pas-à-pas, GDB exécute une instruction en mode pas-à-pas (celle qui est remplacée par le point d'arrêt), réinsère l'ensemble des points d'arrêt et relance l'exécution normale. Dans le cas du mode sans-arrêt, cela pose problème, puisque l'instruction originale ne peut être remise en place pour être exécutée.

Deux solutions principales sont connues pour ce problèmes : la simulation des effets de l'instruction ou l'exécution pas-à-pas déplacée. La simulation fonctionnerait bien dans les cas simples, puisque GDB possède déjà la capacité de décoder et de simuler les instructions sur la plupart des architectures. Le débogueur peut donc interpréter l'instruction et modifier les registres et la mémoire du processus pour en refléter les effets. Toutefois, cela devient plus ardu pour les situations spéciales comme la génération de fautes de pages. Les développeurs ont donc adopté la seconde solution, l'exécution pas-à-pas déplacée. Cette technique consiste à copier l'instruction dans une zone mémoire exécutable inutilisée et y exécuter l'instruction. Cela pose un problème pour les instructions agissant par rapport à la valeur du compteur de programme, qui est évidemment différente si l'instruction est exécuté à

un autre endroit. GDB doit donc décoder l'instruction et corriger tout décalage relatif au compteur de programme. D'autres ajustements doivent être faits après l'exécution de l'instruction. Par exemple, s'il s'agissait d'une instruction d'appel de fonction, l'adresse de retour a probablement été placée sur la pile et doit elle aussi être corrigée.

3.2 Points d'arrêt internes

GDB utilise le concept de points d'arrêt internes pour implémenter plusieurs de ses fonctions. En quelques mots, il s'agit d'utiliser le mécanisme des points d'arrêt pour permettre au débogueur de rester informé sur l'état courant du programme étudié. Lorsqu'un point d'arrêt interne est atteint, des actions sont prises par le débogueur (généralement, la collecte d'information) et l'exécution est immédiatement relancée. Du point de vue de l'utilisateur, ce processus est transparent et n'influence pas le déroulement du programme étudié. Dans GDB, les points d'arrêt internes ne sont pas visibles avec la commande `info breakpoints`. On doit utiliser la commande `maintenance info breakpoints` pour les afficher.

GDB utilise cette technique pour résoudre le problème des bibliothèques partagées chargées dans l'espace d'adressage du programme. Lorsqu'une bibliothèque est chargée, il est impossible de prédire l'adresse où l'éditeur de liens dynamique placera le code. Or, GDB a besoin de cette information pour offrir à l'utilisateur la possibilité de placer des points d'arrêt dans le code des bibliothèques ou de naviguer en mode pas-à-pas. Pour y remédier, GDB place des points d'arrêt internes sur certaines fonctions clés de l'éditeur de liens dynamique. Chaque fois qu'une bibliothèque est chargée ou déchargée, un de ces points d'arrêt est frappé et la liste des bibliothèques est mise à jour. Ainsi, lorsque l'utilisateur demande de placer un point d'arrêt dans une fonction d'une bibliothèque partagée, GDB est en mesure de calculer l'adresse absolue de l'instruction à modifier pour le placer. Cette technique est nécessaire autant pour les bibliothèques chargées avant le lancement du programme que celles chargées à l'exécution (généralement par un appel à `dlopen`).

On constate aussi que cette technique est utilisée pour implémenter la commande `finish`. Cette commande permet à l'utilisateur de continuer l'exécution jusqu'à ce que le programme quitte le contexte actual, ce qui correspond généralement à la fin de la fonction actuelle. En analysant la pile du programme, GDB peut retrouver l'endroit où l'exécution devrait revenir une fois la fonction actuelle terminée. GDB y place donc un point d'arrêt interne temporaire et relance l'exécution de façon normale. Au retour de la fonction, le point est frappé et le contrôle est redonnée à l'utilisateur. Il existe toutefois une situation spéciale où l'exécution ne revient pas à l'endroit prévu, lorsque le programme effectue un `longjmp` (incluant la gestion d'exceptions en C++). GDB place donc en permanence un point d'arrêt

pour surveiller les appels à `longjmp`. Si ce point d'arrêt est frappé alors qu'un appel à `finish` est en suspend, c'est que l'exécution quitte le contexte courant de façon indirecte. GDB arrête donc l'exécution du programme et retire le point d'arrêt qu'il avait placé au point de retour prévu.

3.3 Extensions Python

GDB permet depuis longtemps d'associer des commands aux points d'arrêt. Lorsque le point est atteint, ces commandes seront exécutées comme si c'était l'utilisateur qui les entrait à la main. Cela permet déjà un certain degré d'automatisation des tâches répétitives, mais ne permet pas beaucoup l'utilisation de fonctionnalités et d'analyses non présentes dans GDB. L'API Python a été introduit en 2008, mais est en constante évolution depuis ce jour. Cet API permet d'apporter des modifications au comportement de GDB simplement en chargeant un script Python au démarrage du débogueur. Comme ces modifications peuvent par exemple viser à améliorer l'expérience de débogage d'un programme en particulier, il est avantageux de garder le débogueur générique et d'effectuer les modifications spécifiques de façon complètement dynamique. Autrement, une version de GDB devrait être compilée pour chaque modification spécifique.

L'API Python permet de consulter une grande quantité d'information à propos de l'état du débogueur et des processus analysés : liste des processus/fils d'exécution, des points d'arrêt, des fichiers objets, des tables de symboles, etc. Le comportement de GDB peut aussi être altéré de plusieurs façons : ajout de points d'arrêt, de commandes, de fonctions, de paramètres etc. Les sections qui nous intéressent sont les points d'arrêt et les commandes.

Le lien entre le code C de GDB et le code Python des extensions est rendu possible grâce au module Python `gdb`, lui-même écrit en C. C'est ce module qui fait le pont lorsque le code Python doit appeler une fonction en C (par exemple, lors de l'instanciation d'un point d'arrêt à partir du Python) ou vice-versa (par exemple, lors de l'appel de l'implémentation d'une commande écrite en Python). Si le support Python est activé, GDB charge un interpréteur Python dans son espace mémoire au démarrage et importe automatiquement le module `gdb`, par lequel l'utilisateur accèdera aux différentes fonctions. Comme il s'agit de la même instance de l'interpréteur Python tout au long du cycle de vie du débogueur, les composantes implémentées en Python peuvent sauver des informations dans des variables globales Python, et les retrouver lors d'une invocation subséquente.

Pour notre travail, nous nous intéressons d'abord à l'insertion de points d'arrêt. En dérivant de la classe `gdb.Breakpoint`, on peut spécifier où le point devrait être installé et quel code doit être exécuté lorsqu'il est frappé. Cette fonction de rappel, selon sa valeur de retour,

détermine si l'exécution doit être interrompue (comme pour un point d'arrêt standard) ou si elle doit continuer. Il est donc très simple (en moins de 10 lignes de code), de créer des points d'arrêt au comportement analogue aux points d'arrêt internes décrits plus haut. Un type spécial de point d'arrêt Python sont celui dérivé de la classe `gdb.FinishBreakpoint`. Il s'agit d'un type de point d'arrêt temporaire qui vise à s'arrêter lorsque l'exécution quitte le contexte courant (comme pour la commande `finish` décrite plus haut), soit par retour conventionnel ou par `longjmp`. Ils sont principalement utilisés pour permettre au script Python de récupérer la valeur de retour d'une fonction.

Une fois le programme arrêté sur un point d'arrêt, le code Python appelé peut consulter l'état du processus, par exemple lire la valeur des variables locales. Plus généralement, il peut évaluer n'importe quelle expression supportée par GDB. Dans le cas du langage C, ces expressions peuvent être arbitrairement complexes et peuvent même contenir des déréférences de pointeur. GDB, en évaluant l'expression, ira au besoin lire les valeurs nécessaires dans la mémoire du processus débogué.

Pour permettre d'étendre l'interface utilisateur, l'API Python permet de définir de nouvelles commandes. L'utilisation de cette portion de l'API est plutôt simple. Le script Python n'a qu'à spécifier le nom de la commande et le code à exécuter lors de son invocation.

Finalement, si l'API permet de consulter la liste des fils d'exécution existants et leur état (démarré, en pause, terminé), il ne permet pas de le modifier. En d'autres mots, il n'existe pas encore de méthode pour interrompre un fil (l'équivalent de la commande `interrupt`) ou redémarrer son exécution (l'équivalent de la commande `continue`).

Nous présentons dans la section suivante un article ayant été rédigé pour résumer nos travaux visant à assister les développeurs dans le débogage de programmes basés sur les appels à distance (RPC). L'essentiel des travaux présentés dans l'article se base sur les concepts présentés dans le chapitre actuel. Le paradigme des appels à distance est souvent utilisé dans l'implémentation des systèmes distribués pour abstraire les primitives de communication. Grâce aux RPC, on permet au programmeur d'appeler une fonction dans un contexte différent, possiblement dans un processus sur un hôte différent.

Les appels à distance, qu'ils soient implémentés à l'aide d'une bibliothèque spécialisée ou qu'ils soient implémentés au complet par le programme, constituent une entrave au débogage. Si un problème se trouve dans l'implémentation d'une méthode côté serveur et qu'on débogue le client, on ne pourra que constater la réponse erronée provenant du serveur. Il est parfois possible de déboguer directement le serveur en mettant un point d'arrêt sur la fonction de rappel fautive. Ainsi, dès qu'un client effectue un appel à cette fonction, le débogueur rendra la main à l'utilisateur qui pourra effectuer son travail d'investigation. Cela peut être problématique, notamment si la fonction fautive est appelée plusieurs fois avant que la

situation problématique se présente.

L'article présente donc les résultats de nos travaux de recherche visant à faciliter le débogage d'applications distribuées. Le résultat est l'implémentation d'un cadre logiciel ajoutant à GDB la possibilité de déboguer plus facilement les appels à distance entre un client et un serveur. Le cadre logiciel peut facilement être adapté à différentes bibliothèques d'appels à distance ou même aux appels à distance implémentés de façon non standard par les programmes.

CHAPITRE 4

ARTICLE 1 : RPC DEBUGGING WITH GDB

4.1 Abstract

Remote procedure call (RPC) libraries help building distributed applications by giving the illusion that a program can call a function in another process, possibly on another system. Standard source-level debuggers, which are invaluable tools when it comes to pinpointing the root of a software problem, become less effective when one uses RPC. We propose an extension framework to the GDB debugger that detects remote procedure calls and allows the developer to step from the client code to the server code whenever such a call is made. We show that it is possible with little effort to adapt the framework to existing, generic RPC libraries. This benefits directly developers of existing applications using those libraries. We also show that the framework may be used with minimal effort to debug applications that implement their own RPC scheme.

4.2 Introduction

Programming using remote procedure calls (RPC) is a well-known paradigm for designing distributed software (Nelson, 1981). A server can make a certain number of functions available for calling by remote clients. In the client code, a call is made to a stub function (also known as proxy) which is designed to look as much as possible like a native function call. Under the hood, the RPC library takes care of serializing arguments and transmitting them to the server, as well as metadata specifying which procedure is to be called. On the server-side, arguments are deserialized and the appropriate function is called. Upon return, the result is sent back to the client, which can resume its normal execution. Even though the client and server both have their own call stack, we can consider that the logical call stack of the program spans from the server to the client. In a bigger distributed system, if the server of an RPC in turn makes a call to another server, it is even possible for the logical stack to span many processes on many systems.

It is easy to see the limitations of a source-level debugger when debugging a program that makes use of remote procedure calls. When one reaches a call to a stub function, the best to expect is to step into the code that serializes the arguments and communicates with the server. Often, this communication code is *support code* and is not related to the actual

problem that the program is trying to solve. It is therefore impossible with the debugger to follow the flow of the program when it crosses the boundary between the client and the server.

The main goal of this work is to improve an existing debugger, GDB, to let a developer step "through" RPC in his program as seamlessly as possible. Since many protocols exist for remote procedure calling, and for each of these protocols different library implementations can exist, it is necessary to add knowledge about specific libraries to the debugger. Therefore, an important goal for the proposed method is to abstract as much as possible the common code, such that adding support for a new library is as simple as possible.

Because of the problems with RPC when using a debugger, developers rely more often on inspection of verbose outputs from both the client and the server to identify a problem. Using the debugger with this extension, it becomes possible to walk in the code until the problem presents itself and examine the state of the program at this point.

4.3 Previous work

The Microsoft Visual Studio IDE allows the programmer to step through an RPC, into a server function, when doing a Component Object Model (COM) or Windows Communication Foundation (WCF) call. COM is the programming model that has been used on Windows for a long time, where resources are represented as objects that live in processes. A process can call methods on objects living in-process or out-of-process. With the newer WCF, some processes can present themselves as services and export methods, which other processes can call. Visual Studio has specific features to assist debugging programs using either of these frameworks. When stepping into a stub function, the IDE will bring the focus of the debugger at the corresponding entry point in the server (Microsoft (2013c) and Microsoft (2013f)). When the server-side function returns, focus is brought back to the call site in the client. This allows the programmer to easily walk the logical flow of the program, without having to manually switch back and forth between the client and the server. As show in Figure 4.1, Visual Studio also adapts the call stack view to show a combined backtrace, including only the frames that are relevant to the user. In some cases, the debugger is even able to automatically attach to the server process (Microsoft, 2013g). Since the Visual Studio debugger is a closed-source program, there is no detail readily available on how these features are implemented. Furthermore, the RPC libraries supported are hard-coded and new RPC protocols and libraries cannot be supported.

The GNU debugger, GDB, is probably the most widespread source-level debugger for C programs in the Unix-like world (Free Software Foundation, 2014d). It supports a wide

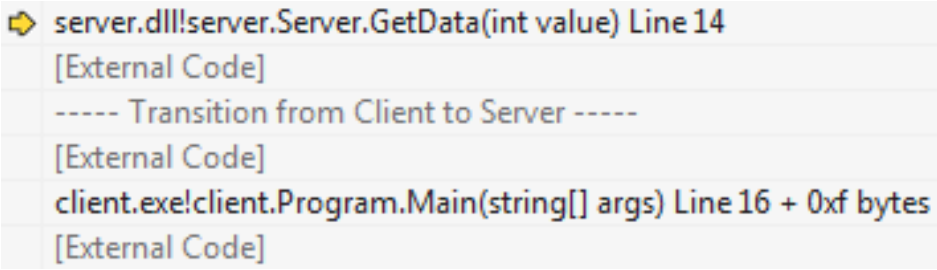

Figure 4.1 Call stack shown by Visual Studio while debugging an RPC.

variety of processor architectures, operating systems and binary formats. It does not have any feature related to RPC debugging, similar to those found in Visual Studio, but its Python API allows extending its features beyond basic debugging tasks (Free Software Foundation, 2014b). It is possible to set breakpoints and specify code to be run whenever they are hit. It also lets Python code inspect existing inferiors (processes), threads, stack frames and evaluate expressions in the context of the inferior. Other notable features of GDB include the multi-process capability, the target-async mode and the non-stop mode. The multi-process feature simply means that an instance of GDB can debug multiple processes at the same time. The target-async mode specifies that GDB can execute and receive commands while the inferiors are running, which was not possible in earlier versions. The non-stop mode means that whenever a thread hits a breakpoint, other threads continue running unaffected. The user can issue *continue* or *interrupt* commands on a thread-level basis. These two modes are orthogonal in functionality, but are still often used together.

Test clients are a class of RPC debugging tools that allow debugging the functional aspects of the servers by acting as mock clients. If the underlying protocol allows it, they can query the server and display a list of available methods. The developer can select one of them, manually input arguments and call the method. This is very useful to check if the server is configured properly or test a function with a particular set of arguments. It is easy to attach a debugger to the server while using a test client. The downside is that the arguments are synthetic and the method is not tested or debugged in the context of a real program. If a sequence of calls are required to get to a given state and reproduce the problem, using a test client can be inconvenient. D-Feet (GNOME, 2013a) is an example of such a test client for the D-Bus protocol. Microsoft also ships a WCF test client (Microsoft, 2012b) with similar capabilities as part of the Visual Studio environment.

Another class is non-interactive tracing or debugging tools. They can be used to record events related to RPC communication and allow post-mortem analysis of the interactions. For example, `rpcdebug` (Banks, 2006) is used to enable verbose logging for the SunRPC protocol,

while `rpctrace` (Free Software Foundation, 2013) is used to track of all communications between RPC servers and clients in the GNU Mach kernel. These tools may help to locate errors related to the use of the protocol itself, but don't help for debugging business logic code.

4.4 User interface

For stepping through RPC, three commands are added to GDB, all modeled after existing commands :

— `step-rpc`
— `finish-rpc`
— `backtrace-rpc`

When calling `step-rpc`, a standard step operation is executed. If the debugger detects that the step caused the program to enter a function known to initiate a remote call, the client's execution is resumed to let it send the request and the server is stopped once it enters the corresponding handler function. From the point of view of the user, the debugger effectively lets him step from the client code to the server code. If no RPC initiation is detected when issuing the `step-rpc` command, it results in a standard step.

When in an RPC, the `finish-rpc` command allows the user to complete the current remote call and stop just after the call site in the client code. It is analogous to the `finish` command in GDB, which completes the execution of the current function and then stops.

In GDB, the `backtrace` command shows the current call stack of the program. Command `backtrace-rpc` shows the logical call stack of the program that spans across the client and the server. It only shows relevant stack frames, leaving out those that belong to RPC support code. Listings 4.1 and 4.2 show the stack traces of the server and client during a sample XML-RPC call, while Listing 4.3 shows the result of `backtrace-rpc` at the same point.

Listing 4.1 Stack trace (backtrace) of server during an XML-RPC call

```
#0  do_things at server.c:12
#1  sample_add at server.c:32
#2  callNamedMethod at registry.c:294
#3  xmlrpc_dispatchCall at registry.c:324
#4  xmlrpc_registry_process_call2 at registry.c:413
#5  processCall at xmlrpc_server_abyss.c:475
#6  handleXmlrpcReq at xmlrpc_server_abyss.c:610
#7  runUserHandler at server.c:541
#8  processDataFromClient at server.c:577
#9  serverFunc at server.c:629
#10 connJob at conn.c:39
#11 pthreadStart at thread_pthread.c:49
#12 start_thread at pthread_create.c:301
#13 clone at clone.S:115
```

Listing 4.2 Stack trace (backtrace) of client during an XML-RPC call

```
#0  __pselect at pselect.c:73
#1  waitForWork at xmlrpc_curl_transport.c:437
#2  finishCurlMulti at xmlrpc_curl_transport.c:570
#3  performCurlTransaction at
                    xmlrpc_curl_transport.c:1050
#4  performRpc at xmlrpc_curl_transport.c:1155
#5  call at xmlrpc_curl_transport.c:1376
#6  xmlrpc_client_call2 at xmlrpc_client.c:580
#7  clientCall_va at xmlrpc_client_global.c:147
#8  xmlrpc_client_call at xmlrpc_client_global.c:174
#9  add at client.c:36
#10 main at client.c:62
```

Listing 4.3 Resulting combined client-server stack trace (backtrace-rpc)

```
server - #0 do_things at server.c:12
server - #1 sample_add at server.c:32
client - #2 add at client.c:36
client - #3 main at client.c:62
```

4.5 Implementation algorithms

The technique proposed in this article is based on internal breakpoints, which are already used by GDB to monitor inferiors. For example, shared library loading is tracked using an internal breakpoint strategically placed in the dynamic loader code. Whenever it is hit, GDB takes note of the newly loaded library and resumes the program automatically, which makes the operation almost invisible to the user.

Finish breakpoints are also used on multiple occasions. Given an active stack frame, it is possible to install a finish breakpoint on it so that GDB will interrupt the inferior once the frame is exited (when the function returns, exception is thrown, longjmp is called, etc.).

The framework consists of a generic Python module containing functions and classes that can be used and extended by RPC library-specific modules. It also defines the commands that are made available to the user. To use the framework with some library, it is necessary to have GDB load the library-specific module.

4.5.1 `step-rpc`

When loading the Python module designed for a specific RPC library, internal breakpoints (dubbed client start breakpoints) are added at the entry points of the functions known to initiate a remote call. They are represented by the breakpoint 1 on Fig. 4.2. The breakpoints are initially disabled so they do not impede normal functionality. When the user enters the `step-rpc` command, we begin by enabling them, issue a standard step and finally disable them again. If one of the breakpoints is hit during the step, we record which one it is as well as any relevant information necessary to identify which remote method is being called. If no breakpoints were hit, nothing happens and the whole operation is equivalent to a normal step. In the case where a breakpoint was hit, we install a finish breakpoint (client finish breakpoint) on the current stack frame (breakpoint 4 on Fig. 4.2). It will help us determine when the RPC is completed. A temporary breakpoint (server start breakpoint) is also placed at the beginning of the appropriate method handler in the server, using the information previously collected (breakpoint 2 on Fig. 4.2). Finally, when this last breakpoint is hit, a finish breakpoint (server finish breakpoint) is installed in the current stack frame of the server (breakpoint 3 on Fig. 4.2). It will help determine that the server is done executing the remote call handler.

Once all this breakpoint machinery is installed, the client's execution is resumed to let it send the request to the server. Eventually, the server will execute the handler function and will hit the breakpoint placed at its entry point. The user is now free to debug the server as usual or even resume its execution. When the server's finish breakpoint is hit, we know

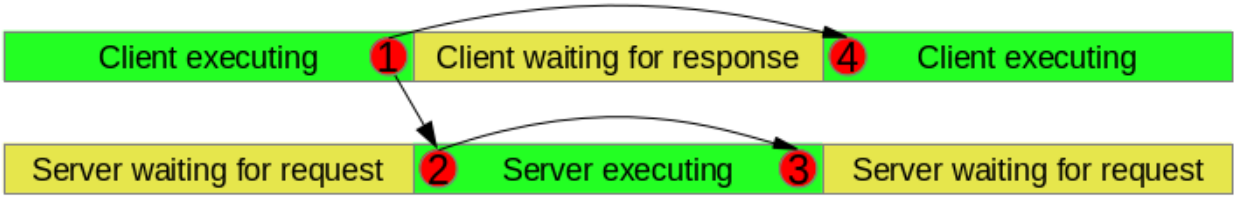

Figure 4.2 Flow of RPC debugging. Red dots represent breakpoints. Arrows denote how breakpoints are set.

the server is done executing the user-defined handler. We then resume the server execution, which will send back the result to the client, therefore unblocking it. Eventually, the stub function in the client will return and thus hit the final, client finish, breakpoint. At this point, standard debugging of the client resumes.

While debugging one client thread, there might be other threads calling the same procedure on the server. There is therefore a risk of stopping at the beginning of a method handler invocation that does not match the call we want to follow. If possible, it is good practice to confirm that an invocation of the handler by the server indeed matches the call of the thread we are debugging. In some libraries, this can be done by checking for a unique request identifier in both the server and the client, ignoring the invocation if they don't match.

Most of the logic associated to this interaction between the debugger, the client and the server is common to all RPC libraries. It was therefore possible to isolate a generic `gdbrpc` Python module containing one base breakpoint class for each of the four breakpoints described previously. All that is needed to write a library-specific module is to specify where start breakpoints should be placed in the client, how to identify the procedure being called and how to identify the corresponding handler in the server.

In its simplest form, this can be a direct, one to one mapping between functions in the client and the server. We want to link both such that a `step-rpc` in the former will bring us directly to the later. All we need to do is instantiate the supplied `SimpleStartBreakpoint` class, such as :

```
SimpleStartBreakpoint("client_func", "server_func")
```

After issuing `step-rpc` on `client_func`, GDB will break the next time execution enters `server_func`. This does not check that the handler invocation matches the right client call, as mentioned earlier. However, it is very easy to setup and should be sufficient for most applications.

If more fine-grained control is needed, the supplied base classes can be extended. For example, when reaching a client start breakpoint, we might need to inspect arguments values

to determine which method is being called, thus where to put the breakpoint in the server. The methods required to be overriden in the client start breakpoint are :

— `stop` : Called when the breakpoint is hit, therefore when the client initiates an RPC, this method should gather any relevant information to identify the destination of the call.

— `install_server_breakpoint` : This method should instantiate the server start breakpoint based on information collected in `stop`.

In the server start breakpoint, the only useful method to override is `stop`. It can be used to verify that the current invocation corresponds to the call we want to track.

4.5.2 `finish-rpc`

When the user enters the `finish-rpc` command, we simply resume the execution of the server thread. The remote call will finish normally and the client will hit its finish breakpoint. As a result, the client's execution will be stopped just after the call site, as expected.

4.5.3 `backtrace-rpc`

To implement the `backtrace-rpc` command, we record the current stack frames start breakpoints of both the server and the client. These correspond to the boundary between frames that are of interest for the user and those that are not. To print a combined stack trace, we first print the stack frames of the server until we cross the boundary frame. Frames after this one belong to the library and are not part of the logical path of the program. Then, we print the client stack trace, but only print the frames after crossing the boundary frame.

4.6 Library examples

In this section, we provide examples of how the proposed solution was applied to some RPC libraries. We used the `python-metrics` (Fink, 2013) package to count the number of source lines of code (SLOC) specific to each module. This gives an idea of the effort needed to adapt the framework to each library.

4.6.1 XML-RPC

XML-RPC is an RPC protocol where information about the called procedure as well as its arguments are encoded in an XML document transmitted over HTTP (Kidd, 2001). The library used is the common `xmlrpc-c` library, version 1.16.24.

During initialization, the server registers methods to export using the `xmlrpc_registry_add_method3` function, which takes as parameters the name of the method and a pointer to the handler. We can place a breakpoint on this function in order to maintain a registry of method names and handler addresses.

The client initiates calls using the `xmlrpc_client_call` function, to which it passes the method name. By looking up in the aforementioned registry, we can easily place the server start breakpoint in the right function.

The code specific to this XML-RPC library contains less than 40 SLOC.

4.6.2 ONC

A more complex example is the Open Network Computing protocol (ONC, formerly known as SunRPC) (Thurlow, 2009). It is an older but still widely used RPC library, notable for its use in the Network File System (NFS) implementation. The ONC implementation used is the one found in the glibc 2.12 library.

When using the ONC protocol, helper code is generated from an interface definition language using the `rpcgen` (Oracle, 2010) tool. Unfortunately, this generated code is indistinguishable from user-defined code. When using our solution, the user has to do a few steps in this generated boilerplate code to get to his code.

A procedure in ONC can be uniquely identified using a (service, version, procedure) tuple. Using the `svc_register` function, the server associates a (service, version) tuple to a single callback, which is part of the generated code. Procedure demultiplexing is done in this callback. This allows us to maintain a registry of (service, version) to callbacks.

On the client side, the program first has to create a client object to connect to a (service, version), which it uses to make the remote calls. We therefore have to keep a mapping between client object addresses and a (service, version) tuple. When a call is made, it is possible to lookup which service is being called from the client object and then lookup which function is going to be called from the service.

Because it is necessary to track client creation and callback registration by the server, the number of SLOC for this library is a little bit higher, at around 60.

4.6.3 D-Bus

D-Bus (Freedesktop.org, 2014a) is a protocol often used by applications on a single system to interact in a transparent manner. The implementation we used is gdbus, found in the glib library, version 2.39.

The `gdbus-codegen` tool is used to generate helper code for implementing both clients

and servers from an XML description of the service. The code used to implement the server makes heavy use of glib signals (GNOME, 2013b) facility. One signal per exported procedure is created and a callback is associated to each of these. By hooking with a breakpoint on the signal registering function, it is possible to maintain a mapping between method name and handler.

To initiate an RPC, the client calls a function generated by the tool that is specific to each exported method. This function in turn calls `g_dbus_proxy_call_sync` and passes it the name of the method to call. From there, we can identify which method is being called and place the server breakpoint in the right handler. Again, the user has to make a few steps in the generated code until he gets to `g_dbus_proxy_call_sync`.

The number of SLOC required for the Gnome D-BUS library amounts to about 40.

4.6.4 Custom remote calls

Many client/server applications don't use an RPC library, but implement their own or are built around the same concept of blocking calls. If they use a somewhat consistent architecture, the current solution proposed in this article can easily be adapted to help with distributed debugging. Often, the client application will contain a function used to initiate a request or function call to the server. On the server-side, a function is used to receive the request, identify which method the clients wants to call and call it. In this case, it is easy to install breakpoints at strategic points to identify when an remote call is being made. Examples of such applications are the LTTng tracer (LTTng, 2014) and Transmission bittorrent client (Transmission, 2014).

LTTng is used for Linux kernel and applications tracing. It uses a daemon (called the session daemon) to keep track of the current state of all the active tracing sessions. The user can interact with it (create and configure sessions, start tracing, etc.) using a command line tool whose sole purpose is to make remote calls to the session daemon. When inspecting the code, we notice that commands available in the command line tool build a structure containing the command identifier and a variable number of arguments. They then call the `lttng_ctl_ask_sessiond_varlen` function which sends the structure to the server and blocks waiting for a response. In the server, we find a `process_client_msg` function which does a number of checks and calls a handler based on the command. Unfortunately, the structure is not systematic, so we can't link directly the function in the client to the right command handler. We can still link `lttng_ctl_ask_sessiond_varlen` to `process_client_msg` such that a `step-rpc` in the former brings us to the later. This requires exactly two SLOC, one import statement and one instance of a SimpleStartBreakpoint.

Transmission is a bittorrent client which can be started in daemon mode. The `transmission-remote`

CLI tool is used to make queries to the daemon. When the client wants to make a request to the server, it fills a map with the method name and an argument list. It then passes this map to a function called `flush` which sends the request using Transmission's own protocol. On the other side, all possible method name and associated handlers are conveniently placed in an array of structures. The `request_exec` function in the daemon walks this array until it finds a match and calls the handler. The trivial solution is to map `flush` to `request_exec`. However, we can do a little bit better. When the start breakpoint on `flush` is hit, we can inspect the map to find out which method is being called. This is made possible by GDB's capability of evaluating expressions in the current context of the inferior. We then find the corresponding handler in the methods array and place our server start breakpoint there. For the Transmission module, about 35 SLOC were required.

4.7 Modifications to GDB

This section details two modifications to GDB that were required in order to develop the proposed solution.

4.7.1 Python API

Although being extensive, the GDB Python API as of GDB 7.6 is missing an important piece : allowing scripts to control the inferior running state. It is therefore not possible to ask a thread to interrupt or resume its execution, as can be done from the command line. An example where this feature is required is during the execution of the `step-rpc` command. If an RPC call start is detected, we need to resume execution of the client program to let it send the request. We added two methods to the API to enable this functionality. Under the hood, interruption of a specific thread works by sending a SIGSTOP with the tkill system call. This means that the inferior will not be interrupted until it gets scheduled and the signal is handled. Before attempting operations that requires it to be stopped, it is necessary to wait until GDB gets the confirmation that the inferior has been stopped (using a member of the wait system call family). For convenience, we made the interrupt method synchronous, meaning that when it returns, you are certain that the inferior is in an interrupted state.

4.7.2 Memory access with running inferior

We found that with GDB 7.6, there is a problem related to breakpoint creation while the inferiors are running. The problem boiled down to GDB trying to make a ptrace call to read or write the inferior's memory while it is running, which is expected to fail. As a workaround, we modified the linux_nat_xfer_partial function to make sure that we avoid

this situation. This function is responsible for writing or reading one word in the inferior's address space. If it is running, GDB will now stop the inferior for the time of the operation. As for the interrupt method in the Python API, it is necessary to wait until the inferior is indeed stopped before proceeding with the read or write. Doing a cycle of stopping and starting the inferior for each word is obviously inefficient. In general it would be preferable to stop the inferior only once for a complete transfer or even for multiple transfers. Still, for this application, this workaround was sufficient and did not add any noticeable delay when debugging.

4.8 Performance considerations

Since the solution is meant to be used in interactive debugging scenarios, performance is not a major issue. Still, it is important to make sure that the debugger does not add some unreasonable delay affecting the user experience. During the execution of `step-rpc`, all client start breakpoints are enabled and then disabled. We therefore depend on the performance of breakpoint management in GDB. If their number is low, `step-rpc` should not take much more time than `step`, since the later already does its share of breakpoint insertion and removal. However, we could easily imagine an automation tool that creates one breakpoint per remote function in a system that contains hundreds of them. Table 4.1 shows the execution time of `step-rpc` in function of the number of client start breakpoints. The time measured is for the activation of the breakpoints, the actual step and disabling of the breakpoints. While execution time increases, it remains reasonable for interactive use at 100 breakpoints, which should be more than enough for most applications.

While the client and server processes are executing normally, internal breakpoints are disabled, so they have no impact.

Tableau 4.1 Average execution time of `step-rpc`

Number of client start breakpoints	step-rpc execution time (ms)
1	1.72
10	2.64
100	14.5
500	121
1000	396

4.9 Limitations and future work

An important limitation of the proposed solution is that the client and server must be debugged by the same GDB instance. This currently means that they must run on the same system, because GDB can't debug multiple targets at the same time (a target being either the local system or a remote system). When debugging a distributed system, this can be cumbersome since we expect the client and the server to run on different targets. At some point, if support for debugging multiple targets concurrently is added to GDB, this limitation should disappear. Another solution could be to use two different GDB instances and handle the logic at a higher level. For example, Eclipse's Debugging Services Framework (DSF, Eclipse's debugging backend for C/C++) already spawns a GDB instance per debugged process, which can be on a remote system. It uses the GDB Machine Interface (MI) to control the debuggers, insert breakpoints, obtain inferior's information, control the execution, etc. The logic for RPC debugging could very well be handled by DSF and would eliminate this restriction.

Also, as we saw with the various examples, it is often hard to distinguish user code from tool-generated code. Even though the user is free to modify by hand the generated code, we can generally assume this code is not interesting for him. It would be a great improvement to be able to skip this code, as it would allow the user to focus more on important parts of his program.

The proposed method allows to step through an RPC from a client to a server. For more complex systems, it is possible that multiple levels of remote calls exist. A client could make a call to a server which in turn makes a call to a second server, and so on. It should be possible to extend the solution to support these multiple levels and follow the logical flow of the program across an arbitrary number of systems.

Finally, it would be an interesting improvement to allow the user of the debugger to easily define function pairs, without having to write any Python code. For a user of the command-line prompt, that would translate in a command that would allow him to declare that a `step-rpc` in `function_a` should bring the focus to `function_b`. In the background, it would instanciate a SimpleClientBreakpoint in our RPC debugging module. For a user of a graphical front-end such as Eclipse, it would be in the form of a simple dialog to declare the same thing.

4.10 Conclusion

In this paper, we proposed an extension to the GDB debugger to assist a programmer in the task of debugging a distributed system based on remote procedure calls. This allows him

to follow the logical flow of a program even when it crosses the boundary between processes. We believe that the framework makes it easy to teach GDB how to handle a new RPC library, or even RPC that are not based on a well-known library. Most of the time, simply mapping some functions in the client to corresponding functions in the server already helps, and it takes no more than a line of code. However, when this is not enough, or we want something more fine-grained, we only need to reimplement two or three methods to adapt it for a specific use case. For some widely used libraries such as Gnome D-Bus or ONC, we were able to get something useful under 60 SLOC.

CHAPITRE 5

TRAÇAGE DE PLATEFORMES À GRAND NOMBRE DE CŒURS

Ce troisième chapitre présente les résultats du travail de port et d'évaluation de la performance du traceur LTTng sur Tilera Tile-Gx et Intel Xeon Phi. Nous détaillons d'abord le travail qui a été nécessaire pour rendre le traceur fonctionnel sur ces plate-formes. Ensuite, nous présentons les procédures utilisées pour évaluer l'impact en performance de LTTng sur des applications où l'on retrouve typiquement ces processeurs. Finalement, les résultats de ces tests sont discutés.

5.1 Adaptations et corrections

Le noyau de Linux est bâti avec l'intention d'être compatible avec un grand nombre d'architectures. Un grand soin est donc porté afin d'offrir aux modules et autres sous-systèmes une interface de programmation indépendante de la plate-forme. Cela réduit grandement l'effort requis pour porter un système tel que LTTng, à condition que celui-ci respecte l'interface de programmation offerte et n'utilise rien qui dépende d'une architecture en particulier. Sachant que LTTng a été testé avec plusieurs architectures, nous nous attendions à ce que son port vers les deux plate-formes avec lesquelles nous travaillons soit une tâche relativement simple. En fin de compte, quelques petits ajouts ou modifications ont dûs être faits, soit pour implémenter des fonctions dont le traceur a besoin ou pour régler des cas où les suppositions que LTTng faisait ne convenaient pas aux nouvelles architectures.

Toutes les modifications décrites dans cette section ont été soumises et intégrées dans les projets correspondants, soit LTTng, la bibliothèque userspace-rcu et le noyau Linux.

5.1.1 Tilera TILE-Gx

Étant plus jeune et moins répandue, la plate-forme de Tilera est celle qui a requis le plus de modifications pour faire fonctionner le traçage noyau.

Peu importe l'architecture, pour que LTTng puisse recueillir les évènements provenant du noyau, il faut que celui-ci soit compilé avec l'option `CONFIG_TRACEPOINT`, qui active l'insertion de points de traces. Pour des raisons d'espace, le noyau fourni par Tilera ne contient pas ces points de traces, il a donc d'abord fallu le recompiler après activé cet élément de configuration. Toutefois, le système de configuration du noyau définit l'option `CONFIG_TRACEPOINT` comme une propriété automatique qui est activée uniquement si un des

systèmes qui en dépend est lui aussi activé. Pour éviter de devoir activer un système dont nous n'avions pas besoin, nous avons préféré modifier le type de la propriété pour la rendre manuelle et ainsi pouvoir l'activer seule. Bien que la compilation du noyau puisse s'avérer être une tâche une peu complexe lorsque nous avons affaire à des architecture quelque peu exotiques, elle aurait de toute façon été nécessaire pour les étapes suivantes.

À ce point, il est possible de compiler et de charger les modules permettant d'effectuer le traçage du noyau. Un premier problème a été rencontré lorsque LTTng échouait à créer les tampons de traçage lors de la création d'une session. La source de ce problème était la taille de page particulière du TILE-Gx (64 Kio) par rapport à la taille plus couramment utilisée par les autres processeurs (4 Kio). Pour des raisons d'efficacité, la bibliothèque de tampon circulaire développée pour LTTng n'accepte que des tailles de tampons au moins aussi grandes que la taille d'une page mémoire du système. Cela permet entre autres d'utiliser l'appel système `splice` pour transférer le contenu des pages mémoires d'un processus à l'autre ou vers un pilote sans copie. Du côté de l'utilitaire en ligne de commande `lttng`, utilisé pour contrôler le traçage, la taille des tampons par défaut pour les méta-données était fixée à 4 Kio. L'utilitaire `lttng` a donc été modifié pour quérir la taille de page du système et s'assurer que la taille de page par défaut est au moins aussi grande que cette valeur.

Un second problème semblait apparaitre de façon plus intermittente lors de l'activation du traçage noyau. Une panique du noyau était provoquée par le déréférencement d'un pointeur nul. Ce comportement apparaissant uniquement lorsque le réseau était actif. Le problème venait du fait que le module responsable du pilote réseau pour cette plate-forme enregistrait une interruption avec un nom de dispositif nul. Lorsqu'une interruption de ce type survenait, le point de trace LTTng tentait de copier le nom, nul, vers un champ de l'évènement, provoquant l'erreur. Le problème a été réglé en modifiant le code du module pour qu'il passe un nom plus éloquent à la fonction d'enregistrement d'interruptions. Une modification a également été apportée du côté de LTTng afin de se prémunir contre d'éventuelles situations semblables dans le futur.

Ces deux problèmes réglés, la fonction de traçage noyau était maintenant fonctionnelle. Une brève tentative d'analyse des traces a toutefois indiqué l'absence des évènements dénotant les appels systèmes par les applications. La présence de ces évènements est cruciale pour la plupart des analyses de comportement des applications, puisqu'ils permettent d'identifier le début et la fin de l'ouverture d'un fichier, d'une lecture ou écriture sur un descripteur de fichiers, l'ouverture d'une connexion réseau, la fourche d'un processus et bien plus. Nous nous sommes rendus compte que le sous-système noyau *tracehook*, dont le traçage des appels systèmes dépend, n'était pas implémenté pour l'architecture TILE-Gx. Le rôle de *tracehook* est d'offrir une couche d'abstraction entre le code spécifique aux architectures et d'éventuels

outils de traçage ou de débogage. Pour qu'une architecture puisse déclarer qu'elle supporte le système *tracehook*, elle doit implémenter plusieurs fonctions qui permettent d'effectuer certaines actions normalement spécifiques à la plate-forme (obtenir la valeur des registres d'un fil, effectuer un pas d'une instruction pour un fil donné). Le code de l'architecture doit également appeler des fonctions à des moments définis pour avertir les outils qu'un évènement est survenu (entrée ou sortie d'un appel système, suspension ou reprise d'une tâche). L'implémentation de l'interface *tracehook* a donc permis d'ajouter les évènements de type appels systèmes aux traces noyaux LTTng.

Par défaut, les évènements correspondant à l'entrée d'un fil dans un appel système contiennent uniquement le numéro de l'appel système, pas son nom. Le problème est que les numéros assignés aux appels systèmes changent d'architecture en architecture. Le fait de ne pas avoir les noms des appels systèmes complique énormément la tâche des outils d'analyses de trace. Il est donc possible et utile d'éduquer LTTng à propos des noms correspondants à chacun des numéros d'appels systèmes pour une architecture donnée, afin qu'ils soient inclus dans la trace. Cette tâche est effectuée à l'aide d'un script qui exporte cette information à partir d'un noyau en exécution et qui génère le fichier à inclure dans la source de LTTng. Le fichier généré contient également la description du type des arguments de chaque appel système, permettant à LTTng d'également les inclure dans la trace. Ce fichier a donc été généré et ajouté à LTTng.

Suite à ces modifications, le traçage noyau LTTng sur l'architecture Tile est fonctionnel.

5.1.2 Intel Xeon Phi

L'architecture du Xeon Phi étant dérivée de x86, peu de modifications ont été nécessaires pour faire fonctionner LTTng, x86 étant de loin l'architecture la plus testée et utilisée.

Une seule adaptation a été requise dans la bibliothèque `userspace-rcu`. Cette bibliothèque permet à des programmes en mode usager d'utiliser des techniques de la famille *Read-Copy-Update* (RCU) pour effectuer la synchronisation entre plusieurs fils en minimisant le blocage. Pour assurer l'intégrité des structures, cette bibliothèque se base sur les instructions de barrières mémoire qui ont pour effet de garantir un certain ordre dans l'exécution des instructions et les accès mémoire. Pour les architectures x86, *userspace-rcu* utilise les fonctions `mfence`, `lfence` et `sfence` pour y parvenir. Cependant, le Xeon Phi, possédant des fonctionnalités réduites, ne possède pas ces instructions. Comme alternative, une barrière peut être implémentée en exécutant une instruction caractérisée par le préfixe `LOCK`. L'ajout de ce préfixe a pour effet de créer une barrière mémoire complète.

5.2 Analyse de l'impact en performance du traçage

Une fois le traçage noyau fonctionnel sur nos deux plate-formes, nous avons effectué des tests afin d'en évaluer l'impact en performance sur des applications typiques. Cette section présente les procédures effectuées ainsi que les résultats obtenus.

5.2.1 Choix des programmes de tests

Puisque nos deux plate-formes ne sont pas destinés aux mêmes applications, nous avons choisi d'exécuter des tests différents pour chacunes.

Selon Tilera, le concepteur du TILE-Gx à 36 cœurs, les domaines visés par ce processeur comprennent notamment l'informatique nuagique, le multimédia et les appareils de réseautique. Le processeur et sa carte de développement possèdent de nombreuses fonctions périphériques pouvant servir pour ces domaines, comme plusieurs connexions réseau 1 et 10 gigabit/s (utiles pour l'informatique nuagique), des accélérateurs de calcul cryptographique matériels (utiles en réseautique) ou encore des connexions à haut débit PCI-express pour communiquer avec d'autres appareils (utiles en multimédia).

Nous avons fait le choix de tester la plate-forme Tilera en l'utilisant comme serveur de cache réseau, comme on retrouve fréquemment dans une pile d'applications web ou infonuagiques. Un tel serveur de cache permet à plusieurs nœuds de stocker le résultat de calculs ou de requêtes lourdes en calcul et de les récupérer plus tard. L'avantage que cette cache soit disponible sur le réseau et non locale à chaque nœuds est que le calcul effectué par un nœuds est disponible à tous les autres. Cela peut permettre à un nœuds de faire le même travail qui a été effectué précédemment par un autre. Puisqu'il s'agit d'un stockage plutôt éphémère, mais qui doit répondre avec la plus petite latence possible, les serveurs de ce type conservent généralement leurs données uniquement en mémoire et ne les écrivent jamais sur le disque. Le programme utilisé pour nos tests est memcached (version 1.14.17) (memcached, 2014), un programme réputé, énormément utilisé en industrie et en production. Son modèle basé sur les fils d'exécution se porte bien à une plate-forme comportant un grand nombre de cœurs comme la nôtre. Des alternatives telles que Redis (Redis Project, 2014) n'auraient pas été un bon choix, puisqu'elles adoptent un modèle unifilaire et traitent les requêtes de façon asynchrone. Pour tester le nombre de requêtes par secondes que memcached est capable de traiter, nous utilisons le banc de test memtier (Garantia Data, 2013), conçu justement par les développeurs de Redis mais également compatible avec le protocole de memcached.

La plate-forme Intel Xeon Phi, quant à elle, possède moins de périphériques et de possibilités d'interactions avec l'extérieur que celle de Tilera. Elle est davantage perçue comme une pièce pouvant être ajoutée comme coprocesseur à un serveur ou une station de travail et

servant à décharger les tâches lourdes en calcul. Le domaine visé est donc davantage celui du calcul haute performance, notamment pour des applications scientifiques. Dans un livre dédié à la programmation haute-performance sur l'Intel Xeon Phi, Jeffers *et al.* (2013) proposent plusieurs exemples de programmes ainsi que les optimisations possibles pour aller chercher autant de puissance que possible du processeur. Parmi les optimisations mentionnées, on retrouve l'ajustement du nombre de fils, l'alignement des données en mémoire, l'utilisation des registres et instructions vectorielles, la simplification des boucles les plus souvent exécutées et la meilleure division du travail entre les fils. Le programme que nous avons choisi pour tester le Xeon Phi est tiré de ce livre. Il s'agit d'un de calcul de diffusion en trois dimensions de particules dans un liquide. Nous avons choisi la version la plus optimisée proposée par les auteurs.

Afin de tester la bande passante mémoire disponible sur les deux plate-formes, nous utilisons également un programme tiré du livre de Reinders et Jeffers. Ce petit programme effectue un nombre déterminé de copies en va-et-vient de deux grand tableaux en mémoire afin de calculer le débit moyen des données en provenance et vers la mémoire. La taille des tableaux est assez grande pour éliminer les effets que la cache des processeurs pourrait avoir.

5.2.2 Tilera TILE-Gx

Cette section présente les procédures ainsi que les résultats des tests que nous avons effectué sur la plate-forme Tilera TILE-Gx.

Procédures de test

Pour effectuer les tests sur le processeur, nous utilisons la carte de développement TILEncore-Gx de Tilera. Physiquement, il s'agit d'une carte d'extension PCI-express destinée à être installée dans un serveur ou une station de travail. La carte possède quatre connexions réseau de format *small form-factor pluggable* dans lesquelles il est possible d'insérer des modules de fibre optique ou Ethernet. Nous avons installé deux de ces module sur notre carte, permettant à la carte d'avoir deux canaux de communication parallèles avec l'hôte. En théorie, un réseau virtuel peut être établi entre la carte et l'hôte par le biais de la connexion PCI-express, mais des tests préliminaires nous ont permis de déterminer que la connexion était moins stable et performante que le réseau Ethernet.

Lors du test, un serveur memcached est exécuté sur la carte, avec un nombre de fils d'exécution variable. Le programme de test *memtier* est exécuté sur l'hôte. Pour chaque test, *memtier* utilise 8 fils, simulant chacun 25 clients. Chaque client effectue 10000 requêtes au serveur, pour un total de 2 millions de requêtes. Une requête vers memcached peut être

soit un *SET* ou un *GET*. Nous avons ajusté la proportion à 1 *SET* pour 10 *GET*. La communication entre les deux nœudss se fait par la première des deux connexion réseau gigabit. L'hôte qui accueille la carte possède les caractéristiques présentées dans le tableau 5.1.

Les scénarios de traçage suivants sont testés :

— Sans traçage : il s'agit du cas de référence.

— Traçage local : la trace est enregistrée localement. Comme le système de fichier est en mémoire vive, l'espace "disque" est limité par la mémoire disponible. Dans le cas présent, le système de fichier peut utiliser jusqu'à 4 des 8 Gio de mémoire. Cette méthode est donc utile pour des séances de traçage de relativement courte durée.

— Traçage réseau : la trace est enregistrée sur l'hôte à l'aide de la fonction d'enregistrement distant de LTTng. Le démon *relayd* est lancé sur l'hôte et les données de traçage passent par la même interface que les requêtes *memcached*.

— Traçage réseau, interface différente : identique au traçage réseau, sauf que les données de traçage sont transmises sur une interface réseau différente des données de *memcached*.

En plus de varier la méthode d'enregistrement des traces, nous varions les point de trace activés. D'abord, pour observer le comportement du système sous un fort débit de traçage, nous activons tous les évènement noyaux. Il s'agit souvent du premier réflexe des utilisateurs explorant les outils de traçage et cela permet de tester le cas le plus demandant. En général, il vaut mieux sélectionner uniquement les points de trace nécessaires à l'analyse que l'on veut conduire afin de générer un débit de traçage plus raisonnable. En second lieu, nous activons donc le sous-ensemble des points de trace permettant de reconstruire l'historique d'exécution des processus (et donc la *Control Flow View* de l'outil *Tracing & Monitoring Framework*). La liste des points de trace correspondants est la suivante :

— appels systèmes

— irq_handler_entry

— irq_handler_exit

— softirq_entry

— softirq_exit

— softirq_raise

— sched_switch

— sched_process_fork

— sched_process_exit

— sched_process_free

— lttng_statedump_process_state

— sched_wakeup
— sched_wakeup_new

Lors du traçage, la taille de tampons choisie a un impact direct sur la consommation de mémoire du traceur. Par défaut, LTTng alloue quatre tampons par cœur. Ainsi, un tampon peut être utilisé pendant que les autres sont en train de se faire consommer. Puisque le TILE-Gx comporte 36 cœurs, cela représente 144 Mo (36 cœurs × 4 tampons / cœur × 1 Mio / tampon) dédiés au traçage.

Tableau 5.1 Configuration matérielle de l'hôte du Tilera TILE-Gx

Carte mère	Intel DH77EB
Processeur	Intel i7-3770 @ 3.40GHz
Mémoire vive	16 Gio DDR3

Résultats

Les graphiques 5.1 et 5.2 présentent les résultats de la performance de memcached en fonction de la méthode de traçage. Dans le premier cas, tous les évènements noyaux sont activés, alors que dans le second, seule la sélection réduite d'évènements est activée. Les données équivalentes sont présentées dans le tableau 5.2. Le tableau 5.3 présente quant à lui le débit de données de traçage pour chaque essai. Le débit de traçage est calculé en divisant la taille de la trace par la durée de l'expérience. Finalement, le tableau 5.4 indique s'il y a eu perte d'évènements lors de l'expérience. Le nombre exact d'évènements perdus nous importe peu. Dès qu'un évènement noyau est perdu, la reconstitution de l'état des processus dans le temps ne sera plus fiable.

Performance sans traçage En observant la courbe sans traçage sur la figure 5.1, on voit que le nombre de requêtes servies suit le nombre de fils jusqu'à une vingtaine de fils. Au delà de 20 fils, le gain en performance de chaque fils ajouté est très faible, voire nul. À première vue, plusieurs facteurs pourraient expliquer un tel comportement.

Selon l'auteur de memcached, des efforts on été effectués pour améliorer la mise à l'échelle sur plusieurs fils (memcached, 2012). Cependant, comme le fonctionnement de l'application est tout de même basé sur des verrous, il se peut qu'un nombre si élevé de fils cause une forte contention autour de ces verrous.

Il se pourrait aussi que 20 fils arrivent à saturer le bus vers mémoire. Si c'est le cas, les fils passent simplement davantage de temps à attendre que leurs accès mémoire soient

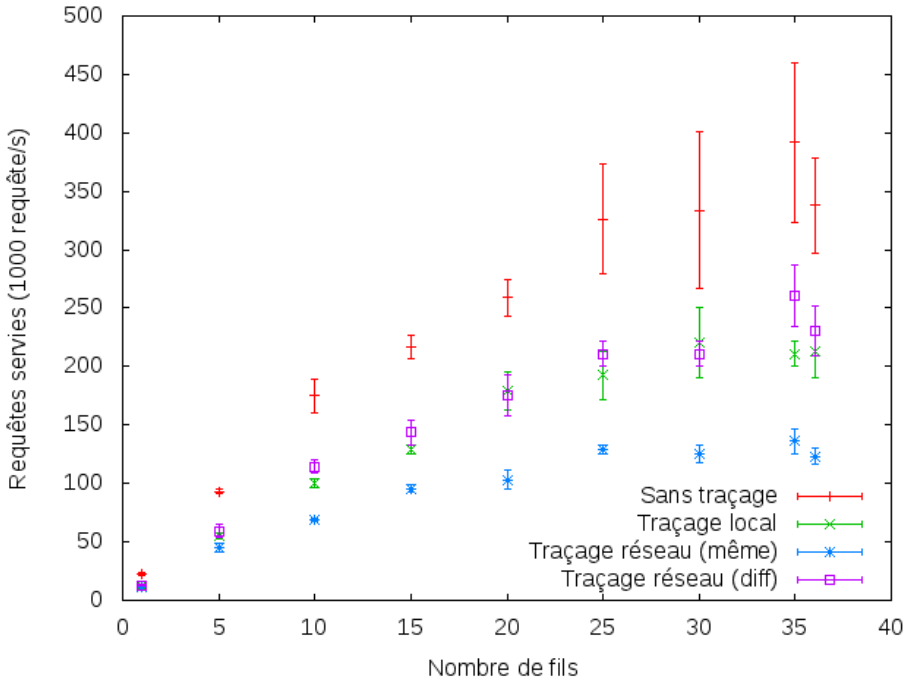

Figure 5.1 Nombre d'opérations servies par memcached en fonction de la méthode de traçage (tous les évènements activés)

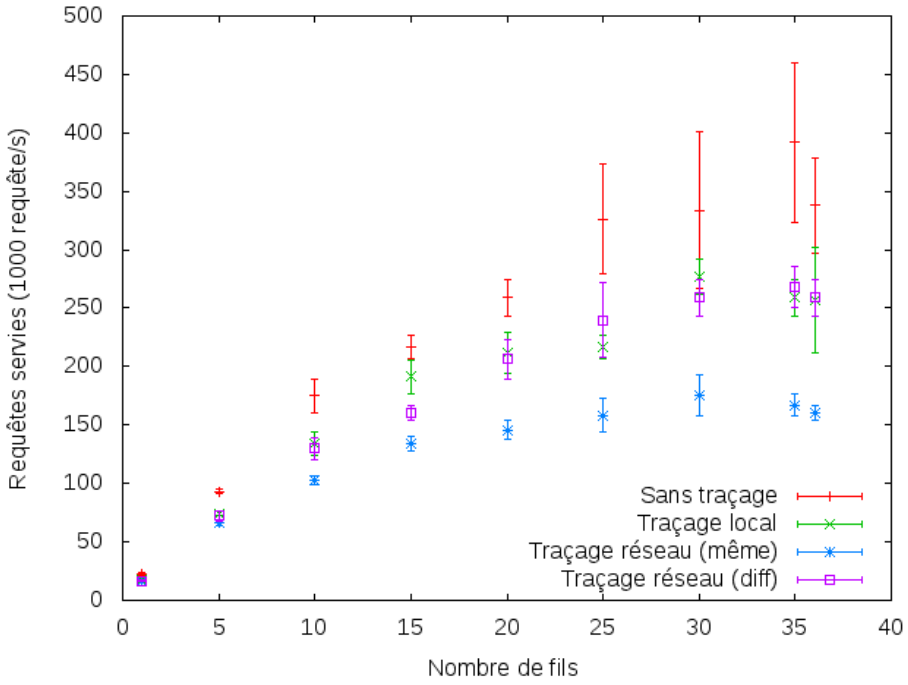

Figure 5.2 Nombre d'opérations servies par memcached en fonction de la méthode de traçage (sélection d'évènements réduite)

Tableau 5.2 Nombre d'opérations servies par memcached (en milliers d'opérations)

# fils	Sans traçage	Traçage local		Traçage réseau (même)		Traçage réseau (diff)	
		Tous	Réduite	Tous	Réduite	Tous	Réduite
1	23	11	16	11	15	11	17
5	100	51	71	45	64	51	74
10	182	97	128	76	103	99	126
15	239	136	164	93	134	124	160
20	273	149	211	110	154	167	192
25	277	179	252	113	164	207	254
30	277	218	259	121	180	224	252
35	282	215	294	130	176	229	289
36	267	201	254	116	170	214	273

Tableau 5.3 Débit de données de traçage généré en fonction du mode de traçage (en Mio/s)

# fils	Sans traçage	Traçage local		Traçage réseau (même)		Traçage réseau (diff)	
		Tous	Réduite	Tous	Réduite	Tous	Réduite
1	-	8,8	3,6	9,2	3,3	9,0	3,7
5	-	36,5	15,9	33,7	14,3	35,5	16,4
10	-	68,3	29,2	49,8	24,0	57,4	28,5
15	-	96,1	39,2	50,6	33,6	53,6	37,8
20	-	106	50,4	60,2	38,7	72,5	45,2
25	-	128	61,9	57,8	42,8	74,5	60,0
30	-	157	66,0	59,1	47,9	70,3	63,0
35	-	157	77,4	64,5	48,4	72,1	75,4
36	-	147	67,2	55,8	48,0	77,7	69,4

Tableau 5.4 Évènements perdus en fonction du mode de traçage

# fils	Sans traçage	Traçage local		Traçage réseau (même)		Traçage réseau (diff)	
		Tous	Réduite	Tous	Réduite	Tous	Réduite
1	-						
5	-			3378659		2456947	
10	-			9137452		9382781	
15	-			14320257		15588129	19310
20	-			18167592	3041	19922867	190202
25	-			20746223	11996	25355095	747052
30	-			23463356	180796	28863393	471267
35	-			23748875	307231	29604835	504430
36	-			24230644	611770	30314552	1164407

complétés. Cette hypothèse est toutefois peu probable, comme nous le verrons un peu plus bas.

Finalement, il est aussi possible que notre banc de test soit la cause de cette limite. Autrement dit, n'importe quel composant entre le générateur de requêtes, la carte réseau de l'hôte, le protocole gigabit et la carte réseau de la carte pourrait être à l'origine de la limite d'environ 275000 requêtes par seconde.

Impact du traçage En ce qui a trait à l'impact du traçage, on constate que l'activation du traçage en mode local cause une baisse de performance d'un peu moins de 50%. En tentant de faire passer les données de traçage sur le réseau, sur la même interface que les requêtes, l'impact est bien plus grand, en divisant pratiquement par 3 le nombre de requêtes par secondes. Finalement, en déplaçant le flux de données de traçage sur l'autre interface, on retrouve une performance équivalente au traçage local. Toutefois, comme nous le verrons, cette solution n'est pas tout à fait adéquate puisqu'elle mène à la perte d'évènements.

En observant les données recueillies avec la sélection d'évènements réduite (figure 5.2), on voit le même genre d'impact, mais avec des proportions réduites. Bien que cette conclusion ne soit pas surprenante, elle montre le bénéfice de limiter l'activation des évènements noyau à ceux qui nous intéressent réellement.

La différence de performance entre les deux cas de traçage réseau est également conforme aux attentes. En utilisant un médium différent pour les données de traçage que pour les requêtes, on obtient un impact de base. En envoyant les données sur le même médium, on a encore ce même impact de base, mais on interfère en plus avec la transmission des requêtes sur le réseau. En concevant une stratégie de traçage, il est donc aussi important de ne pas seulement considérer son impact sur le processeur et la mémoire, mais aussi sur le reste du système. Un chemin hors-bande, dédié aux données de traçage, peut donc être très utile pour minimiser l'empreinte sur le système.

Une chose intéressante à noter sur le graphique 5.2 est qu'avec 35 ou 36 fils, la performance avec traçage local a rejoint celle sans traçage. On peut supposer que dans les deux cas, le débit de données vers la mémoire généré par memcached seul est le même. Comme le traçage génère une charge supplémentaire sur la mémoire, cela indique que celle-ci n'est pas à pleine capacité avec memcached uniquement. Cela élimine l'hypothèse que les rendements décroissants après 20 fils seraient dus à une saturation de la mémoire.

Débit de traçage et perte d'évènements Lorsqu'on s'intéresse aux pertes d'évènements, on constate que la sauvegarde locale des données de traçage noyau est possible. Par contre, la transmission en temps réel par le réseau n'est pas une solution envisageable. Dès

qu'on augmente un peu le nombre de cœurs actifs, le débit de traçage devient trop important. Le consommateur d'évènement, qui est responsable de transmettre les tampons pleins vers leur destination, en reçoit plus que ce qu'il est capable d'envoyer et est forcé de laisser tomber certains tampons.

Un réseau gigabit possède une limite théorique d'environ 120 Mio/s ($\approx$ 125 Mo/s $\approx$ 1000 Gb/s). En pratique, plusieurs facteurs font en sorte que la limite pratique est même bien en deçà de cette limite théorique. Dans le cas du traçage local, les données de traçage ont atteint un débit maximal de 157 Mio/s. Si on voulait ne pas perdre d'évènements, il faudrait que le réseau puisse transmettre ce débit de données de traçage, un débit plus élevé que la limite théorique du réseau Ethernet gigabit. Il est donc impensable que le traçage réseau puisse supporter une telle charge. Les débits de données de traçage les plus élevés que le réseau a pu transmettre pendant nos expérience sont autour de 75 Mio/s, une valeur réaliste pour la limite pratique du réseau gigabit.

Finalement, il est intéressant de noter que le système de fichier en mémoire sur la carte est *tmpfs*. Selon sa description dans la documentation du noyau Linux (Rohland *et al.*, 2010), l'implémentation de *tmpfs* place les données dans des pages mémoires dans la cache de page interne du noyau. Lorsque LTTng a rempli un tampon (un nombre entier de pages mémoires), le consommateur d'évènements utilise l'appel système *splice* pour devenir propriétaire des pages mémoires formant le tampon. Jusque là, aucune copie des données n'a été faite. Il fait ensuite appel à *splice* une seconde fois pour les transférer vers le descripteur de fichier correspondant au fichier de trace. Dans le cas normal d'un système de fichier sur disque dur, les données devraient éventuellement, au strict minimum, être copiées une fois de la mémoire vers le disque dur. Cependant, avec toutes les couches impliquées (système de fichiers, pilote du controleur de disque), il se peut que des copies intermédiaires soient faites. Dans le cas de *tmpsfs*, il est raisonnable de penser que l'écriture via *splice* d'une page mémoire ne causera aucune copie inutile. En effet, la page ne fait que passer d'un endroit à l'autre dans le noyau, elle ne fait que changer de main. Cela pourrait, en partie, expliquer qu'aucun évènement n'ait été perdu lors du traçage local.

Évidemment, il n'y a pas que des avantages à écrire la trace sur le système de fichiers en mémoire. Comme mentionné précédemment, la mémoire est une ressource très limitée sur ce genre de système. En écrivant à 157 Mio/s, le système de fichiers de 4 Gio de la carte serait rempli en environ 25 secondes. Il s'agit donc d'une solution applicable à de très courtes expériences ou à des expériences générant un fabile débit de données de traçage.

5.2.3 Intel Xeon Phi

Cette section présente les procédures ainsi que les résultats des tests que nous avons effectué sur la plate-forme Intel Xeon Phi.

Procédures de test

Pour effectuer les tests sur le processeur Xeon Phi, nous utilisons la carte Intel 3120P. Comme pour la plate-forme de Tilera, il s'agit d'une carte d'extension PCI-express faite pour être installée dans un serveur. Cependant, celle-ci ne possède pas de connexion réseau. La seule façon dont elle peut communiquer avec le monde extérieur est par le réseau virtuel passant par la connexion PCI-express.

Deux versions du programme de test sont utilisées : une s'exécutant de façon native sur le processeur et l'autre utilisant la fonction de délestage du compilateur icc. Le même programme est alors lancé sur l'hôte, et seule la fonction contenant réellement les éléments de calcul est exécutée sur le coprocesseur. Le temps d'exécution mesuré considère le temps du délestage.

Les trois variantes de traçage suivantes sont utilisées :

— Sans traçage : il s'agit du cas de référence

— Traçage local : les données sont enregistrées localement. Ici aussi, le système de fichiers est stocké en mémoire vive, qui est de 6 Gio.

— Traçage réseau : les données sont enregistrées sur l'hôte à l'aide de la fonction d'enregistrement distant de LTTng. Le démon *relayd* est lancé sur l'hôte et les données de trace passent par la connexion réseau virtuelle.

Ici encore nous utilisons des tampons de 1 Mio. Par contre, comme la plate-forme comporte 228 cœurs virtuels, cela porte la taille totale réservée aux tampons de traçage à 912 Mio (1 Mio / tampon $\times$ 4 tampons / cœur $\times$ 228 cœurs). Évidemment, plus on augmente le nombre de cœurs, plus l'impact de la taille des tampons est élevé. L'espace mémoire utilisé pour le traçage ne peut pas être utilisé par les applications. Le paradigme associant une série de sous-tampons à chaque cœur virtuel s'appliquait bien jusqu'à 8 ou 16 cœurs, mais il devra peut-être être repensé pour des processeurs comme celui-ci afin de garder raisonnable l'utilisation de la mémoire.

Résultats

Les résultats des tests effectués sont présentés sur la figure 5.3. Les données équivalentes se trouvent dans le tableau 5.5. Le tableau 5.6 présente les débits de traçage pour chaque cas alors que le tableau 5.7 indique le cas où il y a eu perte d'évènements. Pour mieux

comprendre la façon dont le nombre de cœurs affecte la performance de notre programme de test, nous l'avons exécuté avec un nombre de fils variant de un seul à 228. ces résultats sont présentés à la figure 5.4

Le test utilisé calcule automatiquement le nombre moyen d'opérations à point flottant effectués par seconde (FLOPS). Ce nombre est calculé en divisant le nombre total d'opérations effectuées par le temps d'exécution.

Performance sans traçage Selon le graphique 5.3, la performance semble assez bien suivre le nombre de cœurs jusqu'à concurrence d'environ 100 cœurs. Par le suite, on voit même une régression. L'observation du graphique 5.4 donne une explication partielle. On y voit une croissance régulière, bien que non linéaire, de la performance jusqu'à 57 fils. Or, le Xeon Phi sur lequel nous travaillons possède 57 cœurs physiques, chacun possédant 4 cœurs virtuels. Avec exactement un fil de travail par cœur physique, nous obtenons une division du travail équitable. Toutefois, si on ajoute un fil au programme, un des cœurs physiques devra nécessairement accueillir deux fils. Ces deux fils, étant sur le même cœur, devront se partager les ressources, notamment le temps de processeur et l'espace en cache. Ils s'exécuteront donc les deux moins vite que le reste des fils. Ainsi, même lorsque la majorité des fils auront terminé leur travail, le programme devra attendre que ces deux fils finissent leur travail avant de terminer son exécution. Puisque le facteur important est le temps d'exécution total du programme, la performance est globalement moins bonne avec 58 fils qu'avec 57. Ce comportement est observé parce que le comportement par défaut de l'implémentation d'OpenMP par Intel est de diviser l'ensemble du travail entre les fils avant que ceux-ci ne se mettent à travailler. D'autres options, comme l'ordonnancement dynamique de morceaux de travail ou le vol de tâches permettraient de contrebalancer cet effet.

On voit des baisses similaires, quoique moins prononcées, à 114 fils (57×2) et 171 fils (57×3).

Impact du traçage Dans ce cas-ci, l'impact du traçage est si minime qu'il se perd dans le bruit des variations entre les différents lancements du programme. Par sa nature, nous nous attendions effectivement à ce que ce genre de programme soit moins affecté par le traçage que celui exécuté sur le TILE-Gx. En effet, celui-ci comporte essentiellement du calcul pur, effectué en mode utilisateur. Cela a pour conséquence de générer très peu d'évènements noyau. Comme les opérations de délestage impliquent des transferts via la connexion PCI-express, ce qui génère une certaine activité du noyau, nous aurions pu nous attendre à un ralentissement en traçant les versions avec délestage. Nous n'avons toutefois pas observé un tel comportement.

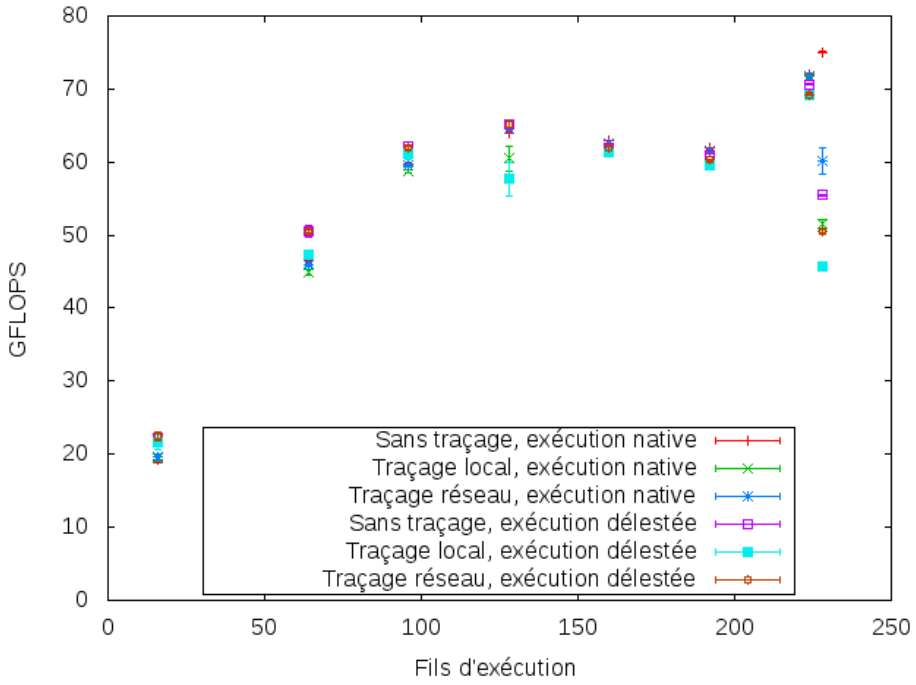

Figure 5.3 GFLOPS en fonction des modes d'exécution et de traçage

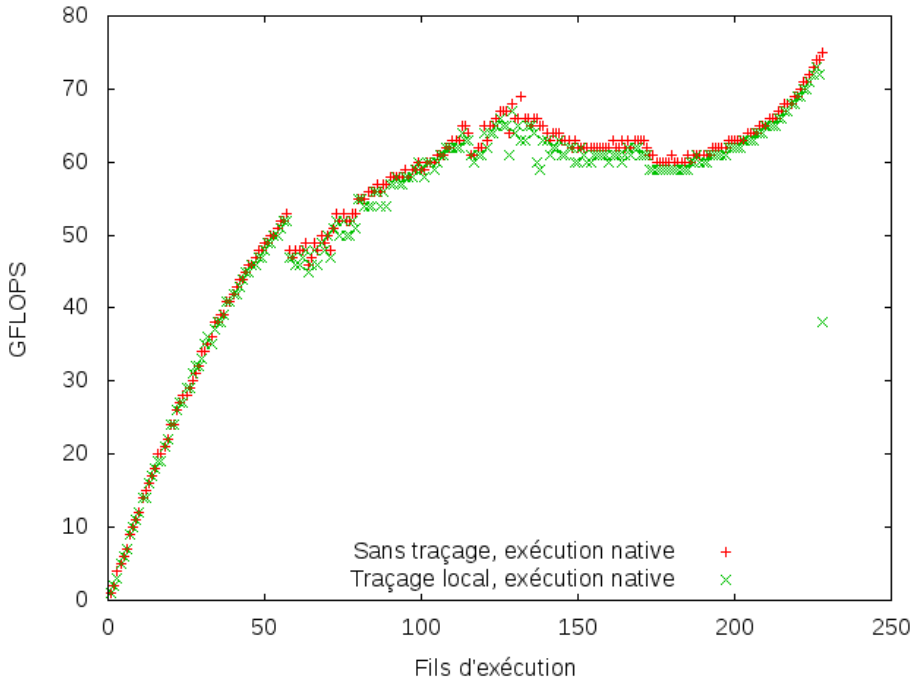

Figure 5.4 GFLOPS en fonction du mode de traçage, 1 à 228 fils

Tableau 5.5 Performance de l'algorithme en GFLOPS en fonction des modes d'exécution et de traçage

# fils	Exécution native			Exécution avec délestage		
	Sans	Local	Réseau	Sans	Local	Réseau
16	19,3	19,4	19,6	22,2	21,7	22,5
64	46,6	44,9	45,9	50,5	47,3	50,5
96	59,8	58,7	59,5	62,1	61,2	62,0
128	64,0	60,5	64,5	65,3	57,7	65,1
160	63,0	61,7	62,5	61,9	61,3	62,0
192	61,9	60,9	61,5	60,9	59,5	60,3
224	72,0	71,6	71,8	70,7	69,2	69,3
228	75,1	51,4	60,2	55,5	45,7	50,6

Tableau 5.6 Débit de données de traçage généré en fonction des modes d'exécution et de traçage (en Mio/s)

# fils	Exécution native			Exécution avec délestage		
	Sans	Local	Réseau	Sans	Local	Réseau
16	-	1,5	1,4	-	2,1	1,6
64	-	14,7	14,0	-	17,1	13,9
96	-	8,5	7,4	-	7,8	6,4
128	-	21,6	15,5	-	28,5	15,1
160	-	13,1	10,9	-	9,4	7,8
192	-	18,9	17,4	-	16,2	15,2
224	-	15,3	14,9	-	12,8	12,4
228	-	57,0	49,7	-	31,4	34,0

Tableau 5.7 Évènements perdus en fonctions modes d'exécution et de traçage

# fils	Exécution native			Exécution avec délestage		
	Sans	Local	Réseau	Sans	Local	Réseau
16						
64						
96						
128					162210	
160						
192						
224			10158			
228		18709660	7309543		2609710	4783698

On peut par contre remarquer le comportement particulier lorsque 228 fils sont utilisés. Sans traçage, la performance à 228 suit de façon régulière la tendance des points précédents. Toutefois, avec le traçage activé, la performance subit une chute importante. Il est probable que cette baisse puisse être expliquée de la même façon que la baisse entre 57 et 58 fils. Lorsqu'on utilise 228 fils sans traçage, l'ordonnanceur a exactement 228 tâches à exécuter sur 228 cœurs, résultant en une bonne distribution du travail. Tous les autres processus sont soit inactifs, soit demandent rarement à être exécutés et ont donc un impact minime.

Toutefois, en activant le traçage, on ajoute le processus consommateur de LTTng à l'équation. En demandant à être exécuté, il vient nécessairement interrompre l'exécution d'un fil de l'application et donc retarde l'échéance de son exécution.

Débit de traçage et perte d'évènements Comme mentionné précédemment, le débit de traçage est plutôt faible. Cela se traduit par une faible quantité d'évènements perdus lorsque moins de 228 fils sont utilisés par l'application. Toutefois, lorsque 228 fils sont utilisés, nous avons observé des niveaux assez impressionants d'évènements perdus. Il semble donc que lorsqu'il doive partager un cœur du Xeon Phi avec un autre processus, le consommateur n'arrive pas à tenir le coup. Il est donc plus efficace de réserver un cœur aux processus autres que l'application principale, afin de permettre au consommateur et autres services du système d'exploitation de s'exécuter sans se piler sur les pieds.

Bien que le traçage noyau ne génère pas de volume important de données, il a toutefois son utilité. Pour le développeur d'une application de calcul telle que celle-ci, le traçage noyau permet de constater l'efficacité de son implémentation. Il peut d'abord vérifier combien de temps les fils passent en état d'exécution par rapport au temps passé en attente. Une mauvaise répartition du travail ou des points de rendez-vous trop fréquents seront visibles puisque la plupart des fils passeront une bonne partie de leur temps bloqués.

CHAPITRE 6

DISCUSSION GÉNÉRALE

Cette section résume les discussions présentées dans les sections 4 et 5.

En ce qui concerne le problème du débogage des appels à distance, tel que présenté dans la section 4, les résultats de nos travaux consistent en un module d'extension au débogueur GDB. La solution apportée est assez générique pour être appliquée à plusieurs bibliothèques d'appels à distance et même à des programmes implémentant eux-mêmes le paradigme.

La complexité de l'implémentation des portions spécifiques varie selon les bibliothèques utilisées. Dans certains cas, l'instanciation d'un seul objet est suffisante pour avoir un support de base d'une bibliothèque, alors que dans d'autres, un peu plus d'efforts sont requis pour être capable de suivre l'appel du client sur le serveur. La qualité de la solution varie aussi en fonction de la façon dont la bibliothèque est conçue. Dans certains cas, on arrive à relier une seule fonction de bas niveau côté client à une fonction équivalente côté serveur. Cela oblige l'utilisateur à se promener un peu dans le code de la bibliothèque. Dans d'autres, on arrive à éviter tout le code intermédiaire et à offrir une meilleure expérience à l'utilisateur.

En rétrospective, les bibliothèques qui demandent un traitement avant la compilation (par exemple, génération de code) sont plus difficiles à adapter. Le code automatiquement généré est difficile à distinguer du code de l'utilisateur. À l'opposée, les bibliothèques qui demandent d'effectuer tout le travail à l'exécution (comme l'enregistrement des fonctions de rappel sur le serveur) sont plus simples. Il est facile pour le débogueur de surveiller les appels aux fonctions d'enregistrement et de se construire un portrait précis des fonctions exportées par le serveur.

Le fait que GDB soit un programme à source ouverte nous a permis d'apporter les quelques changements nécessaires pour faire fonctionner notre solution. Nous avons d'abord réglé un problème d'accès mémoire empêchant un point d'arrêt d'être installé dans un programme en exécution. Ensuite, des fonctions ont été ajoutées à l'interface de programmation Python pour permettre le contrôle (interruption et reprise) des processus débogués.

En ce qui concerne le deuxième volet du travail, nous avons évalué la performance du traceur noyau LTTng sur deux plateformes à grand nombre de cœurs, le Tilera TILE-Gx et l'Intel Xeon Phi.

Nous avons d'abord porté LTTng à ces deux architectures. Dans le cas du processeur Tilera, plusieurs modifications au noyau et à LTTng ont été nécessaires pour pouvoir obtenir une trace noyau. Ces modifications étaient attribuables autant au fait que cette architecture

est très récente et qu'elle est quelque peu exotique. Dans le cas du processeur Intel, le port était presque trivial.

Pour évaluer l'impact en performance de LTTng, nous avons sélectionné des programmes en fonction du domaine d'application prévu de ces deux processeurs. Selon Tilera, le TILE-Gx est notamment destiné à servir diverses fonctions en infonuagique. Nous l'avons testé en tant que serveur *memcached*, un serveur de cache distribuée sur le réseau. De plus, nous avons testé plusieurs modes d'enregistrement de la trace : localement et en réseau. Puisqu'il s'agit d'une application générant un grand volume de données de traçage noyau, l'envoi par réseau en temps réel de la trace était tout simplement trop lourd. L'enregistrement local était assez efficace, mais en imposant une importante limitation d'espace.

Le Xeon Phi cible quant à lui le marché du calcul de haute performance. Les programmes qui y seront exécutés en seront donc de calcul scientifique, de physique, de météorologie, etc. Nous avons donc choisi un programme de ce type, simulant la diffusion dans un liquide en trois dimensions. Ici, vu la nature de l'application, beaucoup moins de données sont générées par le traçage noyau. Aucun problème majeur n'empêche le traçage local ou par réseau virtuel (PCI-express). Il était toutefois très avantageux de garder un cœur du processeur libre afin que le processus consommateur du traceur puisse faire son travail.

CHAPITRE 7

CONCLUSION

Cette dernière section présente un résumé des problèmes abordés, des solutions apportées ainsi que des résultats obtenus. Nous décrivons ensuite leurs limitations et nous proposons des améliorations possibles.

7.1 Synthèse des travaux

Dans le premier volet de ce mémoire, nous avons proposé une solution permettant le débogage transparent des appels à distance entre un client et un serveur. La solution utilise le débogueur GDB, mais le concept derrière la solution est assez générique pour qu'il puisse être appliqué à n'importe quel débogueur possédant des fonctions équivalentes. L'idée générale est de placer des points d'arrêt à tous les endroits d'un programme qui correspondent à l'initiation d'un appel à distance. À partir d'un de ces points d'arrêt, le débogueur extrait les informations nécessaires du client pour permettre de placer un point d'arrêt dans le programme serveur, au début de la fonction appelée par le client. La séquence des évènements étant très similaire dans toutes les bibliothèques d'appels à distance, il est possible d'extraire une grande portion de la logique dans un module générique.

Dans le second volet, nous avons étudié l'impact en performance du traceur LTTng sur deux applications s'exécutant sur des processeurs à grand nombre de cœurs. Nous avons donc adapté LTTng pour le faire fonctionner sur les architectures Tilera TILE-Gx et Intel Xeon Phi. Pour des applications ayant une grande interaction avec le noyau et les périphériques, les données de traçage générées peuvent être considérable et peuvent facilement surpasser la capacité d'enregistrement, que ce soit en mémoire locale (trop petite) ou par le réseau (trop lent).

7.2 Limitations de la solution proposée et améliorations futures

La solution proposée dans le chapitre 4 contribue à améliorer l'expérience des développeurs travaillant avec des appels à distance. Toutefois, le domaine d'application de la solution souffre de plusieurs limitations.

La solution actuelle profite du fait qu'une seule instance du débogueur GDB peut gérer plusieurs processus en même temps. Ainsi, un même script exécuté dans le contexte du

débogueur peut interagir à la fois avec le client et le serveur. Par contre, GDB ne peut pas déboguer plusieurs plateformes à la fois, ce qui force les deux processus à être sur la même plateforme. Il est par exemple possible de déboguer plusieurs processus de façon native, ou encore plusieurs processus à distance (par le biais de `gdbserver`). Il n'est toutefois pas possible de déboguer un processus local en même temps qu'un processus distant, ou encore de se connecter à deux plateformes distantes simultanément.

Cela pose un problème évident : le client et le serveur que nous déboguons doivent s'exécuter sur le même hôte. Pendant la phase de développement, cette limitation n'est peut-être pas très importante, puisque les développeurs vont exécuter les deux parties du programme sur leur station de travail. Toutefois, de façon générale, comme on veut que les deux intervenants d'un appel à distance se trouvent sur des hôtes différents, la solution proposée peut sembler contre productive. Aussi, même lors du développement, il se peut que les deux parties du programme s'exécutent sur des architectures différentes (par exemple, un client Intel et un serveur TILE-Gx). Il est alors impossible des les exécuter sur le même hôte, même temporairement.

Pour pouvoir appliquer la solution à deux processus s'exécutant sur deux hôtes différents, nous envisagerions deux améliorations. D'abord, il serait possible d'étendre GDB pour qu'il supporte plusieurs plateformes simultanément. L'utilisateur pourrait alors déboguer un mélange de processus natifs et distants (même sur plusieurs hôtes distants). Cela demanderait un travail de refonte de plusieurs sous-sections de GDB. Un exemple simple de problème attendu est le fait que plusieurs endroits identifient un processus débogué par son PID. Si GDB débogue simultanément des processus de plusieurs hôtes, il est très possible que deux d'entre eux aient le même PID. Bien qu'il n'y ait pas de date prévue de complétion, ce genre de support est considéré par les développeurs de GDB (Tromey, 2014).

Une fois ce projet complété, il pourrait être possible de déboguer le client de façon native et le serveur de façon distante. Grâce à une importante contribution de Alves (2013), GDB devrait même être capable de déboguer simultanément des programmes d'architectures différentes.

Une autre approche à ce problème pourrait être d'implémenter la solution à un autre niveau, par exemple dans l'environnement CDT d'Eclipse. Lorsque CDT démarre une séance de débogage, il démarre une instance de GDB pour chaque processus (ce qui permet notamment de conserver la compatibilité avec les versions de GDB antérieures au support multi-processus). Il est ainsi possible de choisir une version de GDB différente et des paramètres différents pour chaque processus. Rien n'empêche de démarrer un GDB pour Intel x86 pour déboguer le client localement et un GDB TILE-Gx pour déboguer le serveur distant. La logique de détection des appels à distance devrait donc être implémentée au sein de CDT.

Toutes les fonctions et tous les évènements utilisés par notre solution ont un équivalent dans l'interface machine de GDB, que CDT utilise. Il s'agit en fait de la solution adoptée par Intel pour l'implémentation du débogage de fonctions délestées sur le Xeon Phi.

Nous avons également mentionné plus tôt la difficulté de travailler avec les bibliothèques utilisant la génération de code. Ces bibliothèques demandent souvent de définir l'interface du serveur dans un format spécifique et génèrent ensuite du code basé sur celle-ci. L'utilisateur compile ensuite ce code avec le reste du code de son application. Il devient donc difficile pour le débogueur de différencier le vrai code de l'utilisateur du code généré. Nous devons alors nous contenter de travailler avec les fonctions de base de la bibliothèque qui sont éventuellement appelées par le code généré, obligeant l'utilisateur à faire quelques pas dans du code qui ne l'intéresse probablement pas.

Pour pallier à ce problème, les outils de génération de code pourraient ajouter au code des informations sur les fonctions générées que les débogueurs pourraient utiliser. On pourrait penser à un tableau associant les noms des fonctions auxquelles le client fait appel, aux noms des fonctions correspondantes côté serveur.

Une autre limitation de la solution est qu'elle ne supporte qu'un seul niveau d'appels à distance. Dans des systèmes plus complexes, un programme A peut faire un appel à un programme B qui à son tour fait un appel à un programme C. Il serait intéressant d'étendre l'implémentation pour supporter un nombre arbitraire de niveaux d'appels à distance.

En ce qui a trait au traçage, l'évaluation de la performance de LTTng sur les deux plateformes a permis de constater plusieurs endroits où des améliorations sont souhaitables.

Tout d'abord, force est d'admettre que le traçage noyau sur la carte Tilera TILEncore-Gx est limité. La seule option fonctionnelle est d'enregistrer la trace en mémoire, dans un espace plutôt restreint. Une voie à explorer serait de transmettre les données via la connexion PCI-express 16x. Puisqu'une connexion de ce type supporte des débits beaucoup plus importants que la connexion réseau gigabit, il y aurait davantage de chance que la trace puisse être transférée en temps réel sans perte. Concrètement, dans LTTng, cela prendrait la forme d'un nouveau type de consommateur. Celui-ci utiliserait l'interface de programmation de Tilera, qui permet de communiquer assez facilement avec l'hôte via le bus PCI-express. Nous avons tenté de concevoir un prototype de cette solution, mais des problèmes par rapport à la gestion des pages mémoire dans le noyau se sont présentés. Quelques petites embûches techniques sont donc à prévoir de ce côté.

En ce qui concerne le Xeon Phi, il serait intéressant de tester le traçage dans des situations plus demandantes, même si elles sont moins représentatives de la réalité. Cela permettrait d'avoir une meilleure idée de la limite actuelle du traçage sur cette plateforme et pourrait dévoiler d'éventuels problèmes de performance à régler. On pourrait penser à des applications

distribuées sur plusieurs nœuds Xeon Phi, communiquant à l'aide de la bibliothèque MPI. La présence de communication réseau entrainerait déjà une plus grande activité noyau, donc une plus grande pression du traçage noyau.

Pour analyser les interactions entre l'hôte et l'une des deux cartes d'extension, il serait intéressant de tracer à la fois la carte et l'hôte. Il faudrait ensuite vérifier parmi les méthodes de synchronisation de traces existantes et voir comment elles peuvent être appliquées aux interactions PCI-express. Au besoin, des points de trace pourraient être ajoutés au noyau aux endroits nécessaires.

Finalement, une amélioration intéressante pourrait être apportée du côté des logiciels d'analyse. Comme nous l'avons vu, il y a un avantage assez important à choisir précisément les évènements du noyau à activer plutôt que de tous les activer. Par contre, on ne peut pas s'attendre à ce que les utilisateurs des outils sachent exactement de quels évènements chaque analyse a besoin. Ces outils devraient donc offrir une façon de configurer le traçage en fonction de l'analyse, ou du moins dresser une liste d'évènements nécessaires en fonction des analyses choisies.

RÉFÉRENCES

AGANS, D. J. (2002). *Debugging : the 9 indispensable rules for finding even the most elusive software and hardware problems*. American Management Association.

ALVES, P. (2013). [GDBserver] Multi-process + multi-arch. `https://sourceware.org/ml/gdb-patches/2013-05/msg01057.html`.

BANKS, G. (2006). rpcdebug(8) - Linux man page. `http://manpages.debian.net/cgi-bin/man.cgi?query=rpcdebug&apropos=0&sektion=0&manpath=Debian+7.0+wheezy&format=html&locale=en`.

BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M. et STEELE, K. (2011). Many-core key-value store. *Green Computing Conference and Workshops (IGCC), 2011 International*. 1–8.

BIRRELL, A. D. et NELSON, B. J. (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2, 39–59.

COMER, D. (2004). *Computer networks and internets : with Internet applications*. Pearson/Prentice Hall, Upper Saddle River, N.J, quatrième édition.

CRAMER, T., SCHMIDL, D., KLEMM, M. et AN MEY, D. (2012). OpenMP Programming on Intel Xeon Phi Coprocessors : An Early Performance Comparison.

DESFOSSEZ, J. et DESNOYERS, M. (2011). LTTng-UST vs SystemTap userspace tracing benchmarks. `https://sourceware.org/ml/systemtap/2011-q1/msg00244.html`.

DESNOYERS, M. (2008). Using the Linux Kernel Tracepoints. `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`.

DESNOYERS, M. (2012). Man page of LTTNG-UST. `http://lttng.org/files/doc/man-pages/man3/lttng-ust.3.html`.

DESNOYERS, M. et DAGENAIS, M. R. (2006). The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux. *OLS (Ottawa Linux Symposium)*. vol. 2006, 209–224.

DESNOYERS, M., DESFOSSEZ, J. et GOULET, D. (2012). LTTng 2.0 : Tracing for power users and developers - part 1. `https://lwn.net/Articles/491510/`.

FINK, M. (2013). metrics 0.2.6. `https://pypi.python.org/pypi/metrics/`.

FREE SOFTWARE FOUNDATION (2013). rpctrace. `https://www.gnu.org/software/hurd/hurd/debugging/rpctrace.html`.

FREE SOFTWARE FOUNDATION (2014a). *Debugging with GDB*, chapitre Non-Stop Mode.

FREE SOFTWARE FOUNDATION (2014b). *Debugging with GDB*, chapitre Python API.

FREE SOFTWARE FOUNDATION (2014c). *Debugging with GDB, The gdb/mi Interface*.

FREE SOFTWARE FOUNDATION (2014d). GDB : The GNU project debugger. `https://www.gnu.org/software/gdb/`.

FREEDESKTOP.ORG (2013). D-Bus Bindings. `http://www.freedesktop.org/wiki/Software/DBusBindings`.

FREEDESKTOP.ORG (2014a). D-Bus. `http://www.freedesktop.org/wiki/Software/dbus/`.

FREEDESKTOP.ORG (2014b). D-Bus Documentation. `http://dbus.freedesktop.org/doc/api/html/index.html`.

GARANTIA DATA (2013). memtier_benchmark : A High-Throughput Benchmarking Tool for Redis & Memcached. `http://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached`.

GNOME (2012). GIO Reference Manual, Highlevel D-Bus Support. `https://developer.gnome.org/gio/2.36/gdbus-convenience.html`.

GNOME (2013a). D-Feet. `http://live.gnome.org/DFeet/`.

GNOME (2013b). GObject Reference Manual, Signals. `https://developer.gnome.org/gobject/stable/signal.html`.

GROTKER, T., HOLTMANN, U., KEDING, H. et WLOKA, M. (2008). *The Developer's Guide to Debugging*. Springer.

INTEL (2012). Debugging Intel Xeon Phi Applications on Linux Host. `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`.

INTEL (2012). Intel Debugger for Linux (IDB). `http://software.intel.com/en-us/articles/idb-linux`.

INTEL (2013a). GDB - The GNU Project Debugger for Intel Architecture. `http://software.intel.com/en-us/articles/intel-system-studio-gdb`.

INTEL (2013b). *Intel C++ Composer XE 2013 for Linux Installation Guide and Release Notes*.

INTEL (2013). The Intel Xeon Phi Product Family. `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf`.

INTEL (2014). http://software.intel.com/en-us/articles/debugging-intel-xeon-phi-applications-on-linux-host.

JEFFERS, J., JEFFERS, J. et REINDERS, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science & Technology Books.

KENISTON, J., PANCHAMUKHI, P. et HIRAMATSU, M. (2006). kprobes (Linux Kernel Documentation). https://www.kernel.org/doc/Documentation/kprobes.txt.

KERNELNEWBIES (2008). Kernel 2.6.27. http://kernelnewbies.org/Linux_2_6_27.

KIDD, E. (2001). XML-RPC for C and C++. http://xmlrpc-c.sourceforge.net.

LI, S. (2011). Case Study : Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework. http://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise.

LLDB PROJECT (2014a). LLDB Python Reference. http://lldb.llvm.org/python-reference.html.

LLDB PROJECT (2014b). LLDB Status. http://lldb.llvm.org/status.html.

LLDB PROJECT (2014c). The LLDB Debugger. http://lldb.llvm.org.

LTTNG (2014). LTTng Project. http://lttng.org.

MASCI, F. (2013). Benchmarking the intel xeon phi coprocessor. Rapport technique, Infrared Processing and Analysis Center, Caltech.

MEMCACHED (2012). memcached documentation : threads.txt. https://github.com/memcached/memcached/blob/master/doc/threads.txt.

MEMCACHED (2014). memcached. http://memcached.org.

MICROSOFT (2010a). How to : Create a Windows Communication Foundation Client. http://msdn.microsoft.com/en-us/library/vstudio/ms733133(v=vs.100).aspx.

MICROSOFT (2010b). How to : Define a Windows Communication Foundation Service Contract. http://msdn.microsoft.com/en-us/library/vstudio/ms731835(v=vs.100).aspx.

MICROSOFT (2012a). Fundamental Windows Communication Foundation Concepts. http://msdn.microsoft.com/en-us/library/ms731079(v=vs.110).aspx.

MICROSOFT (2012b). WCF Test Client (WcfTestClient.exe). http://msdn.microsoft.com/en-us/library/bb552364(v=vs.110).aspx.

MICROSOFT (2013a). COM, DCOM, and Type Libraries. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366757(v=vs.85).aspx.

MICROSOFT (2013b). Debugger roadmap. http://msdn.microsoft.com/en-us/library/k0k771bt.aspx.

MICROSOFT (2013c). Debugging COM Clients and Servers Using RPC Debugging. http://msdn.microsoft.com/en-us/library/aa734022(v=vs.60).aspx.

MICROSOFT (2013d). Defining COM interfaces. http://msdn.microsoft.com/en-us/library/windows/desktop/ms688488(v=vs.85).aspx.

MICROSOFT (2013e). How to : Debug COM Clients and Servers Using RPC Debugging. http://msdn.microsoft.com/en-us/library/kf71skzd.aspx.

MICROSOFT (2013f). How to : Step into WCF services. http://msdn.microsoft.com/en-us/library/bb157688.aspx.

MICROSOFT (2013g). Limitations on WCF Debugging. http://msdn.microsoft.com/en-us/library/bb157687.aspx.

MICROSOFT (2013h). The Component Object Model. http://msdn.microsoft.com/en-us/library/ms694363(v=vs.85).aspx.

NELSON, B. J. (1981). Remote procedure call.

NEWBURN, C. J., DEODHAR, R., DMITRIEV, S., MURTY, R., NARAYANASWAMY, R., WIEGERT, J., CHINCHILLA, F. et MCGUIRE, R. (2013). Offload Compiler Runtime for the Intel Xeon Phi Coprocessor. J. M. Kunkel, T. Ludwig et H. W. Meuer, éditeurs, *Supercomputing*, Springer Berlin Heidelberg, vol. 7905. 239–254.

OPERSYS (2012). Linux Trace Toolkit - Who's Behind LTT. http://www.opersys.com/ltt/whoami.html.

ORACLE (2010). rpcgen tutorial. http://docs.oracle.com/cd/E19683-01/816-1435/rpcgenpguide-21470/index.html.

QEMU (2013). QEMU documentation / debugging. http://wiki.qemu.org/Documentation/Debugging.

REDIS PROJECT (2014). Redis. http://redis.io.

ROHLAND, C., DICKINS, H. et KOSAKI, M. (2010). tmpfs (Linux Kernel Documentation). https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt.

SIDWELL, N., PRUS, V., ALVES, P., LOOSEMORE, S. et BLANDY, J. (2008). Non-stop Multi-threaded Debugging in GDB. *GCC Developers' Summit*. 117.

SOFTWARE, R. (2012). Intel Xeon Phi Coprocessor. http://www.roguewave.com/technologies/xeon-phi.aspx.

SRINIVASAN, R. (1995a). RPC : Remote Procedure Call Protocol Specification Version 2. RFC 1831 (Proposed Standard). Obsoleted by RFC 5531.

SRINIVASAN, R. (1995b). XDR : External Data Representation Standard. RFC 1832 (Draft Standard). Obsoleted by RFC 4506.

ST. LAURENT, S. (2001). *Programming Web services with XML-RPC*. O'Reilly, première édition.

SUN MICROSYSTEMS (1988). RPC : Remote Procedure Call Protocol specification : Version 2. RFC 1057 (Informational).

SYSTEMTAP (2014). SystemTap. `https://sourceware.org/systemtap/wiki`.

THURLOW, R. (2009). RPC : Remote Procedure Call Protocol Specification Version 2. RFC 5531 (Draft Standard).

TILERA (2012a). *Programming the Tile-GX Processor (UG505)*.

TILERA (2012b). TILE-Gx8036 Processor, Specification Brief. `http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf`.

TILERA (2012c). *Tile Processor Architecture Overview for the Tile-GX Series (UG130)*.

TILERA (2014). Tile-gx processor family. `http://www.tilera.com/products/processors/TILE-Gx_Family`.

TRANSMISSION (2014). Transmission. `http://www.transmissionbt.com`.

TROMEY, T. (2014). GDB Wiki : MultiTarget. `https://sourceware.org/gdb/wiki/MultiTarget`.

VAJDA, A. (2011a). Debugging and Performance Analysis of Many-core Programs. *Programming many-core chips*, Springer.

VAJDA, A. (2011b). Introduction. *Programming many-core chips*, Springer.