Titre: Title:	Points de trace rapides et flexibles en X86
Auteur: Author:	Christian Harper-Cyr
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Harper-Cyr, C. (2018). Points de trace rapides et flexibles en X86 [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3736/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3736/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

POINTS DE TRACE RAPIDES ET FLEXIBLES EN X86

CHRISTIAN HARPER-CYR
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)
DÉCEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

POINTS DE TRACE RAPIDES ET FLEXIBLES EN X86

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

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

DÉDICACE

Je dédicace ce mémoire à mon grand-père, Peter Harper. Son amour de la science a été pour moi une inspiration dans la poursuite de mes études tout au long de ma vie. Sans son support, je n'aurais pas pu me rendre où je suis aujourd'hui. Il est, et sera toujours, un modèle pour moi.

REMERCIEMENTS

Je tiens tout d'abord à remercier le professeur Michel Dagenais pour l'opportunité de ce projet de maîtrise. Son expertise et son aide ont été très utiles et appréciées durant le cours de ce projet.

Je tiens aussi à remercier Jason Puncher, François Tétrault, Jim MacDonald et Octavian Stelescu de Ciena Inc. ainsi que Mathieu Desnoyers d'Efficios pour leur aide précieuse au cours de ce projet. Ces personnes m'ont fourni des directions et des cibles importantes pour l'accomplissement de ce projet.

Le support financier de NSERC, Prompt, Ericsson, Ciena, Google et EfficiOS est sincèrement apprécié.

Je voudrais aussi remercier Ahmad Shahnejat et Geneviève Bastien du laboratoire DORSAL pour tout le soutien dont j'avais besoin durant ma maîtrise.

Je tiens finalement à remercier ma famille et mes amis pour le support moral dont j'avais grandement besoin pour l'accomplissement de ce projet.

RÉSUMÉ

Le traçage est une technique très utile pour les développeurs afin de trouver les problèmes dans les systèmes complexes, et pour analyser l'exécution de ceux-ci. Le traçage permet d'enregistrer les informations de bas niveau d'un système avec un dérangement minimal de celui-ci. Ces informations présentent une vue d'ensemble de l'exécution du système sous test et permettent d'analyser son comportement. Plusieurs outils de traçage permettent à un développeur d'instrumenter un système en y ajoutant des appels d'enregistrement d'information. L'utilisation de ces outils n'est par contre pas toujours possible. En effet, il est nécessaire de recompiler et redéployer le système afin de modifier la définition des points de trace existants et d'en ajouter.

Pour pallier à ce problème, des outils de traçage avec instrumentation dynamique peuvent être utilisés. Ceux-ci permettent l'ajout de traçage dans un programme, sans modification des exécutables concernés. Certains outils permettent aussi l'ajout d'instrumentation dans un processus en exécution avec un minimum de dérangement de celui-ci. Ainsi, il est possible pour un développeur d'instrumenter un programme déployé sans déranger l'exécution de celui-ci.

La modification dynamique de processus en exécution est un procédé délicat qui demande une modification du code machine sans changement apparent à l'exécution de celui-ci. L'architecture multi-processeur des systèmes modernes pose un problème majeur pour la modification de processus en exécution : l'exécution d'opérations invalides durant la modification du processus. En effet, les fils d'exécution d'un système ne doivent pas être affectés durant la modification du code machine par un des fils. Plusieurs techniques peuvent être utilisées afin de modifier sans danger les programmes multi-fils en exécution.

Tout d'abord, un arrêt complet du processus cible peut être effectué afin d'insérer sans encombre le point de trace. Une fois le processus cible arrêté, l'outil peut sécuritairement insérer un branchement vers le code de traçage. Ceci assure qu'aucun fil n'exécutera une instruction invalide durant l'insertion. Ceci n'est parfois pas possible dans certains systèmes haute performance où l'arrêt du système affecte trop la performance. En effet, tous les fils d'exécutions doivent être arrêtés afin d'assurer la sécurité de l'insertion. Ceci est inacceptable dans certaines situations.

Afin de pallier à ce problème, une modification atomique du processus tracé peut être faite. Une instruction de branchement peut être insérée au point d'instrumentation afin d'exécuter le code de traçage. Ceci permet de réduire considérablement le coût de l'insertion du point de

trace. Il est par contre difficile d'insérer sécuritairement un branchement à la position d'une instruction de moins de 5 octets. En effet, ce branchement de 5 octets pourrait remplacer plusieurs instructions, ce qui pourrait amener un des autres fils à exécuter une instruction invalide à la position d'une instruction remplacée. Une instruction d'interruption d'un octet peut être utilisée dans ce cas. Cette instruction peut être sécuritairement insérée pour remplacer n'importe quelle instruction du processus. Cette instruction est par contre beaucoup plus lente à exécuter qu'un branchement, ce qui la rend inutilisable dans les systèmes haute performance.

Ce projet de recherche présente une nouvelle technique d'insertion de point de trace qui permet d'augmenter significativement le nombre de points d'insertion disponibles de points de trace avec branchement. Lorsque plusieurs instructions seraient remplacées par un branchement, des instructions d'interruptions sont incorporées au déplacement du branchement au début de chaque instruction remplacée. Ceci règle les problèmes présents lors de l'insertion de points de trace avec branchement. En effet, l'utilisation de ces interruptions garantit qu'aucun proccesseur n'exécutera d'instruction invalide. Un fil exécutant l'interruption sera tout simplement redirigé vers l'instruction équivalente créée lors de l'insertion du point de trace. Cette technique permet donc à un utilisateur l'insertion de points de trace performants dans la grande majorité des cas.

ABSTRACT

Tracing is a very useful technique used by developers in order to find problems in complex systems, and to analyze the execution of these systems. Tracing is the gathering of low level information about a system, disturbing said system as little as possible. A multitude of tracing tools enable a developer to instrument a system by adding tracepoints. The use of these tools is not always possible, however, since a recompilation and redeployment of the target system is necessary to modify existing tracepoints or add new ones. Indeed, in most cases, it is impossible to modify the tracing instrumentation of a running system.

To resolve this problem, tracing tools with dynamic instrumentation can be used instead. These tools enable a user to add and modify the tracepoints of a running system, while affecting as little as possible the execution of this system.

The dynamic modification of a running process is a delicate procedure that requires changing the machine code without affecting the execution of the process. The multi-core architecture of modern systems introduces a major hassle when trying to dynamically modify machine code: the execution of invalid instructions during the modification. Indeed, it is important that no thread be affected during the modification of a process. Multiple techniques can be used to safely modify running multi-threaded processes.

First of all, stopping all threads of a process is an effective technique to insert a tracepoint into a running process. While a process is stopped, a tool can safely add instructions that branch to tracing code. This ensures that no thread is executing invalid instructions during the insertion. Every thread must be stopped to guarantee the safety of the insertion. It is not always possible to stop every thread in performance sensitive situations.

An atomic modification of the target process can be used instead to solve this problem. A branch instruction to tracing code can atomically be placed at the insertion location. This reduces considerably the performance cost of the insertion. On the other hand, it is very difficult to safely insert a branch instruction at the location of an instruction of less than 5 bytes long. The branch instruction will overwrite multiple instructions. In this case, a thread could execute an invalid instruction at the position of one of the replaced instructions. In this case, a 1 byte interrupt instruction can be used instead. This instruction can be placed anywhere safely. Unfortunately, this instruction is slower to execute than a branch instruction, which makes it impossible to use in performance sensitive situations.

This research project proposes a new tracepoint insertion technique. The use of this technique

significantly increases the number of valid insertion points for a tracepoint with branch. When multiple instructions would be replaced by a branch, interrupt instructions are embedded in the offset of the branch at the beginning of every replaced instruction. This solves the problems caused by the use of a branch instruction when inserting a tracepoint. The use of interruptions ensures that no thread will execute an invalid instruction. If a thread executes an interruption, it is simply rerouted to an equivalent instruction created for the tracepoint. This technique enables the use of higher performance dynamic tracepoints in most cases.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Program Interface
BCC	BPF Compiler Collection
BPF	Berkeley Packet Filter
CISC	Complex Instruction Set Computer
CPU	Processeur Central, Central Processing Unit
CTF	Common Tracing Format
eBPF	extended BPF
GCC	GNU Compiler Collection
GDB	GNU DeBugger
GNU	GNU's Not Unix
IP	Pointeur d'Instruction, Instruction Pointer
KVM	Kernel Virtual Machine
LTTng	Linux Tracing Toolkit Next Generation
OS	Système d'Exploitation, Operating System
RCU	Read-Copy-Update

CHAPITRE 1 INTRODUCTION

Le développement d'applications et de systèmes complexes est une tâche longue et difficile. L'erreur humaine amène beaucoup de ralentissement à ce développement et la tâche de trouver et régler ces erreurs est longue et difficile. Plusieurs outils sont à la disposition du développeur afin d'aider à ce processus. Le débogage est un des outils disponible au développeur pour trouver et analyser les erreurs durant le développement de ces systèmes. Le débogage permet d'arrêter un processus à des points clés et d'analyser le comportement de celui-ci. L'arrêt d'un système n'est pas toujours utile pour cibler des problèmes particuliers, comme les problèmes de synchronisation de systèmes multi-fils et les problèmes de performance. En effet, ces problèmes ne se manifestent généralement pas si le système est dérangé. Ainsi, une analyse qui ne dérange pas le système sous test doit être faite afin de découvrir ce type de problème.

Les outils existants offrent des solutions qui permettent d'instrumenter dynamiquement un système. Ces solutions demandent par contre un conpromis entre performance et flexibilité. En effet, trois techniques sont généralement utilisées afin d'implémenter l'instrumentation dynamique. La première, l'utilisation d'une interruption, permet d'instrumenter la plupart du système, mais nécessite une perte de performance significative. La deuxième technique, l'utilisation d'un branchement, permet d'obtenir une bonne performance à l'exécution de l'instrumentation au détriment de la flexibilité d'insertion. La troisième technique, l'utilisation de branchement avec arrêt du système, permet de combiner la performance du branchement et la flexibilité de l'interruption, mais demande un arrêt complet du système lors de l'instrumentation.

Ce mémoire propose une solution utilisant des branchements relatifs avec une grande flexibilité d'insertion, et sans arrêt du système instrumenté. L'utilisation d'interruptions incorporées au branchement permet de placer un point d'instrumentation de façon atomique et "thread-safe". Ce mémoire présente un algorithme d'allocation et d'exéctution de point d'instrumentation. Ce mémoire montre ensuite le gain de performance par apport à l'utilisation d'interruptions et le gain de flexibilité par apport à l'utilisation d'un branchement relatif.

1.1 Concepts de base

1.1.1 Instructions en x86

x86 est une architecture CISC avec un encodage d'instruction à taille variable. Les instructions ont une longueur entre 1 et 15 octets (1) et n'ont aucune contrainte d'alignement. Deux catégories d'instructions sont importantes pour ce mémoire : les interruptions logicielles et les branchements. Une interruption logicielle est un appel à une fonction d'interruption enregistrée dans une table d'interruption. De plus, si le code exécutant l'interruption est en mode non privilégié, le processeur se met en mode privilégié. Ainsi, c'est au système d'exploitation de gérer les interruptions. Une interruption importante est l'interruption de breakpoint (*int $3* ou 0xCC) qui a une taille de 1 octet. Cette interruption est utilisée par les outils de débogage et de traçage pour interrompre le flot du programme. Un branchement relatif est une instruction de 5 octets qui modifie directement le pointeur d'instruction (ip) : $ip = ip + décalage + 5$. Le premier octet est l'opcode et les 4 autres représentent le décalage du branchement.

1.1.2 Traçage

Le traçage, ou instrumentation, est une catégorie de techniques pour surveiller et sauvegarder l'information de bas niveau sur un système. Le but du traçage est d'obtenir l'information sur le système dans le but de connaître l'état exact de celui-ci durant son exécution. Ainsi, on trace généralement les appels de fonctions, les paramètres, les valeurs des variables, les valeurs de retour et les messages entre les composantes du système. Puisque le traçage collecte une grande quantité d'information, le système est en général ralenti et donc les bibliothèques/outils de traçage doivent être très performants. De plus, les outils doivent prendre en compte le fait que les systèmes à tracer peuvent être multi-processeur ou multi-ordinateur.

Un point de trace est une procédure/fonction qui enregistre un évènement. Un évènement est un ensemble d'informations enregistrées à un moment donné. Une trace est une liste d'évènements qui peuvent être reliés entre eux. Ainsi, un système tracé produira un ensemble d'évènements produits par des points de trace stratégiquement placés. Cette trace peut ensuite être analysée par un utilisateur afin de connaître l'évolution du système durant la période de traçage. On distingue deux types de traçage : le traçage avec instrumentation statique et le traçage avec instrumentation dynamique. Pour utiliser un outil de traçage avec instrumentation statique, l'utilisateur doit placer les points de trace avant ou durant la compilation. Ainsi, les points de trace sont présents dans les exécutables du système. Pour ajouter des points de trace, l'utilisateur peut ajouter des appels dans le code, ou peut utili-

ser un compilateur pour générer automatiquement le code de traçage. Dans le premier cas, l'utilisation d'une bibliothèque est requise afin de définir et utiliser les points de trace. Pour ajouter, modifier ou enlever un point de trace, la recompilation du système est requise.

Contrairement aux points de trace statiques, les points de trace dynamiques ne sont pas présents dans l'exécutable. Ainsi, aucune recompilation n'est nécessaire pour modifier les points de trace. Par contre, une modification dynamique du programme, avant ou pendant l'exécution, est nécessaire. En effet, un outil de traçage avec instrumentation dynamique doit pouvoir s'insérer dans un processus durant l'exécution et modifier le code machine de celui-ci afin d'y ajouter les points de trace nécessaires. Normalement, l'outil de traçage dynamique remplace une ou plusieurs instructions par une autre instruction qui invoque le traçage. Deux instructions sont considérées dans ce travail pour invoquer le traçage en x86 : un branchement et une interruption. Une interruption est lente à exécuter, puisque l'instruction retourne en mode kernel, invoque la fonction d'interruption, envoie un signal au processus qui a lancé l'interruption, et retourne à l'exécution du processus. Le branchement est plus rapide puisqu'il n'y a aucun changement de mode usager et kernel, et l'instruction est tout simplement plus rapide à exécuter. Lorsque placée, l'interruption remplace une seule instruction, ce qui rend facile l'insertion de cette instruction. Le branchement peut potentiellement remplacer plusieurs instructions, ce qui peut causer des problèmes. Si un fil est sur le point d'exécuter une instruction remplacée par le branchement (sauf la première), le fil exécutera probablement des instructions invalides après que le branchement ait été placé. Un branchement qui atterrit à une instruction remplacée peut causer le même problème.

1.1.3 Tracefs

Tracefs est un pseudo-système de fichier dans le kernel Linux qui présente une interface aux fonctions de traçage du kernel. Les utilisateurs ont accès aux fonctionalités de traçage à travers les pseudo-fichiers créés automatiquement par le kernel. L'utilisateur peut modifier l'état du sous-système de traçage en écrivant des pseudo-fichiers, et obtenir de l'information sur l'état du sous-système en listant des pseudo-fichiers. Toutes les traces collectées par les différents outils de traçage sont accessibles en listant le pseudo-ficher *trace* présent à la racine du système de fichier. Ce système de fichier est normalement monté à */sys/kernel/debug/trace*.

1.1.4 RCU

RCU est un technique de synchronisation sans verrou qui permet la modification et lecture simultanée de structures de données complexes. Cette technique est appropriée lorsqu'un seul modificateur est présent. Avant de commencer une lecture de la structure de donnée,

un lecteur doit signaler sa présence (*rcu_read_lock()*) et doit signaler la fin de la lecture (*rcu_read_unlock()*). Pour modifier la structure de donnée, l'écrivain doit d'abord créer une copie de la version précédente. Ensuite, celui-ci modifie la nouvelle version des données. Finalement, l'écrivain remplace atomiquement la vielle version de la structure par la nouvelle version. Avant de désallouer la vielle version, l'écrivain doit attendre que tous les lecteurs utilisant celle-ci aient terminé avec un appel à rcu_synchronise. Ainsi, l'écrivain attend que tous les lecteurs ayant entré la zone de lecture avant l'appel à rcu_synchronize aient quitté la zone.

1.2 Plan du mémoire

La première section au Chapitre 2 présente les techniques existantes de traçage avec instrumentation statique, de traçage avec instrumentation dynamique et de modification dynamique de programmes. Le Chapitre 4 est un article présentant une solution aux problèmes que présentent les techniques de traçage existantes. Le Chapitre 3 discute des résultats de l'article ainsi que des autres solutions existantes. Finalement, le Chapitre 6 discutte des résultats obtenus et conclut le mémoire.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente la revue de l'état actuel des différents outils d'instrumentation et de mofication dynamique de processus. Les différents outils d'instrumentation statique sont d'abord présentés. Ensuite, les outils d'instrumentation dynamique et les outils de modification dynamique sont présentés. Les outils d'instrumentation dynamique sont séparés en quatres catégories : les outils de modification dynamique aidés par le compilateur ; les outils d'instrumentation assistés par le matériel ; les outils d'instrumentation dynamique logiciels ; et les outils de modification dynamique de processus existants. La dernière section présente un résumé des différents outils d'instrumentation présentés dans ce chapitre.

2.1 Instrumentation Statique

Points de trace du noyau Linux

Le noyau Linux implémente une bibliothèque de traçage qui peut être utilisée par les différents sous-systèmes et modules du système d'exploitation. Cette bibliothèque est implémentée dans les dossiers *include/trace/* et *kernel/trace/* du noyau (2). Pour créer un point de trace, l'usager doit créer une déclaration dans un .h et une définition dans un .c. Ce point de trace peut ensuite être utilisé dans le code source que l'usager veut tracer.

Les Figures 2.3 et 2.1 montrent la création d'un point de trace, en utilisant les fichiers hypothétiques *include/trace/example.h* et *trace/example.c*. Quand un point de trace est déclaré, l'usager définit les arguments de ce point de trace ainsi que les expressions pour sauvegarder ceux-ci. La définition du point de trace se fait simplement en utilisant une macro. La Figure 2.2 montre l'utilisation du point de trace qui a été créé. Les fonctions *trace_example_tp()* et *trace_example_tp_enabled()* sont générées par la bibliothèque lorsque le point de trace est déclaré. La première fonction permet de tracer l'évènement, si ce point de trace est actif, et la deuxième fonction permet de savoir si le point de trace est actif.

Pour contrôler les points de traces, des peusdo-dossiers sont créés dans le tracefs. Chaque sous-système (défini par la macro TRACE_SYSTEM) crée un dossier et chaque point de trace associé à ce sous-système crée un sous-dossier. Chaque dossier permet de contrôler un point de trace. Ainsi, l'utilisateur peut activer, désactiver et filtrer chaque point de trace en utilisant les pseudo-fichiers présents pour chacun. Lorsqu'un point de trace est exécuté, l'information est écrite dans un tampon circulaire temporaire. L'information peut ensuite être retrouvée en lisant le pseudo-ficher *trace*, ou en lisant les traces par CPU dans les fichiers

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM example

#if !defined(_TRACE_EXAMPLE_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_EXAMPLE_H
#include <linux/tracepoint.h>

DECLARE_TRACE(example_tp,
    TP_PROTO(int a, struct mm_struct* m),
    TP_ARGS(a*2, m));

#endif

#include <trace/events/define_trace.h>
```

Figure 2.1 Définition d'un point de trace Linux (trace/example.c)

```
#include <trace/example.h>

#define CREATE_TRACE_POINTS
DEFINE_TRACE(example_tp);
```

Figure 2.2 Utilisation d'un point de trace Linux (example.c)

```
#include <trace/example.h>

void example(int a)
{
    if(trace_example_tp_enabled())
    {
        trace_example_tp(a, current->mm);
    }
}
```

Figure 2.3 Déclaration d'un point de trace Linux (trace/example.h)

per_cpu/cpuXX/trace.

Il est aussi possible d'enregistrer un appel à exécuter lors du point de trace à partir d'un module ou du noyau. La fonction *register_trace_example_tp()* enregistre la fonction qui sera appelée lorsque le point de trace *example_tp()* sera exécuté. Chaque point de trace créé génère une fonction d'enregistrement. La fonction *unregister_trace_example_tp()* enlève la fonction de rappel pour le point de trace. Le module doit appeler la fonction *tracepoint_synchronize_unregister()* afin d'attendre que tous les points de trace enlevés aient terminé leur exécution. Mohamad Gebai (3) a écrit un fichier de documentation qui explique plus en détails comment utiliser les points de trace Linux. Il est ainsi possible d'utiliser les points de trace Linux à partir de plusieurs traceurs. Par exemple, les outils Ftrace, LTTng et Perf permettent l'utilisation de ces points de trace.

OpenTracing

OpenTracing API (4), défini pour plusieurs langages de programmation, permet de tracer des systèmes distribués. Pour l'utiliser, le programmeur doit commencer et terminer des "spans". Un "span" est défini comme un travail qui a un début, une fin et des attributs. Un "span" peut avoir un parent, un autre "span" qui a initié le travail de ce span, et plusieurs enfants, des "spans" qui ont été initiés par le span courant. Un "span" peut représenter une procédure qui est appelée par une procedure ("span" parent) et invoque des sous-procédures ("spans" enfants). L'interface d'OpenTracing est de haut niveau et ne se préoccupe pas de l'implémentation du traçage en tant que tel. Pour utiliser cette bibliothèque, un utilisateur doit fournir un traceur et un transport. Le traceur est appelé par la bibliothèque pour tracer les évènements générés par l'utilisateur et le transport est le processus qui permet d'envoyer l'information de trace entre les différents acteurs du système.

Plusieurs outils accomplissant cette tâche existaient avant OpenTracing. Ces outils offrent un système de traçage distribué dans plusieurs langages de programmation. Dapper (5) est un outil développé par Google qui permet de tracer des systèmes distribués qui utilisent les bibliothèques de communication créées par Google. Sans modifier le code source, l'outil permet de tracer les différents noeuds du système et d'envoyer l'information de traçage entre les différentes composantes du système. Les bibliothèques de communication et de répartition de tâches sont annotées afin d'ajouter l'instrumentation nécessaire à Dapper. Lorsqu'un message est envoyé à un autre noeud, Dapper ajoute au message l'information nécessaire pour le traçage des autres noeuds. Dapper permet aussi de rassembler les traces produites par les différents acteurs du système. Tout comme OpenTracing, Dapper représente les différentes tâches comme des "spans", qui ont un parent et plusieurs enfants. Un arbre d'exécution peut

ainsi être créé à partir de la collection de "span". L'utilisation de Dapper nécessite l'utilisation des bibliothèques Google afin de récupérer l'information. Dapper supporte aussi l'enregistrement d'évènements créés par l'utilisateur dans le code d'application. Puisque le traçage de tous les évènement diminue les performances du système, l'outil supporte aussi l'échantillonage des données de trace. Ceci diminue l'impact de l'outil sur la performance du système tracé.

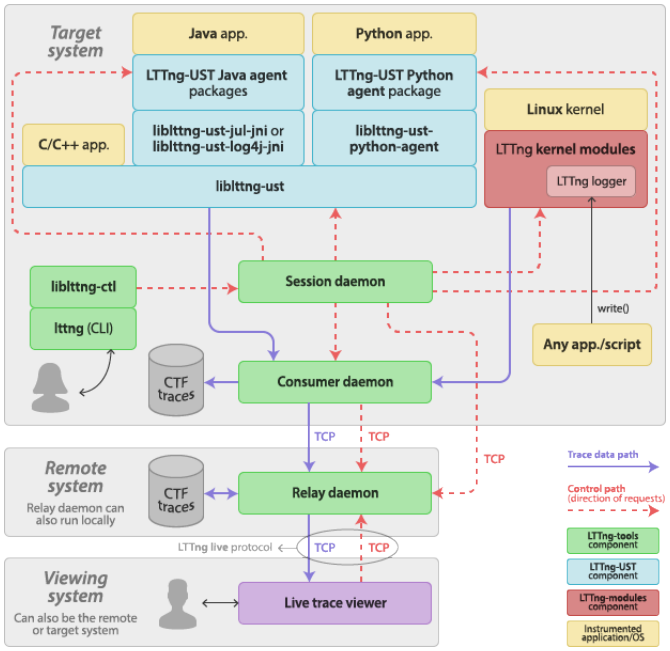
Zipkin (6) est un outil basé sur l'article de Dapper créé par Twitter. Cet outil implémente l'architecture décrite dans l'article, et offre une interface pour l'utilisation de l'outil dans les applications usager. Ainsi, il est possible d'ajouter les appels tracer les "spans" et transmettre l'information entre les noeuds du système. Plusieurs implémentations de l'outil sont disponibles dans plusieurs langages de programmation. De plus, plusieurs bibliothèques de communication ont été instrumentées avec Zipkin.

L'utilisation de bibliothèques de traçage pose un problème pour les développeurs d'applications : l'ajout d'une dépendance entre l'application et le système de traçage. En effet, ces bibliothèques présentent des interfaces incompatibles et spécifiques à chacuns des outils. Pour pallier à ce problème, OpenTracing propose une interface indépendante au traceur pour utilisation par les applications et bibliothèques. Ainsi, les développeurs de bibliothèques peuvent ajouter des appels à OpenTracing qui seront implémentés par l'outil au choix de l'utilisateur de cette bibliothèque. Ainsi, un découplage entre l'application et le traceur est possible avec l'utilisation d'OpenTracing.

LTTng

LTTng (7) est un écosystème de traçage du noyau Linux et des applications usager. Cette suite d'exécutables permet de tracer les processus et le noyau, de contrôler les opérations de traçage, de créer des points de trace personalisés, de centraliser l'information de traçage, et de rediriger l'information de traçage vers plusieurs sorties. La Figure 2.4 montre les différents composants de l'écosystème LTTng.

La première composante importante est la bibliothèque lttng-ust. Cette bibliothèque permet de créer des points de trace statiques personalisés dans un programme usager écrit en C ou C++. L'utilisateur doit d'abord créer une définition de point de trace. Celle-ci permet de définir la catégorie, le nom et les arguments du point de trace. Les points de trace définis peuvent ainsi être appelés dans le code utilisateur afin d'enregistrer les évènements. La syntaxe de définition de point de trace est similaire à celle des points de trace Linux (Figure 2.3 et Figure 2.1). Pour utiliser un point de trace, la macro *tracepoint()* est exécutée. Les arguments définis dans le point de trace doivent être passés en argument à la macro *tracepoint()*.



tiré de `lttng.org/docs/v2.10/` le 2018-09-05

Figure 2.4 Composantes de LTTng

Lorsque la macro est exécutée et que le point de trace est actif, l'évènement et ses arguments sont sauvegardés.

Lttng-ust définit aussi plusieurs points de trace prédéfinis. Les bibliothèques lttng-ust-libc-wrapper, lttng-ust-pthread-wrapper et lttng-ust-dl ajoutent des points de traces aux fonctions de libc, de pthread et de libdl respectivement. En ajoutant ces bibliothèques à la variable d'environnement *LD_PRELOAD*, l'utilisateur peut tracer les fonctions standards utilisateurs du système Linux. Les bibliothèque lttng-ust-cyg-profile et lttng-ust-cyg-profile-fast permettent de tracer l'entrée et la sortie de chaque fonction d'un programme. Ces deux fonctions ont accès à l'adresse de la fonction instrumentée ainsi que l'adresse d'appel de celle-ci Avec GCC, l'utilisation de l'option *-finstrument-functions* lors de la compilation ajoute l'appel *cyg_profile_func_enter()* aux entrées et l'appel *cyg_profile_func_exit()* aux sorties de chaque fonction d'un programme. Les bibliothèques effectue des appels à des points de trace dans l'implémentation de ces fonctions. La bibliothèque lttng-ust-cyg-profile-fast est une variante plus rapide et moins robuste de lttng-ust-cyg-profile. Cette première ne permet pas de reconstruire le flot du programme s'il y a perte d'information, contrairement à cette deuxième. Cela est dû au fait que lttng-ust-cyg-profile sauve plus d'information lors de l'exécution du point de trace.

Le projet Lttng-ust offre aussi des bibliothèques de traçage pour les langages Java et Python.

Pour ces deux langages, LTTng implémente des loggeurs qui peuvent être utilisés pour enregistrer l'information avec l'écosystème LTTng. Pour Java, deux bibliothèques de log sont supportées : le loggeur standard Java et Apache log4j. Pour Python, seul le logueur standard est supporté par LTTng.

La deuxième partie importante de LTTng est le traceur du noyau Linux. Celui-ci permet d'instrumenter tous les points de trace définis dans le noyau Linux (voir Sous-Section "Points de trace du noyau Linux"). Le projet lttng-modules définit des points de trace LTTng qui implémentent les fonctions de rappel des points de trace noyau. Ainsi, pour ajouter un point de trace LTTng dans le noyau, un point de trace noyau équivalent doit être ajouté. Les différents modules implémentent les points de trace LTTng pour les différents sous-systèmes du noyau et ajoutent l'instrumentation aux points de trace noyau lorsqu'ils sont chargés. Ces points de trace sauvegardent l'information en utilisant les mécaniques de l'écosystème LTTng. Le projet offre aussi le pseudo-fichier de traçage */proc/lttng-logger*. Ce fichier peut être ouvert par n'importe quel usager et permet à tous les processus de tracer en utilisant l'écosystème LTTng. Le programme n'a qu'à écrire une chaîne de caractères dans ce fichier et celle-ci sera enregistrée comme un évènement LTTng.

L'écosystème propose plusieurs abstractions afin de contrôler l'exécution des points de trace du système. Une session (*session*) est le point d'entrée pour le contrôle du traçage. Une session possède son propre nom, fichiers, état, mode et canaux. Une session est toujours associée à un utilisateur. Le mode d'une session détermine où l'information est sauvegardée : localement, par réseau, sur demande ou en temps réel. Une sauvegarde locale écrit dans des fichiers du système les évènements tracés sous le format CTF. Une sauvegarde réseau envoie par TCP l'information de traçage à un autre système. Le mode sur demande ne sauvegarde aucune information sauf si un utilisateur le demande. Ainsi, la vielle information non sauvegardée est perdue si elle n'est pas sauvée. Le mode en temps réel permet à un outil de visualisation de trace d'obtenir les informations de trace immédiatement. Chaque canal contrôle la sauvegarde de l'information dans un ou plusieurs tampons circulaires. Un canal est associé à un domaine de traçage (*namespace*). Chaque source de donnée est abstraite par un domaine : programmes utilisateur, noyau, Python, Java logger, et Apache log4j. Chaque canal possède aussi ses propres conditions de traçage. Chaque canal crée au moins un tampon circulaire par CPU. Les canaux possèdent aussi un ensemble de conditions à remplir pour effectuer le traçage. Ainsi, lorsqu'un point de trace est exécuté, pour chaque canal actif, les conditions sont vérifiées, et l'information est sauvegardée dans un ou plusieurs tampons de ce canal. Pour le domaine usager, un canal peut aussi être configuré pour créer un tampon par processus par processeur. Chaque tampon possède plusieurs sous-tampons, qui sont les espaces mémoire utilisés pour sauvegarder l'information. Lorsqu'un sous-tampon est plein, le

prochain sous-tampon est utilisé. Si tous les sous-tampons sont pleins, deux options peuvent être choisies par l'utilisateur : effacer le sous-tampon le moins récent, ou ignorer les nouveaux évènements jusqu'à la libération d'un sous-tampon.

Tous les tampons sont lus par un processus consommateur de trace appelé *Consumer Daemon* (*lttng-consumerd*). Ce processus s'exécute en arrière plan et est initié automatiquement par l'écosystème lors de la création d'une session. Ce processus sauvegarde localement ou par réseau, si besoin, selon le mode de la session. L'utilisation de RCU permet de synchroniser sans verrou les opérations des points de trace et du processus consommateur. La bibliothèque *userspace rcu* (8) est utilisée pour implémenter les RCU pour les composantes usager. Le noyau Linux implémente aussi les RCU.

Afin de contrôler le processus de traçage, le processus de session (*lttng-sessiond*) est utilisé. Ce processus exécute en arrière plan et communique avec les différentes composantes de l'écosystème afin de contrôler les opérations. Ce processus peut activer et désactiver les points de trace, créer des canaux, créer des conditions et contrôler les autres processus de l'écosystème. Afin d'envoyer des commandes au processus de session, l'utilisateur peut utiliser la bibliothèque *lttng-cli*. Cette bibliothèque implémente le protocole de communication avec le processus de session et le rend utilisable pour les programmes C et C++. Une interface en ligne de commande (*lttng*) est aussi disponible pour contrôler ce processus.

Le processus de relais (*lttng-relayd*) peut recevoir les évènements envoyés par le processus consommateur en mode réseau. Celui-ci sauvegarde l'information reçue sur la machine locale sous le format CTF. Ce processus permet aussi aux outils de visalisation de recevoir l'information du consommateur en temps réel. Celui-ci envoie l'information aux outils en utilisant le *lttng live protocol*. *LTTngtop* est un exemple d'outil de visualisation en temps réel pour LTTng. Cet outil affiche des statistiques sur les évènements de la session courante. Cet outil peut aussi être utilisé pour montrer l'information d'une trace CTF.

CTF est un standard d'enregistrement d'information de trace qui peut être lu par plusieurs outils. L'utilisation d'un format standard permet de facilement utiliser l'information de traçage dans plusieurs outils existants et futurs. Babeltrace (9) est un outil qui permet de montrer l'information d'une trace CTF sous un format lisible par l'humain. Babeltrace offre aussi une bibliothèque Python qui permet d'analyser programmatiquement l'information d'une trace. TraceCompass (10) est un outil créé par-dessus Eclipse qui permet de voir et d'analyser les traces sous plusieurs formats, y compris CTF. Cet outil offre plusieurs vues et outils d'analyse afin d'extraire l'information utile d'une trace et du système tracé.

Gprof

Peter B. Kessler et al. (11) ont créé un outil de traçage inclus avec GCC appelé gprof. Pour utiliser cet outil, l'utilisateur doit compiler les fichiers sources avec l'option *-pg*[1], ce qui ajoute un appel à la fonction *mcount()* à chaque début de fonction du programme. Cette fonction est implémentée par une bibliothèque que l'utilisateur peut lier et trace chaque appel de fonction.

La bibliothèque standard de GNU (glibc) implémente la fonction de traçage *mcount()* (12). Lors de l'appel de la fonction, celle-ci retrouve l'adresse de la fonction courante ainsi que l'adresse de la fonction appelante. Ces adresses sont ajoutées à un graphe d'appel présent en mémoire à chaque appel de *mcount()*. La bibliothèque enregistre aussi un rappel de minuterie qui échantillone la valeur du pointeur d'instruction et la valeur de l'horloge. Lors de la sortie normale du programme, la bibliothèque sauvegarde l'information du graphe dans un fichier de type *gmon*. Les informations d'échantillonage et les informations sur l'exécution sont aussi enregistrées dans le fichier de sortie. Ainsi, si le programme termine anormalement, aucune information n'est enregistrée.

L'exécutable *gprof* permet de lire le fichier *gmon* produit par l'exécution du programme instrumenté. Cet outil prend en paramètre l'exécutable source ainsi que le fichier *gmon*. Cet outil lit le fichier d'instrumentation et lie les informations avec l'exécutable afin de produire un rapport d'exécution. On peut tout d'abord y voir les informations d'échantillonage des fonctions. Cette section montre la liste des fonctions, le nombre d'appels et le pourcentage de temps passé dans ces fonctions selon les échantillons de pointeur d'instruction. Ensuite, le graphe d'appels est montré pour chaque fonction. Ce graphe montre les fonctions appelées ainsi que le nombre d'appel par chacune des fonctions non-feuilles de l'exécutable.

Gcov

Tout comme gprof, gcov (13) est un outil inclus avec GCC qui demande une compilation spéciale. Un utilisateur peut instrumenter tous les branchements d'un programme en ajoutant les options *-fprofile-arcs* et *-ftest-coverage* lors de la compilation. Ces options ajoutent des appels à des fonctions d'instrumentation à tous les branchements présents dans le programme. Les fonctions enregistrent le nombre de fois que les branchements ont été pris durant l'exécution du programme. Lors de la sortie du programme, l'information est sauvegardée dans plusieurs fichiers de couverture. Pour chaque fichier source du programme, un fichier de couverture associé est généré. L'exécutable *gcov* permet ensuite de lire l'information des fichiers de cou-

1. `gcc -c -pg <source> -o <obj>` et `gcc -pg <objs> -o <exe>`

verture. L'outil prend en entrée un fichier source et imprime le nombre de fois que chacune des lignes de code a été exécutée.

2.2 Instrumentation Dynamique

2.2.1 Assisté par le Compilateur

XRay

XRay est un outil de traçage dynamique développé par Dean Berris et al. (14) qui nécessite l'aide du compilateur. Pour utiliser cet outil, l'utilisateur doit tout d'abord compiler le programme en utilisant l'option *-fxray-instrument* avec clang. Cette option génère un bloc de 11 octets au début et un autre de 10 octets à la fin de chaque fonction à instrumenter qui ne font rien (no-op). Le compilateur assigne aussi un identifiant de 4 octets à chaque fonction instrumentée. De plus, le compilateur sauvegarde la position et l'identifiant de chaque bloc dans un fichier (possiblement dans l'exécutable lui-même). Ainsi, si un utilisateur exécute le programme, aucun traçage ne sera effectué et l'impact du code ajouté sera négligeable sur le temps d'exécution. Pour activer le traçage, l'utilisateur doit commencer le programme en chargeant la bibliothèque XRay. Lors de l'initialisation, la bibliothèque lit la liste de points de trace xray et va remplacer les blocs générés par deux instructions [2] qui invoquent la bibliothèque. Pour la sortie, le *ret* est aussi remplacé pour un total de 11 octets. Les appels de fonction générés sont implémentés par la bibliothèque afin de sauvegarder les évènements.

Ftrace

Ftrace est un ensemble de traceurs dynamiques présents dans le noyau Linux. Cet outil présente une interface commune à tous les traceurs présents dans le noyau à partir du pseudo système de fichier Tracefs. Les deux premiers traceurs : le traceur de fonction (*function*) et le traceur de graphe (*function_graph*), fonctionnent de la même façon. Lors de la compilation du noyau, l'option *-pg* est utilisée afin d'ajouter des appels à la fonction *mcount()* au début de chaque fonction du noyau. Lorsque le traceur de fonction est utilisé, la fonction *mcount()* enregistre le nombre d'appels de toutes les fonctions du noyau. Lorsque le traceur de graphe est utilisé, la fonction *mcount()* enregistre la fonction courante ainsi que la fonction appelante. Ceci permet de construire un graphe d'appel des fonctions du noyau. Le traceur enregistre aussi le retour de la fonction en remplaçant l'adresse de retour de la fonction par une fonction d'enregistrement. Les deux traceurs enregistrent aussi le temps de l'entrée et

2. *mov $id, %r10d ; call ___xray_FunctionEntryStub* pour l'entrée et *mov $id, %r10d ; call ___xray_FunctionExitStub* pour la sortie

du retour des fonctions tracées. Lors de la compilation du noyau, il est possible d'utiliser l'option de configuration *CONFIG_DYNAMIC_FTRACE* pour créer des points de trace dynamiques au début de chaque fonction. Ceci utilise les options *-pg -mfentry -mnop-mcount* lors de l'invocation de gcc, afin de placer une instruction nop de 5 octets. Par défaut, lorsque le traçage dynamique est utilisé, aucune fonction n'est tracée. Les fonctions non-activés n'ont aucune perte de performance, puisque l'instruction no-op est très rapide à exécuter. Lorsqu'un utilisateur écrit une fonction dans le fichier *set_ftrace_filter* (traceur de fonction) ou *set_graph_function* (traceur de graphe), celle-ci est activée. Le no-op de 5 octets au début de la fonction est remplacé par un appel à la fonction *mcount()*.

Plusieurs traceurs offerts par Ftrace se basent sur des évènements spécifiques enregistrés dans le noyau. Ces traceurs activent des points de trace spécifiques dans le noyau afin de tracer un sous-système spécifique. Le traceur de bloc (*blk*) instrumente le sous-système de disques et de pseudo-disques afin de suivre les opérations. Les requêtes de disque et l'ordonnancement de ces requêtes sont tracés. Le traceur d'interruption permet de tracer l'activation et la désactivation des interruptions du système, ainsi que leur point d'activation/désactivation. Le traceur de préemtion (*preemptoff*) permet de tracer les moments pendant lesquels la préemption est désactivée. Plusieurs autres traceurs sont offerts, selon la configuration et l'architecture du système. Les traceurs disponibles sont listés dans le fichier *available_tracers*.

En plus des traceurs, ftrace supporte l'instrumentation des points de trace statiques du noyau. En écrivant dans le fichier *set_events*, l'utilisateur peut activer le traçage d'évènements spécifiques. Les évènements disponibles sont listés dans le fichier *available_events*.

Lorsqu'un évènement est tracé, l'information de celui-ci est écrite dans un tampon circulaire par processeur. La taille de ce tampon peut être modifiée en écrivant dans le fichier *buffer_size_kb*. Si le tampon est plein, la nouvelle information est enregistrée par-dessus la plus vielle information, perdant une partie de la trace. Pour lire l'information de la trace, l'utilisateur peut lire le fichier *trace* qui produit la trace sous un format lisible. La trace lisible par processeur est accessible en lisant les fichiers *per_cpu/cpuXX/trace* où *XX* est le numéro du processeur. La trace brute par cpu est accessible dans les fichiers *per_cpu/cpuXX/trace_pipe_raw*.

Uftrace

Uftrace est un outil développé par K. Namhuyng (15) qui permet de tracer les appels de fonction dans un programme usager. Pour utiliser cet outil, l'utilisateur peut utiliser soit la version statique, soit la version dynamique. Pour utiliser la version statique, le programme à tracer doit être compilé avec l'option *-pg*, et avec les options *-pg*, *-mfentry* et *-mnop-mcount*

pour la version dynamique. Pour tracer un programme avec uftrace, l'utilisateur doit exécuter le programme avec l'interface uftrace. Il n'est pas possible de tracer un programme déjà en exécution avec uftrace. Pour la version statique, toutes les fonctions qui commencent avec un appel à mcount seront tracées. Pour la version dynamique, une liste de fonctions à tracer est entrée en ligne de commande par l'utilisateur. Quand uftrace lance le programme à tracer, la bibliothèque uftrace sera chargée en utilisant la variable d'environnement *LD_PRELOAD*. Cette bibliothèque implémente la fonction *mcount()* et remplace la version par défaut présente dans glibc (qui produit un fichier pouvant être lu par gprof). Dans la version dynamique, uftrace remplace les *nop* présents au début des fonctions listées par l'utilisateur en ligne de commande par des appels à la fonction *mcount()*.

Lorsque la bibliothèque est chargée par uftrace, des arguments sont passés grâce à des variables d'environnement. Ensuite, le fichier d'enregistrement de trace est créé. Ce fichier peut être un fichier ordinaire ou un tuyau si le mode instantané est utilisé. La bibliothèque enregistre l'adresse de la fonction courante ainsi que l'adresse de la fonction appelante lors de l'appel de *mcount()*. La bibliothèque enregistre aussi le temps d'exécution des fonctions, du départ au retour. Ceci permet de reconstruire le graphe des appels ainsi que les temps d'appel des fonctions du programme. Cette information est enregistrée dans le fichier de sortie sous un format spécifique à Uftrace. L'outil permet ensuite de lire le fichier de données et rapporter les informations importantes.

Pour tracer un programme avec Uftrace, l'utilisateur doit exécuter le programme à travers l'outil. Puisque Uftrace doit charger une bibliothèque dans le processus instrumenté, il est plus simple d'utiliser la variable d'environnement *LD_PRELOAD*. Il est possible d'injecter une bibliothèque dans un processus en exécution mais c'est un processus lent et instable. La section sur GDB et GDB Server explique le processus pour injecter une bibliothèque.

2.2.2 Assisté par le Matériel

Intel Processor Trace (PT)

La technologie Intel Processor Trace (PT) (1) est un ensemble d'instructions et de registres présents dans les processeurs Intel modernes qui implémentent un processus de traçage dynamique. Lorsqu'actif, Intel PT trace les évènements du processeur sous forme de paquet. Chaque paquet représente un type d'évènement et contient toutes les informations sur l'évènements spécifique. Les types d'évènements sont : les branchements, les interruptions, les exceptions, l'instruction *PTWRITE*, les évènements de gestion de l'alimentation, et les évènements de virtualisation. Pour sauvegarder les paquets, le processeur peut utiliser un tampon

de mémoire, plusieurs tampons de mémoire, ou un sous-système de transport spécifique à la plateforme. Le processeur peut aussi filtrer l'activation du traçage selon le niveau de privilège courant, la valeur de CR3 (tables de mémoire virtuelle), ou la valeur du pointeur d'instruction (RIP). Le traçage peut seulement être activé par du code en mode noyau, et peut être utilisé pour tracer les évènements noyau et usager.

ARM CoreSight

CoreSight (16) est une technologie présente dans certains processeurs ARM qui permet de déboguer et tracer le code exécutant sur le processeur. La technologie présente deux interfaces : l'interface interne et le port de débogage. L'interface interne est un ensemble de registres accessibles par le processeur contrôlant l'exécution de CoreSight, et le port de débogage permet à un processeur externe de contrôler l'exécution de cette technologie. Le standard CoreSight ne définit pas d'implémentation de traçage, mais définit plutôt une interface facilitant le développement d'interfaces de débogage et de traçage. C'est aux implémentations de processeurs ARM d'implémenter les différentes composantes de débogage et de traçage, en utilisant l'interface définie par CoreSight. L'interface du port de débogage définit aussi une façon de découvrir les différents débogueurs et traceurs présents sur le processeur.

2.2.3 Instrumentation logicielle

Kprobe, Jprobe et Kretprobe

Kprobe (17) est une bibliothèque de traçage présente dans le noyau Linux qui permet à un utilisateur d'instrumenter n'importe quelle instruction du noyau. L'instruction instrumentée sera remplacée par une instruction d'interruption de 1 octet. L'utilisateur peut définir une fonction pré-instruction, une fonction post-instruction, une fonction de faute, et une fonction d'interruption. Lorsque l'interruption est exécutée, la fonction pré-instruction est tout d'abord appelée, l'instruction remplacée est ensuite exécutée hors-ligne, la fonction post-instruction est appelée, et finalement, le kprobe retourne à la fonction originale. Si l'instruction remplacée crée une erreur, la fonction de faute est appelée. Si une des fonctions définie par l'utilisateur interrompt, la fonction d'interruption est appelée. À partir des fonctions de rappel de kprobe, l'utilisateur a accès à l'instance du kprobe ainsi qu'aux registres du processeur. Ainsi, les arguments, variables locales et valeurs de retour sont accessibles.

Masami Hiramatsu et al. (18) proposent une optimisation des kprobes appelée DJprobe (Direct Jump probe). Cette modification permet aux kprobes d'utiliser une instruction de branchement au lieu d'une instruction d'interruption afin d'exécuter le kprobe. Cette opti-

misation apporte un gain de performance à l'exécution du kprobe puisqu'une instruction de branchement est beaucoup plus rapide qu'une instruction d'interruption. Puisqu'une instruction de branchement relatif est de 5 octets sur x86, plusieurs instructions pourraient être remplacées par le brachement. Ainsi, avant de placer un djprobe, une analyse de la fonction à instrumenter est utilisée pour déterminer si une instruction sauterait au milieu du branchement ajouté, ou si certaines instructions non-remplaçables seraient remplacées. Si l'analyse montre que l'insertion de l'instrumentation est légale, le système doit vérifier qu'aucun processus n'exécute l'une des instructions à remplacer. Si toutes ces conditions sont respectées, le djprobe est inséré.

Deux bibliothèques noyau d'instrumentation ont été créées en utilisant kprobe comme base : jprobe et kretprobe. Jprobe ((19)) est une bibliothèque permettant de tracer les appels de fonction du noyau Linux. En créant un jprobe, l'utilisateur définit une fonction de rappel avec la même signature que la fonction à instrumenter. Cette fonction de rappel sera appelée avant l'exécution de la fonction tracée, avec tous les paramètres passés à la fonction tracée. La bibliothèque kretprobe ((17)), aussi construite par-dessus kprobe, permet de tracer l'entrée et la sortie des fonctions du noyau. Lors de l'entrée de la fonction tracée, une fonction utilisateur d'entrée est appelée et l'adresse de retour de la fonction est remplacée par le rappel kretprobe. Lors de la sortie de la fonction, la fonction de rappel kretprobe appelle une fonction utilisateur de sortie, et la fonction retourne à l'adresse de retour originale. Les deux fonctions utilisateur ont accès aux registres, et indirectement aux arguments et valeurs de retour de la fonction tracée.

Un traceur pour le noyau Linux utilisant kprobe à été implémenté dans le noyau Linux (20). Ce traceur est controllé par des pseudo-fichiers dans le tracefs. Un utilisateur écrit la liste des points de trace à créer dans le fichier *kprobes_event*. À partir de ce fichier, des kprobes peuvent être créés pour n'importe quelle instruction du noyau et des kretprobes pour toutes les fonctions présentes dans le noyau ou les modules. Pour activer les points de trace, le fichier *events/kprobes/enable* doit être utilisé. Les évènements kprobes sont enregistrés dans le tampon circulaire de traçage et peuvent être lus en lisant le fichier *trace*.

Uprobe

Le traceur uprobe (21) est un traceur implémenté dans le noyau Linux qui permet de tracer les processus usagers. L'interface est très semblable à celle du traceur kprobe, où le fichier *uprobes_events* du tracefs permet de créer les points de trace et le fichier *events/uprobes/enable* est utilisé pour activer les points de trace. Pour créer un point de trace, l'utilisateur doit donner le chemin d'un exécutable et une adresse dans l'exécutable. Placer un point de trace dans

une bibliothèque partagée est aussi possible avec uprobe. Contrairement à la plupart des traceurs, tous les processus exécutant l'exécutable seront tracés. Pour implémenter le point de trace, une instruction d'interruption sera placée à l'adresse du point de trace. Lorsque le point de trace est exécuté, l'évènement est enregistré dans le tampon de traçage du noyau et est accessible en listant le fichier *trace* du tracefs. Il est aussi possible de créer des points de trace qui enregistrent l'entrée et la sortie des fonctions usager.

GDB et GDB Server

Le GNU Debugger (GDB) (22) est un outil de débogage et de traçage multi-architecture et multi-plateforme. Cet outil est séparé en deux composantes : le serveur GDB (gdbserver) et le client GDB (gdb). Pour instrumenter un programme, le serveur peut soit s'attacher à un programme existant, ou commencer un programme dans le but de l'instrumenter. Le but principal du serveur est de recevoir des requêtes et d'y répondre. Le serveur expose donc une connexion à laquelle un client peut se connecter, où plusieurs protocoles de communication sont disponibles. Le serveur peut contrôler toute l'instrumentation du processus sous test ainsi qu'envoyer au client toute l'information d'instrumentation. Le client se connecte au serveur via la connexion et permet d'envoyer des commandes au serveur. Le client interprète aussi toutes les informations reçues par le serveur dans le contexte du processus instrumenté. Le client possède aussi toutes les fonctionalités du serveur et il est donc possible d'utiliser GDB sans serveur actif.

Dans le contexte du traçage, GDB supporte les points de trace dynamiques avec une interruption et avec un branchement. Il n'y a aucune restriction sur le placement de points de trace avec interruption sur x86, puisque l'instruction d'interruption a une longueur de 1 octet. Les points de trace dynamiques avec branchement ne peuvent être placés que sur les instructions de 5 octets et plus. Lorsqu'activés, les points de trace peuvent imprimer l'information dans la console GDB ou sauvegarder l'information dans un fichier. Tous les points de trace peuvent être activés conditionellement, où la condition est spécifiée lors de la création du point de trace.

Pour ajouter un point de trace au processus intrumenté, GDB doit modifier la mémoire du processus tracé. L'appel système *ptrace()* permet de modifier la mémoire d'un autre processus. Ainsi, lorsque GDB doit ajouter un point de trace, l'instruction visée est remplacée par une interruption. GDB peut ensuite capturer les interruptions qui sont envoyées au processus lorsque les points de trace sont exécutés. L'utilisation de *ptrace()* empêche l'utilisation des points de trace avec saut. De plus, durant l'écriture de l'interruption, le processus tracé est arrêté. GDB peut aussi utiliser un agent à l'intérieur du processus. Cet agent est chargé durant

le départ du processus tracé. Cette bibliothèque offre la même interface que le serveur GDB. Puisque l'agent réside dans l'espace mémoire du processus tracé, l'utilisation d'opérations atomiques pour l'insertion des points de trace est possible. L'insertion des points de trace avec sauts est aussi possible avec l'utilisation de l'agent. Ainsi, lorsque la bibliothèque agent est utilisée, un canal de communication est créé entre GDB et le processus tracé. Ce canal est utilisé pour envoyer les commandes de l'agent et recevoir le status du processus instrumenté.

GDB supporte aussi les points de trace conditionels. Lors de la création d'un point de trace, l'utilisateur peut définir des conditions d'activation. Si la condition est respectée, le point de trace est activé et l'évènement est enregistré. Les conditions ont accès aux variables et registres du processus. L'utilisateur doit définir la condition dans un langage propre à GDB. Si l'appel système *ptrace()* est utilisé, les conditions sont compilées en bytecode spécifique à GDB. Lorsque le point de trace est exécuté, le bytecode est exécuté et le résultat vérifié avant d'enregistrer l'information. Si la bibliothèque agent est utilisée, la condition est plutôt compilée directement en langage machine. Ce bout de code généré est ajouté à l'exécution du point de trace. Cette compilation permet une exécution rapide du point de trace, particulièrement si un point de trace rapide est utilisé.

La bibliothèque agent peut aussi être injectée dans un processus existant. Le processus d'injection utilise l'appel système *ptrace* afin de prendre le contrôle du processus et modifier la mémoire de celui-ci. Grâce à *ptrace()*, GDB arrête tous les fils du processus. Un de ces fils est utilisé pour charger la bibliothèque. GDB doit tout d'abord écrire dans la mémoire exécutable le code nécessaire pour appeler une fonction à partir d'un registre. Cette mémoire doit être déjà allouée par le processus puisque *ptrace()* ne permet pas d'allouer directement dans la mémoire du processus. Ainsi, GDB doit remplacer une partie de la mémoire exécutable existante par du code propre à GDB. En modifiant la valeur des registres, GDB peut ensuite exécuter des fonctions et des appels systèmes à partir de la mémoire du processus cible. Ainsi, GDB peut exécuter les fonctions de charge de bibliothèque afin de charger la bibliothèque agent. Une fois l'agent chargé, GDB remet le code modifié original et se détache du processus. Avec l'agent chargé dans le processus cible, GDB peut envoyer des commandes à la bibliothèque agent.

H. Zhu (23) a ajouté au noyau Linux un serveur GDB permettant le débogage et le traçage du système d'exploitation. Afin d'instrumenter le noyau, ce serveur utilise la bibliothèque noyau kprobe. Ainsi, les points de trace dynamiques sont sujets aux même restrictions que l'instrumentation dynamique de kprobe.

Dtrace

Dtrace (24) est un outil de traçage dynamique pour le noyau Linux et pour les processus usagers. Pour créer un point de trace, l'utilisateur doit créer un fichier dans le language propre à Dtrace : "D" [3]. Ce fichier définit les points de trace à créer, ainsi que les actions à prendre lorsque le point de trace est exécuté. Le langage "D" a une syntaxe ressemblant au ANSI C qui permet de filtrer les appels, de modifier l'information des points de trace et d'enregistrer l'information pertinente à l'utilisateur. L'outil permet aussi de tracer la sortie des fonctions, appels systèmes et fonctions usagers.

SystemTap

Vara Prasad et al. (25) (26) ont créé un outil de traçage dynamique construit par-dessus kprobe et uprobe appelé SystemTap. Cet outil permet de tracer les fonctions du noyau Linux, les appels systèmes, ainsi que les fonctions des programmes usagers. Pour créer un point de trace, l'usager doit écrire un fichier de point de trace qui définit les fonctions à tracer ainsi que les actions à prendre lors des évènements de traçage. Ce fichier est écrit dans un langage de programmation propre à SystemTap. Ce fichier sera compilé en un fichier C et ensuite en un module noyau Linux. Ce module utilise les bibliothèques kprobe et uprobe pour placer des points de trace dans le noyau et les processus usagers. Tous les évènements capturés sont enregistrés dans un fichier. SystemTap peut aussi créer des points de trace qui utilisent les points de trace statiques du noyau. De plus, l'outil peut enregistrer les informations de profilage du noyau.

Strace

Strace (27) est un outil qui permet d'instrumenter les appels systèmes utilisés et les signaux reçus par un processus. Cet outil utilise l'appel système *ptrace()* (28) pour obtenir les informations du processus à tracer. Chaque fois qu'un signal est reçu, ou qu'un appel système est exécuté, strace arrête le programme tracé, intercepte cette information, l'imprime sur la console, et continue l'exécution du programme tracé. L'utilisation de ptrace baisse la performance de façon significative car le programme tracé doit être arrêté à chaque fois qu'un appel système est utilisé.

3. Plusieurs langages utilisent le nom "D", dans ce cas-ci, on réfère au langage propre à Dtrace

eBPF

eBPF (extended Berkeley Packet Filter) est un outil permettant à un utilisateur d'exécuter du code dans le noyau Linux de façon sécuritaire. Cet outil a originalement été développé pour rapidement filtrer les paquets en entrée du système. Un utilisateur peut produire un programme en bytecode (BPF) indépendant de l'architecture qui sera rapidement exécuté par le noyau pour filtrer l'entrée des paquets. Depuis, cet outil a été utilisé dans d'autres parties du système pour permettre à des utilisateurs d'ajouter des fonctions dans le noyau de façon sécuritaire. B. Gregg (29) discute des différentes utilisations d'eBPF dans son article. Pour utiliser eBPF, le noyau expose l'appel système *bpf()* qui permet de contrôler les opérations de cet outil. Pour créer un point de trace eBPF l'utilisateur doit tout d'abord créer un bytecode BPF à installer dans le système. Ensuite, l'utilisateur doit créer une zone mémoire partagée entre le noyau et le programme utilisateur, appelé "map". L'utilisateur peut ensuite créer le point de trace eBPF qui sauve l'information dans le "map". Lors de l'exécution du traceur, l'utilisateur peut lire l'information de la zone de mémoire partagée. Pour installer le point de trace eBPF, les bibliothèques kprobe et uprobes, les points de trace statiques et les compteurs de performance sont utilisés.

Pour générer le bytecode, l'outil BCC (30) peut être utilisé pour écrire des programmes eBPF. Cet outil est basé sur le compilateur LLVM et clang, et génère le bytecode BPF à partir d'un programme écrit en C ou C++. Seulement les fonctions intégrées à BPF sont disponibles au programme. L'outil offre aussi des programmes pour facilement insérer des points de trace eBPF dans le noyau et dans les programmes usagers.

Perf

Perf (31) est un outil de profilage pour le système Linux. Cet outil rassemble l'information de plusieurs sources afin de présenter une vision d'ensemble de l'état du système. La première source d'information disponible est les compteurs matériels présents dans le processeur. x86 possède plusieurs registres accessibles par le noyau qui comptent des évènements matériels spécifiques. Le noyau présente l'interface *perf_event* qui permet d'accéder aux fonctionnalités de perf. L'outil utilise cette interface pour instrumenter le noyau. L'outil perf peut aussi utiliser les traceurs matériels, tels Intel PT, afin de tracer le processeur.

Le noyau Linux offre des compteurs pour les évènements logiciels exécutés dans le noyau. Perf peut obtenir ces compteurs à partir du pseudo-système de fichier Sysfs. Le nombre de changements de contexte et le nombre de fautes de page sont des exemples de compteurs offerts par le noyau Linux.

Les points de trace statiques du noyau Linux peuvent aussi être utilisés par l'outil perf. Perf ajoute un rappel pour les points de trace noyau lorsqu'activé. Perf offre deux façons de présenter les informations : par compteur, ou par évènement. La présentation par compteur montre le nombre d'exécutions des évènements tracés. La liste de chacun des évènements tracés ainsi que leur compteur est présenté par l'outil à la fin de l'exécution. La présentation par évènement montre la liste des évènements individuels avec leurs paramètres. Tout comme le fichier *trace* du Tracefs, perf montre la liste des évènements, leurs paramètres et leur valeur d'horloge au temps d'exécution de l'évènement.

Perf offre une interface pour la création de points de trace dynamiques dans le noyau et dans les programmes usager. Avec perf, l'utilisateur peut placer un point de trace à n'importe quel endroit dans le système. Ce point de trace a accès aux registres, aux variables et aux valeurs de retour du noyau et des processus usager. Pour placer un point de trace dans le noyau, perf crée un point de trace kprobe. Si perf est utilisé pour placer un point de trace dynamique au début d'une fonction, l'interface ftrace sera utilisée à la place. Uprobe sera utilisé pour placer un point de trace dans les processus usager.

Les types d'instrumentation mentionnés pour perf supportent plusieurs filtres qui réduisent la quantité d'information générée. Tout d'abord, l'utilisateur peut utiliser un seuil "X" pour tracer à chaque "X" d'un type d'évènement. Ceci permet d'échantillonner une partie des évènements, dans le cas où une grande quantité d'évènements d'un type sont exécutés. Ensuite, il est possible de filtrer selon l'identifiant de processus (PID), ce qui permet de tracer les activités de certains processus précis. Finalement, il est possible de faire un échantillonage périodique des évènements. Ceci permet d'obtenir périodiquement les valeurs des compteurs d'évènements du système.

Perf offre aussi des analyses de sous-systèmes précis. Celles-ci utilisent des évènements précis des sous-systèmes afin de présenter une vue de plus haut niveau et une synthèse de l'état de sous-systèmes du noyau. Ces analyses se basent sur les traceurs de sous-systèmes offerts par ftrace ainsi que sur les fonctionnalités de base de perf afin d'obtenir l'information utile à l'analyse. Perf offre aussi des outils pour créer des analyseurs personalisés qui utilisent l'information offerte par perf.

Finalement, perf offre une interface en ligne de commande pour insérer des programme eBPF dans le noyau Linux. Ainsi, un usager peut écrire un programme eBPF, le compiler et l'exécuter grâce à l'interface perf. La ligne de commande permet de présenter les informations de sortie du programme eBPF grâce aux "maps" créés par celui-ci. Perf utilise l'appel système *bpf()* afin de contrôler le programme usager.

Sysdig

Sysdig (32) est un outil de traçage spécialisé au traçage de conteneurs exécutant dans un environnement Linux, MacOS, et Windows. Cet outil utilise les points de trace statiques et l'instrumentation des appels système pour créer une vue d'ensemble de l'exécution des conteneurs et de l'hôte. Un module noyau Linux est créé pour installer l'instrumentation nécessaire sur le système. Lorsque des évènements tracés sont lancés, le module écrit dans un tampon circulaire partagé avec l'espace utilisateur. Un processus usager lit ce tampon, et sauvegarde les informations dans un fichier si nécessaire. L'information lue par le processus peut aussi être analysée par d'autres outils proposés par le projet Sysdig. Sysdig organise l'information reçue du noyau selon le conteneur qui a émis cette information, dans le but d'obtenir une vue cohérente du système tracé.

2.2.4 Modification dynamique de programmes

Valgrind

Valgrind, un outil créé par Nicholas Nethercote et al. (33) (34), est une plateforme de modification et d'instrumentation dynamique. Le projet est séparé en deux composantes principales, le coeur ("core") et les outils ("skins"). Le coeur remplit deux fonctions principales : offir une bibliothèque d'instrumentation aux outils, et piloter l'ensemble des opérations d'instrumentation d'un processus. Valgrind n'exécute jamais directement le programme sous test, mais crée plutôt un programme équivalent instrumenté en temps réel. Pour chaque bloc de base, Valgrind le décompile en représentation intermédiaire (IR) indépendente de l'architecture, lui ajoute l'instrumentation, et le recompile en code machine. Les outils peuvent modifier le IR généré par le coeur pour intercepter l'exécution ou modifier l'état du programme. Valgrind exécute ensuite le code généré, qui retourne le contrôle à Valgrind à la fin de l'exécution du bloc de base. Le processus est répété pour chaque bloc de base du programme. Si le programme instrumenté utilise plusieurs fils d'exécution, ceux-ci sont sérialisés en un seul fil d'exécution actif à la fois. Les signaux sont aussi interceptés par Valgrind et sérialisés avec les autres fils. Tous les appels système sont aussi interceptés par Valgrind, qui vérifie que ceux-ci n'interfèrent pas avec l'exécution de l'instrumentation. Valgrind ralentit considérablement l'exécution du programme instrumenté, puisque l'exécution est constamment interrompue par l'outil. La sérialisation des fils ralentit aussi l'exécution des programmes multi-fils.

Plusieurs outils sont disponibles avec le projet de base (35). Memcheck instrumente toutes les instructions qui accèdent à la mémoire pour détecter les erreurs. Cet outil peut aussi détecter l'accès aux valeurs non initialisées, les fuites de mémoire et les mauvaises allocations

et désallocations. Cachegrind instrumente les accès à la mémoire pour profiler l'utilisation des caches du processeur. Cet outil donne une vue globale des erreurs de cache pour les instructions et données. Callgrind instrumente tous les branchements et les appels de fonction du programme. Cet outil produit un rapport sur l'arbre d'appel du programme et les branchements exécutés. Un rapport de la précision de la prédiction de branche est aussi émis. Helgrind est un outil de détection d'erreurs de synchronisation et d'utilisation de fils d'exécution. L'outil vérifie l'ordre d'utilisation des primitives de synchronisation pour détecter les erreurs d'interblocage. La mauvaise utilisation des bibliothèques de gestion de fils peut aussi être détectée. Massif est un outil de vérification de gestion de mémoire dynamique. L'outil instrumente les fonctions d'allocation et de désallocation de mémoire pour vérifier si le programme ne fait pas d'erreur d'allocation.

Pin

Pin, un outil créé par Bryan Buck et al. (36), est une plateforme d'instrumentation dynamique. Tout comme Valgrind, Pin permet le développement d'outils d'instrumentation grâce à une interface de modification de code. Pin permet l'instrumentation des fonctions, des blocs de base de des instructions. Pin utilise la même technique que Valgrind pour l'exécution du programme instrumenté : la recompilation dynamique des blocs de base avec instrumentation. Pour chaque bloc de base, Pin décompile le bloc vers une représentation intermédiaire (IR), ajoute l'instrumentation, et recompile le IR. Pour instrumenter un programme, Pin peut soit l'exécuter ou s'attacher à un processus existant. Si Pin a été attaché à un processus, l'utilisateur peut le détacher à n'importe quel moment. Ceci enlève toute l'instrumentation qui a été ajoutée par l'outil. Pin n'a pas de restriction sur le nombre de fils d'exécution ou sur les appels système effectués par le processus sous test.

DynInst

DynInst est une bibliothèque C++ de modification dynamique de programmes. Cette bibliothèque a été créée par William Williams (37) (38) et offre une interface, indépendante de l'architecture, d'analyse et de modification dynamique de programme. La bibliothèque permet à un utilisateur d'insérer un "snippet" à un "point" dans un processus existant. Un "snippet" est un morceau de code créé par la bibliothèque et un "point" est un point d'insertion dans le processus à modifier. Pour générer un "snippet", plusieurs primitives sont disponibles. Chaque primitive représente une opération commune à la plupars des architectures : accès mémoire, opérations arithmetiques, branchements, conditions, etc. À partir d'un "snippet", la bibliothèque génère le code machine équivalent. Pour insérer le code dans le processus, Dy-

nInst utilise l'appel système *ptrace()*. Durant l'insertion, le processus visé est bloqué. DynInst copie tout d'abord le "snippet" dans l'espace mémoire du processus. Ensuite, l'instruction ou les instructions au "point" sont remplacée par une instruction de branchement. Puisque l'exécution est bloquée par *ptrace*, DynInst peut vérifier que le remplacement est sécuritaire. Les instructions remplacées seront exécutées hors-ligne avant l'exécution du "snippet".

ksplice

Ksplice (39) est un ajout au noyau Linux qui permet de mettre à jour le noyau durant son exécution. Ksplice est composé de deux parties : le module noyau qui modifie le code du noyau, et l'outil qui génère la mise à jour à ajouter. À partir de la source du noyau et d'une modification à la source, l'outil génère deux fichiers objets : l'objet avant la mise à jour, et l'objet après la mise à jour. Le module noyau compare ces deux fichiers et détermine quelle sont les fonctions qui ont changé. Durant la mise à jour, Ksplice bloque tous les processeurs sauf le processeur de mise à jour. Pour chaque fonction à mettre à jour, le module doit résoudre les adresses de tous les symboles référés par la nouvelle version de la fonction. Ensuite, le module remplace les premières instructions de la fonction par un saut vers la nouvelle version. Une fois toutes les fonctions redirigées vers leurs nouvelles versions, le module débloque tous les processeurs. L'outil permet aussi de défaire la mise à jour en enlevant tous les branchements ajoutés par la mise à jour. L'outil ne permet pas de placer une mise à jour qui modifie une structure de donnée, puisque ceci demanderait de réallouer toutes les structures de données utilisées par les fonctions à mettre à jour.

Conclusion de la revue de littérature

Plusieurs techniques sont disponibles afin d'instrumenter un système. Le traçage avec instrumentation statique permet à un développeur d'ajouter des appels à des bibliothèques de traçage afin d'enregistrer les évènements du système. Les compilateurs offrent aussi des fonctionnalités d'ajout de point de trace dans le code. Ceux-ci génèrent le code nécessaire au traçage lors de la compilation. Certains compilateurs peuvent aussi placer des instructions no-op à des endroits stratégiques afin de permettre le traçage ultérieur. Le traceur peut ensuite remplacer ces instructions par des appels au traceur, au besoin de l'utilisateur. Ces outils ont le désavantage principal qu'il est impossible de changer les points de trace placés sans recompilation.

Les traceurs dynamiques pallient à ce problème en permettant le traçage de programme sans la modification de ceux-ci. Les traceurs aidés du matériel peuvent être utilisés, mais ceux-ci

produisent de l'information de très bas niveau et n'offrent pas beaucoup de flexibilité. Ceux-ci ne nécessitent aucune modification de programme, mais simplement l'activation de leurs fonctions de traçage. Les traceurs dynamiques logiciels modifient le code machine du processus à tracer, afin d'y ajouter les points de trace. La modification dynamique de programme est précaire, car plusieurs fils peuvent être en exécution et il est facile de causer des problèmes d'exécution. Quatre techniques sont généralement utilisées durant l'insertion : l'arrêt du processus durant l'insertion, l'insertion atomique d'instructions d'interruption, l'insertion atomique d'instructions de branchement, et la recompilation en temps réel avec instrumentation du processus. Dans tous ces cas, les traceurs doivent prendre en compte l'état de tous les fils d'exécution, ainsi que la structure du programme à modifier. Certains traceurs, comme GDB et DynInst, utilisent plusieurs techniques d'insertion dynamique simultanément.

CHAPITRE 3 PROBLÉMATIQUE

L'instrumentation statique est en général préférable à l'instrumentation dynamique lorsque possible. En effet, l'ajout de code de traçage par le développeur n'implique en général qu'un appel de fonction à la bibliothèque de traçage choisie. Il est possible d'exécuter le code de traçage conditionellement si l'utilisateur ne veut pas tracer certaines parties du système, ce qui peut être implémenté avec une simple condition. L'instrumentation statique est en général plus rapide et plus stable que l'instrumentation dynamique. Il est aussi possible pour le compilateur de générer ces appels, au début de chaque fonction par exemple. Tout changement aux appels de traçage demande bien sûr une recompilation des parties du système qui ont été modifiées. L'utilisation des points de trace statiques a aussi un impact sur la performance, même si ceux-ci sont désactivés. L'exécution d'un point de trace statique demande au minimum la vérification d'une condition. Si l'impact de performance est trop grand, il est aussi possible pour le compilateur de générer des no-ops à la place des appels de traçage. Ainsi, un outil peut remplacer ces instructions par des branchements qui invoquent le traçage, au besoin de l'utilisateur. Les instructions no-op ont un impact négligeable sur la performance lorsqu'exécutées. Par contre, cela demande quand même une recompilation si un utilisateur veut changer le placement des points de trace. Si la recompilation n'est pas souhaitable, plusieurs techniques de traçage complètement dynamiques sont disponibles.

Le traçage dynamique peut être accompli avec l'aide du processeur. Plusieurs architectures proposent un sous-système de traçage dynamique incorporé au processeur. Ces traceurs sont controlés par l'entremise de registres et d'instructions spécialisés. En général, ces traceurs tracent toute l'exécution du système, avec quelques filtres limitant la quantité d'information produite. Le nombre de points de trace dynamiques est en général très limité et les évènements enregistrés sont de très bas niveau [1]. Ces traceurs sont très restrictifs et ne peuvent être utilisés que par du code en mode kernel. Il n'est pas possible d'utiliser le traçage assisté par matériel dans tous les cas, car le coût en performance peut être très grand. Le support n'est pas non plus présent dans tous les modèles de processeurs d'une architecture.

Pour accomplir le traçage avec instrumentation dynamique de façon complètement logicielle en x86, l'outil de traçage doit remplacer une ou plusieurs instructions au point d'insertion du point de trace. Deux instructions à insérer sont considérées : une interruption et un branchement. Le but de ces instructions est d'invoquer le code de traçage à partir du code que l'utilisateur veut tracer. L'instruction d'interruption ne fait qu'un octet et ne remplace

1. Branchements pris, interruptions, changement de contexte, accès mémoire, ...

donc qu'une seule instruction. L'interruption est très lente comparée au branchement, mais est beaucoup plus flexible. En effet, il est possible d'atomiquement remplacer le premier octet de l'instruction visée par une instruction d'interruption. La deuxième option est d'utiliser un branchement pour invoquer le point de trace. Les branchements relatifs en x86 ont une taille de 5 octets, ce qui pourrait remplacer plusieurs instructions lorsqu'on place celui-ci. Ceci apporte plusieurs problème, comme mentionné dans le Chapitre 1. Une solution est d'empêcher l'utilisation d'un branchement si la longueur de l'instruction remplacée n'est pas d'au moins 5 octets. Ceci restreint grandement le nombre de points d'insertion de point de trace, puisque la grande majorité des instructions ont une longueur inférieure à 5 octets (40). Une autre solution est de tout d'abord vérifier qu'aucun branchement du programme n'atterrit à une des instructions remplacées par le branchement. Si c'est le cas, l'outil peut arrêter tous les fils du programme et vérifier qu'aucun d'eux n'exécute du code à remplacer. Si ces deux conditions sont respectées, un branchement peut être inséré sans problème. Cette solution est plus facile à appliquer à partir du code d'un système d'exploitation, où il est possible d'arrêter l'exécution de tous les processeurs. Pour l'espace utilisateur, des appels système comme *ptrace()* et *tgkill()* peuvent être utilisés avec Linux. Cette solution perturbe l'exécution du système et n'est pas applicable dans tous les cas. Pour utiliser le traçage avec instrumentation dynamique de manière complètement logicielle, l'utilisateur doit choisir entre un point de trace performant (branchement) et un point de trace flexible (interruption).

CHAPITRE 4 ARTICLE 1 : FAST & FLEXIBLE TRACEPOINTS IN X86

Submitted to *Journal of Software : Practice and Experience*
Christian Harper-Cyr
Michel R. Dagenais
Ahmad Shahnejat Bushehri

Tracing is often the most effective technique for analyzing the performance of complex multi-threaded applications. This paper presents an improvement on existing techniques for dynamic tracepoint insertion. To add a tracepoint, the technique inserts a jump at the tracing point, possibly replacing several shorter instructions. This jump embeds trap instructions inside its offset at the address of every replaced instruction. This makes the jump thread-safe if any thread is about to execute a replaced instruction. It also makes it jump safe if a branch landing pad is at one of the replaced instructions. In both cases, a trap will be raised and the thread can be redirected to the out-of-line equivalent instruction. The use of a jump instead of a trap to execute the tracepoint improves the performance of the execution. It also adds the flexibility to place the tracepoint at most instructions, since multiple instructions can be replaced atomically and safely. This is a significant improvement over existing dynamic instrumentation techniques. The downside of this technique is the increased memory usage since it requires unaligned allocations with high external fragmentation.

4.1 Introduction

A tracing tool allows developers to find out the internal state of a process while disturbing as little as possible its execution. It may be used to find bugs in the program, to find performance problems, or to know the target process execution state. A tracer can be seen as a logger for low level information. It must provide the low level state of a process, without disturbing its normal execution, since this information is used to find problems that may not appear if the process is disturbed (performance, multiprocessing, real-time). This paper presents an improvement on existing dynamic tracepoint insertion techniques. It enables any process, without compiler modification, to be modified in order add a tracepoint at almost any point in the process. When using existing techniques, the tracepoints were either inserted using traps, which can be placed anywhere but are very slow, or inserted using jumps, which are fast but can be placed only at selected locations. This paper presents a solution for a jump tracepoint that can be placed almost anywhere, combining the advantages of a trap tracepoint

and of a jump tracepoint. This paper first lays out the problem and the proposed solution. Then, the implementation and execution of the technique are detailed. To conclude, results comparing this technique to existing ones are shown, followed by a discussion of the different implications.

4.1.1 Static & Dynamic Tracepoints

There are two different types of tracepoints that can be added to a program : static and dynamic tracepoints. A static tracepoint is a tracepoint that is present in the executable once it is compiled and linked. Usually, a function call or a macro is written in the code of the traced program. Another possibility for static tracing instrumentation is compiler inserted instrumentation. This type of tracepoint is inserted using compiler/linker command line arguments. An example of this is GCC mcount. This tool can instrument the entry and exit of every function by inserting code at the beginning/end of functions when compiling.

Alternatively, dynamic tracepoints are not present in the executable and are instead added when the process is being run. This sort of tracepoint necessitates a modification of the process code while it is executing. Section 4.2 presents some of the existing tracing tools with dynamic instrumentation. Because of the nature of dynamic tracepoint insertion, thread-safety and jump safety are important issues. The tools have to be thread safe for insertion and removal (if supported) since a thread could be executing the code being modified. This paper considers two ways of implementing a tracepoint : with a trap instruction or with a jump instruction. The difference is explained in Section 4.1.2.

Tracepoints that qualify both as static and dynamic also exist. These usually necessitate compiler support and also modify the program once it is launched. Examples of this are the ftrace and uftrace tools, presented in Section 4.2.3.

4.1.2 x86 Instruction Encoding

The x86 architecture uses a variable length encoding for its instructions ; not all instructions have the same length, which ranges from 1 byte to 15 bytes [1]. Also, instructions do not have any alignment requirement, which makes it possible to jump in the middle of an instruction and to embed instructions inside other instructions. For dynamic tracepoints implementation, two instructions can typically be used to divert the execution flow to the tracepoint code : the trap instruction and the jump instruction. The trap instruction is a 1 byte software interrupt instruction (int \$3, or $0xCC$) that causes the trap interrupt handler to be called. If this

1. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 chapter 2.3.11 page 2-20 ((1))

instruction is called from user mode, the processor will switch to kernel mode. The kernel can then notify the process that the trap instruction was executed (via a signal) and the tracepoint handling can begin. The other instruction is a jump instruction. This instruction is a 5 bytes instruction that has an 4 bytes integer offset as argument. This instruction branches to an address using this formula : $target = current + 5 + offset$, which means that it can only jump 2GB forward or backward. While the jump instruction execution is much faster than a trap instruction, it is bigger and harder to safely place in the program (see Section 4.3.1).

4.2 Related Work

4.2.1 GDB

The GNU Debugger (GDB), (22), is a user-space debugging and tracing tool that uses both traps and jumps to implement tracepoints. When creating a tracepoint, the user has the choice between creating a tracepoint using a jump instruction (fast tracepoint) and a trap instruction (normal tracepoint). To use a jump, the target instruction must be 5 bytes or longer, otherwise it uses a trap instruction. When using a trap, the kernel will send a signal to the process once the instruction is executed. The signal handler will do the tracing and then return to the traced location. When using a jump, the instruction will branch to the handler that will trace and then return to the traced location. The GDB tracer also uses debugging information to trace the state of the process.

4.2.2 Kprobes & Uprobes

Kprobes and uprobes are two Linux dynamic tracing tools implemented in the kernel, using the same underlying technique. These tools can place tracepoints in any running user-space process (uprobe) or in the Linux kernel (kprobe). These tracepoints can access any register and memory location that the process (or kernel) can access at the insertion point. These tools can also place tracepoints which will trace the return of a function. The uprobe interface is exposed as files in the tracefs or debugfs pseudo-filesystem ((21)), that can be written to and read from, in order to control the uprobes placed on different processes in the system. Kprobe ((17)) is a programmatic library in the Linux kernel that can place a probe (tracepoint) at any address in the kernel with a specified handler to execute when the probe is hit. Kprobe also exposes an interface via files in tracefs or debugfs ((20)) and works in the same way as the uprobe interface. Both tools use a trap to implement the tracepoints. The interrupt handler can then do the tracing (using uprobe or kprobe, depending on the location of the

trap) and return to the location of the code. In the case of kprobe, the handler function is called and the tracing is left to the user of the library. For uprobe, the tracing is recorded in a pseudo-file in tracefs (or debugfs) that can be read by the user during the tracing session. Both tools expose a fairly low-level interface to tracing which is used by some other tools to create higher level interfaces. Kprobes also support tracepoints using a jump instruction, but only if the target instruction is at least 5 bytes long or if it can prove with simple flow analysis that no jump within the targeted 5 bytes can occur.

4.2.3 Ftrace & Uftrace

Ftrace and uftrace are two tools that use dynamic tracepoints with jump and with compiler support. In order to use these tools, the target process has to be compiled with 5-bytes no-op instructions [2] placed as the first instruction of every function that the user wants to trace. This instruction ensures that it is safe to place a jump at the beginning of every function in the process. Once the program is launched, the tools will replace the traced function first instruction (no-op) with a jump to the tracing handler, when the tracepoint is activated. These tracepoints can also trace the return of the function. Ftrace is implemented in the Linux kernel and is used to trace function calls in the kernel itself. When compiling Linux, it is possible to add no-ops at the beginning of every function, which enables ftrace to be used. The interface to control ftrace is found in debugfs/tracefs using pseudo-files that the user can write to and read from. Ftrace also supports multiple tracers, that will each trace different information. Uftrace is a user-space tool that traces user processes. When invoking the process, it is possible to trace some or all functions in the executable. This tool also supports output to multiple formats.

4.2.4 Dyninst

(37) created a library called Dyninst that can instrument, analyse and modify binary programs before or at runtime. This library presents multiple APIs that implement a complete ecosystem to modify existing binary executables or to manipulate processes at runtime. While it can be used to instrument a process, this library is very generic and can be used for more. This library abstracts machine instructions into "snippets" and "points". A "snippet" is a vector of instructions to be inserted into a target executable. The "point" represents an insertion point for the "snippet". A "snippet" can be generated using architecture-independent pseudo-instructions that implement the instructions present in all architectures. When executing a

2. *nopl* $0(\%eax, \%eax, 1)$ or *nop dword ptr* $[eax + eax * 1 + 00H]$, from Intel 64 and IA-32 Architectures Software Developer's Manual Vol 2B page 4-163, (1)

"snippet", a branch instruction is used to jump to a trampoline (on x86) that will save the state, call the "snippet", restore the state and jump back to the original location.

4.2.5 Valgrind

Valgrind is an instrumentation framework created by Nicholas Nethercote and al. (33) (34). This framework is composed of two parts : the core, and valgrind tools (memcheck, callgrind, cachegrind). The valgrind core has three purposes. The first is to initialize the framework and to run the selected tool. The second purpose is to implement utility functions that can be used by the tools to implement the instrumentation. The third is to invoke the tool when it has to instrument the target process. To instrument a program, valgrind does not run the executable directly. Instead, for each block, it decompiles the executable into an architecture-independent intermediate representation (IR), adds instrumentation according to what the tool needs, and recompiles the IR back into machine executable code. When invoked, the tools can analyse and modify the target code, and alter the execution of the process. Because of this decompile-modify-recompile cycle, the instrumented program runs between 5 (Nulgrind) and 100 (Memcheck) times slower than the original program.

4.3 Motivation

4.3.1 Problem

On x86, most problems with dynamic tracepoints stem from the fact that the instructions have variable length depending on the operation and memory addressing mode (see Section 4.1.2). There are two ways of executing a dynamic tracepoint that are considered in this paper, a trap instruction, which is 1 byte long, and a relative jump, which is 5 bytes long. The first method also requires support from the operating system (or an interrupt handler if no OS is present) since the instruction will generate an interrupt. The second method requires a jump target within 2GB of the jump location, since the offset of a relative jump is a 4 bytes signed integer.

A trap tracepoint can be easily inserted anywhere in the target code for three reasons : the instruction is 1 byte long, which means it will only replace one instruction ; it can also be inserted atomically on all x86 processors that support atomics ; the handler can be at any address in the processes address space, since there is no restriction regarding the location of a signal handler (on most operating systems) or interrupt handler. The main problem with this technique is that an interrupt is slower than a branch on x86. If the target process is running with an operating system, it will add even more overhead since it requires a switch

from unprivileged (user) mode to privileged (kernel) mode in order to handle the interrupt, and back to call the signal handler. In performance sensitive code, the latency of an interrupt call is unacceptable and a jump must be used ((18)).

A jump tracepoint is harder to insert than a trap tracepoint because it is 5 bytes long. This means that the modification is not thread safe if any thread in the process is running. Another problem is that the jump could replace up to 5 instructions, which again, is not thread safe. If a thread has its instruction pointer at an instruction that was replaced, it will try to execute invalid code as soon as it runs. On top of that, if an instruction jumps to a replaced instruction, it will also execute invalid code. Figure 4.1 and Figure 4.2 show an example of a 5 bytes instruction, before and after being replaced with a jump, respectively. The ˆ symbol represents the current instruction pointer. The offset of the jump is irrelevant and is set to 0 for this example. Since the instruction to replace is 5 bytes long, the instruction pointer can never be in the middle of the instruction. Because of this, the instruction is always safe to be atomically replaced with a jump. Figure 4.3 shows an example of 3 instructions that span 5 bytes, with the instruction pointer set to the address of the second instruction. Figure 4.4 shows the jump that would replace those 3 instructions. Since the instruction pointer is set to the second instruction, the instruction that will be executed after the insertion will be *add %al, (%rax)*, which changes the behavior of the original program. In this example, the offset of the jump is 0, but the consequences are similar for any offset value that does not embed traps. If the program jumps back to one of the replaced instructions (except the first), or if a thread is currently executing one of the replaced instructions, the behavior of the program will most likely change and the program may crash. For these reasons, most implementations will only use a jump if the replaced instruction is 5 bytes or longer, since the jump will then replace only one instruction. There are also ways using traps to store 5 bytes in a thread safe manner on x86_32 (see Section 4.4.3). On the other hand, jump tracepoints are a lot faster than trap tracepoints, since it is no more costly than a normal function call.

4.3.2 Proposed Solution

This paper proposes a new solution using jump tracepoints, which addresses the shortcomings of this technique. The first problem with the jump is that it could replace multiple instructions, and that it is not thread/branch safe to do that. To address this problem, trap instructions are embedded in the offset of the jump, at the beginning of each replaced instruction. If a thread is still executing a replaced instruction, it will execute a trap instead. The trap handler (interrupt or signal) can then point the instruction pointer to the equivalent instruction that was copied in the tracepoint code (see Section 4.4.3). If an instruction

```
0x67 0x0f 0x1f 0x04 0x00    nopl 0(%eax, %eax, 1)
^                           ^
```

Figure 4.1 5 bytes no-op instruction

```
0xe9 0x?? 0x?? 0x?? 0x??    jmp <handler>
^                           ^
```

Figure 4.2 replaced 5 bytes instruction

jumps to a replaced instruction, the trap will also be executed and the instruction pointer set to execute the replaced instruction. Figure 4.5 shows the jump that will be inserted if the first instruction of Figure 4.3 is instrumented. After the replacement of the instructions, the processor will execute a trap. The Implementation Section (4.4) shows in more detail how to create, execute and destroy this kind of tracepoint. The Result Section (4.5) shows a comparison between this technique (implemented in the Dyntrace project) and different existing dynamic tracepoint techniques. Finally, the Discussion Section (4.6) compares the advantages and disadvantages of this technique relative to other dynamic tracepoint techniques.

This paper differentiates two types of tracepoints : "point" and "entry-exit". A "point" tracepoint is a tracepoint that traces only once when it is executed. It can be safely placed anywhere in the code, since it only does the tracing and does not modify the state of the process. An "entry-exit" tracepoint, on the other hand, can only be safely placed at the beginning of a function (at the location of the first instruction). When it is hit (when the traced function is called), it acts like a "point" tracepoint, except that it also replaces the return address (top of the stack on x86) with another handler. Once the "entry" handler is executed and the return address is replaced, it normally returns and executes the function. When the traced function returns, it will return to the "exit" handler that will also trace the return of the function. This handler then returns to the original call location, once all the handling is completed.

```
0x90 0x66 0x90 0x66 0x90    nop; xchg %ax, %ax; xchg %ax, %ax
     ^                           ^
```

Figure 4.3 Less than 5 bytes instructions

```
0xe9 0x00 0x00 0x00 0x00    ??; add %al, (%rax); add %al, (%rax)
     ^                          ^
```

Figure 4.4 Less than 5 bytes instructions, replaced with a jump

```
0xe9 0xcc 0x?? 0xcc 0x??    ??; int $3; ??; int $3; ??
     ^                          ^
```

Figure 4.5 Less than 5 bytes instructions, replaced with a jump with embedded traps

4.3.3 Dyntrace

In order to compare the technique presented in this paper, an implementation of the proposed fast tracepoints was created, called Dyntrace. This project is a collection of programs and libraries that enables a user to create, destroy, and manage jump tracepoints. The project is hosted on GitHub ((41)). This project consists of five major parts : the tracepoint library, the control daemon, the in-process agent, the tracers, and the command line client. Figure 4.6 shows the main components of the project. Blue nodes represent running processes and green nodes shared libraries. The tracepoint library is a library called dyntrace-fasttp ("libdyntrace-fasttp.so") that implements jump tracepoints. This library presents a programmatical interface that enables a user to insert a tracepoint at any address in the same address space. The control daemon is a process that manages every tracepoints on a machine. This daemon, called "dyntraced", has to run in order to add, remove or manage any tracepoint on the machine. To communicate with the daemon, it exposes two UNIX sockets, one for clients ("command.sock") and one for target processes ("process.sock"). This daemon routes the commands from the command line client to every in-process agent. Additionally, this

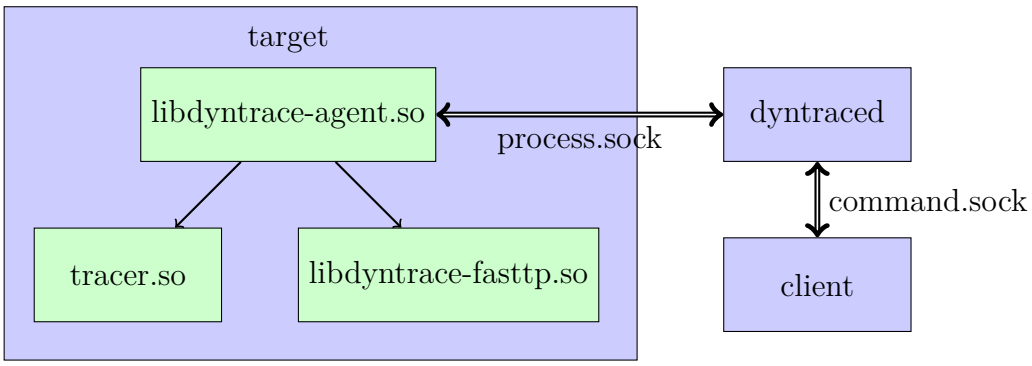


Figure 4.6 Dyntrace components

daemon is able to dynamically inject the in-process agent into any running process on the machine. Only one instance of the daemon can run at any time. The in-process agent is a shared object called dyntrace-agent ("libdyntrace-agent.so") that is loaded into a process in order to control its tracepoints and to receive commands from the control daemon. This library can be loaded in three ways : by linking to it during compilation, by preloading it using the LD_PRELOAD environment variable and the linux-ld loader, or by injecting it using the control daemon. Since this library is dynamically linked to dyntrace-fasstp, the latter will also be loaded when the former is loaded. Dyntrace-agent starts a thread on the target process that connects to the socket created by the daemon and waits for commands. The tracers are small shared objects that are loaded by the in-process agent ("tracer.so"). When creating a tracepoint, these libraries are loaded by the agent and called by the created tracepoint to execute the actual tracing. The user can choose which tracer to call when a tracepoint is executed. The last part of the project is the command line interface ("dyntrace") that connects to the command socket created by the control daemon. This client will send commands to the daemon, which will in turn send commands to the in-process agent. Figure 4.6 shows the five major parts of the project. Blue nodes represent processes, and green nodes represent shared libraries. The "target" component in the Figure represents the process in which the user wants to insert tracepoints.

— "libdyntrace-fasttp.so" : This library implements the insertion and removal of dynamic tracepoints using jumps ("fast" tracepoints). It is a C++ programmatical interface that can be used to insert tracepoints in the current process. This component is implemented as a shared library to save memory when multiple processes are traced at the same time. Since it is a library, it is also possible to use it outside of the ecosystem implemented by the project.

— "dyntraced" : This process is a daemon that manages every tracepoint present on a machine. This process creates two UNIX sockets to communicate with the other components, one for clients ("command.sock") and one for traced processes ("process.sock"). This daemon has three goals : centralize information about traced processes, route commands to the required processes, and implement the injection of the in-process agent ("libdyntrace-agent.so"). This process currently has to be run as root for two reasons : the injection process needs permissions that only root has by default, because the injection process uses ptrace internally (28) ; for security reasons, the sockets created by this process have the root user-id, and the "dyntrace" group-id. This alows only root and trusted users (members of the "dyntrace" group) to control tracepoints on the machine.

— "libdyntrace-agent.so" : This library is an in-process agent that is loaded into the

target process in order to control the tracepoints and to receive commands from the daemon. This library uses "libdyntrace-fasttp.so" to control tracepoints in its process. The use of an in-process agent, instead of an external process, makes it possible to atomically place a tracepoint without stopping any thread in the process, which is more efficient. When creating a tracepoint, this library will also load the required tracer into the process. To listen to commands from the daemon, this library starts a thread and waits for commands.

— "tracer.so" : This small library is loaded by the in-process agent when creating a tracepoint and is invoked when the tracepoint is executed. This tracer has access to function arguments, return value and variables when invoked. The use of small libraries to implement the tracing makes it easy to add new tracers and to extend existing ones. It also decouples the tracing control from the tracing execution.

— "client" : This component is a process that connects to the daemon to send commands to target processes. The Dyntrace project implements a command line interface, called "dyntrace", that acts as a client. Since the interface between the daemon and the client is well defined, new clients can easily be added if needed. Also, because the client connects to the command socket, it can be run as any user in the "dyntrace" group.

To create a tracepoint from command line, the following steps are followed.

1. The user starts the control daemon ("dyntraced") as root.
2. If needed, the user executes the command line client ("dyntrace") to inject the in-process agent library into the target process.
   i. The client sends the command to the daemon, via the command socket.
   ii. The daemon injects the dyntrace-agent library into the target process.
   iii. The daemon replies with the injection status back to the client.
3. The user executes the client to add a tracepoint in the target process, at a given location and using a given tracer.
   i. The client sends the command to the daemon, via the command socket.
   ii. The daemon routes the command to the selected process agent library.
   iii. The agent loads the required tracer into the process.
   iv. The agent creates a tracepoint at the chosen location that will call the tracer library.
   v. The agent replies with the creation status to the daemon, which will route this message back to the client.

4.4 Implementation

4.4.1 Overview

For this implementation, a tracepoint consists of three elements : a data structure, an execution structure (for both, see Section 4.4.2), and a jump to the execution structure. To create a tracepoint, both structures are created and the jump is inserted at the desired location. As mentioned earlier in Section 4.3.1, a relative jump is 5-bytes in length and can replace up to 5 instructions on x86. For this reason, the bytes in the jump offset for which an instruction has been starting, are replaced with a trap instruction code (0xCC), which is one byte long, for thread and jump safety. A tracepoint is said to be enabled when the jump is present in the target code, and disabled if it is not. A tracepoint can be disabled but still be allocated. A tracepoint is said to be deleted or deallocated when all structures are deallocated from memory.

To create a tracepoint, the structures will be allocated and the tracepoint will then be enabled (Section 4.4.3). To delete a tracepoint, the opposites steps are executed. First, it is disabled and then deallocated when known safe to do so (Section 4.4.5). When a tracepoint is executed, it jumps to the execution structure, traces the event, executes the replaced instructions and then jumps back to continue the execution after the inserted jump. If the tracepoint is an "entry-exit" tracepoint, it will also execute a handler when returning from the function (Section 4.4.4).

4.4.2 Tracepoint Structures

When creating a tracepoint, two interdependent structures are allocated : the data structure, which is allocated in the data mapping of the process, and the execution structure, which is allocated in the executable mapping. The first structure is shown in Figure 4.7. The "Reference count" is used to know if it is safe to deallocate the tracepoint. If over 0, is is not safe to do so. The "Execution structure address" and "Execution structure size" members contain the address and the size of the execution structure. The last member is the address of the user handler that does the actual tracing.

In Figure 4.8, we can see the members for a "point" tracepoint execution structure. The first two members [3] are the code that will be executed first when a tracepoint is hit. The member "tracepoint data address" contains the tracepoint data structure address. The next member, "generic handler address" contains the address of a generic tracepoint handler that is common

3. "skip red-zone" and "call generic handler"

Reference count
Execution structure address
Execution structure size
Handler address

Figure 4.7 Tracepoint data structure

for every "point" tracepoint. The remaining members are the code that will be executed when the "point" tracepoint exits. The execution of the "point" tracepoint is explained in Section 4.4.4. The execution structure of an "entry-exit" tracepoint is shown in Figure 4.9. The first member, "call generic entry handler", is the code that is executed when the tracepoint is hit. The second member, "call generic exit handler", is executed when the traced function returns. The third member is the pointer to the tracepoint data, the same as the "point" handler. The fourth and fifth members are the addresses of the generic entry handler and generic exit handler, respectively. The last members are the code that will be executed when the entry tracepoint exits.

4.4.3 Tracepoint Creation

The first step in creating a tracepoint is to find an allocation point for the execution structure. As mentioned in Section 4.3.2 and Section 4.4.1, the tracepoint jump offset has some bytes fixed to the value of a trap instruction code. Because of this, the tracepoint execution structure cannot be allocated using the operating system usual allocators. Instead, a special purpose allocator is needed to find an allocation address which satisfies the constraints on the fixed bytes in the jump offset. Once the address is found, the instructions that are replaced with the jump need to be adapted (patched) to run at another address, from within the execution structure. Finally, the tracepoint is inserted (enabled) atomically at the tracepoint target location.

skip red-zone (x86_64)
call generic handler
tracepoint data address
generic handler address
out of line code
jump back

Figure 4.8 "Point" tracepoint execution structure

call generic entry handler
call generic return handler
tracepoint data address
generic entry handler address
generic return handler address
out of line code
jump back

Figure 4.9 "Entry-exit" tracepoint execution structure

Tracepoint Allocation

To find a valid address for the execution structure, the total size of this structure must be known. Every member of the execution structure (for "point" and "entry-exit" tracepoints) has a fixed size, except for the "out-of-line" member, which contains the patched instructions that were replaced by the jump. To find the size of this member, the replaced instructions have to be analyzed before the allocation to find the patched instruction size (which may not be the same as the replaced instructions size, see Section 4.4.3 for more details).

Once the size is known, the algorithm will try to find the first location in free memory that is large enough to fit the execution structure and that respects the fixed bytes in the offset. The algorithm assumes that the process can allocate memory at specific locations with a byte granularity, which is not sufficient on most operating systems. Indeed, most operating systems allocate at fixed locations with a page granularity. However, the algorithm can easily be extended to allocate the pages it needs to satisfy the conditions on the execution structure location. The first step is to find the range of reachable addresses from the tracepoint location, which is $2GB$ addresses before and $2GB-1$ addresses after the end of the jump [4]. This range is called the reachable range. To do this, it has to find the minimal and maximal addresses that have the bytes fixed to 0xCC. Then, for every free memory range in the address space, it calculates the intersection with the reachable range, which gives the valid range. The last step is to find the first address in the range that satisfy the offset with the fixed bytes. We calculate the offset from the tracepoint location to the beginning of the valid range. Every byte in the offset, starting with the least significant, is set to 0xCC if it is the beginning of a replaced instruction. After setting each byte, if the target address goes under or over the valid range, the next byte is incremented/decremented to put the target address in the valid range. If it is not possible, the range cannot contain the execution structure. Once all bytes are set, it checks that the target address plus execution structure size is smaller than the end

4. The offset is a signed 32-bit integer and the target is calculated from the end of the jump instruction in x86 ($target = address + 5 + offset$)

of the valid range. If that is the case, the structure can be allocated there. The data structure is then allocated using the usual data allocator [5] and every member, except the "out-of-line" member, can be initialized.

Out-of-line instruction patching

The "out-of-line" member is meant to execute the instructions that are overwritten by the jump to the execution structure. For x86, there are four categories of instructions that may require different patching to compensate for the fact that they are executed from a different address : branching instructions, ip [6]-relative instructions, ip-relative branching instructions (x86_64 only), and everything else. In x86_32, ip-relative instructions do not exist, but compilers will generate special function calls that will put the value of the instruction pointer in a general purpose register. This technique is considered ip-relative for the purpose of this algorithm. For instructions that are not branches or ip-relative, the bytes can be copied as is to the "out-of-line" member.

For branching instructions, the offset is recalculated from the address of the "out-of-line" instruction. If the target is unreachable, a register is saved on the stack, the address of the target is put in the register, the old value is swapped with the new value of the register and the "ret" instruction is called. Listing 4.10 shows an example of a branch patch. If it was a call, the return address is also pushed on the stack. There are also four special branching instructions that require more instructions : "*loop*", "*loopz*", "*loopnz*" and "*jecxz*". These instructions have to be replaced with multiple instructions emulating the behavior of the replaced instructions. These are the steps to emulate the instructions : push the flags ; test

5. malloc, new or other
6. instruction pointer, eip for 32-bit, rip for 64-bit

```
call $offset
; Becomes
push %rax
movabs $return_address, %rax
; The code above is not generated for jumps
push %rax
movabs $(offset + original_ip), %rax
xchg %rax, (%rsp)
ret
```

Figure 4.10 Patched branch instruction

```
    loopz target
    ; Becomes
    lea -1(%rcx), %rcx ; For loop, loopz and loopnz
    jnz 0f ; For loopz, jz for loopnz
    jmp target
    ; The code above is not generated for loop and jecxz
0:  pushf
    test %rcx, %rcx
    jz 1f ; For loop, loopz and loopnz, jnz for jecxz
    popf
    jmp target
1:  popf
```

Figure 4.11 Patched loop/loopz/loopnz/jecxz instruction

the condition on the zero flag (for "*loopz*" and "*loopnz*") and jump to the original target if it is fulfilled ; test the "*cx*" register and jump to the original target if the value is not 0 (or 0 for "*jecxz*") ; restore the flags register. It is important to note that before jumping to the target, the flags register also has to be restored from the stack. If the target address is too far, the generated jump is patched using the same steps as described before. Listing 4.11 shows the generated assembly.

For ip-relative instructions, different methods are used for x86_32 and x86_64. For x86_32, the current instruction pointer is retrieved with a call to a small function that puts the pushed value into a register. This instruction is replaced with a move of the target into the same register. Listing 4.12 shows the generated assembly. For x86_64, a register that is not used by the instruction is saved on the stack, the target address is then loaded in that register. Next, the instruction is adapted to do a memory reference using the register, it is finally restored from the stack. Listing 4.13 shows the generated assembly.

To patch an ip-relative branching instruction, the return address (only for a call) and the value of an unused register are first pushed onto the stack. The target address is then loaded

```
call .get_pc_thunk.bx
lea %eax, offset(%ebx)
; Becomes
mov $(offset + original_ip), %ebx
lea %eax, (%ebx)
```

Figure 4.12 Patched x86_32 ip-relative instruction

```
lea %rax, offset(%rip)
; Becomes
push %rdx
movabs $(offset + original_ip), %rdx
lea %rax, (%rdx)
pop %rdx
```

Figure 4.13 Patched x86_64 ip-relative instruction

```
call *offset(%rip)
; Becomes
push %rax
movabs $return_address, %rax
xchg %rax, (%rsp)
; The code above not generated for jmp
push %rax
movabs $(offset + original_ip), %rax
mov (%rax), %rax
xgch %rax, (%rsp)
ret
```

Figure 4.14 Patched x86_64 ip-relative branch instruction

into the saved register. This register is then dereferenced into the same register. The value is exchanged with the value on the top of the stack (the saved register value) and the "ret" instruction is called. Listing 4.14 shows the generated assembly for an ip-relative branch.

For every replaced instruction, one or multiple out-of-line equivalent instructions are created in the execution structure. This usually means that the size of the resulting out-of-line member is larger than the size of the original instructions.

Tracepoint Insertion

To insert a tracepoint in a thread-safe manner, we need to atomically place a 5-bytes jump at the location of the traced code. On x86_64, the 5-bytes can atomically be put using a 8-byte atomic store. For x86_32, the 5-bytes can atomically be put using a 8-byte atomic compare-exchange (*cmpxchg8b*) operation.

4.4.4 Tracepoint Execution

"Point" tracepoint

When a tracepoint is hit, it jumps to the first member of the execution structure. On x86_64, the first instruction that is executed is to skip the "red-zone" [7]. It then does a call to the generic handler, which effectively pushes the address of the data contained in the execution structure. This data contains the address of the data structure. The generic handler then saves the state (pushes all general purpose registers), atomically increments the tracepoint reference counter (member of the tracepoint data), does the tracing, sets the return address to the out-of-line code, decrements the reference count, restores the states and returns from the call, which puts the instruction pointer at the out-of-line code. It then executes the patched out-of line code and then jumps back to the traced function.

"Entry-exit" tracepoint

When executing an "entry-exit" tracepoint, we assume that it is placed at the first instruction of a function. Figure 4.9 shows the structure of an "entry-exit" tracepoint execution structure. When jumping to the allocated tracepoint code, it immediately calls the generic entry handler, which pushes the address of the return handler on the stack. The address of the data structure can also be found after the exit handler. It then saves the state, increments the reference count of the tracepoint, and does the tracing. After this, it replaces the traced function

7. useable 128 bytes zone above the stack pointer, required by the x86_64 System V ABI (42)

return address with the return handler address (which was pushed earlier), saves the old return address in a thread-local variable, and advances the return address to the out-of-line code. It then restores the state, executes the out-of-line code, and then jumps back to the traced function. When this function returns, it will jump to the return handler. This return handler does a call to the generic return handler, which pushes the address of the address of the tracepoint data. The generic return handler (second row in Figure 4.9) then saves the state, executes the return tracing, restores the old return address, decrements the reference count, restores the state and returns to the original location.

4.4.5 Tracepoint Removal

There are two steps to remove a tracepoint : to disable and to deallocate it. In order to disable the tracepoint, the instructions that were replaced are written back into the original location, reversing the technique used for the insertion (Section 4.4.3). Once the tracepoint is disabled, the two structures are put in a list for later deletion. The reason to delay the deallocation is to protect a thread that could be executing the tracepoint structure code while it is being disabled. To ensure that no thread is executing a tracepoint, two methods are used. The first is the reference count, found in the data structure. If this reference count is not 0, it is unsafe to deallocate the thread at that moment. Unfortunately, it is impossible to protect the whole tracepoint using only a reference count, since some unprotected code has to run before and after the increment/decrement of the reference count. To ensure that no thread is running any unprotected code, the deallocator checks the value of the instruction pointer. If it is executing tracepoint code before or after the reference count protected code, the tracepoint is not safe to remove. One way to check the value of the instruction pointer is to stop every thread with a signal and verify the value of the instruction pointer in the signal handler. Since this is a very disruptive operation, it is better to only periodically check the reference count and the instruction pointer. If both the reference count test and the instruction pointer test pass, it is safe to deallocate the tracepoint structures, in any order.

Another technique that can be used to remove a tracepoint is to handle the tracepoint as a RCU data structure (8). This technique is similar to the previous with a few differences. Instead of using a per-tracepoint reference count, a per-thread counter can be used for all tracepoints. When a tracepoint is entered, a global counter value that is not 0 is copied into the local counter. When the tracepoint exits, the same thing is done. When trying to reclaim a tracepoint, the deallocator saves the current global value into a temporary variable, increments it, and checks that every thread counter value is higher than the temporary value. If that is the case, it is safe to deallocate the tracepoint. As before, the deallocator has to be

called periodically to check the value of the instruction pointer, since some instructions are not protected by the counter. The advantage of this technique is that there is no write to a shared variable when executing a tracepoint, which can lead to performance improvements. However, it requires the use of a thread local variable that must be accessible from the deallocator thread. The Dyntrace project implements the first technique for tracepoint removal.

4.5 Results

4.5.1 Tracepoint Execution Performance

The goal of this test is to compare the performance of the technique presented in this paper (using jump tracepoints) to another technique using trap tracepoints. The performance of two tools implementing the two tracepoint types were compared. For each tool, the execution of a "point" tracepoint and "entry-exit" tracepoint were timed. Also, the performance was measured for multiple thread counts [8] running the same tracepoint. For every configuration (tool, tracepoint type and thread count), the time was calculated for 50'000 execution of a simple function that took about $1\mu s$ to execute without tracing. [9] A tracepoint was placed at the first instruction of this function. The result is then divided by the number of iterations (50'000). If multiple threads were executing the tracepoint, the performance per iteration is averaged over every thread, which gives the average execution time of a traced function. For every configuration, the process was repeated 1'000 times to calculate the average and standard deviation of the execution time. To calculate the overhead of a tracepoint, the performance was also calculated for the execution of the same function with no tracepoint, for every thread count, using the same steps. The execution time without a tracepoint is then subtracted from the execution time with a tracepoint, which gives the execution overhead of the tracepoint.

The first tool is the Dyntrace tool, which was presented in Section 4.3.3. For the purpose of this experiment, the tracepoint called a no-op handler that did not do any tracing. To insert the tracepoint, the control daemon (dyntraced) was first started. Then, the in-process agent (libdyntrace-agent.so) was preloaded in the test process (using the LD_PRELOAD environment variable). Finally, the command line client (dyntrace) was used to insert a tracepoint at the first instruction of the target with the no-op tracer, which does not do any tracing.

The second tool is uprobe, which was presented in Section 4.2.2. To insert a tracepoint,

8. 1, 2, 4, 8, 16, 24, 32, 40, 48, 56 and 64 threads
9. The function calculates a modular exponentiation using an iterative algorithm

the debugfs [10] interface was used. The first step is to create the tracepoint by writing the tracepoint pseudo-file (/sys/kernel/debug/tracing/uprobe_events) and then to activate the tracepoint by writing in the enable file (/sys/kernel/debug/tracing/events/uprobes/<uprobe name>/enable). Once it is activated, the traced function can be executed. It is currently [11] not possible to execute the tracepoint without tracing, since uprobe writes into a circular buffer that can be read by the user. The overhead for uprobe has to include the overhead of writing into the trace buffer.

For both tools, the calculated overhead also includes the overhead of getting the current clock value, which was obtained using the *std*::*chrono*::*steady_clock*::*now*() function, which internally uses the *clock_gettime* system call [12]. This system call uses a seqlock [13] which can slow down the time calculation if multiple threads are getting the clock at the same time. The specification of the machine that was used for all the tests are available in Table 4.1. Figure 4.15 shows a plot of the overhead of both tools with both a "point" tracepoint and "entry-exit" tracepoint (denoted with a "ee" after the tool name) as a function of the number of threads. Table 4.2 shows the same data plus the uncertainty of every measurement. Table 4.3 shows the execution performance improvement of Dyntrace over Uprobe for each number of threads. The data is calculated using Table 4.2.

4.5.2 Memory usage

The memory usage of this technique is linearly proportional to the number of tracepoints. Every tracepoint created is independent from the others and does not require any shared data. When creating a tracepoint, two structures are allocated, plus any required by the implementation. The data structure is allocated using the normal operating system allocator and thus takes 16 bytes for x86 and 32 bytes for x86_64 (4 pointers). The execution structure uses a custom allocation algorithm that requires a very specific alignment. The structure has a variable size depending on the tracepoint type ("point" or "entry-exit"), the architecture (32-

10. usually mounted at /sys/kernel/debug
11. Linux Kernel 4.16.14
12. libstdc++ 6.0.25
13. Linux Kernel 4.16.14 file kernel/time/timekeeping.c line 875, (2)

Table 4.1 Execution Performance Experiment Machine Specifications

Processor	4x Intel Xeon E7-8867 v3 @2.50GHz
Memory	512GB
Operating System	Fedora 28 with Linux 4.16.14-300.fc28.x86_64
Compiler & Libraries	GCC 8.1.1 with GNU libc 2.27 and libstdc++ 6.0.25

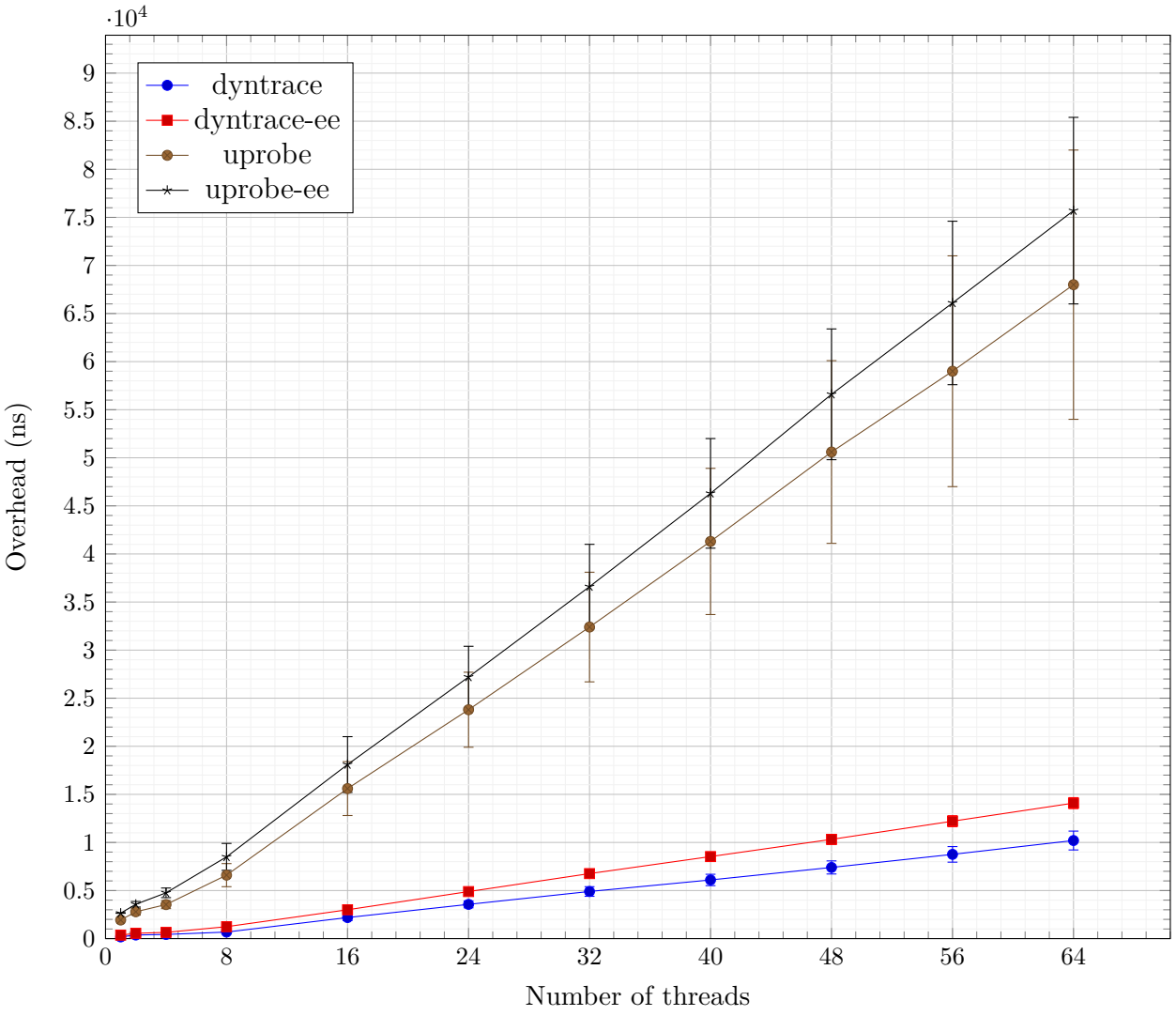


Figure 4.15 Tracepoint Execution Overhead

Table 4.2 Tracepoint Execution Overhead

	Overhead (ns) ± Error (ns)			
Threads	**Dyntrace**	**Dyntrace EE**	**Uprobe**	**Uprobe EE**
1	178 ± 6	360 ± 6	1933 ± 20	2650 ± 20
2	374 ± 44	553 ± 37	2790 ± 350	3570 ± 290
4	433 ± 52	647 ± 69	3540 ± 420	4760 ± 500
8	680 ± 140	1240 ± 140	6600 ± 1200	8500 ± 1400
16	2190 ± 312	3000 ± 220	15600 ± 2800	18100 ± 2900
24	3560 ± 360	4890 ± 310	23800 ± 3900	27200 ± 3200
32	4900 ± 500	6760 ± 350	32400 ± 5700	36600 ± 4400
40	6100 ± 590	8530 ± 430	41300 ± 7600	46300 ± 5700
48	7400 ± 680	10320 ± 520	50600 ± 9500	56600 ± 6800
56	8760 ± 810	12210 ± 570	59000 ± 12000	66100 ± 8500
64	10200 ± 980	14090 ± 570	68000 ± 14000	75700 ± 9700

Table 4.3 Dyntrace Execution Performance Improvement over Uprobe

	Improvement (%) ± Error (%)	
Threads	**Point Tracepoint**	**EE Tracepoint**
1	1086 ± 48	736 ± 18
2	750 ± 190	646 ± 96
4	820 ± 200	340 ± 160
8	979 ± 380	690 ± 190
16	710 ± 230	600 ± 140
24	670 ± 180	560 ± 100
32	660 ± 190	541 ± 94
40	680 ± 190	543 ± 95
48	680 ± 200	548 ± 94
56	670 ± 200	541 ± 95
64	670 ± 210	537 ± 91

bit or 64-bit) and the length of the replaced out of line instructions. When allocating multiple tracepoints, external fragmentation can appear between execution structures, because of the alignment requirement, which depends on the instructions being replaced.

4.5.3 Tracepoint Insertion Availability

The goal of this test is to compare the number of jump tracepoints that can be placed in a program when using Dyntrace (see Section 4.3.3) and GDB (see Section 4.2.1). Three programs were used for this test : "random" : a randomly generated no-op program, the nano editor, and the Mozilla Firefox browser. To generate the "random" program, no-op instructions between 1 and 15 bytes were generated for a total of 1000 bytes. The average length of an instruction on x86 is 3.1 bytes according to Jaikrishnan Menon and al. (40), which is the number that was used as the average random instruction length. To calculate the number of successful insertions, for each insertion location, a tracepoint was inserted, using Dyntrace or GDB, then removed. To find the address of every instruction, GNU binutils objdump was used. This utility shows the address of every instruction. Only one tracepoint is active at one time, so that no other tracepoint affects the availability of the insertion. On each instruction of every program, a tracepoint was inserted. Only the instructions present in the executable (no shared librairies) were traced. The total number of successful insertions is shown in Figure 4.4, for every combination of tracer and test program. Each row shows the result for a test program and the rightmost column the total number of insertion locations that were tested.

4.5.4 Tracepoint Insertion Time

This test measures the difference in tracepoint insertion time between Dyntrace, DynInst and Uprobe. For Dyntrace, a tracepoint was repeatedly placed at the first instruction of a test function. The creation time of this tracepoint was calculated using the Google Benchmark library (43). For DynInst, a no-op code "snippet" was inserted at the beginning of a function in the nano editor. The "snippet" insertion time was also calculated using Google Benchmark. In both cases, the instrumented function was never run. For uprobe, a tracepoint was also

Table 4.4 Fast tracepoint successful insertions

	dyntrace	**GDB**	**available insertion locations**
random	310	73	311
nano	37698	742	38092
firefox	13076	187	14004

Table 4.5 Insertion Performance Experiment Machine Specifications

Processor	Intel i7-7820X CPU @ 3.60GHz, 8 cores
Virtualization	KVM
Memory	8GB
Operating System	Fedora 28 with Linux 4.16.3-301.fc28.x86_64
Compiler & Libraries	GCC 8.1.1 with GNU libc 2.27 and libstdc++ 6.0.25

Table 4.6 Code insertion time

Tool	**Time** (μs)
Dyntrace	28.6
DynInst	9434
Uprobe	25.0

repeatedly placed at the first instruction of a test function. The *write()* system call into the *uprobe_events* pseudo-file of the Tracefs was timed to calculate the insertion time of a uprobe. Table 4.5 shows the specifications of the machine used for this test and Table 4.6 shows the insertion times for Dyntrace, DynInst and Uprobe. The goal of the test is to show the difference in insertion time when using an atomic operation versus a remote process modification using the *ptrace()* system call. Dyntrace and Uprobe insert tracepoints from a process adress space using an atomic operation. DynInst uses the *ptrace()* system call to modify the target process memory in order to insert instrumentation. The target process is stopped while the process is being instrumented.

4.6 Discussion

4.6.1 Single Thread Performance

The first point of comparison between a jump tracepoint and a trap tracepoint is the performance of the execution in a single thread. The first row of Table 4.2 shows the overhead of Dyntrace and Uprobe for a single thread. The first row of Tablle 4.3 shows the improvement of Dyntrace over Uprobe. It is clear that there is a significant overhead difference between Dyntrace and Uprobe. This can be mainly attributed to the fact that Uprobe uses traps to implement its tracepoints, which has a significant overhead compared to a jump tracepoint. Shebs (44) shows the overhead of a trap tracepoint compared to a jump tracepoint implemented in GDB. Gebai and al. survey (45) also supports this fact when discussing the different tracing mechanisms present in the Linux kernel. Also, since Uprobe places tracepoints in user mode, the cost of switching to and from kernel mode contributes to the overhead of

a trap tracepoint. For Uprobe, it is important to consider being unable to disable tracing when testing the performance of the technique. This does add some extra execution overhead, impossible to remove from the calculated overhead in this experiment. Nonetheless, it does not completely explain the overhead difference between Dyntrace and Uprobe.

The next comparison point is the performance difference between a "point" tracepoint and an "entry-exit" tracepoint. For both Uprobe and Dyntrace, a "point" tracepoint executes faster than an "entry-exit" tracepoint. This is expected since there is an additional handler to call when the traced function exits. For Dyntrace, the execution time of an "entry-exit" tracepoint is double the execution time of a "point" tracepoint ($(178 \pm 6)ns$ vs $(360 \pm 6)ns$), with the additional step of replacing the return address with the "exit" handler address. The "exit" handler is also very similar to the "point" handler, thus executing in about the same time as the "entry" handler. For Uprobe, the "entry-exit" variant takes longer to execute than the "point" variant but it does not double the time. This is explained by the need to find which Uprobe fired the trap using a binary search tree [14] every time the tracepoint is executed. It does not have to do this search when executing the "exit" handler, since the "entry" handler saves the active Uprobe for the thread [15].

4.6.2 Multi-Thread Scaling

Next, rows for threads 2 to 64 in Table 4.2 show the overhead as a function of the number of threads. Figure 4.15 shows the equivalent information in a graph. Rows for threads 2 to 64 in Table 4.3 show the improvement for every thread configuration. This figure shows a linear relation between the number of threads and the execution overhead of both tools, for both types of tracepoints. As with a single thread, Dyntrace outperforms Uprobe for every thread count. It is also clear that the performance of a "point" tracepoint is better than the performance of an "entry-exit" tracepoint, for every configuration. The scaling can be explained by the multi-thread protection mechanisms used by both tools. In the case of Dyntrace, the use of an atomic reference count to protect the removal of the tracepoint slows down the execution of a tracepoint, when multiple threads are executing the same tracepoint. The atomic increment and decrement of the reference count cause cache misses in every other thread executing the tracepoint, which slows down the execution for every additional thread running the tracepoint. Note that this can be considered a worst case since every thread executes continuously the same function simultaneously. For Uprobe, the overhead increase is explained by the use of three locks to protect the Uprobe data. The first lock protects the

14. Uprobe : :rb_node in Linux kernel 4.16.14 file kernel/events/Uprobes.c line 67 (2)
15. task_struct : :utask in Linux kernel 4.16.14 file include/linux/sched.h line 1075 (2)

Uprobe data tree and is only locked while the Uprobe instance is being found, when the trap is first executed. This lock serializes the execution of the same Uprobe but is not locked during the whole operation. This lock still affects the performance of the tracepoint execution. The second lock is a reader/writer lock that protects the list of Uprobe consumers, which uses the read lock when executing the tracepoint [16]. This lock does not prevent multiple threads from executing the Uprobe code but it does add the same overhead as an atomic increment. The third lock is a per-cpu lock that protects the per-cpu trace buffer [17] which creates the same overhead as the second lock.

4.6.3 Memory Usage

Memory usage for both Uprobe and Dyntrace can be calculated without the need for an experiment. For Dyntrace, every tracepoint created is completely independent from every other, since no data is shared between tracepoints. This means that the memory usage is linearly proportional to the number of tracepoints created. The data structure is always 4 pointers wide and the execution structure size only depends on the tracepoint being created. When creating a Uprobe tracepoint, a single structure is allocated, which contains the tracepoint data. The Uprobe also needs to allocate a page in the target process memory that contains the out-of-line replaced instruction and return handler. This means that the Uprobe tracepoint memory usage also scale linearly with the number of tracepoints created.

4.6.4 Tracepoint Placement Availability

The ability to place a tracepoint is different for Dyntrace and Uprobe. The Uprobe tracepoint placement location is not limited by the location of the tracepoint or the replaced instruction. This is due to the fact that an interrupt handler can be at any address on x86 and that the kernel can jump to any address from this handler. For jump tracepoints in general, the handler entry has to be within 4GB around the tracepoint location, since the offset of a jump is a 4 bytes signed integer offset. On x86_32, this does not impose any restriction on the location of the handler since the whole address space is addressable from the tracepoint jump. For x86_64, only a fraction of the address space is available to allocate the handler. It means that a generic handler cannot be used for every possible tracepoint placement location, for every executable and library that are loaded into the address space.

For Dyntrace and the technique presented in this paper, there is the additional restriction that some bytes in the offset have to be fixed in order to overwrite multiple instructions. This

16. Linux kernel 4.16.14 file kernel/events/Uprobes.c line 639 (2)
17. Linux kernel 4.16.14 file kernel/trace/trace_Uprobe.c line 747 (2)

effectively restricts the available address set that can be used to place the handler. Since every byte that is fixed in the offset restricts the available allocation addresses differently, and that the layout of the memory of every process is different, it is hard to predict how many tracepoints can be placed in a given process. It is clear that the more bytes are fixed in the offset, the less allocation addresses are available. It is recommended, when allocating multiple tracepoints at the same time, to allocate the tracepoints sorted by the number of bytes fixed, in descending order. The size of the executable can also affect the availability of a tracepoint insertion point. As the executable becomes bigger, the available space to allocate the execution structure becomes smaller. Indeed, the 4GB restriction around the jump point and the alignment requirement limit the number of tracepoints that can be placed in an executable.

Compared to GDB, it is clear that Dyntrace has a better insertion success for a single tracepoint (see Table 4.4). Since Dyntrace permits the insertion of tracepoints on instructions that are shorter than 5 bytes, which GDB does not, it is expected that the number of successful insertion is higher when using Dyntrace. The use of this technique is an improvement over the existing constraints present in GDB and kprobes for fast tracepoint insertion.

4.6.5 Tracepoint Insertion Time

When comparing the insertion time of Dyntrace, DynInst and Uprobe in Table 4.6, a significant difference can be seen. Dyntrace and Uprobe both use an atomic operation to insert the tracepoint into the traced process memory. In the case of Uprobe, all processes using the traced executable/library are modified. DynInst uses the *ptrace()* system call to modify the target processes memory. The use of this system call is significantely slower for multiple reasons. First, it must stop all threads of the process before executing any operation. Second, this call must be used for every word that is read or written by DynInst. Since DynInst must copy a significant amount of data to the target process, the operation is very slow. Third, this call must be used to execute any function on behalf of the traced process. This makes DynInst insertion time significantly slower than Uprobe or Dyntrace. The Uprobe insertion time is slightly better than for Dyntrace, since Uprobe does not need to find a constrained address for the tracepoint data. Since Dyntrace tracepoints have to satisfy an offset contraint, the allocation takes more time.

4.7 Conclusion

This article presented an improvement over existing jump tracepoints techniques on the x86 architecture. The main advantage of using a jump tracepoint over a trap tracepoint is performance. As discussed in the article, a jump tracepoint is faster and does not need to invoke the kernel. The main disadvantage of jump tracepoints is that they are not as flexible as a trap tracepoint. Since the relative jump instruction is 5 bytes wide, it can overwrite multiple instructions when inserting a tracepoint. This can lead to problems when executing multiple threads or if there is a jump back to one of the replaced instruction. For this reason, most implementations only permit jump tracepoints to be placed on instructions 5 bytes or longer. The improvement proposed in this article is to use a jump tracepoint with trap instructions embedded in the jump offset, which makes the jump thread and jump safe. This also means that the jump can be placed at most locations, at the cost of additional memory used. When comparing Dyntrace, an implementation of jump tracepoints, with uprobe, an implementation of trap tracepoints, it is clear that the performance of the former is superior to the performance of the latter. Compared to other techniques, the memory usage of this type of jump tracepoints is higher, since the allocation of the execution structure is prone to external fragmentation, because of the alignment requirements.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Ce mémoire à présenté plusieurs techniques d'instrumentation statique et dynamiques. Ces techniques présentaient toutes des problèmes reliés à la performance, ou à la flexibilité du placement de l'instrumentation. En effet, la plupart des techniques existantes demandaient un compromis entre performance et flexibilité. Par exemple, les outils tels que GDB et Kprobe ne permettent le placement de points de trace rapides (avec branchement) qu'à une minorité des instructions présentes. D'autres outils, tel que Uprobe, utilisent une interruption dans toutes les situations afin de placer un point de trace. Les outils tels Ftrace et Uftrace utilisent des points de trace avec branchement mais demandent une compilation spéciale afin de réserver l'espace pour l'instruction de branchement à ajouter. Les autres outils présentent d'autres inconvénients qui ne permettent pas de conserver une performance proche de celle du système sans instrumentation.

L'article présenté dans ce mémoire démontre qu'il est possible d'utiliser un point de trace avec branchement qui garde une flexibilité de placement proche de celle des points de trace avec intrruption. Afin d'exécuter un point de trace dynamique, une interruption ou un branchement peut être placé au point d'insertion. Le temps d'exécution d'un branchement est inférieur au temps d'exécution d'une interruption, ce qui rend l'utilisation de ce premier préférable. La taille de 5 octets du branchement rend difficile l'insertion de cette instruction au point d'instrumentation, puisque plusieurs instructions pourraient être remplacées. La technique présentée dans ce mémoire permet d'insérer sans encombre un branchement à n'importe quelle instruction, peu importe sa taille. Ceci est une amélioration lorsque comparé aux contraintes pour le placement de cette instrumentation dans les outils existants. En effet, GDB empêche de placer un point de trace avec branchement sur une instruction de moins de 5 octets, ce qui limite les points d'instrumentation. De plus, l'insertion ne nécessite pas d'arrêt de tous les processus afin de vérifier qu'aucun fil n'exécute une instruction à remplacer, contrairement à DynInst et Djprobe. L'insertion ne nécessite qu'une opération atomique pour x86. Ainsi, ce mémoire présente une solution complètement logicielle et sans risque à l'insertion d'un point de trace avec branchement, à n'importe quelle instruction.

5.1 Limitations de la solution proposée

L'utilisation de la technique présentée dans ce mémoire ne permet pas d'insérer un point de trace avec la même flexibilité qu'un point de trace avec interruption. En effet, il est possible que certaines combinaisont d'octets fixés et de points d'insertion ne puissent pas

être tracés. Ainsi, cette technique ne peut pas complètement remplacer l'utilisation de points de trace avec interruption mais étend l'utilisation de points de trace avec branchement. De plus, l'utilisation de mémoire des points de trace est supérieure aux techniques existantes. L'allocation des points de trace doit respecter un alignement particulier, ce qui cause une fragmentation importante de mémoire.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Le traçage est une technique très utilisée pour trouver des erreurs de programmation dans des systèmes complexes. L'insertion dynamique de points de trace est utilisée par les développeurs afin d'instrumenter des processus en exécution, sans recompilation. Sur x86, l'insertion de points de trace est complexe, puisque les instructions sont de taille variable. En effet, l'ajout d'instrumentation peut remplacer plusieurs instructions, ce qui peut amener à des problèmes dans les programmes multi-fils, à cause des branchements présents dans le programme.

Tout d'abord, le Chapitre 2 a présenté plusieurs librairies et outils de traçage avec instrumentation statique ainsi que les différentes interfaces pour créer des points de trace statiques. L'instrumentation dynamique aidée par le compilateur a ensuite été présentée. Ces techniques utilisent le compilateur pour générer des no-ops aux points d'insertion de points de trace et remplacent dynamiquement ces instructions par des sauts au code d'instrumentation, au besoin de l'utilisateur. Lorsqu'un point de trace est inactif, il n'y a aucun coût à la performance de l'exécution. Par contre, pour changer les points de trace, une recompilation est quand même nécessaire. Ensuite, l'instrumentation aidée par le matériel avec Intel et ARM a été discutée. L'utilisation de ces techniques permet de tracer un nombre limité de points d'instrumentation. Il est aussi possible de tracer un type d'évènement particulier, mais ceci a un impact important sur la performance du programme tracé. Cette technique ne requiert pas de compilation spéciale dans le programme sous trace. Les techniques logicielles de traçage dynamique ont par la suite été présentées. Ces techniques remplacent dynamiquement des instructions du processus tracé par des appels au code de traçage. Ainsi, aucune modification du programme original n'est nécessaire afin d'y ajouter du traçage. Sur x86, une instruction d'interruption d'un octet peut être insérée au point d'instrumentation. Cette instruction est lente et ne peut pas être utilisée dans les situations de haute performance. Une instruction de branchement de 5 octets peut être utilisée en remplacement. Cette instruction est plus performante mais ne peut pas être facilement placée à tous les points d'instrumentation, puisque le branchement pourrait remplacer plusieurs instructions. Les outils de traçage dynamique avec branchements doivent arrêter le processus instrumenté ou seulement permettre l'insertion d'un point de trace aux instructions d'au moins 5 octets. Finalement, les outils de modification dynamique de programmes ont été présentés. Ces outils permettent de modifier dynamiquement l'exécution d'un processus, en recompilant à la volée le code machine original ou en ajoutant un branchement vers la nouvelle version du code.

Le Chapitre 3 a présenté le problème des techniques de traçage dynamique logiciel existantes. Les outils existants proposent trois solutions afin d'ajouter dynamiquement des points de trace dans un processus : utilisation d'interruption, utilisation de branchement sur les instructions de 5 octets et plus, utilisation de branchement avec arrêt du processus durant l'insertion. Ces techniques présentent chacune des lacunes qui empêchent leur utilisation dans certaines situtations. Dans les systèmes haute performance, le coût d'une interruption est inacceptable et l'utilisation d'un branchement est nécessaire. La restriction de 5 octets pour le placement d'un branchement limite le nombre de points d'instrumentation. Un arrêt du processus peut aussi être utilisé afin de placer le branchement sans danger, ce qui est inacceptable dans une situation de haute performance. Ainsi, l'utilisateur des outils de traçage doit faire un compromis entre performance et flexibilité.

Le Chapitre 4 est un article qui présente une solution partielle aux problèmes des outils existants. Celui-ci présente un point de trace avec branchement qui peut être placé à la plupart des points d'instrumentation sans arrêt du programme. Plusieurs instructions peuvent être remplacées par ce point de trace sans problème de multi-fils et de sauts. En fixant les octets aux positions des instructions remplacées dans le décalage du branchement à la valeur d'une instruction d'interruption, l'insertion sans danger d'un branchement devient possible dans la plupart des situations. Ainsi, si le pointeur d'instruction d'un fil se trouve au milieu du branchement, l'interruption incorporée au saut sera exécutée. Le point de trace peut ensuite rediriger le pointeur vers l'instruction hors-ligne équivalente. Le même processus est répété si un branchement vise le milieu de l'instruction de branchement. Cette technique permet donc de garder la performance du branchement, avec une flexibilité proche de celle de l'interruption.

Le Chapitre 5 synthétise les avantages de l'utilisation des points de trace avec interruptions incorporées au branchement, ainsi que les limites de cette technique. En effet, le respect des octets fixés du branchement nécessite une allocation non-alignée des points de trace, ce qui amène une fragmentation importante de la mémoire. Il se peut aussi que l'allocation du point de trace soit impossible. Si aucune zone de mémoire libre ne respecte les contraintes du branchement, l'utilisation d'une autre technique est nécessaire.

6.2 Améliorations futures

Plusieurs améliorations de la technique présentée dans ce mémoire sont possibles. Tout d'abord, l'allocation des points de trace pourrait être améliorée. En effet, la technique présentée alloue le code pour chacun des points de trace. Un partage, partiel ou total, de ce code pourrait être implémenté afin d'alléger la quantité de mémoire utilisée. Un appel de fonction (*call*) remplace le branchement afin de savoir quel point de trace est invoqué.

Ensuite, plusieurs autres instructions de branchement pourraient être utilisées en lieu du branchement relatif afin d'implémenter le point de trace. Le branchement absolu indirect [1] a une longueur de 7 octets et permettrait de brancher à toute addresse de l'espace mémoire, sans restriction. Une autre option est la combinaison des instructions *push $adresse ; ret*. Si l'addresse est plus basse que 256, 3 octets sont nécessaires pour encoder le point de trace. L'utilisation des registres de segment est aussi possible afin d'étendre les adresses d'allocation de point de trace.

1. *jmp *adresse*

RÉFÉRENCES

[1] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes : 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, May 2018.

[2] L. Torvalds, "Linux kernel (v4.16.14)," 2018. Hosted on `https://elixir.bootlin.com/linux/v4.16.14/source/`.

[3] M. Desnoyers, "Using the linux kernel tracepoints," 2018. Found in the Linux kernel 4.16.14 at Documentation/trace/tracepoints.txt.

[4] Cloud Native Computing Foundation, "opentracing.io," 2018. Hosted on `opentracing.io`.

[5] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure,"

[6] Twitter, "Openzipkin a distributed tracing system," 2018.

[7] The LTTng Project, "Lttng v2.10 - lttng documentation - lttng," 2018.

[8] M. Desnoyers, P. E. McKenney, A. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," 2009.

[9] The Linux Foundation, "Babeltrace."

[10] Eclipse Foundation, "Trace compass," 2018.

[11] S. L. Graham, P. B. Kessler, and M. K McKusiclc, "gprof : a call graph execution profiler," 1982.

[12] *GNU gprof*, 2018.

[13] *Using the GNU Compiler Collection (GCC) : Gcov*, 2018.

[14] A. Veitch, D. Berris, E. Anderson, N. Heintze, and N. Wang, "Xray : A function call tracing system," 2016.

[15] K. Namhyung, "namhyung/uftrace : Function (graph) tracer for user-space," 2018. Hosted on GitHub at `https://github.com/namhyung/uftrace`.

[16] ARM, "Arm® coresight™ architecture specification v3.0," 2017.

[17] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, "Kernel probes (kprobes)," 2018. Found in the Linux kernel 4.16.14 at Documentation/kprobes.txt.

[18] M. Hiramatsu and S. Oshima, "Djprobe—kernel probing with the smallest overhead," 2007.

[19] S. Goswami, "An introduction to kprobes," 4 2005. Found at `https://lwn.net/Articles/132196/`.

[20] M. Hiramatsu, "Kprobe-based event tracing," 2018. Found in the Linux kernel 4.16.14 at Documentation/trace/kprobetrace.rst.

[21] S. Dronamraju, "Uprobe-tracer : Uprobe-based event tracing," 2018. Found in the Linux kernel 4.16.14 at Documentation/trace/uprobetracer.rst.

[22] Free Software Foundation, Inc., *Debugging with GDB*, 2018. Found at `https://sourceware.org/gdb/current/onlinedocs/gdb/`.

[23] H. Zhu, "Linux kernel gdb tracepoint module 2011-03-02," 2011. KGTP Patch for the Linux Kernel at `https://lwn.net/Articles/430666/`.

[24] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," 2004.

[25] V. Prasad, W. Cohen, F. Ch. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," 2005.

[26] F. Ch. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen, "Architecture of systemtap : a linux trace/probe tool," 2005.

[27] *General Commands Manual : strace(1)*, 2018. Retrived from the Linux man-pages.

[28] *Linux Programmer's Manual : ptrace(2)*, 2018. Retrived from the Linux man-pages.

[29] B. Gregg, "Linux extended bpf (ebpf) tracing tools," 2018. Hosted on `http://www.brendangregg.com/ebpf.html`.

[30] IO Visor Project, "iovisor/bcc : Bcc - tools for bpf-based linux io analysis, networking, monitoring and more," 2018. Hosted on Github at `https://github.com/iovisor/bcc`.

[31] B. Gregg, "perf examples," 2018. Hosted on `http://www.brendangregg.com/perf.html`.

[32] Sysdig Inc., "Sysdig | the first unified approach to container security, docker monitoring and forensics for cloud-native applications," 2018.

[33] N. Nethercote and J. Seward, "Valgrind : A program supervision framework," 2003.

[34] N. Nethercote and J. Seward, "Valgrind : A framework for heavyweight dynamic binary instrumentation," 2007.

[35] Valgrind Developers, *Valgrind User Manual*, 2017.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood, "Pin : Building customized program analysis tools with dynamic instrumentation," 2005.

[37] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," 2000.

[38] W. Williams and J. Detter, "dyninst/dyninst : Dyninstapi : Tools for binary instrumentation, analysis, and modification.," 2018. Hosted on GitHub at `https://github.com/dyninst/dyninst`.

[39] J. Arnold and M. F. Kaashoek, "Ksplice : Automatic rebootless kernel updates," 2009.

[40] E. Blem, J. Menon, and K. Sankaralingam, "A detailed analysis of contemporary arm and x86 architectures," 2013.

[41] C. Harper-Cyr and M. R. Dagenais, "charpercyr/dyntrace : Dynamic tracing in linux using fast tracepoints," 2018. Hosted on GitHub at `https://github.com/charpercyr/dyntrace`.

[42] H. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell, *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0*, 1 2018.

[43] Google, "google/benchmark : A microbenchmark support library," 2018. Hosted on GitHub at `https://github.com/google/benchmark`.

[44] S. Shebbs, "Gdb tracepoints, redux," 2009.

[45] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead," 2018.