Noyau Linux : à propos des outils de trace

Xavier de Rochefort

▶ To cite this version:

Xavier de Rochefort. Noyau Linux : à propos des outils de trace. ComPAS 2014 : conférence en parallélisme, architecture et systèmes, Apr 2014, Neuchâtel, Suisse. ⟨hal-00998017⟩

HAL Id: hal-00998017

https://hal.science/hal-00998017v1

Submitted on 2 Jun 2014

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

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

HAL Authorization

Noyau Linux : une étude des outils de trace

Xavier de Rochefort

Université de Bordeaux,
LaBRI - 351 cours de la Libération F-33405 Talence cedex - France
xavier.de-rochefort@labri.fr

Résumé
Linux est un programme complexe par nature. La compréhension de ce que le système exécute, et de comment et quand il l'exécute est nécessaire pour obtenir des métriques de performances, pour s'approprier son fonctionnement ou pour analyser du code ajouté quand on développe un module ou lorsqu'on ajoute du code au sein même du noyau. Afin d'obtenir des informations d'exécution sans être dépendant d'un support d'exécution particulier, du code est ajouté au système pour produire des traces. Leur production doit induire un surcout le plus faible possible pour réduire les risques de perturber le code observé. Un nombre important d'outils est apparu pour permettre ce type de mise en œuvre. L'article apporte un éclairage sur les points de conception associés à leur développement dans un contexte d'exécution multicœur, et propose une étude comparée de 11 outils.

Mots-clés : Linux, trace, instrumentation, étude, multicoeur

---

1. Introduction

Linux représente aujourd'hui plus de 15 millions de lignes de codes, modifiées et maintenues par plus de 1300 contributeurs. Le développement des dernières versions fait état de l'intégration de plus de 7 patchs par heure en moyenne [2, 3]. Des outils existent pour aider le développeur à faire face à cette complexité croissante. L'analyse statique de code source peut mettre en évidence certaines informations utiles. Coccinelle [9], permet par exemple d'automatiser la détection de fautes. Lawall et al. [32] et Bissyandé et al [14] offrent des solutions pour aider le développeur dans l'utilisation des API. Ce type d'analyse ne permet cependant ni de saisir toute la complexité qui peut survenir à l'exécution, ni de répondre au besoin d'informations ne pouvant être récupérées qu'à l'exécution du système. Ce besoin peut être motivé par plusieurs cas d'utilisations. Nous en distinguons principalement quatre. Le *monitoring* s'intéresse aux informations relatives à des évènements marquants du système comme une erreur ou l'insertion d'un module pour permettre un suivi de l'état du système, et en diagnostiquer les éventuels problèmes. Les messages de monitoring émis par le système ne font généralement pas référence au code source du système. Le *débogage* permet au développeur de suivre les états successifs d'une variable et le chemin d'exécution du programme. On distingue du *débogage*, le *traçage* qui s'intéresse particulièrement aux chemins et à l'ordre d'exécution d'évènements à grain fin dans le système, comme l'entrée et la sortie d'une fonction, ou un appel système. Pour finir, le *profilage*, s'intéresse à des données plus numériques, et consiste à associer des métriques à des

zones du code, ex. un temps d'exécution, un nombre de cycles processeur ou un nombre de défauts de cache.

Afin de produire des informations nécessaires à ces cas d'utilisation, du code est ajouté et exécuté lorsque le système atteint certains points de son exécution. On parle d'*instrumentation* du code système et de *points d'instrumentation* pour désigner les points d'observation. L'observation peut être intrusive, dans le cas de *points d'arrêt*, interrompant l'exécution pour permettre une interaction avec l'utilisateur, ou avoir comme objectif d'enregistrer des données en perturbant le moins possible le cours du système. Dans ce dernier cas, on parle de *génération de traces*. Cette approche permet d'analyser l'exécution du système avec le minimum de biais. C'est cette approche à laquelle nous nous intéressons dans la suite de l'article.

On peut différencier deux méthodes d'instrumentation permettant de tracer l'exécution d'un système. La première consiste à effectuer l'instrumentation depuis l'extérieur du système en servant de proxy à son exécution à la manière d'un hyperviseur. C'est ce qui est proposé, par exemple, par Bungale et al. avec PinOS [16] (2007). Cette approche a l'avantage de ne pas être dépendante du système instrumenté en traitant ce dernier comme une boîte noire. Ce point de vue externe permet d'observer l'exécution dans son ensemble, par exemple, pour détecter des fuites mémoires [35]. L'autre méthode consiste à insérer du code au sein du système. L'instrumentation s'effectue au sein du système et sa mise en œuvre lui est spécifique.

Nous nous intéressons à cette seconde méthode d'instrumentation et plus particulièrement aux outils spécifiques à Linux adaptés à son observation sur une architecture multiprocesseur [1]. Le projet Linux a vu naître un nombre de systèmes de trace important en l'espace de quelques années. Leurs évolutions rapides, leurs similitudes, et le nombre important de fonctionnalités proposées peuvent être déroutants. À notre connaissance, aucun document ne traite de ces outils à leur stade actuel de développement. Nous en proposons une étude. La suite du document se décompose comme suit : la section 2 détaille les points de conception en jeu dans ces outils et certains des mécanismes présents dans le noyau Linux pour y répondre, la section 3 présente les outils et les compare en mettant en lumière leurs avantages et inconvénients. Enfin, la section 4 synthétise et conclue l'article.

2. Conception

Un système de trace offre une solution pour (i) mettre en place des points d'instrumentation, (ii) écrire des traces en mémoire, (iii) associer une date à une trace, (iv) rendre les traces accessibles en espace utilisateur et (v) proposer de les écrire sur disque, de les afficher, ou de les soumettre à un périphérique de sortie du système (ex. port série, réseau). Aborder les outils de trace suivant ce découpage fonctionnel nous a amener à étudier cinq points de conception majeurs associés : (i) l'instrumentation, (ii) la concurrence, (iii) le temps, (iv) les compteurs de performances, et (v) la récupération des données en mémoire en cas de panne du système. Dans cet ordre, nous discutons ci-après chacun de ces points dans une section indépendante.

2.1. Instrumentation du code

L'instrumentation du système initie la procédure de génération de traces (Fig.1, (1)). L'instrumentation consiste à faire exécuter du code supplémentaire à un programme dans le but de récupérer des informations sur son exécution. On parle d'*instrumentation statique* pour qualifier une instrumentation intervenant avant l'exécution du programme, et à l'inverse d'*instrumentation dynamique* pour une instrumentation s'effectuant pendant. Un *code d'instrumentation* est un mor-

1. Nous utilisons les termes génériques de *processeur* ou *CPU* pour parler d'un processeur monocœur, d'un cœur au sein d'un processeur multicœur, ou d'un *thread* matériel au sein d'un cœur ou d'un processeur multithreadé.

ceau de code permettant de relever des données pendant l'exécution d'un programme. Un *point de trace* est un point d'instrumentation permettant de collecter des données sans interrompre l'exécution du système. Un *évènement* correspond à l'exécution d'un code d'instrumentation. L'*enregistrement d'un événement* est l'écriture en mémoire des données relevées par le code d'instrumentation. Une *trace* correspond aux données écrites lors de l'enregistrement de l'événement, c'est pourquoi nous parlerons aussi d'*écriture d'une trace* pour parler de l'enregistrement d'un évènement.

Parmi les techniques d'instrumentation utilisées, l'*instrumentation du code source* est l'instrumentation statique la plus simple. Un outil de trace utilisant cette approche met à disposition du développeur une fonction ou un ensemble de fonctions prêtes à être insérées aux emplacement du code source correspondant à ce que l'on souhaite instrumenter. L'insertion et le retrait d'un point de trace nécessitent la compilation du noyau et le redémarrage du système. Elle s'oppose à l'*instrumentation du code binaire*, impliquant la manipulation statique ou dynamique du code compilé. Une autre type d'instrumentation statique intervient pendant les phases de compilation. GCC (*GNU Compiler Collection*), par exemple, dispose d'une option de compilation permettant d'ajouter un appel de fonction au début de chaque fonction qui sont tout autant de points d'instrumentation exploitables en jouant sur l'implémentation de la fonction appelée [2]. L'instrumentation statique est adaptée dans les cas où l'on peut redémarrer le système. L'instrumentation dynamique permet d'intervenir sur un système en fonctionnement en remplaçant l'instruction à l'adresse que l'on veut tracer par une instruction qui redirige le flux d'exécution du système vers une fonction entourant l'instruction remplacée de fonctions procédant au traitement effectif de l'instrumentation (*pre-handler*, *post-handler*). Sur certaines architectures les

2. Documentation de Gprof en ligne. Quelques indications sur l'option -pg de GCC : `http://sourceware.org/binutils/docs/gprof/Implementation.html`

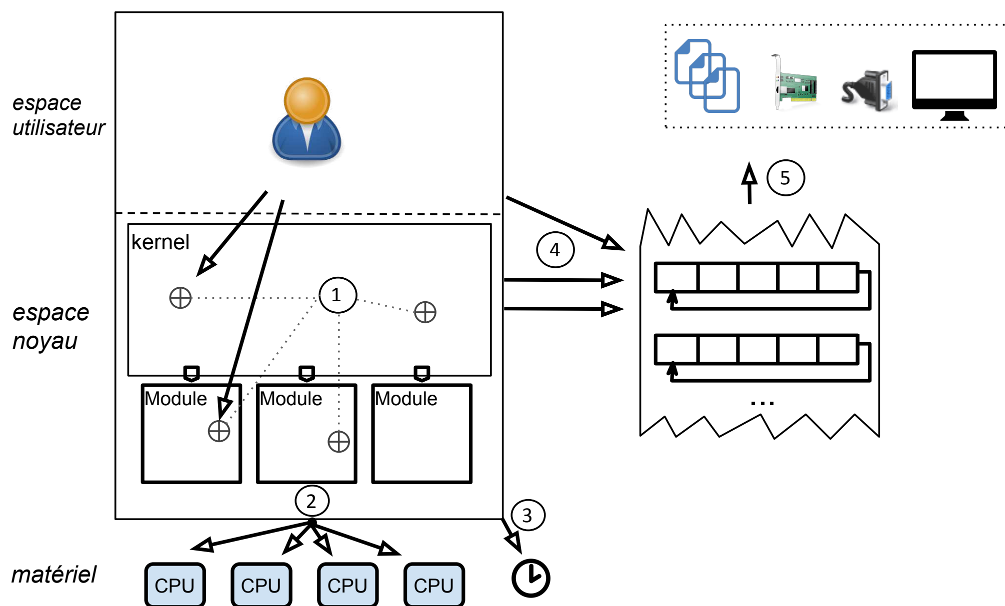

FIGURE 1 – Points de conception abordés dans la section 2. (1) Mise en place des points et du code d'instrumentation, (2) L'horodatage, (3) La lecture des PMC (*Performance Monitoring Counters*) (4) L'écriture concurrente, (5) Le *dump* des données

instructions de saut ont une portée limitée, et l'utilisation de plusieurs fonctions relais peuvent être alors nécessaires pour atteindre les *handlers*. Les fonctions permettant d'atteindre les *handlers* sont désignées par le terme de *trampoline*. Cette technique, appelée *code patching*, *probing*, ou encore *code slicing*, est proposée pour instrumenter le code d'un système d'exploitation dans les travaux de référence de Tamches et Miller [40] (1999). Les architectures à jeu d'instructions à taille fixe, telles que PowerPC ou UltraSPARC, permettent d'utiliser systématiquement une instruction de saut, n'introduisant qu'un faible surcout. Sur des architectures utilisant un jeu d'instructions à taille variable, telles que les très répandues architectures de type x86, l'instruction de saut ne peut pas remplacer toute instruction du code. L'utilisation d'une instruction d'interruption logicielle (*trap*), de taille minimale, est alors nécessaire au dépend des performances. Cette méthode est utilisée par Moore [34] (2001) pour le développement d'un outil de trace utilisant l'instrumentation dynamique sur une architecture x86, DProbes.
Linux intègre depuis sa version 2.6.9 (2004) cette partie de l'approche développée dans DProbes, KProbes [33]. Depuis Linux 2.6.38 des optimisations ont été ajoutées pour les architectures x86 afin de remplacer dans certains cas le mécanisme d'interruption par une instructions de saut, moins couteuse [30].
Ce type de transformation dynamique de code peut être utilisé pour rediriger le code du système vers n'importe quelle portion de code. Cette approche générique peut donc être utilisée vers un code contenant des erreurs pouvant compromette le fonctionnement du système. L'utilisation de cette approche s'adaptant particulièrement bien aux contraintes de machines en production, des travaux ont proposé d'utiliser des propriétés d'un langage dédié [38] ou d'un environnement d'exécution [34, 18] pour limiter les risques de corruption du système.
L'instrumentation du code source intègre un surcout, et ce même avec un système d'activation dynamique des points de trace en utilisant des branchements du type `if(trace_activated) {...}`. L'instrumentation dynamique peut être très couteuse lorsque l'utilisation d'interruption est nécessaire. Un bon compromis permettant d'éviter ces inconvénients est possible. En effet, une solution mixte utilisant des points d'instrumentation statiques activables et désactivables par transformation binaire dynamique peut être adéquate. C'est ce qui est par exemple proposé dans les travaux de Brandenburg et Anderson [15], et Desnoyer et Dagenais [23].
Les points d'instrumentation statique peuvent faire partie intégrante du code du système afin de proposer des points d'instrumentation facilement exploitables dans les zones de code qui sont susceptibles d'avoir un intérêt assez général dans l'observation du système, comme la fonction de réveil d'une tâche (*sched_wakeup*) ou de changement de tâche sur un processeur (*sched_switch*). Solaris intègre ce type de point d'instrumentation depuis l'introduction de DTrace avec l'interface SDT (*Statically Defined Tracing*) [3]. C'est aussi le cas de Linux depuis la version 2.6.32 avec les *kernel tracepoints*.

2.2. Tampon circulaire et concurrence

Un code d'instrumentation a pour but principal d'écrire des traces en mémoire (Fig.1, (4)). Afin de cloisonner les portions mémoire allouées à la récolte de données et d'éviter la fragmentation, les systèmes de traces utilisent un ou plusieurs tampon(s) circulaire(s) (*ring/circular buffer*) : une fois la portion de mémoire remplie, l'écriture reprend au début du tampon. Les traces sont directement écrites en mémoire pour éviter le surcoût d'une communication avec un périphérique pendant l'écriture des traces. Le lien entre la mémoire et les différents périphériques intervient ponctuellement, par exemple lorsqu'un tampon est rempli. L'utilisation d'un sous-tampon (*sub-buffer*) est utilisée pour enregistrer les évènements intervenant pendant

3. documentation Oracle en ligne sur l'interface SDT de DTrace : `https://wikis.oracle.com/display/DTrace/sdt+Provider`

cette période [41, 26]. Si aucune recopie des données n'est prévue vers un périphérique on parle d'enregistrement en mode boîte noire (*flight recorder*).

Un principe de la programmation sur architecture parallèle est de partager le moins de ressources possibles entre processeurs pour éviter les goulets d'étranglement inhérents aux accès concurrents à une ressource. L'utilisation d'un tampon par processeur est conséquemment un principe récurrent dans la conception des systèmes de traces.

L'entrelacement des contextes d'exécution, l'exécution du système sur des architectures multiprocesseurs, et la récupération des traces en espace utilisateur impose une synchronisation de plusieurs écrivains et, dans le cas général, d'un lecteur (*single reader multiple writers*). Le cas particulier de l'instrumentation concurrente du système par plusieurs utilisateurs ajoute la contrainte de synchronisation avec plusieurs lecteurs (*multiple readers multiple writers*). Le choix d'une synchronisation bloquante empêche l'utilisation dans les zones du code participant à la gestion des interruptions non-masquables (*NMI*). De plus, ce type de synchronisation empêche de borner le temps d'exécution des sections protégées, ce qui la rend incompatible avec l'instrumentation d'un code au temps d'exécution critique comme un traitant d'interruption, et plus largement, d'un système ayant des contraintes de temps réel. Pour permettre ce type d'instrumentation, des implémentations de tampons circulaires utilisant des algorithmes de synchronisation non bloquants (*non-blocking*, *lock-free*, *starvation-free*) ont été proposés.

Wisniewski et al. [41] utilisent un tampon par processeur, comme toutes les approches suivantes. Chaque tampon est exclusivement accédé par le processeur auquel il est attaché. Le tampon accepte des évènements à taille variable, indiquée dans une portion d'en-tête. L'accès aléatoire à une entrée est accéléré par l'insertion de bornes avec lesquels s'alignent l'écriture des évènements. La concurrence en écriture est opérée par une réservation atomique d'espace dans le tampon, en utilisant une instruction *compare-and-swap* (CAS), préalablement à toute écriture. En cas d'échec, l'instruction est à nouveau exécutée jusqu'au succès. Ce type de synchronisation est non-bloquante mais ne permet par de borner le temps exécution. On parle de synchronisation sans-verrou [29] (*lock-free*). À noter que la terminaison d'une tâche après la réservation d'espace mais avant l'écriture des logs, ainsi qu'une interruption avant la fin de l'écriture suffisamment longue pour que le tampon se remplisse et écrase l'espace réservé sont deux situations pouvant corrompre le tampon. Les accès en lecture sont facilités par un partage des tampons entre l'espace noyau et l'espace utilisateur. Cette approche performante rend possible la corruption des données par du code en espace utilisateur. Bien que possible dans un système expérimental tel que K42, n'est pas imaginable dans un système réel. Brandenburg et Anderson [15] proposent une implémentation de tampon permettant l'enregistrement d'évènements à taille variable mais uniforme, et n'admettant qu'un seul lecteur. Chaque tampon utilise un compteur de places disponibles. Un instruction atomique de type *exchange-and-add* (XADD) est utilisée pour incrémenter et décrémenter atomiquement le compteur. La réservation d'un emplacement s'effectue en décrémentant. Si la valeur est négative, le tampon est plein, la valeur du compteur est alors incrémentée pour revenir à 0. La réservation d'un emplacement s'effectue en un temps borné. L'algorithme de synchronisation est dit sans attente [29] (*wait-free*). Dans Linux, le besoin de tampons performants adaptés à la récolte de traces à donner lieu au développement de plusieurs solutions. Zanussi et al. [42] reprend le même type d'approche sans verrou que Wisniewski et al. [41]. Une alternative utilisant un verrou est intégrée pour les architectures supportée par Linux ne disposant pas d'une instruction équivalente à *compare-and-store*. Ces travaux ont donné lieu à l'intégration de l'interface *relay* (appelée originellement *relayfs*). Les travaux conjoints de Rostedt et Desnoyer [13], motivés par la volonté d'ajouter une implémentation performante et générique de tampon utilisable par tous les outils de trace du noyau, a abouti à son intégration dans le noyau 2.6.28. Les travaux associés sont décrits dans

[24] et [12].

2.3. Horodatage des évènements

L'écriture des traces en mémoire induit souvent de les horodater, afin de déduire l'ordre d'exécution exact des évènements et déterminer le temps d'exécution d'une portion de code (Fig.1, (3)). Deux repères de temps se côtoient traditionnellement au sein d'un système : l'heure réelle (*wall-clock time*) et le temps écoulé depuis le démarrage de la machine (*uptime*). Un outil de trace pour du monitoring est plus susceptible d'être intéressé par l'heure réelle, avec une précision à la seconde. Le type d'évènements considéré est à l'échelle temporelle humaine. Une solution courante est l'utilisation d'un composant de type RTC (*Real Time Clock*) alimentée par une pile et fonctionnant indépendamment du reste du système. Pour le traçage d'évènements à l'échelle du code, on préférera utiliser l'*uptime* et une précision de l'ordre de la micro, voire de la nanoseconde afin de pouvoir ordonner avec le maximum d'exactitude les évènements issus de plusieurs processeurs et avoir un indice de performance précis. Cette précision nécessite l'utilisation d'un *timestamp* de 64 bits. Cette précision est couramment obtenue en exploitant les compteurs de cycle du processeurs et les High Precision Event Timers (HPET). Un compteur de cycles est un registre processeur incrémenté à chaque cycle d'horloge. Son utilisation est très efficiente, et permet sur les architectures les plus courantes de récupérer un compteur de 64 bits conférant une résolution sans équivalent. Son utilisation pour associer un entier ou un indice de temps à chaque évènement tracé n'est cependant pas possible sur toutes les architectures. Certaines implémentations matérielles ne garantissent pas la synchronisation des compteurs de cycles entre les processeurs et leur incrémentation sans changement de fréquence. Une incrémentation stable sur une période connue est nécessaire pour pour obtenir une mesure de temps précise en divisant le nombre de cycles par la fréquence. Le changement d'état et les changements de fréquences permis par les processeurs disposant d'une interface ACPI (*Advanced Configuration & Power Interface*) n'ont pas les mêmes répercussions sur les compteurs de cycles sur toutes les architectures. Les architectures multicoeurs Intel modernes garantissent cette stabilité et une synchronisation entre les cœurs. À défaut, l'exploitation du HPET, standard visant le remplacement des PIT, permet d'obtenir un *timestamp* de 64 bits avec ce type de garantie, au détriment d'un cout plus élevé.

2.4. Compteurs de performance

Il est souhaitable de pouvoir récupérer des indices de performances matériel pour profiler un système sur une architecture donnée (Fig.1, (2)). Les processeurs modernes intègrent une unité de suivi des performances (*Performance Monitoring Unit* ou *PMU*). Les PMU permettent de récupérer des indices de performances de plus en plus nombreux et sont d'une aide précieuse dans la démarche d'optimisation d'un système. Les PMU de certains processeurs Intel et AMD permettent par exemple d'échantillonner l'exécution des instructions ne tirant pas pleinement partie du pipeline du processeur ou de la mémoire hiérarchique [25]. Lachaise et al. ont proposé l'exploitation de cette fonctionnalité pour profiler les accès mémoire sur une architecture NUMA [31]. La diversité des interfaces permettant d'exploiter ces unités complexifie la conception d'un outil de trace voulant unifier l'accès aux informations mises à disposition par les PMU. Etsion et al. [26] (2007) proposent par exemple aux développeurs de décrire les évènements dans un langage haut niveau et d'y associer le type de compteur de performance désiré. Depuis la version 2.6.31 de Linux (sept. 2009), l'interface *perf events* simplifie l'accès aux PMU. Cette interface est exploitable en espace utilisateur par l'intermédiaire d'un unique appel système (`sys_perf_event_open`). Cet aspect était jusque là intégré dans des projets externes à Linux, tels que Oprofile [10] et perfmon2 [11], sous la forme d'un patch associé à un outil en

espace utilisateur.

2.5. Récupération post-mortem des traces

Les traces sont utiles pour diagnostiquer une erreur système. Une erreur grave peut entraîner la panne du système, c.-à-d. l'arrêt de son exécution (appelée *panique noyau* dans Linux). Le problème de la récupération des données en mémoire se pose alors. Prévoir l'envoi des données vers un périphérique lors de la détection de ce type d'erreur est une première possibilité. Les systèmes disposent généralement d'un mécanisme pour persister l'intégralité de la mémoire utilisée dans un fichier lorsqu'une une erreur grave a lieu (*crash dump*). Envoyer les données vers un périphérique nécessite toutefois que certaines parties du système soient encore opérationnelles (ex. pilote du périphérique). La méthode est envisageable si l'origine de l'erreur est localisée et n'implique pas la corruption des fonctionnalités du système permettant la communication avec le périphérique. Une autre solution consiste à procéder à la récupération des données après avoir basculé sur un système sain préalablement chargé en mémoire. Kdump [36], présent dans le noyau Linux depuis la version 2.6.13, est un exemple de solution utilisant ce type d'approche. Kdump utilise une autre fonctionnalité du noyau, Kexec [37], pour amorcer un système sain depuis le système en panne sans redémarrage du matériel, évitant ainsi l'effacement de la mémoire.
La récupération des traces dans le fichier contenant le *dump* système est alors possible si les portions mémoire des tampons utilisés par l'outil de trace sont identifiables ou si leurs adresses sont connues. Cette technique est une méthode générique applicable à tous les outils utlisables sur le noyau Linux, et ne nécessite que le développement d'une solution de lecture du *dump*. Crash [6], développé au sein de RedHat, est un utilitaire d'analyse de *crash dumps* enregistrés au format ELF (*Executable and Linkable Format*), format utilisé par exemple par KDump. Ses fonctionnalités peuvent être étendues par l'intermédiaire de *plugins* pour servir d'outil de récupération à n'importe quel outil de trace utilisant un système de *dump* générant des fichiers au format ELF.

3. Étude

Nous comparons dans cette partie le fonctionnement de onze outils. Printk est la primitive historique de trace du noyau. Feather-trace [15] est l'outil de trace développé conjointement aux travaux de Calandrino et al. [17] sur le *framework* d'ordonnancement temps réel LITMUS^RT. Il n'est plus maintenu mais dispose de la maturité acquise pendant son utilisation au sein du projet LITMUS^RT et peut être intégré à un noyau récent. LTTng [8] est issu des recherches sur les outils de trace système initiées il y a plus de 14 ans au sein de l'équipe DORSAL de Montréal [4]. La motivation de ces travaux, et plus particulièrement ceux réalisés au cours de la thèse de Mathieu Desnoyer [22], est la réalisation d'un outil permettant de tracer l'intégralité du code d'un système (code noyau et code utilisateur) de manière efficiente. Le projet est maintenu par Mathieu Desnoyer, assurant un support auprès des industriels [5]. Son utilisation est possible sans modification du noyau à partir de la version 2.6.38 du noyau. DTrace est l'outil d'instrumentation d'Oracle intégré à Solaris [18] depuis 2003 (Solaris 10), et porté sur MacOS et FreeBSD. Dtrace apparait comme l'outil d'instrumentation système de référence [28] et a suscité beaucoup de discussions sur l'intégration d'un outil similaire à Linux [19]. À ce jour deux projets développent une version de DTrace pour Linux. Le premier est intégré dans la version entreprise du noyau Linux 3.8 d'Oracle (Unbreakable Enterprise Kernel Release 3).

4. site de l'équipe : `www.dorsal.polymtl.ca`
5. Site de EfficiOS : www.efficios.com

Le second est un projet indépendant à un stade précoce de développement [5]. En réponse à DTrace, RedHat a initié dès 2005 le développement de SystemTap. La version 1.0 est sortie en 2009. Ktap [7] est une alternative à SystemTap et DTrace, initiée récemment par un développeur de la société *Huawei*. Malgré la jeunesse du projet (la version 0.1 date de mai 2013, et la version actuelle, 0.4, du 8 décembre dernier), il semble pouvoir gagner sa place parmi le panel des outils de traces Linux [21]. Ftrace était originellement développé au sein de la branche temps réel de Linux en tant que traceur de fonction afin d'analyser les latences du noyau. L'outil tient sa place dans la branche principale du projet Linux depuis la version 2.6.28. KGDB est dans le noyau depuis la version 2.6.35. KGDB est une implémentation de la partie serveur de GDB (gdb *stub*) permettant son instrumentation à l'aide de la traditionnelle partie cliente du débogueur depuis une machine connectée par le port série ou une connexion réseau. La partie cliente a besoin des informations de débogage du noyau instrumenté. De la même manière, KDB permet une instrumentation dynamique du noyau par l'intermédiaire d'un interpréteur de commandes depuis l'espace utilisateur. Les deux outils ont été Les deux outils disposent de fonctionnalités pour tracer l'exécution du noyau. L'ajout de KDB dans le noyau 2.6.35 s'est accompagné d'une mise en commun des deux outils. KGTP [6] est une alternative à KGDB/KDB exploitant KProbes. Le projet est une implémentation partielle de la partie serveur de GDB se focalisant sur la production de traces.

3.1. Instrumentation du code source

Les outils proposant une instrumentation du code source sont printk, feather-trace et Ftrace. Ils permettent tous les trois un formatage des données collectées sans limitation dans leur nombre. Les autres outils permettent de se connecter aux tracepoints du noyau Linux présentés en section 2.1. Printk a pour objectif de fournir une fonction de trace utilisable dans toutes les portions de code où des messages de monitoring doivent être ajoutés. Les messages sont émis avec un niveau de priorité parmi 8 à dispositions (du moins critique, `KERN_DEBUB`, au plus critique `KERN_EMERG` [7]). Si le niveau de priorité est supérieur à un niveau palier configurable, les messages sont écris sur la console du système. Les traces sont écrites dans un unique tampon dont les accès sont synchronisés par des verrous à attente active (*spinlock*) et par la désactivation des interruptions lors d'une lecture ou d'une écriture. Printk est aussi utilisée à des fins de débogage. La version 2.6.39 du noyau a introduit de nouvelles macros ajoutant une interface d'activation et de désactivation dynamique de chacun des points de trace [20]. L'utilisation d'un unique tampon circulaire n'en font pas le candidat idéal pour collecter des traces sur une architecture multiprocesseur. De plus, l'utilisation de verrous pour sérialiser l'accès au tampon et à la console ainsi que la désactivation des interruptions peuvent être des points de conception rendant son utilisation incompatible avec certaines parties du code noyau, comme les handlers d'interruptions non-masquables ou le code de l'ordonnanceur [1]. Feather-trace ne permet de récolter qu'un nombre limité de données du fait de sa conception utilisant un tampon circulaire à emplacements à taille variable, mais unique pour un tampon donné. Les points de trace sont insérés par une collection de macro, chacune dédiée à la collecte d'un nombre de données précis. Sa conception et ses performances permettent de l'utiliser dans n'importe quelle partie du noyau. Ftrace met à disposition deux fonctions pour instrumenter le code source : `trace_printk`, s'utilisant de manière similaire à printk sans la logique de priorité, et `trace_put` pour l'enregistrement de données brutes, sans formatage. L'écriture des traces exploite l'implémentation de tampon circulaire générique de Linux. L'ajout de points de trace

6. Dépot du projet KGPT : `https://code.google.com/p/kgtp/`
7. Niveaux de criticité des messages de printk `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/linux/printk.h`

peut s'effectuer dans toutes les zones du code du noyau. Les tracepoints permettent l'ajout de traces par une macro un peu plus complexe [39, 4]. Leur conception n'a pas pour objectif de faciliter l'ajout rapide de traces dans le noyau, mais de définir des points d'instrumentation dynamiquement exploitables faisant partie intégrante de Linux. L'ajout d'un tracepoint permet son utilisation par LTTng, Ftrace, SystemTap, Perf, et Ktap.

3.2. Instrumentation dynamique

LTTng, Ftrace, Perf, Dtrace, SystemTap, Ktap, KGDB, et KDB permettent tous d'interagir dynamiquement avec des points d'instrumentation statiques, et dynamiques. Les différences tiennent dans la flexibilité et la facilité avec laquelle ils permettent d'exploiter les points d'instrumentation et les données à y collecter. LTTng, Ftrace et Perf proposent tous les trois une interface exploitant en direct les fonctionnalités de KProbes. Un point d'instrumentation dynamique est ajouté en indiquant une adresse, ou en se servant des références symboliques du code. Les données à récolter sont définies de la même manière. Perf permet d'y d'ajouter des données de profilage. Ftrace permet de tracer les entrées et sorties de fonction de la quasi totalité du noyau de manière plus performante que Kprobes. L'outil exploite les appels de fonction générés par l'option *-pg* de GCC en enregistrant les emplacement des appels à la fonction *mcount* pendant la compilation du noyau [8]. L'enregistrement de ces emplacements est ensuite utilisé pour remplacer dynamiquement l'appel de fonction par une instruction *No Operation* (NOP) pendant l'amorçage du système. L'activation d'un point de trace se fait à l'inverse par substitution de l'instruction NOP par un appel à la fonction `mcount`. DTrace, SystemTap et Ktap fournissent un langage spécifique au domaine (*DSL*) pour faciliter la définition les évènements et des traitements associés. DTrace et Ktap s'appuient sur une machine virtuelle évoluant en espace noyau gérant l'exécution du code intermédiaire résultant de la compilation du langage. Les évènements et les données du systèmes qu'ils peuvent exploiter sont ceux prévus par l'environnement d'exécution. La sureté d'exécution du code ajouté au système est garanti par l'environnement. Dtrace permet d'ajouter des prédicats pour limiter le déclenchement d'un évènement, de définir des variables globales associé à une fonction d'agrégation. SystemTap en revanche génère du code natif utilisant sous forme de module noyau. Le langage permet de manipuler directement les noms de variables et les noms de fonctions du noyau. Les données de débogage sont utilisées pour faire le lien entre les noms symboliques dans le script d'instrumentation et leurs adresses. SystemTap permet d'utiliser les points d'instrumentations définis par tracepoint et ceux définissables dynamiquement avec Kprobes. SystemTap garantit une certaine sécurité d'exécution du code exécuté. Le langage ne permet pas d'allocation dynamique, et les conversions de type y sont limités ainsi que les opérations sur les pointeurs. Des vérifications systématiques sont ajoutées au code généré pour vérifier l'absence de division par zéro, de boucles infinies et d'accès mémoires indésirables. Un mode avancé est toutefois utilisable pour intégrer du code en C à l'intérieur du script, échappant à toute vérification. KGDB, KDB, et KGTP se démarquent par la possibilité de définir l'instrumentation dynamique par l'intermédiaire d'une console. Sur le même principe que DTrace et Ktap, l'instrumentation est décrite en un langage de script. Le code intermédiaire est transformé en code natif pour améliorer les performances. Enfin, la description de traitements périodiques ne se retrouve que dans Perf, et les trois outils Dtrace, SystemTap et Ktap permettent de définir des actions à intervalle régulier.

8. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/scripts/recordmcount.pl`

TABLE 1 – Comparatif des neufs outils de l'étude. (m = monitoring, d = débogage, p = profilage, t = trace, TP = tracepoint)

Outil	Vocation	Maturité	Statut	Statique	Instr. dyn.	Post-mortem
Printk	m,d	***	Linux	fonction	non	crash
Feather-trace	d,t	**	off	macro	non	non
SystemTap	d,t,p	***	dev	TP	oui	oui
Ktap	d,t,p	*	dev	TP	oui	non
DTrace	d,t,p	*** / *	dev	SDT	oui	SMD
LTTng	m,d,t,p	**	Linux	TP	oui	-
Ftrace	t,p	**	Linux	fonction, TP, mcount	oui	crash
Perf	t,p	**	Linux	TP	oui	-
KGDB/KDB	d, t	***	Linux	-	oui	-
KGPT	d, t	**	dev	TP	oui	crash

3.3. Bilan

Les outils présentés permettant l'instrumentation du code source se limitent actuellement à l'écriture formatées de données. Trois outils utilisant cette approche permettent de tracer l'intégralité du noyau à des fins de débogage et de traçage : Printk, Ftrace et Feather-trace. Cette approche est sure, ne limite pas les données récoltées. De plus le code d'instrumentation est ainsi un code natif optimisé. Cependant il n'est pas toujours possible de passer par une compilation du noyau et un redémarrage de la machine comme c'est par exemple le cas sur une machine de production. Parmi les 11 solutions présentées, 10 permettent une instrumentation dynamique : LTTng, Ftrace, Perf, SystemTap, Ktap, Dtrace, KGDB, KDB et KGPT. SystemTap, DTrace et KTap mettent à disposition un langage facilitant la création de points d'instrumentation et l'insertion de code d'instrumentation. SystemTap génère un module noyau. Le code d'instrumentation est donc un code natif optimisé. DTrace et Ktap compilent et interprètent le code d'instrumentation sous forme de code intermédiaire pour contrôler entièrement son exécution. DTrace est un projet mur mais uniquement disponible sur le noyau Linux adapté par Oracle. Un portage indépendant est dans un stade précoce de développement. Ktap est une alternative légère dans ces premiers stades de développement. LTTng, Ftrace et Perf utilisent tous les 3 les tracepoints et l'interface kprobes de Linux. LTTng vise une instrumentation globale et performante du système. LTTng et Ftrace sont des interfaces simples à kprobes et ne permettent pas de lui adjoindre du code utilisateur. Ftrace exploite une fonctionnalité singulière d'instrumentation statique de GCC pour tracer de manière très performante les entrées et sorties de fonction du noyau. Perf permet de profiler le noyau en permettant d'associer aux évènements émis par les tracepoints ou kprobes la récupération et l'aggrégation de métriques, comme les compteurs de performances matériels. Enfin, KGDB, KDB et KGPT sont des solutions d'instrumentation à interface console permettant d'ajouter des points d'instrumentation et du code d'instrumentation interprété et compilé en code natif à l'exécution. Ils permettent tous les trois d'instrumenter le noyau depuis une autre machine. Nous synthétisons ces différents points

TABLE 2 – Comparatif instrumentation dynamique

Outil	Timers	Code dyn.	Sureté	Console	Distant
LTTng	non	non	-	non	oui
Ftrace	non	non	-	non	non
Perf	oui	non	-	non	non
SystemTap	oui	oui	langage et vérif. à l'exec.	non	non
Ktap	oui	oui	code managé	non	non
DTrace	oui	oui	code managé	non	oui
KGDB/KDB/KGPT	non	oui	oui	oui	oui

dans la table 1. Les fonctionnalités spécifiques aux outils d'instrumentation dynamique sont présentées en table 2.

4. Conclusion

Linux est un système complexe, développé avec des objectifs de performances sur des architectures variées. Ces architectures tendent majoritairement à proposer un parallélisme MIMD complexe à exploiter et offrant une place de choix aux outils de traces, permettant d'observer le comportement du système avec un minimum de surcout. Nous avons introduit le domaine et présenté cinq points de conception principaux d'un outil de trace de code noyau, l'instrumentation, la concurrence, le temps, les compteurs de performances, et la récupération des données en mémoire en cas de panne du système. Nous avons choisi de présenter onze outils permettant de collecter des traces dans le noyau Linux. : Printk, Feather-trace, LTTng, Ftrace, Perf, SystemTap, Ktap, Dtrace, KGDB, KDB et KGPT. L'instrumentation du code source proposée par Printk, Feather-trace et Ftrace apportent une solution de formatage et d'écriture simplifiées de trace lors du développement. Les approches performantes et peu intrusives de feather-trace et Ftrace pourraient être utilisées dans le cadre de développement d'outils d'analyse statique accompagnant le développeur dans le placement des points de trace. Diagnosys [14] par exemple a été développé pour des machines monocœur, mais la conception du système de traces intégré dans l'outil limite son utilisation sur des architectures multicœur. Une adaptation pourrait être envisagée dans des travaux futurs. Les outils d'instrumentation dynamique présentés utilisent tous la technique de trampoline de Tamches et al. [40]. Cette technique est trop intrusive pour être utilisée dans du code critique. Une technique alternative exploitant des méthodes efficaces de traduction binaire dynamique a été proposée par Feiner et al. [27] et pourrait être envisagée pour étendre l'utilisation de ces outils.

Bibliographie

1. Archive lkml. discussions autour d'utilisation de printk dans le code de l'ordonnanceur de tâches.
2. Article du jdn sur le projet linux. [En ligne ; accédé le 10-jan-2013].
3. Article du site pingdom.com "linux kernel development numbers". [En ligne ; accédé le 10-jan-2013].

4. Documentation linux sur les tracepoints.
5. Dépot du projet dtrace4linux, port linux de dtrace. [En ligne ; accédé le 20-jan-2013].
6. Page web de l'utilitaire crash.
7. Page web du projet ktap.
8. Page web du projet lttng.
9. Site web du projet coccinelle. [En ligne ; accédé le 3-fev-2013].
10. Site web du projet oprofile. [En ligne ; accédé le 5-fev-2013].
11. Site web du projet perform. [En ligne ; accédé le 5-fev-2013].
12. Description du brevet de tampon circulaire déposé par s. rostedt, 2009.
13. Proposition de rfc pour un tampon circulaire générique dans linux, May 2010.
14. Bissyandé (T. F.), Réveillère (L.), Lawall (J. L.) et Muller (G.). – Diagnosys : Automatic generation of a debugging interface to the linux kernel. *In : Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. pp. 60–69. – New York, NY, USA, 2012.
15. Bjorn B. Brandenburg (J. H. A.). – Feather-trace : A light-weight event tracing toolkit. *In : In Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications (OSPERT'07*, pp. 61–70.
16. Bungale (P. P.) et Luk (C.-K.). – Pinos : A programmable framework for whole-system dynamic instrumentation. *In : Proceedings of the 3rd International Conference on Virtual Execution Environments*. pp. 137–147. – New York, NY, USA, 2007.
17. Calandrino (J. M.), Leontyev (H.), Block (A.), Devi (U. C.) et Anderson (J. H.). – Litmusrt : A testbed for empirically comparing real-time multiprocessor schedulers. *In : Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, pp. 111–126.
18. Cantrill (B. M.), Shapiro (M. W.) et Leventhal (A. H.). – Dynamic instrumentation of production systems. *In : Proceedings of the Annual Conference on USENIX Annual Technical Conference*. pp. 2–2. – Berkeley, CA, USA, 2004.
19. Corbet (J.). – Article lwn : "on dtrace envy, Aou 2007. [En ligne ; accédé le 6-fev-2013].
20. Corbet (J.). – The dynamic debugging interface. *LWN*, March 2011.
21. Corbet (J.). – Ktap almost gets into 3.13. *LWN*, Nov 2013.
22. Desnoyers (M.). – *Low-impact operating system tracing*. – Thèse de PhD, École Polytechnique de Montréal, 2009.
23. Desnoyers (M.) et Dagenais (M.). – LTTng : tracing across execution layers, from the hypervisor to user-space. *In : Linux Symposium*.
24. Desnoyers (M.) et Dagenais (M. R.). – Lockless multi-core high-throughput buffering scheme for kernel tracing. *SIGOPS OSR*, vol. 46, n3, décembre 2012, p. 65–81.
25. Drongowski (P. J.) et Center (B. D.). – Instruction-based sampling : A new performance analysis technique for amd family 10h processors. 2007.
26. Etsion (Y.), Tsafrir (D.), Kirkpatrick (S.) et Feitelson (D. G.). – Fine grained kernel logging with klogger : Experience and insights. *In : Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. pp. 259–272. – New York, NY, USA, 2007.
27. Feiner (P.), Brown (A. D.) et Goel (A.). – Comprehensive kernel instrumentation via dynamic binary translation. *In : ACM SIGARCH Computer Architecture News*. ACM, pp. 135–146.
28. Gregg (B.) et Mauro (J.). – *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. – Prentice Hall Professional, 2011.
29. Herlihy (M.), Luchangco (V.) et Moir (M.). – Obstruction-free synchronization : Double-ended queues as an example. *In : Proceedings of the 23rd International Conference on Distributed Computing Systems*. pp. 522–. – Washington, DC, USA, 2003.
30. Hiramatsu (M.) et Oshima (S.). – Djprobe—kernel probing with the smallest overhead. *In :*

Linux Symposium, p. 189.
31. Lachaize (R.), Lepers (B.) et Quéma (V.). – Memprof : A memory profiler for numa multicore systems. *In : Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. pp. 5–5. – Berkeley, CA, USA, 2012.
32. Lawall (J.), Brunel (J.), Palix (N.), Hansen (R.), Stuart (H.) et Muller (G.). – Wysiwib : A declarative approach to finding api protocols and bugs in linux code. *In : Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pp. 43–52.
33. Mavinakayanahalli (A.), Panchamukhi (P.), Keniston (J.), Keshavamurthy (A.) et Hiramatsu (M.). – Probing the guts of kprobes. *In : Linux Symposium*.
34. Moore (R. J.). – A universal dynamic trace for linux and other operating systems. *In : Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference*. p. 297–308. – Berkeley, CA, USA, 2001.
35. Nethercote (N.) et Seward (J.). – Valgrind : A program supervision framework. *Electronic notes in theoretical computer science*, vol. 89, n2, 2003, pp. 44–66.
36. NetworkX (L.). – Kdump, a kexec-based kernel crash dumping mechanism. *In : Linux Symposium*, p. 169.
37. Pfiffer (A.). – Reducing system reboot time with kexec.
38. Prasad (V.), Cohen (W.), Eigler (F.), Hunt (M.), Keniston (J.) et Chen (B.). – Locating system problems using dynamic instrumentation. *In : Linux Symposium*, pp. 49–64.
39. Rostedt (S.). – Using the trace_event() macro. *LWN*, Mar 2010.
40. Tamches (A.) et Miller (B. P.). – Fine-grained dynamic instrumentation of commodity operating system kernels. *In : Proceedings of the Third Symposium on Operating Systems Design and Implementation*. pp. 117–130. – Berkeley, CA, USA, 1999.
41. Wisniewski (R. W.) et Rosenburg (B.). – Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. *In : Supercomputing, 2003 ACM/IEEE Conference*, p. 3–3.
42. Zanussi (T.), Yaghmour (K.), Wisniewski (R.), Moore (R.) et Dagenais (M.). – relayfs : An efficient unified approach for transmitting data from kernel to user space. *In : Linux Symposium*, p. 494.