UNIVERSITÉ DE MONTRÉAL

TRAÇAGE LOGICIEL ASSISTÉ PAR MATÉRIEL

ADRIEN VERGÉ
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE LOGICIEL ASSISTÉ PAR MATÉRIEL

présenté par : VERGÉ Adrien
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
Mme NICOLESCU Gabriela, Doct., membre

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche, Michel Dagenais, pour le parc à vélos qu'il a fait installer devant l'École et qui m'a permis de venir chaque jour 15 minutes plus tôt et avec bonne humeur. Ses conseils, sa disponibilité et son engouement pour le partage des connaissances font du laboratoire DORSAL un environnement de travail convivial et stimulant.

J'aimerais aussi adresser des remerciements à tous mes collègues du laboratoire, tout d'abord pour la très bonne ambiance qui régnait au sein de l'équipe, mais aussi pour le soutien qu'ils apportaient toujours avec le sourire. En particulier, mes travaux ont pu être menés grâce à l'aide de François, Francis, Raphaël, Simon, Suchakra et Yannick, qui ont guidé mes réflexions et fait de ma maîtrise une expérience enrichissante.

Je souhaite aussi remercier nos partenaires Ericsson, EfficiOS ainsi que le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG), pour le soutien financier qui a rendu possible ma maîtrise.

Je remercie enfin chaleureusement tous mes proches, amis et rencontres qui ont fait de mon séjour au Québec une expérience inoubliable. Finalement, merci à ma « blonde » qui pensait bien faire en corrigeant mes fautes d'orthographe et qui finalement m'a donné plus de travail.

RÉSUMÉ

Les logiciels deviennent de plus en plus complexes. Avec l'avènement de l'informatique embarquée, la limitation des ressources les contraint à s'exécuter en économisant le temps, la mémoire et l'énergie. Dans ce contexte, les développeurs ont besoin d'outils pour déboguer et optimiser les programmes qu'ils écrivent. Parmi ces outils, le traçage est une solution particulièrement adaptée qui enregistre l'occurrence d'événements, en interagissant peu avec l'exécution. Elle permet de mettre en évidence les causes de bogues ou les goulots d'étranglement qui ralentissent le programme. LTTng est un traceur focalisé sur les performances : grâce à des structures de données propres à chaque cœur et à des verrous non-bloquants, l'enregistrement d'un événement prend moins d'une microseconde sur une machine récente. Ce délai est toutefois non-négligeable, il empêche de tracer un nombre arbitraire de points sans affecter les performances. De plus, le code et les données liées au traçage sont stockés dans l'espace mémoire du processus étudié, ce qui cause un impact sur son exécution.

L'utilisation de blocs matériels dédiés au débogage pallie à ces limitations. Il existe une multitude de ces circuits, présents sur la plupart des processeurs du marché, à des fins de débogage et de profilage. En réutilisant leurs capacités à des fins de traçage, nous proposons de soulager la partie logicielle d'outils comme LTTng, et ainsi d'accroître leurs performances. Pour ce faire, nous utilisons les modules matériels STM, ETM et ETB de la suite CoreSight sur les processeurs ARM, ainsi que BTS sur les processeurs x86 d'Intel. Certains offrent une fonctionnalité de traçage d'exécution, c'est-à-dire d'enregistrement de la liste des instructions exécutées ; d'autres fournissent des ressources spécialisées pour l'estampillage de temps, l'envoi de messages sur des canaux dédiés, et le stockage de traces.

Dans ce mémoire, nous proposons des implémentations de traçage logiciel s'aidant du matériel pour être moins intrusifs que les outils purement logiciels. Nous visons à réduire le surcoût en temps engendré par le traçage, c'est-à-dire le nombre de cycles ajoutés à une exécution normale, tout en gardant le même détail d'information que fournit une trace. Nous montrons que l'utilisation conjointe des modules STM et ETB pour faire transiter les traces par des circuits matériels dédiés économise la mémoire du processus et que la durée des points de trace est divisée par dix par rapport à LTTng. En utilisant ETM et ETB, le surcoût du traçage est lui aussi réduit : entre -30% et -50% par rapport à notre traceur de référence. En revanche, les capacités du traceur d'exécution ETM limitent notre système à seulement quelques points de trace enregistrables dans tout le programme. Finalement, l'utilisation de BTS sur les processeurs Intel est aussi plus efficace : les points de trace sont presque deux fois plus rapides que ceux de LTTng. Cependant, ce système ne permet pas de

choisir quels événements tracer : tous les branchements pris par le processeur sont enregistrés. Cette lourdeur rend BTS inutilisable pour faire du traçage d'événements ; néanmoins pour du traçage d'exécution, la ré-implémentation que nous proposons est 65% plus rapide que celle de Perf, l'outil par défaut sous Linux.

ABSTRACT

Software is becoming increasingly complex. With the advent of embedded computing, resource limitations force it to run in a way saving time, memory and energy. In this context, developers need tools to debug and optimize the programs they write. Among these tools, tracing is a particularly well suited solution that records the occurrence of events, while minimally interacting with the execution. It allows to identify the causes of bugs or bottlenecks that slow down the program. LTTng is a tracer focused on performance: through per-core data structures and non-blocking locks, recording an event takes less than one microsecond on a typical computer. However, this delay is not negligible, and tracing an arbitrary number of points is not possible without affecting performance. In addition, the code and data related to tracing are stored in the memory space of the process being studied, causing an impact on its execution.

The use of dedicated debug hardware blocks overcomes these limitations. There are a multitude of these circuits, present on most processors on the market, for of debugging and profiling purposes. By reusing their capacity for tracing purposes, we propose to alleviate the software part of tracing tools such as LTTng, and thereby increase their performance. To do this, we use STM, ETM and ETB hardware modules from the CoreSight suite on ARM processors, as well as BTS on Intel x86 processors. Some offer an execution tracing feature, i.e. recording the list of executed instructions; others provide specialized resources for timestamping, transfering messages on dedicated channels, and storing traces.

In this thesis, we propose implementations of software tracing that take advantage of hardware to be less intrusive than pure-software tools. We aim to reduce the time overhead induced by tracing, i.e. the number of cycles added to a normal execution, while keeping the same detailed information as a trace provides. We show that the combined use of STM and ETB modules to send traces through dedicated hardware circuits saves process memory and that each tracepoint duration is divided by ten as compared to LTTng. Using ETM and ETB, the overhead of tracing is also reduced: between -30% and -50% as compared to our reference tracer. However, the capacity of the ETM execution tracer limits our system to only a few recordable tracepoints throughout the program. Finally, the use of BTS on Intel processors is also more efficient: tracepoints are almost two times faster than LTTng. However, it is not possible to choose which events to trace with this system: all branches taken by the processor are stored. This limitation makes BTS unusable for event tracing; however, for execution tracing the re-implementation we offer is 65% faster than Perf, the default tool on Linux.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ABI	Application Binary Interface
AHB	Advanced High-performance Bus
AXI	Advanced eXtensible Interface
BTS	Branch Trace Store
CPU	Central Processing Unit
CR3	Control Register number 3 (processeurs x86)
CTF	Common Trace Format
DDR	Double Data Rate
DebugFS	Debug File System
DNS	Domain Name System
DORSAL	Distributed Open Reliable Systems Analysis Lab
ETB	Embedded Trace Buffer
ETM	Embedded Trace Macrocell
FIFO	First In, First Out
HTM	AHB Trace Macrocell
ITM	Instrumentation Trace Macrocell
JTAG	Joint Test Action Group
LBR	Last Branch Record
LKML	Linux Kernel Mailing List
LPDDR	Low Power Double Data Rate
LTT	Linux Trace Toolkit
LTTng	Linux Trace Toolkit next generation
MD5	Somme de contrôle Message Digest 5
MIPI	Mobile Industry Processor Interface
MSR	Model-Specific Register
NFS	Network File System
NSERC	Natural Sciences and Engineering Research Council of Canada
OMAP	Open Multimedia Applications Platform
OS	Operating System
PID	Process IDentifier
PMU	Performance Monitoring Unit
PT	Processor Trace
PTM	Program Trace Macrocell

PXE	Pre-boot eXecution Environment
RAM	Random-Access Memory
RCU	Read-Copy Update
SDRAM	Synchronous Dynamic Random Access Memory
SHA256	Somme de contrôle Secure Hash Algorithm à 256 bits
SSH	Secure Shell
STM	System Trace Module ou System Trace Macrocell
STP	System Trace Protocol
SWIT	Software Instrumentation Trace
SysFS	Système de fichiers virtuel pour le contrôle du système
TFTP	Trivial File Transfer Protocol
TI	Texas Instruments
TMC	Trace Memory Controller
TMF	Tracing and Monitoring Framework
VDSO	Virtual Dynamically linked Shared Objects

CHAPITRE 1

INTRODUCTION

Dans les domaines nécessitant des logiciels optimisés pour s'exécuter le plus rapidement possible et consommer moins de ressources (serveurs à haute disponibilité, centres de calcul, applications embarquées), le traçage est une solution efficace pour améliorer le logiciel. Ce procédé d'enregistrement d'événements à certains points d'intérêt du système d'exploitation et du logiciel permet d'avoir des renseignements très précis sur les conditions de l'exécution. Ils mettent en évidence les problèmes à bas niveau tels qu'une mauvaise gestion de la mémoire, mais aussi ceux à plus haut niveau tels que l'accès concurrent à une ressource partagée. Les traces sont alors très utiles pour aider les développeurs à corriger des erreurs et accélérer leurs applications.

Les outils de traçage d'événements sont optimisés pour influer le moins possible sur les logiciels tracés. Pourtant, ils interfèrent quand même avec l'exécution car l'enregistrement des points de trace nécessite au minimum l'exécution de code supplémentaire et la réservation de mémoire. L'utilisation de ressources matérielles dédiées est une solution pour contourner ces coûts. La plupart des processeurs récents embarquent des circuits spécialisés dans le débogage, dont l'usage peut être adapté ou détourné afin d'assister les traceurs logiciels dans leur tâche.

1.1 Définitions et concepts de base

Dans cette section, nous introduisons différents concepts qui seront utilisés tout au long de ce mémoire.

Événements et points de trace

Un **événement** est un point précis de l'exécution d'un programme, auquel sont associées une date et la production d'une action. La date fixe l'occurrence de l'action dans le temps, et est généralement représentée sous la forme d'une estampille de temps (*timestamp*). L'action associée, quant à elle, précise le type d'événement enregistré. Cela peut être un appel système, une interruption, ou l'appel d'une fonction spécifique. Lors de leur enregistrement, les événements peuvent être accompagnés de valeurs donnant des informations sur l'état du programme à ce moment précis. Il peut s'agir de l'identifiant du processus, du nombre de cycles écoulés, etc. Ces données additionnelles sont appelées « contexte ».

Dans la pratique, les événements sont enregistrés par des **points de trace**, caractérisés

par des emplacements précis dans le code du programme ou du système d'exploitation. L'atteinte d'un tel emplacement provoque l'enregistrement de l'événement correspondant dans une trace.

Trace

Il faut distinguer deux concepts caractérisés par la même appellation :

1. Une **trace d'événements** rassemble une succession d'événements (tels que définis dans le paragraphe précédent). Elle enregistre les points marquants de la vie d'un programme ou d'un système pour permettre son étude ultérieure. Contrairement à l'écriture de fichiers journaux (*logs*), le traçage d'événements se veut très léger, pour ne pas interférer avec l'exécution normale du programme. Alors que la journalisation induit de fortes latences (notamment à cause du coût de l'accès disque), l'enregistrement de traces affecte beaucoup moins les performances grâce à un format compact et des mécanismes optimisés. Le traçage permet ainsi l'enregistrement de plusieurs milliers d'événements par seconde.
2. Une **trace d'exécution** liste les adresses de toutes les instructions exécutées par le processeur. Il s'agit d'une information exhaustive, beaucoup plus complète qu'une trace d'événements : on peut retracer la vie d'un programme dans sa totalité. Enregistrer un tel volume de données a néanmoins un coût important en termes de dégradation de performance et de quantité de trace à stocker, ce qui rend le traçage d'exécution en continu inapplicable en production. Toutefois, ce procédé peut être utilisé par intermittence pour tracer des portions spécifiques de code.

La figure 1.1 illustre la différence entre traçage d'exécution et traçage d'événements. Elle montre que pour une même exécution, les deux types de trace ne fournissent pas les mêmes données.

Notre recherche se concentre sur le **traçage d'événements**. Pour cette raison, dans la suite de ce mémoire, les termes « traçage » et « trace » feront référence au traçage d'événements. Cependant, nous présenterons différents outils matériels dont certains sont conçus pour faire du **traçage d'exécution** ; dans ce cas nous utiliserons l'appellation complète pour clarifier.

Instrumentation

L'**instrumentation** est la technique consistant à modifier un logiciel pour enregistrer des événements. En pratique, elle consiste en l'ajout de points de trace dans le code du programme ou du noyau de système d'exploitation tracé. Nous verrons que l'instrumentation

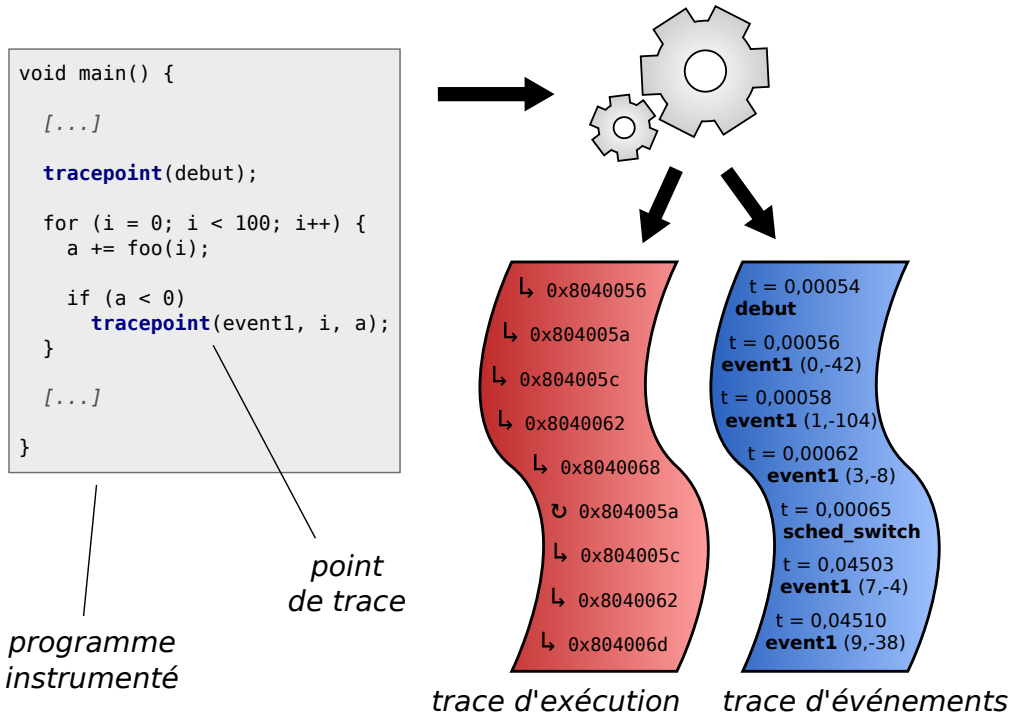


Figure 1.1 Différence entre trace d'exécution et une trace d'événements

n'est parfois pas nécessaire pour tracer un logiciel, car certains modules matériels permettent de détecter le passage par des points spécifiques du programme.

Surcoût

Le **surcoût** (*overhead*) d'une méthode de traçage est l'ensemble des modifications qu'elle provoque sur l'exécution du programme tracé. Sauf conditions exceptionnelles, ces modifications sont néfastes pour les performances du programme, d'où la qualification de « surcoût ». Elles comprennent entre autres une pollution de l'espace mémoire du programme, des ajouts dans le code exécuté, des interruptions supplémentaires et une consommation d'énergie accrue. Tous ces désagréments résultent en général en une augmentation du temps d'exécution, c'est pourquoi on utilise souvent le terme « surcoût » comme raccourci pour « surcoût temporel ».

Les concepteurs d'outils de traçage cherchent à le minimiser. Les valeurs typiques des surcoûts temporels des traceurs logiciels peu intrusifs sont inférieures à une microseconde par point de trace. Notre recherche vise à trouver des méthodes pour limiter l'impact du traçage, c'est-à-dire à en réduire le surcoût.

Matériel dédié

Nous utilisons les termes circuit matériel dédié ou **matériel dédié** pour désigner les ressources physiques embarquées sur une puce électronique pour assister une tâche particulière. De tels composants spécialisés sont aujourd'hui présents sur presque tous les nouveaux processeurs et systèmes sur puce, par exemple pour accélérer le calcul de sommes de contrôle ou traiter des flux vidéos.

Ceux qui nous intéressent sont conçus pour le débogage, le traçage et la mesure des performances. En pratique, il va s'agir de tampons de mémoire, de comparateurs d'adresses ou encore de générateurs d'estampilles de temps, dont la fonction est effectuée plus rapidement par matériel que par logiciel. Notre but est d'utiliser les capacités offertes par ce matériel dédié pour accélérer l'enregistrement de traces d'événements.

1.2 Éléments de la problématique

Des solutions matérielles de débogage et de profilage ont été proposées depuis les années 1970. Alors que les premières n'existaient que sous la forme de modules coûteux et peu répandus, de nouveaux circuits ont lentement pris place dans les processeurs destinés au grand public, notamment des compteurs de performance qui établissent des statistiques sur l'utilisation des processeurs et de la mémoire. Depuis une dizaine d'années, on assiste à une inclusion quasi-systématique de modules matériels de traçage d'exécution : *Last Branch Record*, *Branch Trace Store* et *Processor Trace* chez Intel, *Embedded Trace Macrocell* et *Program Trace Macrocell* sur les systèmes ARM, ou encore *Nexus Trace* dans les processeurs de Freescale. Plus récemment, des éléments matériels pour accélérer le traçage d'événements ont fait leur apparition : *Instrumentation Trace Macrocell*, *System Trace Module* et d'autres sont présents sur des puces largement répandues. Avec le développement de l'informatique embarquée, friande de débogage et de profilage pour économiser des ressources limitées, on risque de voir ce phénomène s'accroître davantage.

Les outils de traçage logiciel connaissent quant à eux un usage de plus en plus répandu, dû à la fois à une intrusivité de plus en plus faible et à une simplicité d'utilisation grandissante. Des fonctions spécifiques au traçage logiciel sont aujourd'hui présentes dans le code de nos systèmes d'exploitation. Cependant, l'impact du traçage n'est pas nul : l'exécution d'un programme tracé est altérée par les quelques dixièmes de microseconde pris par chaque point de trace. Ce surcoût a plusieurs facteurs, principalement l'exécution de code supplémentaire pour l'enregistrement des points de trace, mais aussi des appels système induits par le calcul des estampilles de temps (si l'appel `gettimeofday` n'est pas implémenté en VDSO) ou par les fautes de page si les données de trace ont été évincées de la mémoire principale. À cause

de ces désagréments, le nombre d'actions enregistrables est limité et il n'est pas possible de tracer un nombre arbitraire d'événements sans induire un ralentissement.

Là où les solutions purement logicielles ne peuvent plus rien améliorer car elles sont déjà optimisées au maximum, l'emploi de matériel dédié peut s'avérer efficace. Pour réduire encore l'impact du traçage d'événements, certaines étapes logicielles peuvent être accélérées par des modules matériels, à condition que ceux-ci comprennent des ressources adaptées. En effet, la plupart sont conçus pour du débogage et non du traçage, et doivent être reconfigurés pour les détourner de leur usage principal. Toutes leurs fonctionnalités matérielles ne seront pas forcément utilisées, et certaines pourront même être remaniées afin de mieux répondre aux besoins du traçage. Enfin, les traceurs logiciels n'étant pas faits pour utiliser ces ressources, leur architecture doit être repensée pour intégrer ces modifications. De plus, chaque élément matériel étant spécifique à une plateforme, les données qu'ils apportent diffèrent. Tous ces éléments de problématique demandent une étude rigoureuse, notamment pour concevoir un système unifiant les différents types de traces matérielles en un format commun.

1.3 Objectifs de recherche

Dans le contexte présenté dans la section précédente, le déroulement de la maîtrise était axé autour de la question de recherche suivante :

> *L'utilisation du matériel dédié au débogage présent dans les processeurs récents permet-il de réduire l'impact du traçage d'événements par logiciel ?*

Pour répondre à cette problématique, nous avons défini les objectifs de recherche suivants :

1. Dresser une liste des modules matériels potentiellement intéressants.
2. Étudier les caractéristiques techniques de chaque infrastructure matérielle pour déterminer les possibilités en termes de traçage.
3. Pour les modules retenus, proposer des implémentations tirant profit du matériel tout en donnant un résultat équivalent au traçage purement logiciel.
4. Évaluer les différences de performance apportées par ces modifications.

L'ensemble de ces objectifs a pour but de répondre à la question de recherche, et si possible de proposer des améliorations pour les solutions de traçage existantes. Si ces améliorations sont efficaces, un objectif complémentaire est de les intégrer au traceur LTTng.

1.4 Plan du mémoire

Au chapitre 2, nous présentons une revue de littérature faisant le point sur le matériel dédié au débogage, profilage et traçage, ainsi que sur les solutions logicielles de traçage d'événements. Le chapitre 3 décrit notre démarche de recherche en présentant la méthodologie employée pour obtenir les résultats de l'article de revue « *Hardware-Assisted Software Event Tracing* » au chapitre 4. Certains résultats complémentaires ne sont pas inclus dans l'article mais détaillés au chapitre 5. Enfin, le chapitre 6 discute nos travaux et la portée de nos résultats, tandis que le chapitre 7 conclut notre mémoire et envisage de futures évolutions.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Les infrastructures matérielles

Depuis les premiers programmes informatiques, la recherche de la cause des erreurs et leur résolution ont fait naître de nombreuses techniques de débogage qui permettent de s'immiscer dans l'exécution du processus et éventuellement d'interagir avec celui-ci. De nos jours, ces techniques peuvent aussi servir pour faire du profilage et optimiser les performances des programmes. Tout ceci peut aussi être fait de manière logicielle, ce qui nécessite une modification du code originel, ou au moins, une altération de l'exécution. Cela a pour conséquence de ne pas observer le programme étudié dans son état normal, et peut conduire aux fameux *heisenbugs* (bogues qui ne se manifestent plus lorsqu'on active le débogage). Pour éviter ces désagréments, de nombreuses solutions matérielles ont été imaginées. Aujourd'hui, elles existent sur la plupart des processeurs grand public.

Les systèmes matériels que nous avons étudiés se divisent en deux catégories : les traceurs d'exécution, qui enregistrent les adresses des instructions exécutées, et les traceurs d'événements, dont le but est d'accélérer l'enregistrement de points de trace déclenchés par logiciel.

2.1.1 Traçage d'exécution

Le traçage du flot d'exécution fut l'une des premières solutions de débogage. Dès les années 1970, des articles et brevets décrivent des architectures pour intégrer de nouveaux circuits dans les processeurs, afin d'enregistrer des informations sur l'exécution [53, 11]. Introduite sous les noms de traçage d'instruction (*instruction tracing*), traçage de programme (*program tracing*) ou traçage d'exécution (*execution tracing*), cette technique consiste à enregistrer les adresses des instructions exécutées par un processeur. La liste de ces adresses, corrélée avec la connaissance du code binaire tracé, permet de reconstituer le flot de contrôle pendant une exécution donnée. Une telle information peut servir à des fins de simple débogage, pour s'assurer que le déroulement d'un programme s'effectue normalement ou déterminer la cause d'un dysfonctionnement. Lorsque des données temporelles sont ajoutées à la trace, par exemple des estampilles de temps (*timestamps*) placées à intervalles réguliers, on peut aussi procéder à l'évaluation des performances du programme et faire du profilage [14].

Compte tenu du volume des traces générées, le choix de la destination de stockage est crucial. Alors que certaines implémentations choisissent de la mémoire sur la puce, certaines

stockent les traces en mémoire principale (RAM). D'autres exportent ces données en dehors de la machine, *via* une connexion JTAG [24], ce qui a l'avantage de ne pas modifier l'utilisation des ressources de la machine tracée. De manière à réduire la taille des traces et la bande passante consommée, certains modules matériels n'enregistrent que les déviations au flot normal d'exécution, ce qui évite de tracer le passage évident d'une adresse à la celle qui la suit immédiatement. D'autres réduisent encore le nombre d'instructions tracées, considérant par exemple que les branchements inconditionnels et les appels de fonctions peuvent être déduits de l'analyse statique du programme. Certains blocs matériels plus récents comme CoreSight ETM [2] optimisent l'encodage des traces pour réduire la production à environ 1 bit par cycle [7]. Enfin, des propositions pas encore implémentées intègrent un véritable système de compression qui permet de réduire la taille des données générées jusqu'à 454 fois [30], ou encore repensent l'architecture de traçage pour arriver à 0,15 bit par instruction [59].

Il est important de préciser que la plupart des solutions matérielles de traçage d'exécution supposent que l'utilisateur a connaissance du binaire exécuté, et peut lier chaque adresse d'exécution à l'instruction correspondante dans le code binaire. Ainsi, les traceurs matériels présentés ici ne sauvegardent jamais des instructions, mais leurs adresses, ce qui permet un encodage beaucoup plus compact.

L'ensemble de solutions matérielles de traçage présentées dans cette section est résumé dans le tableau 2.1.

Branch Trace Store (BTS) d'Intel

Sur les processeurs Intel, les unités de contrôle de performance (PMU) sont familières aux développeurs, notamment grâce à leur prise en charge par des outils de profilage comme Perf. Ces derniers permettent de calculer des statistiques sur des grandeurs comme le nombre d'instructions exécutées ou l'occurrence de fautes de cache. Pourtant, elles ne constituent pas l'ensemble des solutions de débogage : d'autres ressources sont à disposition, en particulier pour les traceurs.

La fonctionnalité *Branch Trace Store* permet de faire du traçage d'exécution. Après avoir configuré une zone tampon de taille arbitraire en mémoire principale, le processeur sauvegarde chaque branchement pris dans un enregistrement de 24 octets comprenant l'adresse de départ, celle de destination, ainsi que des drapeaux (*flags*) de contexte. Elle est présente sur les nouveaux processeurs d'Intel implémentant les registres MSR depuis le Pentium 4 [28]. Une interruption peut être soulevée en cas de dépassement d'un certain seuil de remplissage du tampon.

BTS étant initialement prévu pour tracer de petites portions de code, il présente de sérieuses limitations en termes de performance. En effet, il ne permet pas de limiter l'en-

Tableau 2.1 Principales solutions matérielles de traçage d'exécution

Nom	Plateforme	Compression	Stockage dans tampon dédié	Stockage en mémoire principale	Export *via* JTAG	Documentation disponible	Depuis
BTS	Intel			✓		✓	2000
LBR	Intel		✓			✓	2000
PT	Intel	✓	✓	✓		✓	pas encore
Program Trace	Freescale	✓	✓	✓	✓		2010
CoreSight ETM	ARM	✓	✓		✓	✓	1999
CoreSight PTM	ARM	✓	✓		✓	✓	2008

registrement des branchements à une fonction ou à un processus donné, ni à un type de branchement particulier, ce qui produit un nombre très élevé d'enregistrements. De plus, l'encodage de ces entrées n'étant pas optimisé (ce qui pourrait par exemple être fait en n'encodant que des décalages (*offsets*) d'adresses), leur poids est de 24 octets, ce qui contribue à la production de traces très volumineuses. La taille des données de trace entraîne de nombreux effets néfastes comme une pollution de la mémoire cache, une utilisation importante du bus de communication avec la mémoire principale, et la nécessité de vider le tampon d'enregistrement fréquemment. Les spécialistes de l'embarqué s'accordent à dire que BTS induit un surcoût de 20% à 100% [45], mais notre étude montre que celui-ci peut atteindre plusieurs milliers de pourcents pour des programmes ayant des hauts taux de branchement (de l'ordre de 10^8 branchements par seconde). De tels taux sont facilement atteints par des programmes usuels, voir la section 5.2.1 à ce sujet.

Une autre limitation de BTS est l'absence d'estampilles de temps (*timestamps*) dans la trace, ce qui impose une datation logicielle lors du vidage du tampon. Une telle datation étant faite ponctuellement et pour un large nombre d'enregistrements à la fois, elle est évidemment peu précise.

Last Branch Record (LBR) d'Intel

Tout comme BTS, la fonctionnalité *Last Branch Record* enregistre la trace d'exécution logicielle sur les processeurs Intel depuis la micro-architecture NetBurst (prédécesseuse de l'architecture Core). La différence majeure est que LBR sauvegarde les données dans des registres dédiés (et non en mémoire principale), ce qui élimine le surcoût dû à l'accès au bus et au partage de la cache et de la mémoire avec le processus tracé. En revanche, le

nombre de ces registres est très limité : 4 à 16 enregistrements peuvent être stockés, selon le processeur [27]. Ceci donne accès à une rétrotrace (*backtrace*), ce qui est utile si on veut remonter à la cause d'un bogue précis, inspecter le flot de contrôle avant une interruption, ou lorsqu'on examine un programme « pas à pas » [45]. Outre le fait que LBR n'induit pas de ralentissement, il est aussi possible de choisir quel type de branchement enregistrer, et ainsi faire une sélection parmi les appels de fonctions, les sauts ou les retours.

En revanche, ce système est totalement inadapté au traçage d'événements pendant une exécution normale, car il demanderait de lire les registres LBR tous les 16 branchements pour vérifier si l'adresse de l'événement recherché a été atteinte. Pour cette raison, nous n'avons pas inclus LBR dans nos expérimentations.

Processor Trace (PT) d'Intel

Processor Trace est une nouvelle solution matérielle de traçage d'exécution conçue par Intel, qui améliore et unifie les fonctionnalités de ses précédentes implémentations. Cette extension écarte les limitations de BTS et LBR en misant sur la performance et la flexibilité. Selon Intel, PT cause une perturbation minimale sur les programmes tracés [26] grâce à des ressources dédiées et à des filtres pour ne tracer que les portions d'intérêt.

Parmi ces ressources, Intel a fait le choix d'inclure des tampons dans l'unité de calcul pour stocker les traces de petite taille sans accéder à la mémoire principale [47], ce qui évite les importants délais de communication sur le bus. Le contrôle du traçage peut être fait avec des événements tels que le changement du niveau de privilège ou du contenu du registre CR3, ce qui permet respectivement de suivre du code en espace utilisateur ou noyau, ou bien un processus particulier. Ainsi, les ressources ne sont pas surchargées par des données qui n'intéressent pas l'utilisateur. Enfin, les paquets de trace produits peuvent contenir des estampilles de temps et des informations de contexte qui viennent enrichir la trace avec des données de performance.

Les caractéristiques de PT présentées par Intel semblent très prometteuses, et avec des fonctions assez similaires à ce que ARM propose déjà avec CoreSight ETM et PTM (présentés à la section 2.1.1). Malheureusement, aucun processeur implémentant Intel PT n'est disponible à l'heure actuelle. En attendant, Intel fournit des bibliothèques et exemples de code source ouvert pour générer et décoder des traces PT [47]. La prise en charge de cette fonctionnalité dans Linux est aussi prévue, en utilisant l'ABI (interface application-binaire) de Perf [52].

Quand Intel PT sera disponible, l'étude de ses performances pour une application de traçage d'événements présentera un fort intérêt.

CoreSight d'ARM

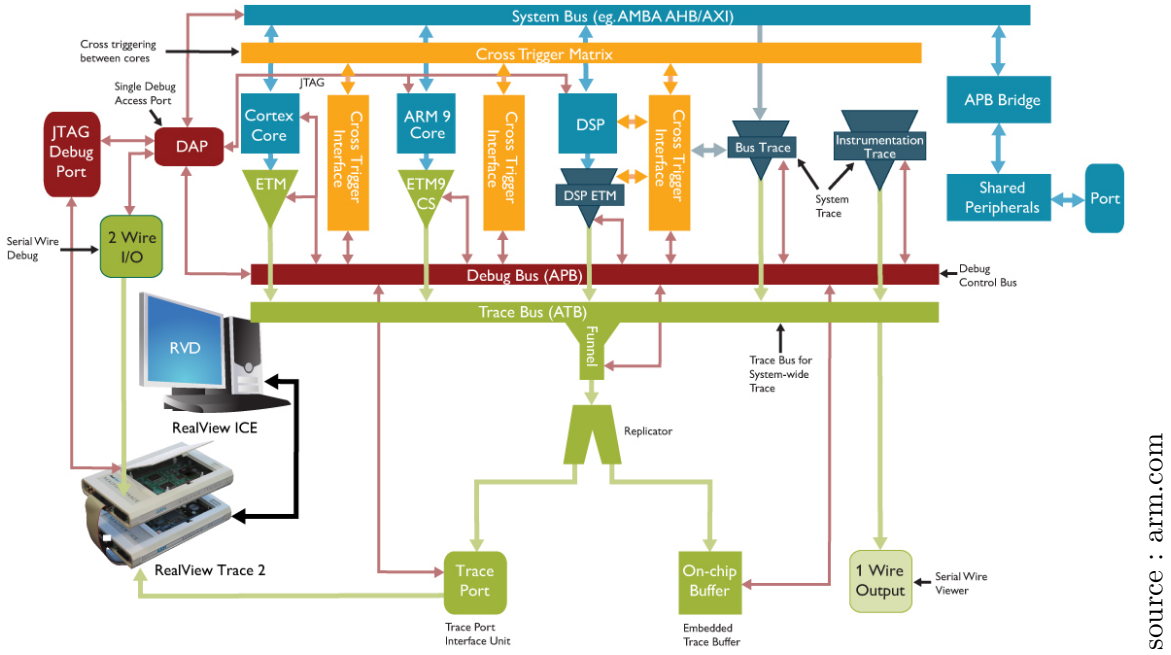
CoreSight est un ensemble de modules matériels destinés à apporter des solutions de débogage et de profilage. Chacun de ces modules, appelés macrocellules (*macrocells*), est destiné à une fonction spécifique et est indépendant des autres. Ils ont été conçus avec un fort accent sur la non-intrusivité, grâce à des ressources dédiées, et le débogage sans avoir à arrêter le processeur. Ainsi, CoreSight permet l'analyse des programmes dans leurs conditions réelles d'utilisation, ce qui permet en outre de profiler leurs performances.

En général, une combinaison de plusieurs macrocellules est intégrée dans un système embarqué en fonction des besoins de débogage de la plateforme. Une vue générale des macrocellules de CoreSight est présentée à la figure 2.1, mais nous ne nous intéresserons par la suite qu'à celles qui sont utiles au traçage.

La macrocellule *Embedded Trace Macrocell* (ETM) permet de tracer l'activité d'un processeur par le biais des instructions exécutées et des données accédées en mémoire. ETM existe depuis les premières versions de CoreSight, ce qui la rend disponible sur de nombreuses plateformes basées sur l'architecture ARM. La particularité de ce système est qu'il « écoute » le bus pour détecter les accès d'intérêt, sans agir de manière active avec l'unité de calcul. Ceci résulte en la production d'une trace qui transcrit le chemin d'exécution suivi, et qui date chaque élément avec un estampille précise au cycle près. Grâce à des déclencheurs, ETM est en mesure de n'activer le traçage que dans les conditions désirées, par exemple lorsque le contenu du registre *Context ID* correspond à un motif (pour le suivi d'un processus ou groupe de processus), selon les privilèges (espace noyau et/ou espace utilisateur) ou à l'intérieur d'un intervalle d'adresses donné (traçage d'une portion de code uniquement). Enfin, des filtres autorisent de choisir quelles informations sont à exporter dans la trace, ce qui permet d'y intégrer plus de détails ou au contraire de l'alléger pour gagner en performance [2].

Le bloc *Program Trace Macrocell* (PTM) peut être considéré comme une évolution d'ETM. Disponible seulement depuis les processeurs Cortex-A9, il est spécialisé dans le traçage d'exécution et ne permet pas d'enregistrer les accès mémoire pour des données [6]. En revanche, le traçage d'instructions a été repensé pour accroître les performances, notamment grâce à un protocole plus compact [58].

La macrocellule HTM (*AHB Trace Macrocell*) est l'équivalent de ETM pour surveiller le bus système (AHB pour *Advanced High-performance Bus*) au lieu des cœurs de processeur. Tout comme ETM, elle permet de surveiller certaines données ou adresses d'intérêt, mais à des fins de débogage et de profilage [4]. Elle met en avant les détails de l'exécution concernant le bus, qui ne sont pas déductibles de l'analyse du flot de contrôle. Notre étude portant sur le traçage d'événements, nous avons laissé HTM de côté.



source : arm.com

Figure 2.1 Vue générale des macrocellules de CoreSight

Pour collecter les données produites par toutes ces sources de trace, le modèle CoreSight propose deux options. La première est l'export des traces hors de la machine tracée, à destination d'un ordinateur « cible » qui collecte les traces *via* une interface JTAG. Autre option, les données de trace peuvent être lues directement depuis la machine hôte *via* un tampon sur la puce, appelé *Embedded Trace Buffer* (ETB) [3]. À l'intérieur de ce tampon commun, les paquets en provenances des différentes sources (ETM, HTM, etc.) sont multiplexés. Ceci présente l'avantage de corréler les événements d'origines différentes, qui n'ont pas parfois pas d'estampilles de temps ou bien des horloges de datation à fréquences différentes. Cette mémoire sur puce sert aussi de zone de transit où l'information est stockée rapidement, en attendant d'être récupérée à un moment propice. Cependant, le tampon ETB est assez petit (typiquement entre 4 et 16 Kio [58]), ce qui est trop réduit pour contenir la trace de l'exécution d'un programme courant. Par conséquent, ETB doit être sondé souvent et la mémoire vidée régulièrement pour éviter les débordements.

Nexus Trace de Freescale

Tout comme ARM, Freescale conçoit des semi-conducteurs à destination des systèmes embarqués. Pour répondre aux forts besoins de débogage et d'optimisation de ce domaine, ses processeurs embarquent plusieurs solutions comme des *Performance Monitor*, un ensemble de registres qui collectent des statistiques sur des événements matériels (faute de cache, nombre

de branchements pris ou d'instructions exécutées par unité de temps) [22]. Ceux-ci permettent de faire du profilage sur les performances d'une application.

En ce qui concerne le traçage d'exécution, les processeurs Freescale fournissent une aide matérielle au travers du système Nexus [43], un standard de modules matériels et de connecteurs pour déboguer les systèmes sur puce dans leur ensemble, indépendamment du type de processeur ou de sa marque. Nexus comprend un solution de traçage d'exécution sobrement appelée *Program Trace*, dont les données peuvent être écrites en mémoire principale ou bien exportées vers une autre machine [29]. De manière similaire à ARM CoreSight, *Program Trace* intègre des comparateurs d'adresses et de données, pour cibler la trace sur un processus ou une fonction de choix, ainsi qu'un tampon sur la puce pour accélérer le stockage de la trace produite [22]. Cette fonctionnalité est présente sur certains processeurs membres de la famille QorIQ : les séries P3 et P5 ainsi que les séries T.

Malheureusement, nous n'avons pas pu expérimenter avec *Program Trace*, car la documentation pour configurer l'accès aux traces n'est pas disponible publiquement.

2.1.2 Traçage d'événements

La section précédente présentait différents systèmes pour obtenir la trace d'exécution d'un programme. Celle-ci donne accès à l'ensemble des instructions exécutées, et permet de reconstituer le flot de contrôle. Cette information est très complète mais a un impact lourd en termes de dégradation de performance ; parfois, seuls certains événements représentent un intérêt, et l'altération du comportement normal du programme n'est pas souhaitée.

Le traçage d'événements est une solution pour enregistrer le passage par des points spécifiques d'un programme, que le développeur choisit pour ses besoins personnels : élimination d'erreur, profilage ou sécurité. Ces événements moins fréquents relatent le comportement du logiciel tracé d'une vue à un plus haut niveau, par opposition à la liste de toutes les instructions exécutées. Historiquement purement logicielle (voir la section 2.2), cette méthode bénéficie aujourd'hui d'appuis matériels : des circuits électroniques conçus pour accélérer l'enregistrement des points de trace logiciels. « Trace Système » (*System Trace*) ou « Trace d'Instrumentation » (*Instrumentation Trace*) sont les noms donnés par les constructeurs à ces blocs matériels.

N'offrant pas le flot de contrôle entier, mais seulement une sous-partie choisie par le développeur pour répondre à ses besoins, le traçage d'événements assisté par matériel offre un compromis moins intrusif.

L'ensemble des solutions matérielles de traçage présentées dans cette section est résumé au tableau 2.2.

Tableau 2.2 Principales solutions matérielles de traçage d'événements

Nom	Plateforme	Stockage dans tampon dédié	Stockage en mémoire principale	Export *via* JTAG	Documentation disponible	Depuis
Data Acquisition	Freescale	✓	✓	✓		2010
CoreSight ITM	ARM	✓		✓	✓	2004
CoreSight STM	ARM	✓	✓	✓	✓	2010
TI STM	ARM	✓		✓		2011

CoreSight d'ARM

Le système CoreSight, déjà présenté dans la section 2.1.1, comprend des modules spécialisés dans le traçage d'événements. Leur utilisation se fait *via* une zone mémoire directement adressable depuis le processeur, où chaque écriture va générer un paquet de trace pouvant contenir des données arbitraires ainsi qu'une estampille de temps (*timestamp*). Les paquets ainsi créés sont stockés dans un tampon dédié, pour ne pas interférer avec la mémoire principale. La présence d'un collecteur commun présente l'intérêt d'avoir des traces d'exécution et des traces d'événements corrélées dans le temps, et de ne nécessiter qu'un seul outil pour récupérer les données. L'avantage principal que fournit le traçage d'événements avec CoreSight, par comparaison avec un traçage purement logiciel, est l'inclusion automatique d'estampilles de temps dans les paquets de trace. Cette opération est assez coûteuse à réaliser de manière logicielle, nécessitant dans le pire des cas un appel système, ce qui induit une intrusion significative dans l'exécution du programme. Les estampilles de temps des modules CoreSight sont lues directement depuis une horloge matérielle, ce qui a l'avantage de ne pas prendre de temps processeur et d'avoir une précision au cycle près.

La première implémentation d'un tel système fut la macrocellule *Instrumentation Trace Macrocell* (ITM). Elle rassemble les caractéristiques présentées ci-dessus, et produit des paquets de données, entrelacés de paquets d'estampilles de temps [5]. Ceux-ci sont envoyés vers le tampon ETB (décrit dans la section 2.1.1), c'est-à-dire au même endroit que ceux issus des autres sources de trace.

ITM a depuis été remplacée par le bloc *System Trace Macrocell* (STM), qui apporte plusieurs améliorations [9]. Premièrement, STM est accessible directement depuis le bus système à haut débit (AXI), ce qui augmente la vitesse d'enregistrement des traces. Cette réimplémentation est compatible avec les systèmes à cœurs multiples et à plusieurs fils d'exé-

cution, en proposant 65536 canaux où l'écriture induit un point de trace, chaque canal pouvant être accédé indépendamment. D'autres fonctionnalités sont apportées par cette nouvelle version, comme la possibilité d'arrêter le processeur en cas de débordement du tampon de traçage, et la datation des paquets eux-mêmes au lieu de paquets d'estampille de temps intercalés dans les données [40]. Enfin, STM exporte la trace vers le *Trace Memory Controller* (TMC), un tampon flexible qui peut se comporter classiquement (comme ETB), mais aussi comme une file premier arrivé premier sorti (FIFO) ou un routeur qui envoie les paquets en mémoire principale [8]. Ces améliorations rendent CoreSight STM peu intrusive et hautement configurable, des qualités très prometteuses pour améliorer les logiciels de traçage d'événements.

Seuls les systèmes sur puce OMAP5 et plus récents disposent de la version CoreSight de STM (une autre version moins performante existe : TI STM, présentée dans la section 2.1.2). N'ayant accès qu'à des puces des familles OMAP3 et OMAP4, nous n'avons pas expérimenté avec CoreSight STM, mais il serait intéressant d'acquérir de nouvelles cartes de développement intégrant cette fonctionnalité.

System Trace Module de Texas Instruments

System Trace Module, dont l'acronyme est aussi STM, est une version antérieure à *System Trace Macrocell* mise au point par Texas Instruments (TI) pour la même finalité : tracer des événements logiciels. Tout comme CoreSight STM, TI STM est accessible par une zone mémoire où les écritures génèrent des paquets de trace automatiquement datés avec des estampilles de temps (*timestamps*). Contrairement à l'implémentation d'ARM, les paquets produits par TI STM sont directement envoyés vers le tampon ETB, il est alors nécessaire de les récupérer à intervalles réguliers. La version de Texas Instruments, bien que moins flexible, reste très performante et protège les accès concurrents grâce à un nombre élevé de canaux d'écriture.

Pour résumer, le *System Trace Module* de Texas Instruments fournit la même fonction que la version d'ARM décrite dans la section précédente, avec une performance légèrement réduite et moins de flexibilité. C'est cette version que nous avons utilisée au cours de notre recherche.

Nexus Trace de Freescale

Parmi les éléments du système Nexus [43] décrit plus haut, la fonction *Data Acquisition Messages* autorise du code logiciel à écrire des données arbitraires dans le canal de débogage. Freescale implémente *Data Acquisition Messages* en exposant des registres à usage spécial

où l'écriture va générer un paquet *Data Acquisition* [22]. Selon la configuration de Nexus, ce paquet est récupérable par la suite par un débogueur externe *via* une interface JTAG, ou bien accessible directement en mémoire principale. Les messages produits par ce système peuvent être agrémentés d'un identifiant de 8 bits, configurable lui aussi par logiciel, ce qui autorise jusqu'à 256 sources de trace distinctes.

Ici encore, nous n'avons pas pu expérimenter avec les *Data Acquisition Messages* car la documentation nécessaire à la récupération des traces n'est pas disponible publiquement.

Encodage des traces d'événements

Les traces générées par un outil purement logiciel comme ceux de la section 2.2 sont dans un format arbitraire, qui peut être axé sur la réduction de l'espace occupé ou au contraire, la lisibilité. Ce format est défini par le logiciel traceur selon les besoins, et peut être redéfini *a posteriori*. Dans le cas d'une infrastructure matérielle comme celles que nous étudions, la représentation binaire doit être bien pensée dès le départ, pour convenir aux usages nécessitant une grande précision ou bien des traces compactes. Les modules de CoreSight et TI présentés dans les deux sections précédentes atteignent cet objectif grâce à des traces constituées de paquets modulaires, dont le contenu dépend du type et dont la taille se base sur une unité d'un demi-octet.

Le bloc ITM de CoreSight génère des paquets de synchronisation, de débordement, de datation et de données, dans des formats décrits dans un manuel de référence publiquement accessible [5]. Par exemple, les données sont intégrées dans des paquets de type SWIT, dont la taille peut varier entre 2 et 5 octets.

La *System Trace Macrocell* ainsi que le *System Trace Module*, quant à eux, produisent une trace au format STP (*System Trace Protocol*). Le module STM de Texas Instruments respecte la version 1 de STP, tandis que la macrocellule STM de CoreSight se conforme à la version 2. La spécification de ce format est fermée [41], au sens que son accès est réservé aux industriels membres de l'Alliance MIPI, un consortium comprenant des entreprises telles que ARM ou STMicroelectronics. Nous avons tenté d'en obtenir une copie, sans succès car MIPI exige pour cela une inscription à l'Alliance à un coût significatif, et surtout ne permet pas nécessairement de publier le code source composé à l'aide de cette information. Pour utiliser STM, il a donc été nécessaire d'effectuer un travail de rétro-ingénierie pour comprendre l'encodage des traces STP. Ceci est détaillé en section 5.4.

2.2 Les traceurs logiciels sous Linux

Le traçage logiciel est une technique consistant à récupérer des informations au cours de la vie d'un programme. Contrairement au débogage, pendant lequel une application est exécutée pas à pas, un traceur enregistre des événements à la volée tout en laissant le programme s'exécuter. Cette faible intrusivité vise à laisser le logiciel tracé dans ses conditions normales d'exécution, de manière à reproduire le même comportement et les mêmes problèmes que dans une configuration réelle. Le traçage logiciel apporte donc une solution complémentaire au débogage ; il ne faut pas non plus le confondre avec le traçage d'exécution évoqué dans la section 2.1.1. Alors que ce dernier consistait à enregistrer tous les branchements pris par le processeur, les traceurs logiciels se concentrent sur des événements à plus haut niveau, qui se produisent beaucoup moins fréquemment.

Dans cette section, nous présentons les principaux outils de traçage d'événements disponibles sur les systèmes Linux.

2.2.1 Ftrace

L'outil Ftrace [49] permet de tracer des événements du noyau tels que les appels système, les gestionnaires d'interruption ou les fonctions d'ordonnancement. Il fait partie intégrante de Linux, qu'il instrumente en des points précis, représentatifs de toutes les fonctions du noyau [12]. Les événements à enregistrer sont sélectionnables *via* des entrées dans le pseudo-système de fichiers DebugFS. Une fois lancé, le traceur détecte ces événements et les affiche à l'utilisateur avec leur temps d'exécution, permettant ainsi de mesurer les performances. Leur affichage peut se faire sous différentes formes, par exemple une représentation graphique des appels de fonctions successifs et imbriqués. Cependant, Ftrace a été pensé pour une analyse des traces en temps réel, avec un format de sortie conçu pour la lecture en direct par l'utilisateur. Il n'y a pas de représentation compacte de la trace, ce qui rend cet outil inadapté pour un traçage de longue durée ou en continu. De plus, Ftrace n'est pas optimisé pour les systèmes à plusieurs cœurs car il se synchronise grâce à des verrous tournants (*spin locks*) avec les interruptions désactivées, ce qui peut induire de fortes latences [16]. Enfin, ce traceur ne peut pas être utilisé sur des applications en espace utilisateur.

2.2.2 Perf

L'outil Perf est lui aussi intégré au noyau Linux [17]. Ce n'est pas à proprement parler un traceur : sa fonction première est de récolter des statistiques sur certains événements de la vie d'un programme, comme le nombre de fautes de cache ou le nombre de cycles processeur utilisés. Perf fonctionne par échantillonnage : la production d'événements à bas

niveau incrémente des compteurs matériels, dont le dépassement est enregistré grâce à une interruption. Cet outil n'est donc pas fait pour retracer la succession de ces événements, mais indique le nombre de fois où ils se sont produits.

Perf a toutefois des fonctions secondaires : il est capable de tracer une application, mais en réutilisant l'interface de Ftrace [32]. Depuis 2010, il peut aussi faire du traçage d'exécution avec BTS [42]. Voir la section 2.3.1 à ce sujet.

2.2.3 SystemTap

SystemTap [19] est un outil à destination des administrateurs qui souhaitent recueillir des informations sur le comportement du système. Son mécanisme se base sur les Kprobes [23], des sondes placées dynamiquement à différents endroits du noyau Linux. Depuis peu, SystemTap est aussi capable d'instrumenter des programmes en espace utilisateur *via* Uprobes [31], un équivalent des sondes noyau. Il est possible de connecter des scripts à ces sondes, de manière à programmer des actions arbitraires lors de l'occurrence des événements associés aux Kprobes. Ainsi, au lieu d'enregistrer tous les points sondables dans une trace très dense, un administrateur peut se limiter aux événements d'intérêt, et choisir une action adaptée : agréger des statistiques, écrire une trace en ajoutant des estampilles de temps, etc. SystemTap mise donc sur la flexibilité, et non sur les performances. Ceci se manifeste par un surcoût non-négligeable sur le temps d'exécution, notamment dû à un mécanisme de synchronisation par verrouillage à verrous tournants (*spin locks*), mais aussi par des données trop volumineuses lorsque l'on souhaite enregistrer la trace en continu. Effet connexe, SystemTap ne possède aucun moyen de visualisation pour de larges traces (de l'ordre du Gio).

2.2.4 LTTng

LTTng [15] est un traceur axé sur la non-intrusivité avec le logiciel tracé. Pour atteindre ce but, il utilise des mécanismes optimisés pour les performances. Par exemple, la synchronisation est faite par des opérations atomiques plutôt que par des verrouillages. Ces verrouillages étant une des principales causes de latences, le mécanisme RCU [39] est utilisé partout où cela est possible pour les éviter. Afin d'atteindre une mise à l'échelle efficace, des tampons sont alloués pour chaque processeur, et ceux-ci sont utilisés circulairement (*ring buffers*).

LTTng peut tracer des événements du noyau comme les fonctions d'ordonnancement, les fautes de page, les interruptions. Pour cela il utilise des points de trace statiques ou les sondes Kprobes, comme SystemTap. Il est aussi possible de tracer un programme en espace utilisateur avec la version adéquate de l'outil (LTTng-UST), en instrumentant l'application *via* des points de trace positionnés aux endroits voulus dans le code.

La trace générée se conforme au format CTF [18], une spécification qui autorise à la fois flexibilité et compacité. Elle peut ensuite être lue par plusieurs outils, avec une interface graphique ou en ligne de commande. Cette dernière méthode, très légère, permet de travailler avec des traces de plusieurs dizaines de Gio.

2.3 Les logiciels s'aidant du matériel

Certains outils existants tirent déjà profit des solutions matérielles offertes par les nouveaux processeurs. Chaque fonctionnalité matérielle est généralement accompagnée d'un logiciel qui la prend en charge ; malheureusement il s'agit bien souvent de logiciels propriétaires dont les fabricants espèrent tirer profit en gardant la source fermée. Néanmoins, pour la majorité du matériel de traçage que nous avons présenté, une documentation complète est disponible, ce qui a permis des implémentations accessibles à tous. La disponibilité du code de ces logiciels nous a autorisé à les étudier.

Dans cette section, nous décrivons certaines implémentations logicielles existantes, qui utilisent le matériel de traçage BTS et ETM.

2.3.1 Perf et BTS

Le système *Branch Trace Store* (BTS) d'Intel, présenté dans la section 2.1.1, est pris en charge par le logiciel Perf et le noyau Linux. Il est ainsi possible d'enregistrer la trace d'exécution de n'importe quel programme sous Linux, c'est-à-dire de connaître l'intégralité des branchements que le processeur a suivi.

Les deux commandes suivantes en donnent une illustration : la première enregistre la trace dans un fichier temporaire, tandis que la deuxième l'affiche dans un format lisible par l'utilisateur.

```
perf record -e branches:u -c 1 -d PROGRAMME
perf script -f ip,addr,time
```

Un exemple de résultat, présenté ci-après, affiche les adresses source et destination des branchements, ainsi que les estampilles de temps correspondantes. Le fait que toutes ces estampilles aient la même valeur montre bien que BTS ne fournit pas de datation : celle-ci est ajoutée de manière logicielle lors du vidage du tampon, une fois de temps en temps.

```
# ========
# nrcpus online : 8
# cpudesc : Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz
# total memory : 12223968 kB
```

```
# ========
#
29968.676878:  ffffffff8166deec =>        34b4401420
29968.676878:  ffffffff8166deec =>        34b4401420
29968.676878:        34b4401423 =>        34b4404920
29968.676878:  ffffffff8166deec =>        34b4404920
29968.676878:  ffffffff8166deec =>        34b440493f
29968.676878:  ffffffff8166deec =>        34b4404946
29968.676878:        34b44049b2 =>        34b44049cf
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049df =>        34b4404bb8
29968.676878:        34b4404bc0 =>        34b4404be0
29968.676878:        34b4404bea =>        34b4404c00
29968.676878:        34b4404c17 =>        34b44049c3
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
29968.676878:        34b44049d3 =>        34b44049b8
[...]
```

L'implémentation du mécanisme de traçage est faite à la fois du côté du noyau Linux et du côté utilisateur, c'est-à-dire dans le code de Perf. Celle-ci effectue une double copie des données générées par BTS, qui est selon nous une cause de fort ralentissement. Nous décrivons davantage ce problème dans la section 5.1.

2.3.2 Linux et ETM

Linux fournit une prise en charge basique de l'*Embedded Trace Macrocell* (ETM), présentée en section 2.1.1. Au travers d'entrées dans le pseudo-système de fichiers SysFS, un utilisateur peut activer le traçage du noyau, et récupérer les données produites en lisant un fichier dans `/dev`. Plusieurs outils à source ouverte peuvent ensuite être utilisés pour décoder la trace et l'afficher dans un format lisible par un humain [51, 46].

Les commandes ci-dessous montrent comment le traçage peut être activé, et à quoi ressemble le début d'une trace d'exécution enregistrée grâce à ETM.

```
# Démarrer le tracage
echo 1 > /sys/devices/etm/trace_running
# Provoquer un appel système
touch /tmp/nouveaufichier
# Arrêter le tracage
echo 0 > /sys/devices/etm/trace_running
# Lire la trace et la décoder
cat /dev/tracebuf | ./etmdecoder --etm
    Cycle count 0
  I-sync Context 006ce2c4, IB 29, Addr 000085e8
  E(000085e8) Waited 2
    Cycle count 527
  I-sync Context 006ce2c4, IB 29, Addr 00008900
  E(00008900) Waited 1
    Branch 000085e8 ARM
  E(000085e8)
    Cycle count 248
  I-sync Context 006ce2c4, IB 29, Addr 00008900
  E(00008900) Waited 1
    Branch 000085e8 ARM
  E(000085e8)
    Cycle count 248
  I-sync Context 006ce2c4, IB 29, Addr 00008900
  E(00008900) Waited 1
    Branch 000085e8 ARM
  E(000085e8)
    Cycle count 248
[...]
```

Avec l'implémentation originale de Linux, il n'était pas possible de tracer autre chose que le code du noyau, ni de se restreindre à une portion de code d'intérêt. Pour ces raisons, nous avons modifié le noyau Linux pour lui donner plus de flexibilité. Ce travail est présenté dans la section 5.3.

2.3.3 Outils à source fermée

Plusieurs logiciels propriétaires permettent de configurer les infrastructures de traçage matériel pour générer et récupérer des traces. Ils sont en général conçus par les fabricants

de ce matériel eux-mêmes, et vendus à des prix pouvant dépasser 19000 US$ [57]. Nous présentons ici ceux liés aux blocs de traçage décrits dans cette revue de littérature.

Les quatre logiciels suivants sont des environnements de développement qui prennent en charge certaines des macrocellules de CoreSight sur des systèmes embarqués :

— *Code Composer Studio* de Texas Instruments ;
— *DS-5 Development Studio* d'ARM ;
— *STWorkbench* de STMicroelectronics ;
— *TRACE32-ICD* de Lauterbach.

L'environnement de développement *CodeWarrior Development Studio* de Freescale permet quant à lui d'utiliser les éléments Nexus des processeurs Freescale tels que *Program Trace* et *Data Acquisition*.

2.4 Conclusion de la revue de littérature

Au cours de notre revue de littérature, nous avons présenté les infrastructures matérielles dédiées au traçage, qu'il s'agisse d'enregistrement du flot d'exécution ou bien d'événements à plus haut niveau. Ces systèmes étant relativement nouveaux, ils ne sont pas encore très utilisés et peu d'études ont été faites dessus. De façon analogue, les outils en tirant profit sont rares, surtout dans le domaine du logiciel libre.

D'autre part, nous avons présenté différents outils logiciels dédiés au traçage. Un de leurs points communs était de chercher à réduire le surcoût lié à leur utilisation, de manière à étudier des programmes dans leurs conditions normales d'utilisation.

Il semble donc important d'étudier comment des logiciels de traçage comme LTTng peuvent bénéficier des capacités prometteuses offertes par le matériel dédié, présent sur la plupart des nouveaux processeurs. Tirer profit de ce potentiel rendrait le traçage plus transparent et moins intrusif, et pourrait favoriser son utilisation par les développeurs, pour rendre leurs programmes plus performants.

CHAPITRE 3

MÉTHODOLOGIE

L'amélioration des performances d'un traceur logiciel en utilisant des blocs matériels passe d'abord par la mise en place d'une méthodologie. Celle-ci doit notamment définir la manière de mesurer les performances, et proposer une échelle objective pour comparer les résultats et affirmer si oui ou non il y a amélioration.

3.1 Environnement de travail

Le laboratoire DORSAL avait à sa disposition plusieurs machines intégrant des modules matériels de traçage. La première étape a été de déterminer comment utiliser au mieux ces ordinateurs pour évaluer le surcoût lié à l'utilisation d'un traceur. Chacune de ces machines ayant les capacités d'un vrai ordinateur, deux possibilités s'offraient à nous :

— installer un système d'exploitation complet sur chaque machine de test pour produire et analyser les résultats ;

— installer un système minimal sur les machines de test et analyser les résultats sur une station de contrôle.

Nous avons choisi la deuxième solution, de manière à ce que nos mesures influent le moins possible sur le comportement des machines de test. Notre configuration matérielle comprenait ainsi quatre machines : deux ordinateurs de bureau (l'un servant de station de contrôle), et deux cartes de développement à système sur puce.

La figure 3.1 illustre les trois machines de test utilisées.

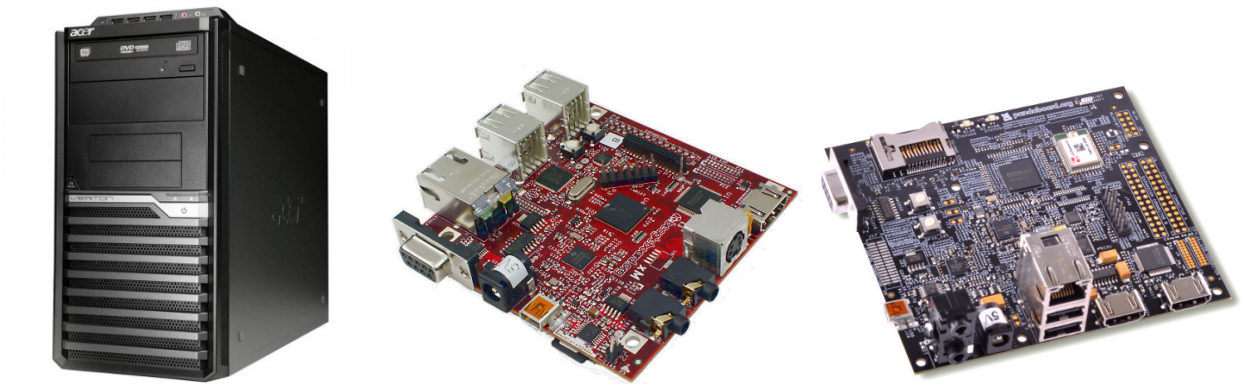
3.1.1 Station de contrôle

Les expériences ont été lancées depuis une station de contrôle, qui était simplement un ordinateur de bureau. C'est depuis cette machine que les tests étaient lancés à distance, et que les traces étaient récupérées pour analyse.

La station de contrôle était connectée aux machines de test par un lien Ethernet à 100 Mbit/s, en utilisant les services suivants.

— Une connexion SSH pour administrer le système d'exploitation une fois la machine démarrée.

— Un serveur NFS pour « partager » des fichiers et répertoires communs entre la station de contrôle et les machines de test, sans avoir à les copier explicitement à chaque



sources : jrlinton.co.uk, liquidware.com, prnewswire.com

Figure 3.1 Photographies des machines de test. De gauche à droite : un ordinateur de bureau à processeur Intel x86 ; une carte Beagleboard-xM ; une carte Pandaboard.

modification.

— Des serveurs DNS et TFTP pour un démarrage en PXE (*via* le réseau). Ainsi lors d'une réinitialisation, les machines de test téléchargent le noyau Linux à démarrer depuis la station de contrôle. Ceci était particulièrement utile car nous avons parfois compilé plusieurs centaines de noyaux différents pour une même machine ; une réinstallation de Linux « à la main » aurait été extrêmement fastidieuse.

3.1.2 Machine de test Intel x86

L'ordinateur de bureau choisi pour expérimenter sur le module *Branch Trace Store* d'Intel comprenait un processeur à 4 cœurs (8 cœurs virtuels grâce à l'*hyperthreading*) et 6 Gio de mémoire principale. Ce processeur récent dispose de nombreuses fonctionnalités de débogage et profilage, notamment BTS et LBR.

Tableau 3.1 Caractéristiques de l'ordinateur de bureau

Carte mère	DX58SO d'Intel
Processeur	Intel Core i7-3770 cadencé à 3,4 GHz
Cœurs physiques	4
Mémoire vive	6 Gio, Kingston DDR3 SDRAM à 1333 MHz
Système d'exploitation	Arch Linux
Matériel de traçage	BTS, LBR

Pour cette machine comme pour les autres, nous avons installé le système d'exploitation Arch Linux, reconnu pour sa légèreté. Arch Linux n'a en effet ni environnement graphique, ni module superflu installé par défaut, et occupe un volume total inférieur à 350 Mio une

fois installé. Ceci nous a permis de disposer des logiciels dont nous avions besoin, mais rien d'autre, afin de perturber au minimum nos tests.

3.1.3 Machine de test Beagleboard-xM

La carte de développement choisie pour expérimenter sur le module CoreSight *Embedded Trace Buffer* est une Beagleboard-xM, disposant d'un processeur à architecture ARM, et des modules matériels dédiés ETM et ETB. Pour la même raison que pour la machine du précédent paragraphe, nous y avons installé la distribution Arch Linux.

Tableau 3.2 Caractéristiques de la Beagleboard-xM

Système sur puce	OMAP3530 de Texas Instruments
Processeur	ARM Cortex-A8 cadencé à 720 MHz
Cœur physique	1
Mémoire vive	512 Mio, low-power DDR à 200 MHz
Système d'exploitation	Arch Linux
Matériel de traçage	CoreSight ETM, CoreSight ETB

3.1.4 Machine de test Pandaboard

La carte de développement choisie pour expérimenter sur le module *System Trace Module* de Texas Instruments est une Pandaboard, qui embarque un processeur bi-cœur Cortex-A9 de ARM, accompagné du matériel de traçage STM et ETB. Encore une fois, le système installé sur cette machine est Arch Linux.

Tableau 3.3 Caractéristiques de la Pandaboard

Système sur puce	OMAP4430 de Texas Instruments
Processeur	ARM Cortex-A9 cadencé à 1000 MHz
Cœurs physiques	2
Mémoire vive	1 Gio, low-power DDR2 à 400 MHz
Système d'exploitation	Arch Linux
Matériel de traçage	TI STM, CoreSight ETB

3.2 Traçage d'une application

Au cours de nos expériences, nous devions mesurer le surcoût lié au traçage, c'est-à-dire les effets secondaires induits sur l'exécution du programme tracé. Le surcoût se manifeste en général par des désagréments : latences, altération des mémoires cache, interruptions supplémentaires ou encore consommation d'énergie. N'étant pas en mesure d'enregistrer toutes ces données, nous avons fait le choix de nous concentrer sur la variation du temps d'exécution, pour deux raisons. Premièrement, le temps est la première quantité directement perçue par l'utilisateur, c'est elle que les développeurs cherchent à réduire en priorité. D'autre part, la plupart des autres désagréments que peut induire le traçage vont au final affecter le temps d'exécution : par exemple, l'éviction de pages mémoire pour enregistrer la trace demande de charger ces pages à nouveau ultérieurement, ce qui prolonge l'exécution *in fine*.

Dans la suite, nous proposons donc de définir le « surcoût » et d'utiliser cette quantité mesurable comme témoin de l'impact d'un traceur.

3.2.1 Définition du surcoût

Pour un programme logiciel et un outil de traçage donnés, nous définissons le surcoût comme la différence relative entre du temps d'exécution moyen du programme non-tracé et celui du programme tracé.

3.2.2 Mesure du surcoût

Programmes représentatifs

Le choix des programmes à tracer est important. Pour être complète, notre étude doit être portée sur un ensemble de logiciels représentatifs de catégories, chaque catégorie ayant une utilisation caractéristique des différentes ressources de l'ordinateur. Nos tests ont donc été conduits sur de nombreux programmes, certains utilisant beaucoup d'entrées-sorties, d'interruptions, de calcul ou encore des combinaisons hybrides : `wget`, `gcc`, `md5sum`, `sleep`, etc.

Cependant, la diversification des programmes de test et des résultats qui leur sont associés est un obstacle à la compréhension de cette étude. De plus, nous avons constaté que ces résultats allaient toujours qualitativement dans le même sens, quels que soient les logiciels tracés. Pour cette raison, nous avons choisi de présenter les résultats des programmes pour lesquels les données étaient les plus constantes : ceux qui font peu d'entrées-sorties et beaucoup de calcul. En effet, les programmes à fortes entrées-sorties dépendent de conditions extérieures (disque, réseau, etc.) et sont plus sujets à des variations dans les résultats.

Dans la suite, les résultats présentés sont valables pour le traçage :

— de logiciels personnalisés, forgés par nos soins pour faire du calcul (pour ETM et STM) ;
— des logiciels `md5sum` et `sha256sum`, calculant respectivement les sommes de contrôle MD5 et SHA256 d'un fichier de 700 Mio (pour BTS).

Traceur représentatif

Pour évaluer les bénéfices de nos implémentations de traçage s'assistant du matériel, il fallait un traceur logiciel de référence, pour comparaison. Notre choix s'est porté sur LTTng, qui est la référence en matière de faible intrusivité.

Le traçage des programmes s'effectuait avec les commandes suivantes :

```
lttng create
lttng enable-event -a -u
lttng start
# Lancement et chronométrage du programme à tracer
time PROGRAMME
lttng stop
lttng destroy
```

Conditions de test

Chaque série de tests consistait à mesurer le temps d'exécution d'un logiciel donné, soit non-tracé, soit tracé avec LTTng, soit tracé avec l'aide du matériel. Afin d'obtenir des résultats fiables et non-dépendants de conditions occasionnelles, nous avons :

— répété chaque mesure plusieurs dizaines de fois et calculé une moyenne ;
— ignoré les premières mesures de chaque série par mesure préventive, afin de se concentrer sur des situations de régime établi ;
— pris nos mesures sur des exécutions longues (entre une et plusieurs centaines de secondes) pour que le temps d'activation de l'horloge soit négligeable ;
— pris les conditions (non-tracé, traceur A, traceur B) dans un ordre aléatoire.

Horloge

Le temps d'exécution était mesuré grâce à la commande Unix `time` (qui l'obtient *via* l'appel système `wait3`), ou par la différence des temps retournés par deux appels système `gettimeofday`.

Le temps était enregistré uniquement une fois l'infrastructure de traçage prête, pour ne pas tenir compte du temps de mise en place (qui peut être fait une seule fois au démarrage

du système). Pour LTTng, cela veut dire que la session et la trace étaient lancées avant le départ du chronomètre.

Producteur de trace, consommateur de trace

Les configurations de traçage que nous avons utilisées comportaient au moins deux processus distincts : un producteur de trace (i.e. le logiciel tracé) et un consommateur. Ce dernier est chargé de récupérer la trace et de l'enregistrer. Dans le cas de LTTng, le consommateur est appelé démon de session. Des les cas matériels, il s'agit de programmes chargés de vider le tampon ETB, ou encore du noyau Linux lui-même qui copie la trace pour la sauvegarder. Le processus de récupération peut être différent mais dans tous les cas, la trace est écrite dans un fichier.

Ce modèle autorise une exécution asynchrone du producteur et du consommateur : le logiciel tracé peut avoir fini de s'exécuter alors que le consommateur de trace n'a pas fini de vider un tampon. Nous n'avons pourtant pas mesuré la durée d'exécution des consommateurs, considérant que dans nos conditions de régime établi (exécutions de plusieurs secondes ou centaines de secondes), la trace totale est largement plus grosse que la taille des tampons utilisés, et le vidage du dernier tampon ou son écriture sur le disque n'est pas significative, pas plus que ne le sont les propriétés de mise en cache du système d'exploitation.

CHAPITRE 4

ARTICLE 1 : HARDWARE-ASSISTED SOFTWARE EVENT TRACING

Authors

Adrien Vergé
École Polytechnique de Montréal
adrien.verge@polymtl.ca

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

Submitted to

ACM Transactions on Embedded Computing Systems, February 21st, 2014.

Abstract

Event tracing is a reliable and low-intrusiveness method to debug and optimize systems and processes. Low overhead is particularly important in embedded systems where resources and energy consumption is critical. The most advanced tracing infrastructures achieve a very low footprint on the traced software, bringing each tracepoint overhead to less than a microsecond. To reduce this still non-negligible impact, the use of dedicated hardware resources is promising. In this paper, we propose complementary methods for tracing, that rely on hardware modules to assist software tracing. We designed solutions to take advantage of CoreSight STM, CoreSight ETM and Intel BTS, which are present on most newer ARM-based systems-on-chip and Intel x86 processors. Our results show that the time overhead for tracing can be reduced by up to 10 times when assisted by hardware, as compared to software tracing with LTTng, a high-performance tracer for Linux. We also propose a modification to the Perf tool to speed BTS execution tracing up to 65%.

Keywords

ARM CoreSight, debugging, dedicated hardware, event tracing, Intel BTS, LTTng

4.1 Introduction

Tracing is a monitoring method to record the runtime behavior of a program, for debugging, optimization or performance measurement purposes. Traces are generated during execution. They contain series of timestamped events, which may be used to understand and model a process execution. Traces can be analyzed live or later, locally or on a remote host. Unlike classical debugging, tracing is focused on low-intrusiveness, allowing to study a process with a minimal alteration of its execution. Although lightweight, software tracing solutions have non-zero side-effects because of extra executed code, cache perturbation and other alterations to the execution path [14].

To facilitate program development, hardware manufacturers such as Intel and ARM embed dedicated debugging circuits in their newer processors. For instance, Intel BTS and ARM CoreSight ETM provide program tracing (recording the sequence of executed instructions), whereas ARM CoreSight STM is designed to timestamp instructions that write to a specific area. Although having different purposes, hardware debugging circuits provide dedicated resources (comparators, buffers) that can be used to trigger events and store data with almost zero overhead. By reconfiguring these resources, we want to take advantage of hardware circuits to make tracing more lightweight.

The objective is to measure the benefits of using specialized hardware components in the Linux Trace Toolkit next-generation (LTTng) [15], a reference infrastructure for kernel and user-space tracing for Linux. We designed solutions to retrieve trace information, similar to LTTng software traces, from hardware blocks on ARM (with CoreSight ETM, STM, ETB [5]) and Intel (with BTS [25]). We developed tracing tools that configure these hardware circuits and retrieve data of interest to produce traces. We measured the time overhead of such devices versus non-traced executions, in order to compare to LTTng-UST pure-software tracing.

We describe related work in section 4.2. We detail the test environment and the hardware debugging components that we tested in section 4.3. The methodology and results are presented in section 4.4 for using hardware-assisted software tracing, and in section 4.5 for using execution path tracing hardware. We conclude and discuss future work in section 4.6.

4.2 Related work

This section presents related work in the two main related areas: software tracing tools and tracing systems using hardware components.

4.2.1 Software tracing

A wide range of solutions have been developed to trace programs with pure-software. They either use interrupt-based methods or instrumentation (code modification). We present here the main software tracers for Linux.

Ptrace

The ptrace [44] system call is an old Unix functionality enabling a process to control another one, including reading its state and intercepting its system calls. It is used in debuggers and programs with moderate requirements on performance, such as gdb and strace. By giving access to a process internals, ptrace provides powerful control, albeit at a significant performance cost. This is in large part because each communication between the tracer and the traced process needs at least two context switches. Because of ptrace's intrusiveness, newer tools have been designed with lower impact on the traced process execution.

Ftrace

Ftrace [12] is a tool to monitor the system's behavior by instrumenting the kernel. It is meant to debug and profile kernel-level problems by tracing events such as system calls, interrupt handlers or scheduling functions. As Ftrace was developed to trace the kernel's internals, it does not support user-space program tracing. Moreover, it synchronizes on multi-core systems by spinlocking with interrupts disabled, which has a non-negligible impact on performance [16].

SystemTap

SystemTap [19] is an infrastructure that provides functional and performance debugging for Linux. It uses dynamic probes (Kprobes [23]) to hook on specific points of execution in the kernel, and more recently Uprobes [31] to instrument functions in user-space. Custom instrumentation can be defined via a scripting language, that once compiled into a kernel module, outputs trace information in text format. SystemTap is limited by its output format, which is not efficient for tracing programs with highly frequent events or very large traces.

LTTng

LTTng [15] is a tracing infrastructure focused on performance and output format flexibility. It achieves efficiency by using scalable and lockless methods such as read-copy-update (RCU) [39] and allocating per-CPU data structures. Hence, tracepoints are fast and the

impact on cache is minimized. LTTng's output trace complies with the Common Trace Format [18], a flexible and lightweight format supporting arbitrary event types and compression. LTTng supports high-performance kernel-space and user-space tracing (both sharing the same clock source), and all traces can be displayed in a graphical analyzer such as TMF [37], for better interpretation. For these reasons, we chose LTTng-UST (user-space tracing version of LTTng) as a reference to conduct our experiments. The results described in the rest of this paper refer to LTTng version 2.3.

4.2.2 Hardware-assisted tracing

Some tools take advantage of dedicated hardware capabilities in order to debug and measure program performance. We present here software solutions that use hardware components related to tracing.

Perf

Perf [17] is a program profiler for Linux. It was not initially meant for tracing, yet it can trace the sequence of instructions executed by a process on new Intel platforms. To achieve this, Perf uses Intel's BTS hardware registers, which are detailed in section 4.3. BTS control is done in the kernel through the Perf application binary interface, hence custom tracing programs can re-use this ABI to take advantage of these hardware capabilities.

Since recently, Perf also provides support for Last Branch Record (LBR) [27] registers on Intel processors. This feature enables automatic recording of the last taken branches. Depending on the CPU version, LBR stores from 4 to 16 records [27], which is useful for call stack debugging, but too limited for event-based tracing.

Linux

The Linux kernel provides basic CoreSight ETM support for OMAP3 chips. CoreSight ETM is a hardware facility to trace the sequence of executed instructions. It is detailed in section 4.3. Controling ETM is achieved via an entry in the sysfs virtual filesystem, but only a limited subset of the options offered by ETM is available. For instance, it is neither possible to change the address range to trace, nor can a specific context ID be followed (which however, ETM is capable of). Patches (including ours) were proposed to add more functionality and support.

4.2.3 Other tracing hardware

Other hardware facilities exist on different architectures for program tracing. We present here two common architectures offering interesting features.

Freescale

Freescale processors offer debug facilities compliant with the Nexus standard [43], in particular an on-chip trace buffer that can capture real-time bus information [20]. It can be configured to either store data for specific memory accesses, or changes in the execution flow, which allows program tracing. The triggering system is similar to CoreSight ETM, allowing to start and stop trace upon combinations of conditions, such as address range matching and opcode type. Another feature called Data Acquisition enables instrumented software to write data directly in the debugging channel. Also, trace buffers have enough capacity (typically 16 KiB) and bandwidth to trace events from all cores without loss [21].

The generated trace data can then be fetched by another machine via a JTAG interface connector [24], stored in a dedicated buffer or sent to main memory. Although trace can be collected by proprietary software, there is no public documentation available. A direct comparison was therefore not possible and it was not included in our study.

Intel

Intel has designed a new execution tracing solution called Processor Trace (PT) [26]. It enhances and unites previous tracing implementations [25, 27] in an optimized extension that is capable of timestamping, filtering, and tracking specific processes (via the CR3 register). The dedicated hardware facilities also include caching buffers to store small traces without accessing main memory [47], thus avoiding the BTS bus usage drawbacks.

Processor Trace is promising but not available on silicon at this time. Yet, Intel provides open-source libraries for generating and decoding traces [47]. Support in Linux is also planned through the Perf ABI [52].

4.2.4 Existing use of CoreSight and BTS

The hardware facilities that we used in our study have existed for a few years. They are already taken advantage of in existing tools, for various purposes.

Debugging

CoreSight provide debugging features such as hardware and software breakpoints that can be used to stop a program or execute step by step. These mechanisms are designed for multi-core processors, hence they are able to send stop signals to other processors and act on a whole system. These features are used by development environments [56, 10], and software-hardware co-debug platforms have been proposed [34, 55].

Security

Tracing infrastructures are also used for security. Scherer and Horváth propose a watchdog solution to monitor the system on ARM-based platforms. They use CoreSight debug and trace hardware blocks to make and record measurements inside the system and estimate its healthiness [50]. On Intel x86 platforms, Branch Trace Store (BTS) can be used to detect security breaches. By monitoring addresses of executed instructions to detect deviations, Yuan et al. propose system security enhancements. They rely on the BTS debugging registers to detect abnormal control flows and identify unexpected code execution [62].

Programming

Another use of execution tracing is to speed up development processes by exposing execution information alongside the source code. Tralfamadore [35] is a system that displays execution trace analysis in a source code browser. By using BTS, it helps developers to track the control flow and write their programs more logically.

4.3 Test environments

Our experiments were run on three specific platforms. However, the hardware modules used for tracing are found in many other processors and systems-on-chip.

The platforms used were:

— an OMAP3530 system-on-chip with a ARM Cortex-A8 CPU and 512 MiB of LPDDR, integrated onto a Beagleboard-xM development card;

— an OMAP4430 system-on-chip with a dual-core ARM Cortex-A9 and 1 GiB of LPDDR2, integrated onto a Pandaboard development card;

— a desktop computer with an Intel Core i7-3770 processor (4 physical cores at 3,4 GHz) and 6 GiB of DDR3 RAM.

The first two platforms integrate ARM CoreSight debugging components, whereas the third one embeds Intel Branch Trace Store registers. These devices are presented in the next

subsections.

4.3.1 ARM CoreSight

ARM CoreSight [5] is a set of hardware blocks that provide trace and debug functionalities for complex systems-on-chip. There is a variety of trace sources and collectors, allowing traces of different types to be timestamped and multiplexed in the same output.

The only CoreSight components that we used are ETM, STM and ETB. Their interaction is summarized in figure 4.1.

CoreSight ETM

The Embedded Trace Macrocell (ETM) is an instruction and data tracer. It enables reconstruction of program execution by recording jumps. When activated, ETM watches the core's internal buses to detect branches with low interference with execution. ETM timestamps branches with cycle-level precision, and supports output filtering and compressing.

CoreSight STM

The System Trace Macrocell (STM) records and timestamps software events [40]. It allows real-time instrumentation of software by providing a memory area where software writes are converted to hardware messages. These messages are automatically timestamped and assigned to the requested channel. The area is divided into several channels, which allows multiple programs to be debugged at the same time.

The version we used is the TI STM, present on our development board. It is an earlier version of STM, not designed by ARM but with very close capabilities and the same output format [41]. In the rest of this article, we use the acronym "STM" indifferently for both versions.

CoreSight ETB

The Embedded Trace Buffer (ETB) stores traces from different sources in a single place. It collects streams output from the ETM and the STM, and allows deferring the retrieval of traces, acting as a buffer.

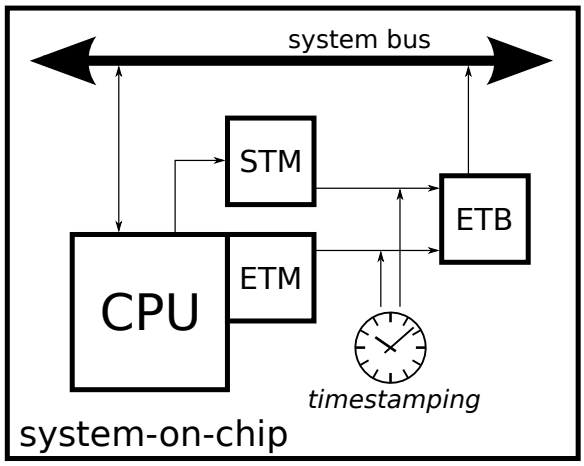


Figure 4.1 Overview of the CoreSight components used

4.3.2 Intel

BTS

The Branch Trace Store (BTS) [25] system stores every branch taken during execution to a user-defined area in memory. The BTS registers are part of Intel's Model-Specific Registers (MSR), embedded in newer processors made by Intel since the Pentium 4 [28].

BTS allows to define a zone in main memory where the CPU will automatically store entries when encountering a branch. Any deviation from the execution flow (that is, not executing the next instruction) is saved in a 24-byte entry. BTS can throw an interrupt when the buffer overflows a given threshold. The user is responsible for draining this buffer to save tracing data. It can also be configured as a circular buffer, if only a backtrace is needed.

Embedded systems specialists estimate the BTS overhead between 20% and 100% [45], partly because the CPU enters a specialized debug mode associated with a 25 to 30 times slowdown [54, 25].

The BTS design is summarized in figure 4.2.

4.3.3 Measuring the overhead of tracing

The experiments presented here aim at lowering the tracing overhead, thus minimizing the impact on the traced processes execution. Different measurable side effects can estimate this impact, for example the total elapsed execution time, extra system calls, accesses to memory and page faults, as well as cache memory usage or energy consumption. Among these, we selected the execution time as reference, since it is the major criteria for developers in most cases. Moreover, most other effects (such as page faults) also impact the execution

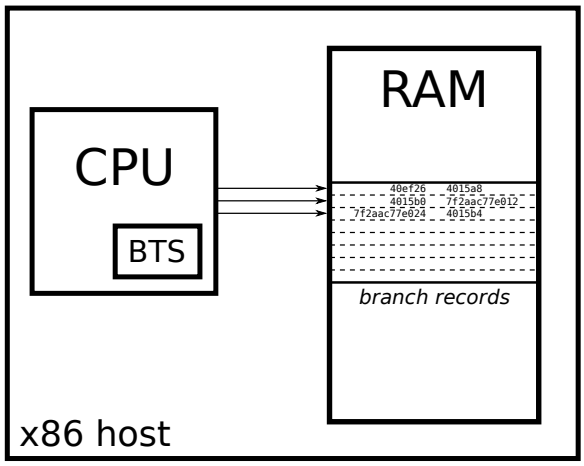


Figure 4.2 Overview of Intel Branch Trace Store

time.

In the rest of this article, the term "overhead" will refer to the modification to traced processes normal execution. To estimate the "overhead", we will measure the execution time difference associated with tracing. Depending on the experiment, this is obtained via the `time` command or with the `gettimeofday` system call. To reduce variability and influence of unrelated events, each test was run several times and over long execution times (between one and hundreds of seconds). In the case of two programs running in parallel (a trace producer and a trace consumer), only the traced program is timed. We assume that with long execution times, both the producer and the consumer enter a steady state, so that the producer's execution time is relevant by itself.

4.4 Using system tracing hardware

The main functionality of tracing systems is to provide tracepoints. When the execution of a program reaches a tracepoint, an event is recorded along with customizable information such as timestamp or process ID.

Modern tracers like LTTng, focused on performance, achieve a good efficiency with low-overhead and precise tracepoints. However, pure-software methods still induce a slowdown, mainly due to synchronization and timestamp computation (which may involve a system call in the worst case).

4.4.1 System Trace Macrocell

Design

The STM module offers system tracing capability: writing to a specific zone in memory generates timestamped messages, and stores them in a dedicated buffer. This led us to design a tracing solution that takes advantage of these hardware circuits. Instead of using the `tracepoint` function provided by the LTTng-UST library, our prototype directly writes to the STM memory area. Similarly, the trace consumer does not read from shared memory, but retrieves data from the ETB when needed.

The software part of recording tracepoints is lightened: timestamping is automatically done via a hardware clock, as is transportation to the ETB for further fetching. Synchronization between trace producers is guaranteed, as long as each process writes to its own STM channel. There are hundreds of possible channels (the exact number depends on the hardware implementation), so many programs can be traced at the same time. Channel information is included in the generated output, to enable message decoding and demultiplexing after retrieval from the ETB.

Our implementation results in two programs: a trace producer, i.e. a traced program in which events occur; and a trace consumer, i.e. the program that will fetch raw hardware data from the ETB, decode and store it. The ETB has a fixed size, for this reason, the producer and the consumer must communicate to avoid overflow. They share a common page in memory to communicate and synchronize with semaphores. In particular, they both update a counter that represents buffer usage. The system's scheduler is meant to keep a balance between trace production and trace consumption. However, if the ETB becomes full, the producer stops and yields CPU time to the consumer to empty it.

Results

We measured the performance difference between this approach and the original LTTng-UST in two configurations, meant to be representative of either the worst case, or a more realistic situation. In the first case, the tracing overhead is maximum: the benchmark process runs a loop that does only tracing. In the second case, some arithmetic computation is executed in each iteration, to simulate a real program behavior. In both cases, tracing at each iteration is done either by calling LTTng-UST's `tracepoint`, or by interacting with our library to use CoreSight STM. This is then compared to the execution with no tracing at all. In the same manner, the trace consumer is either LTTng's session daemon, or our custom program to retrieve data from the ETB, decode it and store it to disk. It is important to note that these results show intentionally high tracing overhead, because we want worst-case

results. For real-life programs, overhead is much lower, typically a few percent [15].

The benchmarks were run on a Pandaboard and showed that LTTng-UST's tracing time can be significantly reduced when using CoreSight STM and ETB. Here, we use a simple tracepoint type with a 24-bit integer payload. Figure 4.3 presents the average execution time of iterations when looping 10^7 times, with our test program recording a tracepoint at each iteration. We show results for the untraced program (a simple loop iterating a volatile variable), the same program traced with LTTng-UST, and then traced using STM and ETB. If both case, the LTTng tracing delay is reduced by approximately 91% when using hardware-assisted tracing.

We then measured tracing performance with respect to tracepoint types, i.e. the tracepoints payload length. We ran the same programs with a fixed number of iterations (10^6) but with payload varying from 3 to 100 bytes. The results are shown in figure 4.4. Using the STM and ETB hardware modules is efficient for small messages, but does not scale well for payloads bigger than 60 bytes. These results highlight the time taken by LTTng-UST to synchronize with the trace consumer with memory barriers, but also its ability to take advantage of the cache when trace payloads get longer.

Since most tracepoints typically used with LTTng do not exceed a few bytes, using CoreSight STM and ETB is a significant improvement to the low-intrusiveness of LTTng. Moreover, the timestamping provided by CoreSight is cycle-precise, which is not the case with LTTng on every platform.

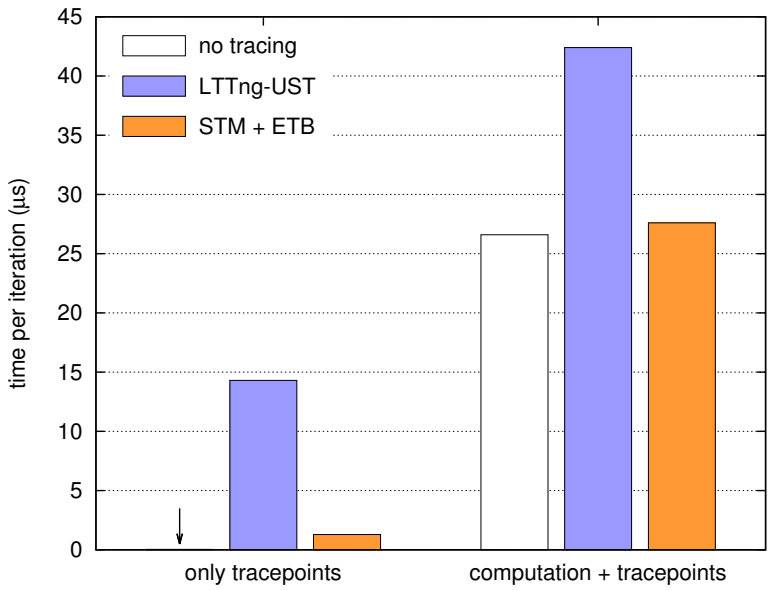


Figure 4.3 Average execution time of programs traced with LTTng-UST, with hardware (STM + ETB), and not traced. We present a program looping, first with only a tracepoint in each iteration, and then the same program performing real calculation in each iteration.

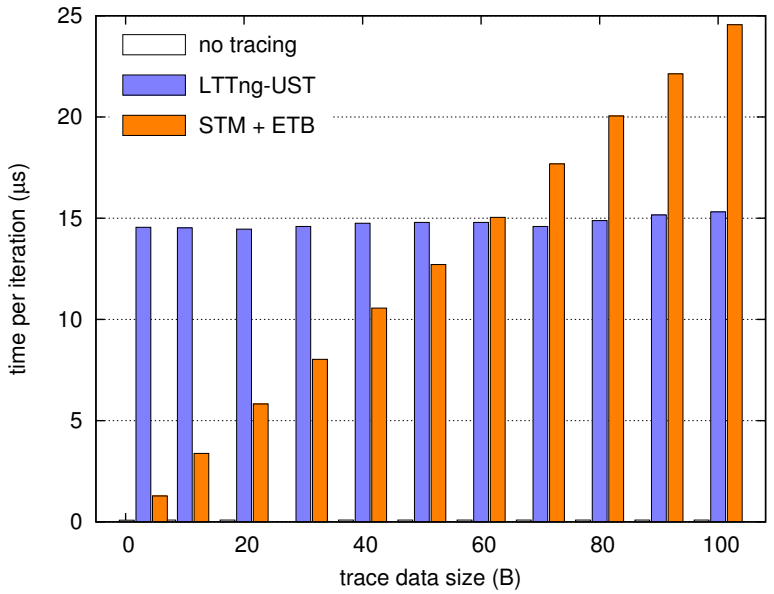


Figure 4.4 Average execution time of programs traced with LTTng-UST, with hardware (STM + ETB), and not traced. We present tracepoints with different payload size.

4.5 Using execution path tracing hardware

Some hardware debugging components provide the ability to save the execution path, i.e. the sequence of addresses for executed instructions. When timestamped, this information is sufficient to trace the complete execution of a program, or a section thereof, and constitutes a superset of the program location information conveyed by tracepoints. Nonetheless, tracing infrastructures like LTTng have the ability to associate a payload, and even a context, with the tracepoints (for instance, the thread ID or the number of encountered page faults), which is not available with hardware tracing components. However, if this context information is not needed, and only the sequence of reached tracepoints is of interest, software tracing can benefit from hardware assistance.

4.5.1 Embedded Trace Macrocell

Design

The ETM hardware module performs execution path saving in a highly compressed format, and without interfering with the traced program execution. Moreover, it can be enabled and disabled by triggers, including address matching and context ID matching. This allows tracing a specific process, in a specific address range. Moreover, the traced program does not need to be recompiled to be traced: only the virtual addresses of its symbols must be known.

We chose to use ETM to trace the program when it enters the portion of code corresponding to the tracepoint. For instance, if we want the call to function `foo` to be traced, we set up ETM to start when the program is at the address of `foo`, and stop at the same address + 4. ETM is also configured to trigger only when the interesting process is running. Due to hardware limitations, this design only allows one tracepoint, with no payload. If several events need to be traced, a full execution trace or a mix of pure-software and hardware tracepoints can be used.

We implemented this design with two programs: a trace producer, which represents the traced program; and a trace consumer, whose role is to regularly drain the ETB and save its contents to disk for further decoding. The trace producer enables the ETM by communicating with the kernel, via an entry in `sysfs`. We patched the Linux kernel to enable full configuration of the ETM, especially the address range selection and context ID tracking [61]. This patch has been sent to the Linux Kernel Mailing List. Once the ETM is activated, it is configured to trigger at the address of a simple function that does an arithmetic computation. The test program, after activating the ETM, enters a loop calling this function at each iteration. It is thus equivalent to the other test cases studied.

Results

This implementation is compared to two others: the first one is the same program but untraced; the second one has hardware tracing replaced by a call to LTTng-UST's `tracepoint` in each iteration (and the hardware trace consumer replaced by LTTng's session daemon). We evaluated the tracing time overhead by running these on a Beagleboard. Figure 4.5 presents the execution time of our three benchmark programs in three different configurations: recording only tracepoints, performing some extra computation in each iteration, and performing even more computation. Once again, it is important to note that these results show the overhead in the worst-cases, because our benchmark programs do almost nothing but recording tracepoints.

First, it is noticed that, for very high tracing frequencies, events are lost by the ETM. However, this situation is infrequent: it happens when events occur more often that 10^5 times per second. Apart from cases with lost events, results show that using ETM and ETB reduces the time overhead from 30% to 50% when compared to LTTng-UST. This demonstrates the performance benefits of using dedicated hardware to trace a given location in a program, in addition to the fact that there is no need to recompile the traced program. However, this solution only provides the time (although cycle-precise) when the program reaches a specific point, and no other information. Also, its most significant drawback is that it only allows tracing a few points in the program (because the number of available address triggers in ETM remains very limited). Configuring the ETM to trace the whole program is possible, but every branch would be recorded and a significant tracing overhead would be incurred.

4.5.2 Branch Trace Store

Design

The Branch Trace Store (BTS) registers, included in most newer x86 processors from Intel, allow saving every branch taken by the CPU to an area in main memory. Each branch is stored as a couple (origin address, destination address). The area starting address and size is user-definable, but cannot be configured to limit tracing to a specific process or address range. BTS does not timestamp records either. For these reasons, software has to spend more time for tracing control and synchronization: enable or disable it on context switches (to trace only one process), separately collect timestamps, and drain the buffer to save data to disk.

With BTS, every branch of the program is stored. The advantage of this behavior is that it provides much more information than the use of sparse tracepoints. Its drawback is a significant perturbation of the traced process execution, due to frequent memory accesses

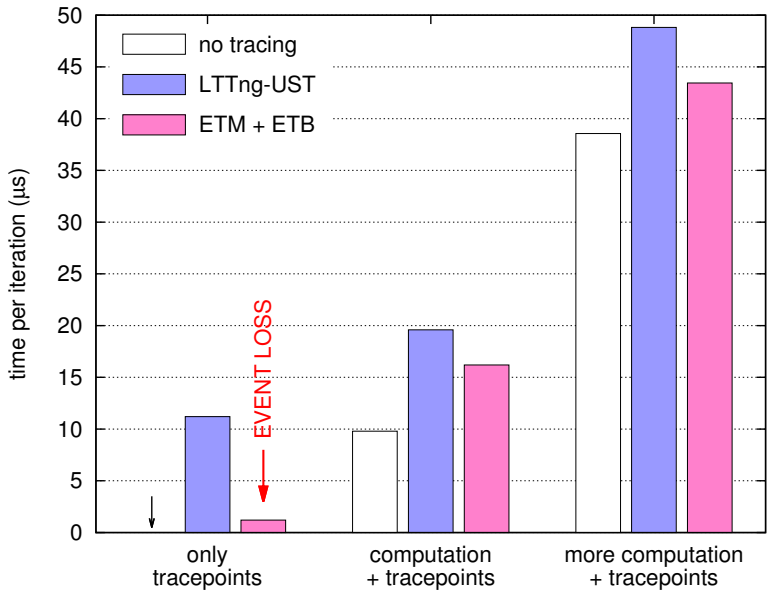


Figure 4.5 Average execution time of programs traced with LTTng-UST, with hardware (ETM + ETB), and not traced. We present programs that do a loop where each iteration performs either only tracing, or tracing plus computation (with more time spent on computation in the third program).

(to store branches) and related cache usage.

We wanted to know if the use of BTS, although known to have a significant performance impact, could do better than pure-software tracing, which is also very intrusive when used to trace the complete execution path. For this purpose, we designed test programs meant to be traced either with LTTng, or using BTS. In order to have comparable results, we needed to devise programs that trigger BTS branching records only where wanted, that is at the place where we would insert tracepoints. For this purpose, our benchmark programs are just loops performing arithmetic computations (to simulate real software activity): at the end of each iteration, the looping branch is recorded. The LTTng-traced version of these programs contains a `tracepoint` at the end of each iteration. Finally, in order to measure the effect of tracepoints sparsity, we varied the length of the loop computation. This changes the relative frequency of branches in the program.

Re-implementing Perf

We modified the Linux kernel to handle BTS more efficiently [60]. The default behavior (illustrated in figure 4.6) was to set up one buffer per CPU, and to copy its whole content to a larger place in RAM every time a buffer full interrupt or a context switch occurs. Then, the user-space tracing daemon would copy the contents of this intermediate buffer to disk.

The trace file being opened without the `O_SYNC` flag, writing to disk is not synchronized and data is possibly copied once more to a temporary buffer.

To avoid the time-wasting multiple copies in RAM (especially the one performed during the handling of an interrupt), we re-implemented this Linux kernel section to use ring-buffers. BTS is then configured to store entries in a sub-buffer; when it is full, BTS is reconfigured to use the next sub-buffer in the ring-buffer. This way, there is no urgent need to copy the BTS data to make room: this data can be saved to disk later by a kernel task. The size of the sub-buffers is the same as in Perf (64 KiB), their number was chosen to allocate a bit less resources than Perf does in its default implementation, that is 16 kernel-space pages plus 128 user-space pages per core (576 KiB per core for original Perf, 512 KiB in our implementation). Because the trace data does not transit through user-space before being stored, we use the term "splice" for our design. It is illustrated in figure 4.7.

Results

We measured the overhead induced by tracing with LTTng-UST, with BTS using the regular Perf interface, and with BTS using our modified kernel interface ("splice"). The results for programs with various branching rates are shown in figure 4.8. To give the reader an idea of the branching rates presented here, we measured as a reference the rates for common programs: `md5sum`: 7.4×10^6; `sha256sum`: 3.7×10^7; `gcc`: 6.70×10^8. These values, given in branches per second, were measured on the desktop machine described in section 4.3.

First, this experiment once more shows that the tracing overhead highly depends on the tracing frequency. Bars on the left show the overhead for programs that do nothing but recording tracepoints. Secondly, we compare the different tracing solutions overhead to LTTng, our reference tracer. The experiment reveals that using BTS through the regular Perf interface incurs between 30% and 60% time overhead, when compared to tracing with LTTng. This highlights the heavy bus usage and the double copy done by the Branch Trace Store system in its original Linux implementation. However, when using our "splice" design (in a kernel patched to use ring-buffers and avoid a double copy), BTS tracing performs much better than the original. It even overtakes LTTng with an overhead reduced by 10% to 45%.

These results highlight the limits of Perf's handling of BTS: multiple copies inducing a heavy bus usage that competes with the traced program's execution. Our re-implementation use ring-buffers to avoid multiple copies, resulting in a performance enhancement that makes BTS lighter than LTTng. Still, BTS remains costly for a hardware mechanism, due to a large trace volume (24 bytes per branch). The reader should note that our benchmark is valid for programs that record tracepoints at each branch, which is not often the case with higher level event tracing.

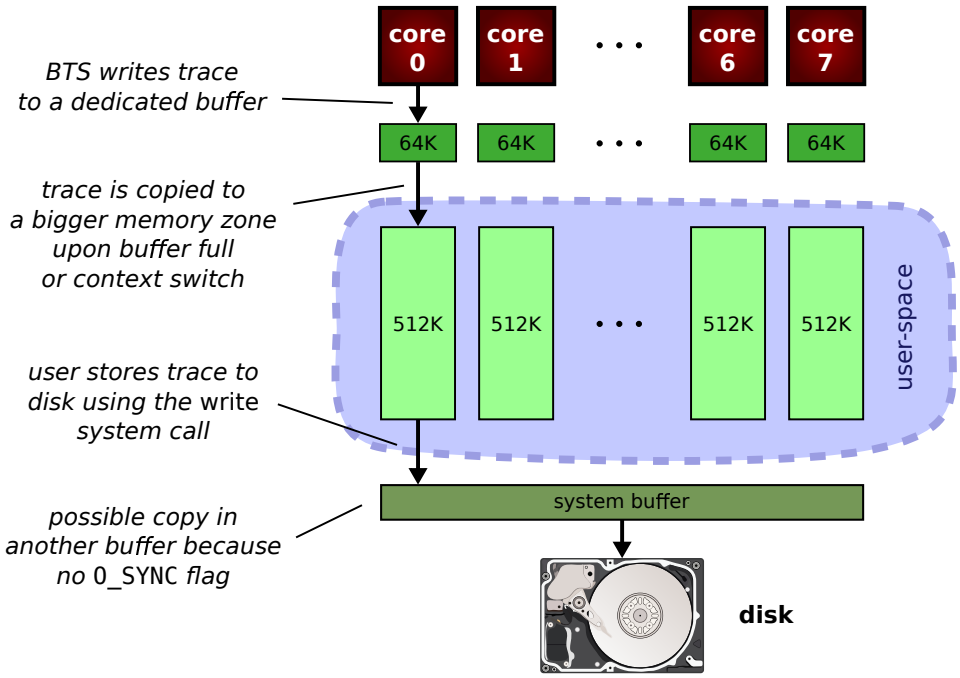


Figure 4.6 Operation of Perf and BTS in the original implementation

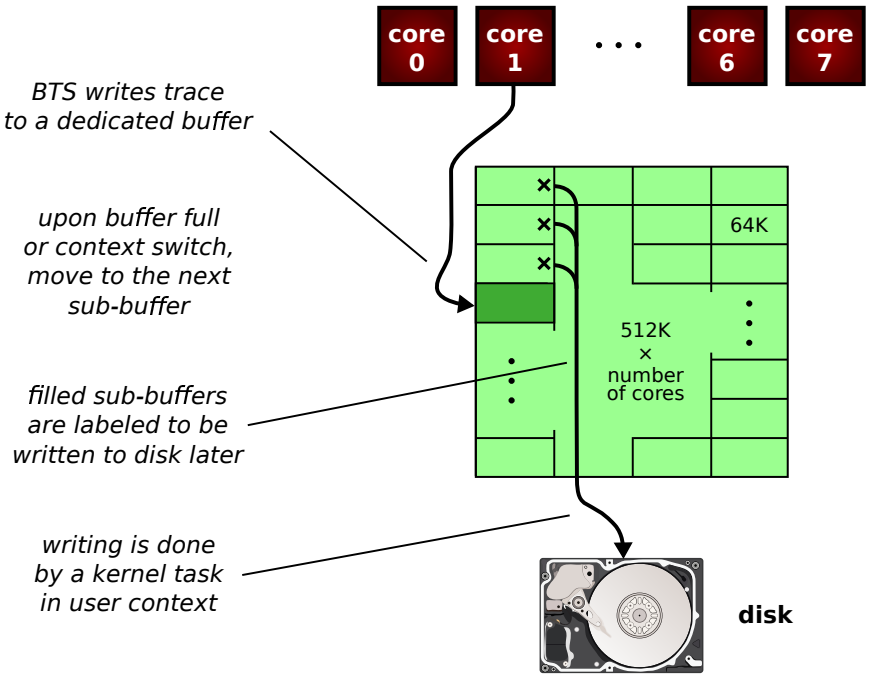


Figure 4.7 Operation of Perf and BTS in our new “splice” implementation

Figure 4.8 Time overhead of tracing when using LTTng-UST or Intel BTS hardware module, for programs with events traced at various frequencies. BTS results are presented for the original Perf implementation, and with our "splice" modification to avoid a multiple copies.

4.6 Conclusion and future work

We have presented solutions to take advantage of various hardware debugging modules when doing software tracing, and compared their overhead to LTTng, a performance oriented tracer for Linux. The hardware modules studied belonged to two categories: software writes timestampers and program tracers.

We used STM from the former category, to instrument programs without needing to synchronize and compute a timestamp when recording a tracepoint. We showed a 10 times decrease in time overhead when tracing a user-space program on a Pandaboard. We then studied program tracers, i.e. hardware modules to record the flow of executed instructions. These do not provide the same information as tracers do but can be suited for tracing, depending on one's needs. On ARM platforms, ETM offers good efficiency because of its trace compression and triggering capabilities: tracing a program takes 30% to 50% less time when hardware-assisted. On Intel x86, the BTS registers are not adapted to event recording, mainly because they lack automatic enabling/disabling with address matching and trace compression. However, they can be used for this purpose by tracing every branch. Using BTS for recording user-space tracepoints on an Intel Core i7-3770 is 30% to 60% slower than using LTTng with the standard implementation, whereas our kernel modification with ring-buffers performs up to 45% faster.

The significant performance increase offered by some of our solutions, especially STM, should lead to an integration into LTTng in the near future. We also plan to extend this work to other interesting hardware features, including Freescale processors (which provides a program tracing functionality), and Intel Processor Trace (PT) [47], a new hardware tracing infrastructure that Intel will embed in its future processors. Intel PT is promising because of its triggering and filtering capabilities, similar to CoreSight ETM.

CHAPITRE 5

RÉSULTATS COMPLÉMENTAIRES

L'article présenté dans le chapitre précédent fait état de nos résultats les plus significatifs, mais n'inclut pas les études intermédiaires ayant guidé nos expériences. Ce chapitre rassemble ces résultats complémentaires.

5.1 Double copie dans l'ABI Perf avec BTS

L'outil Perf permet d'utiliser la fonction *Branch Trace Store* des processeurs x86 d'Intel pour faire du traçage d'exécution (voir l'exemple donné dans la section 2.3.1). Néanmoins, cette trace est obtenue avec un fort surcoût : nous avons mesuré des augmentations du temps d'exécution entre 400% et 1150%, alors qu'Intel mentionne un surcoût de 20% à 100% [45]. Nous avons donc voulu savoir à quoi était due cette différence, puis essayé de modifier l'implémentation originale pour corriger ce défaut.

5.1.1 Étude de l'implémentation originale

Afin de trouver pourquoi l'utilisation de BTS par Perf est anormalement longue, nous avons étudié le code responsable du traçage. Celui-ci est dans le noyau Linux, et est déclenché par Perf *via* l'appel système `perf_event_open`. Une fois le traçage configuré et activé par Perf, la production de trace se fait selon les étapes suivantes (illustrées dans la figure 4.6) :

A - Les registres BTS de chaque processeur sont configurés pour écrire la trace dans un tampon de 16 pages (64 Kio), propre à ce cœur.

B - Lorsque ce tampon est plein (ou lors d'un changement de contexte), le contenu du tampon est copié dans une plus grande zone mémoire de 512 Kio. Ceci est fait pendant le traitement de l'interruption. Après cela, le processeur peut à nouveau écrire dans le petit tampon qui lui est affecté.

C - Régulièrement, Perf lit les données de la grande zone mémoire pour les enregistrer dans un fichier. Ce dernier est ouvert avec les drapeaux `O_CREAT|O_RDWR|O_TRUNC` mais pas `O_SYNC`, ce qui veut dire que l'écriture n'est pas synchronisée : le système peut copier les données dans un nouveau tampon pour rendre la main au programme appelant plus rapidement.

En instrumentant le noyau Linux de manière à ne pas effectuer certaines des étapes ci-dessus, nous avons mesuré la responsabilité de chacune de celles-ci dans le surcoût de Perf. Le temps ajouté par chaque étape sur l'exécution des logiciels `md5sum` et `sha256sum` est présenté dans le tableau 5.1. La taille de la trace étant directement liée à la fréquence de branchement d'un programme, nous incluons cette donnée dans le tableau.

Les résultats obtenus montrent que chaque étape induit un fort surcoût. Celui-ci est inévitable pour l'activation de BTS et l'écriture de la trace sur le disque. En revanche, la double copie effectuée par le noyau Linux peut être évitée. Selon les chiffres du tableau, on peut espérer au minimum une réduction d'un tiers du surcoût de traçage.

5.1.2 Ré-implémentation sans double copie

Nous avons imaginé une modification de Perf pour utiliser BTS sans faire les copies coûteuses de l'implémentation actuelle. Cette modification ne remplace pas l'implémentation originale mais ajoute une option « splice », dont l'activation change la manière dont BTS est utilisé. Lorsque le mode « splice » est sélectionné (*via* un appel système `ioctl`), la trace est générée et traitée selon notre méthode, qui écrit les données directement sur le disque sans passer par l'espace utilisateur. Ainsi, lorsque le but de l'utilisateur est de sauvegarder la trace sur le disque (ce que Perf fait par défaut), il peut activer le mode « splice » et profiter de notre ré-implémentation. Au contraire, s'il désire accéder à la trace directement en mémoire, il n'a qu'à rester dans le mode original.

L'idée guidant notre ré-implémentation est d'éviter deux copies :

— la copie du petit tampon de 64 Kio vers celui de 512 Kio, qui est faite soit pendant le traitement d'une interruption, soit pendant un changement de contexte et ralentit de ce fait l'ensemble du système d'exploitation ;

— la copie potentielle des données dans la zone mémoire de 512 Kio vers un tampon système lors de l'appel à `write`.

Pour cela, nous remplaçons les tampons de chaque processeur par un seul tampon circulaire (*ring-buffer*) commun à tous les cœurs. Ce tampon circulaire est constitué de sous-tampons

Tableau 5.1 Surcoût décomposé des étapes du traçage avec Perf et BTS. Les valeurs indiquées représentent le temps ajouté à l'exécution totale par rapport à une exécution non-tracée.

programme tracé	taux de branchement	trace	activation de BTS	copie du noyau	écriture sur le disque	total
`md5sum`	7×10^6 br/s	77 Mio	+167%	+96%	+104%	+368%
`sha256sum`	32×10^6 br/s	1,1 Gio	+425%	+467%	+325%	+1217%

(*sub-buffers*) qui auront le même rôle que les petits tampons de l'implémentation originale : accueillir la trace directement générée par BTS. Pour cette raison, leur taille reste la même (64 Kio). La différence est la suivante : en cas de débordement ou de changement de contexte, au lieu de vider le tampon instantanément pour faire de la place, on reconfigure BTS pour écrire dans le prochain sous-tampon libre. Ainsi, il n'est pas nécessaire de copier les 64 Kio de trace pour les sauvegarder. Ils seront écrits sur le disque plus tard par une tâche noyau, et surtout en dehors du traitement de l'interruption (qui est sensé se faire le plus rapidement possible). La taille totale du tampon circulaire a été choisie pour allouer autant de ressources que l'implémentation standard (et même un peu moins). Alors que cette dernière allouait 512+64 Kio par processeur, « splice » utilise 512 Kio multiplié par le nombre de processeurs.

Notre nouvelle implémentation est illustrée dans la figure 4.7, et disponible sur internet [60].

5.2 Taux de branchement

Dans le cadre de nos expérimentations sur le système de traçage d'exécution *Branch Trace Store* (BTS), nous avons été amenés à mesurer le surcoût d'un tel système sur différentes classes de programmes. La trace étant constituée de l'historique des sauts dans l'exécution, le surcoût est directement lié au taux de branchement du programme tracé, c'est-à-dire le nombre de branchements suivis par le processeur par unité de temps.

5.2.1 Taux de branchement de programmes usuels

Nous avons mesuré le taux de branchement pour différents programmes usuels. L'objectif n'était pas d'enregistrer des valeurs précises, mais d'obtenir un ordre de grandeur afin de cibler nos expériences ultérieures sur des cas typiques. Nous avons choisi des programmes représentatifs de différentes classes de logiciels : forte utilisation de l'unité de calcul, dépendance des entrées-sorties, accès intensif au disque, utilisations hybrides.

Pour chaque logiciel, la méthodologie utilisée a été la suivante :

— mesure de la durée d'exécution par moyenne sur plusieurs exécutions non-tracées ;

— calcul du nombre de branchements suivis pendant toute la vie du processus grâce à l'outil Perf (commande `perf stat -e branches:u,cycles:u,task-clock`).

Ce découplage assure que l'influence de l'enregistrement du nombre de branchements n'affecte pas la durée d'exécution moyenne que nous mesurons.

Nous présentons les résultats dans le tableau suivant :

Tableau 5.2 Taux de branchement de programmes usuels

Logiciel	Taux de branchement
md5sum	$7,3 \times 10^6$ br/s
dd	$1,5 \times 10^7$ br/s
sha256sum	$3,7 \times 10^7$ br/s
grep	$1,0 \times 10^8$ br/s
gcc	$6,7 \times 10^8$ br/s

5.2.2 Programmes à taux de branchement arbitraires

Les programmes usuels présentés ci-dessus posaient un problème : il n'était pas possible de comparer le surcoût du traçage avec BTS à celui dû à LTTng. Cela aurait en effet nécessité de recompiler ces logiciels avec des points de trace LTTng à chaque emplacement du code source provoquant un branchement dans le code assembleur.

Pour remédier à cela, nous avons forgé nos propres binaires traçables aussi bien par BTS que par LTTng-UST. Il s'agit de paires de programmes au comportement identique :

— la version traçable par BTS est une simple boucle à l'intérieur de laquelle sont faits des calculs arithmétiques, ainsi seul le branchement en fin de boucle amenant à l'itération suivante est tracé par BTS ;

— la version traçable par LTTng reproduit les même calculs arithmétiques, mais ajoute un point de trace logiciel à la fin de chaque itération.

Ainsi, les deux programmes effectuent exactement le même travail, et produisent des éléments de trace aux mêmes endroits.

Nous rappelons que cette instrumentation n'est faite que pour comparer les performances des traçages avec LTTng et BTS avec des programmes similaires ; ceci ne pourrait pas être fait avec n'importe quel logiciel. La comparaison n'est valide que pour des programmes que l'on voudrait tracer en enregistrant chaque branchement, ce qui n'est pas la finalité d'un traceur d'événements logiciels comme LTTng.

5.3 Modification du noyau Linux pour un traçage avec ETM plus flexible

Un pilote pour le module CoreSight *Embedded Trace Macrocell* (ETM) existe dans le noyau Linux. Celui-ci crée des entrées dans le pseudo-système de fichiers SysFS pour interfacer le module de traçage d'exécution. Par exemple, l'écriture du caractère `1` dans le fichier `/sys/devices/etm/trace_running` active le traçage ; tandis que la lecture du fichier `/dev/tracebuf` donne accès à la trace brute. Un exemple de traçage avec ETM a été présenté dans la section 2.3.2.

Ce pilote présentait cependant de fortes limitations. Alors qu'ETM permet de concentrer le traçage sur le code associé à un *Context ID* spécifique (c'est-à-dire à un fil d'exécution, processus ou groupe de processus particulier), il n'était possible de tracer que le système d'exploitation. Similairement, ETM dispose de comparateurs d'adresses pour ne tracer que la ou les portions de code d'intérêt ; pourtant le pilote n'autorisait pas de cibler une gamme d'adresses à tracer.

Afin de pouvoir mener à bien nos expériences, nous avons modifié le noyau Linux pour rendre la configuration d'ETM plus souple. Nos correctifs incluent de nouvelles entrées dans SysFS, qui permettent de limiter le traçage à un processus en particulier (caractérisé par son identifiant de processus *PID*, lui-même écrit dans une partie du registre *Context ID*), et à un intervalle d'adresses ciblant une portion de code d'intérêt. Ce correctif (*patch*) a été envoyé à la liste de diffusion du noyau Linux (LKML) et est disponible sur internet [61].

Cette modification nous a permis de tracer des points spécifiques dans des programmes en espace utilisateur, de manière similaire à LTTng-UST. Cela a rendu possible notre comparaison des performances d'ETM et LTTng.

5.4 Rétro-ingénierie du format STP

Comme mentionné dans la section 2.1.2, les traces produites par l'aide matérielle au traçage d'événements STM sont encodées dans un format fermé. Ces deux modules STM se conforment au *System Trace Protocol* (STP) [41], qui spécifie le comportement que doivent adopter les producteurs de trace, et définit les types de messages constituant une trace STM. Ceci est vrai aussi bien pour le *System Trace Module* de Texas Instruments (STP version 1) que pour la *System Trace Macrocell* d'ARM (STP version 2). La documentation du STP n'étant pas disponible publiquement, il ne nous a pas été possible d'y accéder pendant notre recherche.

On pourrait penser que le décodage des traces n'est pas utile pour la seule mesure des performances du système matériel, pourtant il est nécessaire pour plusieurs raisons. Tout d'abord, les messages de trace sont multiplexés dans d'autres messages (alignement, synchronisation, numéro de canal) : il est requis de reconnaître les messages de trace ne serait-ce que pour les compter. Ce décompte permet de vérifier que les point de traces sont bien enregistrés par le système. De plus, le tampon de collecte des données de trace est circulaire : sa lecture n'est pas arrêtée par un marqueur particulier. Il faut alors être en mesure de reconnaître la fin de la trace. Enfin, le décodage du format STP permettra une intégration du traçage assisté par matériel dans LTTng.

Nous avons effectué une rétro-ingénierie du format STP afin d'être capable de décoder les

traces générées par STM. Nous ne dévoilerons pas les détails de l'encodage de la trace, mais les étapes que nous avons effectuées pour arriver à comprendre le format et à en écrire un décodeur.

Dans un premier temps, nous avons étudié un format de trace similaire à STP et dont la documentation est fournie : celui de ITM (voir la section 2.1.2). Ce dernier détaille le format des « paquets de trace » que l'on peut retrouver dans le tampon de sortie. Sont entre autres présentés des paquets de synchronisation, de débordement ou d'estampilles de temps. Ceux-ci ne se retrouvent pas dans les traces STP (par exemple, les estampilles de temps sont intégrées aux paquets de données, et l'unité de taille n'est pas d'un octet), pourtant leur structure se révèle utile pour comprendre les paquets STP. Nous avons ensuite identifié 9 types de paquets différents, mais nous soupçonnons l'existence d'autres. Enfin, le codage des estampilles de temps étant fait d'une manière très particulière, nous avons conçu des programmes pour envoyer des messages de trace à des intervalles arbitraires et comparé ceux-ci aux estampilles produites. Après avoir tracé plusieurs graphiques de façon à reconnaître une logique, nous avons écrit un modèle qui semble correspondre, au moins pour des durées inférieures à la seconde.

Par la suite, nous avons écrit une bibliothèque de décodage mettant en application nos découvertes, qui est utilisée par les programmes de test et de vérification que nous utilisons dans nos expériences.

CHAPITRE 6

DISCUSSION GÉNÉRALE

Ce chapitre revient sur les résultats obtenus aux chapitres 4 et 5, discute leur impact et propose une solution d'intégration dans le traceur LTTng-UST.

6.1 Impact des résultats

Notre étude a présenté plusieurs propositions pour alléger l'impact du traçage de logiciels, en utilisant divers systèmes matériels. Nos tests ont montré que parmi celles-ci, certaines apportent de fortes améliorations sur les performances, notamment celles utilisant le *System Trace Module*, la *Embedded Trace Macrocell*, ou encore le système *Branch Trace Store*. Cette section envisage l'impact de ces résultats sur l'écosystème des solutions de traçage logiciel.

6.1.1 Traçage d'événements avec STM

Une de nos principales contributions est présentée dans la section 4.4. Il s'agit d'une méthode pour utiliser l'infrastructure matérielle *System Trace Module* (STM) pour accélérer l'enregistrement de points de trace dans des programmes en espace utilisateur.

Pour des points de trace simples (charge utile de quelques octets, ce qui est généralement le cas), le surcoût dû au traçage est réduit de manière significative : environ dix fois plus faible sur notre plateforme de test. De plus, la précision des estampilles de temps est accrue par le module STM. En contrepartie, ce système n'est utilisable que sur les plateformes à architecture ARM embarquant l'infrastructure STM. De plus, il n'est pas compatible avec d'autres systèmes logiciels utilisant le module (par exemple pour du débogage), à moins de définir et d'implémenter un moyen de communication et d'arbitrage pour l'utilisation des ressources communes. Enfin, ce type de traçage n'est pas entièrement matériel car il demande une instrumentation : de la même manière que pour le traçage logiciel, un appel de fonction doit être inséré dans le code source du programme tracé. Ceci exclut le traçage d'événements pour un logiciel dont la source n'est pas accessible.

Le traçage d'événements par des outils logiciels peut donc être amélioré sur les plateformes bénéficiant de STM, dans le sens où leur surcoût serait grandement réduit. La version du *Linux Trace Toolkit* pour les programmes en espace utilisateur (LTTng-UST) pourrait utiliser ce système, grâce à des modifications comme celles décrites dans la section 6.2.

6.1.2 Traçage d'événements avec ETM

Un des résultats de cette recherche est un prototype de traceur d'événements utilisant du matériel de traçage d'exécution. Dans la section 4.5.1, nous expliquons comment nous configurons l'infrastructure matérielle *Embedded Trace Macrocell* (ETM) pour enregistrer les points de trace correspondant à des événements particuliers.

Nos expériences montrent que sur une plateforme Beagleboard, le surcoût est réduit de 30% à 50% par rapport à LTTng et qu'encore une fois, la précision des estampilles de temps est améliorée. De plus, le traçage avec ETM ne nécessite pas de recompilation du programme à tracer : seules les adresses des points de trace sont requises, cette information est aisément accessible si les symboles sont présents dans le binaire. Cependant, comme pour STM, ces résultats ne s'appliquent qu'aux plateformes à architecture ARM embarquant le module ETM. L'utilisation de notre prototype requiert aussi un système de communication et d'arbitrage si d'autres logiciels ont besoin d'utiliser les modules matériels en question. Enfin, ETM ne dispose généralement que de quelques comparateurs d'adresse, ce qui limite notre système à quelques points de trace. Il n'est pas possible d'enregistrer un nombre arbitraire d'événements différents, ce qui est à nos yeux une limitation majeure.

Pour cette raison, nous pensons que l'intégration de notre système dans un traceur logiciel comme LTTng ne vaut pas la peine. Néanmoins, notre code est disponible et peut être réutilisé pour une situation spécifique.

6.1.3 Traçage d'exécution avec BTS

Dans les sections 4.5.2 et 5.1, nous détaillons une méthode alternative pour la récupération des traces d'exécution avec l'outil Perf et le système *Branch Trace Store* (BTS) d'Intel. Celle-ci consiste en des modifications dans le code du noyau Linux pour éviter les copies multiples de la trace avant son écriture sur le disque. Le résultat direct est une diminution du surcoût de traçage entre 30% et 65%.

Malgré cette nette amélioration des performances, nos résultats doivent être pris avec certaines réserves. D'abord, ce système suppose la présence des registres BTS sur le processeur, ce qui le limite aux systèmes à architecture x86 d'Intel. Ensuite, le système *Branch Trace Store* s'applique mal au traçage d'événements, notamment à cause du fait que BTS enregistre tous les branchements à bas niveau, et non des occurrences ponctuelles.

Ces raisons excluent BTS de la liste du matériel intéressant pour le traçage d'événements (ce qui était notre but premier) ; pour autant, notre travail peut être utile pour d'autres utilisations. En effet, les modifications proposées améliorent les performances du traçage d'exécution, et pourraient être intégrées au logiciel Perf pour servir à d'autres utilisateurs.

Notre travail est disponible publiquement [60], il s'agit de correctifs à appliquer au code source de Linux pour ajouter une option « splice » au système `perf_event_open`.

6.2 Intégration dans LTTng

Cette section discute les problématiques liées à l'intégration d'un système de traçage assisté par matériel dans LTTng-UST, la composante pour les programmes en espace utilisateur de LTTng. Elle ne traite que notre proposition d'utilisation du matériel avec STM, qui est la plus apte à être utilisée dans un traceur logiciel.

6.2.1 Utilisation des ressources

Les infrastructures matérielles utilisées par notre implémentation sont STM pour la génération des traces et le tampon ETB pour la récupération ultérieure des données. Tous les deux sont faits pour être accédés concurremment : STM est accessible *via* une centaine de canaux différents, tandis que ETB centralise les traces issues de plusieurs modules matériels ayant des fonctions différentes. Il est donc possible qu'un autre logiciel utilise ces ressources matérielles pour un usage différent. Ceci n'empêche pas LTTng de se servir de STM et ETB, mais demande un système d'arbitrage et de synchronisation pour accéder à ces ressources partagées.

Le meilleur système d'arbitrage serait à nos yeux un pilote pour ces éléments matériel, sous la forme d'un module noyau qui offrirait à l'espace utilisateur un moyen de réserver des canaux STM et d'obtenir la trace correspondante *via* un descripteur de fichier. Ainsi, le noyau Linux serait responsable de l'attribution des ressources et empêcherait différentes applications de corrompre des données en tentant d'accéder simultanément aux mêmes structures matérielles.

Cependant, l'utilisation d'un module noyau est difficile dans certaines conditions, et demande une recompilation à chaque mise à jour du noyau. Si elle n'est pas possible, une autre solution est de configurer STM et ETB et d'accéder à leurs ressources *via* des correspondances mémoires (*mapping*) en espace utilisateur. C'est la méthode que nous avons utilisée. En revanche, ce procédé n'est pas à l'abri de collisions dues à un autre logiciel de débogage. De plus, il nécessite un traceur s'exécutant avec les droits *root*.

6.2.2 Choix dynamique de l'utilisation du matériel dédié

Même si l'utilisation de STM accélère le traçage logiciel, il faut laisser au traceur la possibilité d'enregistrer les événements par la méthode purement logicielle. Tout d'abord, parce que l'utilisateur peut le demander (par exemple pour effectuer des mesures de banc

d'essai sur des méthodes comparables). L'autre raison est simple : STM n'est pas disponible sur toutes les plateformes.

Nous pensons donc que les points de trace dans les programmes instrumentés ne doivent pas être modifiés : elle resterait un simple appel à une fonction `tracepoint`. Le changement se ferait alors *via* l'interface de contrôle du traceur, pour activer l'accélération matérielle là où elle est disponible. Par exemple, la commande suivante pourrait être utilisée pour activer l'utilisation de STM pour les points de trace :

```
lttng enable-hardware stm
```

Le procédé que nous venons de proposer autorise une grande flexibilité. Néanmoins, une autre implémentation moins souple donnerait peut-être de meilleures performances en termes de surcoût : la compilation statique du contrôle de STM dans les points de trace. Ainsi, le choix entre traçage logiciel ou aidé par matériel serait fait à la compilation. Un binaire instrumenté donné serait donc dépendant de la plateforme, mais pourrait enregistrer des événements en exécutant moins d'instructions, et donc en utilisant moins de temps.

6.2.3 Décodage des traces

Comme mentionné dans la section 5.4, les traces générées par STM sont au format STP. Ce type d'encodage est fermé et les données à l'intérieur fortement altérées, ce qui le rend incompatible avec le format général CTF, pourtant très souple. Ainsi, les traces produites par STM ne sont pas lisibles telles quelles par les outils de la suite LTTng : il est nécessaire de les décoder pour les convertir en un format compatible avec le CTF.

Cela apporte un autre problème : la lourdeur du décodage. Ce dernier demande de multiples copies de petits blocs de données par forcément alignés sur des mots ou des octets, et une lecture en sens inverse (de la fin du flux vers le début). La conséquence est que ce format demande des ressources en calcul et en mémoire non-négligeables. Deux alternatives sont alors possibles.

1. Une méthode « à la volée » : la conversion immédiate des traces lors de leur lecture depuis le tampon ETB. Ceci serait fait par le consommateur de trace, qui utiliserait par conséquent plus le processeur et la mémoire, et pourrait influer davantage sur le programme tracé.
2. Le stockage des données « telles quelles » : aucune conversion n'est faite pendant l'enregistrement, et c'est au visionneur de trace de le faire. Ceci peut être fait *a posteriori*, ou sur une autre machine.

Nous préférons la deuxième méthode, qui risque moins de nuire aux performances de LTTng, traceur axé sur la faible intrusivité. Elle pourrait être implémentée *via* un module

(*plugin*) de décodage dans le visionneur de trace Babeltrace [38].

6.2.4 Architecture

D'un point de vue architectural, LTTng-UST trace les événements de la manière suivante :

— la bibliothèque `liblttng-ust` fournit la fonction `tracepoint` que les programmes instrumentés appellent pour sauvegarder l'occurrence d'un événement ;

— le processus `consumerd` (démon consommateur) « consomme » la trace et l'enregistre dans un tampon interne.

L'intégration du support pour STM demande une couche d'abstraction au niveau du producteur de trace (la bibliothèque `liblttng-ust`) pour que la création d'un point de trace générique puisse se faire sous plusieurs formes : purement logicielle, aidée du matériel STM, ou encore aidée du matériel d'un autre type (pour les futures améliorations, par exemple Intel PT).

Le consommateur de trace (processus `consumerd`), quant à lui, doit être modifié pour être capable d'écrire un nouveau type de trace brute (au format STP), en utilisant un nouveau fichier de trace par exemple.

CHAPITRE 7

CONCLUSION ET RECOMMANDATIONS

Ce chapitre conclut notre mémoire en synthétisant nos travaux de recherche. Il aborde aussi les limitations de nos résultats, ainsi que les améliorations qui peuvent être envisagées dans le futur.

7.1 Synthèse des travaux

Dans ce mémoire, nous avons étudié la problématique de la réduction de l'impact du traçage d'événements en utilisant du matériel dédié. L'objectif était d'utiliser les infrastructures matérielles de débogage, aujourd'hui largement présentes dans les processeurs, pour améliorer les performances des outils de traçage et rendre leur utilisation encore plus transparente. Pour répondre à cet objectif, nous avons suivi une méthodologie en quatre étapes.

La première consistait à dresser une liste des modules matériels potentiellement intéressants, et étudier les caractéristiques techniques de chaque infrastructure matérielle pour déterminer les possibilités en termes de traçage. Ceci a été fait au cours de notre revue de littérature, qui a couvert de nombreuses solutions matérielles de débogage, de profilage et de traçage pour nous permettre de retenir trois candidats. Il s'agit de *System Trace Module* (STM) et *Embedded Trace Macrocell* (ETM) pour les plateformes à architecture ARM, et de *Branch Trace Store* (BTS) pour les processeurs x86 d'Intel. D'autres candidats nous ont semblé prometteurs : les systèmes *Program Trace* et *Data Acquisition Messages* sur les processeurs Freescale. Malheureusement nous n'avons pas retenu ces deux derniers, faute de documentation nous permettant de les utiliser.

Dans une deuxième étape, nous avons imaginé des implémentations pour tirer profit des capacités matérielles de chaque système retenu. L'objectif était d'obtenir une trace d'événements comparable à celle produite par un traceur entièrement logiciel comme LTTng. Pour cela, nous avons développé un outil logiciel pour chacune de nos trois infrastructures candidates, de manière à les configurer pour produire une trace d'événements. L'outil de configuration était à chaque fois accompagné d'un outil logiciel de récupération des traces, pour vérifier la validité de données produites et faire nos tests dans des conditions de traçage réel. Dans le cas de STM, notre solution consistait à utiliser les capacités matérielles d'estampillage de temps automatique, du bus et du tampon dédiés pour acheminer nos traces. Dans les cas de ETM et BTS, nous avons détourné l'usage original du matériel (traçage

d'exécution) pour l'adapter à nos besoins (traçage d'événements). Ceci a été fait en faisant correspondre les adresses des événements à enregistrer avec les portions de code tracées par le matériel.

La troisième étape consistait à évaluer les différences de performance apportées par nos modifications. Nous avons utilisé LTTng-UST comme élément de comparaison, car à notre connaissance, il s'agit du traceur purement logiciel avec le moins d'impact sur l'exécution, ce qui fait de lui la meilleure référence. La grandeur mesurable pour comparer nos implémentations à notre traceur étalon est le surcoût temporel, c'est-à-dire le temps ajouté par rapport à une exécution non-tracée. Lors de l'utilisation de STM pour assister le traçage, le surcoût est largement réduit (environ -91%). En ce qui concerne ETM, le surcoût décroît significativement aussi (entre -30% et -50%), et le traçage ne nécessite pas d'instrumentation du logiciel. Enfin, le traçage avec BTS n'est comparable au traçage d'événements que sous des conditions très restrictives ; néanmoins il réduit aussi le surcoût (environ -45%). Une de nos contributions connexes est que la ré-implémentation de la façon dont Linux utilise BTS réduit le surcoût du traçage d'exécution jusqu'à 65%.

La dernière étape n'a été que partiellement atteinte. Il s'agissait d'intégrer les solutions présentant de bons résultats dans le traceur LTTng. Au lieu de les développer, nous avons proposé une méthode d'intégration dans l'architecture de ce traceur, en soulevant les problèmes majeurs posés par une lourde ré-implémentation comme celle-ci. Pour chaque problématique, nous avons apporté les éléments en faveur ou en défaveur de chaque choix. Nous croyons que l'intégration dans LTTng nécessitera la prise de décisions ainsi qu'un travail de développement supplémentaire, qui sortent du cadre de notre recherche.

7.2 Limitations des solutions proposées

Les solutions de traçage logiciel assisté par matériel permettent de réduire le surcoût du traçage. Elles sont néanmoins sujettes à des limitations.

La première de ces limitations est la disparité des infrastructures de débogage matériel. Il existe une multitude de ces circuits, conçus par différents fabricants ayant différents besoins. Des modules issus d'un même constructeur et ayant la même fonction sont parfois même complètement réinventés de génération en génération, comme en témoigne l'évolution des cellules de traçage sur ARM : ITM, puis STM et aujourd'hui STM-500. Malgré certaines normes pour encadrer les capacités offertes et standardiser le format des données de trace (comme Nexus ou CoreSight), la plupart des systèmes sont incompatibles entre eux. La conséquence de cette diversité est qu'on ne peut pas concevoir de système universel de traçage assisté par matériel. Les capacités offertes sur une plateforme donnée sont généralement

différentes sur une autre, ce qui implique le développement de solutions uniques pour chaque cas. Les résultats que nous avons obtenus avec STM, ETM et BTS ne seront probablement plus valables pour les prochaines versions des modules de débogage et de profilage d'ARM et Intel.

Une autre limitation concerne l'utilisation des modules matériels ETM et BTS. Nous les avons utilisés pour faire du traçage d'événements, alors que ce sont des systèmes de traçage d'exécution. Dans le cas d'ETM, nous avons pu détourner la fonction originale du module en le configurant pour ne tracer que les portions de code entre l'adresse d'un point de trace et cette même adresse + 4 octets, ce qui n'enregistre que ce point de trace. On arrive alors bien à du traçage d'événements, mais il n'est possible de définir qu'un nombre limité de points de trace, à cause du faible nombre de comparateurs d'adresses fournis par ETM (trois sur la Beagleboard). Pour ce qui est de BTS, l'usage est encore plus limité. En effet, il n'est pas possible de programmer le matériel pour limiter la trace à une zone donnée : si la fonctionnalité est activée, tout le code est tracé. L'utilisation de BTS est donc inadaptée au traçage d'événements épars.

Enfin, une dernière limitation affecte nos solutions, mais son occurrence est plus rare. Il s'agit des pertes dues au dépassement dans les tampons. Si trop d'événements sont enregistrés, ou si la zone mémoire où est enregistrée la trace n'est pas vidée à temps, certains points de trace peuvent être perdus. Dans le cas de STM et ETM, dont le tampon de collecte est ETB, ce cas de figure intervient quand le consommateur de trace ne suit pas le rythme du producteur. Pour BTS, cela arrive lorsque la tâche noyau responsable de la vidange du tampon circulaire n'est pas lancée à temps. Dans tous les cas, cette limitation peut être contournée en bloquant l'exécution du producteur de trace (le programme tracé) tant qu'il n'y a pas de place (par exemple avec un système de verrou bloquant ou une valeur `nice` élevée). Pour le cas d'ETM spécifiquement, il y a aussi un risque de perte des paquets de trace avant même qu'ils arrivent au tampon de collecte, si leur génération est trop fréquente. Cependant, ce phénomène est très rare et demande des conditions extrêmes : plus de 10^5 événements par seconde.

7.3 Améliorations futures

Plusieurs améliorations sont envisageables dans le futur. Tout d'abord, l'intégration de nos travaux dans LTTng donnerait une application concrète à notre recherche, au travers d'un outil véritablement utilisé.

Ensuite, notre recherche ouvre aussi la voie à l'utilisation d'autres infrastructures matérielles, déjà existantes ou à venir. Parmi les modules matériels déjà disponibles, ceux de Freescale semblent prometteurs : ils fournissent des capacités similaires à CoreSight STM et

ETM, qui nous ont donné de bons résultats sur les plateformes ARM. Le laboratoire DORSAL, où nous avons conduit nos travaux, est en contact avec les ingénieurs de Freescale pour obtenir de la documentation sur ces systèmes de traçage matériel.

Pour ce qui est des modules à venir, la fonction *Processor Trace* (PT) sur les processeurs d'Intel semble offrir une solution complète de traçage, pensée pour être à la fois flexible et peu intrusive. Enfin, on peut s'attendre à une généralisation des modules de « trace système » comme STM sur les autres architectures, ce qui est la solution la plus adaptée pour le traçage d'événements.

RÉFÉRENCES

[1] Anant Agarwal, Richard L Sites, and Mark Horowitz. *ATUM : A new technique for capturing address traces using microcode*, volume 14. IEEE Computer Society Press, 1986.

[2] ARM. *Embedded Trace Macrocell Architecture Specification*, 1999. Available on ARM website.

[3] ARM. *Embedded Trace Buffer Technical Reference Manual*, 2001. Available on ARM website.

[4] ARM. *AMBA^{TM} AHB Trace Macrocell (HTM) Technical Reference Manual*, 2004. Available on ARM website.

[5] ARM. Coresight components technical reference manual, 2006. Available as `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf`.

[6] ARM. *CoreSight PTM-A9 Technical Reference Manual*, 2008. Available on ARM website.

[7] ARM. Arm coresight promotional flyer, 2009. Available as `http://www.arm.com/files/pdf/CoreSight_Flyer_2009.pdf`.

[8] ARM. *CoreSight Trace Memory Controller Technical Reference Manual*, 2010. Available on ARM website.

[9] ARM. *System Trace Macrocell Programmers' Model Architecture Specification*, 2010. Available on ARM website.

[10] ARM. Ds-5 development studio, 2014. Available as `http://ds.arm.com/`.

[11] Dan H Barnes and Larry L Wear. Instruction tracing via microprogramming. In *Conference record of the 7th annual workshop on Microprogramming*, pages 25–27. ACM, 1974.

[12] Tim Bird. Measuring function duration with ftrace. In *Proc. of the Japan Linux Symposium*, 2009.

[13] Législation Canadienne. Loi sur le droit d'auteur. `http://lois-laws.justice.gc.ca/fra/lois/C-42/TexteComplet.html`, 1985-2012.

[14] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

[15] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, pages 209–224, 2006.

[16] Mathieu Desnoyers and Michel R Dagenais. Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer, 2009.

[17] Jake Edge. Perfcounters added to the mainline, 2009. Available as `http://lwn.net/Articles/339361/`.

[18] EfficiOS. Common trace format (ctf) specifications, 2013. Available as `http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.txt`.

[19] Frank Ch Eigler and Red Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268, 2006.

[20] Freescale Semiconductor. Application note an4420 : Linux kernel program tracing using nexus, 2011. Available on Freescale website.

[21] Freescale Semiconductor. Debug facilities. In *EREF 2.0 : A Programmer's Reference Manual for Freescale Power Architecture® Processors*, chapter 9, pages 787–803. 2011. Available on Freescale website.

[22] Freescale Semiconductor. Debug and performance monitor facilities. In *e500mc Core Reference Manual*, chapter 9, pages 295–378. 2012. Available on Freescale website.

[23] Sudhanshu Goswami. An introduction to kprobes, 2005. Available as `http://lwn.net/Articles/132196/`.

[24] IEEE-SA Standards Board. Ieee standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture. Technical report, IEEE Computer Society, 2010.

[25] Intel. Branch trace store (bts). In *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B, chapter 17.4.5, page 2411. 2013.

[26] Intel. Intel® processor trace. In *Intel® Architecture Instruction Set Extensions Programming Reference*, chapter 11, page 893. 2013.

[27] Intel. Lbr stack. In *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B, chapter 17.4.8, page 2412. 2013.

[28] Intel. Table b-2. ia-32 architectural msrs. In *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B, chapter Appendix B. 2013.

[29] Manish Jindgar. Understanding debug architecture in a system on chip (soc), 2011. Available as `http://www.techonlineindia.com/techonline/design_centers/171049/understanding-debug-architecture-chip-soc`.

[30] Chung-Fu Kao, Shyh-Ming Huang, and I-J Huang. A hardware approach to real-time program trace compression for embedded processors. *Circuits and Systems I : Regular Papers, IEEE Transactions on*, 54(3) :530–543, 2007.

[31] Jim Keniston and Srikar Dronamraju. Uprobes : User space probes. *Linux Foundation Collaboration Summit*, 2009.

[32] Namhyung Kim. perf tools : Introduce new 'ftrace' command, 2013. Available as `http://lwn.net/Articles/570503/`.

[33] James R. Larus. Efficient program tracing. *Computer*, 26(5) :52–61, 1993.

[34] Kuen-Jong Lee, A Su, Long-Feng Chen, Jia-Wei Jhou, J Kuo, and M Liu. A software/-hardware co-debug platform for multi-core systems. In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 259–262. IEEE, 2011.

[35] Geoffrey Lefebvre, Brendan Cully, Michael J Feeley, Norman C Hutchinson, and Andrew Warfield. Tralfamadore : unifying source code and execution experience. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 199–204. ACM, 2009.

[36] Frank E Levine, Brian C Twichell, and Edward H Welbon. Hardware mechanism for instruction/data address tracing, August 29 1995. US Patent 5,446,876.

[37] LTTng team. Tmf (tracing and monitoring framework), 2009. Available as `http://lttng.org/eclipse`.

[38] LTTng team. Babeltrace, 2010. Available as `http://www.efficios.com/babeltrace`.

[39] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[40] Roberto Mijat. Better trace for better software. Technical report, ARM, 2010. Available as `http://www.arm.com/files/pdf/Better_Trace_for_Better_Software_-_CoreSight_STM_with_LTTng_-_19th_October_2010.pdf`.

[41] MIPI Alliance. Specification for system trace protocol (stp), 2013.

[42] Akihiro Nagai. perf : Introduce bts sub commands, 2010. Available as `http://lwn.net/Articles/420593/`.

[43] Hugh O'Keeffe. Ieee-isto 5001™-1999, the nexus 5001 forum™ standard providing the gateway to the embedded systems of the future. Technical report, Ashling Microsystems Ltd., 2000.

[44] Pradeep Padala. Playing with ptrace, part i. *Linux Journal*, 2002(103) :5, 2002.

[45] Craig Pedersen and Jeff Acampora. Intel code execution trace resources. In *Intel® Technology Journal*, volume 16, pages 130–136. 2012.

[46] The Android Open Source Project. Decoding the ptm traces, 2012. Available as `http://lists.linaro.org/pipermail/linaro-dev/2012-November/014439.html`.

[47] James Reinders. Intel processor tracing, 2013. Available as `http://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`.

[48] Steven Rostedt. Ftrace - function tracer, 2008. Available as `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[49] Steven Rostedt. Finding origins of latencies using ftrace. *Proc. RT Linux WS*, 2009.

[50] Balázs Scherer and Gábor Horváth. Trace and debug port based watchdog processor. In *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pages 488–491. IEEE, 2012.

[51] Alexander Shishkin. etm2human : Arm's etm v3 decoder, 2009. Available as `https://github.com/virtuoso/etm2human`.

[52] Alexander Shishkin. perf : Add support for intel processor trace, 2013. Available as `http://lwn.net/Articles/576551/`.

[53] Richard Deming Smith. Computer with program tracing facility, 1972. US Patent 3,673,573.

[54] Mary Lou Soffa, Kristen R Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 888–891. ACM, 2011.

[55] Alan P Su, Jiff Kuo, Kuen-Jong Lee, Jer Huang, Guo-An Jian, Cheng-An Chien, Jiun-In Guo, and Chien-Hung Chen. Multi-core software/hardware co-debug platform with arm coresight™, on-chip test architecture and axi/ahb bus monitor. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–6. IEEE, 2011.

[56] Texas Instruments. Code composer studio ide, 2011. Available as `http://www.ti.com/tool/ccstudio`.

[57] Texas Instruments. Code composer studio ide - floating 50 user pack (f50) - 19994.95 usd, 2011. Available as `http://www.ti.com/tool/ccstudio`.

[58] Lauterbach Development Tools. Code-coverage – simplified. In *News 2012 : A Debugger for all Phases of the Project*. 2012.

[59] Vladimir Uzelac, Aleksandar Milenkovic, Milena Milenkovic, and Martin Burtscher. Real-time, unobtrusive, and efficient program execution tracing with stream caches and last stream predictors. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 173–178. IEEE, 2009.

[60] Adrien Vergé. perf : Intel bts : Add "splice" output mode, 2013. Available as `https://github.com/adrienverge/linux/tree/patch_perf_bts_splice`.

[61] Adrien Vergé. Arm coresight : Etm : Fix a vmalloc/vfree failure and enhance tracing control, 2014. Available as `https://github.com/adrienverge/linux/tree/patch_etm_v3`.

[62] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as pmu deviation : detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 6. ACM, 2011.