View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

TRAÇAGE DE SYSTÈMES EMBARQUÉS HÉTÉROGÈNES

THOMAS BERTAULD

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)

AOÛT 2016

© Thomas Bertauld, 2016.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE DE SYSTÈMES EMBARQUÉS HÉTÉROGÈNES

présenté par : BERTAULD Thomas

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente

M. DAGENAIS Michel, Ph. D., membre et directeur de recherche

M. BELTRAME Giovanni, Ph. D., membre

DÉDICACE

À mes parents, qui ont toujours cru en moi et dont l’enseignement et l’éducation ont fait de moi l’homme que je suis.

À mes grands-parents, qui m’ont toujours soutenu.

À mon oncle et à son infatigable bonne humeur.

À ma compagne, Cynthia, qui me supporte et me rend meilleur chaque jour.

À ma famille et mes amis, sans qui ma vie serait bien terne.

REMERCIEMENTS

Je remercie tout d’abord mon directeur de recherche, Michel Dagenais, pour son soutien, sa gentillesse et sa disponibilité. Il a toujours été présent pour me conseiller et me guider à travers ma maîtrise.

Je remercie Francis, Geneviève et Suchakra, pour leur aide et leurs précieux conseils. Partagez mes problèmes avec eux et profiter de leurs expériences fut un réel plaisir. En particulier, Francis m’a aidé à apprivoiser le noyau Linux, Suchakra m’a emmené toujours plus loin dans l’univers des systèmes embarqués et m’a fait partager son enthousiasme pour la technologie. Finalement, sans Geneviève, je n’aurai jamais été en mesure de produire des résultats visuels aussi rapidement et efficacement.

Je remercie également tous les autres membres du laboratoire de recherche sur les systèmes répartis ouverts et très disponibles (DORSAL), et en particulier Cédric, Didier, Franscesco, Hani, Houssem et Yves, pour leur bonne humeur et leur soutien.

Je remercie tous les professeurs qui m’ont accompagné et guidé lors de ma scolarité, c’est grâce à leur savoir et leurs enseignements que j’ai pu me rendre jusqu’ici.

Je remercie finalement ma famille qui m’a toujours soutenu et a toujours cru en moi. Je remercie en particulier mes parents, qui ont fait de moi ce que je suis et sans qui je ne serai certainement jamais arrivé au Canada. Merci à mon frère, Vincent, qui m’a également soutenu dans toutes les étapes de ma maîtrise. Enfin, je souhaite remercier ma compagne, Cynthia, et mes amis, Bastien, Benjamin, Camille, Cédric, Clément, Coraline, David, Djouneïd, Gwendal, Ismaël, Ludovic, Rémi, Sébastien, Thomas et tous les autres. Ils ont toujours été une source d’inspiration et ont su m’aider et me supporter dans les moments difficiles.

RÉSUMÉ

Pouvoir analyser et comprendre les interactions entre les composants d'un système embarqué hétérogène est essentiel pour détecter les fautes, trouver la cause de latences et optimiser les ressources. Bien souvent, des solutions propriétaires d'analyse sont directement fournies par les distributeurs. Cependant, ces solutions sont souvent incomplètes ou insuffisantes : elles nécessitent parfois de mettre le système en pause, ne sont pas adaptées pour plus d'une dizaine de processeurs analysés ou induisent des baisses de performance trop importantes.

Le traçage est une technique répandue qui consiste à enregistrer des évènements, associés à des estampilles de temps, à certains points de l'application. Tracer un système permet d'obtenir toutes les informations imaginables, avec une granularité de l'ordre de la nanoseconde. Cela permet entre autres d'effectuer des débogages complets et de diagnostiquer les problèmes de performance, que ce soit sur une machine isolée ou dans un système distribué. Néanmoins, si le traçage est très utilisé dans les systèmes « classiques », il n'en reste pas moins marginal pour les systèmes embarqués, qui offrent souvent des caractéristiques techniques bien différentes.

L'objectif de ce travail est de montrer comment il est possible de surmonter les difficultés techniques introduites par les systèmes embarqués hétérogènes (processeurs spécialisés, absence de système d'exploitation, peu de mémoire disponible, architectures exotiques…) pour produire une solution de traçage universelle sur de tels systèmes. Nous espérons ainsi démontrer que toute plateforme hétérogène embarquée peut être tracée avec les mêmes outils et dans le même format, généralisant ainsi le traçage de ces systèmes et facilitant par le fait même le travail des développeurs.

Nous montrons ainsi comment l'utilisation de *barectf*, un outil python produisant du code C destiné à générer des points de trace CTF dans des applications tournant sans système d'exploitation (bare-metal), permet de tracer virtuellement n'importe quelle plateforme. La carte Parallella et le système sur puce Keystone 2 de TI seront nos deux modèles d'expérimentation.

Nous verrons ensuite comment la synchronisation de traces peut être généralisée à de telles plates-formes pour permettre l'analyse de traces provenant d'un environnement multi-cœurs hétérogène.

Finalement, nous démontrerons à travers un cas d'étude que les méthodes et solutions proposées sont valides, fonctionnent et permettent bien de répondre aux besoins spécifiques de ces plates-formes, leur apportant une solution de traçage générique, portable et efficace.

ABSTRACT

Being able to analyze and understand interactions between all the components of a heterogeneous embedded system is mandatory to detect bugs, find the causes of latencies and optimize the resources. Proprietary solutions are often directly shipped by the producing companies. However, such solutions are rarely sufficient: they sometime require the system to be paused, are not suitable for more than a few cores and might impact the overall performances.

Tracing is a well-known technic which goal is to record timestamp-matched events. Tracing a system allows a deep understanding of the system as a whole and brings information at a nanosecond rate. This allows, among other things, to debug complete systems and diagnose performances issues, on a single machine as well as on a distributed system. Nevertheless, even if tracing is well-used in classical systems, it is still marginal on embedded systems, which are often a lot different.

The goal of this work is to show how it is possible to overcome the difficulties induced by heterogeneous embedded systems (specialized processors, no operating system, few available memory, exotic architectures…) and to have a generic tracing solution for such devices. We hope to demonstrate that every heterogeneous embedded platform can be traced with the same tools and the same output format, thus generalizing the tracing solutions on those devices and easing the developers' work.

To do so, we show how *barectf*, a python tool generating C code providing CTF tracepoints on devices with no operating system (bare-metal), allow the tracing of virtually any platform. The Parallella board and the System-on-Chip Keystone 2 from TI will be our two experimenting devices.

We will then see how traces synchronization can be generalized on such platforms and allow traces analysis on many-cores heterogeneous environments.

Finally, we will demonstrate through a use-case that the proposed solutions and methods are valid and are well-suited for those platforms, thus bringing a generic, portable and efficient tracing solution.

TABLE DES MATIÈRES

DÉDICACE ... III

REMERCIEMENTS ... IV

RÉSUMÉ ... V

ABSTRACT ... VII

TABLE DES MATIÈRES ... VIII

LISTE DES TABLEAUX ... XII

LISTE DES FIGURES ... XIII

LISTE DES SIGLES ET ABRÉVIATIONS ... XIV

CHAPITRE 1 INTRODUCTION ... 1

1.1 Définitions et concepts de base ... 2

1.1.1 Plateformes embarquées hétérogènes ... 2

1.1.2 Traçage ... 2

1.1.3 L'utilisation de coprocesseurs ... 3

1.1.4 Systèmes *bare-metal* ... 4

1.2 Éléments de problématique ... 4

1.3 Objectifs de recherche ... 5

1.4 Plan du mémoire ... 6

CHAPITRE 2 REVUE DE LITTÉRATURE ... 7

2.1 De l'intérêt des plateformes hétérogènes embarquées ... 7

2.1.1 Les coprocesseurs dans les systèmes classiques et embarqués ... 7

2.1.2 Le cas des DSPs ... 11

2.2 Les défis de communications interprocessus et hétérogènes ... 12

2.2.1 Le cas des communications interprocessus non-distribuées ... 13

2.2.2 Le cas des systèmes répartis 14

2.2.3 Comparaison au cas étudié 17

2.3 L'ère du traçage 18

2.3.1 Traçage, profilage, logging et débogage 18

2.3.2 Quelques outils de profilage 19

2.3.3 Quelques outils de traçage classiques 22

2.3.4 Traçage sur plateforme hétérogène embarquée 25

2.3.5 Analyse des traces collectées 28

2.4 De la nécessité de la synchronisation 30

2.4.1 Techniques de synchronisation dans les systèmes distribués 31

2.4.2 L'algorithme de l'enveloppe convexe 32

2.5 Conclusion de la revue de littérature 34

CHAPITRE 3 MÉTHODOLOGIE 35

3.1 Définition du problème 35

3.1.1 Vérification de l'intégration des outils classiques 35

3.1.2 Vérification du fonctionnement de *barectf* 35

3.1.3 Définition et implémentation d'une solution de synchronisation de traces 36

3.1.4 Prototypage 36

3.1.5 Test de la solution sur un problème réel 36

3.2 Présentation des plates-formes de test 37

3.2.1 Adapteva Parallella 37

3.2.2 Texas Instrument Keystone 2 39

3.3 Architecture de la solution 41

CHAPITRE 4 ARTICLE 1: TRACING HETEROGENEOUS EMBEDDED SYSTEMS 43

4.1 Abstract 43

4.2 Introduction 43

4.3 Literature review and related works 44

4.4 Studied devices 47

4.4.1 Adapteva’s Parallella 47

4.4.2 TI's Keystone 2 51

4.5 Tracing with barectf 54

4.6 Correlating heterogeneous traces 57

4.6.1 Generating pairs of matching events 58

4.6.2 Workflow and synchronization 59

4.6.3 Post-analysis treatment 61

4.7 Results 62

4.7.1 Benchmarks – Tracing overhead 62

4.7.2 Use-case 63

4.7.3 Discussion 68

4.8 Conclusion and future work 69

CHAPITRE 5 DISCUSSION GÉNÉRALE 71

5.1 Retour sur la solution proposée 71

5.1.1 Cas d’utilisation 71

5.1.2 Généralisation du traçage de *SYS/BIOS* 71

5.1.3 Contribution apportée 72

5.1.4 Validité de la solution 72

5.2 Autres travaux 73

CHAPITRE 6 CONCLUSION 75

6.1 Synthèse des travaux 75
6.2 Limitations de la solution 75
6.3 Suggestions futures 76
BIBLIOGRAPHIE 78

LISTE DES TABLEAUX

Table 1 : Benchmarking results (in cycles) on the TI Keystone 2 63

LISTE DES FIGURES

Figure 1: Exemple de sortie de *valgrind* sous Linux 20

Figure 2 : Exemple de sortie de *perf* sous Linux...21

Figure 3 : Exemple de sortie de *strace* sous Linux 23

Figure 4 : Instrumentation d'un programme avec barectf..28

Figure 5 : Exemple de sortie de babeltrace 29

Figure 6 : Exemple de visualisation de trace noyau par TraceCompass.......................................30

Figure 7 : Exemple de représentation de l'enveloppe convexe..34

Figure 8 : Architecture haut-niveau de la puce Epiphany...38

Figure 9 : Schéma des composants principaux de la Keystone 2...40

Figure 10 : Schématisation du processus de génération de paires d'évènements 41

Figure 11 : Basic setup used to trace embedded heterogeneous systems.....................................46

Figure 12 : High-level view of Parallella's main components. 48

Figure 13 : Parallella's Epiphany network design. 49

Figure 14 : Keystone 2 SoC components high-level view. 52

Figure 15 : High-level view of barectf's workflow 55

Figure 16 : Generating pairs of matching events. 58

Figure 17 : High-level view of the synchronization process..60

Figure 18 : Description of the callstack view of processes. 66

Figure 19 : Global view of the application...66

Figure 20 : Zoom on the main processing function. 67

Figure 21 : Zoom on the memory allocation part...67

Figure 22 : Zoom on the problematic area 68

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
ARM	Advanced RISC Machines
C66x	Texas Instrument CorePacs DSPs
CPU	Central Processing Unit
CTF	Common Trace Format
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GDB	The GNU Project Debugger
GPGPU	General-Purpose (computing on) GPU
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HLOS	High-Level Operating System
IPC	InterProcess Communication
LTTng	Linux Trace Toolkit next generation
RISC	Reduced Instruction Set Computer
RTSC	Real-Time Software Components
SDK	Software Development Kit
SYS/BIOS	Micro-kernel designed by TI
TI	Texas Instruments
VHDL	VHSIC Hardware Description Language

CHAPITRE 1 INTRODUCTION

Conformément à la conjecture de Moore [74], le nombre d'unités de calcul par système ne cesse d'augmenter, alors que les performances par unité de calcul n'augmentent que peu, dû à la difficulté de miniaturisation des architectures actuelles. En conséquence, les gains de performance sont maintenant atteints par l'augmentation des ressources de calculs, ou, dans notre cas, du nombre de processeurs. Cette tendance ne s'arrête pas aux seules machines « classiques » et s'étend aux plates-formes embarquées, rendant encore plus urgent le besoin de disposer d'outils d'analyse efficaces pour ces environnements multi-cœurs.

Ces plates-formes sont au cœur de ce que recherchent de grandes industries car elles sont économes en énergie mais délivrent tout de même une capacité de calcul satisfaisante. Elles peuvent ainsi être utilisées du stade de prototypage à la production. Avec les technologies actuelles et l'essor de l'*open hardware*, il est même possible de recréer soi-même son propre appareil.

Le traçage est une méthode qui se veut performante et non-intrusive pour surveiller l'état d'un système et le diagnostiquer efficacement. De nombreux outils et techniques ont été développés et testés sur des systèmes non embarqués, donnant naissance, entre autres, au Linux Trace Toolkit next generation (LTTng), traceur pour le système d'exploitation Linux.

Cependant, ces méthodes et outils sont souvent restreints à l'utilisation d'une architecture ou d'un système d'exploitation particulier. De plus, les architectures embarquées disposent de leurs propres particularités et peuvent, par exemple, intégrer des processeurs sans système d'exploitation (*bare metal*).

Pour ces raisons, tracer des systèmes hétérogènes embarqués de façon générique et être en mesure de corréler et d'analyser les traces est un défi de taille. Ce mémoire propose une solution générique répondant à cette problématique en présentant *barectf*, un utilitaire permettant de tracer des processeurs sans système d'exploitation et en introduisant une technique de synchronisation générique de traces.

1.1 Définitions et concepts de base

1.1.1 Plateformes embarquées hétérogènes

Dans tout ce qui suit, on appellera « plate-forme embarquée hétérogène » tout système (généralement constitué d'une seule et même carte) embarqué composé de plusieurs types de processeurs. Par exemple, la *Raspberry Pi*[1], bien que rentrant dans la catégorie des plates-formes embarquées, ne peut pas être qualifiée de système hétérogène selon nos critères car seuls des processeurs ARM y sont présents.

Ces plates-formes sont opposées aux systèmes dit « classiques », représentant les ordinateurs conventionnels (souvent affublés de l'architecture x86) et leurs combinaisons : clusters, systèmes distribués etc… Bien que partageant des propriétés similaires, ces deux catégories de matériel doivent être étudiées séparément, pour refléter la nature des spécificités de chacune.

1.1.2 Traçage

Tracer un système est une manière efficace d'obtenir de l'information sur son fonctionnement. En particulier, cela permet de traquer des bugs, de retrouver les causes de latences anormales, de déterminer l'élément limitant d'une application et bien d'autres choses. Cette technique repose sur l'instrumentation d'un code utilisateur avec des points de trace. Ces points de trace peuvent être vus comme des *printf* extrêmement optimisés et ne correspondent généralement qu'à une ligne de code supplémentaire.

Une trace est un recueil d'évènement associés à des estampilles de temps. Ces estampilles, extrêmement précises, sont de l'ordre de la nanoseconde. Un évènement est lui-même composé d'un contexte et d'une charge utile. Cette charge dépend entièrement du point de trace aillant généré l'évènement et peut aller du simple entier à une structure éminemment plus complexe.

Puisque le traçage a une granularité aussi fine, il est absolument nécessaire de limiter la taille des évènements, afin que la taille totale de la trace reste raisonnable sur de longues périodes de traçage. Ainsi, il est commun d'enregistrer ces évènements dans un format binaire, occupant le moins de

---

[1] https://www.raspberrypi.org/

place possible mais interdisant une lecture directe par un utilisateur. C'est pourquoi un jeu de traces doit toujours être couplé avec une définition des évènements qu'il contient. Ces définitions, appelées *métadonnées* permettront ensuite à un parseur de lire et de traiter correctement les évènements de la trace. Ce traitement peut être fait en ligne de commande ou à l'aide de logiciels graphiques plus sophistiqués.

De la même façon que l'on veut limiter l'impact sur la mémoire du traçage, il est également préférable que celui-ci interfère le moins possible avec le système étudié. Autrement dit, il est nécessaire de limiter la surcharge induite par le traçage sur le système. En effet, si cette condition n'est pas réalisée, les informations collectées seront faussées et pourraient indiquer de mauvaises causes aux erreurs constatées.

Puisque les traces reposent sur des estampilles de temps acquises sur différents systèmes, il est nécessaire de recourir à des processus de synchronisation de ces traces afin de construire une unique trace, reprenant tous les évènements de toutes les traces regroupées, sous une même origine de temps, préservant ainsi la cohérence des liens de cause à effet. En effet, comme les estampilles se basent sur les horloges des systèmes tracés, deux systèmes distincts ne produiront pas les mêmes estampilles.

Tracer une application ou un système doit se faire avec un but en tête : on ne tracera pas de la même manière un système destiné à passer en production et une application présentant de fortes latences en phase de test. Dans le premier cas, même si le faible surcoût des points de trace nous permet de tracer le système, seuls quelques points subsisteront pour, par exemple, s'assurer qu'un problème occasionnel ne survient pas. Dans le second cas, on sera tenté de placer plus de points de trace pour comprendre la cause de ces latences : on pourrait par exemple placer des points de trace en début et en fin de chaque fonction pour vérifier quelle partie de l'application prend le plus de temps à être complétée.

1.1.3 L'utilisation de coprocesseurs

Bien souvent, un jeu de coprocesseurs est mis à disposition du système principal pour le soulager de quelques tâches spécifiques. Par exemple, un coprocesseur peut être chargé de gérer les paquets réseaux sur une machine, diminuant ainsi la charge du processeur principal.

Si ces coprocesseurs peuvent être des versions « amoindries » des processeurs classiques habituellement utilisés, ils peuvent également avoir été développés avec des objectifs très particuliers en tête, leur permettant ainsi de bénéficier de capacités matérielles répondant à certains besoins.

C’est par exemple le cas des processeurs pour le traitement de signaux (*Digital Signal Processors* ou DSPs), qui, de par leur architecture, intègrent des fonctionnalités matérielles leur permettant d’effectuer des opérations de traitement de signal plus efficacement et rapidement qu’un processeur ordinaire. En particulier, on retrouve souvent dans leur architecture des composants leur permettant de traiter des fonctions mathématiques complexes, telles que des transformées de Fourier, directement au niveau matériel.

1.1.4 Systèmes *bare-metal*

Un processeur, ou plus généralement un système est dit *bare-metal* lorsqu’il ne fait l’utilisation d’aucun système d’opération. En conséquence, un système bare-metal n’a pas directement accès à des fonctionnalités telles que la gestion des tâches ou le partage des ressources et ne peut compter sur l’utilisation d’aucun outil destiné à des systèmes d’exploitation spécifiques.

Programmer de tels systèmes implique que le développeur doit faire lui-même la gestion des ressources et notamment des allocations mémoire. Cela signifie également que tout lui est permis, au risque de rendre le système inutilisable. C’est pourquoi il est si important de pouvoir tracer de tels appareils, dont les coprocesseurs font souvent partie.

1.2 Éléments de problématique

De nombreux systèmes ont recours à des plates-formes hétérogènes embarquées pour satisfaire des besoins spécifiques tels que le traitement de signaux. Ces environnements restreints, mêlant plusieurs types de processeurs et dont certains n’ont pas recours à un système d’exploitation, rendent le traçage de tels systèmes difficiles.

En effet, sans système d’exploitation, il est impossible d’utiliser les outils classiques tels que LTTng, et les outils propriétaires fournis par les producteurs ne sont souvent pas adaptés pour le traçage, relevant plus du profilage. En conséquence, ils n’offrent pas des résultats satisfaisant en

termes d'informations gagnées par rapport à la surcharge induite en comparaison avec des outils de traçage conventionnels.

De plus, puisque les plates-formes hétérogènes embarquées promettent de fortes puissances de calcul couplées à une faible consommation énergique, le tout pour des prix généralement bas, elles sont de plus en plus utilisées dans les milieux industriels. La présence de DSPs, notamment, est un atout non négligeable lorsqu'un quelconque besoin de traitement de signaux est présent.

Disposer d'un moyen universel de tracer ces environnements ouvrirait la porte vers encore plus de performance et des temps de développement réduits.

Néanmoins, si obtenir les traces d'un processeur sur un système est un excellent moyen de comprendre ce qu'il se passe au niveau de ce cœur, il est généralement nécessaire de pouvoir corréler toutes les traces d'un même système pour pouvoir mettre en exergue les dépendances de ses différentes parties.

Si des techniques reposant sur la synchronisation de traces existent pour les systèmes répartis classiques, aucune méthode générique n'a été développée pour réaliser cette tâche sur les plates-formes hétérogènes embarquées.

1.3 Objectifs de recherche

Les travaux présentés dans ce mémoire visent à répondre à la question suivante :

Comment construire une méthode générique permettant de tracer efficacement tout type de système embarqué hétérogène tout en maintenant la cohérence entre les traces obtenues ?

Pour répondre à cette question, nous passerons à travers les étapes suivantes :

1. Comprendre les caractéristiques et besoins spéciaux amenés par les plates-formes hétérogènes embarquées.
2. Effectuer des tests avec *barectf* afin de pouvoir tracer des processeurs bare-metal.
3. Développer un premier prototype complet permettant de tracer et de synchroniser les traces sur la plate-forme Parallella.
4. Vérifier la généricité et étendre le précédent prototype à la plateforme Keystone 2.
5. Formaliser et généraliser la méthode à toutes les plates-formes hétérogènes embarquées.

1.4 Plan du mémoire

Le chapitre 2 commence par faire un état de l'art sur les plates-formes hétérogènes embarquées, l'utilisation de coprocesseurs, les techniques de traçage classiques, les méthodes de communication interprocessus et les moyens de synchronisation d'évènements dans des systèmes multiprocesseurs. Par la suite, le chapitre 3 présente la méthodologie adoptée pour construire la solution décrite dans ce mémoire. Le chapitre 4 est constitué de l'article « Tracing heterogeneous embedded systems » qui décrit une solution de traçage générique pour les systèmes hétérogènes embarqués et propose un cas d'utilisation complet permettant de démontrer ce qu'il est présentement possible de faire et ce qui pourrait être possible dans le futur. Finalement, le chapitre 5 fait état de résultats complémentaires et évalue la réalisation des objectifs posés avant de conclure sur les travaux présentés et sur de possibles travaux futurs au chapitre 6.

CHAPITRE 2 REVUE DE LITTÉRATURE

La présente section dresse un portrait de l'état de l'art sur ce qui concerne le traçage, la synchronisation de traces et l'utilisation de plates-formes hétérogènes embarquées. Son objectif est de présenter les concepts et outils sur lesquels repose la solution proposée par ce travail. Cette étude se base sur le système d'exploitation Linux et ses dérivés. Par abus de langage, c'est à cet ensemble de systèmes d'exploitation que les termes « HLOS » ou « système d'exploitation » font référence.

2.1 De l'intérêt des plateformes hétérogènes embarquées

Avant de rentrer en détails sur les techniques de traçage de ces plates-formes, il est bon de comprendre pourquoi il est intéressant de les tracer. Comme nous l'avons défini plus haut, ces systèmes embarquent plusieurs types de processeurs que l'on peut généralement placer dans deux catégories :

- Les processeurs « maîtres », souvent capables d'utiliser un système d'exploitation de haut niveau (HLOS) tel que Linux, et en charge de commander les processeurs « esclaves ».
- Les processeurs « esclaves », généralement présents en tant que coprocesseurs, sont chargés d'effectuer des tâches très précises et en général demandant de fortes capacités de calcul. Ils prennent leurs ordres des processeurs « maîtres » et, dans la majorité des cas, ne peuvent pas faire tourner de système d'exploitation. Ils sont alors qualifiés de processeurs « bare-metal » et feront l'objet d'une étude plus approfondie par la suite.

Leur principal intérêt est donc de fournir un environnement complet regroupant des processeurs classiques et des coprocesseurs de calcul pouvant être intégré dans des systèmes plus complexes. De telles plates-formes offrent de plus d'excellentes performances pour du matériel embarqué et consomment beaucoup moins d'énergie que des systèmes classiques.

2.1.1 Les coprocesseurs dans les systèmes classiques et embarqués

Bien que les coprocesseurs qui nous intéressent dans cette étude se trouvent sur des plates-formes embarquées, on peut également en trouver couramment sur des systèmes classiques.

Néanmoins, s'ils permettent généralement des gains de performances sur des applications spécialisées, ce n'est pas sans apporter un lot de problèmes à prendre en considération. Conte et al.

[13] en décrivent certains tels que la gestion de plusieurs langages de programmation, de plusieurs compilateurs parfois moins connus, et la problématique de la répartition des tâches et charges entre les processeurs génériques et les coprocesseurs.

En effet, le principal intérêt d'avoir à disposition des coprocesseurs est de pouvoir soulager le ou les processeurs principaux en effectuant des tâches spécialisées comme le traitement d'un signal ou la réalisation d'opérations mathématiques complexes. Il est donc nécessaire de distinguer les différents types de coprocesseurs et leurs caractéristiques et d'analyser les meilleures façons de répartir la charge de travail entre tous les éléments d'un système hétérogène.

Graphics Processing Units (GPUs)

Les GPUs sont un parfait exemple de coprocesseurs utilisés au quotidien aussi bien par les particuliers que les industries. Un usage classique de GPU est d'effectuer le rendu graphique d'images, vidéos ou jeux-vidéos, utilisant ainsi à leur plein potentiel les milliers de cœurs que peuvent offrir ces unités de calcul.

Néanmoins, avec l'avancée des technologies, les GPUs ont acquis un rôle plus vaste que le simple rendu graphique et peuvent être utilisés en tant que GPGPU, c'est-à-dire comme des unités de calcul génériques pour certaines tâches hautement parallélisables. Zamith et al. [4] montrent en particulier comment les GPUs peuvent être utilisés pour gérer des calculs complexes liés notamment à la physique ou à l'intelligence artificielle dans une application graphique en temps réel telle qu'un jeu-vidéo. Une part importante de leur travail consiste à étudier la répartition optimale des traitements à effectuer entre le CPU et le GPU, ce qui est une constante lorsque l'on souhaite répartir des tâches entre processeurs et coprocesseurs.

Boyer et al. [1] ont également prouvé les bienfaits de l'utilisation de GPUs pour des calculs scientifiques, en montrant comment la réécriture de certaines parties parallélisables d'un algorithme de détection et de suivi de globules blancs sur GPU pouvait accélérer plus de 200 fois les performances de l'application. Outre les gains techniques indiscutables, cela épargne également aux chercheurs de nombreuses heures de travaux fastidieux.

Si les deux travaux précédents démontrent qu'il est possible d'atteindre des performances grandement améliorées via l'utilisation de GPGPU, ils montrent également que le problème principal du portage d'applications sur de tels systèmes réside dans la répartition des données et traitements entre les différentes unités de calculs (CPUs, GPUs, autres coprocesseurs…). En effet,

certaines tâches seront toujours plus efficacement traitées sur CPU et d'autres ne pourront même pas être portées sur GPU par manque de parallélisme interne.

Bien qu'il soit impossible de définir une méthode complètement générique pour découper efficacement une application sur un système hétérogène, car le découpage en question repose principalement sur la forme de l'algorithme utilisé, Haidar et al. [65] discutent de techniques à base d'intergiciels destinées à équilibrer la charge de travail entre différentes unités hétérogènes d'un même système. En particulier, ils montrent comment l'ordonnancement de tâches en accord avec leurs dépendances de données, l'utilisation de QUARK (*QUeing and Runtime for Kernels*) et l'attribution de cotes de capacités à chaque unité de calcul, permet de maximiser les gains obtenus sur chacune d'entre elles.

Field Programmable Gate Arrays (FPGA)

Les FPGAs sont des entités que l'on retrouve à la fois dans les systèmes classiques et embarqués. Il s'agit d'éléments programmables au niveau matériel, c'est-à-dire pouvant changer d'architecture pour satisfaire au mieux les besoins de l'utilisateur. En particulier, certaines tâches nécessitant d'être répétées de nombreuses fois peuvent être optimisées en étant transcrites sur un module FPGA.

Néanmoins, comme le notent Park et al. [19], les performances obtenues via FPGA dépendent fortement de l'implémentation de l'algorithme et du langage utilisé pour produire le code final. Ainsi, des algorithmes décrits en *DIME-C* ou *Mitrion-C*, deux langages de haut-niveau se rapprochant du C et destinés à être compilés en VHDL (qui peut être vu comme le langage canonique de bas-niveau pour programmer un FPGA), peuvent avoir des performances jusqu'à 10 fois inférieures à celle obtenues à l'aide d'un GPU. Au contraire, les algorithmes directement décrits en VHDL offrent de nettes améliorations de performance, au détriment du temps de développement.

La principale force du FPGA repose dans sa capacité à pouvoir être programmé pour effectuer des tâches au niveau matériel. À ce titre, Franchini et al. [6] montrent comment un module FPGA peut être programmé à servir de coprocesseur graphique, implémentant directement au niveau matériel des opérateurs de l'algèbre géométrique de Clifford, offrant ainsi jusqu'à 20 fois plus de performance que sur un processeur générique.

Si les FPGAs peuvent être utilisés comme coprocesseurs, leur utilité va bien au-delà de cela et ils peuvent, par exemple, également être utilisés en tant que « pont » permettant de relier différentes entités d'un même système. C'est par exemple le cas sur la carte Parallella d'Adapteva où le module FPGA sert principalement de « hub » aux différents composants du système et permet d'assurer la liaison entre le processeur ARM et le coprocesseur, la puce Epiphany.

Ekas et Jentz [23] expliquent en particulier comment peuvent coexister des DSPs, coprocesseurs, GPPs et FPGAs dans un même système et énoncent encore une fois les enjeux ayant trait à la répartition de la charge de travail entre ces différents éléments. Plus encore, ils montrent comment un tel système peut gagner encore plus de performance en interfaçant tous ses éléments sur le FPGA et en construisant un tampon sur celui-ci, agissant comme une cache mémoire locale pour les coprocesseurs. Ce constat résulte du fait que dans de tels environnements, les communications entre les unités de calcul, et donc la performance globale du système, sont fortement impactées par l'efficacité du placement et de l'accès aux données en mémoire. Par ailleurs, tout comme Steiner et al. [32], ils démontrent comment un FPGA peut être utilisé pour créer des DSPs ou coprocesseurs génériques, permettant encore une fois des gains de performances non-négligeables pour des tâches transférées depuis un processeur générique.

Epiphany

L'Epiphany (voir [30]) est une puce développée par la compagnie Adapteva et sur laquelle repose la puissance de calcul de la carte Parallella, étudiée dans ce travail de recherche. Une description complète de la puce et de ses caractéristiques sera donnée par la suite et nous nous contentons ici d'en exhiber les principales caractéristiques et les travaux y étant liés.

Olofsson et al. [37] de la compagnie Adapteva ont présenté la puce Epiphany, il y a quelques années, comme étant un réseau de processeurs à usage générique. La structure globale de la puce s'articule donc autour de ce réseau de 16 à 64 processeurs, programmables en C et utilisant un modèle de mémoire partagée distribuée. L'objectif principal de cet ensemble de (co)processeurs est d'offrir des performances raisonnables tout en étant 10 fois moins énergivore que les processeurs traditionnels.

Comme c'est une puce relativement récente, peu de travaux la concernant ont été publiés. Aaberge [5] propose néanmoins une analyse sommaire des performances de la version 16 cœurs de l'Epiphany contre un cœur de l'ARM présent sur la carte Parallella. Les résultats semblent indiquer

des performances jusqu'à 13 fois supérieures pour l'Epiphany. Cependant, ces tests ne sont pas représentatifs car les algorithmes utilisés pour tester les performances, tels que le calcul de l'espace de Mandelbrot [29], ne sont pas vraiment pertinents et que la comparaison se fait entre une application mono-thread sur ARM et parallélisée 16 fois sur l'Epiphany.

Une étude plus complète et réaliste réalisée par Malvoni et al. [10] compare les performances de l'Epiphany à celles de processeurs classiques à technologies équivalentes. L'application de test, *Bcrypt*, un programme de hash de mots de passe, compare notamment les puces Epiphany en version 16 et 64 cœurs avec un processeur Intel core i7-2600K. Si la version 16 processeurs de l'Epiphany est 4 fois moins performante, la version 64 cœurs offre des performances très proches et les deux versions offrent un rapport opérations effectuées sur énergie consommée des centaines de fois supérieur. Autrement dit, si l'Epiphany n'est pas le supercalculateur annoncé à sa sortie, il n'en reste pas moins que la promesse de délivrer des performances satisfaisantes (dans la version 64 cœurs) à bas coûts énergétiques semble tenue. De possibles raisons limitant les performances de cette puce seront expliquées dans la section dédiée à la carte Parallella.

2.1.2 Le cas des DSPs

Le cas des DSPs est discuté à l'écart des autres coprocesseurs présentés car leur utilisation est beaucoup moins générique que les autres. Comme nous l'avons vu, il est possible d'utiliser les GPUs comme GPGPUs ou n'importe quel ensemble de coprocesseurs générique tel que l'Epiphany pour remplir des tâches génériques et soulager le processeur principal. De même, puisque les FPGAs sont entièrement programmables, il est possible de les transformer en coprocesseurs génériques ou spécialisés, comme des DSPs.

Comme leur nom le suggère, les DSPs, même s'ils peuvent être utilisés à remplir des tâches plus génériques, seront mieux employés à effectuer des opérations mathématiques demandantes telles que du traitement de signal, notamment dans les appareils de télécommunications. Leurs principales forces, telles que soulignées par Lawlor [72], est de pouvoir traiter des opérations mathématiques complexes en un nombre record d'instructions tout en consommant moins d'énergie qu'un processeur classique, grâce à leur faible fréquence de fonctionnement. À ce titre, et tel que Gatherer et al. [27] le prédisaient en l'an 2000, l'utilisation et la puissance des DSPs ne fait qu'augmenter, proportionnellement à nos besoins en matière de télécommunications.

Ainsi, Liu et Hang [2] utilisent les DSPs de TI pour construire un encodeur JPEG très performant grâce aux optimisations réalisées par le compilateur de ces processeurs. De même, Huang et al. [49] montrent comment l'utilisation de DSPs (provenant encore une fois de TI) à la place de processeurs standards, pour effectuer des algorithmes de contrôle, permet une nette amélioration de la précision du mouvement dans des systèmes mécatroniques. Enfin, Wallace et al. [8] démontrent comment les DSPs peuvent parfaitement s'intégrer pour le meilleur à l'architecture d'une antenne de télécommunication « multiple-input multiple-output » (MIMO).

Un article de Texas Instrument [21], les créateurs originaux des DSPs, explique comment l'usage de DSPs s'est étendu du simple traitement de signaux à la réalisation d'analyses de données en temps réel. Ils montrent également que, grâce aux outils fournis, il est aisé de programmer un DSP pour en tirer les meilleures performances possibles. En particulier, ils prônent l'utilisation de leur outil de profilage, qui permet de vérifier l'état général d'un DSP (valeurs des registres, tâche en cours d'exécution…) à un instant donné et d'obtenir différentes statistiques comme le temps passé dans chaque fonction. Néanmoins, aucune des solutions proposées dans cet article ne se rapproche de ce que l'on aimerait obtenir : des traces ne perturbant pas ou peu l'exécution du système. Puisqu'aucune autre solution ne semble avoir déjà été proposée dans ce domaine, cela fait partie de nos objectifs de recherche.

2.2 Les défis de communications interprocessus et hétérogènes

Les communications en environnements hétérogènes ont pour défi de permettre à des systèmes et architectures différents de pouvoir communiquer efficacement. L'objectif recherché est d'être en mesure de pouvoir échanger des messages, ou plus généralement des données, d'une machine à une autre ou d'un processus à un autre. La plupart des considérations ci-dessous sont également valables dans le cas de communications interprocessus classiques, c'est-à-dire ne faisant pas intervenir de composante hétérogène.

Lamport [43] pose les bases théoriques des besoins pour la communication interprocessus et la réduit au partage d'un médium par un processus producteur et un processus consommateur. Dans ce modèle, le consommateur scrute les changements du médium ou d'une balise (*flag*) indiquant le changement du médium (une zone de mémoire partagée par exemple). Déjà, le principal problème de ces communications est caractérisé par le problème classique du producteur consommateur dans

lequel de l'exclusion mutuelle entre plusieurs producteurs ou entre producteurs et consommateurs est requise pour garantir la conformité des données échangées.

2.2.1 Le cas des communications interprocessus non-distribuées

Nous considérons ici le cas de machines multiprocesseurs uniques telles que des ordinateurs de bureau ou des plates-formes embarquées. Ici, le système d'exploitation est en charge de l'attribution des ressources partagées et gère les communications interprocessus.

Le travail de Cheng et Zhang [9] se concentre sur l'amélioration des performances des communications interprocessus dépendant de micro-noyaux. Il montre comment un très simple mécanisme, utilisant une zone de mémoire partagée entre les processus, permet une communication efficace dans un environnement multi-cœurs. Si ici cet espace mémoire est initialement instancié par le micro-noyau, dans le cas général qui nous intéresse, l'utilisation directe d'adresses physiques, ou l'allocation statique d'un bloc mémoire au démarrage de l'appareil via un module noyau du maître, est suffisant pour disposer de cet espace d'échange commun.

D'une manière plus complète, Bershad et al. [66] discutent des bienfaits de l'utilisation de la mémoire partagée en espace utilisateur pour des communications entre plusieurs processeurs. Le principal avantage est de pouvoir outrepasser le noyau en transmettant des messages de processus à processus à travers un espace mémoire partagé. Ils construisent ainsi URPC (*User-Level Remote Procedure Call*), une technique destinée à effectuer des appels de procédures distants (d'un processeur sur un autre) passant directement par la mémoire partagée. Cela permet entre autres de gagner en performance car les appels deviennent directs et n'ont pas besoin de faire des allers-retours entre l'espace utilisateur et le noyau. En outre, les auteurs affirment que cela amène une meilleure répartition des tâches entre le noyau et l'espace utilisateur.

Nahro et Omar [17] proposent un système de communication entre les processeurs de la puce Epiphany, qui fait l'objet d'une partie de cette recherche. Dans le contexte de leur travail, les auteurs utilisent un portage de *FreeRTOS*, un noyau léger destiné à tourner sur les processeurs embarqués les plus modestes. Chaque cœur de l'Epiphany possède une instance du noyau et leur problématique est d'arriver à un modèle de communication entre ces instances. Ils proposent ainsi l'utilisation de « mailboxes », modélisées par des tableaux multi-dimensions, sur chaque processeur. Une instance souhaitant communiquer avec une autre doit alors écrire directement dans

la mailbox de la cible, située dans la mémoire locale de cette dernière. Si leur solution semble efficace et fonctionnelle, elle n'est néanmoins pas satisfaisante pour notre étude pour plusieurs raisons :

- Elle repose sur l'utilisation d'un noyau sur les coprocesseurs, ce qui n'est pas souhaitable.
- Puisque les boîtes de réception des messages sont dans la mémoire locale des coprocesseurs, elles sont limitées à quelques Ko en taille. Cela n'est évidemment pas suffisant pour stocker des paquets d'évènements d'une trace, qui, comme nous le verrons dans la section suivante, peut avoir une taille exubérante. Un recours à une plus grande zone de mémoire partagée, par exemple dans la RAM, sera donc nécessaire.
- Le mécanisme d'exclusion mutuelle employé pour assurer la cohérence des lectures et écritures repose sur des APIs propriétaires fournies par le constructeur de la puce et n'est donc pas généralisable.

2.2.2 Le cas des systèmes répartis

Puisque les systèmes répartis ne peuvent pas toujours compter sur la présence d'une zone mémoire partagée, les communications entre les différents acteurs se font à travers le réseau, via l'envoi et la réception de messages.

Comme le rappelle Tanenbaum dans [26], cela requiert un accord sur les données échangées, ou *protocole*, entre les différents participants. Notamment, des précautions doivent être prises vis-à-vis de l'encodage des messages, que l'on parle de messages textuels (encodage ASCII ou Unicode par exemple), ou bien d'ordre des bits (petit-boutiste ou grand-boutiste), pour assurer que les données puissent être correctement lues par les deux parties. Cela est d'autant plus vrai que les différentes machines participantes seront certainement d'architectures différentes et qu'on ne peut raisonnablement pas supposer à l'avance l'architecture de la machine avec laquelle on souhaite communiquer.

Tanenbaum fait également l'inventaire des moyens de communication classiquement mis à la disposition des systèmes distribués :

- **L'appel de fonctions distantes**, ou *RPC* (*Remote Procedure Call*), consiste à exécuter une fonction sur un serveur distant tout en simulant un appel local. En d'autres termes,

le système appelant n'a besoin d'aucun *a priori* sur le serveur et considère l'appel comme s'il était local. Tout comme dans le cas d'un appel local, un appel RPC peut être asynchrone (non-bloquant) ou synchrone (bloquant).

- **La communication orientée messages** est ce qui se rapproche le plus d'un échange traditionnel entre un émetteur et un receveur. Si le principe semble simple, la principale difficulté est d'établir un protocole et un format de communication valables à travers des architectures différentes. C'est dans ce contexte qu'est né MPI (*Message-Passing Interface*), proposant un moyen générique de communication par messages dans des systèmes distribués ne nécessitant comme prérequis que la connaissance et l'établissement de groupes de processus communiquant. Un brevet datant de 1985 [41] faisait d'ailleurs déjà état de la possibilité de communications génériques dans un environnement distribué où l'organisation des processus est connue à l'avance. Les échanges MPI sont destinés à être rapides (de l'ordre de la seconde).

- **La communication orientée messages par files d'attentes** se rapproche de la communication orientée messages vue au point précédent à ceci près que des files locales à l'envoyeur et au receveur sont utilisées pour stocker les messages, n'offrant ainsi généralement pas d'autre garantie à l'émetteur que celle que son message sera un jour placé dans la queue du récepteur. Ce type de communication peut être grossièrement vu comme un échange d'e-mail entre deux personnes et est donc moins approprié à des échanges rapides qu'une solution comme MPI.

L'auteur présente également d'autres moyens de communications tels que la communication par flots, qui dépassent le cadre de cette étude. Même si les techniques présentées ci-dessus sont destinées aux systèmes distribués, il n'est pas rare de les voir adaptées à des systèmes hétérogènes embarqués. Texas Instrument [56], par exemple, fournit une librairie générique pour tous ses modèles de plates-formes hétérogènes permettant d'effectuer des communications par messages se rapprochant beaucoup des communications orientées messages par files d'attentes.

De nombreux travaux se basent sur ces principes. Par exemple, Rajkumar et al. [60] discutent d'un prototype de modèle producteur/consommateur temps-réel basé sur la communication orientée messages. La base de leur implémentation repose sur la création d'estampilles logiques à portée du système complet pour les messages. Ainsi, il devient facile pour un producteur de traquer ses

consommateurs et à des consommateurs de souscrire aux messages de producteurs. Cela fait écho à un autre problème des communications distribuées qui sera discuté dans une partie lui étant consacré : la synchronisation des processus.

Tanenbaum discute également d'un autre problème primordial dans la communication entre processus distribués : le cas de l'accès à des données partagées et l'exclusion mutuelle. Il explique que quatre approches principales peuvent être distinguées :

- La mise en place d'un serveur centralisé faisant les comptes des ressources détenues par les processus. Dans ce cas, si un processus veut accéder à une ressource, il demande à ce serveur de vérifier qu'aucun autre processus n'y accède présentement et, si tel est le cas, il en récupère l'accès. Comme toutes les solutions centralisées, un problème évident survient si le serveur gérant l'accès aux ressources faillit.
- La mise en place d'un gestionnaire de ressources pour chaque processus, reprenant la solution précédente mais en éliminant quelque peu les défauts de l'aspect centralisé.
- L'utilisation d'un système distribué, plus résilient aux erreurs mais sensible à la famine, consiste à faire en sorte que chaque processus nécessitant l'accès à une ressource envoie un message à tous les autres processus du système. S'ils n'ont pas la possession de la ressource, ils donnent leur accord au processus demandeur par message. Dans le cas contraire, la demande est mise en attente.
- La passation d'un « jeton » (*token* en anglais) entre les processus. De cette façon, on garantit qu'exactement un seul processus peut avoir accès à la ressource. Une fois qu'il en a fini avec elle, il peut passer le jeton à un autre processus, par exemple son voisin (la notion de voisin peut être induite par la numérotation des processus). Si cette solution est la plus simple, elle est également vulnérable à la perte du jeton lors des communications ou à la défaillance du processus le détenant.

Toutes ces approches fonctionnent en théorie sans aucun problème sur des systèmes parfaits, sans fautes. Néanmoins, puisque cela ne reflète pas la réalité, l'utilisation de l'une ou l'autre des solutions dépendra des besoins particuliers du système, en tenant compte des imperfections de chacune.

2.2.3 Comparaison au cas étudié

Pour fixer les idées, nous considérons le cas où le maître est un processeur classique (architecture de type x86 ou ARM par exemple) sur lequel roule un HLOS et l'esclave est un coprocesseur générique (ou un DSP) ne pouvant pas faire usage d'un système d'exploitation. Ce cas est le plus réaliste car, même si Durrant et al. [54] ou [25] ont montré qu'il est possible de faire fonctionner un noyau Linux sur des DSPs sous des conditions raisonnables, cela est généralement contre-productif car la surcharge induite par la présence d'un système d'exploitation diminuerait fortement les gains apportés par l'utilisation de ces DSPs. En effet, cela amènerait de nombreux niveaux supplémentaires à la couche de code et risquerait de faire mauvais usage des capacités matérielles fournies par ces composants.

Puisque nous nous intéressons à des systèmes hétérogènes dans lesquels un composant « maître » (par exemple un processeur ARM) doit communiquer avec ses « esclaves » (par exemple, des DSPs), il est essentiel d'étudier les moyens de communication à leur disposition. On parle ici de communication sur un même système, les communications réseaux sont donc exclues de la discussion. Comme nous faisons l'hypothèse que les esclaves ne peuvent pas faire tourner de HLOS, il est nécessaire de trouver une méthode générique capable de fonctionner même sur des systèmes bare-metal. Dans le cas étudié, les esclaves ont un accès direct à la mémoire du système (ou à une partie de la mémoire) et, puisque le maître dispose d'un HLOS, il est généralement en charge des allocations mémoire à travers le noyau de son système d'exploitation (dont la gestion de la mémoire est l'un des rôles principaux).

De manière évidente, si les deux parties ont accès à un système d'exploitation tel que Linux, des moyens classiques de communication tels que des *pipes,* APIs IPC ou même l'usage d'appels système *ioctl()* peuvent être exploités (voir par exemple le chapitre 6 du *Linux Programmer's Guide*).

D'après l'étude des documents qui précèdent, la seule façon absolument générique d'échanger de l'information entre un maître et ses esclaves est d'utiliser un espace mémoire partagé leur permettant de stocker des données communes. Des modèles plus poussés (mais toujours aussi génériques) peuvent également être imaginés. Par exemple, le maître peut écrire des données en mémoire à l'intention d'un esclave et lui envoyer une interruption pour lui signifier l'arrivée

desdites données. Ce faisant, l'esclave n'est pas obligé de scruter la mémoire à chaque instant jusqu'à l'arrivée des informations et peut passer plus de temps à effectuer des opérations utiles.

Par ailleurs, puisque l'on ne peut pas supposer la présence d'outils matériel pour faciliter la synchronisation de processus entre le maître et les esclaves, différents canaux seront utilisés pour communiquer entre un maître et l'un de ses esclaves et vis-versa. De cette façon, il est garanti qu'aucun problème de concurrence ne se présentera entre les différents acteurs du système. En particulier, on définira des zones mémoires non-entrelacées servant de tampons aux données échangées entre les maîtres et les esclaves.

2.3 L'ère du traçage

Le traçage est une technique n'ayant plus à faire ses preuves et étant largement utilisée dans l'industrie. Elle consiste à engranger de l'information sur un système sous la forme d'évènements générés à certains endroits du code du système étudié. Le code nécessaire à la génération d'un évènement est appelé un *point de trace*. Ces évènements peuvent être de haut-niveau (traçage d'une application utilisateur) ou de très bas-niveau (traçage du noyau Linux), l'important étant de garantir que, malgré la quantité astronomique d'évènements pouvant être générés durant une session de traçage, l'application tracée est perturbée le moins possible. Dans le cas contraire, le traceur pourrait causer de nouveaux problèmes de performance, se substituant ainsi potentiellement à la cause initialement recherchée.

Lors de leur génération, ces évènements sont associés à des estampilles de temps, permettant de retracer l'exécution complète d'une application. La précision de ces estampilles étant de l'ordre de la nanoseconde, des compteurs de 64 bits sont nécessaires pour les générer. Une collection d'évènements associés à des estampilles de temps correspond à ce que l'on appelle une trace.

2.3.1 Traçage, profilage, logging et débogage

Parallèlement, le *logging* (ou journalisation) concerne également l'enregistrement d'évènements mais à un niveau beaucoup plus haut que le traçage. En général, cela consiste à enregistrer des « points de passage » atteints par une application tels que la fin de l'exécution d'une tâche. La forme la plus simple de logging consiste à écrire des messages sur la sortie standard ou dans un fichier à l'aide de primitives de type *printf*. Parce que ces évènements ne sont pas destinés à être

nombreux, leur impact sur les performances globales d'un système est négligeable. Si, au contraire, on essayait d'utiliser des outils de logging pour générer autant d'évènements que dans une trace, les performances seraient terribles et c'est pourquoi, dans ce qui suit, nous ne nous intéressons qu'au traçage, qui peut être vu comme une forme de logging hautement optimisée.

Le but du profilage est d'obtenir des informations globales et des statistiques sur le déroulement d'une application dans un système. Pour cette raison, le traçage et le profilage ne recherchent pas les mêmes objectifs. De même, le but du débogage est simple et consiste à retrouver et analyser les causes d'erreurs dans un système. De nombreux outils sont dédiés à ces trois techniques et peuvent intervenir à différentes étapes du développement.

Généralement, le profilage est utilisé pour repérer des portions de l'application consommant beaucoup de ressources, aussi bien CPU que mémoire. Le profilage, contrairement au traçage, ne se soucie donc pas d'un quelconque ordre des informations collectées, et se contente de donner une vision globale de certaines parties du système sur un intervalle de temps donné [36].

Si le rôle du débogage est évident lors du cycle de développement d'une application, le rôle du traçage dépend quant à lui des besoins ciblés. En effet, puisqu'il est extrêmement versatile, le traçage doit se faire avec un objectif en tête et peut être utilisé pour déboguer, profiler, surveiller le flot d'exécution, ou, entre autres, effectuer du logging sur une application. Les traces collectées reflèteront l'instrumentation réalisée.

Un scénario classique consiste à utiliser le débogage couplé au traçage lors de toutes les phases de développement, de manière à détecter et prévenir plus efficacement les causes d'erreurs, puis à utiliser le profilage sur des prototypes fonctionnels afin d'observer les composants les plus demandants en ressources. Ensuite, à l'aide du traçage, il est possible de trouver spécifiquement quelles parties de l'application peuvent être optimisées. Une fois les optimisations effectuées, on peut revenir à l'étape de profilage jusqu'à être pleinement satisfait des performances du système.

2.3.2 Quelques outils de profilage

Valgrind

Valgrind[2] est un outil de profilage classique que tout développeur C a au moins utilisé une fois, ne serait-ce que pour vérifier la cohérence des allocations mémoire dynamiques. Plus précisément, valgrind est un framework permettant l'instrumentation dynamique de binaires [69]. Il pose donc les fondations nécessaires à la création de profileurs tels que *Memcheck* ([68]).

Son utilisation canonique sous Linux (reposant sur *memcheck*) permet néanmoins d'obtenir directement des statistiques de l'utilisation de la mémoire d'un programme, permettant de détecter par exemple des fuites mémoires ou des lectures/écritures invalides. Un exemple de rapport généré par valgrind est donné par la figure 1.

```
[01:13] skinner@sweet-E6420 :testsC $ valgrind ./quicksort
==19416== Memcheck, a memory error detector
==19416== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==19416== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19416== Command: ./quicksort
==19416==
==19416==
==19416== HEAP SUMMARY:
==19416==     in use at exit: 60 bytes in 1 blocks
==19416==   total heap usage: 30 allocs, 29 frees, 416 bytes allocated
==19416==
==19416== LEAK SUMMARY:
==19416==    definitely lost: 60 bytes in 1 blocks
==19416==    indirectly lost: 0 bytes in 0 blocks
==19416==      possibly lost: 0 bytes in 0 blocks
==19416==    still reachable: 0 bytes in 0 blocks
==19416==         suppressed: 0 bytes in 0 blocks
==19416== Rerun with --leak-check=full to see details of leaked memory
==19416==
==19416== For counts of detected and suppressed errors, rerun with: -v
==19416== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1: Exemple de sortie de *valgrind* sous Linux

Perf

Perf[3] est à la base un outil léger destiné à requérir et analyser les données collectées par les compteurs de performances des systèmes étudiés. Ces compteurs sont des registres spéciaux dont les valeurs varient en fonction de l'état du système et reflètent les occurrences de certains évènements tels que le passage d'un cycle CPU, l'arrivée d'une interruption, l'apparition d'un défaut de cache et bien d'autres.

La figure 2 montre une exécution sommaire de *perf*, exhibant le nombre d'apparitions de certains évènements lors de l'exécution d'une application.

[2] http://valgrind.org/

[3] https://perf.wiki.kernel.org/

```
root@kali:~# perf stat /bin/ls
Desktop  Documents  Downloads  Music  perf.data  Pictures  Public  Templates  Videos

 Performance counter stats for '/bin/ls':

          3.804290      task-clock (msec)         #    0.847 CPUs utilized
                 2      context-switches          #    0.526 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               115      page-faults               #    0.030 M/sec
         3,299,794      cycles                    #    0.867 GHz
         1,637,846      stalled-cycles-frontend   #   49.63% frontend cycles idle
         1,281,474      stalled-cycles-backend    #   38.83% backend  cycles idle
         3,335,328      instructions              #    1.01  insns per cycle
                                                  #    0.49  stalled cycles per insn
           573,660      branches                  #  150.793 M/sec
            10,054      branch-misses             #    1.75% of all branches          (2.67%)

       0.004490180 seconds time elapsed
```

Figure 2 : Exemple de sortie de *perf* sous Linux

Perf a régulièrement évolué et est aujourd’hui en mesure d’utiliser des points de trace du noyau Linux et d’ajouter des estampilles de temps aux évènements collectés afin d’agir comme un traceur. Il permet en outre d’avoir des statistiques d’occurrences d’évènements logiciels liés au système d’exploitation, tels que des changements de contextes.

Flater [18] met néanmoins en garde les développeurs et avise de bonnes pratiques à respecter lors du développement d’applications C sous Linux pour tirer le meilleur des capacités des profileurs tels que perf.

Comme le montre [52], même si perf dispose de concurrents dans le secteur du profilage Linux, il reste incontestablement le meilleur en termes de performance et de simplicité d’utilisation.

CCStudio

CCStudio[4] est un outil propriétaire développé par TI et qui est le principal (si ce n’est l’unique) outil de débogage et de profilage des DSPs et coprocesseurs TI. Il permet entre autres l’acquisition de statistiques sur l’usage des fonctions (nombre de fois qu’une fonction a été appelée, temps passé dans chaque fonction…) ou encore de voir le contenu des registres, les tâches couramment exécutées etc…

Si cet outil est très pratique et relativement complet, il n’en demeure pas moins insuffisant vis-à-vis du traçage. L’un des objectifs de ce travail est de montrer comment le traçage des DSPs peut

---

[4] http://www.ti.com/tool/ccstudio

diminuer significativement le temps passé à chercher des problèmes et des optimisations sur cet IDE, améliorant ainsi les conditions de développement sur de tels appareils.

2.3.3 Quelques outils de traçage classiques

Gregg [14] souligne avec amusement qu'il existe un très grand nombre de traceurs pour Linux. En effet, puisqu'une simple macro, nommée *TRACE_EVENT* permet la création de points de trace universels (*i.e* ne dépendant d'aucun traceur particulier), il est possible d'utiliser n'importe lequel de ces traceurs [67]. Certains traceurs peuvent également tirer parti de l'instrumentation dynamique offerte par *kprobes*, au détriment des performances.

Le fonctionnement global de ces traceurs est par ailleurs assez simple : lors de l'exécution du code, les points de traces évaluent une condition pour savoir s'ils ont été activés ou non par un traceur. Si oui, un évènement est généré et mis en tampon par ledit traceur. Dans le cas contraire, le seul surcoût engendré est celui de la condition, limitant ainsi l'impact des points de trace non activés au strict minimum, en accord avec la philosophie du traçage. Cela permet donc de conserver de nombreux points de trace à travers le noyau Linux sans pénalité (ou presque) en termes de performances.

Nous décrivons ici brièvement trois traceurs communs pour le noyau Linux : *strace, ftrace* et, celui qui sera utilisé dans nos expérimentations, *LTTng*. Le choix d'un traceur dépendra dans la réalité des besoins (informations à tracer, vitesse d'acquisition, précision des résultats…) de l'utilisateur. En outre, il est à noter que, bien que ces traceurs utilisent les mêmes mécaniques pour acquérir les évènements et générer des traces, leurs formats de sortie sont souvent différents, ce qui peut limiter les outils utilisables dans une étape d'analyse des résultats obtenus.

strace

strace[5] est un outil extrêmement simple et puissant traçant l'ensemble des appels systèmes, ainsi que leurs arguments et valeurs de retours, effectués par une application. Il permet également des opérations de profilage, comme l'analyse du temps total passé dans le noyau par l'application ou dans les appels systèmes par exemple.

---

[5] http://linux.die.net/man/1/strace

Bien que ses fonctionnalités soient quelque peu rudimentaires, strace permet des analyses approfondies de systèmes à des fins d'analyse ou de débogage rapides. Dayang et Tchakounté [57] montrent par exemple comment strace peut être utilisé pour observer les comportements d'applications Android à travers les appels systèmes afin de détecter des comportements malicieux.

La figure 3 donne un exemple des résultats produits par strace.

```
[01:15] skinner@sweet-E6420 :testsC $ strace ./quicksort
execve("./quicksort", ["./quicksort"], [/* 66 vars */]) = 0
brk(0)                                  = 0x9c3000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab0514c000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=166366, ...}) = 0
mmap(NULL, 166366, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fab05123000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`\v\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1869392, ...}) = 0
mmap(NULL, 3972864, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fab04b61000
mprotect(0x7fab04d21000, 2097152, PROT_NONE) = 0
mmap(0x7fab04f21000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7fab04f21000
mmap(0x7fab04f27000, 16128, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fab04f27000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab05122000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab05121000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab05120000
arch_prctl(ARCH_SET_FS, 0x7fab05121700) = 0
mprotect(0x7fab04f21000, 16384, PROT_READ) = 0
mprotect(0x601000, 4096, PROT_READ)     = 0
mprotect(0x7fab0514e000, 4096, PROT_READ) = 0
munmap(0x7fab05123000, 166366)          = 0
brk(0)                                  = 0x9c3000
brk(0x9e4000)                           = 0x9e4000
exit_group(0)                           = ?
+++ exited with 0 +++
```

Figure 3 : Exemple de sortie de *strace* sous Linux

ftrace

ftrace[6] est un traceur faisant partie intégrante du noyau Linux et capable de se brancher sur tous les points de trace du noyau ainsi que sur les points dynamiques générés par *kprobes*. Il permet des acquisitions rapides de traces ne nécessitant l'intervention d'aucune autre application.

Comme le montre Ficheux [35], ftrace offre de nombreuses possibilités allant du simple traçage d'appels systèmes à la construction complète de l'arbre d'appel des fonctions du noyau utilisées durant la session de traçage.

Même si Gregg [14] regrette qu'il ne soit pas directement programmable, ftrace permet l'ajout de greffons permettant d'étendre ses fonctionnalités.

---

[6] https://www.kernel.org/doc/Documentation/trace/ftrace.txt

LTTng

LTTng[7] est un traceur très performant, offrant de nombreuses fonctionnalités et ayant pour objectif d'avoir le moins d'impact possible sur le système étudié [59]. C'est pour cela qu'il a été choisi pour réaliser ce travail, dans l'optique de tracer les composants « maîtres » basés sur Linux.

LTTng peut s'interfacer sur tous les points de trace du noyau, qu'ils soient statiques ou dynamiques. Les évènements collectés sont stockés dans des tampons mémoire, un tampon étant associé à chaque cœur du système.

À la différence de traceurs axés sur le noyau Linux, LTTng permet également le traçage d'évènements en espace utilisateur. Dans ce cas, l'utilisateur doit spécifier les points de traces utilisés (nom de domaine, nom du point de trace, charge utile…) à la main, offrant ainsi des possibilités quasi-illimitées [15].

Le principal atout de LTTng est d'introduire un format de traçage universel : le *Common Trace Format*, ou CTF[8]. Ce format binaire extrêmement optimisé, destiné à utiliser le moins de mémoire possible lors de l'enregistrement des évènements, permet par exemple la définition d'entiers de tailles non conventionnelles (3, 5, 42 bits…). Bien entendu, ce format n'est pas destiné à être lu directement par un humain et requiert l'utilisation d'un parseur externe.

Le parseur se base sur les *métadonnées* accompagnant les traces et qui représentent la définition des points de traces utilisés, permettant ainsi de distinguer les différents évènements dans la trace binaire.

LTTng offre en outre d'autres fonctionnalités intéressantes mais sans applications directes avec ce travail. Par exemple, il est possible de ne garder en mémoire que les n derniers évènements enregistrés et de vider le tampon dans une trace lors d'une erreur. Ce mode de traçage « à la volée » permet de ne consommer que le minimum de mémoire nécessaire et peut être un excellent moyen de récupérer de l'information sur une exécution spécifique du système.

---

[7] https://lttng.org

[8] http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master

Conformément à l'idée que les points de trace du noyau peuvent être activés ou non, LTTng permet de choisir les évènements qu'il doit enregistrer. Cela permet d'accéder directement à de l'information pertinente sur l'exécution d'un système tout en réduisant au minimum la surcharge induite par le traçage.

Finalement, LTTng est très facile d'utilisation et ne nécessite que l'appel de quelques commandes Linux pour créer une session de traçage, définir les évènements à enregistrer et lancer le traçage à proprement parler [64].

2.3.4 Traçage sur plateforme hétérogène embarquée

LTTng et ARM

Remarquons tout d'abord que, puisque LTTng est construit pour ne pas dépendre d'une architecture particulière, il est tout à fait possible de l'utiliser sur n'importe quel système faisant tourner Linux. Certains prérequis subsistent néanmoins, tels que la présence de compteurs suffisamment précis (sur 64bits) pour pouvoir générer des estampilles de temps qui, rappelons-le, peuvent avoir une granularité de l'ordre de la nanoseconde [22].

Ainsi, LTTng est tout à fait compatible avec les architectures ARM qui représentent la base des systèmes étudiés dans ce travail (architecture du maître). Il est néanmoins à noter que, puisque LTTng n'est pas directement intégré au noyau Linux (contrairement à *ftrace* par exemple), cela nécessite la cross-compilation de ses éléments sur le système ciblé. Cette étape, d'apparence non complexe, nécessite le plus grand soin dans sa réalisation pour assurer le bon fonctionnement de LTTng.

Défis

Tracer des plates-formes hétérogènes embarquées est un défi de taille car il nécessite le traçage du maître (processeurs ARM dans notre cas), des coprocesseurs (génériques ou DSPs) et la mise en relation des traces obtenues, l'objectif étant d'obtenir une vue pertinente globale de tous les évènements du système.

Arriver à une solution universelle nécessite de tenir compte des problèmes suivants :

- L'utilisation d'un format de trace commun aux deux parties serait préférable.
- Les coprocesseurs n'utilisent pas de système d'exploitation (ils sont bare-metal).

- Il n'y a généralement que peu de mémoire disponible sur la plateforme pour stocker les traces.
- Les coprocesseurs ne disposent souvent que de peu de mémoire interne destinée à accueillir à la fois le code de l'application et ses données (dont font partie les tampons de traçage).
- Les esclaves doivent pouvoir envoyer les paquets de traçage à leur maître.
- Les différents éléments tracés ne sont pas alignés sur la même origine d'estampille de temps (horloges physiques différentes), ne tournent probablement pas à la même fréquence et ont des politiques de modulation de la fréquence différentes en fonction de la charge.

Ce sont principalement ces problèmes qui nécessitent d'être adressés si l'on veut produire une solution générique, applicable à toutes les plates-formes hétérogènes embarquées.

Travaux liés

Peu de travaux ont été effectués en ce sens et la plupart des utilisateurs de ces plates-formes se contentent de l'utilisation d'outils fournis par les fabricants ou créés « à la main » pour le système utilisé. Aucune solution générique de traçage à proprement parler n'a encore été proposée.

Le travail de David Couturier [38] montre comment il est possible, en interceptant des appels OpenCL, de tracer des GPGPUs en utilisant le même format que LTTng (le CTF). Cela permet une indépendance complète du programmeur vis-à-vis des solutions de traçage, ou plus souvent de profilage, proposées par les constructeurs. De tels travaux forment une base solide pour l'unification des traces entre les CPUs et les coprocesseurs sur des plates-formes hétérogènes.

Introduction à *barectf*

Introduit par Philippe Proulx en 2014, *barectf*[9] est un outil écrit en Python et ciblant spécifiquement les systèmes bare-metal. Destiné à tenir compte des restrictions s'appliquant aux processeurs embarqués les plus exigeants, il permet l'utilisation universelle de points de trace en format CTF.

En particulier, Proulx montre dans [11] et [28] comment il est possible d'utiliser son outil pour générer des traces CTF sur les coprocesseurs de la carte Parallella, qui fait l'objet de la première

[9] https://github.com/efficios/barectf

partie de ce travail. Les résultats obtenus sont particulièrement impressionnants compte tenu du fait que la puce Epiphany n'offre que 32Ko de mémoire locale pour chaque coprocesseur et que barectf arrive tout de même à s'y intégrer avec le reste de l'application chargée.

Son fonctionnement global est simple : l'utilisateur commence par décrire les métadonnées de la trace à acquérir en spécifiant le contexte ainsi que les types d'évènements et les informations qu'ils contiennent. Une fois cela fait, barectf parcourt le fichier de configuration et produit des fonctions C faisant office de points de trace. Puisque le format des traces générées est le CTF, barectf fournit automatiquement toutes les macros et méthodes utilisées pour sérialiser les évènements dans le format binaire.

À partir de là, l'utilisateur doit encore instrumenter son application et la lier avec les fichiers produits par barectf et, plus important encore, il doit écrire les fonctions en charge du traitement des évènements enregistrés sur le processeur bare-metal.

L'ensemble de ces fonctions, dénommé « plate-forme barectf », consiste à traiter le stockage des évènements générés sur le processeur tracé sous forme de paquets (équivalent au stockage des évènements dans un tampon mémoire par un traceur classique). Il est notamment nécessaire de fournir des fonctions capables de décider du sort des paquets ou du tampon local, une fois remplis. Généralement, lorsqu'un paquet est plein il sera transféré à un processus démon roulant sur le maître et en charge d'écrire ledit paquet dans une trace, permettant ainsi d'utiliser les APIs Linux. Néanmoins, puisque le comportement à adopter est à la charge du développeur, celui-ci peut très bien décider de supprimer le paquet sous certaines conditions ou de l'envoyer directement dans une zone mémoire partagée ou encore de générer une interruption sur l'hôte etc…

L'écriture de cette plate-forme, dépendante du système étudié, est indispensable par construction de barectf. En effet, puisque cet outil a pour but d'être le plus générique possible, il ne peut pas fournir de méthode de gestion des paquets génériques, car aucune fonction ne sera garantie de fonctionner sur tous les systèmes, dont les fonctionnalités fournies peuvent grandement diverger (par exemple, tous les systèmes embarqués hétérogènes n'ont pas un support complet de la librairie C et certains ne gèrent pas les fonctions d'allocation mémoire dynamique (de type *malloc*)).

Si Proulx fourni une plate-forme fonctionnelle pour la carte Parallella, une partie de ce travail consiste à porter barectf sur la Keystone 2 de TI. L'avantage du fonctionnement par plate-forme

est de permettre à chacun d'utiliser toutes les optimisations disponibles pour traiter les paquets d'évènements le plus efficacement possible.

En somme, l'instrumentation d'un système par barectf peut être résumée par la figure suivante :

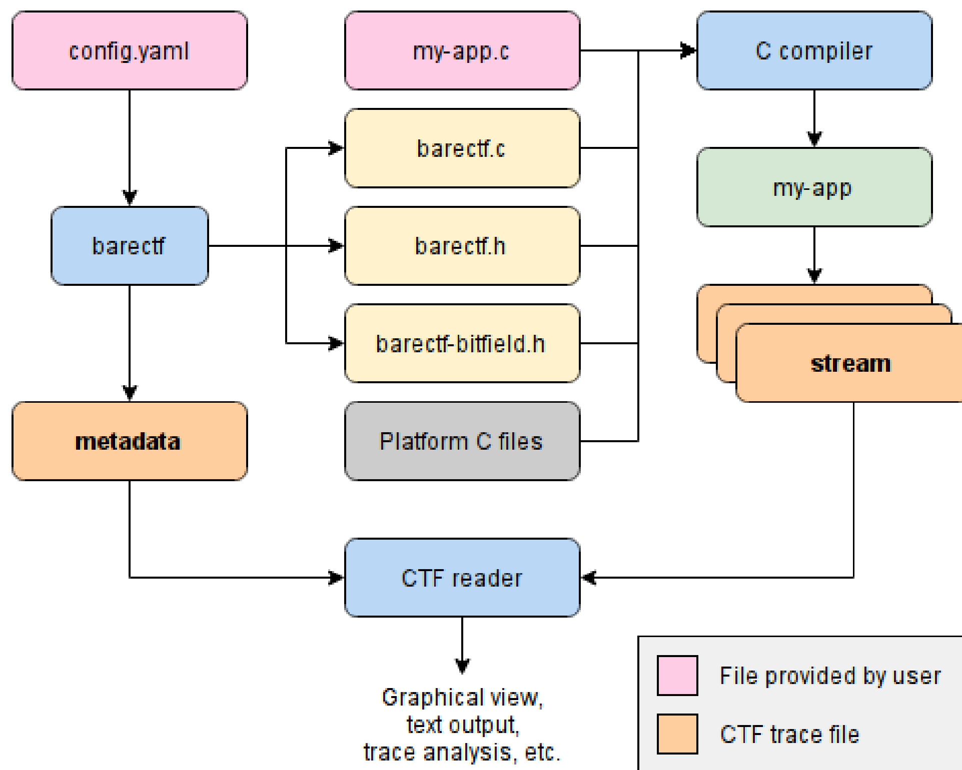

Figure 4 : Instrumentation d'un programme avec barectf

Tirée de [11] et reproduite avec permission.

2.3.5 Analyse des traces collectées

Une fois les traces collectées sur un système, il est nécessaire de pouvoir les analyser, que ce soit de manière statistique ou graphique. Divers utilitaires permettent de parser des traces et de transformer le format binaire en un format facilement lisible par un humain.

babeltrace

Babeltrace[10] est l'outil canonique de lecture de traces CTF. Il convertit simplement le format binaire en un format humainement compréhensible en se basant sur les métadonnées embarquées

[10] https://www.efficios.com/fr/babeltrace

avec la trace. Il permet en outre d'effectuer divers post-traitements sur la trace collectée et de personnaliser l'affichage des données.

La figure 5 représente une partie d'une trace noyau en format CTF vue par babeltrace et exhibant des appels systèmes effectués lors d'une session de traçage :

```
[06:24:39.548703173] (+0.000004205) k2hk-evm net_dev_xmit: { cpu_id = 0 }, { skbaddr = 3714465344, rc = 0, len = 1514, name = "eth0" }
[06:24:39.548706668] (+0.000003495) k2hk-evm irq_handler_entry: { cpu_id = 0 }, { irq = 92, name = "hwqueue-8716" }
[06:24:39.548709348] (+0.000002680) k2hk-evm irq_softirq_raise: { cpu_id = 0 }, { vec = 3 }
[06:24:39.548710453] (+0.000001105) k2hk-evm irq_handler_exit: { cpu_id = 0 }, { irq = 92, ret = 1 }
[06:24:39.548714128] (+0.000003675) k2hk-evm net_dev_xmit: { cpu_id = 0 }, { skbaddr = 3714464768, rc = 0, len = 1514, name = "eth0" }
[06:24:39.548715188] (+0.000001060) k2hk-evm skb_consume: { cpu_id = 0 }, { skbaddr = 3676989824 }
[06:24:39.548717058] (+0.000001870) k2hk-evm irq_softirq_entry: { cpu_id = 0 }, { vec = 3 }
[06:24:39.548719598] (+0.000002540) k2hk-evm skb_consume: { cpu_id = 0 }, { skbaddr = 3714465344 }
[06:24:39.548720883] (+0.000001285) k2hk-evm kmem_kfree: { cpu_id = 0 }, { call_site = 0xC03DCF98, ptr = 0xDD375C00 }
[06:24:39.548722013] (+0.000001130) k2hk-evm kmem_cache_free: { cpu_id = 0 }, { call_site = 0xC029F64C, ptr = 0xDD663E40 }
[06:24:39.548723638] (+0.000001625) k2hk-evm skb_consume: { cpu_id = 0 }, { skbaddr = 3714464768 }
[06:24:39.548724683] (+0.000001045) k2hk-evm kmem_kfree: { cpu_id = 0 }, { call_site = 0xC03DCF98, ptr = 0xDD3EE600 }
[06:24:39.548726758] (+0.000002075) k2hk-evm irq_handler_entry: { cpu_id = 0 }, { irq = 80, name = "hwqueue-8704" }
[06:24:39.548729043] (+0.000002285) k2hk-evm irq_softirq_raise: { cpu_id = 0 }, { vec = 3 }
[06:24:39.548730043] (+0.000001000) k2hk-evm irq_handler_exit: { cpu_id = 0 }, { irq = 80, ret = 1 }
[06:24:39.548733103] (+0.000003060) k2hk-evm kmem_cache_free: { cpu_id = 0 }, { call_site = 0xC029F64C, ptr = 0xDD663C00 }
[06:24:39.548736368] (+0.000003265) k2hk-evm napi_poll: { cpu_id = 0 }, { napi = 3713997936, dev_name = "eth0" }
[06:24:39.548738388] (+0.000002020) k2hk-evm kmem_cache_alloc: { cpu_id = 0 }, { call_site = 0xC03DB680, ptr = 0xDD663C00, bytes_req = 192, bytes_
alloc = 192, gfp_flags = 32 }
[06:24:39.548740358] (+0.000001970) k2hk-evm kmem_cache_free: { cpu_id = 0 }, { call_site = 0xC0349C1C, ptr = 0xDDBD1C00 }
```

Figure 5 : Exemple de sortie de babeltrace

TraceCompass

L'outil libre TraceCompass[11], développé par la compagnie Ericsson, introduit la visualisation graphique de traces, permettant ainsi de plus facilement distinguer la cause de problèmes « à la main » ou à l'aide de fonctionnalités intégrées.

TraceCompass représente une trace comme un ensemble d'états liés entre eux par différentes propriétés dans un arbre. Chaque processus tracé est ainsi visualisé comme une suite d'états (en marche, bloqué, en attente du CPU…). S'il n'est en général pas très utile (voire même contre-productif) d'étudier les états d'un processus isolé, il est possible de visualiser les liens de cause-à-effet entre différents processus. Par exemple, il est possible de voir que le processus x s'est retrouvé bloqué parce qu'il attendait une ressource détenue par le processus y.

Par défaut, TraceCompass offre tous les services nécessaires à la visualisation de traces du noyau Linux. En particulier, la représentation graphique des états du système permet de visualiser les tâches exécutées, de distinguer les différents processus présents et les processeurs sur lesquels ils

[11] http://tracecompass.org/

se sont exécutés ou encore de voir à quel moment un processus s'est retrouvé en attente d'une ressource et a donc atteint l'état « bloqué ». Il est également possible d'utiliser l'arbre d'états créé et de le modifier pour produire des vues adaptées aux problèmes étudiés.

Par exemple, si l'on sait que les points de trace d'un système correspondent aux entrées et aux sorties des fonctions appelées, il est aisé de construire une vue représentant la pile d'appel des fonctions utilisées par l'application durant la session de traçage. C'est d'ailleurs une vue de la sorte qui sera utilisée dans ce travail pour analyser des problèmes de latence d'une application de traitement d'images sur des DSPs.

La figure ci-dessous donne un aperçu de TraceCompass et de sa vue permettant de visualiser les traces noyaux Linux.

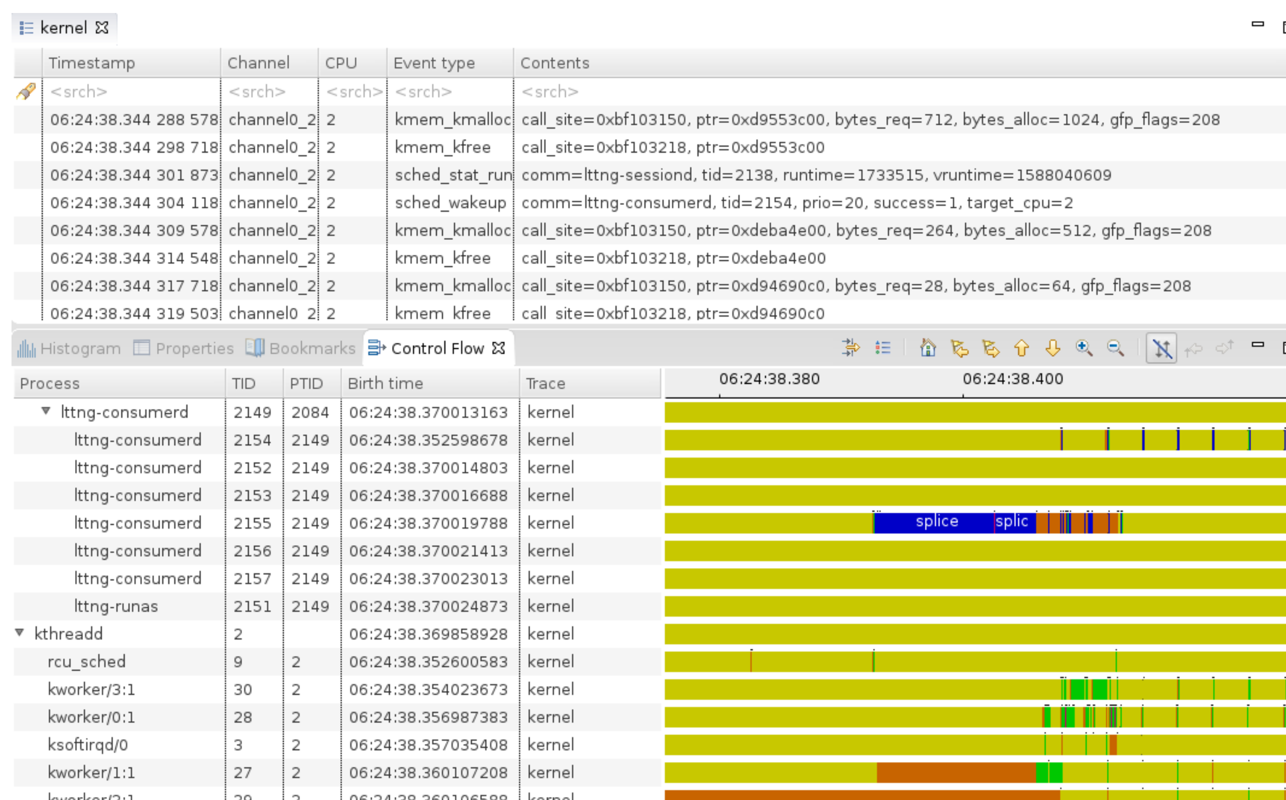

Figure 6 : Exemple de visualisation de trace noyau par TraceCompass

2.4 De la nécessité de la synchronisation

Le but du travail présenté ici est d'analyser des systèmes embarqués hétérogènes et plus précisément les interactions entre un ou plusieurs « maîtres » et leurs « esclaves ». La revue des

documents présentés plus haut permet d'affirmer qu'un traceur tel que LTTng est suffisant pour tracer les processus maîtres (généralement sur des processeurs de type ARM) et qu'un outil comme barectf permet de tracer les processus esclaves sur des coprocesseurs.

Néanmoins, puisque les traces proviennent de matériels différents, leurs estampilles de temps ne sont en aucun cas corrélées car les origines de temps ne sont pas partagées et que leurs fréquences de fonctionnement sont potentiellement différentes. Les comparer directement n'a donc pas de sens et le besoin de les synchroniser sur une même échelle de temps, pour conserver les rapports de cause-à-effet, devient évident.

Cette partie présente donc des techniques de synchronisation de processus classiques dans le but de voir lesquelles pourraient être appliquées à notre étude.

2.4.1 Techniques de synchronisation dans les systèmes distribués

Le problème énoncé plus haut dans le cas des systèmes embarqués hétérogènes n'est pas étranger au même problème rencontré dans les systèmes répartis : puisque différentes machines communiquent, il est nécessaire de synchroniser leurs échanges, toujours dans une optique de conservation des rapports de cause-à-effet. Tanenbaum [26] décrit quelques algorithmes permettant de remédier à ce problème dans les systèmes distribués.

Network Time Protocol (NTP)

NTP [33] est un protocole permettant une synchronisation des horloges entre différentes machines d'un système réparti. Son fonctionnement repose sur l'établissement d'un serveur de temps central dont l'objectif est de donner l'heure courante du système à la machine en faisant la demande.

Puisque des délais sont inévitables dans les communications réseau ou que des paquets peuvent se perdre, une partie importante du serveur consiste à donner la meilleure estimation du temps réel en fonction de plusieurs échanges entre lui et la machine cible en tenant compte des latences.

La machine faisant la demande de l'heure courante ajuste alors son horloge interne en fonction de la réponse du serveur.

L'algorithme de Berkeley

Cet algorithme repose sur les mêmes fondements que le protocole NTP : un serveur central est en charge d'indiquer aux éléments du réseau les ajustements à effectuer sur leurs propres horloges internes pour atteindre un état de synchronisation global [58].

Néanmoins, à la différence de NTP, où les différentes machines du réseau effectuent des requêtes périodiques au serveur central, ici c'est le serveur central qui est en charge d'envoyer des requêtes aux éléments du système.

Plus concrètement, le serveur demande périodiquement à chaque machine l'heure courante relevée localement et calcule une moyenne des résultats reçus. Cette moyenne sert ensuite d'heure commune et le serveur envoie les ajustements nécessaires à effectuer sur chaque machine.

Si les deux solutions précédentes peuvent être acceptables dans des systèmes distribués, elles ne le seront pas dans notre contexte où, puisque l'on souhaite synchroniser des évènements dont la granularité est de l'ordre de la nanoseconde, la précision ne serait pas suffisante.

Horloge logique de Lamport

Dans un article de 1978 [61], Lamport explique comment bien souvent une synchronisation des horloges (temps absolus) n'est pas nécessaire car seuls les liens de causalités ont réellement de l'importance (*i.e* on veut toujours pouvoir être en mesure de savoir quel évènement d'un système est arrivé avant un autre).

En décrivant formellement cette relation « d'arrivée avant », Lamport propose un moyen reposant sur des compteurs logiques (*i.e* des compteurs incrémentés par l'arrivée ou l'envoi d'évènements) d'assurer les liens de causalité d'un système.

Cette technique implique néanmoins l'ajout des données des compteurs logiques dans chaque message. Pour cette raison, et parce que le traçage précis d'un système requiert une synchronisation absolue des horloges, cette solution ne sera pas utilisée dans ce travail.

2.4.2 L'algorithme de l'enveloppe convexe

Synchronisation post-traçage

La synchronisation de traces par l'algorithme de l'enveloppe convexe, telle que décrite dans [3] et [55], est une méthode qui repose sur la génération de paires d'évènements entre la trace de référence (notée « A ») et la trace à synchroniser (notée « B »). Cette technique, présentée dans le cadre

d'échanges TCP, est utilisée et étendue par ce travail dans le cadre de la synchronisation de traces dans un environnement hétérogène embarqué. L'objectif de cet algorithme est de trouver la transformation linéaire appropriée à appliquer aux estampilles de temps de la trace B pour la synchroniser sur la trace A.

Son principe est simple : lors d'un échange TCP, un système de « serrage de main » est mis en place, dans lequel un émetteur envoie un message puis attend la confirmation de la réception de son message sur le récepteur. Le récepteur de son côté attend le message et envoie la confirmation de réception, constituant ainsi les différentes étapes du *handshake*.

En plaçant des évènements lors de l'envoi depuis A et de la réception sur B d'un message et lors de l'envoi depuis B et la réception sur A de son accusé de réception, on obtient deux paires d'évènements étroitement liés sur les deux systèmes à synchroniser (deux évènements d'une même paire représentant ainsi le délais de communication d'un message). L'élément important à remarquer ici est que ces deux paires ont des directions différentes, représentant les directions réelles des communications : la première va de A vers B et la seconde de B vers A.

La remarque précédente prend toute son importance lors de l'étape de traitement des traces collectées. En effet, au moyen de l'algorithme de l'enveloppe convexe, les deux groupes de paires d'évènements (de A vers B et de B vers A) vont produire deux ensembles de points distincts servant à fabriquer les deux parties de l'enveloppe (voir figure 7).

La transformation linéaire recherchée se trouvera nécessairement entre les deux morceaux de l'enveloppe, approximés par les transformations de plus forte et de plus basse pente contenues dans l'enveloppe, afin d'éviter des effets d'inversion d'évènements.

Puisque toutes les transformations à l'intérieur de l'enveloppe convexe sont des candidats valables à la fonction de synchronisation, Duda et al. [31] suggèrent d'utiliser la bissectrice de l'angle formé par les deux transformations ayant la plus forte (respectivement la plus basse) pente.

Généralement, ce processus peut être effectué au complet par des utilitaires de traitement de traces tels que TraceCompass, la seule responsabilité de l'utilisateur étant de produire suffisamment de paires d'évènements pour rendre la synchronisation efficace.

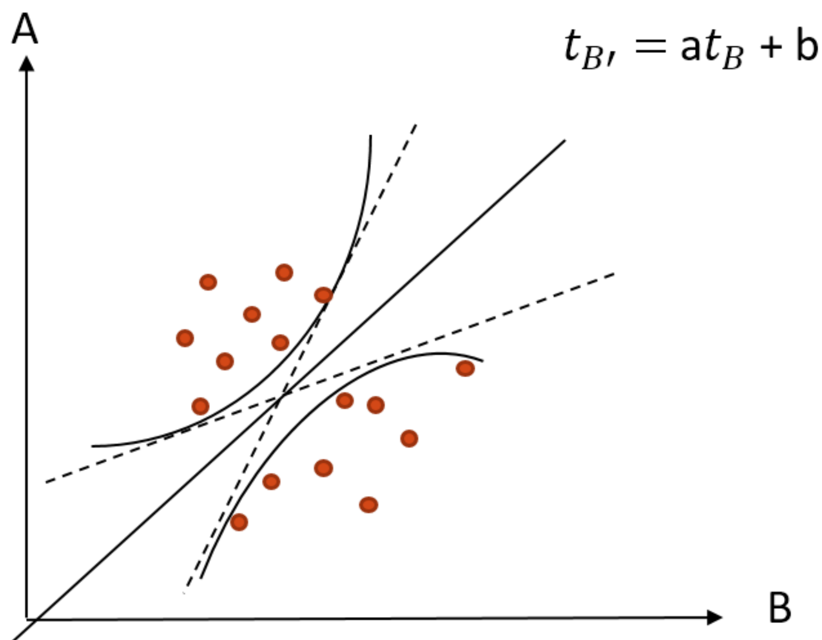

Figure 7 : Exemple de représentation de l'enveloppe convexe

Synchroniser « à la volée »

Une autre méthode, basée sur la précédente, existe pour synchroniser les traces au fur et à mesure de leur acquisition [44]. Reposant encore une fois sur l'algorithme de l'enveloppe convexe, cette technique limite l'ensemble des paires à traiter à l'aide d'une structure en arbre.

Pour limiter au maximum l'impact du traçage sur les systèmes étudiés, ce travail se basera sur les mécanismes de synchronisation post-acquisition de traces.

2.5 Conclusion de la revue de littérature

Le traçage est une technique incontournable de collecte d'informations et d'analyse de systèmes, simples ou complexes. Si des outils dédiés au traçage sont déjà présents dans les systèmes Linux, il n'existe pas de méthode universelle de traçage pour les processeurs sans système d'exploitation, et notamment pour les coprocesseurs dans des environnements hétérogènes embarqués. En conséquence, ce travail de recherche s'appuie sur les différents outils et techniques exposés plus haut pour construire une solution de traçage générique des systèmes hétérogènes embarqués, tenant compte de leurs particularités et des besoins de synchronisation des traces. Le chapitre suivant présente la méthodologie employée pour produire la solution présentée.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Définition du problème

Ce travail s'intéresse aux plates-formes hétérogènes embarquées, c'est-à-dire aux plates-formes embarquées disposant d'au moins deux sortes de processeurs différents. Dans la plupart des cas, les environnements qui nous intéressent sont composés de processeurs classiques et de coprocesseurs de calcul.

Le problème que ce travail entend résoudre est le manque d'outils génériques capables de tracer toutes les entités d'un tel système en même temps et de corréler les traces obtenues pour pouvoir les analyser. Présentement, les développeurs utilisent principalement des solutions propriétaires de profilage et/ou des outils produits par les fabricants des plates-formes étudiées, ce qui ne permet pas d'avoir une compréhension aussi poussée et globale qu'avec une solution de traçage.

L'objectif de ce mémoire est donc de démontrer comment il est possible (et utile) de tracer conjointement les processeurs classiques et les coprocesseurs dans des environnements hétérogènes embarqués, et d'aboutir à des analyses complètes et adaptées aux besoins de chacun. Pour cela, nous pouvons découper la problématique principale en divers sous-problèmes :

3.1.1 Vérification de l'intégration des outils classiques

Comme vu lors de la revue de littérature, LTTng, le traceur Linux utilisé dans ce travail, peut être porté sur une large majorité d'architectures. En particulier, le faire fonctionner sur un processeur ARM, représentant bien souvent le cœur des plates-formes hétérogènes étudiées, est un prérequis à la réalisation de la solution finale.

En conséquence, vérifier la compatibilité de LTTng avec les systèmes étudiés est la priorité numéro une. Comme souligné dans la revue de littérature, cela passe par des étapes de cross-compilation des modules existants.

3.1.2 Vérification du fonctionnement de *barectf*

Barectf étant présentement à l'état de prototype, une étape importante de l'établissement de notre solution consiste à le tester en profondeur et à adapter ses fonctionnalités sur les plates-formes étudiées.

Dans un premier temps, nous reprenons le travail de Philippe Proulx sur la carte Parallella, afin d'étudier et d'analyser le fonctionnement de barectf. Cette étape permettra en outre d'effectuer des premiers tests capables de faire ressortir d'éventuels problèmes.

Il est également nécessaire de porter barectf sur les autres plates-formes étudiées, c'est-à-dire d'écrire les modules C en charge de la gestion des paquets de trace, tenant compte des spécificités des environnements ciblés.

3.1.3 Définition et implémentation d'une solution de synchronisation de traces

Tel qu'attendu, tracer un système hétérogène produit des traces dont les estampilles de temps ne peuvent pas être directement comparées. La revue de littérature du chapitre précédent donne un aperçu de différentes méthodes de synchronisation de traces, destinées à fusionner les différentes traces obtenues en une seule et même trace, unifiant ainsi les estampilles de temps.

L'analyse des techniques classiques nous amène à penser qu'une solution basée sur un traitement post-acquisition des traces serait plus adaptée à notre problématique. Comme indiqué précédemment, cela requiert la génération de paires d'évènements sur les différentes parties du système étudié.

Proposer une solution générique permettant de synchroniser des traces hétérogènes par la production de paires d'évènements est l'étape finale conduisant à l'obtention d'un premier prototype.

3.1.4 Prototypage

Une fois l'ensemble des éléments principaux de la solution mis en place, il est nécessaire d'effectuer une batterie de tests sur des plateformes représentatives. À cet effet, les plates-formes Parallella et Keystone 2 (présentées en détails dans la section suivante), seront utilisées.

La phase de prototypage est également l'occasion d'améliorer les techniques mises en place et d'analyser les portions théoriques ou pratiques nécessitant d'être retravaillées.

3.1.5 Test de la solution sur un problème réel

Puisque l'objectif de ce mémoire est de démontrer le fonctionnement du traçage hétérogène, il est essentiel de produire un exemple concret de ce qu'il est possible d'accomplir sur un problème réel.

N’ayant néanmoins pas de problème directement relié à ceux auxquels font face les industries à notre disposition, nous proposons l’étude d’une application de traitement d’images sur des DSPs. En simulant des problèmes de concurrence et en introduisant des latences forcées sur certains des DSPs, on entend démontrer que le traçage est une solution adaptée et fonctionnelle pour résoudre des problèmes communs, autrement plus difficiles à détecter à l’aide d’autres méthodes.

3.2 Présentation des plates-formes de test

Afin de développer et de tester la solution proposée dans ce mémoire, deux plates-formes ont été retenues. La première, la carte Parallella d’Adapteva, repose sur la présence de deux cœurs ARM Cortex A9 et d’une puce Epiphany composée de 16 coprocesseurs génériques. Un module FPGA vient par ailleurs assurer la liaison entre les processeurs principaux et les coprocesseurs. De par son modèle *open-source, open-hardware* et son architecture tournant autour du module FPGA, il s’agit d’un parfait modèle de prototypage pour les industries.

La seconde, un module d’évaluation du système sur puce Keystone 2 de Texas Instrument, est une carte plus mature et articulée autour de 4 ARM Cortex A15 et 8 DSPs C66x. Ces DSPs, largement utilisés en industrie, permettent la création de systèmes robustes et efficaces. Pouvoir les tracer comme n’importe quel autre processeur permettrait certainement d’en tirer le meilleur en termes de performance et de temps de développement.

3.2.1 Adapteva Parallella

3.2.1.1 Architecture matérielle

Le principal intérêt de la carte Parallella est la présence de la puce Epiphany, regroupant 16 coprocesseurs à usage général. Le design de cette puce repose sur une formation d’un réseau de coprocesseurs. Ainsi, chaque coprocesseur, appelé *eCore* est vu comme un processeur couplé à un routeur, l’ensemble formant un *nœud* du réseau.

Les nœuds sont reliés entre eux et vers l’extérieur de la puce par trois canaux de communication distincts :

- *rMesh* : utilisé pour les lectures;
- *cMesh* : utilisé pour les écritures sur la puce;

- *xMesh* : utilisé pour les écritures externes à la puce.

Chaque eCore est un constitué d'un processeur mono-cœur RISC 32 bits cadencé à 1Ghz et disposant de 35 instructions. Les eCores sont programmables en C/C++ ou OpenCL (le C est le langage ayant été choisi pour cette étude). Ils disposent chacun de 32 Ko de mémoire locale, destinée à accueillir en priorité les applications à exécuter.

La figure suivante dresse un schéma de haut-niveau de l'architecture de la puce Epiphany :

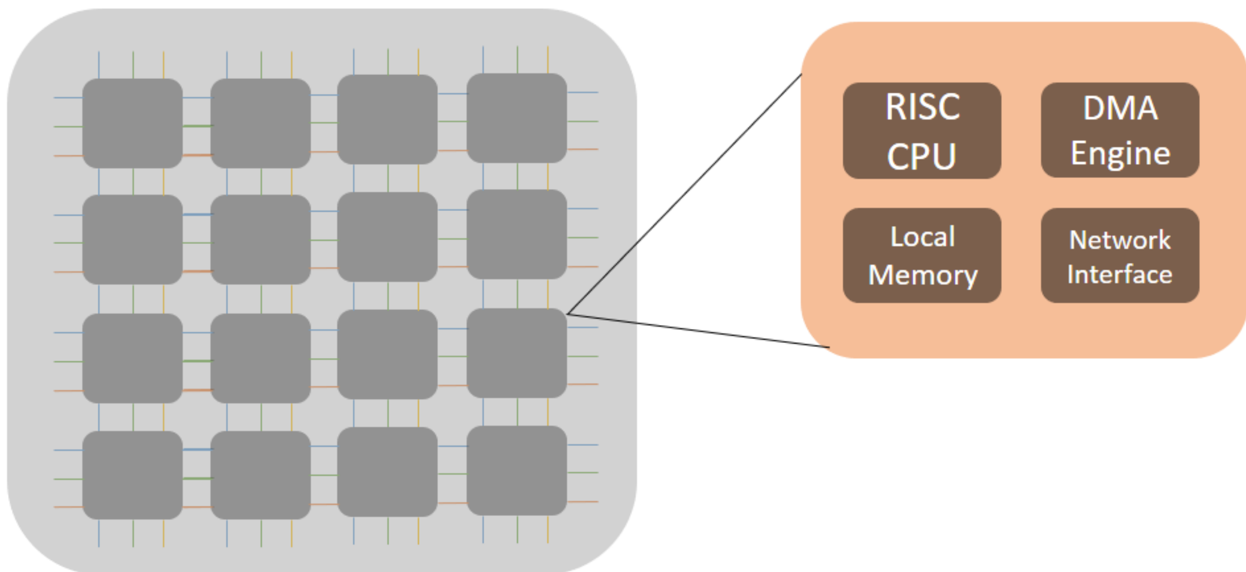

Figure 8 : Architecture haut-niveau de la puce Epiphany

1 Go de mémoire RAM est également disponible. Par défaut, 32Mo sont alloués par le noyau Linux s'exécutant sur l'ARM pour servir de médium de communication entre les processeurs et les eCores.

3.2.1.2 Architecture logicielle

Les processeurs maîtres, ayant accès à Linux, sont programmables à travers les librairies standards C. Les coprocesseurs, en revanche, sont utilisés sans système d'exploitation (*bare-metal*) et ne disposent donc pas des mêmes fonctionnalités.

Leur programmation se fait via des APIs C fournies par Adapteva. Ces APIs reprennent une partie des librairies standards C et certaines fonctionnalités (telles que l'allocation mémoire dynamique) ne sont pas disponibles.

Adapteva fournit également les outils nécessaires aux processus maîtres pour charger une application sur les eCores et contrôler leur état (en marche, en attente, arrêté…). Néanmoins, aucun moyen de communication inverse (de l'esclave vers le maître) n'est fourni. Ceci implique par exemple qu'un esclave n'a aucun moyen direct de communiquer la fin d'une tâche à son maître.

L'utilisation de la zone de mémoire partagée pour assurer les communications entre processeurs et coprocesseurs est donc obligatoire.

3.2.2 Texas Instrument Keystone 2

Pour tester le système sur puce Keystone 2 de Texas Instrument, la carte d'évaluation EVMK2H[12] a été utilisée. Par abus de langage, on désigne dans ce qui suit cette plate-forme par « Keystone 2 ».

3.2.2.1 Architecture matérielle

Embarquant en son centre quatre processeurs ARM Cortex A15, la Keystone 2 met à disposition de l'utilisateur huit DSPs C66x, cadencés à 1,2 GHz. Les performances annoncées vont jusqu'à 19,2 GFlops par cœur. Chaque DSP dispose en plus de trois couches de cache mémoire, pour des performances accrues.

Le système dispose en outre de 2 Go de mémoire RAM extensible et de 6 Mo de mémoire partagée dédiée.

La figure suivante présente une vue globale des composants du système sur puce :

[12] http://www.ti.com/tool/EVMK2H

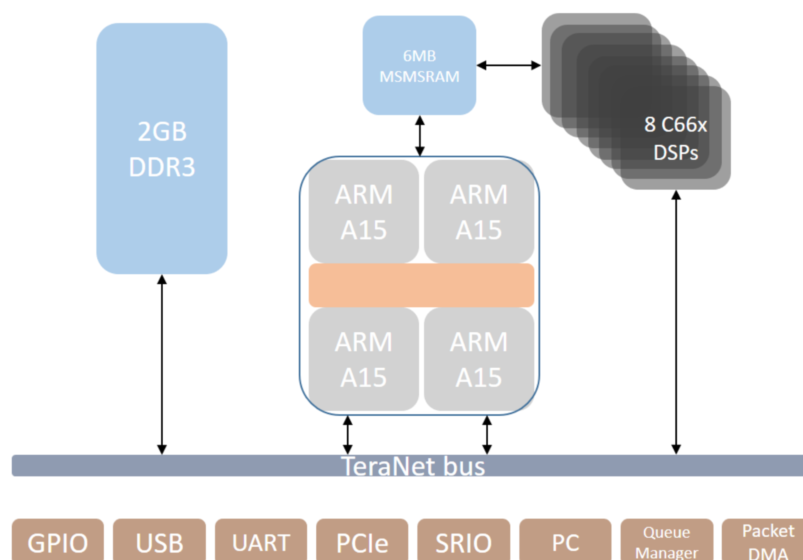

Figure 9 : Schéma des composants principaux de la Keystone 2

3.2.2.2 Architecture logicielle

Contrairement à la carte Parallella, qui n'offre que des APIs aux fonctionnalités limitées, TI fournit un ensemble complet d'outils et de SDKs pour développer sur leurs produits.

En particulier, TI met à disposition un micro-noyau capable de s'exécuter sur les DSPs et offrant toutes les fonctionnalités basiques d'un noyau, telles que la gestion de tâches, de la mémoire, de la concurrence etc…

Ce micro-noyau, portant le nom de *SYS/BIOS*[13], se présente comme un ensemble de modules C indépendants entre eux et indépendants de l'architecture sur laquelle ils sont déployés, que l'utilisateur peut choisir ou non d'inclure. L'architecture employée est basée sur le système des *Real-Time System Components*[14] d'Eclipse. L'idée est de donner à l'utilisateur le choix des modules utilisés et de leurs paramètres d'utilisation afin de répondre au mieux à ses besoins, tout en limitant au maximum l'impact sur la mémoire, qui est généralement une ressource très précieuse sur les plates-formes embarquées hétérogènes.

---

[13] http://processors.wiki.ti.com/index.php/Category:SYSBIOS

[14] http://www.eclipse.org/rtsc/

À ce titre, l'utilisation de ces composants nécessite leur instanciation par le biais de fichiers de configuration XML. Ces fichiers sont parsés avant la compilation de l'application afin de générer le code source correspondant aux paramètres spécifiés et à l'architecture ciblée. Une fois cette étape réalisée, le code produit peut être lié au code de l'utilisateur pour construire le produit final.

Puisque *SYS/BIOS* peut être utilisé avec tous les coprocesseurs TI, l'instrumenter avec des points de trace, à la manière du noyau Linux, serait un pas en avant vers l'unification du traçage de ces processeurs.

3.3 Architecture de la solution

Puisque l'on souhaite tracer à la fois des processeurs classiques ARM et des coprocesseurs, deux traceurs distincts devront être utilisés : LTTng et barectf. L'utilisation de ce dernier nécessite la création de modules C capables de traiter les paquets générés. L'avantage d'utiliser ces deux traceurs plutôt que d'autres est qu'ils génèrent tout deux nativement des traces en format CTF, rendant leur corrélation plus aisée.

Un esclave doit également être en mesure de transmettre les paquets qu'il génère à son maître, nécessitant ainsi la mise en place de canaux de communications entre les deux. Comme nous l'avons vu auparavant, la seule solution générique consiste à utiliser une zone mémoire partagée.

La synchronisation des traces peut être assurée à l'aide d'un **mécanisme par interruptions** : le maître envoie périodiquement des interruptions à ses esclaves (générant ainsi la première paire d'évènements, correspondant à un point d'un enveloppe) auxquelles ceux-ci répondent le plus rapidement possible (générant ainsi la seconde paire d'évènements, correspondant à un point de l'autre enveloppe). Ce processus peut ainsi être résumé par la figure suivante :

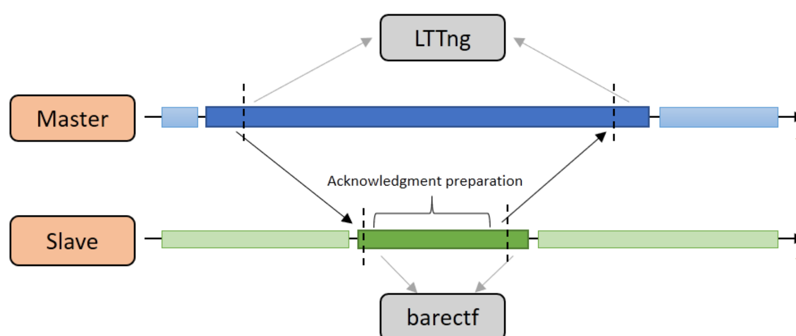

Figure 10 : Schématisation du processus de génération de paires d'évènements

Une description complète de la solution proposée est donnée dans le chapitre suivant, qui correspond à l'article de recherche écrit sur le sujet, soumis à l'*EURASIP Jounal on Embedded Systems*[15] et écrit par l'auteur de ce mémoire, sous la supervision de Michel Dagenais.

[15] http://jes.eurasipjournals.springeropen.com/

CHAPITRE 4 ARTICLE 1: TRACING HETEROGENEOUS EMBEDDED SYSTEMS

Thomas Bertauld (main author), Michel Dagenais

Submitted to l'*EURASIP Jounal on Embedded Systems.*

4.1 Abstract

This paper discusses the possibilities of tracing heterogeneous embedded systems. Its goal is to show how efficient tracing can be achieved without having common system tools, such as the Linux Trace Toolkit (*LTTng*), at hand. We propose a generic solution to trace embedded heterogeneous systems and overcome the challenges brought by their peculiar architectures (little available memory, bare-metal CPUs or exotic components for instance). We also discuss a way of correlating traces among different kinds of processors thanks to *traces synchronization*. Even though this work should be reproducible on a large variety of devices, we experimented with two devices, the Adapteva Parallella and the TI Keystone 2, focusing on the TI Keystone 2 System-On-Chip for the performance evaluation.

4.2 Introduction

Heterogeneous embedded systems combine the peculiarities of classical embedded systems (for instance, little available memory or exotic architectures) with the complexity of having numerous processors of different types (with different architectures) on the same board. Usually, some processors are referred to as *masters* because they are the "main" cores, typically running a High-Level OS (HLOS) like Linux, and giving orders to the *slaves* which are in charge of some specialized tasks. The slave processors can be for instance DSPs tasked with signal processing algorithms. Having a complete understanding of the interactions between all the CPUs and finding the causes of bottlenecks, abnormal latencies or simple bugs can quickly become difficult without the proper tools. Even proper tools can sometimes encounter limitations on such devices: running the GDB debugger on a thousand cores, for instance, is not really ideal. As such, heterogeneous systems vendors often provide their own, more suited, diagnosis tools for a specific device.

Tracing is an elegant and efficient way of obtaining information on a system without disturbing it too much. It requires the instrumentation of the application being traced (*i.e* the addition of *tracepoints*) to output timestamp-matched events and give insights on the execution of specific parts of a system. A set of such events is called a *trace*. Because of its granularity (tracing can be done for instance inside every system call of the Linux kernel), traces can be huge and are not well-suited for every situation. However, tracing allows a better information-gain/performance-loss ratio than common logging methods and requires much less time than classical step-by-step debugging.

In this paper, we present a generic way of tracing heterogeneous embedded systems in an attempt to show how efficient it is to solve the problems stated above and how it can lead to a standardization of the means to analyze those systems. For that, we will see how tracing can easily be brought to virtually any platform and how it is possible to correlate the traces obtained from different kinds of processors. This is particularly important as those devices are in widespread use since they offer small, multicore, energy-efficient yet powerful, cheap environments. Although the work presented in this paper should be easily reproducible on any device, it is still in a prototyping state and its main purpose is to demonstrate what can already be done and what could be done beyond that.

This paper is structured as follow: section II discusses the use of DSPs, and covers related work on generic tracing, bare-metal CPUs tracing and traces synchronization. Section III describes the architectures of two devices used in this work: Adapteva's Parallella board featuring 16 low-power computing cores and TI's Keystone 2 SoC featuring 8 TI's DSPs. Section IV introduces *barectf*, a tool used to generate traces on bare-metal systems. Section V then discusses in details the challenges and ways of correlating heterogeneous traces. Finally, section VI exposes and discusses some results through a set of benchmarks and a complete use-case on the Keystone 2, before concluding on the state of tracing heterogeneous embedded systems.

4.3 Literature review and related works

Digital Signal Processors (DSPs) are specialized CPUs with dedicated mechanisms allowing faster mathematical operations (often through built-in hardware). As their name suggests, they are mainly used for signal processing purposes such as telecommunications or video processing. Chien-Chih

Liu and Hsueh-Ming Hang [2] showed how TI DSPs can be used to implement an efficient JPEG encoder. Wallace et al. [8] discussed the benefits of multiple-input multiple-output (MIMO) wireless systems using TI DSPs. Finally, Huang and al. [49] demonstrated how DSPs can be used to perform control algorithms, allowing better motion accuracy in mechatronic systems. These are only a few examples of what can be done with DPSs, thus demonstrating why having proper analysis tools is mandatory.

Because this is specialized hardware dedicated to certain tasks (FFT, complex computations and so on), DSPs don't offer the same characteristics as common CPUs, such as the mainstream x86 architecture. In particular, they might have lesser working frequencies or limited instructions sets (see [62] and [30]). The same generally applies to any co-processor used as a computing aid by a master CPU. However, because the SoCs they are implemented on are part of larger complex systems, they often have direct access to different parts of the hardware like faster on-chip memory or external peripherals. Although Durrant et al. [54] showed that, under certain circumstances (such as the presence of some required instructions), it would be possible to run a Linux kernel on such cores, it is generally not advised to do so as it would probably diminish their overall performance. For this reason, they are most likely used "as-is", *i.e* as bare-metal CPUs.

Tracing Linux-based systems has been proven many times to be as easy as it is efficient. Common tools such as LTTng (*Linux Tracing Toolkit next generation* [39]) are widely used to trace both the Linux kernel and user-space applications [15,59]. Having the ability to trace both domains at once allows a better understanding of a system as a whole. It might, for instance, give a lot more explanations on abnormal behaviors: the roots of an abnormally high latency cannot always directly be found in user-space and might require tracing system calls. Desnoyers and Dagenais [22] also showed that porting LTTng to different architectures can easily be achieved as long as some requirements, such as the presence of fine-grained timers able to generate different timestamps for events potentially very close to each others (some nanoseconds apart), are met.

Tracing bare-metal systems is a bit more tedious as, by definition, there is no access to any of the usual Linux system tools. However, Proulx created a python-based tool called *barectf* (see [28]) able to generate C99 code implementing different tracepoints, which can then be linked with the user's application to generate native CTF (Common Trace Format) traces. This is particularly interesting given that CTF is also the default output format for LTTng traces and aims to

standardize trace output across different systems. Proulx particularly showed in [11] and [63] how barectf could be used to trace some very constrained co-processors inside the Epiphany chip. Because barectf is a platform-independent tool, the user is required to write the client-side code in charge of relaying the generated packets to a consumer (generally a master CPU running an HLOS) for every new platform. As such, we will use Proulx's work on the Parallella platform and extend it for the Keystone 2 board.

Figure 11 sums up a basic setup enabling tracing of both a master CPU and its associated slave.

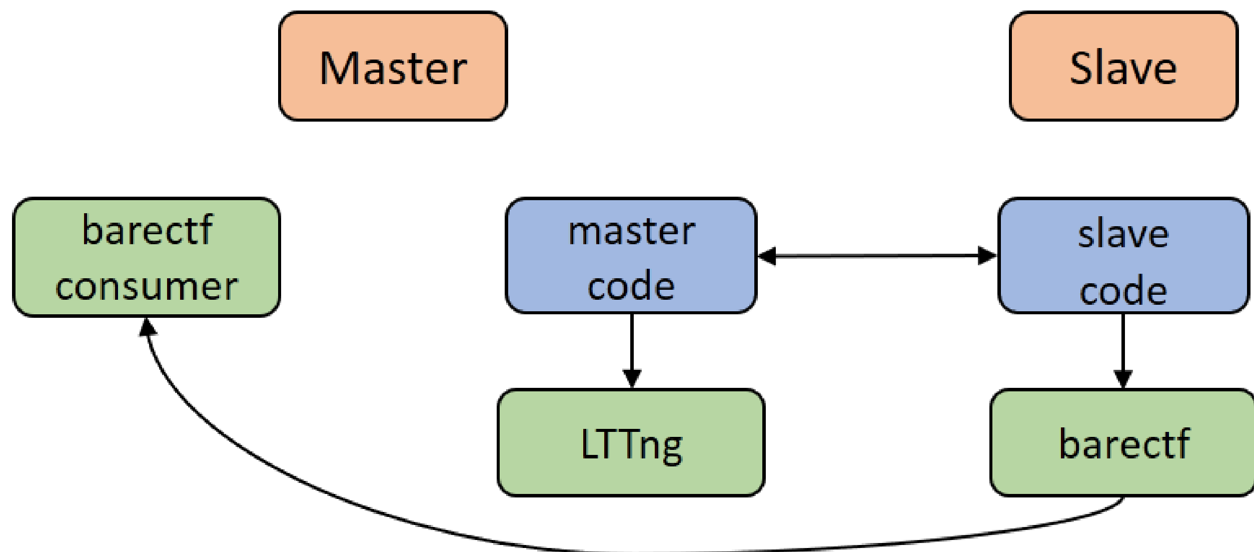

Figure 11 : Basic setup used to trace embedded heterogeneous systems

This previous work shows how it is possible to virtually trace any existing system. Nevertheless, even if one generates traces in the same format (CTF), on different CPUs with possibly different architectures (heterogeneous systems), one would still need to find a way to correlate said traces. In fact, traces obtained from different machines will most likely not share the same timestamps origin, have different working frequencies and frequency scaling policies. This is a common problem when it comes to tracing distributed systems in which, even if all the machines are running Linux and have the same architecture, they all have different clocks. As such, directly comparing traces obtained on all devices wouldn't make much sense.

In this context, Jabbarifar [44] showed that traces correlation can be achieved through synchronization, merging two or more traces with different timestamps origin into a single one with the same origin. Although this work discussed "live" synchronization (*i.e* synchronization

done while the traces are being recorded), it is generally preferred to run a post-analysis on existing traces with a trace analysis and viewing software such as *TraceCompass*. This way, we keep tracing from interfering too much with the user's application and limit its performance overhead to the minimum. Poirier et al. [3] in particular showed how generating pairs of matching events between traces allows a post-tracing synchronization process. Since this method is the one we chose to use, it will be further discussed in section V.

As far as we are aware, no other published work was directed at tracing embedded heterogeneous systems in a structured way, with traces taken at different levels being synchronized and analyzed in a suitable trace viewing tool. By using the tools and concepts previously presented, we intend to show how such tracing can be achieved and how to address its main challenges.

4.4 Studied devices

In this section we will present the two primary devices used to develop and test our generic tracing methods. Although some limitations or remarks stated below may depend on the targeted device, they often are consistent with generic heterogeneous embedded devices. We believe those two devices to be relevant as the first one (Adapteva's Parallella) is a FPGA-based custom board that could easily be used as a prototyping device, and the second one (TI's Keystone 2) is widely used in industry for production purposes. As such, investigating those devices should bring enough content to demonstrate how virtually any heterogeneous embedded platform can be efficiently traced.

4.4.1 Adapteva’s Parallella

The Parallella board [46] was the first platform we used to better understand the constraints of heterogeneous embedded systems. It provides between 16 and 64 co-processors and soon up to a thousand cores, for a relatively low price. As such, the board is investigated by several companies as a low-cost power-efficient embedded system for different dedicated tasks, which is why one might be interested in having traces on it.

A high-level view of its main components is displayed in Figure 12.

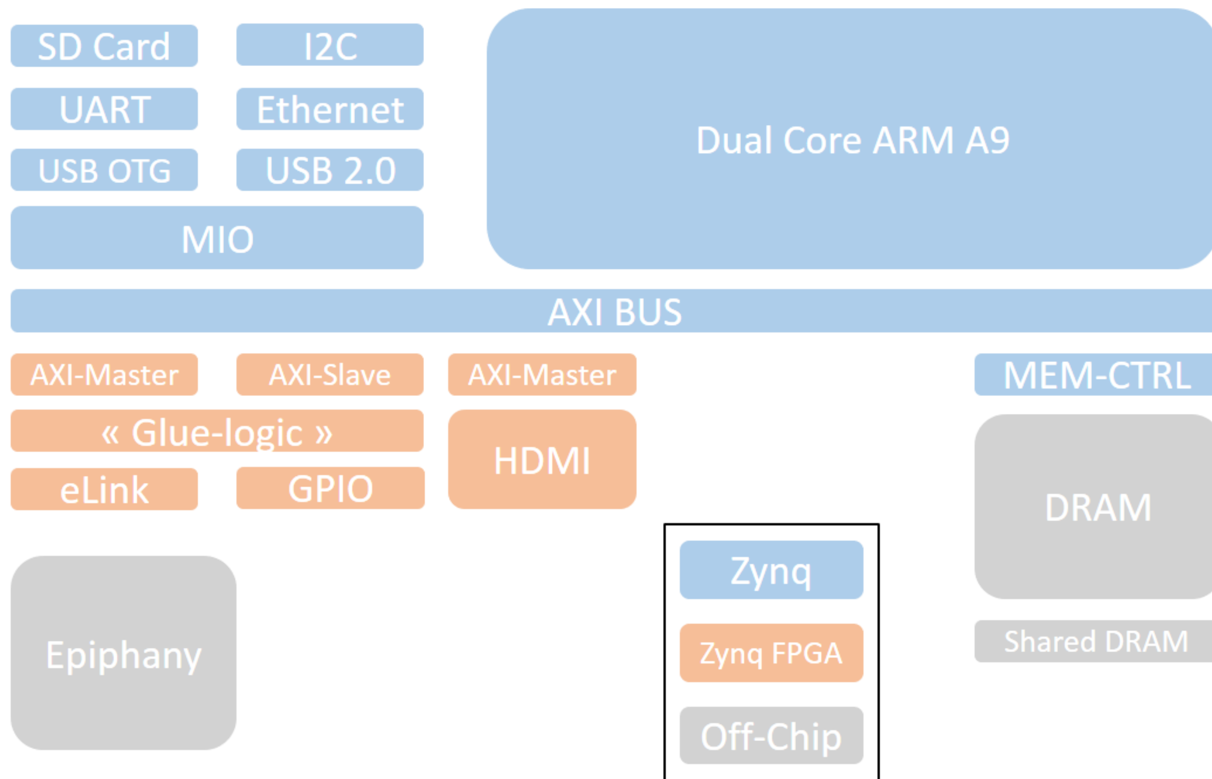

Figure 12 : High-level view of Parallella's main components.

4.4.1.1 Hardware specifications

The core of the Parallella board is composed of a Zynq 70xx SoC containing a dual-core ARM Cortex A9 and a Artix-7 FPGA, an *Epiphany* chip and 1GB DDR memory. The Epiphany chip is the one containing from 16 to (soon) a thousand cores, depending on the model. The 16 cores model was used in this study.

As seen in Figure 13, the Epiphany chip is modeled as a network where each node (called a *router*) contains a RISC CPU, a DMA engine and a Network Interface. The nodes are linked to each other and to the outside of the chip through three different channels. The *rMesh* channel is used for read requests, the *cMesh* is used for on-chip writing and the last one, the *xMesh* performs off-chip writings.

The board itself displays three layers of memory for each core, listed from fastest to slowest: local memory, on-chip memory, off-chip memory.

Each CPU inside a node of the Epiphany network (called *eCore*) has the following characteristics:

- 32 bits little-endian single-core CPU with 35 instructions cadenced at 1Ghz
- C/C++ or OpenCL programmable
- 32 bits IEE754 floating points
- 512kB on-chip distributed shared memory (32kB local memory for each core)

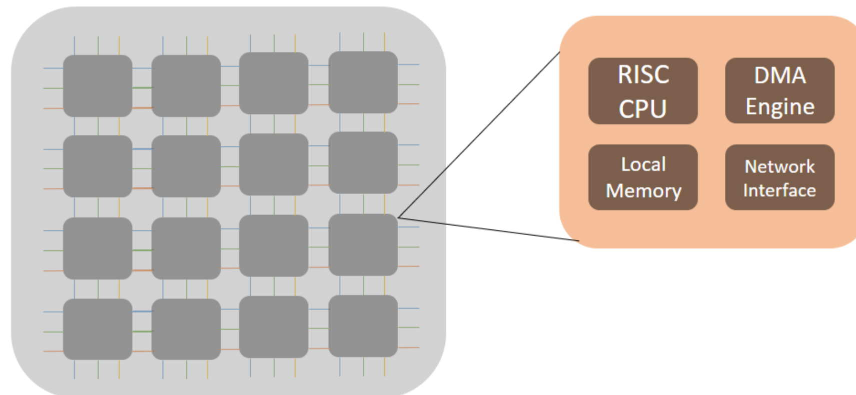

Figure 13 : Parallella's Epiphany network design.

4.4.1.2 Software design

Since the *master* CPUs are ARM CPUs, they can run Linux and act as a command center for the Epiphany chip. Interactions between the two are made through a set of C APIs provided by Adapteva. The Linux version distributed with the board also provides an Epiphany driver which only role is to allocate a chunk of shared memory in the 1GB available DDR. Due to their characteristics, the eCores cannot run a common HLOS such as Linux and are best used as bare-metal CPUs.

The provided APIs grant full access to the user on the Epiphany chip through ARM-side code, allowing him to send data to a core, modify its registers or even interrupt it. However, the reverse situation is different. The other way around, no eCore can directly interact with the ARM-side (the master). For instance, the master can run some task on an eCore (load the code, load the data and run the code) but the eCore has no way to warn the master that its task is done. Every interaction in this direction must use the 32MB shared memory space or the eCore's own memory space (through flags placed in this area for instance).

4.4.1.3 State of tracing

As we can see in [63], tracing the Parallella board is possible on both the Epiphany chip (thanks to barectf) and the common ARM Cortex A9. However, due to certain limitations listed below, the state of tracing on this platform is (as of today) not optimal and could be slightly improved.

Facing those limitations gave us a great insight onto what could be the challenges of tracing heterogeneous embedded systems. Trace packets communication, for instance, is an important part of the process, as the packets generated on the slave have to somehow be sent to the master. Having no way of direct communication from the eCores to the ARM made us experiment with on-chip shared memory, thus allowing us to not rely on any API and keep things at the bare-metal level.

The following remarks sum up our conclusions regarding the possibilities of tracing this specific Parallella model.

Hardware limitations:

- Lack of global memory: little place to stock the traces, little place to exchange data between master and slave.
- Lack of internal memory: each core has access to very little memory (only 32KB), stressing the need to have an instrumented code as short as possible, given that to achieve optimal performance, both the application code and the data used should be in the internal local memory.
- No memory order: as Proulx discussed in [63], the 3-channels communication network inside the Epiphany chip doesn't offer any guarantee as far as memory ordering is concerned. This can lead to surprising situations where one would normally not expect memory transactions to appear that way.
- No interrupt queue: only one interruption can reach an eCore. If one is already waiting to be treated and another one reaches the eCore, the other is simply discarded. Nested interruptions are possible although some code twisting is required in the provided APIs.

Software limitations:

- The provided APIs lack some basic communication mechanisms from an eCore to a master.

- Classical tools may not work as intended, or it can be hard to make them work on such particular systems. For instance, gdb was not working well on the Epiphany chip and would have required to connect 16 gdb clients. One can imagine what a trouble it would be for a thousand or even 64 cores, hence the need for efficient tracing solutions.
- We also found that the custom gcc version used to compile code on the eCores places some *gid/gie* instructions (respectively globally disabling or enabling interruptions) around certain operations, to protect them from external interrupt sources, even though it was not always necessary.

All those previous remarks make tracing the Parallella board all the more interesting. Overcoming the challenges brought by the lack of memory, or the limitations regarding the interruption mechanism on which the synchronization procedure heavily relies (see section V), is a great demonstration of what can be done on very constrained devices. We believe that this platform should evolve for the better with its future models and could bring tracing on those devices to the next level.

With a first working prototype and a better understanding of global challenges regarding heterogeneous embedded systems, we subsequently focused on the TI Keystone 2 platform which offers popular DSPs and an interesting setup to trace complex applications.

4.4.2 TI's Keystone 2

To test the Keystone 2 SoC, TI's EVMK2H evaluation board was used, featuring a 66AK2H12 SoC containing 4 ARM Cortex A15 and 8 TI's C66 CorePacs DSPs (see figure 14) [42]. This board also provides 2GB DDR memory and a faster 6MB shared memory. This SoC is commonly used in industry, which makes it an interesting platform to trace. From the user point of view, the major feature is the presence of 8 DSPs, meaning that 8 computing devices specially designed for signal processing needs can be simultaneously used to perform complex calculations.

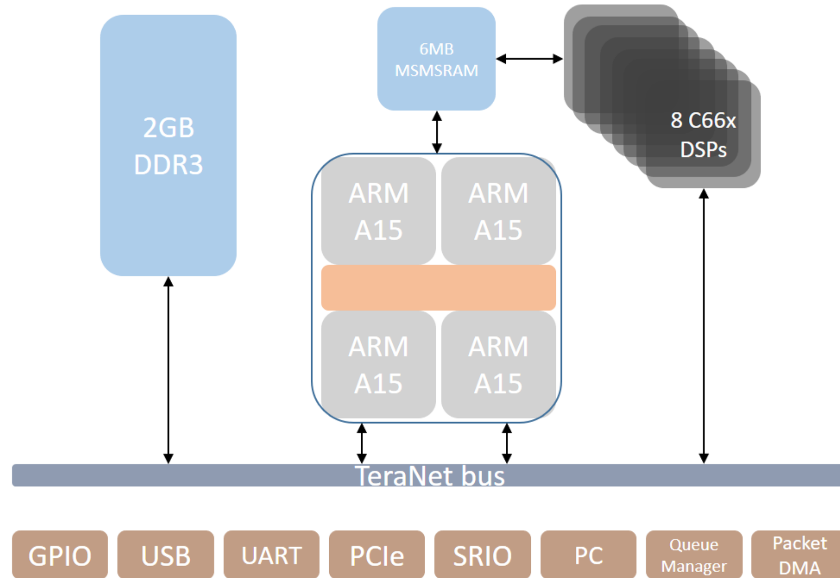

Figure 14 : Keystone 2 SoC components high-level view.

Just like for the Parallella board, the 4 ARM Cortex A15 are able to run Linux and will thus be called "master" CPUs, in charge of giving orders to the DSPs (slaves). Below are more details about the specifications of the DSPs:

- 8 TMS320C66x DSPs (C66x CorePacs) cadenced at 1.2GHz
- Multi-layer cache system on each CorePac:
    - 32KB L1P
    - 32KB L1D
    - 1MB local L2
- Up to 38.4 GMacs/Core for fixed point
- Up to 19.2 GFlops/Core for Floating point

4.4.2.1 Software design

Texas Instrument provides several SDKs and tools to develop on their products. Because their goal is to give standardized APIs which would work on all their platforms, the software environment is a bit complex but handles a lot of useful features. As we will later see, it provides a "micro-kernel" on the DSPs, allowing powerful applications to be easily developed. As usual, the ARM CPUs can run Linux, just like it was the case with the Parallella.

Since the DSP applications are written in C, the compilation can be done with GCC, although TI provides its own toolchain, taking advantage of the DSPs particularities. This toolchain can be executed by hand but is best managed by TI's CCStudio IDE (based on Eclipse). In fact, every step in the development process on the DSPs can be dealt with by CCStudio: from compilation to analysis of the application.

TI's SYS/BIOS is the main tool one will use to develop on the DSPs. Presented as a mini real-time kernel, it offers the basic functionnalities one might need from a regular operating system. SYS/BIOS can in fact take care of tasks management (although each DSP is mono-core so no task can run in parallel on the same DSP), hardware and software interrupts, memory management and much more. It is based on Eclipse's *Real-Time Software Components* \[53] (RTSC) and ships as a set of C modules (or components).

Because this mini-kernel aims at working with every relevant TI product, each component is hardware-agnostic and needs to be configured before use. The configuration phase is managed by the XDC tools: one has to write a configuration script describing which module is to be used and with what parameters. Then, the script is computed by XDC tools and platform-specific code is generated from the generic one according to provided configuration. The complete steps needed to create an application on a DSP using the SYS/BIOS environment are as follows:

1. Create a configuration script (.cfg) describing the modules you want to use, and how do you plan to use them.
2. Create your own application using the provided generic APIs.
3. Use the XDC tools as a pre-compilation phase to instantiate the modules (defining missing *defines* for instance) and create the specific code needed by your application to use said modules (and only those).
4. At compilation-time, link your application with the provided code to create the final executable.

Of course, the provided IDE can take care of all those steps by itself.

Other APIs shipped with the SDKs include libraries taking into account the strengths of the DSP such as image processing.

The DSPs can be used as bare-metal devices *i.e* the use of SYS/BIOS C modules is not mandatory) if one has very specific constraints in terms of either memory space or "raw" performance. Otherwise, the overhead induced by SYS/BIOS is worth bearing because of all the well-optimized methods it provides to handle basic core management.

Since it is presented as a real-time operating system, our main interest in this product is to see how we could instrument it like we are able to instrument the Linux kernel to obtain traces. For instance, tracing context switches between tasks would provide a lot of information regarding the global state of the system. Being able to trace the SYS/BIOS kernel along with user-specific applications would open the way to a number of possibilities such as critical path analysis [71].

4.4.2.2 State of tracing

We encountered on this platform the same problems we faced on the Parallella board: there are mechanisms to analyze what is happening on the system provided by TI but there is no direct method to obtain traces in the same way as with conventional tracers. Logging (provided through *printf-like* functions) can be a useful option for fast debugging but it is obviously not as optimized as tracing. CCStudio also gives a view in which it is possible to visualize a lot of information about the system's components at a given time (one might need to pause the system). For instance, one can see the status of a task (running or not) or the content of a particular register.

However, neither of these solutions are sufficient to fulfill our main objective, gaining as more information as possible on specific system parts without breaking the application flow and with low overhead. Since barectf has not yet been ported to this platform, section IV will explain how such porting can be achieved and display the LTTng/barectf combo abilities to trace a complete heterogeneous system.

4.5 Tracing with barectf

In this section, we will briefly present barectf and describe the work required to port it to a new platform. The Common Trace Format (CTF) is an efficient representation for tracing data and associated metadata (description of event types and payload). It is open and well documented, and supported by a large number of trace visualization tools. Bit-manipulation functions are needed to generate events in this format for it to be as efficient as possible memory-wise. For instance, it is perfectly acceptable to declare and trace a 3 bits integer. To insure that a parser can read through

such traces, a set of metadata, describing the content of each event, is required. Barectf takes care of creating the trace metadata and all the bit-manipulation functions.

Barectf is implemented as a python-based tool helping users getting CTF traces by generating a set of C tracepoints. To do so, the user provides a configuration file written in YAML describing what are the tracepoints needed: each tracepoint is given a name and a payload that can basically be anything from a single integer to a structure including strings and floats.

Once the configuration file is written, barectf will generate the corresponding traces metadata and its associated tracepoints implementations.

The following figure sums up a basic barectf setup:

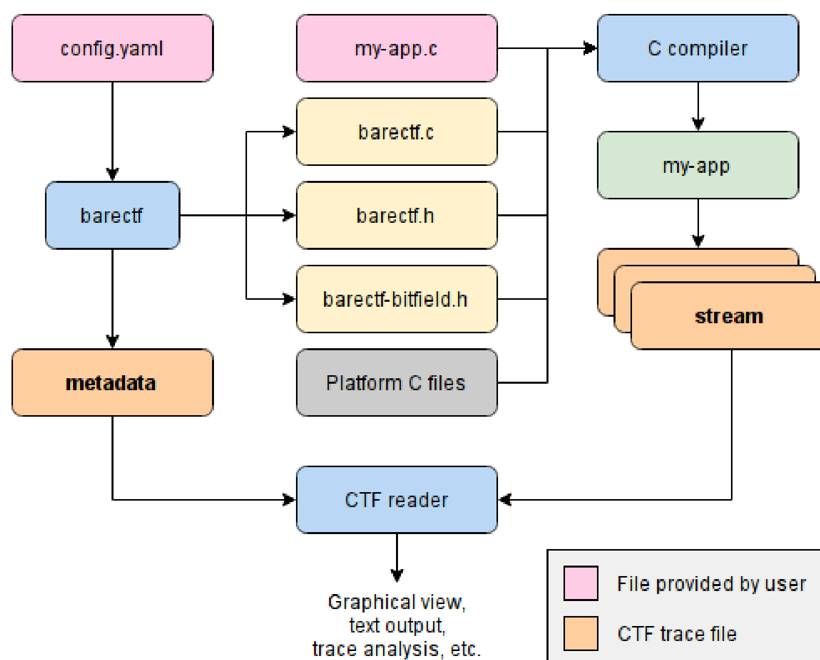

Figure 15 : High-level view of barectf's workflow

extracted with permission from [11].

In order to use barectf with any device, one needs to implement some client-side code managing the packets generated by barectf. Packets are sets of events, initially recorded in a buffer. Packets have a limited length and thus can only store a finite set of events and need to be handled when full before recording a new event. Therefore, for a new system, one has to implement what is called a "barectf platform" because the barectf instrumentation only manages the life-cycle of a packet (opening it, filling it with events and closing it) on the slave. Because it aims at being as generic as

possible, it is hardware-agnostic and thus doesn't provide any API to store the packet on the device or send it to the master. It also doesn't provide any locking mechanism to protect the same packet from being overwritten by two different calls to tracepoints (which is possible if some task management is present on the device or through interrupts while already being in the tracepoint code).

As such, one needs to write code to properly initialize all buffers and structures used to store packets, to finalize them when needed and, more importantly, to handle a full packet. In this case, one has to decide what to do with the packet: depending on the application needs and the state of the system, one might want to write the packet to memory elsewhere, send it to the master or even drop it. It is all up to the user implementation.

For instance, we decided to use TI's *MessageQ* API to handle a full packet from the slave and send it to the host on the Keystone 2. This API allows the host to sleep on a semaphore's lock while waiting for a packet to be received. we also experimented with a generic locking mechanism using a *static volatile char* being set and TI's semaphores to achieve mutual exclusion between tracepoints.

Here is the basic API one would need to implement to have barectf working on a new device:

```
/* Instantiates the tracing context  and initializes every structure needed to take care of the
  * recorded events and stored packets.*/
int8_t barectf_init(void);

/* Returns the tracing context.*/
barectf_ctx_t *barectf_get_ctx(void);

/* Specifies the way to access 64bits timestamps on the targeted device.*/
uint64_t barectf_get_clock(void *ctx);

/* Initializes (if needed) the counter used to get timestamps.*/
void barectf_init_clock(void *ctx);

/* Opens a new packet containing recorded events.*/
void barectf_open_packet(void *ctx);

/* Takes care of a full packet. The packet can be sent to the host, put in another memory
  * location, discarded... */
void barectf_close_packet(void *ctx);

/* Finalizes the tracing session.*/
int8_t barectf_close(void);
```

4.6 Correlating heterogeneous traces

We have seen how tracing different parts of an heterogeneous environment can be achieved thanks to tools like LTTng and barectf. Our main goal is thereafter to take the obtained traces and synchronize them such that they all rely on the same timestamp origin. Synchronized traces can provide an actual view of the global system workflow. As previously stated, doing so is mandatory since the 8 DSPs and the master can have different time origins. In fact, directly analyzing all the

traces without modifying the raw timestamps would absolutely mean nothing and could at worst bring misunderstanding about what is really going on: one could for instance see events in reverse order or even coming from years apart! This is why synchronizing the traces taken on DSPs and ARM is a required task to preserve timestamps and cause/effect coherency.

4.6.1 Generating pairs of matching events

As explained earlier, synchronization can be achieved either "live" or as a post-tracing process. In order to minimize the disturbance of the monitored system, we chose to use the second option. For this to work, one need to generate *pairs of matching events* during the tracing session. Those events can be seen as a "handshake" between a master, which will be used as the synchronization origin, and a slave, for which the trace timestamps will be adjusted to match those of the master. The process is as follows:

1. The master generates its first matching event of sequence n and proceeds to ask for its counterpart on the slave to be generated.
2. Once the request is handled by the slave (which should happen as fast as possible for more accurate results), the second event (seen as an *ACK*) of the first pair is generated on the slave (with sequence number n).
3. The slave then generates the first event of the second pair of sequence n+1 and notifies the master.
4. Once the master handles the notification, it generates the second event of the second pair with sequence n+1.

Figure 16 gives a graphical representation of this process:

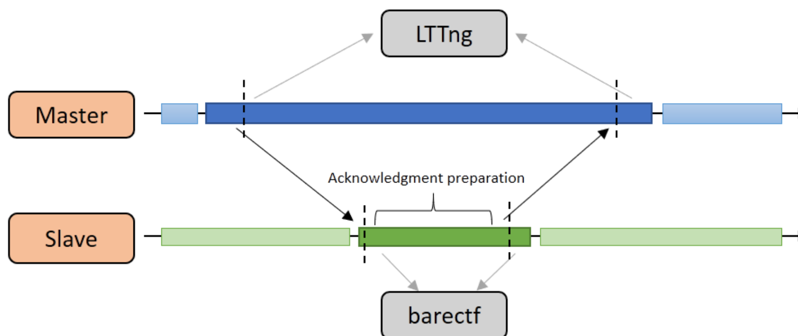

Figure 16 : Generating pairs of matching events.

Those events will later be treated by the post-tracing process in order to compute the right timestamps modification of the slave's traces according to the master's traces. Note that each pair needs to be *uniquely* identified during the post-tracing process, thus explaining the need of unique sequences id.

In the next subsection, we propose a generic and efficient way to make the right interactions between a master and slave during the matching events' generation process.

4.6.2 Workflow and synchronization

Since the proposed method should be as generic as possible, we assume that the slave's CPU is mono-core and thus cannot handle simultaneous threads. We also make the reasonable assumption that the slave's CPU can be interrupted. On the master side, we only assume that Linux is running, enabling multiple processes at a time. Although this is not mandatory for what follows, we also assume for the sake of the argument that the slave is running a very specific task that needs to finish fast, thus implying that it should not be disturbed too much. Finally, we suppose that there is at least one way to communicate data from a slave to a master (the other way is mandatory if we want the master to send tasks to the slaves). This can simply be a shared memory space or a more complex message passing mechanism.

As seen in the previous section, to generate pairs of matching events across different traces, one needs the master to send a request to the slave. This can be done by having the master triggering an interrupt on the slave, thus forcing its workflow to be suspended in order to process the incoming interruption. Said treatment then proceeds to send an acknowledgment message to the master. If no message passing mechanism is present on the target, then a simple flag indicating the sequence number and the associated core will do. After generating its second event, the slave will then return to its previous task.

On the master's side, once the interrupt is sent, depending on the communication's mechanism used between the two, one needs to either wait for the ACK message to come or poll into a specific shared memory location until a flag is set by the corresponding slave.

To generate a set of matching events, a background task (referred as *synchronization daemon*) can run on the master to periodically generate the interrupts and corresponding events. The interrupt

frequency needs to be adjusted so that the synchronization is accurate enough (enough matching events can be used) but does not disturb the slave's task more than necessary.

The global scheme of this process can be summarized by Figure 17, where:

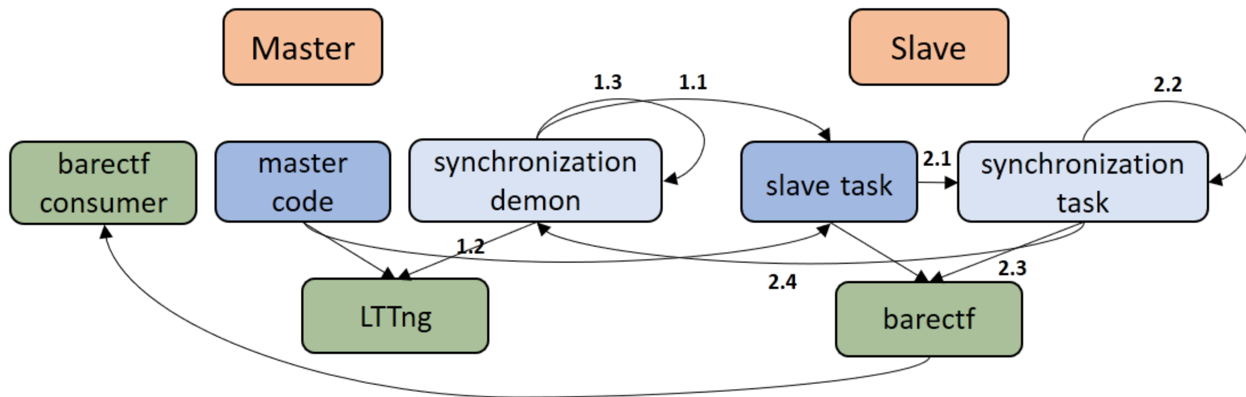

Figure 17 : High-level view of the synchronization process.

1. The synchronization daemon (master's side):
    1. Interrupts the slave workflow to request synchronization
    2. Generates the first event of the first pair
    3. Waits for the ACK to come from the slave
2. On the slave:
    1. The interruption is handled and thus begins the synchronization task (can be a real task or a function, in which case the next step doesn't apply)
    2. The synchronization task is "awoken" from its waiting state since an interrupt arrived
    3. The second event of the first pair is then generated
    4. The slave ends its synchronization task by sending the ACK to the master and consecutively generating the first event of the second pair
3. The synchronization daemon (master's side):
    1. Receives the ACK and thus leaves the 1.3 state
    2. Generates the last matching event (same transition as 1.2)

3. Stays on standby until the next periodic synchronization

The way we implemented this process on the Keystone 2 uses the IPC generation register to trigger interruptions on the slave from the master (the registers are accessible once the correct memory page is mapped on the master's application) and TI's *MessageQ* API to send the ACKs from the slave to the host, thus eliminating the need to constantly poll a shared memory location.

4.6.3 Post-analysis treatment

Once the previous process is done, the only thing left to do is to run a post-analysis on the traces which will handle the timestamps transformation.

The method we chose to use for its efficiency is based on the *convex hull* algorithm, as discussed in [3] and [31]. It has been widely used to synchronize kernel traces using network packets inside distributed systems. The pairs of matching events generated are the generic equivalent of those packets.

Let's consider traces A and B, coming from two different devices. Knowing that those traces are somehow linked we want to achieve a correct cause/effect order between them. In this case, where a master sends commands to a slave, the trace computed on the slave (B) should be merged with the one obtained on the master (A). Poirier et al. [3] showed how a linear transformation of B's timestamps ($t_{B'} = a.t_B + b$) would produce such a merged trace. The slope can be seen as a frequency mitigation factor and the offset's role would be to synchronize the first event of each trace together.

The convex hull algorithm's goal is to find the suitable linear transformation. For that, the pairs of events are displayed on a 2D graph where each axis represents the timestamps of the events of one device. The upper-half (respectively lower-half) of the hull is used to determine the conversion function with the maximum (respectively minimum) slope. Because any function inside the hull could be used to synchronize the traces, those two functions can approximate the boundaries of the hull as no conversion function should have a higher (respectively lower) slope.

Duda et al. [31] suggest to take the bisector of the angle formed by those two lines as the conversion line, thus giving us the linear transformation to apply. For this to work, at least two points should be found in each part of the hull, and the more points you have the better the approximation gets. However, in the case of tracing, the original system shouldn't be disturbed too much, which is why

a compromise should be reached to have enough satisfying points without impacting the system much.

A tool such as TraceCompass can easily take a set of traces and compute the hull and the linear transformations to apply to each, based on a "master" trace.

4.7 Results

4.7.1 Benchmarks – Tracing overhead

Since tracing should be as non-intrusive as possible, we executed some benchmarks to evaluate the performance of our barectf implementation on the TI Keystone 2. The barectf platform built relies on the TI MessageQ API to send the packets from a slave to the master. This way, we take advantage of the built-in waiting queues used for message passing and don't have to worry about memory overlapping when writing packets. However, this API induces some latency, which is why benchmarking is even more important with this solution.

The configuration of the platform running the benchmarks is as follows:

- 8 C66 CorePac DSPs running at 1.2Ghz.
- barectf platform using MessageQ API.
- barectf platfrom configured to allow at most 256 packets, of 256 bytes each, at a time.
- All compilation optimizations turned down.

The results (given in cycles) are computed in table 1. Four workloads were executed, each of them based on computing sha-256 hash on a set of strings. This set, composed of the 5040 permutations of the word "barectf", is recursively created as the benchmark is being executed.

- (a): Computes the sha-256 hash of each permutation and produces a tracepoint for each result (5040). Note that the tracepoint is composed of a 32-bits integer and not of the 256-bits result, because directly tracing the result would require either 4 (respectively 8) 64-bits integer (respectively 32-bits integer) tracepoints or a string to be sent, thus artificially increasing the overhead.

- (b): Computes the sha-256 hash and produces a tracepoint for every set of 5 permutations (1008).
- (c): Computes the sha-256 hash and produces a tracepoint for every set of 10 permutations (504).
- (d): Computes the sha-256 hash and produces a tracepoint for every set of 100 permutations (50).

Workload \ Instrumentation	None	Barectf	Overhead	Cost per trace-point
(a)	52400000	68440000	30.61%	3182
(b)	13150000	16370000	24.49%	3194
(c)	12340000	13930000	12.88%	3154
(d)	7912000	8035000	1.56%	2460

Table 1 : Benchmarking results (in cycles) on the TI Keystone 2

As expected, producing 5040 tracepoints during 43ms induces a 30% overhead on the traced application. This overhead is greatly reduced with the number of tracepoints. It could be further reduced by eliminating the use of TI's MessageQ APIs since they perform a lot of checking and wrapping around their message posting methods.

However, the cost per tracepoint doesn't really increase with the load of the system, although it somewhat diminishes when the load is low (as seen with the last workload). This indicates that there is no bottleneck for tracepoints and that even with a demanding workload, one should not expect heavy latency peaks induced by tracing.

The platform built, even though not perfect, has reasonable enough performance to use in a prototyping state and eliminates the need for memory polling and checking for memory overlapping.

4.7.2 Use-case

In order to demonstrate how the described tracing solution would perform on a real-life problem, we chose to instrument an image processing algorithm running on TI's C66x CorePacs (through the Keystone 2).

The algorithm uses Sobel's filter to do edges detection on an image given as an input by the user. It is provided with TI's SDK, thus this use-case is easily reproducible.

Sobel's filter computes an intensity gradient for each pixel in the image, thus detecting brutal changes in lighting and their direction, which might point towards an edge. Even though this method is rudimentary, the mathematical operations performed still benefit from the dedicated image processing APIs available on TI's DSPs.

The global application goes through 3 steps:

1. A program running on the master (ARM) is waiting for the user to interactively provide an image to process.
2. Then, the master parses the image and sends some memory allocation requests to the "leader" DSP (DSP #0).
3. Finally, the image is cut into 8 pieces, each piece being sent to a different DSP, and the DSPs reply back with their share of the image processed.

Communication between master (ARM) and slaves (DSPs) is achieved through TI's *MessageQ* API, which relies, as its name suggests, on messages queues. The master opens a single queue, on which each slave will be connected and every slave can open its own queue, so that the master can give them orders. This API is quite high-level and adds more latency than basic shared-memory communication. However, it is less prone to errors and allows a core to switch tasks when awaiting a message, as it will wait for a semaphore to be unlocked. If one were to wait for a message in shared memory, one would have to poll the memory location until something came up.

Tracing a system must be done with an idea of what one wants to see in the traces. In fact, since tracepoints can be placed anywhere in the application, it can serve a lot of different purposes. For instance, one might want to place tracepoints at certain places to check when the application reaches them. In other words, tracing can be used for debugging purposes. It can also serve monitoring purposes as one could put tracepoints to generate an event every time a result is awaited.

We decided to monitor our system by adding tracepoints at the beginning and the end of every important function call. This way, we can spot the weakest link in our application by measuring the elapsed execution time of each function.

To have more insight into the device, we also instrumented TI's SYS/BIOS APIs. As discussed before, those APIs (presented as a set of C-modules) allow the user to use basic "kernel-like" features, such as tasks, memory management and some inter-process communication mechanisms.

With the help of well-placed hooks (some were already present in the APIs and some were later added), we were able to monitor the main message passing functions (*MessageQ_get* and *MessageQ_put*) along with some task management related events, such as task switching. This way, the generated traces can inform us on the task running on the DSP at all time. Each DSP being single-core, only one task can run at a time.

Note that every added tracepoint is not just a "checkpoint" in the application's workflow but contains useful information inside its payload. For instance, in the case of the message passing API instrumentation, receiving or emitting queues are described, along with the message size and more.

In summary we:

- Added a tracepoint at the beginning and at the end of every "useful" function.
- Thanks to hooks, also added tracepoints inside critical APIs functions such as message passing or tasks management functions.

And the underlying objective is to:

- Monitor the whole system.
- Spot and understand abnormal latency.
- Analyze dependencies between cores.

The instrumentation provided allows us to compute a partial call stack view in the CTF trace viewer *TraceCompass*. The presented screenshots are generated from this software. This view presents traces as seen in Figure 18:

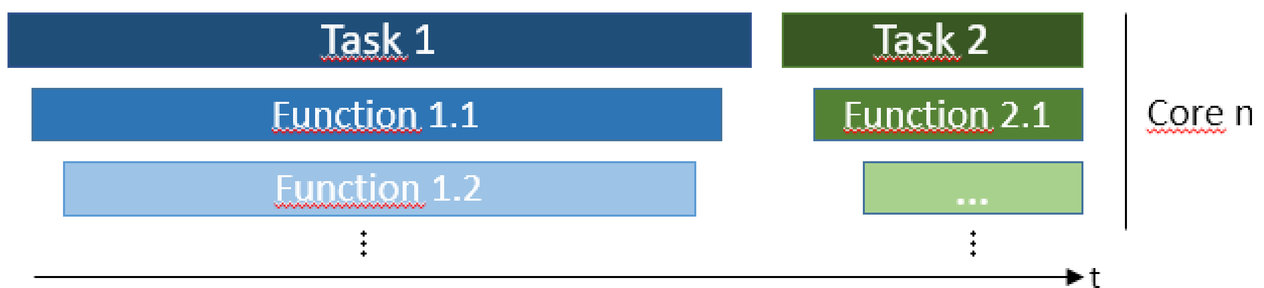

Figure 18 : Description of the callstack view of processes.

Figure 19 shows the global view of our system from the beginning of the application to its end. Note that for display purposes, only the four first processes (the master and the first three slaves) are displayed.

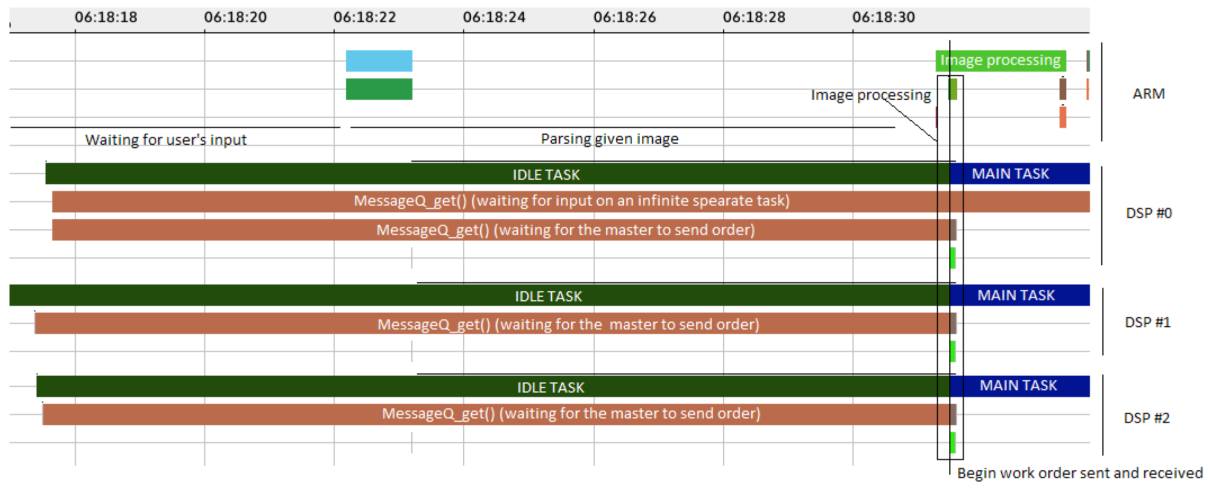

Figure 19 : Global view of the application

We first can see that the synchronization is working since the DSPs are all entering the main task (displayed in dark blue) approximately at the same time, *i.e* when they each receive the command from the master. Without trace synchronization, some processes might appear well ahead or behind of others, and any cause/effect relationship would be lost.

As highlighted, recognizing the different parts of the workflow is immediate: as long as the DSPs are in the "idle" task, they are waiting for the master's orders. If the master is not doing anything, then it means that it is itself waiting for user input.

Zooming on the beginning of the main task shows the main function, which actually processes a part of the image and allows one to see how long it took (see figure 20). In this case, the average processing time was around 96ms.

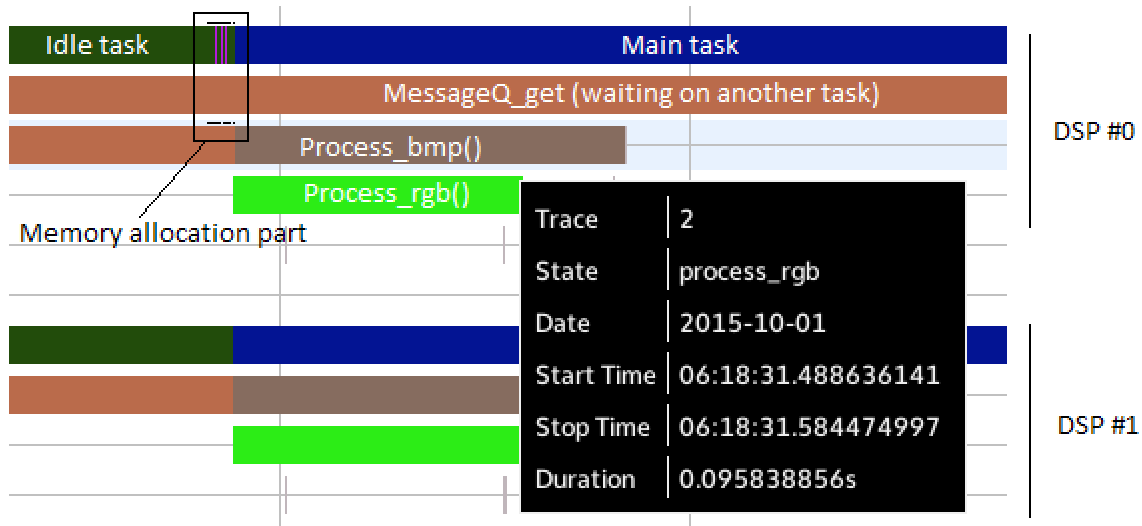

Figure 20 : Zoom on the main processing function.

Zooming on the memory allocation part better shows what are the interactions between the master and the first slave. Figure 21 displays this situation.

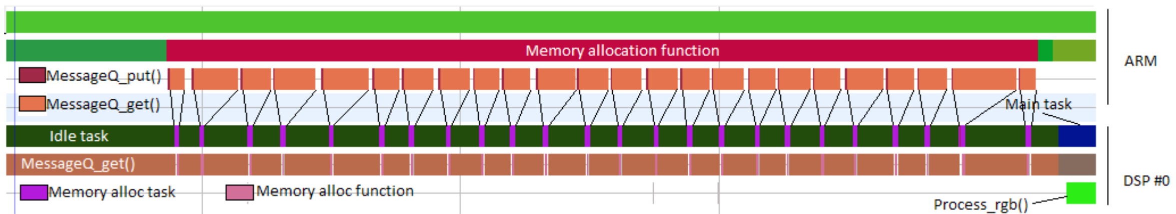

Figure 21 : Zoom on the memory allocation part

As we can see, there are some "ping-pong" exchanges going on between the master and the slave. By reading the state, one can see that the master is adding a message (a memory allocation command) inside the slave's queue and then waits for an answer. At the same time, the slave is waiting for an order, then allocates memory accordingly when one is received and finally sends back the information on said memory to its master.

This way, we can clearly see the dependencies between two heterogeneous cores, as one is always waiting for the other.

During normal execution, after this step comes the actual processing of the image, as shown in Figure 21.

We now examine a more problematic context where another high-priority task is awoken during the image processing. Because of its high priority, this task will run to completion before the image

processing task can be resumed. Without tracing, this problem would only be seen at the end, where the total processing time would jump from an average 96ms to 296ms. Finding the roots of this abnormal latency without tracing would be very hard.

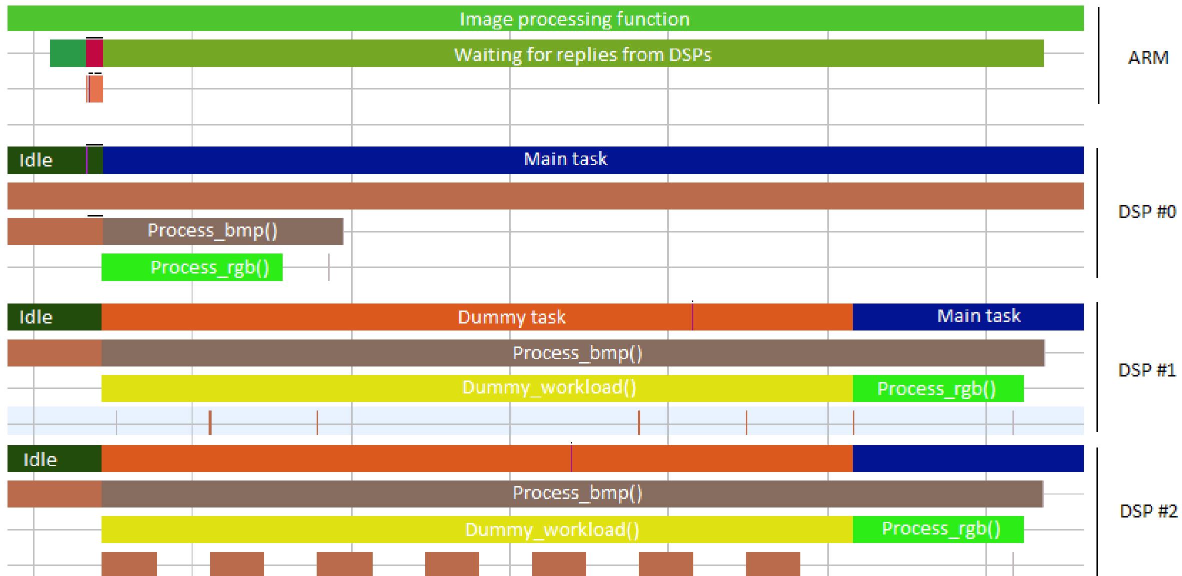

Figure 22 : Zoom on the problematic area

However, thanks to our instrumentation, one simple look at the main task section of the trace would reveal that DSPs 1 and 2 are actually preempted by another "dummy" task which makes them both communicate. Once the communication is over, they return to the main task, thus allowing the master to finally get all the results. As can be seen, other DSPs are not disturbed by this task. Only the master is also impacted since it is waiting for every slave to send back their share of the image. Figure 22 displays this situation.

4.7.3 Discussion

Thanks to the information gathered through tracing, we discovered that another task might preempt our image processing task and that it is exchanging data between cores 1 and 2. With that in mind, one might try (if possible) to protect the image processing task from being preempted, or insure that the conditions triggering the other task are only met once the image processing function is finished.

Detecting the previous problem as easily would not have been possible without the help of traces. Even though this example is a deliberate construction, we believe it to be fairly representative of

problems that we encountered in real systems. It thus demonstrates that system monitoring through tracing on an heterogeneous device is both possible and useful.

4.8 Conclusion and future work

Tracing heterogeneous embedded systems in a generic way, using CTF traces, is now possible thanks to LTTng, barectf and a synchronization daemon.

Bringing standard formats and tools to these new devices could help developers to easily adapt tracing methods to virtually any platform, thus allowing them to use multi-platform tools, avoiding the reliance on proprietary or limited single-platform tools. This would also allow quick comparisons, with the same tools, of the same application running on different platforms for performance analysis purposes.

The work done on the Parallella and Keystone 2 platforms reflects the possibilities of what can achieved on heterogeneous embedded systems and opens the way for even more possibilities, such as critical path analysis in an heterogeneous environment. In particular, we showed how a custom-made micro-kernel can be instrumented to obtain the same kind of information readily available on a linux-based device.

Even though the generic method described to obtain correlated traces on such devices appears fairly good, local implementations of barectf platforms could slightly influence the overhead generated by tracing. For instance, the platform we wrote for the Keystone 2 uses the MessageQ API, thus adding more latency than a simple shared memory-based solution. Moreover, barectf is still a work in progress and will surely see its features enhanced in the near future.

Traces visualization is also something that deserves further work. The presented callstack view will not easily scale to more than a few cores. A new approach will be required to visually investigate systems with a thousand cores.

In addition, it would be interesting to study the optimal frequency for generating synchronization points, as a compromise between performance overhead and synchronization accuracy. In theory, only two points in each hull are required to achieve a basic synchronization, but having more points brings more accuracy at the cost of some overhead.

Finally, we would like to see if trace synchronization can be directly achieved by matching regular message exchanges, without requiring an external synchronization process that adds dummy message exchanges for that purpose, just like we are able to synchronize traces thanks to existing TCP exchanges. Although this approach would lose its generality, it would remove this component of the overhead.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre nous revenons sur des points généraux liés à la solution présentée au chapitre précédent. Nous discutons également brièvement de travaux complémentaires.

5.1 Retour sur la solution proposée

5.1.1 Cas d'utilisation

Si le cas d'utilisation présenté précédemment introduit des charges de travail artificielles sur les DSPs dans le but de provoquer des latences, nous pensons qu'il représente néanmoins correctement une partie des défis auxquels doivent faire face les industries sur les systèmes hétérogènes embarqués.

En effet, la détection de latences et la mise en relation d'évènements survenant à la fois sur un processus maître et son esclave, sont des tâches impossibles à mener efficacement avec la seule aide des outils de profilage généralement à la disposition de l'utilisateur.

Un problème similaire à celui simulé dans notre étude pourrait très plausiblement se produire sous la forme d'une interruption matérielle ou logicielle ou encore par le biais de la réception d'un ordre du maître sur l'esclave. Si un tel comportement devait se produire, la solution proposée ici permettrait toujours de le détecter très rapidement, et pose les bases nécessaires à sa résolution.

5.1.2 Généralisation du traçage de *SYS/BIOS*

L'un des points importants du travail présenté dans ce mémoire réside dans l'instrumentation du micro-noyau SYS/BIOS, fournit par TI et utilisable sur tous les types de DSPs proposés par l'entreprise.

Être en mesure de pouvoir tracer le système à la base d'une vaste majorité des DSPs du marché est une excellente chose, nous rapprochant des possibilités offertes par l'instrumentation du noyau Linux.

En effet, le cas d'utilisation précédemment étudié montre comment des changements de contextes et des communications interprocessus peuvent être directement tracées dans le micro-noyau. Grâce au système de « hooks » introduit, il est aisé de placer des points de trace n'importe où dans le code du noyau, offrant ainsi des possibilités illimitées en termes d'informations recueillies.

Par ailleurs, le système de « hooks » proposé permet la configuration de l'instrumentation à adopter à travers les mêmes fichiers de configuration que le noyau lui-même. Grâce à cette méthode, les points de trace ajoutés ne sont même pas intrinsèquement liés à *barectf* et peuvent s'interfacer avec n'importe quel traceur.

Disposer d'une instrumentation complète du micro-noyau SYS/BIOS ouvrirait la porte à de nouvelles analyses telles que celle du chemin critique [71].

5.1.3 Contribution apportée

La principale contribution de la solution présentée ici est d'étendre une technique de synchronisation utilisée dans des systèmes distribués sur des plates-formes hétérogènes embarquées de manière générique, de l'interfacer avec un outil de traçage de systèmes baremetal (barectf) et de proposer une façon de visualiser les résultats obtenus. Nous créons ainsi le premier système capable de tracer tous les processus d'une plate-forme hétérogène embarquée pouvant comporter des processeurs sans systèmes d'exploitation.

5.1.4 Validité de la solution

La solution proposée pour tracer des plates-formes hétérogènes embarquées présentée dans ce document a pour vocation d'être la plus générique possible et ne nécessite que deux prérequis :

1. Le système étudié doit disposer d'une zone de mémoire partagée. D'autres méthodes de communication peuvent être employées au cas par cas selon le système pour améliorer les performances ou la simplicité de la solution.
2. Les coprocesseurs étudiés doivent supporter un mécanisme d'interruptions basique.

Sous ces conditions, et puisque l'implémentation des modules barectf doit, de toute façon, être adaptée à chaque plate-forme, il est raisonnable d'affirmer la généricité théorique de la solution proposée.

En outre, le fait de l'avoir testée et validée sur deux systèmes très différents : l'un très restreint et pouvant servir de modèle de prototypage, et l'autre beaucoup évolué pouvant être utilisé en production, permet d'affirmer que, malgré les différences entre la multitude de systèmes embarqués existants, la solution proposée a de grandes chances d'être directement applicable à une grande partie des systèmes embarqués hétérogènes.

5.2 Autres travaux

Barelog

Reprenant le même esprit que *barectf*, barelog[16] est un ensemble d'APIs C ayant pour but de fournir une solution de logging générique, facilement adaptable aux plates-formes hétérogènes embarquées. Aucun système d'exploitation n'est nécessaire à son fonctionnement.

Tout comme *barectf*, barelog requiert l'implémentation d'une interface donnant les fonctions de base pour la gestion des évènements de la part de l'utilisateur, afin de tenir compte des spécificités de l'architecture ciblée.

Il est possible, entre autres choses, de configurer la taille maximale prise par les tampons mémoire, de définir différentes priorités pour les évènements et d'utiliser des stratégies prédéfinies pour la gestion des évènements.

L'objectif de cet utilitaire est de réduire les temps de développement sur les processeurs sans système d'exploitation en proposant une solution facile et efficace pour effectuer du débogage rapide. Par définition, cette solution n'est pas destinée à être utilisée en production, si ce n'est pour faire de la journalisation légère.

Contribution à barectf

L'utilisation intensive de barectf lors des différentes étapes d'établissement de la solution proposée dans ce mémoire a conduit à la découverte de bogues mineurs. Les corrections proposées ont été acceptées et font désormais partie intégrante de la deuxième version de l'outil.

D'autres améliorations, telles que la possibilité d'intégrer nativement un système d'exclusion mutuelle entre les différents points de trace (normalement à la charge de l'utilisateur), sont également en réflexion.

Noyaux temps réel

Effectuer des travaux sur du matériel exotique est toujours une bonne occasion pour l'explorer plus en profondeur, parfois hors du cadre initial. À ce titre, nous avons vérifié la bonne intégration des

[16] https://github.com/SkinnerSweet/barelog

noyaux Linux temps réel (Linux-RT) aux deux plates-formes de test, dans l'optique d'exécuter des tests de performance (mesure des latences).

Les résultats obtenus sur la Keystone 2 semblaient être dans les normes attendues alors que ceux produits sur la Parallella étaient de presque un ordre de grandeur supérieurs. Cela était certainement dû à la non prise en charge d'un noyau « Full-RT » par la carte. Ainsi, la Keystone 2 enregistre (via l'outil *cyclictest*[17]) des latences moyennes de 13 microsecondes, avec une latence minimale de 12 microsecondes et maximale de 95 microsecondes. La Parallella, de son côté, produit des latences de l'ordre de 45 microsecondes en moyenne et de 18 microsecondes au minimum, avec des pics allant jusqu'à 24870 microsecondes.

Le chapitre suivant dresse une conclusion sur les travaux effectués et donne une vue d'ensemble des limitations de la solution dressée ainsi que de futures améliorations.

[17] https://rt.wiki.kernel.org/index.php/Cyclictest

CHAPITRE 6 CONCLUSION

Ce chapitre final conclut l'étude du traçage des systèmes hétérogènes embarqués en revenant sur les objectifs initiaux, les limitations de la solution proposée et des suggestions de travaux futurs.

6.1 Synthèse des travaux

L'objectif de ce travail de recherche était de proposer une solution générique de traçage de systèmes hétérogènes embarqués. L'idée était d'obtenir une vision globale de l'état du système, incluant les interactions entre ses processeurs et coprocesseurs, se rapprochant ainsi de ce qui existe en matière de traçage de systèmes distribués. La solution proposée montre comment, en combinant deux traceurs distincts et un processus de synchronisation des évènements générés, il est possible d'atteindre un tel objectif. En effet, en traçant un système complet de traitement d'image composé de processeurs ARM et de DSPs, nous avons montré qu'il est aisé d'exhiber les communications interprocessus et de détecter des problèmes courants de concurrence et de latence.

En particulier, nous avons pu atteindre tous les objectifs fixés. Nous avons ainsi bien vérifié l'intégration de LTTng à des processeurs ARM embarqués, nous avons également étendu barectf à une nouvelle architecture (et par construction, à la grande majorité des DSPs TI). Un mécanisme générique à base d'interruptions a également été mis en place pour assurer la synchronisation des évènements, généralisant ainsi l'approche reposant sur les paquets TCP dans les systèmes répartis. Finalement, nos différents tests et cas d'utilisation ont bien permis d'affirmer l'efficacité de la solution proposée mais aussi d'en exhiber les faiblesses.

La solution proposée dans ce travail a également un intérêt théorique et scientifique car il s'agit de la première méthode étendant et généralisant le traçage à toute plate-forme hétérogène embarquée sous des conditions raisonnables (voir section précédente). Cela laisse place à de nouvelles possibilités d'analyse de ces environnements et pourrait, de plus, permettre une uniformisation des outils utilisés sur ces systèmes.

6.2 Limitations de la solution

Même si la partie théorique de la solution proposée est totalement générique, nous n'avons testé son implémentation complète que sur deux plates-formes distinctes. Parce que les systèmes embarqués ont tendance à ne jamais se comporter de la même manière et à intégrer des composants

parfois « exotiques », il n’est pas possible de garantir complètement la généricité de l’implémentation de la solution.

L’implémentation de barectf sur la Keystone 2 repose sur l’API *MessageQ* de TI. Celle-ci utilise un système de communication par messages de haut-niveau, amenant plus de latence qu’une solution basée sur des communications en mémoire partagée. En outre, communiquer via la mémoire partagée est l’unique moyen universel de passer de l’information d’un processeur à un autre et en faire l’utilisation apporterait d’autant plus de généricité à la solution proposée.

Le cas d’utilisation présenté ici, bien que représentatif d’une partie des problèmes courants auxquels font face les entreprises, n’est pas absolument générique et ne se base pas sur un problème réellement rencontré. Seule l’utilisation de notre solution sur un problème réel concret pourrait définitivement attester de sa validité.

En outre, l’analyse graphique des traces proposée n’est pas adaptée à la visualisation de plus de quelques processeurs et sera complètement inutilisable sur des systèmes en dénombrant des centaines.

Finalement, barectf étant toujours à l’état de prototype, il est possible que des bogues aient échappé à notre attention et puissent nuire à la validité des traces obtenues dans certains cas très particuliers.

6.3 Suggestions futures

Afin de vérifier la généricité de la solution proposée, il serait intéressant d’obtenir des retours de son utilisation sur diverses plates-formes et divers cas d’utilisation se rapprochant plus des « vrais » problèmes auxquels sont confrontées les entreprises.

Il serait également judicieux d’intégrer le processus de synchronisation des traces, en charge de la génération des paires d’évènements, directement dans barectf. Cela nécessiterait, comme pour le reste, la définition d’interfaces à compléter par l’utilisateur.

Par ailleurs, il serait peut être possible de modifier le processus de synchronisation dans des cas particuliers afin de prendre en compte les communications réelles entre les processeurs et coprocesseurs. De cette façon, le système serait encore moins perturbé car les paires d’évènements seraient directement générées par des communications qui auraient eu lieu quoiqu’il arrive. Néanmoins, de telles modifications feraient perdre à la méthode originale sa généralité.

Finalement, nous manquons d'une analyse du rapport optimal de la fréquence de génération des paires d'évènements pour la synchronisation sur la précision de ladite synchronisation. En d'autres termes, il serait extrêmement intéressant d'estimer le seuil à partir duquel la fréquence de génération devient trop élevée et nuit aux performances sans apporter de précision additionnelle.

BIBLIOGRAPHIE

[1] M. Boyer, D. Tarjan, S. T. Acton, et K. Skadron, « Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors », in *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, 2009, p. 1-12.

[2] C. C. Liu et H. M. Hang, « Acceleration and Implementation of JPEG2000 Encoder on TI DSP Platform », in *2007 IEEE International Conference on Image Processing*, 2007, vol. 3, p. III-329-III-332.

[3] B. Poirier, R. Roy, et M. Dagenais, « Accurate offline synchronization of distributed traces using kernel-level events », *ACM SIGOPS Operating Systems Review*, vol. 44, nº 3, p. 75–87, 2010.

[4] M. P. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, R. C. P. Leal-Toledo, P. A. Pagliosa, L. Valente, et B. Feij, « A game loop architecture for the GPU used as a math coprocessor in real-time applications », *Computers in Entertainment*, vol. 6, nº 3, p. 1-19, oct. 2008.

[5] T. Aaberge, « Analyzing the performance of the Epiphany processor », 2014.

[6] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, et S. Vitabile, « An embedded, FPGA-based computer graphics coprocessor with native geometric algebra support », *Integration, the VLSI Journal*, vol. 42, nº 3, p. 346-355, juin 2009.

[7] S. Goswami, « An introduction to KProbes ». [En ligne]. Disponible sur: https://lwn.net/Articles/132196/. [Consulté le: 09-juill-2016].

[8] J. W. Wallace, B. D. Jeffs, et M. A. Jensen, « A real-time multiple antenna element testbed for MIMO algorithm development and assessment », in *IEEE Antennas and Propagation Society International Symposium, 2004*, 2004, vol. 2, p. 1716-1719 Vol.2.

[9] X. Cheng et L. Zhang, « A research of inter-process communication based on shared memory and address-mapping », in *2011 International Conference on Computer Science and Network Technology (ICCSNT)*, 2011, vol. 1, p. 111-114.

[10] K. Malvoni et J. Knezovic, « Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware », in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[11] P. Proulx, « barectf 2: Continuous Bare-Metal Tracing on the Parallella Board — LTTng ». [En ligne]. Disponible sur: http://lttng.org/blog/2015/07/21/barectf-2/. [Consulté le: 05-juill-2016].

[12]« CDT and Parallella ». [En ligne]. Disponible sur: http://www.eclipse.org/cdt/presentations/CDTAndParallella_EclipseCon_NA_2014/index.html#/. [Consulté le: 05-juill-2016].

[13] T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, et A. Wolfe, « Challenges to combining general-purpose and multimedia processors », *Computer*, vol. 30, nº 12, p. 33-37, déc. 1997.

[14] B. Gregg, « Choosing a Linux Tracer (2015) ». [En ligne]. Disponible sur: http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html. [Consulté le: 09-juill-2016].

[15] P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, « Combined Tracing of the Kernel and Applications with LTTng », *Proceedings of the Linux Symposium*, p. 87-94, 2009.

[16] M. Desnoyers, « Common Trace Format (CTF) Specification (v1.8.2) ». [En ligne]. Disponible sur: http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master. [Consulté le: 09-juill-2016].

[17] N. Nahro et J. Omar, « Communication mechanism among instances of many-core real time system », 2015.

[18] D. Flater, « Configuration of profiling tools for C/C++ applications under 64-bit Linux », National Institute of Standards and Technology, NIST TN 1790, mars 2013.

[19] S. J. Park, D. R. Shires, et B. J. Henz, « Coprocessor Computing with FPGA and GPU », in *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, 2008, p. 366-370.

[20] C. Q. Yang et B. P. Miller, « Critical path analysis for the execution of parallel and distributed programs », in *, 8th International Conference on Distributed Computing Systems, 1988*, 1988, p. 366-373.

[21] T. Hahn, J. Humphreys, A. Fritsch, et D. Greenstreet, « Demystifying digital signal processing (DSP) programming », mars 2015.

[22] M. Desnoyers et M. Dagenais, « Deploying LTTng on exotic embedded architectures », in *Embedded Linux Conference*, 2009, vol. 2009.

[23] P. Ekas et B. Jentz, « Developing and integrating FPGA coprocessors », *Embedded Computing Design*, Fall 2003.

[24] J. Chen et J.-H. Liu, « Developing Embedded Kernel for System-On-a-Chip Platform of Heterogeneous Multiprocessor Architecture », in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, 2006, p. 246-250.

[25] « Digital signal processor (DSP) with Linux ». [En ligne]. Disponible sur: https://relinux.de/en/applications/linux-dsp.html. [Consulté le: 05-juill-2016].

[26] A. S. Tanenbaum, *Distributed systems*. Prentice-Hall, 2007.

[27] A. Gatherer, T. Stetzler, M. McMahan, et E. Auslander, « DSP-based Architectures of Mobile Communications: Past, Present and Future », *IEEE Communications Magazine*, janv. 2000.

[28] « efficios/barectf », *GitHub*. [En ligne]. Disponible sur: https://github.com/efficios/barectf. [Consulté le: 08-juin-2016].

[29] « Ensemble de Mandelbrot », *Wikipédia*. 06-juill-2016.

[30] Adapteva, « Epiphany Architecture Reference (Rev. 14.03.11) ». 2014.

[31] A. Duda, G. Harrus, Y. Haddad, et G. Bernard, « Estimating global time in distributed systems », présenté à Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin, 1987, vol. 18.

[32] G. Steiner, K. Shenoy, D. Isaacs, et D. Pellerin, « How to accelerate algorithms by automatically generating FPGA coprocessors | EE Times », *EETimes*. [En ligne]. Disponible sur: http://www.eetimes.com/document.asp?doc_id=1274524. [Consulté le: 07-juill-2016].

[33] D. L. Mills, « Internet time synchronization: the network time protocol », *IEEE Transactions on Communications*, vol. 39, nº 10, p. 1482-1493, oct. 1991.

[34] « Interprocess communications system and method utilizing shared memory for message transfer and datagram sockets for message control ».

[35] P. Ficheux, « Introduction à Ftrace », *Linux Embedded*. .

[36] « IPM -- Profiling vs. Tracing ». [En ligne]. Disponible sur: http://ipm-hpc.sourceforge.net/profilingvstracing.html. [Consulté le: 05-juill-2016].

[37] A. Olofsson, T. Nordström, et Z. Ul-Abdin, « Kickstarting high-performance energy-efficient manycore architectures with Epiphany », in *2014 48th Asilomar Conference on Signals, Systems and Computers*, 2014, p. 1719-1726.

[38] D. Couturier et M. R. Dagenais, « LTTng CLUST: A System-Wide Unified CPU and GPU Tracing Tool for OpenCL Applications », *Advances in Software Engineering*, vol. 2015, p. 1-14, 2015.

[39] EfficiOS, « LTTng v2.7 Documentation — LTTng ». [En ligne]. Disponible sur: https://lttng.org/docs/. [Consulté le: 01-déc-2015].

[40] « MCSDK User Guide for KeyStone II - Texas Instruments Wiki ». [En ligne]. Disponible sur: http://processors.wiki.ti.com/index.php/MCSDK_User_Guide_for_KeyStone_II. [Consulté le: 05-juill-2016].

[41] « Method of inter-process communication in a distributed data processing system ».

[42] Texas Instrument, « Multicore DSP+ARM KeyStone II System-on-Chip (SoC) ». nov-2012.

[43] L. Lamport, « On interprocess communication », *Distrib Comput*, vol. 1, nº 2, p. 86-101, juin 1986.

[44] M. Jabbarifar, « On line trace synchronization for large scale distributed systems », École Polytechnique de Montréal, 2013.

[45] M. Jabbarifar, M. Dagenais, R. Roy, et A. S. Sendi, « Optimum off-line trace synchronization of computer clusters », *Journal of Physics: Conference Series*, vol. 341, p. 12029, févr. 2012.

[46] Adapteva, « Parallella -1.x Reference Manual (Rev. 14.09.09) ». 2014.

[47] « Parallella Community • View topic - How to check the state of the Epiphany from the ARM ». [En ligne]. Disponible sur: https://parallella.org/forums/viewtopic.php?f=49&t=986. [Consulté le: 05-juill-2016].

[48] « Perf Wiki ». [En ligne]. Disponible sur: https://perf.wiki.kernel.org/index.php/Main_Page. [Consulté le: 09-juill-2016].

[49] W. S. Huang, C. W. Liu, P. L. Hsu, et S. S. Yeh, « Precision Control and Compensation of Servomotors and Machine Tools via the Disturbance Observer », *IEEE Transactions on Industrial Electronics*, vol. 57, nº 1, p. 420-429, janv. 2010.

[50] A. Varghese, B. Edwards, G. Mitra, et A. P. Rendell, « Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor ».

[51] G. Saxena, S. Ganesan, et M. Das, « Real time implementation of adaptive noise cancellation », in *2008 IEEE International Conference on Electro/Information Technology*, 2008, p. 431-436.

[52] R. Haas, « Robert Haas: perf: the good, the bad, the ugly ». .

[53] D. Russo, « RTSC Home page ». [En ligne]. Disponible sur: http://www.eclipse.org/rtsc/. [Consulté le: 04-juill-2016].

[54] M. Durrant, J. Dionne, et M. Leslie, « Running Linux on a DSP? Exploiting the Computational Resources of a programmable DSP Micro-Processor with uClinux », in *Ottawa Linux Symposium*, 2002, p. 130.

[55] B. Poirier, « Synchronisation de traces distribuées à l'aide d'événements de bas niveau », École Polytechnique de Montréal, 2010.

[56] Texas Instrument, « SYS/BIOS Inter-Processor Communication (IPC) 1.25 User's Guide ». sept-2012.

[57] F. Tchakounté et P. Dayang, « System calls analysis of malwares on android », *International Journal of Science and Technology*, vol. 2, nº 9, p. 669–674, 2013.

[58] R. Gusella et S. Zatti, « The Berkeley UNIX 4.3BSD Time Synchronization Protocol », juin 1985.

[59] M. Desnoyers et M. Dagenais, « The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux », *Proceedings of the Linux Symposium*, vol. 1, p. 209-224, 2006.

[60] R. Rajkumar, M. Gagliardi, et L. Sha, « The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation », in *Real-Time Technology and Applications Symposium, 1995. Proceedings*, 1995, p. 66-75.

[61] L. Lamport, « Time, clocks, and the ordering of events in a distributed system », *Communications of the ACM*, vol. 21, nº 7, p. 558–565, 1978.

[62] Texas Instrument, « TMS320C66x DSP CorePac User Guide (Rev. C) ». juill-2013.

[63] P. Proulx, « Tracing Bare-Metal Systems: a Multi-Core Story — LTTng ». [En ligne]. Disponible sur: https://lttng.org/blog/2014/11/25/tracing-bare-metal-systems/. [Consulté le: 01-déc-2015].

[64] S. Sharma, « Turbocharged Tracing with LTTng », *Open Source For You*, mars 2014.

[65] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, et J. Dongarra, « Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment », in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014, p. 491-500.

[66] B. N. Bershad, T. E. Anderson, E. D. Lazowska, et H. M. Levy, « User-level interprocess communication for shared memory multiprocessors », *ACM Transactions on Computer Systems (TOCS)*, vol. 9, nº 2, p. 175–198, 1991.

[67] S. Rostedt, « Using the TRACE_EVENT() macro ». [En ligne]. Disponible sur: http://lwn.net/Articles/379903/. [Consulté le: 09-juill-2016].

[68] N. Nethercote, « Using Valgrind to detect undefined value errors with bit-precision ». [En ligne]. Disponible sur: http://static.usenix.org/legacy/events/usenix05/tech/general/full_papers/seward/seward_html/. [Consulté le: 09-juill-2016].

[69] N. Nethercote et J. Seward, « Valgrind: a framework for heavyweight dynamic binary instrumentation », in *ACM Sigplan notices*, 2007, vol. 42, p. 89–100.

[70] « Valgrind Home ». [En ligne]. Disponible sur: http://valgrind.org/. [Consulté le: 09-juill-2016].

[71] F. Giraldeau et M. Dagenais, « Wait analysis of distributed systems using kernel tracing », *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, nº 99, p. 1-1, 2015.

[72] T. Innovation, speed of next generation mobile devices I. his spare time, he enjoys all the outdoor activities S. D. has to offer P. holds a B. in electrical engineering, computer science from U. Berkeley, a M. from UCLA, et erson., « What's So Special about the Digital Signal Processor? », *Qualcomm*, 06-déc-2013. [En ligne]. Disponible sur: https://www.qualcomm.com/news/onq/2013/12/06/whats-so-special-about-digital-signal-processor. [Consulté le: 07-juill-2016].

[73] F. Pierre Doray, « Analyse de variations de performance par comparaison de traces d'exécution », masters, École Polytechnique de Montréal, 2015.

[74] « Loi de Moore », *Wikipédia*. 02-juill-2016.