HAL
open science

Analyse de systèmes embarqués par structuration de traces d'exécution

Alexis Martin, Generoso Pagano, Jérôme Correnoz, Vania Marangozova-Martin

To cite this version:

Alexis Martin, Generoso Pagano, Jérôme Correnoz, Vania Marangozova-Martin. Analyse de systèmes embarqués par structuration de traces d'exécution. ComPAS 2014 : conférence en parallélisme, architecture et systèmes, Apr 2014, Neuchâtel, Suisse. hal-00989532

HAL Id: hal-00989532

https://inria.hal.science/hal-00989532v1

Submitted on 12 May 2014

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

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

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Analyse de systèmes embarqués par structuration de traces d'exécution *

Alexis Martin[1], Generoso Pagano[1], Jérôme Correnoz[2], Vania Marangozova-Martin[3]

[1] Inria Grenoble, {alexis.martin,generoso.pagano}@inria.fr
[2] STMicroelectronics, jerome.correnoz@st.com
[3] Université Joseph-Fourier, vania.marangozova-martin@imag.fr

---

Résumé
Le traçage d'une application est une technique classique utilisée lors de l'optimisation et du débogage. Toutefois, dans le domaine embarqué, les traces d'exécution sont volumineuses et difficiles à exploiter. Dans ce papier, nous proposons une structuration d'un modèle événementiel de traces qui garde la généricité de représentation des données, tout en améliorant l'efficacité d'analyse. Nous montrons que ce modèle permet un traitement plus rapide avec une empreinte mémoire faible. L'approche est validée grâce à des scénarios réels du monde industriel en collaboration avec STMicroelectronics.

Mots-clés : Systèmes embarqués, trace d'exécution, SoC-TRACE, modèle de données, analyse.

---

1. Introduction

Les systèmes embarqués sont devenus omniprésents et apparaissent dans toutes sortes d'objets "intelligents". Toutefois, la diversité des usages et la complexification des architectures matérielles représentent de vrais défis pour le processus de développement et de mise au point.

Les traces d'exécution [16, 22, 24] sont largement utilisés pour la compréhension, le débogage et l'optimisation du comportement des systèmes. Toutefois, dans le cadre des systèmes embarqués, l'exploitation des traces est face à des problèmes liés à la taille des traces (plusieurs Gigaoctets pour quelques secondes de décodage) et au manque de sémantique des informations système capturées (interruptions, changements de contexte). La majorité des outils existants sont propriétaires [6,8,20] et se limitent à fournir une représentation graphique des données de traces et au calcul de quelques métriques statistiques.

Le projet SoC-TRACE [4] a pour objectif de faire avancer l'état de l'art concernant l'exploitation de traces d'exécution dans le domaine des systèmes embarqués. Le projet s'intéresse au développement de nouvelles méthodes d'analyse de traces, d'une part, et à la conception d'infrastructure logicielle standardisant les traitements de stockage, d'analyse et de visualisation de traces, d'autre part. Dans ce contexte, nous présentons notre travail sur la structuration de la représentation interne des traces d'exécution. Nous montrons qu'il est possible d'introduire

*. Financé par le projet FUI SoC-TRACE.

de la sémantique métier et d'accélérer les traitements d'analyse. Nous validons notre approche à travers un scénario réel fourni par STMicroelectronics.

L'article est organisé comme suit. La Section 2 dresse un bref état de l'art des solutions d'exploitation de traces existantes. Dans la Section 3, nous présentons le projet SoC-TRACE et l'architecture du prototype actuel. Nous décrivons le principe de notre contribution dans la Section 4 et discutons de la validation en Section 5. Nous exposons en Section 6 les limites de notre approche et concluons en Section 7.

2. Autour de la gestion de traces : état de l'art

La gestion de traces comprend deux phases principales : l'acquisition et l'analyse. La phase d'acquisition consiste à intercepter et à enregistrer les informations composant la trace. La phase d'analyse soumet la trace à différents types de traitements afin d'en extraire de la connaissance sur le fonctionnement du système.

Dans le domaine des systèmes d'exploitation [11] et des systèmes embarqués [1], les solutions de traçage se concentrent majoritairement sur l'aspect d'acquisition. Elles proposent des mécanismes d'instrumentation des systèmes et des formats des traces [15,23] qui visent à maximiser la quantité d'informations capturées tout en minimisant l'intrusion. Les solutions sont propriétaires ou spécifiques au domaine d'application et donc difficilement réutilisables dans des contextes différents. Surtout, l'analyse des traces est laissée à l'appréciation des développeurs expérimentés.

Dans le domaine du calcul haute performance, les années d'expérience avec les applications parallèles et l'existence de modèles de programmation comme OpenMP [10] ou MPI [12] ont permis l'identification de situations typiques à problèmes. Ainsi, en plus d'outils d'instrumentation efficace comme TAU [21], il existent des outils comme Scalasca [13] ou Vampir [17] qui permettent d'analyser et de détecter automatiquement des situations anormales. Si les différents outils de gestion de traces proposent tous des fonctionnalités de capture et d'analyse intéressantes, il ne sont malheureusement pas toujours inter-opérables et ne peuvent généralement pas être utilisés conjointement. Ce dernier point est adressé par le projet Score-P [9] qui vise à faire collaborer les outils existant et définir des standards.

Dans ce contexte, SoC-TRACE se positionne comme un projet qui développe l'aspect analyse de traces pour les systèmes embarqués, tout en s'intéressant à l'interopérabilité des traitements d'analyse et leur standardisation [18,19].

3. Le projet SoC-TRACE

Après une introduction du contexte général du projet SoC-TRACE (Section 3.1), nous présentons l'architecture du prototype actuel (Section 3.2) et le modèle générique pour les données de trace (Section 3.3).

3.1. Contexte général

Le projet SoC-TRACE [4] est un projet du Ministère de l'Industrie (projet FUI) avec comme partenaires les sociétés STMicroelectronics, ProBayes et Magillem Design Services. Du coté académique participent l'Université Joseph Fourier et Inria Grenoble.

SoC-TRACE a pour objectif le développement d'une infrastructure logicielle pour l'exploitation de traces d'exécution d'applications embarquées multicoeur qui permettra à leurs développeurs de les optimiser et les valider plus rapidement. Les verrous technologiques sont

liés à la complexité d'une exécution multicoeur, au volume considérable des traces collectées et au manque de généricité et d'extensibilité des outils d'analyse. Sur ce dernier point, SoC-TRACE s'intéresse aux mécanismes nécessaires au stockage, au traitement et à la visualisation des traces. L'objectif est de pouvoir stocker des traces de formats hétérogènes tout en assurant un accès efficace aux données. L'infrastructure doit faciliter l'intégration de différents outils d'analyse, permettre l'enchainement de différents traitements et pouvoir sauvegarder les résultats obtenus.

3.2. L'infrastructure FrameSoC

Le premier prototype de l'infrastructure logicielle, FrameSoC, est développé sous forme d'outil intégré à Eclipse qui est le standard *de facto* dans les environnements de développement pour systèmes embarqués. Son architecture est présentée sur la Figure 1.

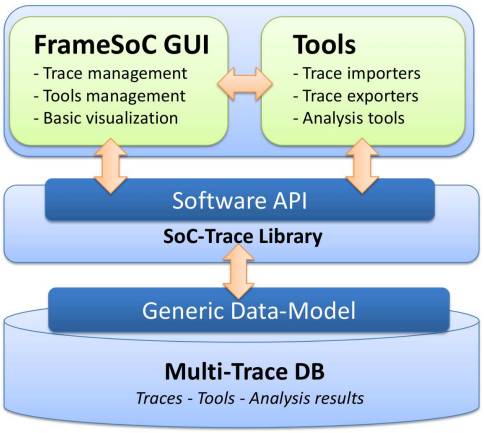


FIGURE 1 – Architecture de FrameSoC.

Les trois couches de l'architecture de FrameSoC concernent respectivement le stockage des traces (`MultiTrace DB`), l'accès aux données (`SoC-TRACE Library`) et les outils d'analyse (`GUI & Tools`). Pour le stockage, nous avons défini un modèle de données générique pour les données des traces. L'implantation actuelle est faite au sein d'une base de données relationnelle (SQLite [7]) mais pourrait être basée sur un autre support de stockage comme par exemple un système de fichiers. En ce qui concerne l'accès aux données, FrameSoC fournit une bibliothèque avec toutes les fonctions de base de manipulation de données comme la lecture, l'écriture ou le filtrage. En ce qui concerne les outils d'analyse de traces, FrameSoC vient actuellement avec une interface utilisateur simple, ainsi que des outils de base d'agrégation et de calcul statistique.

3.3. Le modèle de données FrameSoC

Le modèle de données FrameSoC (Figure 2) sépare les données sauvegardées en trois parties : les données contenues dans la trace (*Raw Trace Data*), les informations sur l'acquisition de la trace (*Trace General Information*) et les résultats d'analyse (*Analysis Data*).

L'entité qui représente une trace est (`TRACE`). Elle a pour rôle d'enregistrer les conditions expérimentales (logiciels et plateforme matérielle) dans lesquelles la trace a été produite. Une trace est composée d'événements (`EVENT`) qui représentent les actions se produisant au sein d'un système et déclenchées par des producteurs (`EVENT_PRODUCER`). Des exemples de producteurs d'événements sont typiquement les processus ou les threads. Les outils (`TOOL`) d'analyse

de traces produisent des résultats (ANALYSIS_RESULT) qui peuvent être des annotations, des groupes, des filtres ou autres.

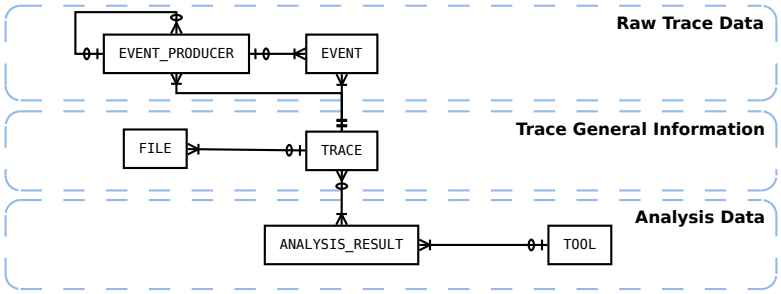


FIGURE 2 – Modèle de données dans FrameSoC.

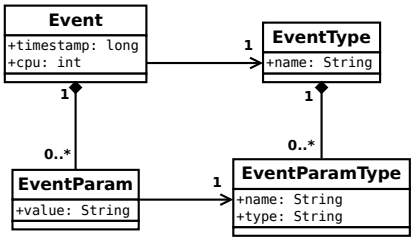


FIGURE 3 – Modèle auto-défini.

Pour pouvoir représenter différentes informations, l'entité EVENT est auto-descriptive (*Self Defining Pattern*, Figure 3). En effet, chaque entité EVENT a des attributs génériques (e.g l'estampille) qui sont communs à tous les événements. Ces attributs minimalistes peuvent être étendus avec des informations métier spécifiques en utilisant les entités EVENT_TYPE et EVENT_PARAM_TYPE. Ainsi, chaque EVENT_TYPE référence plusieurs EVENT_PARAM_TYPE où il spécifie les différents types d'informations qui peuvent être attachées à un événement. Les valeurs spécifiques de ces paramètres vont se retrouver dans des entités EVENT_PARAM.

Comme illustré à la Figure 4, en termes de stockage, il existe deux tables : la table des événements qui contient les données des entités EVENT et la table des paramètres qui correspond aux entités EVENT_PARAM. Dans l'exemple, la table des événements contient un événement avec l'identifiant 23. Son estampille est accessible directement. Par contre, l'accès aux informations de Context et de PrevContext nécessite une jointure avec la table des paramètres ce qui est plus lent.

ID	Type	Estampille
...	...	...
23	switch_to	1018502
24	switch_to	1078487
25	switch_to	1078494
...	...	...

(a) table des événements (EVENT)

ID	Type	ID événement	Valeur
...	...	...	...
56	Context	23	0
57	PrevContext	23	1021
58	Context	24	20
59	PrevContext	24	0
60	Context	25	0
61	PrevContext	25	20
...	...	...	...

(b) table des paramètres (EVENT_PARAM)

FIGURE 4 – Exemple de table des événements et des paramètres dans FrameSoC.

4. Structuration de la trace par un modèle de données enrichi

Le modèle de données FrameSoC est générique et permet de représenter de nombreux formats de traces. Néanmoins, c'est une modèle structurel qui n'impose que très peu de sémantique sur les événements et les informations les concernant. La sémantique métier d'une trace obtenue d'un système embarqué se retrouve cachée dans les paramètres additionnels et ne peut être exploitée efficacement sans la connaissance préalable du contexte d'acquisition de la trace.

Nous proposons une évolution du modèle de données FrameSoC. Il s'agit d'enrichir les informations génériques qui caractérisent les événements en y ajoutant la notion de *catégorie*. En nous inspirant des travaux existant dans le domaine du calcul parallèle et de la visualisation de traces [14], nous introduisons quatre catégories : événement ponctuel, état, lien et variable.

— *Événement ponctuel* : La notion d'événement ponctuel correspond à la notion actuelle d'événement dans FrameSoC. Il s'agit d'une action se produisant au sein du système à un moment donné.
— *État* : Un état représente un intervalle avec un début et une fin. Les états peuvent être imbriqués et modélisent des phases de calcul, des exécutions des fonctions, etc.
— *Lien* : Un lien représente une relation entre deux entités. La relation peut être une relation causale, une relation de communication ou autre.
— *Variable* : Une variable suit l'évolution d'une valeur numérique pendant le temps. Elle peut représenter la valeur d'un paramètre mesuré ou alors une valeur calculée.

En termes de modèles de données, les événements FrameSoC se retrouvent avec trois attributs supplémentaires pour refléter leur catégorisation. Le premier attribut donne la catégorie, alors que les deux autres attributs contiennent des précisions sur celle-ci. Ainsi, pour un état, ces deux attributs contiennent l'estampille de fin d'état et le niveau d'imbrication. Pour une variable, les attributs contiennent l'identifiant de la variable et sa nouvelle valeur. Dans le cas d'un lien, les deux attributs contiennent l'entité destinataire et l'estampille de fin de relation.

En termes de stockage, les informations sur les catégories sont introduites dans la table des événements et bénéficient d'un accès direct. L'idée de base est de garder la séparation entre les informations communes à toutes traces (génériques) et les informations spécifiques à leurs contexte (métier). Néanmoins, en ajoutant des catégories, nous transférons une partie de la sémantique métier vers les informations génériques. L'évolution du modèle est rétro compatible, il est toujours possible d'exprimer tous les événements comme événement ponctuel.

Nous avons implémenté notre proposition de catégorisation au sein d'un environnement synthétique et dans FrameSoC. L'environnement synthétique utilise la base de données SQLite, son interface C/C++ et l'outil statistique R [5]. Il a l'avantage d'être simple d'utilisation et de servir nos besoins de prototypage.

5. Validation

Dans cette section nous évaluons de manière qualitative et quantitative notre proposition. Dans le première partie (5.1), nous expliquons comment nous avons assigné les catégories aux événements de la trace brute. Dans la seconde partie (5.2) nous exposons l'étude qualitative, qui montre sur un cas réel que l'introduction de catégories facilite l'analyse de la trace. Dans la troisième partie (5.3) nous montrons qu'utiliser les catégories permet d'obtenir de meilleurs performances.

5.1. Catégorisation de la trace pour un cas réel fourni par STMicroelectronics

Le scénario fourni par STMicroelectronics concerne le fonctionnement d'un décodeur vidéo. Le décodeur lit un fichier depuis le réseau et le diffuse sur sa sortie vidéo. Durant l'exécution surviennent des sauts d'images et des craquements audio qui indiquent un problème de décodage.

La trace brute correspondant à ce scénario est produite en traçant les différentes couches logicielles (noyau Linux, intergiciel ST). La trace fait 10Mo pour 150000 événements et correspond à environ 45 secondes de décodage. Le fichier de trace contient des informations sur appels de fonctions, les interruptions logiciels et matériels, les communications réseau, etc.

Le format de trace utilisé est basé sur KPTrace [3], un format de traces de STMicroelectronics. La figure 5 montre un extrait de la trace correspondant au scénario de décodage. La ligne 1 liste les différentes informations relatives à un événement : estampille de début, estampille de fin, la durée, la durée d'activité effective, etc. Ainsi, la ligne 5 concerne un événement d'interruption, généré par "eth0", s'étant produit sur le cpu0 au temps 1092491 et ayant une durée de 8µs.

```
1| Start,End,Time,Active,Name,Context,PrevContext,CPU,Arg1,Arg2,Arg3,Arg4,message,ReturnValue,
2| ...
3| 266047,,,,sys_select,1036 (sshd),,0,0x0000000b,0x01a80e18,0x01a80dc8,0x00000000,,,
3| 1018502,,,,__switch_to,0,1021 (flush-0:11),0,,,,,,,
4| 1078487,,,,__switch_to,20 (kworker/0:1),0,0,,,,,,,
5| 1092491,1092499,8,8,Interrupt,Interrupt 168 (GIC  eth0),,0,,,,,,,
6| 1092501,1092581,80,80,SoftIRQ,SoftIRQ (net_rx_action),,0,,,,,,,
7| ...
```

FIGURE 5 – Extrait de la trace brute correspondant au scénario de décodage vidéo.

Nous avons traité cette trace pour y introduire nos catégories. Le résultat est illustré dans la Figure 6. Nous y retrouvons une colonne pour la catégorie et deux colonnes (`A` et `B`) pour les deux attributs correspondant. Nous avons utilisé les deux premiers attributs des événements de la trace et avons procédé comme suit :

- Si l'événement possède deux estampilles, il est catégorisé comme un *état*. Ainsi, les événements des lignes 5 et 6 de la Figure 5 sont ceux avec identifiants 25 et 26 de la Figure 6. L'événement avec ID = 26 de la Figure 6 (ligne 6 de la Figure 5) est de type `SoftIRQ`, le producteur est `net_rx_action`, la durée est de 80µs et c'est un état non imbriqué (B='/').
- Si l'événement n'a qu'une information temporelle, il est catégorisé comme un *événement ponctuel*. C'est le cas de l'événement `sys_select` ligne 3 de la Figure 5 et ID = 22 de la Figure 6.
- En nous inspirant des outils de visualisation pour KPTrace et afin de représenter l'information de manière plus compacte, nous effectuons un traitement spécifique pour les événements de type `switch_to`. Ces événements sont catégorisés comme *liens* exprimant le passage d'un contexte d'exécution (`PrevContext`) à un autre (`Context`). Pour l'événement avec ID = 23 (ligne 4), par exemple, le contexte de départ est enregistré comme le producteur de l'événement (`PrevContext`=1021), alors que le nouveau contexte est dans un des paramètres additionnels pour la catégorie (B=0).

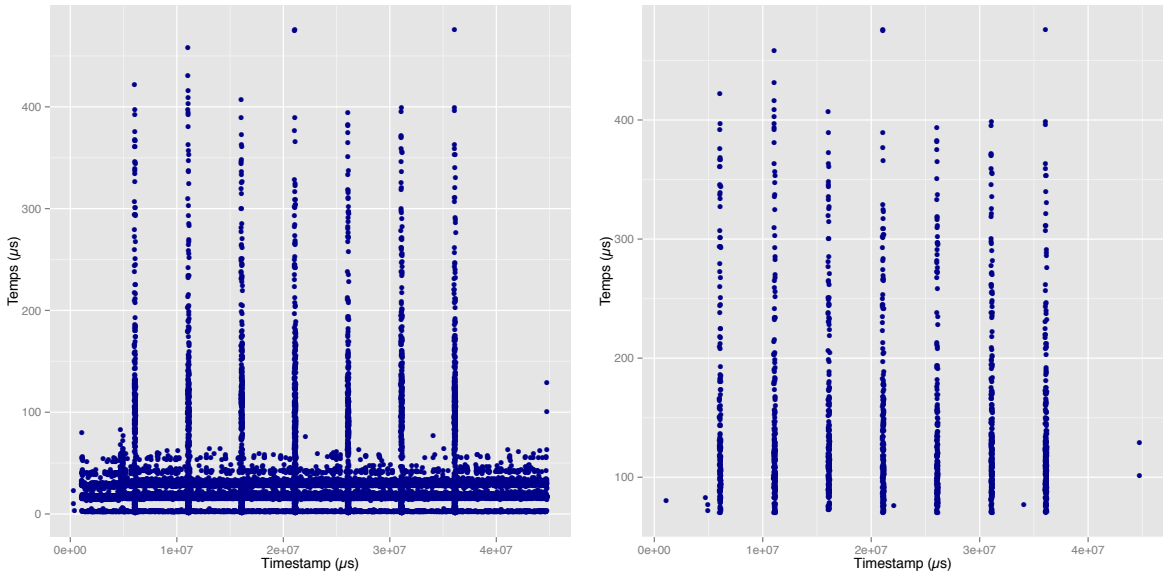
ID	Type	Producteur	Estampille	Catégorie	A	B
...	...	...	...	...	...	...
22	sys_select	1036 (sshd)	266047	Évt. ponctuel	-	-
23	switch_to	1021 (flush-0 :11)	1018502	Lien	0	0
24	switch_to	0	1078487	Lien	0	20 (kworker...)
25	Interrupt	Interrupt 168 (GIC ...)	1092491	Etat	8	/
26	SoftIRQ	SoftIRQ (net_rx...)	1092501	Etat	80	/
...	...	...	...	...	...	...

FIGURE 6 – Extrait de la trace catégorisée correspondant au scénario de décodage vidéo.

5.2. Analyse de la trace catégorisée

Les sauts d'images et les craquements indiquent un problème de respect des échéances temporelles lors du décodage. Pour retrouver la cause de ce problème, nous nous sommes concentrés sur les durées d'exécution des différentes actions au sein du système i.e sur les événements de catégorie *état*. Pour l'analyse, nous avons utilisé des mesures statistiques simples, notamment la moyenne des durées $\bar{x}$ et l'écart-type σ. Afin de détecter des déviances dans les durées d'exécution, nous avons recherché les événements dont la durée s'écarte considérablement de la moyenne : $|x - \bar{x}| > 3 \times \sigma$.

Nous avons repéré un type d'événement qui pose problème : *softIRQ*. La figure 7.a représente les durées de tous les *softIRQ* tracés. La majorité des événements ont un le temps d'exécution qui ne dépasse pas les 100µs. Toutefois, nous remarquons clairement un phénomène périodique où les durées sont supérieures. Les points qui ne font pas partie de l'intervalle de confiance par rapport à la moyenne, sont représentés à la figure 7.b.

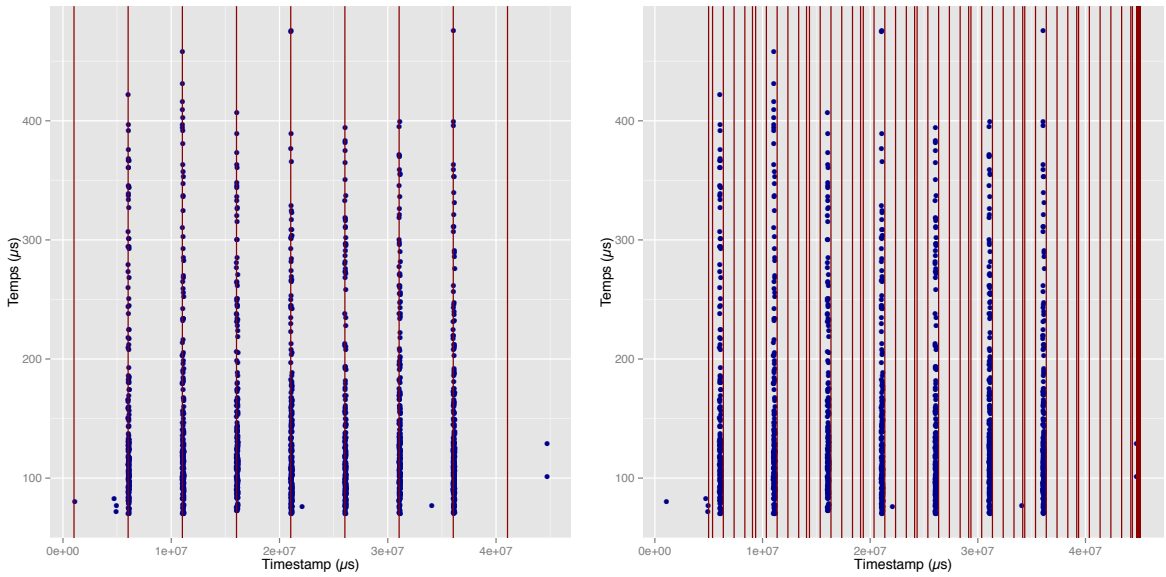


(a) Durées de tous les appels à *softIRQ*

(b) Durées hors de l'intervalle de confiance

FIGURE 7 – Durées des appels à *softIRQ*.

Nous avons essayé de corréler, en utilisant l'information de temps, les événements `SoftIRQ` trop longs avec d'autres événements. La figure 8 représente deux de ces corrélations. La Figure 8.a montre les appels à la fonction `flush`, alors que la Figure 8.b nous montre les appels à la fonction `cyclesoaks`. Les moments des appels sont indiqués par des barres verticales. Contrairement aux appels à `cyclesoaks`, les appels à `flush` coïncident exactement avec les appels à `SoftIRQ` trop longs. Cette comparaison à été faite pour tous les événements et les appels à `flush` sont les seuls qui corespondent.



(a) correlation avec les appels à *flush*

(b) correlation avec les appels à *cyclesoaks*

FIGURE 8 – Correlation entre la durée de *softIRQ* et des événements.

On remarque cependant que le premier et le dernier appel à `flush` n'ont pas d'impact sur le temps de `SoftIRQ` (Figure 8.a). Nous avons donc fait un filtrage de ces événements en fonction de la prochaine fonction appelée. Cette informations est facilement accessible à travers les événements liens `switch_to`. La figure 9 nous montre les appels à `flush` suivies de `kworker`. On remarque que ce filtrage à supprimé les deux événements au début et à la fin de la période tracée qui n'avaient pas d'impact. Toutefois, ce filtrage fait également apparaître un "trou" en milieu de période. La question est donc si ce point peut être négligé et si le ralentissement est vraiment lié aux appels de `flush` suivis d'appels à `kworker`. Nous pouvons donc proposer au développeur un point d'entrée pour le débogage qui correspond aux appels de `flush` suivis de `kworker`.

Notre analyse est valide par rapport à la connaissance des développeurs STMicroelectronics sur le scénario d'utilisation considéré.

5.3. Performances

Nous avons introduit les catégories au niveau de la table des événements où ils peuvent être accédés de manière immédiate. L'idée est de pouvoir effectuer une première analyse de la trace

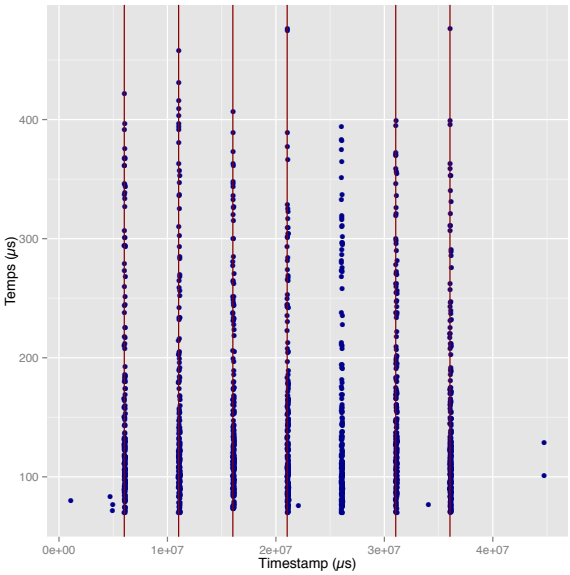


FIGURE 9 – Correlation entre ralentissement et appels à *flush* suivis de *kworker*

en n'accédant qu'aux informations de catégorie. Ainsi, l'analyse peut bénéficier de temps d'accès rapides et de quantités réduites de données à charger. Nous démontrons ces deux points en étudiant les performances au sein de notre environnement synthétique et dans FrameSoC (cf. 3.3).

5.3.1. Temps d'accès aux données

Au sein de notre environnement synthétique (SQLite,C++), nous avons comparé le temps d'accès aux informations dans la table des événements (accès direct) versus le temps d'accès dans la table des paramètres (accès indirect). Pour cela, nous avons généré plusieurs traces de tailles différentes (10^4, 10^5 et 10^6 événements) et avec un nombre différent de paramètres (1, 5 et 10). Nous avons mesuré le temps d'accès à un événement pris au hasard. Nous donnons les valeurs moyennes, obtenues sur 30 tests avec un écart-type $< 5\%$.

Les mesures sur les temps d'accès directs sont présentés dans la Table 10.a. Nous constatons que le temps est constant et ne dépend ni du nombre d'événements dans la trace, ni du nombre de leurs paramètres.

En revanche, la Table 10.b montre que le temps pour un accès indirect est proportionnel à la taille de la trace ou au nombre de paramètres des événements. Ceci s'explique par le fait qu'il est nécessaire de faire une jointure pour récupérer les paramètres correspondant à un événement particulier, puis ensuite de sélectionner le bon parmi l'ensemble. Avec un nombre d'événements et de paramètres croissant, le facteur entre un accès direct et un accès indirect passe de 1 à 420. Ces valeurs penchent clairement vers les accès directs pour les catégories et donc leur introduction dans la table des événements.

5.3.2. Temps de chargement et place mémoire des données

Pour travailler avec les données d'une trace, il est nécessaire de les charger en mémoire depuis le stockage persistant. En introduisant les catégories dans les traces, nous faisons l'hypothèse que l'analyse de trace pourra démarrer en ne considérant que les informations correspondantes

# paramètres	# événements		
	10k	100k	1000k
1	2.0	2.2	2.0
5	2.0	2.0	2.2
10	2.0	2.0	2.1

(a) Accès direct (ms), table des événements

# paramètres	# événements		
	10k	100k	1000k
1	3.2	12.7	92.7
5	7.5	49.0	446.5
10	12.6	92.1	882.4

(b) Accès indirect (ms), table des paramètres

FIGURE 10 – Temps d'accès à un événement

# paramètres	# événements		
	10k	100k	1000k
1	1.08	4.22	33.45
5	1.07	4.15	34.18
10	1.09	4.17	33.89

(a) Temps de chargement (s), avec catégories

# paramètres	# événements		
	10k	100k	1000k
1	0.80	8.00	80.00
5	0.80	8.00	80.00
10	0.80	8.00	80.00

(b) Place mémoire (Mo), avec catégories

# paramètres	# événements		
	10k	100k	1000k
1	3.50	19.77	194.03
5	4.53	32.15	336.83
10	5.84	47.21	510.35

(c) Temps de chargement (s), sans catégories

# paramètres	# événements		
	10k	100k	1000k
1	2.36	23.60	236.00
5	4.12	41.20	412.00
10	6.32	63.20	632.00

(d) Place mémoire (Mo), sans catégories

FIGURE 11 – Temps de chargement et place mémoire

et donc sans changer tous les attributs des événements. Accéder moins de données se traduit automatiquement par un plus petit temps de chargement et moins d'occupation mémoire.

Afin de montrer ce gain, nous mesurons dans FrameSoC le temps de chargement des données de tous les événements. Le temps de chargement correspond à la requête SQL, suivie de l'instanciation des classes Java pour contenir les données. Nous utilisons les mêmes traces que dans la section précédente. Les mesures sont faites à l'aide de l'outil JVisualVM [2] et sont présentées dans la Figure 11.

On observe que pour les traces catégorisées, le temps de chargement (Table 11.a), ainsi que la place en mémoire (Table 11.b) dépendent uniquement du nombre d'événements dans la trace. Ceci est cohérent avec le fait que nous n'avons pas chargé les paramètres associés à ces événements. En revanche, sans les catégories, afin d'accéder aux mêmes informations, il est nécessaire de charger les paramètres additionnels des événements. Le temps de chargement (Table 11.c), ainsi que la place en mémoire (Table 11.d) dépendent du nombre d'événements, ainsi que du nombre de paramètres.

6. Discussion

La trace catégorisée est une représentation de plus haut niveau d'une trace brute où sont incluses des informations sémantiques liés à la logique métier. Il faut donc à un moment traiter la trace brute afin de l'exprimer sous forme d'événements à catégories. Cela doit être fait avec

la connaissance du format et de la sémantique des événements contenus dans la trace initiale. Dans le contexte de SoC-TRACE, ceci peut être fait à l'importation initiale de la trace ou alors ultérieurement par un outil spécialisé. La question qui se pose est quelle partie de la catégorisation pourrait être automatisée, éventuellement avec des indications métier.

Une autre question intéressante est d'évaluer la complétude et l'utilité des quatre catégories présentées. En effet, nous nous sommes inspirés de notions manipulées dans le domaine des applications parallèles. Dans l'étude présentée, nous avons utilisé avec succès les notions d'événement ponctuel, d'état et de lien. Néanmoins, nous manquons encore d'expérience et de vision pour savoir si ces quatre catégories suffisent et si elles sont pareillement utiles dans le domaine des systèmes embarqués.

Dans notre scénario d'étude, l'analyse statistique a été effectuée manuellement. La question que nous nous posons est de définir un ensemble de traitements statistiques que l'on pourrait appliquer de manière automatique aux traces de systèmes embarqués afin de mettre en corrélation les différents événements. Quelques traitements statistiques simples qui nous semblent intéressants sont : le calcul des moyennes de durées par type d'événements pour la catégorie *état*, le calcul de fréquence pour les *événements ponctuels* le calcul des durées pour les événements de catégorie *lien*, l'analyse de l'évolution des variable dans le temps (linéaire, exponentielle,...).

7. Conclusion

Dans cet article nous nous sommes intéressés à la question d'analyse de traces d'exécution afin d'identifier des problèmes de fonctionnement ou de performances dans les systèmes embarqués. Dans le contexte du projet SoC-TRACE, nous avons proposé une évolution du modèle de données représentant la trace afin d'y introduire la notion de catégorie. Les catégories structurent et rajoutent de la sémantique au niveau de la trace, tout en gardant la séparation entre informations génériques et informations métier. Nous avons validé notre proposition en utilisant un scénario réel de décodage vidéo fourni par STMicroelectronics.

Nous avons montré que la classification des événements par catégories à plusieurs avantages. Elle permet de compresser la trace, de structurer la trace en une trace de plus haut niveau et d'utiliser des traitements statistiques simples afin de mettre en évidence des corrélations entre différents types d'événements.

L'utilisation de catégories pour l'analyse de traces dans le domaine embarqué pose encore plusieurs questions. Tout d'abord, il n'est pas clair si d'autres catégories, en dehors des quatre définies, ne seraient pas également intéressantes et utiles pour l'analyse de traces. Deuxièmement, les stratégies de catégorisation pourraient être manuelles, automatiques ou semi-automatiques. Une trace pourrait être catégorisée de différentes manières et donner lieu à différentes analyses et interprétations. Enfin, le plus grand défi est d'arriver à définir des traitements d'analyse automatique qui aiguillent le développeur dans la recherche de problèmes. Dans ce sens, il serait intéressant de voir comment la catégorisation se marie avec les autres types d'analyses, notamment la fouille de données ou l'analyse probabiliste, investigués au sein du projet SoC-TRACE.

Bibliographie

1. Common Trace Format (CTF). `http://www.efficios.com/fr/ctf`.
2. Java VisualVM. `http://visualvm.java.net`.
3. KPtrace. `http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html`.

4. Projet SoC-Trace. `http://tinyurl.com/minalogic-soc-trace/`.
5. R. `http://www.r-project.org`.
6. Spiker. `http://www.lynuxworks.com/products/spyker/spyker.php3`.
7. SQLite. `http://www.sqlite.org`.
8. Windriver. `http://www.windriver.com/products/OCD/wind_river_trace/`.
9. An Mey (D.), Biersdorff (S.), Bischof (C.), Diethelm (K.), Eschweiler (D.), Gerndt (M.), Knüpfer (A.), Lorenz (D.), Malony (A. D.), Nagel (W. E.), Oleynik (Y.), Rössel (C.), Saviankou (P.), Schmidl (D.), Shende (S. S.), Wagner (M.), Wesarg (B.) et Wolf (F.). – Score-P – A Unified Performance Measurement System for Petascale Applications. – In *Proc. of the CiHPC : Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, number June 2010, pp. 85–97. Springer, 2012.
10. Dagum (L.) et Menon (R.). – OpenMP : An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, vol. 5, 1998, pp. 46–55.
11. Desnoyers (M.) et Dagenais (M. R.). – The LTTng tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux. – In *Ottawa Linux Symposium*, 2006.
12. Gabriel (E.), Fagg (G. E.), Bosilca (G.), Angskun (T.), Dongarra (J. J.), Squyres (J. M.), Sahay (V.), Kambadur (P.), Barrett (B.), Lumsdaine (A.), Castain (R. H.), Daniel (D. J.), Graham (R. L.) et Woodall (T. S.). – Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. – In *11th European PVM/MPI Users' Group Meeting*, pp. 97–104, 2004.
13. Geimer (M.), Wolf (F.), Wylie (B. J. N.), Abraham (E.), Becker (D.) et Mohr (B.). – The Scalasca Performance Toolset Architecture. – In *In International Workshop on Scalable Tools for High-End Computing (STHEC*, number 01, pp. 702–719, 2008.
14. Kergommeaux (J. C. D.) et Stein (B. D. O.). – Pajé : An Extensible Environment for Visualizing Multi-threaded Programs Executions. – In Bode, Arndt and 0002, Thomas Ludwig and Karl, Wolfgang and Wismüller (R.) (édité par), *Euro-Par*, pp. 133–140. Springer, 2000.
15. Knüpfer (A.), Brendel (R.), Brunst (H.), Mix (H.) et Nagel (W. E.). – Introducing the Open Trace Format (OTF). – In *International Conference on Computational Science (2)*, 2006.
16. Kraft (J.), Wall (A.) et Kienle (H.). – Trace Recording for Embedded Systems : Lessons Learned from Five Industrial Projects. – In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*, 2010.
17. Nagel (W. E.), Arnold (A.) et Weber (M.). – VAMPIR : Visualization and Analysis of MPI Resources. *Supercomputer*, vol. 12, 1996, pp. 69–80.
18. Pagano (G.), Dosimont (D.), Huard (G.), Marangozova-Martin (V.) et Vincent (J.-M.). – Trace Management and Analysis for Embedded Systems. *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, septembre 2013, pp. 119–122.
19. Pagano (G.) et Marangozova-Martin (V.). – *SoC-Trace Infrastructure*. – Rapport technique n July, 2012.
20. Prada-Rojas (C.), Riss (F.), Raynaud (X.), Paoli (S. D.) et Santana (M.). – Observation Tools for Debugging and Performance Analysis of Embedded Linux Applications. – In *Conference on System Software, SoC and Silicon Debug 2009*, 2009.
21. Shende (S. S.) et Malony (A. D.). – The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, vol. 20, n2, mai 2006, pp. 287–311.
22. Spear (A.), Levy (M.) et Desnoyers (M.). – Using Tracing to Solve the Multicore Problem. *Computer*, 2012, pp. 60–64.
23. Stein (B. D. O.). – *Visualisation Interactive et Extensible de Programmes Parallèles à Base de Processus Légers*. – Thèse de PhD, 1999.
24. Toupin (D.). – Using Tracing to Diagnose or Monitor Systems. *IEEE Explore*, 2011, pp. 87–91.