Débogage des systèmes embarqués multiprocesseur basé sur la ré-exécution déterministe et partielle

Kiril Georgiev

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Kiril Georgiev

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Vania Marangozova-Martin et Miguel Santana**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Débogage des systèmes embarqués multiprocesseur basé sur la ré-exécution déterministe et partielle

Thèse soutenue publiquement le **04 décembre 2012**,
devant le jury composé de :

Mr, Philippe Navaux
Professeur à l'Universidade Federal do Rio Grande do Sul, Président
Mr, Pierre Sens
Professeur à l'Université Pierre et Marie CURIE, Rapporteur
Mr, Smail Niar
Professeur à l'Université de Valenciennes et du Hainaut Cambrésis, Rapporteur
Mr, Jean-François Méhaut
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mme, Vania Marangozova-Martin
Maître de conférences à l'Université Joseph Fourier, Co-Directrice de thèse
Mr, Miguel Santana
Directeur du centre IDTEC à STMicroelectronics, Co-Directeur de thèse

Remerciements

Je tiens avant tout à remercier mon co-directeur de thèse Monsieur Miguel Santana et la société STMicroelectronics pour m'avoir proposé cette thèse dans les meilleures conditions possibles. Miguel a toujours été disponible pour superviser les avancements dans mes travaux, me donner des conseils pertinents dans mes propositions, m'aider dans la recherche d'information technique et me donner l'opportunité de me familiariser avec la recherche industrielle. Il m'a également poussé vers une direction dans les avancements de la thèse qui a permis de la mener jusqu'à son bout.

Je remercie tout particulièrement mon directeur de thèse Professeur Jean-François Méhaut pour la confiance et la liberté qu'il m'a accordée. C'est lui qui m'a initié et qui m'a donné le goût de la recherche scientifique à partir de ma quatrième année d'études universitaires. Son recul et sa vision globale sur mes travaux m'ont toujours motivés et inspirés pour aller de l'avant, explorer de nouvelles pistes et rechercher des solutions. Je le remercie tout particulièrement des nombreuses réunions enrichissantes qui m'ont permis d'éviter les mauvais choix et d'approfondir et de se concentrer sur les bons. Son support, sa disponibilité, son aide et sa patience m'ont aidé à mener ce projet à son terme. Je lui dirai une fois encore merci.

Je remercie ma co-directrice Madame Vania Marangozova-Martin pour ses conseils avisés et son aide. Elle m'a toujours poussé vers la recherche de la rigueur, de la précision, du détail et de la pédagogie. Je le remercie également de m'avoir encouragé dans les moments difficiles et de toutes les discussions agréables et reposantes. Merci de tous les efforts que tu as fourni particulièrement sur la fin de la thèse.

J'adresse tous mes remerciements à Monsieur Smail Niar, Professeur à l'Université de Valenciennes ainsi qu'à Monsieur Pierre Sens, Professeur à

l'Université Pierre et Marie CURIE, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse.

Je remercie toute l'équipe IDTEC et notamment Serge De Paoli ainsi que les équipes MESCAL et MOAIS pour leur disponibilité, leur conciles et leur accueil chaleureux.

Je remercie mes parents et ma sœur qui ont toujours respecté mes choix et m'ont poussé vers le plus haut niveau d'études. Leur soutien continuel m'a aidé à passer au travers de tout obstacle aussi bien au niveau professionnel qu'au niveau personnel.

Je te remercie Nora de m'avoir donné les forces pour surmonter toutes les difficultés. Merci de t'être dévouée dans notre relation, de m'avoir fait confiance et d'être restée à mes côtés sans réserve. Je te remercie d'être comme tu es et je suis heureux de partager le cours de la vie avec toi.

Je remercie Philippe Vauquelin de m'avoir accueilli en France comme un membre de sa famille. Merci de ton aide au début, là où j'en ai eu le plus besoin, je ne l'oublierais jamais. Je te remercie également de m'avoir toujours soutenu dans mes études jusqu'à la thèse.

Je te remercie bien sur Claf pour tous les moments inoubliables qu'on a passé et notamment le bateau et le ski, mais aussi des soirées au labo et en dehors du labo. Merci également de ton soutien, de tous les moments de divertissement trois étoiles qui ne se comptent pas. J'espère qu'on va garder notre amitié et toujours trouver un peu de temps pour accumuler de bons souvenirs.

Je te remercie Carlos pour notre collaboration enrichissante, pour la compatibilité qu'on a eu dans notre travail et pour toutes les conversations autour des sciences, de la vie et aux fins inattendues. La vie qu'on a partagé autour de la thèse m'a apportée des valeurs inestimables.

Je remercie tous les collègues et notamment Ben, Jérome et Augustin qui ont fait de la thèse un moment de la vie que ne s'effacerait jamais.

Je remercie également notre technicien Christian et notre responsable du bâtiment Stellio qui ont maintenu l'environnement de travail dans les meilleurs conditions possibles.

Table des matières

Chapitre 1

Introduction

Les systèmes embarqués utilisés dans le domaine du multimédia et de la télécommunication ont besoin de plus en plus de puissance de calcul. Ils doivent s'adapter à la complexité croissante des décodages vidéo, de la réalité augmentée, de la visualisation 3D, etc. Utilisés dans de multiples dispositifs comme les téléphones portables, les boîtiers décodeurs (set-top box) et les consoles de jeux, ces systèmes ont de fortes contraintes sur le temps du développement logiciel, la consommation énergétique et les délais de mise sur le marché.

Les systèmes embarqués multi-processeur (MPSoC pour Multiprocessor System-on-Chip) ont été proposés afin de répondre aux contraintes annoncées. Ils sont complexes, intégrant une multitude de processeurs, de blocs de mémoire et de périphériques, organisés dans une hiérarchie par un réseau d'interconnexion. Le nombre important de processeurs accroît la puissance, tandis que leur faible fréquence d'horloge permet de réduire la consommation énergétique [FKH+08, BHMC10]. Cependant, l'augmentation du nombre de processeurs et des blocs de mémoire dans la puce représente un défi important pour la conception et la mise au point du logiciel.

Deux objectifs importants mais antagonistes de la conception du logiciel MPSoC sont d'exploiter efficacement l'architecture et de réduire le temps de développement. Afin de satisfaire ces deux objectifs, de nouveaux environnements de programmation parallèle ont été proposés. Ces environnements fournissent des mécanismes de programmation de haut niveau pour les applications qui abstraient la complexité du matériel ou du noyau système. Cependant, les exécutions de ces applications, effectuées sur un nombre important de processeurs en interaction, deviennent plus complexes du point de vue de la mise au point du logiciel.

La mise au point du logiciel MPSoC est principalement basée sur deux techniques : l'analyse de traces d'exécution et le débogage. L'analyse de traces consiste à collecter des informations sur l'exécution. Ces informations sont visualisées après

l'exécution, afin d'identifier des erreurs.

Le débogage permet d'identifier une erreur en utilisant des points d'arrêt qui figent une exécution, mais aussi en poursuivant l'exécution dans un mode pas à pas. À partir de ces points d'arrêt ou à chaque pas de l'exécution, les développeurs analysent l'état du programme, en observant des valeurs de registres et de la mémoire, la pile d'appels de fonctions, etc. Cependant, le débogage MPSoC est confronté à deux problèmes majeurs.

- Les exécutions MPSoC peuvent être non-déterministes, *i.e.* se dérouler d'une manière différente. Par la suite, il n'est pas garanti que toutes les exécutions produiront les mêmes résultats ou les mêmes erreurs. Plus particulièrement, les erreurs qui ne se produiront pas durant la session de débogage ne pourront donc pas être identifiées.

  Le non-déterminisme est principalement dû à la concurrence dans le logiciel. En effet, l'ordre des événements observés comme ceux de synchronisation ou de communication peuvent différer d'une exécution à une autre, ce qui peut donner des résultats différents. Selon l'ordre observé, des erreurs peuvent se produire ou rester cachées.

- Les exécutions peuvent être caractérisées par un nombre important d'entités (tâches, threads, processus) qui interagissent d'une manière complexe. Le nombre d'éléments du logiciel durant l'exécution à analyser afin d'identifier une erreur peut donc devenir très élevé.

  Dans une exécution MPSoC complexe, il peut devenir très difficile, voir impossible de figer toute l'exécution sur un point d'arrêt. Cependant, il est possible de figer des entités d'exécution de manière individuelle. Les développeurs pourraient donc être amenés à analyser de multiples entités correctes avant d'identifier celle qui est erronée, ne permettant plus de satisfaire les contraintes sur le temps du développement logiciel.

1.1 Objectifs de la thèse

Nous voulons une nouvelle approche de mise au point MPSoC qui permettrait d'analyser les erreurs logiciels dans un contexte d'exécution non-déterministe, comprenant un nombre important d'entités qui effectuent des calculs parallèles et distribués. Notre approche traite et apporte des solutions aux problématiques suivantes :

- *Reproduction de l'exécution* : la mise au point MPSoC nécessite de réexécuter le logiciel plusieurs fois, afin d'analyser en détail un même comportement erroné. Il est donc important que notre solution puisse enregistrer ce comportement et le reproduire.

- *Passage à l'échelle* : il doit être possible d'analyser une erreur dans une exécution caractérisée par un nombre de plus en plus important d'entités d'exécution. Dans un tel contexte, notre approche devra permettre au développeur de se focaliser plus rapidement sur les sources potentielles de l'erreur. Une possibilité est de sélectionner et d'effectuer ces analyses sur une partie de l'exécution qui est supposée fautive. Cependant, les critères de sélection doivent s'appliquer aux différents types d'architectures et d'applications.
- *Intrusion* : l'enregistrement d'une exécution du logiciel pour la reproduction induit des modifications sur son comportement, que l'on considère comme étant une intrusion. Par exemple, cette intrusion peut modifier les occurrences des erreurs ou encore créer des retards qui rendent le respect des échéances impossible. Il est donc important de maîtriser l'intrusion afin de minimiser son impact sur le comportement à l'exécution.

1.2 Contexte scientifique

Les travaux de cette thèse ont été développés au sein du centre d'expertise IDTEC de STMicroelectronics et encadrés scientifiquement par l'équipe-projet MESCAL (LIG-INRIA). Le centre d'expertise IDTEC est spécialisé dans la conception d'outils de débogage et d'observation pour la mise au point des systèmes embarqués. L'équipe de recherche MESCAL s'intéresse à la mise en place et au fonctionnement des futurs générations de plate-formes de calcul intensif.

Les tendances montrent une orientation vers des processeurs basse consommation, utilisés dans les systèmes embarqués. Nos travaux de recherche sur la mise au point des applications exécutées sur ces processeurs représentent donc un aspect critique qui permettrait de réduire les délais de mise en place des plate-formes.

1.3 Organisation de la thèse

Le manuscrit de la thèse est organisé en trois parties comme suit :

- *État de l'art* : cette partie introduit le contexte scientifique des travaux dans deux chapitres. Le chapitre 2 présente les architectures matérielles MPSoC, ainsi que les environnements de programmation. Le chapitre 3, présente les approches de mise au point d'une manière générale, puis se focalise sur les méthodes de ré-exécution déterministe et partielle.
- *Contribution* : nous présentons une nouvelle méthodologie de mise au point du logiciel MPSoC sur deux chapitres. Le chapitre 4 décrit les étapes de cette méthodologie qui correspondent à la mise au point du logiciel à partir

d'une exécution de référence jusqu'à l'identification de l'erreur. Nous décrivons deux méthodes que nous avons mises en place : la ré-exécution déterministe et la ré-exécution partielle. Dans le deuxième chapitre de cette partie (chapitre 5), nous présentons le prototype RedSoC qui fournit les outils nécessaires pour appliquer cette méthodologie.

- *Validation* : la dernière partie comprend trois chapitres qui valident notre approche sur deux configurations matérielles et deux études de cas de mise au point d'erreurs non-déterministes. La première étude de cas, présentée dans le chapitre 6, valide notre approche sur une plate-forme réelle MPSoC. Dans le chapitre suivant (chapitre 7), nous validons le passage à l'échelle en organisant les processeurs, la mémoire et les périphériques d'une machine Non-Uniform Memory Access (NUMA) sous forme de plate-forme MPSoC. Le dernier chapitre de cette partie (chapitre 8), évalue notre méthodologie sur trois critères : la mise au point d'erreurs non-déterministes, le passage à l'échelle et la performance.

Première partie

État de l'art

Chapitre 2

Les systèmes embarqués multi-processeur (MPSoC)

Dans ce chapitre, nous introduisons les plates-formes MPSoC. Plus particulièrement, nous nous focalisons sur deux points principaux : les architectures matérielles et le cycle du développement logiciel. Nous étudions les architectures matérielles MPSoC afin de généraliser leurs caractéristiques. Dans ce contexte, nous analysons l'organisation des processeurs, de la mémoire et des périphériques ainsi que les interactions entre ces éléments.

En ce qui concerne le développement logiciel pour systèmes embarqués, nous abordons les environnements de programmation, avant de nous concentrer sur les outils de mise au point. Cette analyse nous permet d'identifier les lacunes des solutions de mise au point actuelles pour leur application sur les MPSoC.

2.1 Architectures matérielles MPSoC

Comme leur nom l'indique, les systèmes embarqués multiprocesseur ont des architectures matérielles basées sur plusieurs processeurs. La tendance est vers une augmentation du nombre de ces processeurs, ce qui résulte en une complexité matérielle de plus en plus importante.

Les premières architectures MPSoC, parues dans les années 2000 [TBS+11], sont basées sur un processeur d'utilisation générale et plusieurs processeurs spécifiques. Le processeur d'utilisation générale est utilisé pour contrôler les autres processeurs. Les processeurs spécifiques ont un jeu d'instructions réduit mais optimisé pour un domaine spécifique. Le processeur généraliste est communément appelé GPP pour General Purpose Processor. Parmi les processeurs spécifiques, appelés accélérateurs, nous pouvons citer les DSP ou Digital Signal Processor spé-

cialisés dans le traitement du signal numérique, les NP ou Network Processor, les ASP ou Application Specific Processor, *etc.*

Comme le montre la figure 2.1, les premières architectures MPSoC utilisent un bus de communication afin de connecter les différents éléments, *i.e* les processeurs, la mémoire, et les périphériques. Ces architectures sont déjà disponibles sur le marché dans des produits comme les téléphones portables, les boîtes multimédia *etc.* Des exemples sont Snowball [SE], STNomadik [DBL03], OMAP 5912 [Ins] et Freescale MC8126 [WWW06].

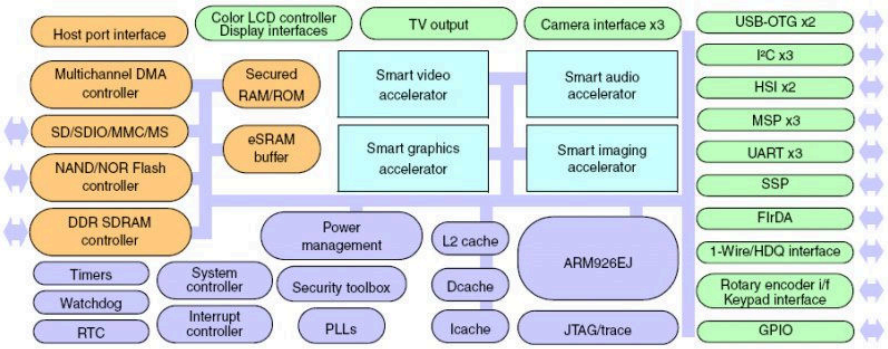


FIGURE 2.1 – Architecture de la plate-forme Nomadik STn8815.

Dans cette section, nous montrons les architectures MPSoC d'une complexité croissante afin de fournir un support de calcul efficace pour les processeurs de nombre de plus en plus élevé. Ces processeurs sont interconnectés et ont accès à la mémoire et aux périphériques par un bus de communication. Le processeur d'utilisation générale est utilisé pour contrôler les accélérateurs. Aujourd'hui, les processeurs généraux intègrent plusieurs cœurs. Un exemple est le ARM Cortex-A9 [ARM]. Les accélérateurs ont un jeu d'instruction réduit mais optimisé pour un domaine d'application spécifique. L'infrastructure de la mémoire comprend des blocs de mémoire partagées et spécifiques aux processeurs. Un ou plusieurs bus de communication permettent l'échange de données entre les processeurs, la mémoire et les périphériques.

Pour permettre le passage à l'échelle au niveau des performances, le bus de communication est remplacé par un réseau d'interconnexion Network-on-chip (NoC). Ce réseau d'interconnexion permet notamment d'éviter la contention en utilisant plusieurs liens entre les processeurs et les blocs de mémoire. En conséquence, on peut utiliser un nombre important de processeurs de faible fréquence et ainsi améliorer la performance, le prix, ainsi que la consommation énergétique de la puce. L'idée principale est de représenter le MPSoC comme un ensemble de blocs appelés Intelectual Property ou IP, qui échangent des données à travers un réseau hiérarchique de routeurs. Les IP peuvent être des circuits comme les processeurs,

les bancs de mémoire, les périphériques, *etc.* L'unique bus de données est donc remplacé par une multitude de routeurs et un réseau de transport efficace.

Dans le domaine de la recherche académique et industrielle, différentes topologies de NoC ont été proposées, selon les différents domaines d'application. La conception de ces topologies est en effet un compromis entre la performance des transferts de données, la difficulté de conception et le coût de production.

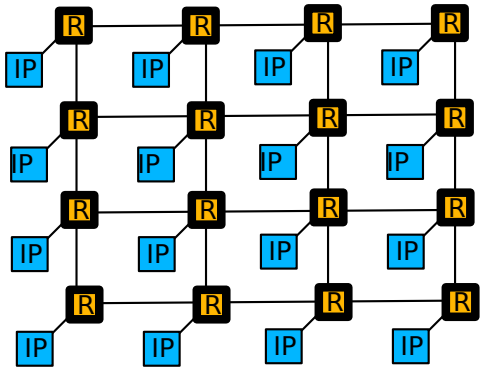


FIGURE 2.2 – Architecture network-on-chip basée sur la topologie grille.

La figure 2.2, représente un exemple de topologie NoC de type grille. Un élément de la grille est composé d'un IP connecté à un routeur. Ces routeurs sont connectés avec quatre routeurs voisins excepté ceux situés sur les bords de la grille. Il est intéressant de remarquer que ce réseau permet l'échange de données simultanées entre plusieurs IP éliminant ainsi le problème de contention lié à l'unique bus de communication. Ces topologies sont étudiées aujourd'hui principalement dans certains travaux de recherche mais il existe des produits qui seront bientôt disponibles sur le marché comme le SPEAr1340 [STMb].

Dans la suite, nous montrons la diversité et l'augmentation de la complexité des MPSoC en présentant quatre architectures. La première, STi7200, est industrialisée et est basée sur des bus de communication. La deuxième, SPEAr1340, est basée sur un NoC, d'un faible nombre de processeurs et bientôt disponible sur le marché. La troisième (PIRATE), est simulée et fait partie d'un projet de recherche académique qui exploite les performances des NoC. La dernière, Plateforme 2012, utilisant un nombre important de processeurs, interconnectés dans un NoC est en cours de développement dans le domaine de la recherche industrielle. Une version réelle sera bientôt disponible.

2.1.1 STi7200

L'architecture STi7200 [MSP07], représentée sur la figure 2.3 est proposée par STMicroelectronics afin d'être intégrée dans les décodeurs multimédia. Cette architecture comprend un processeur d'utilisation générale ST40 et quatre accélérateurs programmable ST231 pour les traitements multimédia. Deux accélérateurs sont programmés pour des traitements audio et deux autres pour les traitements vidéo. Le ST40 accède à un bloc de mémoire DDR2 et échange des données audio et vidéo avec les accélérateurs par des canaux First in, First out (FIFO). Les accélérateurs audio et vidéo ont accès respectivement à quatre blocs de mémoire locale.

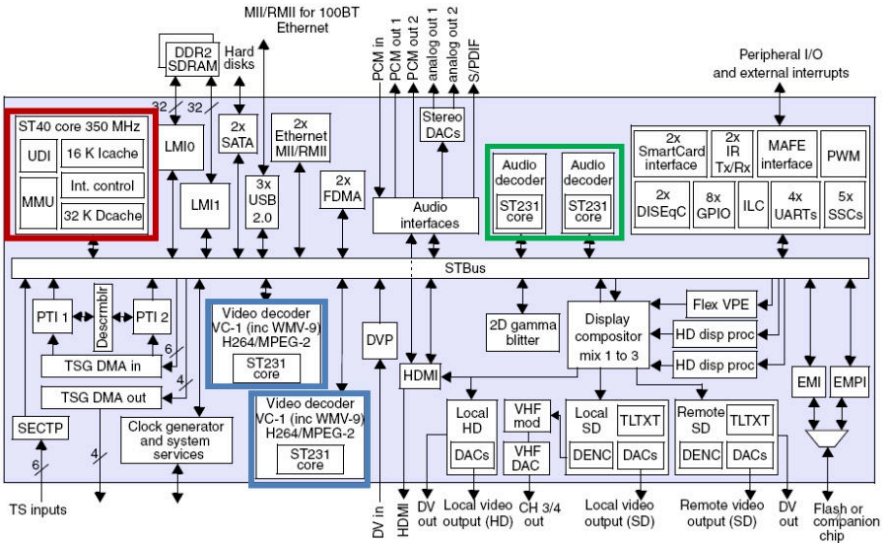


FIGURE 2.3 – Architecture de la plate-forme STi7200.

Cette architecture ne permet pas le passage à l'échelle au niveau des performances. En effet le bus de communication peut causer des latences importantes lorsque le volume de données échangés entre les processeurs, les blocs de mémoire et les périphériques dépasse sa capacité.

2.1.2 SPEAr1340

La plate-forme SPEAr1340 [STMb], illustrée sur la figure 2.4 est également proposée par STMicroelectronics pour une utilisation dans des tablettes, des téléphones portables, des imprimantes industrielles, *etc.* Cette plate-forme est caractérisée par un processeur double cœur ARM Cortex-A9, un bloc de mémoire partagée par les deux cœurs, des accélérateurs avec leur mémoire pour les traitements multimédia ainsi qu'un nombre important de périphériques. L'utilisation de

NoC permet de limiter la contention lorsque plusieurs IP échangent des données. Par exemple, les deux cœurs ARM peuvent accéder simultanément à la mémoire partagée, aux périphériques, ou échanger des données avec les accélérateurs.

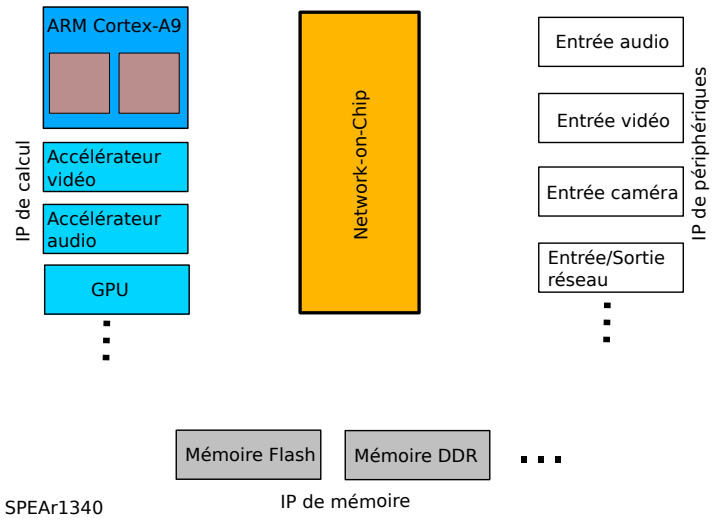


FIGURE 2.4 – Architecture de la plate-forme SPEAr1340.

2.1.3 PIRATE

PIRATE [PS04] est plate-forme basée sur un NoC configurable permettant de simuler différentes topologies. PIRATE est composé d'un ensemble de routeurs représentant la topologie, d'un ensemble de processeurs et des blocs de mémoire connectés aux routeurs par des interfaces réseau (NI). Chaque processeur accède à un espace de mémoire privée et à un espace de mémoire partagée. L'espace de mémoire privée est mappée sur les caches L1 et L2 des processeurs et sur de la mémoire externe. La mémoire partagée est mappée sur des blocs de mémoire interne. Ces blocs de mémoire sont physiquement distribués et logiquement partagés. L'infrastructure de la mémoire partagée est basée sur le paradigme non-uniform memory access (NUMA).

2.1.4 Plateforme 2012

L'architecture embarquée Platform 2012, en cours de conception par STMicroelectronics est composée d'un ensemble de *briques* multipliables pour permettre le passage à l'échelle (figure 2.5). Le nombre de briques est variable afin de per-

mettre une flexibilité au niveau des performances et une adaptation dans plusieurs domaines.

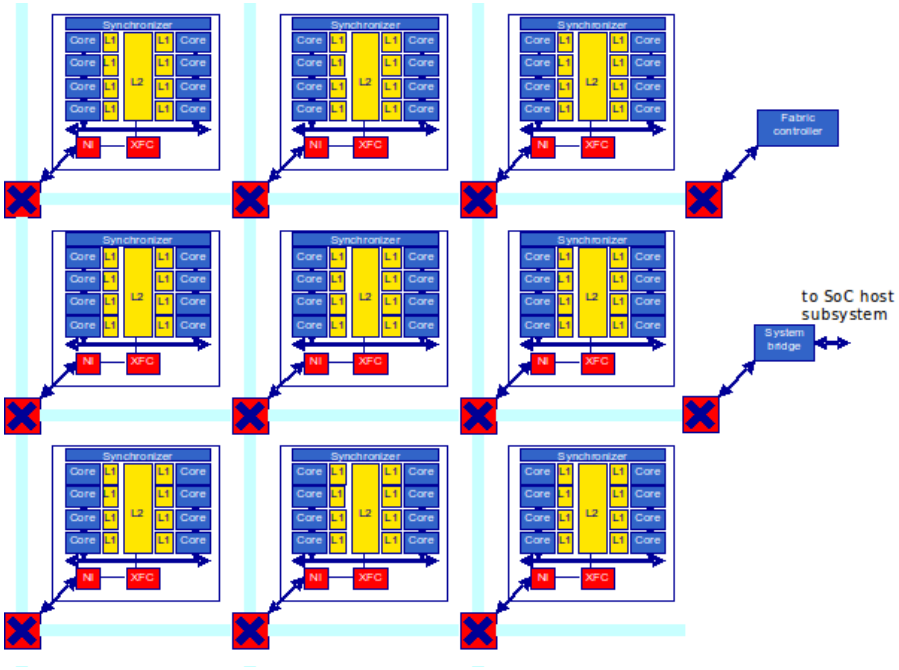


FIGURE 2.5 – Architecture du NoC de la plate-forme P2012.

Chaque *brique* comprend des processeurs, une hiérarchie mémoire et des mécanismes de communication. Un NoC de topologie grille est utilisé pour connecter les briques les unes aux autres et au reste de la puce embarquée en utilisant un pont (bridge) miniature. Chaque processeur a accès à un bloc de mémoire locale (cache L1) et à un bloc de mémoire partagée L2 entre les processeurs dans une *brique*.

2.1.5 Synthèse

Ce chapitre donne un aperçu sommaire des architectures MPSoC, afin d'identifier leurs caractéristiques communes au niveau des composants matériels. Les architectures MPSoC existantes sont principalement conçues pour un ensemble de traitements déterminés, en utilisant des processeurs spécifiques comme les DSP, les audio, les vidéo etc. Dans les futures architectures, les tendances sont d'élargir l'ensemble de leur application en allant vers une utilisation comme des ordinateurs de bureau. Pour effectuer, cette transition, les concepteurs ont proposé d'aller vers des architectures qui intègrent de multiples processeurs génériques afin de fournir des performances élevées et également de pouvoir les adapter au différents trai-

tements. Afin de gérer la contention au niveau de la mémoire, une organisation hiérarchique s'impose. De la mémoire partagée commence à être utilisée au sein des groupes d'éléments de calcul et distribuée entre ces groupes. Les interactions entre les éléments de calcul et les accès à la mémoire se feront par des réseaux, en utilisant de multiples liens de communication. La croissante complexité de l'architecture matérielle posera de nouveaux défis pour le développement logiciel.

2.2 Développement du logiciel MPSoC

La réalisation des systèmes embarqués comme les MPSoC se fait en utilisant des environnements de développement qui intègrent d'un côté, des outils de conception du système et d'un autre côté des outils de développement logiciel.

Les outils de conception ont comme objectif de représenter un systèmes de manière la plus fidèle possible. Au niveau des processeurs, les simulateurs à une précision au niveau des cycles (Cycle-accurate simulator) permettent de représenter le comportement fonctionnel, mais aussi le comportement temporel. La précision est donc proche du processeur réel mais la simulation est très lente. Les simulateurs au niveau du jeu d'instruction (Instruction set simulator) sont plus rapides mais permettent de représenter uniquement le comportement fonctionnel. Des outils comme SystemC [GLMS02] et QEMU [Bel05] permettent de représenter les processeurs mais aussi d'autre composants matérielles comme les périphériques, ainsi que leurs interactions.

Les outils de développement logiciel ont comme objectif de donner les moyens pour la conception du logiciel. Les compilateurs, les assembleurs et les éditeurs de liens permettent de transformer le code source en binaire pour l'architecture ciblée. Dans les MPSoC, le code source peut concerner plusieurs couches logiciel comme le noyau système, les intergiciels ou encore l'applicatif. Des environnements de programmation permettent de concevoir chacune de ces couches en utilisant une interface qui abstraient la complexité de la couche inférieur. La dernière phase de la réalisation du système vise à analyser et à corriger les erreurs en utilisant des outils de mise au point. Dans la suite de cette section, nous nous focalisons sur les environnements de programmation, ainsi que sur les méthodes de mise au point.

2.2.1 Environnements de programmation

Un environnement de programmation fournis un ensemble de fonctions prédéfinis (une bibliothèque) qui peuvent être utilisées telles quelles. Ce sont typiquement des traitements complexes qui sont souvent utilisés par les applications et qui, du coup sont programmés de manière optimisée. Les environnements de programmation définissent souvent des règles de programmation ou adhèrent à un modèle de

programmation.

Deux besoins contradictoires s'opposent lors de la conception de logiciels MPSoC. Un premier besoin est l'exploitation efficace de l'architecture en utilisant de la programmation de bas niveau. Un deuxième besoin est la facilité de programmation qui consiste à utiliser des fonctionnalités de plus haut niveau. Des environnements de programmation font un compromis entre ces deux besoins. D'un côté, il y a les environnements de programmation qui fournissent des paradigmes de programmation de haut niveau comme ceux orientés objets [PPL+04] ou orientés composants [Con]. D'un autre coté, il y a des environnements de programmation basés sur la synchronisation de tâches comme OpenCL [SGS10], CUDA [MV08] et de la communication par messages [TCR+10], privilégiant les performances.

Dans la suite de cette section, nous présentons trois environnements de programmation pour MPSoC. Le premier, TTL [vdWdKH+04] fournit une interface de haut niveau. Le deuxième, Multiflex [PPL+04] et le troisième, MEDEA [TCR+10] fournissent respectivement une interface mixte et de bas niveau. Notre objectif est de comparer les mécanismes mis en place afin d'identifier leurs caractéristiques communes.

TTL

Task Transaction Level (TTL) [vdWdKH+04] est spécialement conçu pour programmer des applications de type multimédia qui sont caractérisées par des échanges de flots de données. L'application est décomposée en plusieurs tâches déployées sur les différents processeurs de l'architecture. Ces tâches peuvent interagir en utilisant un ensemble de fonction afin d'échanger des données.

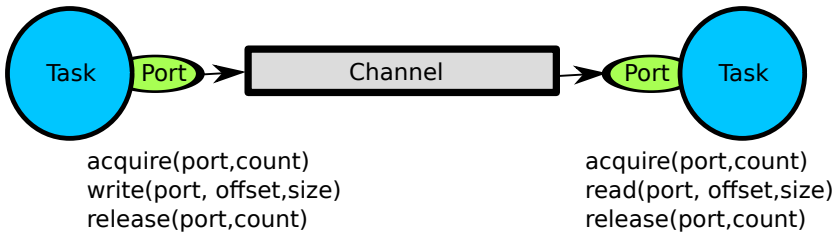


FIGURE 2.6 – Interaction entre les tâches TTL.

La figure 2.6, montre le modèle d'interaction entre les tâches, basé sur un canal de communication, appelé *channel*. Pour être transférée, une donnée doit être écrite sur le *channel* et ensuite lue. Un port est utilisé pour lier les tâches aux *channels*. Le transfert de données est effectué en appelant des fonctions et en spécifiant le port.

Plusieurs fonctions sont disponibles pour effectuer différentes interactions. Certaines sont dédiées à la communication entre les tâches comme *read/write(port, offset, size...)* qui effectuent respectivement une lecture et une écriture d'un vecteur de taille *size* sur le channel qui est connecté au port *port*. D'autres sont dédiées à la synchronisation comme *acquire/release(port, count)* qui respectivement, prennent et libèrent d'une manière bloquante une ressource appelée *jeton* de nombre d'unités *count*. Ce jeton est associé au channel qui est connecté au port *port* .

Il est important de montrer que dans le modèle proposé, la communication ainsi que la synchronisation se font par des objets partagés comme les ports ou encore les jetons. Selon les auteurs, l'ensemble des fonctions proposées est suffisant pour programmer des applications multimédia et peut être implémenté facilement et indépendamment de l'architecture. Cependant, plusieurs fonctions peuvent être utilisées pour programmer une même application. Un choix inapproprié de la fonction ne permettrait pas d'exploiter efficacement l'architecture et dégraderait les performances de l'application.

Multiflex

L'environnement de programmation Multiflex [PPL+04] est dédié aux applications MPSoC de type multimédia et réseau. Cet environnement comprend un modèle de programmation, ainsi qu'une implémentation pour la plate-forme StepNP [PPB02] intégrant des technologies de STMicroelectronics.

Le modèle de programmation est composé de deux parties. La première partie comprend un noyau système embarqué qui permet de lire et d'écrire des données périphériques. De plus, ce noyau intègre un ensemble standard de primitives de synchronisation comme les verrous, les sémaphores, les variables conditions etc. pour la partie à mémoire partagée de l'architecture. La deuxième partie comprend un modèle appelé Distributed System Object Component (DSOC). Ce modèle permet la gestion d'objets et des appels de méthodes sur les objets à distance dont les fonctionnalité sont similaires à celles fournies par CORBA [HV99]. La communication entre les objets se fait donc par des appels distants implémentés par des envois et des réceptions de messages.

L'application est représentée par des objets en communication. Un objet peut soit exécuter du code multi-threadé, soit communiquer avec d'autres objets en appelant des fonctions à distance. Les objets multi-threadés sont ordonnancés par un moteur d'ordonnancement. Ce moteur intègre différentes stratégies d'ordonnancement standard comme *premier arrivé, premier sorti*, *le plus court temps restant*, *tourniquet*, etc.

Multiflex, tout comme TTL, fournit une interface de programmation (API) abstraite pour programmer des applications multimédia sur des architectures MP-

SoC. La mémoire partagée est exploitée en utilisant de la synchronisation et des objets partagés comme TTL. Contrairement à TTL, les transferts de données pour exploiter la mémoire distribuée sont effectués sans utiliser des objets partagés, en spécifiant un objet source et un objet destinataire.

MEDEA

Le projet MEDEA [TCR+10] propose un environnement de programmation pour des applications de calcul matriciel s'exécutant sur une architecture MPSoC basée sur un NoC. L'architecture proposée est composée d'un nombre important de cœurs homogènes et intègre de la mémoire partagée et de la mémoire distribuée.

Les applications sont composées d'un ensemble de flots d'exécution déployés statiquement sur les processeurs. Ces flots d'exécution accèdent à la mémoire partagée par des primitives de synchronisation de bas niveau (lock/unlock). La communication entre les flots d'exécution se fait en utilisant des primitives MPI [GLS99] qui est une interface qui permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages. Les primitives sont notamment *MPI_Send, MPI_Receive et MPI_Barrier* qui permettent respectivement d'envoyer, de recevoir de données et de se synchroniser sur un point de l'exécution effectué par les processeurs.

Comparé aux deux approches précédentes, MEDEA propose un modèle de communication standard, implémenté spécifiquement pour l'architecture. Ce modèle est adapté pour des applications de calcul matriciel, mais ne permettrait pas de satisfaire les besoins des applications multimédia ou réseau. Par exemple, il serait difficile de concevoir des applications multimédia caractérisées par des traitements de flots de données ou avec des contraintes temporelles sans fournir une interface de plus haut niveau.

Comme Multiflex, MEDEA utilise des objets partagés pour synchroniser les accès aux données partagées et des transferts de données en spécifiant les unités d'exécution sources et destinataires. Cependant MEDEA offre un ensemble très limité de configurations pour les méthodes de transferts de données ou des accès aux objets partagés.

Synthèse sur les environnements de programmation

Des environnements de programmation MPSoC standards ne sont pas disponibles, à cause de la diversité des applications MPSoC ainsi que de celle des architectures matérielles. En étudiant des environnements très différents, nous constatons qu'ils proposent des interfaces pour simplifier l'exploitation de la mémoire partagée et la mémoire distribuée par les processeurs. Nous avons plus particuliè-

rement identifié deux caractéristiques communes à ces environnements.

La première est liée à la représentation des applications. Ces applications sont représentés par un ensemble de flots d'exécution (tâches, threads, processus) déployés sur les processeurs ou sur les cœurs. Ces flots d'exécution peuvent interagir pour accélérer le calcul.

La deuxième caractéristique concerne les interactions entre les flots d'exécution. Ces interactions se font par des primitives de synchronisation pour la gestion de la mémoire partagée et des primitives de communication pour les transferts de données.

2.2.2 Techniques de mise au point

Nous considérons la mise au point en tant que processus d'élimination d'erreurs logicielles. Trois méthodes sont principalement utilisées pour la mise au point des applications embarquées : les tests de non-régression [TNP+02, CRV94], l'analyse de traces d'exécution [ESL01] et le débogage.

- *Tests de non-régression* : C'est une méthode qui permet de vérifier l'absence d'erreurs logiciel après une modification. La méthode consiste à générer un ensemble d'exécutions avec un choix de paramètres en entrée bien défi, afin de tester un ensemble de cas d'exécution le plus exhaustif. Une exécution est erronée lorsque la sortie de l'exécution n'est pas conforme à la spécification. Cette technique constate l'erreur mais ne permet pas l'analyse et l'identification de la cause de l'erreur.
- *Analyse de traces d'exécution* : permet principalement d'identifier un comportement anormal durant l'exécution. Cette méthode est utilisée afin d'identifier des problèmes de performances, des états du système incorrects, des états de la mémoire incorrects, *etc.* L'analyse de traces consiste à collecter un historique de l'exécution de l'application. Cet historique est analysé post-mortem ou en ligne pour découvrir des anomalies.

  L'analyse de traces est une méthode complémentaire au test de non-régression, permettant d'avoir plus d'informations sur les séquences d'instructions qui ont donné lieu à l'erreur. Cette technique permet d'analyser l'exécution sur un ensemble prédéterminé d'éléments d'observation. Cette technique peut donc être inefficace lorsque l'erreur ne dépend pas de cet ensemble d'éléments.
- *Débogage* : donne les moyens d'identifier le point d'infection, *i.e* les instructions erronées d'une application. Cette méthode est utilisée afin de remonter jusqu'à l'origine de l'anomalie à un point de l'exécution. Le débogage consiste à interrompre l'exécution de l'application sur une instruction donnée, sur un

état du système ou sur un état de la mémoire. Cette exécution peut être contrôlée en exécutant les instructions une à une ou en groupe. De plus, les valeurs en mémoire ainsi que les registres peuvent être observés à chaque pas de l'exécution.

Le débogage est utilisé lorsque les tests de non-régression et l'analyse de traces ne permettent pas de mettre au point l'application. Cependant, il peut être accéléré en utilisant des analyses de traces pour identifier un point proche de l'anomalie.

Le processus de débogage est lent, nécessitant plusieurs interruptions de l'application, des exécutions pas à pas etc. En conséquence, son utilisation dans la phase de simulation, qui est également lente, reste problématique. Le débogueur doit donc pouvoir s'utiliser sur la puce réelle afin d'analyser les dernières erreurs restantes, après la phase de simulation.

Parmi les trois méthodes, le débogage permet d'analyser de façon la plus détaillée une exécution et donc d'identifier le plus grand ensemble d'erreurs. Dans la suite de cette section, nous étudions de façon plus détaillée la mise au point des applications embarquées en utilisant des débogueurs.

La mise au point des systèmes embarqués est un problème complexe. D'un côté, les applications utilisent d'une manière optimale le système, laissant peu de ressources à dédier aux outils de débogage. En conséquence, les ressources à dédier aux outils de débogage sont limitées ou indisponibles. Aujourd'hui, une solution couramment utilisée est de laisser une partie indispensable du débogueur sur le système embarqué et de déporter l'autre partie du débogueur sur une machine distante appelée *hôte*. Cet *hôte* est connecté au système embarqué par des protocoles spécifiques de communication.

D'un autre côté, le manque de standards de programmation des systèmes embarqués ne permet pas d'avoir des outils logiciels standards de débogage. Or, la conception d'outils spécifiques est coûteuse. Face à ce problème, les concepteurs de circuits ont intégré du matériel dédié au débogage dans les processeurs. Le fonctionnement des processeurs peut donc être interrompu, des instructions peuvent être exécutées pas à pas et des valeurs en mémoire et des registres peuvent être observées. Cette partie, dédiée au débogage dans les processeurs, peut être accédée par un dispositif externe à la puce. Les spécifications de ce type de débogage sont données par la norme Joint Test Action Group (JTAG) [JTA]. JTAG permet notamment d'accéder aux broches dédiées au débogage dans les processeurs embarqués et par suite de contrôler le fonctionnement du processeur. Le contrôle se fait au niveau des instructions processeur et au niveau des valeurs réelles en mémoire. La mise au point du code applicatif est difficile surtout lorsqu'un noyau système gère plusieurs espaces d'adressage (système, utilisateur). Un exemple de cette difficulté est l'identification de l'adresse physique à partir d'une adresse virtuelle.

Des solutions de mise au point des systèmes embarquées propriétaires sont disponibles comme Green Hills [Hil], Wind River [Riva], Lauterbach [Too] mais à cause de la confidentialité, peu de détails sont fournis. Dans la suite de cette section, nous montrons deux solutions pour déboguer des systèmes embarqués qui sont publiquement disponibles, bien documentées et donc génériques.

Serveur GDB

Le Serveur GDB [Gdb] fait partie du débogueur GDB et permet de déboguer les applications sur plates-formes quelconques d'une manière distante. Cette partie s'exécute sur la plate-forme embarquée appelée *cible* et communique avec l'autre partie de GDB, exécutée sur l'hôte. Serveur GDB comprend notamment les fonctionnalités de bas niveau du débogueur comme la lecture, l'écriture des registres, la communication distante avec GDB etc. En effet, pour déboguer une nouvelle plate-forme il suffit d'implémenter serveur GDB, d'où son intérêt pour les systèmes embarqués.

Pour déboguer les applications, serveur GDB s'exécute dans l'espace utilisateur. Il n'a donc pas accès aux détails de l'architecture, accessibles par le noyau système. D'un côté, le noyau système doit pouvoir fournir les informations demandées par serveur GDB. D'un autre côté, serveur GDB doit pouvoir demander au noyau système la modification de valeurs en mémoire, l'interruption de l'exécution de l'application *etc.*

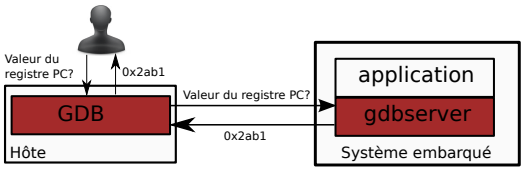


FIGURE 2.7 – Utilisation de GDB pour la lecture de la valeur du registre PC.

Sur la figure 2.7, nous montrons une session de débogage à distance. GDB s'exécute sur une machine hôte et serveur GDB contrôle l'exécution de l'application sur le système embarqué cible. Dans cet exemple, l'exécution de l'application est interrompue. GDB demande la valeur du registre PC et serveur GDB lui renvoie la valeur demandée.

Le débogage avec GDB et serveur GDB est un cycle d'exécutions successives de l'application erronée. Chaque itération permet de mieux analyser l'exécution jusqu'à l'identification de l'erreur. Cependant, il est indispensable d'avoir le même comportement de l'application lors de chaque exécution.

Le débogage conventionnel avec GDB et serveur GDB se fait sur l'intégralité

de l'application et donc sur tous les processeurs et les interactions entre eux simultanément. Lorsque le nombre de processeurs est important comme sur les MPSoC, la recherche de l'erreur peut devenir une tâche insurmontable ou nécessitant un temps qui ne permet pas de respecter les délais de mise sur le marché. Plus particulièrement, la difficulté de la recherche de l'erreur est due à la complexité de l'analyse des multiples contextes des processeurs et des interactions entre eux.

Les limites des méthodes conventionnelles de débogage comme GDB et serveur GDB sont atteintes lorsque le chemin dans le graphe de flot de contrôle caractérisant une exécution n'est pas déterminé. En conséquence, une exécution ne peut plus être répétée pour être mise au point. Ce cas se présente notamment dans les systèmes concurrents où l'ordre des interactions entre les flots d'exécutions n'est pas déterminé. En étant des systèmes concurrents, les MPSoC seront touchés par ce même problème.

KGDB

KGDB [ksld] est un outil qui permet de déboguer le noyau Linux à distance. Cet outil est un code noyau qui permet d'interagir avec GDB, exécuté sur une machine hôte. En effet, KGDB correspond à serveur GDB où l'application à déboguer est le noyau Linux.

Le débogueur du noyau Linux comprend trois parties. Une partie contrôle le noyau Linux suivant les commandes en provenance du GDB sur la machine hôte. Une deuxième partie est un pilote de communication qui permet de transférer les données entre KGDB et GDB. La dernière partie permet de transférer le contrôle au débogueur lors des plantages du noyau.

KGDB est largement utilisé pour déboguer le noyau Linux. Cependant, cette solution permet de déboguer uniquement un noyau particulier ce qui limite son utilisation pour les MPSoC.

2.3 Synthèse

Dans ce chapitre, nous avons d'abord montré les architectures MPSoC et notamment l'organisation des processeurs, de la mémoire et des périphériques. Nous avons ensuite présenté les environnements de programmation, ainsi que les techniques de mise au point pour la conception d'applications MPSoC.

Les architectures MPSoC doivent permettre le passage à l'échelle au niveau des performances et maintenir une consommation énergétique, ainsi qu'un coût de production faible. Ces besoins contradictoires mènent vers la conception d'architectures avec un nombre important de processeurs, une structure hiérarchique

de la mémoire, ainsi que des réseaux de communication pour optimiser le débit d'échange de données entre les composants. L'exploitation d'une telle infrastructure pose de nouveaux défis pour la conception des applications.

Les applications MPSoC deviennent de plus en plus complexes. Elles sont distribuées sur les processeurs multi-cœur de l'architecture et accèdent aux différents blocs de mémoire. Au niveau applicatif, des mécanismes de communication et de synchronisation de haut niveau sont utilisés pour simplifier le développement. Au niveau du noyau système c'est mécanismes sont traduits de manière à exploiter au mieux l'architecture matérielle. Dans tous les cas, les applications MPSoC sont distribuées et parallèles en utilisant des intergiciels et des noyaux système pour exploiter l'architecture.

Les méthodes de mise au point classiques, en utilisant des outils comme GDB ne permettent plus de satisfaire les délais de mise sur le marché des MPSoC. Cette phase serait alourdie, d'une part, par la quantité importante de données à analyser, générées par le nombre important de processeurs. D'autre part, par la difficulté de mise au point des applications concurrentes. Dans le chapitre suivant, nous détaillons ces deux problématiques ainsi que les travaux qui les abordent.

Chapitre 3

Mise au point logicielle des MPSoC

Nous nous intéressons à la mise au point des erreurs logicielles dans les MPSoC. La mise au point de ces erreurs est difficile à cause de deux points majeurs : le non-déterminisme d'exécution et le passage à l'échelle. Le non-déterminisme consiste à avoir des comportements imprévisibles lors de différentes exécutions. Par conséquent, des erreurs potentielles peuvent ne pas se produire à toutes les exécutions. Le passage à l'échelle implique des exécutions sur un nombre important de processeurs aux interactions complexes. Le temps d'identification des erreurs peut donc dépasser les délais de mise sur le marché des MPsSoC.

Dans ce chapitre, nous présentons d'abord le problème du non-déterminisme. Nous faisons ensuite une étude des méthodes de ré-exécution déterministe qui sont la réponse au problème du non-déterminisme. Nous continuons en discutant des approches de ré-exécution partielle qui commencent à apparaître pour répondre au problème de passage à l'échelle. À la fin du chapitre, nous synthétisons les limitations des outils de mise au point existants et définissons les besoins de ces outils pour les MPSoC.

3.1 Les exécutions non-déterministes

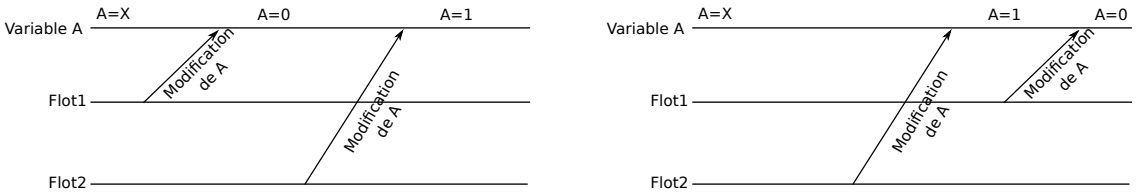
Un logiciel s'exécute de manière déterministe si pour un même ensemble de données en entrée, son exécution se déroule toujours de la même manière. Ceci implique, d'une part, qu'il produit les mêmes sorties. D'autre part, il exécute le même ensemble d'instructions dans le même ordre.

Par exemple, une application séquentielle qui fait l'addition de deux entiers (entrées) est déterministe. En effet, le logiciel exécute les mêmes instructions dans le même ordre. Le résultat ne dépend que des données en entrée. En conséquence, pour les deux mêmes entiers en entrée, le résultat sera le même.

Un logiciel s'exécute de manière non-déterministe si, pour un même ensemble de données en entrée, il peut se dérouler différemment. L'ordre des instructions exécutées peut changer, des ensembles d'instructions différents peuvent être exécutés et même le résultat produit peut varier [RDBC+03].

Le non-déterminisme est dû à la concurrence au sein du logiciel et à ses interactions avec l'environnement d'exécution. Un logiciel est concurrent lorsqu'il génère durant son exécution plusieurs flots d'exécution (enchaînements d'instruction) qui accèdent à une ressource partagée (mémoire, périphérique, canal de communication etc.). L'ordre des accès à cette ressource peut varier d'une exécution à l'autre.

L'exécution concurrente d'un logiciel peut être déterministe ou non-déterministe. Un exemple d'exécutions déterministes sont celles qui accèdent à une ressource mais ne la modifient pas. En conséquence, l'ordre des accès à cette ressource n'influe pas sur le déroulement de l'exécution. Cependant, la modification d'une ressource dans un ordre différent entraîne des exécutions différentes, et en conséquence, du non-déterminisme. Ce cas est présenté sur la figure 3.1.



(a) Première exécution

(b) Seconde exécution

FIGURE 3.1 – Diagramme temporel de deux exécutions d'un logiciel parallèle non-déterministe.

Sur la figure 3.1, nous présentons un exemple de deux exécutions de logiciel parallèle non-déterministe. Ce logiciel donne lieu à deux flots d'exécution qui modifient en concurrence une variable partagée A. La modification est faite de manière atomique *i.e.* pendant que l'un des deux flots travaille avec la variable, l'autre flot n'a pas droit d'y accéder. Les flèches sur la figure montrent l'ordre des accès de la variable dans le temps durant les deux exécutions. Ces deux exécutions produisent un résultat différent (valeur de la variable A) indépendamment des entrées du logiciel. Dans la première exécution, montrée sur la figure 3.1 (a), le processeur 1 écrit sur la variable A la valeur 0 suivi du processeur 2 qui écrit la valeur 1. Dans la deuxième exécution montrée sur la figure 3.1 (b), cette même variable A est mise à 1 d'abord par le processeur 2 et ensuite remise à 0 par le processeur 1. La différence entre les valeurs calculées par le logiciel suite aux deux exécutions est due à un ordre différent d'accès à la variable par les processeurs. En effet, l'état des processeurs, leurs charges, la température extérieur a évolué entre les deux

exécution, ce qui a également modifié leur vitesse d'exécution et en conséquence l'ordre des accès à la variable.

L'environnement d'exécution fournit un support pour le déroulement du logiciel. Cet environnement d'exécution peut être soit l'architecture matérielle, soit une couche logicielle de bas niveau comme le noyau système qui permet d'exploiter cette architecture, soit l'intergiciel qui fournit une interface de plus haut niveau pour accéder au noyau. L'environnement d'exécution influe sur la manière dont le logiciel s'exécute par le fait que c'est lui qui lui fournit les ressources / services nécessaires à son exécution. Il a donc le *pouvoir* de fournir des ressources différentes, dans un ordre différent et à des moments d'exécution du logiciel différents.

Un exemple typique est l'ordonnanceur. Un autre exemple est le déploiement des différents flots d'exécution sur les processeurs disponibles *i.e.* la décision de quel flot s'exécutera sur tel ou tel processeur. Ceci peut accélérer ou ralentir le logiciel et s'il est temps réel, générer des erreurs de non respect d'échéances dans certains configurations de déploiement.

3.2 La difficulté de mise au point

Le logiciel non-déterministe est difficile à mettre au point. En effet, il n'est pas garanti qu'une erreur se produira durant la phase de mise au point. En conséquence, l'utilisation d'un débogueur sera inutile et retardera le processus de mise au point puisque le développeur recherchera une erreur qui ne se produira pas.

Ce problème a été identifié depuis de nombreuses années par la communauté de conception d'outils de mise au point. L'approche classique pour la mise au point du logiciel non-déterministe est la ré-exécution déterministe. Dans la section suivante, nous expliquons le principe de cette approche.

3.3 La ré-exécution déterministe

La ré-exécution déterministe est une approche pour aider à la mise au point du logiciel non-déterministe et non-reproductible. Cette approche permet de reproduire à l'identique l'exécution d'un logiciel. Il s'agit plus particulièrement de reproduire l'exécution erronée. Cette exécution est reproduite jusqu'à l'identification de l'erreur avec un débogueur classique.

La ré-exécution déterministe est une méthode qui se déroule en deux phases. La première phase consiste à enregistrer le comportement non-déterministe lors d'une première exécution que l'on appelle exécution de référence. Il s'agit d'intercepter et d'enregistrer tout ce qui conditionne l'exécution du logiciel. Il s'agit plus

particulièrement des modifications de variables, les chemins pris lors de l'exécution de saut conditionnels, des données d'entrée, *etc.* Ces informations constituent un historique de l'exécution de référence. La deuxième phase consiste à ré-exécuter le logiciel. Cette ré-exécution intercepte et modifie tout ce qui conditionne l'exécution de manière à la reproduire telle qu'elle est enregistrée.

Reprenons l'exemple de l'application non-déterministe 3.1 qui accède en concurrence à une variable partagée. La ré-exécution déterministe permet de reproduire l'ordre des accès à cette variable. Durant l'exécution de référence, chaque accès à la variable est intercepté. L'ordre de ces accès est enregistré dans un historique. Durant la ré-exécution, les accès qui ne respectent pas l'ordre enregistré sont bloqués. Le déblocage est effectué lorsque cet ordre est respecté.

Une manière naïve de reproduire une exécution est d'enregistrer les entrées de chaque instruction lors de l'exécution de référence. Lors de la ré-exécution, il suffirait de fournir les données enregistrées aux instructions et ainsi de garantir une exécution identique à celle de référence. Ceci est en effet impossible du fait des importants volumes de données à enregistrer et de la perturbation de l'exécution due à l'enregistrement des données, appelée effet de sonde [MP96, Thaa] ou encore intrusion. Un des défis dans les approches de ré-exécution déterministe est de maîtriser cette intrusivite. Il s'agit d'approcher le comportement d'une exécution enregistrée le plus possible d'une exécution normale. Dans le cas de la mise au point, cet enregistrement ne doit pas avoir un impact sur les occurrences des erreurs.

Les fonctionnalité d'interception, d'enregistrement et de reproduction d'une exécution ne sont pas fournis dans les systèmes standard. Ces fonctionnalités font partie d'environnements de ré-exécution déterministe que nous étudions dans la section suivante.

3.4 Les approches de ré-exécution déterministe

Nous avons classifié les approches de ré-exécution déterministe selon le type d'architecture matérielle à laquelle elles s'appliquent. Nous avons distingué les solutions pour architecture à mémoire partagée, pour architectures à mémoire distribuée et pour les architectures embarquées mono-processeur. Nous évaluons ces travaux suivant trois critères :

- Type du logiciel : le type du logiciel correspond à son niveau dans la pile logicielle (applicatif, intergiciel et noyau). Le choix du type détermine les mécanismes de ré-exécution déterministe qui doivent être mis en place. Nous nous intéressons aux mécanismes qui peuvent être appliqués dans les MPSoC. Ces mécanismes doivent plus particulièrement être génériques, *i.e.* appliqués

sur un ensemble important de logiciel qu'on peut retrouver sur les MPSoC.

- Facilité d'intégration : notre deuxième critère évalue les mécanismes d'interfaçage entre le logiciel et l'environnement de ré-exécution déterministe. Il s'agit plus particulièrement, des modifications qu'il faut apporter sur le logiciel, afin de pouvoir utiliser l'environnement de ré-exécution déterminisme. Ce modifications peuvent varier de la ré-conception du logiciel à l'ajout d'un simple drapeau de compilation. Une solution facilement intégrable permettrait de réduire le temps de mise sur le marché du MPSoC, d'où notre intérêt pour ce critère.
- Intrusivité : le dernier critère est la perturbation du système lors du traçage durant l'exécution de référence du logiciel. Intuitivement, plus on trace, plus on perturbe. Or, plus on trace, plus on a des infos sur l'exécution et la ré-exécution sera précise.

  Dans le cas des MPSoC, la perturbation est un facteur décisif, surtout dans le cas des systèmes à temps réel puisqu'elle ne permet pas l'accomplissement des tâches dans les délais. Nous devons donc veiller à ce que la perturbation soit maîtrisé, *i.e.* n'a pas d'impact significatif sur les occurrences des erreurs.

3.4.1 Architectures à mémoire partagée

Les approches de ré-exécution déterministe pour les architectures à mémoire partagée concernent les exécutions parallèles, *i.e.* à multiples flots d'exécution. Dans ce cas, le non-déterminisme est dû aux accès concurrents à la mémoire partagée. Il est clair que si tous les accès sont en lecture, les flots concurrents ne se perturbent pas. Par contre, si certains accès sont en écriture, les valeurs des variables partagées dépendent de l'ordre de ces accès. La ré-exécution déterministe consiste donc à reproduire l'ordre de ces accès.

Les approches existantes sont logicielles ou matérielles. Nous n'analysons pas les travaux au niveau du matériel [BG91, HMC+09, XBH03]. En effet, ces travaux nécessitent l'ajout de composants matériels dédiés à la ré-exécution. Le coût de production de la puce serait plus élevé et ne pourrait pas faire face à la concurrence dans le domaine MPSoC. Nous nous concentrons sur les approches logicielles.

Instant Replay

Un des premiers travaux sur la ré-exécution déterministe d'applications exécutées sur des plate-formes à mémoire partagée est *Instant Replay* [LMC87]. Ce travail est basé sur la reproduction de l'ordre des accès aux objets partagés qui sont alloués sur la mémoire partagée. Chaque lecture ou écriture sur l'objet partagé (structure de données allouée sur la mémoire partagée) est interceptée. Lors

de la phase d'enregistrement, un compteur de numéro de version associé à chaque objet en mémoire partagée est incrémenté après chaque accès en écriture. Lors des lectures sur les objets partagés, le numéro de version courant est enregistré dans un fichier de traces. Lors des écritures sur les objets partagés, le numéro de version courant est enregistré, ainsi que le nombre de lectures depuis la dernière écriture.

Durant la phase de ré-exécution, les flots d'exécution sont bloqués avant les lectures sur les objets partagés jusqu'à ce que leur numéro de version atteigne celui enregistré. Lors des écritures sur les objets partagés, les flots d'exécution sont bloqués jusqu'à ce que leur numéro de version atteigne celui enregistré et le nombre de lectures devienne égal à celui enregistré.

L'approche proposée nécessite d'utiliser des primitives d'accès à la mémoire partagée explicitement dans le code. Cette approche n'est donc pas applicable lorsque les instructions accèdent à la mémoire partagée implicitement. De plus, l'unique protocole d'accès à la mémoire partagée supporté est Concurrent-Reader-Exclusive-Writer (CREW), *i.e.* lorsque plusieurs processeurs peuvent effectuer des lectures simultanées à la mémoire mais un seul peut écrire à la fois.

- *Type du logiciel* : *Instant replay* cible la mise au point de la couche applicative et intergicielle exécutées sur le noyau système *Chrystalis OS*, présenté dans le même travail.
- *Facilité d'intégration* : *Instant replay* est une bibliothèque au niveau noyau qui encapsule les primitives pour accéder à la mémoire partagée de *Chrystalis OS* et implémentent les mécanismes de ré-exécution déterministe. Afin d'utiliser *Instant replay*, les développeurs doivent modifier manuellement leurs applications en remplaçant les appels de fonctions *Chrystalis OS* par celles d'*Instant replay*. En conséquence, cette tâche peut devenir difficile lorsque le nombre de primitives à interfacer est important.
- *Intrusivité* : les auteurs ne donnent pas de mesures mais indiquent que l'intrusion est importante lorsque beaucoup de flots d'exécution travaillent sur les mêmes données et utilisent donc des appels de fonctions de synchronisation d'une fréquence élevée.

Netzer

Netzer [Net93] vise une minimisation du volume de données enregistrées. Il utilise des horloges vectorielles [Mat89] pour chaque flot d'exécution et pour chaque objet partagé. Ces horloges sont mises à jour lors de chaque accès à la mémoire partagée et permettent de détecter des accès concurrents. Uniquement ces accès concurrents sont tracés. Netzer supporte tout protocole d'accès à la mémoire partagée.

- *Type du logiciel* : les auteurs décrivent une méthode indépendante du type

du logiciel. Cependant, la validation porte sur la couche applicative d'un ensemble de logiciels.

- *Facilité d'intégration* : Dans ce travail, les auteurs ne décrivent pas l'instrumentation du logiciel afin d'utiliser la ré-exécution déterministe.
- *Intrusivité* : une horloge doit être attachée à tous les objets partagés et comparée lors de chaque accès en mémoire partagée. En conséquence, le surcoût au niveau du temps d'exécution lors du traçage peut être important. Cependant, l'intrusion est moins importante que *Instant Replay*.

Levrow

Pour éviter d'utiliser des horloges vectorielles qui sont coûteuses, dans [LAVC94], les auteurs proposent d'utiliser les horloges scalaires de Lamport [Lam78]. Comme dans l'approche précédente, les valeurs de ces horloges sont sauvegardées lors des accès aux objets partagés. Durant la ré-exécution, les flots d'exécution sont bloqués tant que l'ordre des accès n'est pas conforme à celui enregistré. Cette approche est implémentée dans trois projets notamment RecPlay[RDd99], JaRec [GCRDB04] et Athapascan [RDd99]. Afin de réduire l'intrusion, ces trois travaux se concentrent que sur les accès à des variables partagées où la synchronisation entre les flots est explicite. Cependant, dans RecPlay et dans JaRec, lors de la ré-exécution, un mécanisme coûteux en terme de temps détecte les accès non-synchronisés à la mémoire comme des erreurs.

- *Type du logiciel* : dans les trois projets, la méthode proposée est utilisée pour la mise au point du code applicatif. Pour cela, la couche applicative accède l'environnement de ré-exécution déterministe qui encapsule les primitives de synchronisation natives.
- *Facilité d'intégration* : Dans ces trois travaux, les appels de fonctions de synchronisation non-déterministe doivent être remplacés par ceux de la bibliothèque de ré-exécution déterministe.
- *Intrusivité* : la reproduction de tous les accès est jugée d'une intrusion importante. En conséquence, le traçage, ainsi que la ré-exécution déterministe est faite sur les objets comme les verrous, gérés par des primitives de synchronisation. Les accès non-synchronisés ne sont pas reproduits.

PinPlay

PinPlay [PPS+10] est une solution qui permet de reproduire l'ordre de tous les accès en mémoire partagée en utilisant une machine Just-In-Time (JIT) [SOT+00]. Cette machine intercepte les instructions processeur et ajoute du code avant et

après leur exécution. Lorsqu'un thread exécute une instruction qui accède la mémoire partagée, un compteur d'instructions (IC) spécifique au processeur ainsi qu'un identifiant de thread sont enregistrés. Lors de la ré-exécution, les instructions accédant à la mémoire partagée sont bloquées jusqu'à ce que l'IC atteigne la valeur enregistrée.

- *Type du logiciel* : cette solution permet de mettre au point uniquement le code applicatif Linux et Windows. Cette restriction est due à l'utilisation d'une machine JIT qui interprète les instructions de l'application accédant au système. Cette approche est donc fortement couplée au noyau système et son intégration pour un noyau différent nécessite un ré-conception.
- *Facilité d'intégration* : l'intégration pour Linux et Windows est instantanée, il suffit de fournir l'exécutable du logiciel comme paramètre en entrée de PinPlay.
- *Intrusivité* : le ralentissement de l'application lors du traçage est très important, de l'ordre de 80X-120X. D'une part, la machine JIT applique un traitement à chaque instruction processeur. D'autre part, le traçage de tous les accès à la mémoire partagée génère un gros volume de données à enregistrer.

ODR

ODR [AS09], PRES [PZX+09] et Respec [LWV+10] utilisent une autre approche pour reproduire les accès à la mémoire partagée. L'idée principale est de réduire l'intrusion en enregistrant la sortie de l'application lors d'une exécution de référence. Afin de reproduire l'exécution de référence, il suffit d'identifier l'ordre précis des accès à la mémoire partagée qui produit la sortie enregistrée. Afin d'identifier cet ordre précis, l'application est exécutée en lui imposant un ordre parmi toutes les possibilités. Lorsque la sortie de l'application n'est pas correcte, l'application est ré-exécutée avec un ordre différent. Cette procédure est répétée jusqu'à ce que la sortie de l'exécution corresponde à celle enregistrée. L'ordre imposé durant chaque exécution est enregistré afin d'être reproduit.

Une exécution est considérée comme ré-exécutée d'une manière déterministe lorsqu'un chemin du graphe de flot de contrôle correspond à la sortie enregistrée. Cependant, le chemin identifié n'est pas forcément celui qui a donné lieu à la sortie de l'application.

L'utilisation d'ODR pour la mise au point du logiciel est limitée. L'identification d'un ordre des accès à la mémoire partagée parmi tous nécessite un nombre important d'exécutions. Ces exécutions font l'objet de délais inacceptables avant la phase de mise au point.

- *Type du logiciel* : les auteurs décrivent une méthode indépendante du type

de logiciel. Cette méthode nécessite des mécanismes d'interception des accès à la mémoire partagée entre le logiciel et l'environnement d'exécution. Cependant, ces mécanismes d'interception ne sont pas décrits dans ce travail.

- *Facilité d'intégration* : la facilité d'intégration n'est pas abordée dans ce travail mais elle est très dépendante du choix du noyau système et des couches logicielles à mettre au point, des sources de non-déterminisme, des sources de non-reproductibilité etc.
- *Intrusivité* : les auteurs proposent des optimisations qui permettent de limiter le nombre d'exécutions pour reproduire la sortie de l'application. Cependant, ces optimisations nécessitent d'enregistrer de l'information supplémentaires lors de l'exécution de référence.

Synthèse

D'une manière générale, les méthodes proposées sont basées sur la conception d'une bibliothèque d'encapsulation des primitives d'accès à la mémoire. Tous les travaux numérotent les accès à l'aide d'horloges logiques et ne sont donc pas dépendants des horloges physiques.

Le traçage de tous les accès à la mémoire partagée fait dans PinPlay et Instant Replay fait l'objet d'une perturbation importante des logiciels. Pour réduire cette perturbation, certains travaux ne se concentrent que sur les accès aux variables réellement partagées par les différents flots. Même, ils ne traitent que les accès explicitement synchronisés, *i.e.* reconnaissant le partage des variables. Les accès non-synchronisés sont ignorés ou considérés comme erreurs. Vu que les variables sont explicitement synchronisées, les mécanismes d'enregistrement se focalisent sur les objets utilisés pour cette synchronisation - verrous, conditions, *etc.* Dans ODR, l'intrusion est encore réduite en traçant uniquement le résultat de l'exécution. Cependant, la recherche de l'ordre des accès à la mémoire qui a produit cette sortie nécessite de nombreuses ré-exécutions et n'est pas adaptée au cycle de mise au point.

Les auteurs se focalisent sur la reproduction de la couche applicative et intergiciel. Le noyau est considéré comme stable et standard, typiquement Linux ou Windows.

3.4.2 Architectures à mémoire distribuée

Les travaux sur la ré-exécution déterministe pour les architectures à mémoire distribuée se focalisent sur des échanges de messages qui sont envoyés et reçus en concurrence. Une première source de non-déterminisme peut être considérée lorsqu'un message reçu correspond à plusieurs messages envoyés. La deuxième

source est due aux opérations non-bloquantes de consommation de messages. Ces opérations sont utilisées pour superposer le calcul et la communication. En effet, une consommation non-bloquante se termine indépendamment de la disponibilité des données et donc ne ralentit pas le calcul. Cependant, le volume de données consommées est indéterminé et dépend du nombre de messages envoyés.

MPI et MPV

Dans [NM92], les auteurs présentent une approche qui distingue les messages envoyés et reçus en concurrence de ceux qui ne le sont pas. La communication concurrente est détectée en utilisant des horloges vectorielles, attachées à chaque message. Ces messages sont détectés lorsqu'un message reçu peut correspondre à plusieurs messages envoyés. Dans ce cas, l'ordre des messages reçus est enregistré. Durant la ré-exécution, il suffit de décaler les messages reçus, pour respecter l'ordre enregistré. Cette méthode est implémentée dans MPI [CFMR95] et dans MPV [NBK99].

Dans MPV [NBK99] les auteurs n'abordent pas la ré-exécution déterministe des mécanismes de communication non-bloquants. Cependant, dans MPI [CFMR95] les traces sont enrichies avec le nombre d'opérations non-bloquantes pour permettre leur reproduction.

- *Type du logiciel* : la méthode proposée est indépendante du type de logiciel. Cependant, cette méthode est implémentée dans les deux intergiciels de communication MPV [NBK99] et MPI [CFMR95] permettant de mettre au point la couche applicative.
- *Facilité d'intégration* : dans ces deux travaux, le code applicatif accède aux primitives de communication de l'intergiciel. L'environnement de ré-exécution déterministe encapsule ces primitives et fournit un support pour le traçage et la reproduction d'une exécution.
- *Intrusivité* : dans les trois travaux, l'intrusion est maîtrisée et n'excède pas les 5%.

TCP et UDP

Dans [KSC00], les auteurs présentent une solution pour la ré-exécution déterministe d'applications basées sur des sockets TCP et UDP. Les opérations non-déterministes concernent la connexion entre un client et un serveur et les opérations non-bloquantes de réception de données. En effet, une connexion corresponde à un échange de messages où le serveur reçoit des messages du client. Cette connexion est non-déterministe puisque l'ordre dans lequel les clients se connectent au serveur peut différer entre deux exécutions. La ré-exécution déterministe de la connexion

consiste à tracer un couple qui représente les identifiants du client et du serveur. Ces identifiants sont enregistrés par le serveur. Lors de la ré-exécution, le serveur établit et mémorise les connexions qui ne respectent pas l'ordre de connexions enregistré. Par ailleurs, avant d'établir une connexion, le serveur recherche dans la mémoire des connexions déjà établies avant l'attente d'une connexion avec un client.

La ré-exécution des opérations de réception de données non-bloquantes est faite en enregistrant la taille des données reçues lors de l'exécution de référence. Lors de la ré-exécution, les opérations non-bloquantes se terminent lorsque la taille des données reçues atteint la taille enregistrée.

L'approche proposée est limitée aux protocoles de communications basées sur des sockets TCP et UDP et au logiciel Java. La conception des mécanismes de ré-exécution déterministe dans la machine JVM (Java Virtual Machine) nécessite un effort considérable et nécessite Java Developpment Kit (JDK) [CW96]. De plus, ces mécanismes nécessitent une machine Java avec le code source disponible et modifiable.

- *Type du logiciel* : L'approche proposée est limitée aux applications JAVA qui utilisent les bibliothèques natives basées sur des sockets TCP et UDP. Le mécanisme de ré-exécution déterministe est directement implémenté dans la JVM.
- *Facilité d'intégration* : L'intégration ne nécessite pas la modification du code de l'application. Cependant, il est nécessaire de redémarrer la machine JAVA pour être configuré en en mode traçage, ré-exécution et natif.
- *Intrusivité* : les travaux sur les performances sont menés sur un benchmark spécifique. Ce benchmark permet de spécifier le nombre de threads dans le client et dans le serveur participant au calcul. L'intrusion varie entre 2% pour un thread et 60% pour 32 threads. Cependant, la solution proposée intègre un mécanisme coûteux de ré-exécution des changements de contexte entre les threads. On ne peut donc pas conclure sur l'intrusion due aux opérations de communication.

Liblog

Liblog [GASS06] propose une solution de mise au point pour applications distribuées. Cette approche est spécifiquement proposée pour des environnements mixtes où une partie des processus ne sont pas visibles. Lors de la première exécution de l'application, les processus visibles utilisent un protocole simple pour identifier des communications avec des processus non-visibles. Les processus visibles tracent le contenu des données reçues à partir des processus non-visibles. De plus, les processus visibles tracent l'ordre des communications entre eux en utilisant des horloges

de Lamport. Lors de la ré-exécution déterministe, la communication avec les processus non-visibles est simulée en utilisant les données enregistrées. De plus, l'ordre des communications est forcé pour représenter celui qui est tracé.

La ré-exécution déterministe de la connexion entre deux processus n'est pas abordée. Des connexions en concurrence entre les clients et le serveur ne peuvent donc pas être ré-exécutées d'une manière déterministe.

- *Type du logiciel* : Liblog permet la ré-exécution déterministe du code applicatif et de l'intergiciel basée sur des appels systèmes Linux pour la communication par sockets TCP entre processus.
- *Facilité d'intégration* : l'environnement de ré-exécution déterministe est une bibliothèque dans l'espace d'adressage utilisateur qui encapsule les appels systèmes TCP. Les appels systèmes qui fournissent des données non-reproductibles sont tracées et reproduits. L'intégration de cette solution est aisée pour le système Linux en utilisant les mécanismes existants d'interception d'appels systèmes. Cependant, l'intégration de cette solution peut être une tâche délicate pour d'autres noyaux systèmes.
- *Intrusivité* : l'enregistrement du contenu des données est une opération coûteuse. En conséquence, les auteurs suggèrent d'utiliser leur bibliothèque sur des applications dont la granularité des opérations de communication est faible entre les processus visibles et non-visibles.

Synthèse

La majorité des approches de ré-exécution déterministe pour mémoire distribuée sont basées sur la conception d'une bibliothèque d'encapsulation des primitives de communication. Elles utilisent des horloges logiques pour tracer l'ordre des communications concurrentes. Elles considèrent également la ré-exécution déterministe des opérations non-bloquantes nécessite d'enregistrer la taille des données reçues durant l'exécution de référence.

Comme les approches de ré-exécution déterministe pour les architectures à mémoire partagée, les travaux se focalisent sur la reproduction de la couche applicative et intergicielle. Le noyau système est considéré standard et donc stable, typiquement Linux ou Windows.

3.4.3 Architectures embarquées mono-processeur

Dans cette section, nous considérons les architectures embarquées classiques. Dans ces systèmes, l'environnement d'exécution est le matériel. Le non-déterminisme logiciel est dû aux interruptions. Dans [AL94, MKSR09], les auteurs

montrent qu'il suffit de reproduire uniquement les interruptions de l'horloge et celles des entrées/sorties pour ré-exécuter d'une manière déterministe un système embarqué.

Time Machines

Time Machines [TSHP03], présent une approche qui consiste à reproduire les changements de contexte. Elle consiste à enregistrer une chaîne calculée à partir des valeurs des registres, de la pile et d'une partie de la mémoire. Lors de la ré-exécution, les changements de contexte sont désactivés et l'application est instrumentée avec des points d'arrêt aux instructions susceptibles d'être interrompues. Lorsqu'une telle instruction est atteinte pendant la ré-exécution, la chaîne générée est comparée à celle enregistrée. Un changement de contexte artificiel est effectué lorsque les deux chaînes sont équivalentes. Le principal défaut de cette approche est la possibilité de générer une même chaîne pour deux états différents du système ce qui fait que l'approche n'est pas fiable. Dans ce cas, lors de la ré-exécution, une interruption imprévue sera levée et empêchera la ré-exécution déterministe.

- *Type de logiciel* : cette approche permet de mettre au point les logiciels embarqués à temps réel qui sont caractérisés par un ensemble de tâches exécutées sur un processeur. Les tâches sont ordonnancées par un noyau en utilisant une horloge interne et un mécanisme d'interruptions. Ce mécanisme d'interruption est utilisé pour notifier le noyau de la disponibilité des données reçues par les périphériques. L'environnement de ré-exécution déterministe encapsule les traitants d'interruptions pour isoler le matériel du code système.
- *Facilité d'intégration* : Une difficulté importante d'intégration consiste à fournir une version du noyau hors-ligne. Dans cette version, les interruptions du matériel sont désactivées et peuvent être levées artificiellement. Les auteurs proposent d'utiliser un débogueur spécifiquement conçu pour lever les interruptions lorsqu'il détecte qu'un état du système enregistré correspond à l'état du système après l'exécution d'une instruction.
- *Intrusivité* : les auteurs rapportent une intrusion de 4% sur une seule application industrielle de contrôle de mouvement. Cependant, cette expérience n'est pas suffisante pour conclure sur l'intrusion dans le cas général.

Interrupt Replay

Interrupt Replay [AL94], ainsi que [MKSR09] reproduisent d'une manière similaire les interruptions de l'horloge et celles des entrées/sorties. Durant la phase d'exécution de référence, [AL94] utilise un SIC (Software instruction coun-

ter) [MCL89] et un identifiant d'interruption pour représenter l'état du système lors des interruptions. Ce SIC est incrémenté et tracé lors des entrées et des sorties des traitants d'interruptions. Le SIC correspond à un compteur incrémenté lors de l'exécution d'instructions de sauts conditionnels et les appels de fonctions.Dans [MKSR09], le SIC est remplacé par un PC (Program Counter) associé à une valeur qui indique la différence de temps entre deux interruptions. Dans les deux travaux, l'état du système est tracé lorsqu'une interruption est levée.

Dans [AL94], durant la ré-exécution déterministe, les instructions de sauts conditionnels et les appels de fonctions sont interceptés pour comparer l'état du système enregistré avec celui qui est généré lors de la ré-exécution. Lorsque les deux états sont équivalents, le système est mis en attente jusqu'à l'occurrence de l'interruption enregistrée. Les interruptions enregistrées sont levées dans l'ordre enregistré par un générateur d'interruptions.

Dans [MKSR09], avant la phase de ré-exécution, les instructions susceptibles d'être interrompues sont remplacées par des trappes. Lors de la phase de ré-exécution, ces trappes lancent une procédure de vérification d'émulation d'interruption. Lorsque l'émulation est terminée, les trappes sont remplacées par les instructions supprimées. Les interruptions sont désactivées durant la ré-exécution déterministe.

Dans les deux approches, le moment de reproduction des interruptions n'est pas identique à celui de l'exécution de référence. Dans [AL94], ce moment est identique à un appel de fonction ou à un saut conditionnel près. Dans [MKSR09] le moment est approximé par l'imprécision de l'horloge interne.

- *Type du logiciel* : cette méthode permet de mettre au point les applications embarquées à temps réel qui sont caractérisées par un ensemble de tâches exécutées sur un processeur.
- *Facilité d'intégration* : l'environnement de ré-exécution déterministe encapsule les traitants d'interruptions dans le code du noyau système. Ces traitants d'interruptions sont utilisés pour notifier le noyau de la disponibilité de données des périphériques et pour l'ordonnancement des tâches dû aux interruptions de l'horloge interne. Un logiciel complexe est nécessaire pour générer les interruptions lors de la phase de ré-exécution. Cette complexité est notamment due aux interruptions matérielles utilisées pour la synchronisation entre le processeur et les périphériques/mémoire. Les auteurs ont utilisé des instructions de synchronisation à la place des interruptions. Cependant, cette approche n'est pas possible lorsque le processeur ne supporte pas des instructions de synchronisation.
- *Intrusivité* : dans les deux travaux l'intrusion ne dépasse pas les 5% du temps d'exécution de référence. Cependant, le faible nombre d'expériences dans les deux travaux n'est pas suffisant pour conclure sur l'intrusion dans le cas

général.

Synthèse

Le non-déterminisme dans les SoC est dû aux interactions entre le matériel et le logiciel. Ces interactions s'effectuent par un mécanisme d'interruptions au niveau du matériel et des traitants d'interruptions au niveau du logiciel. L'environnement de ré-exécution déterministe encapsule ces traitants d'interruptions afin d'enregistrer et de reproduire le comportement non-déterministe.

Les solutions de ré-exécution sont spécifiques aux mécanismes des traitements d'interruptions et aux outils de mise au point fournis pour la plate-forme SoC propriétaire.

3.5 Les approches de ré-exécution partielle

Les applications exécutées sur un nombre important de processeurs qu'ils soient distribués, parallèles ou mixtes sont complexes à concevoir, à implémenter et à mettre au point. L'état du système comprend de nombreux éléments et est distribué entre les différentes parties de l'architecture. Les différents parties de l'application qui sont en interaction peuvent échouer individuellement ou en groupe.

Lorsque l'application échoue, l'analyse de l'état global est un défi. Le code fautif ainsi que le code correct s'exécutent en même temps, interagissant d'une manière complexe. L'analyse de ces interactions demande beaucoup de temps et d'efforts.

Un exemple est présenté dans [WLW+10] concernant le serveur Apache HTTP qui est composé d'un nombre important de modules. Ce logiciel communique avec des clients et interagit avec des bases de données. Si lors de l'exécution un module échoue, l'analyse du module en isolation permettrait de réduire la complexité de la mise au point. En effet, si le module est examiné en tout seul, cela éviterait de passer du temps à examiner d'autres parties logicielles qui n'ont pas contribué à l'erreur.

Un mécanisme de ré-exécution partielle pourrait grandement faciliter la mies au point. En effet, s'il est possible de ré-exécuter uniquement une partie du logiciel afin d'analyser le comportement, impliquerait à ne considérer que ce qui se passe dans cette partie, *i.e.* reconsidérer qu'une partie de l'état global. La ré-exécution partielle a un sens uniquement lorsqu'un logiciel est déterministe ou lorsqu'il peut être ré-exécuté d'une manière déterministe.

L'idée dans la ré-exécution partielle déterministe est donc la suivante. Les utilisateurs doivent choisir une partie du logiciel à ré-exécuter en séparation. Le logiciel est ensuite exécuté une première fois. Lors de cette exécution, les données néces-

saires pour la ré-exécution de la partie cible sont enregistrées. Lors d'une deuxième exécution, la partie cible est exécutée en séparation en se servant des données préalablement enregistrées.

Les travaux de ré-exécution partielle pour les MPSoC sont à ce stade inexistants. Cependant, nous avons identifié trois travaux récents sur la ré-exécution partielle dans le domaine du calcul pour la haute performance (HPC). Deux travaux parmi les trois sont similaires nous les avons donc regroupés dans une même section.

3.5.1 Wu

Dans [WLW+10], l'exécution d'un logiciel est représentée par un graphe. Dans ce graphe, il y deux types de nœuds : instructions et valeurs en mémoire. Une instruction accédant à une valeur en mémoire est représentée par un lien entre les deux nœuds respectifs. Un flot d'exécution est représenté par un ensemble d'instructions dans ce graphe. La ré-exécution partielle permet de ré-exécuter d'une manière déterministe une partie des flots d'exécution. Pour ré-exécuter la partie sélectionnée, il suffit de définir une coupure dans le graphe. Cette coupure correspond à la séparation des flots d'exécution sélectionnés des autres. La méthode proposée consiste à enregistrer les valeurs en mémoire nécessaires pour l'exécution en séparation des flots d'exécution sélectionnés.

La ré-exécution partielle a été faite uniquement pour les parties parallèles de l'application. Cette limite est imposée par le choix de ne considérer que les accès à la mémoire partagée.

- *Type de logiciel* : dans ce travail, les logiciels cibles sont ceux composés d'un ensemble de modules comme Apache HTTP Server, Berkely DB etc. Cette approche est donc restreinte au niveau de l'ensemble des applications à mettre au point.

- *Facilité d'intégration* : l'interfaçage entre le code applicatif et l'environnement de ré-exécution partielle se fait au niveau des primitives non-déterministes ainsi que des primitives en interaction entre les modules.

- *Intrusivité* : les expérimentations sont menées sur un ensemble important d'applications. L'intrusion est proportionnelle à la taille des données à enregistrer pour l'exécution partielle. Dans le pire des cas, l'intrusion est très importante, avoisinant les 50%.

3.5.2 Charm++ et MPIWIz

Il existent deux projets sur la ré-exécution partielle d'applications distribuées, basés sur la communication par messages. Dans MPIWIz [XLW+09], l'application est un ensemble de processus Charm++ [KK93], exécutés sur un ensemble de processeurs. La ré-exécution partielle consiste à isoler un sous-ensemble de ces processeurs et est définie en trois pas consécutifs :

- tracer l'ordre des messages entre les processus Charm++.
- sélectionner un ensemble de processeurs et ré-exécuter l'application en respectant l'ordre enregistré. Les messages provenant des processeurs non-sélectionnés sont également enregistrés.
- les processus Charm++ concernant les processeurs sélectionnés lors du pas précédent sont lancés en séparation. Les contenus des messages enregistrés sont utilisés pour remplacer les transferts de messages avec les processus non-sélectionnés.

Dans le deuxième travail [XLW+09], l'application est représentée par un ensemble de groupes MPI, eux même composés de processus MPI. La ré-exécution partielle consiste à isoler un groupe ou plusieurs groupes MPI. Durant une première exécution, l'ordre dans lequel les messages sont échangés est enregistré pour les communications intra-groupe ainsi que le contenu des messages pour les communications inter-groupe. Les développeurs ont la possibilité de sélectionner un ou plusieurs groupe(s) à ré-exécuter. Comme dans l'approche précédente, le contenu des messages enregistrés est utilisé pour remplacer les communications entre les groupes sélectionnés et ceux non sélectionnés. De plus, les échanges des messages intra-groupe sont effectués d'une manière déterministe par rapport à l'exécution de référence. Les entrées applicatives sont reproduites en utilisant le framework R2 [GWT+08]. Pour reproduire ces entrées, l'application doit être instrumentée manuellement avec les fonctions de l'API de R2.

Dans Charm++, le processeur sélectionné est celui qui a échoué lors de l'exécution. Cependant, ce choix sera délicat lorsque l'application se termine mais avec un résultat erroné. Dans MPI [XLW+09], la sélection de la partie de l'application à ré-exécuter est limitée par la composition des groupes MPI. En effet, le développeur n'a pas la possibilité de composer un groupe spécifique de processus qu'il soupçonne fautifs. De plus, les deux approches sont spécifiques à un environnement de programmation précis.

- *Type d'application* : les logiciels ciblés sont le code applicatif utilisant respectivement MPI et Charm++. L'environnement de ré-exécution partielle est une bibliothèque dans l'espace d'adressage utilisateur et encapsule des appels de fonction MPI et Charm++.

- *Facilité d'intégration* : l'intégration consiste à remplacer les appels de fonction de l'environnement de programmation par des appels de fonction de l'environnement de ré-exécution partielle.
- *Intrusivité* : dans MPI [XLW+09], l'intrusion est proportionnelle à la taille des données échangées entre les groupes MPI. Cette intrusion a une valeur maximale de 20% pour l'ensemble des applications testées. Cependant, l'intrusion due au traçage de l'ordre des échanges des messages ne dépasse pas les 2%.

  Dans Charm++ [XLW+09], l'intrusion est maîtrisée et ne dépasse pas les 2%. Ces résultats confirment que le traçage de l'ordre de l'exécution des opérations de communication entraîne une faible intrusion.

3.5.3 Synthèse

Dans ces trois travaux sur la ré-exécution partielle, l'objectif est de réduire le code exécuté pendant la mise au point logicielle. Le premier travail cible des logiciels modulaires où ces modules interagissent par des appels de fonctions. Les deux autres travaux ciblent respectivement des logiciels MPI et Charm++ représentés par un ensemble de processus en interaction. Dans tous les travaux, l'environnement de ré-exécution partielle est une bibliothèque dans l'espace utilisateur interfacée avec le code du logiciel qui enregistre les données intervenant dans les interactions applicatives.

Le premier travail ne supporte pas des logiciels distribués et les deux autres sont spécifiques à l'environnement de programmation. Les trois travaux supposent une connaissance ultérieure sur la partie erronée du logiciel à mettre au point.

3.6 Problématiques de mise au point MPSoC

Dans le premier chapitre nous avons étudié le matériel, les environnements de programmation, ainsi que les outils de mise au point pour les MPSoC.

Le matériel est complexe, caractérisé par un nombre important de processeurs, de bancs de mémoire ainsi que de périphériques, interconnectés par un réseau de communication.

Des environnements de programmation ainsi que des outils de mise au point sont utilisés pour la conception de logiciels SoC et MPSoC. Il existe plusieurs travaux de recherche sur les environnements de programmation pour les MPSoC mais des standards ne sont pas disponibles. Ces standards sont en effet difficiles à établir à cause de la confidentialité et de la concurrence entre les entreprises.

Les outils de mise au point pour les SoC sont basés sur les mécanismes de débogage au niveau du matériel. Ces outils sont indépendants des logiciels et permettent d'inspecter l'état du matériel au cours de l'exécution. Cependant, le lien entre l'état du matériel et l'état du logiciel se complexifie lorsque les logiciels intègrent plusieurs espaces d'adressages (système, utilisateur), des mécanismes de communication et de synchronisation etc., caractérisant les logiciels MPSoC.

L'accroissement du nombre de processeurs dans les MPSoC pose de nouveaux défis pour les outils de mise au point des logiciels. Ces logiciels deviennent non-déterministes dû à la concurrence. La problématique du non-déterminisme est déjà adressée pour des architectures à mémoire partagée, à mémoire distribuée et SoC. Cependant plusieurs difficultés se posent pour la conception d'une solution de ré-exécution déterministe pour les MPSoC. À quel niveau du logiciel positionner l'environnement de ré-exécution déterministe pour les MPSoC ? Les travaux de ré-exécution déterministe pour les architectures à mémoire partagée et à mémoire distribuée ciblent la reproduction du code applicatif et de l'intergiciel. Ceux pour les SoC ciblent le noyau système mais le choix pour les MPSoC est indéterminé.

Quelles sont des sources de non-déterminisme pour les logiciels MPSoC ? Pour les architectures à mémoire partagée, c'est l'ordre des accès à la mémoire partagée. Pour les architectures à mémoire distribué, il s'agit de l'ordre des échanges de messages. Pour les SoC, ce sont les interruptions. Pour les MPSoC, ces sources ne sont pas identifiées.

Comment interfacer l'environnement de ré-exécution déterministe avec les logiciels et avec l'environnement d'exécution ? Les travaux existants ciblent soit des environnements de programmation, soit des noyaux spécifiques. Cependant, ces standards sont aujourd'hui inexistants pour les MPSoC.

Comment maîtriser l'intrusion due au traçage durant l'exécution de référence ? Les logiciels MPSoC pour le multimédia seront caractérisés par des contraintes de temps pour accomplir des tâches. Lorsque les ressources disponibles ne leur permettent pas d'accomplir ces tâches dans les délais, le comportement de l'application est indéterminé. Le processus de traçage lors de la première phase de la ré-exécution déterministe consomme des ressources supplémentaires et de ce fait peut occasionner des dégradations dans les performances non-acceptables. La consommation de ces ressources doit donc être maîtrisée.

Le problème de complexité de mise au point des applications s'exécutant sur un grand nombre de processeurs dû au volume de code exécuté en parallèle est déjà adressé dans le domaine du calcul à haute performance. Des solutions de ré-exécution partielle aident à simplifier ce problème. Cependant, les solutions proposées sont spécifiques aux environnements de programmation HPC standard. Des environnements standard ne sont pas disponibles pour les MPSoC. En conséquence, une nouvelle direction doit être envisagée pour réduire le code exécuté

en parallèle afin d'aider la mise au point. De plus, les approches de ré-exécution partielle nécessitent que la partie du logiciel à mettre au point soit connue.

Deuxième partie

Contribution

Chapitre 4

Proposition de nouvelle méthodologie pour la mise au point du logiciel MPSoC

Dans le chapitre 2, nous avons présenté les architectures MPSoC et avons identifié leurs principaux objectifs : passage à l'échelle des performances, faible consommation énergétique et coût de production minimal. En réponse à ces objectifs, les architectures intègrent de plus en plus de processeurs et des bancs mémoire organisés dans une topologie par des réseaux d'interconnexion. Avec l'évolution des architectures MPSoC, la conception de logiciels embarqués se complexifie et fait émerger le besoin de nouveaux environnements de programmation.

Les environnements de programmation pour les MPSoC essayent d'abstraire la complexité de l'architecture sous adjacente et de fournir une interface de programmation unifiée. Cette interface comprend typiquement des primitives d'ordonnancement de flots d'exécution sur les processeurs, des fonctions de synchronisation, ainsi que des primitives de communication. Les applications basées sur ces interfaces sont concurrentes, composées d'un nombre important de flots d'exécution et s'exécutent de manière non-déterministe.

Dans ce contexte d'exécution, on peut rencontrer des erreurs de type séquentiel, parallèle et distribué. Dans un système séquentiel Clarke et McDermid [CM93] ont identifié cinq types d'erreurs logiciels :

- *De contrôle* : L'exécution prend un chemin, différent de celui attendu.
- *D'adressage* : Affectation d'une valeur incorrecte à une variable.
- *De terminaison* : Terminaison imprévue de l'exécution.
- *D'entrées* : Affectation d'une valeur d'entrée incorrecte à une variable.

Dans les systèmes à temps réel, les erreurs de types violence échéances tempo-

relles sont identifiés dans [Hus02]. Un autre type d'erreurs, concernant la violence de l'ordre d'exécution d'instructions est discuté dans [Thab]. Ces erreurs se produisent lorsque l'ordonnancement des flots d'exécutions ne correspondent pas à celui prévu. Dans les études d'une thèse [Kra00], l'auteur identifie un autre type d'erreurs dans le contexte des systèmes parallèles : blocage infini et inter-blocage. L'inter-blocage est le plus souvent dû à l'attente mutuelle entre deux flots d'exécution pour utiliser une ressource partagée. Netzer [Net91], identifie des erreurs dites d'accès non-synchronisé qui ne sont pas prévues aux ressources partagées. Ces erreurs sont fréquemment des oublis de la part des développeurs d'utiliser des structures de synchronisation sur des ressources partagées. Dans les systèmes distribuées, [PW01] identifient les erreurs qui correspondent aux communications désordonnées. Ce type d'erreurs correspondent à une inconsistance entre un envoi et une réception d'un message.

Les erreurs présentées peuvent être analysés en utilisant un débogueur classique lorsqu'elle sont déterministes et qu'elles n'impliquent pas l'analyse de plusieurs flots d'exécution qui interagissent d'une manière complexe. Nous avons identifié deux problèmes majeurs pour le débogage du logiciel MPSoC. Le premier problème concerne les erreurs non-déterministes qui ne se produisent pas dans toutes les exécutions. Le deuxième problème concerne la difficulté d'analyser une erreur lorsque de multiples flots d'exécution interagissent d'une manière complexe.

Dans le chapitre trois, nous avons étudié les travaux qui traitent les deux problèmes identifiés. Les solutions proposées sont respectivement la ré-exécution déterministe et la ré-exécution partielle. Néanmoins, ces approches n'ont pas encore été appliquées dans le domaine des MPSoC. En effet, en ce qui concerne la ré-exécution déterministe, les sources de non-déterminisme MPSoC ne sont pas encore explicitées et par conséquent il n'existe pas de proposition pour la ré-exécution déterministe. En ce qui concerne la ré-exécution partielle, il n'existe pas de modèle de programmation standardisé. Il n'est donc pas trivial d'appliquer des approches de découpage et d'isolation de parties du logiciel.

Dans la suite, nous présentons notre solution à ces deux problèmes. Nous proposons une nouvelle méthodologie de mise au point pour un large ensemble de plate-formes MPSoC. Cette méthodologie regroupe la ré-exécution déterministe et la ré-exécution partielle.

Dans la suite de ce chapitre, nous définissons d'abord les objectifs de notre proposition. Ensuite, nous présentons une vue générale de notre méthodologie de mise au point. Enfin, nous décrivons d'une manière détaillée les méthodes utilisées.

4.1 Objectifs

Nous voulons une solution de mise au point du logiciel MPSoC qui satisfait les objectifs suivants :

- *Reproduction des erreurs non-déterministes* : notre premier objectif est de pouvoir reproduire les erreurs non-déterministes durant les phases de mise au point du logiciel. Avec une telle reproductibilité, il serait possible d'utiliser un débogueur standard afin d'identifier des erreurs. Par exemple, il serait possible de mettre au point une application de traitement vidéo où certaines exécutions échouent suite à un nombre important d'images perdues. En reproduisant une exécution erronée, la cause de la perte des images pourrait être identifiée dans un cycle de débogage standard.
- *Reproduction d'une partie de l'exécution* : notre deuxième objectif est de circonscrire le code à mettre au point, afin de pouvoir se focaliser sur la partie erronée de l'exécution. L'exécution devrait pouvoir être découpée en parties et le débogage effectué sur une partie supposée fautive. Notre solution doit définir des critères selon lesquels l'exécution est découpée ainsi que la méthode permettant l'exécution partielle. Typiquement, dans le cas d'une application multimédia, il devrait être possible de découper les différents flots audio et vidéo. Lorsque cette application échoue à cause des traitements concernant un flot vidéo, le débogage devrait se concentrer sur ce flot fautif. Le traitement sur un flot vidéo devrait également pouvoir être découpé en parties selon les images.
- *Efficacité* : l'efficacité représente le compromis entre la complétude et la performance de la ré-exécution déterministe. La complétude correspond à la précision de la ré-exécution déterministe. La performance est meilleure lorsque l'intrusion durant la collecte des données est faible. Ces deux objectifs sont contradictoires. D'une part, la complétude nécessite l'enregistrement de gros volumes de données pour reproduire tous les éléments non-déterministes. D'autre part, l'enregistrement d'un gros volume de données perturbe l'exécution d'une manière significative. Dans certains cas, cette perturbation peut rendre la solution de ré-exécution déterministe inexploitable. Par exemple, dans un logiciel avec des contraintes temporelles, le respect des échéances durant les exécutions peut échouer à cause de la perturbation élevée de collecte de traces. Nous privilégions donc la performance par rapport à la complétude. Il s'agit plus particulièrement de ne reproduire qu'une partie des sources de non-déterminisme afin de maîtriser l'intrusion.

Afin de remplir ces objectifs, nous proposons une méthodologie de mise au point du logiciel MPSoC. Cette méthodologie a donné lieu aux contributions suivantes :

- Nouvelle méthode de ré-exécution partielle.

- Définition de deux critères génériques pour le découpage et la ré-exécution partielle d'un logiciel MPSoC.
- Sélection de sources de non-déterminisme MPSoC et choix de méthodes pour leur reproduction.

Dans la section suivante, nous présentons une vue générale de notre méthodologie et positionnons les contributions apportées.

4.2 Vue générale de la proposition

L'idée principale de la méthodologie que nous proposons est de permettre la mise au point de différentes parties de l'exécution. La mise au point de chaque partie poursuit la propagation de l'erreur jusqu'à son identification. Nous avons défini deux dimensions pour découper l'exécution : un sous-ensemble de groupes de processeurs parmi ceux participant dans l'exécution du logiciel et une fenêtre temporelle parmi le temps d'exécution du logiciel.

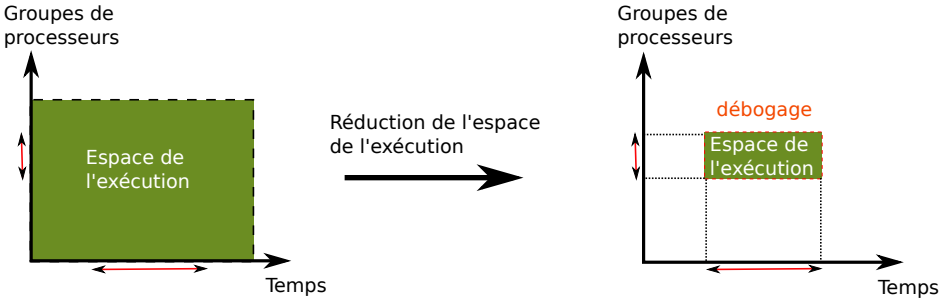


FIGURE 4.1 – Réduction de l'espace d'exécution pour le débogage du logiciel MPSoC.

Sur la figure 4.1, nous montrons la sélection ainsi que la mise au point d'une partie d'une exécution erronée. L'espace de l'exécution qui est montré par les carrés verts correspond au déroulement de l'exécution dans le temps, par les groupes de processeurs de l'architecture. Sur l'abscisse, nous représentons la durée de l'exécution. Sur l'ordonnée, l'ensemble de groupes de processeurs. Suite à cette exécution, les développeurs supposent que le problème se situe dans la partie de l'exécution qui est déployée sur une partie des processeurs, ainsi que durant un intervalle de temps. La figure 4.1(b) montre la mise au point de la partie problématique de l'exécution. Le débogage est effectué sur la partie sélectionnée de l'exécution. Les éléments de l'exécution sont réduits, ce qui simplifie considérablement son analyse.

La méthodologie de mise au point que nous proposons est un ensemble d'étapes effectuées en séquence, menant vers l'identification d'une erreur d'exécution. Ces

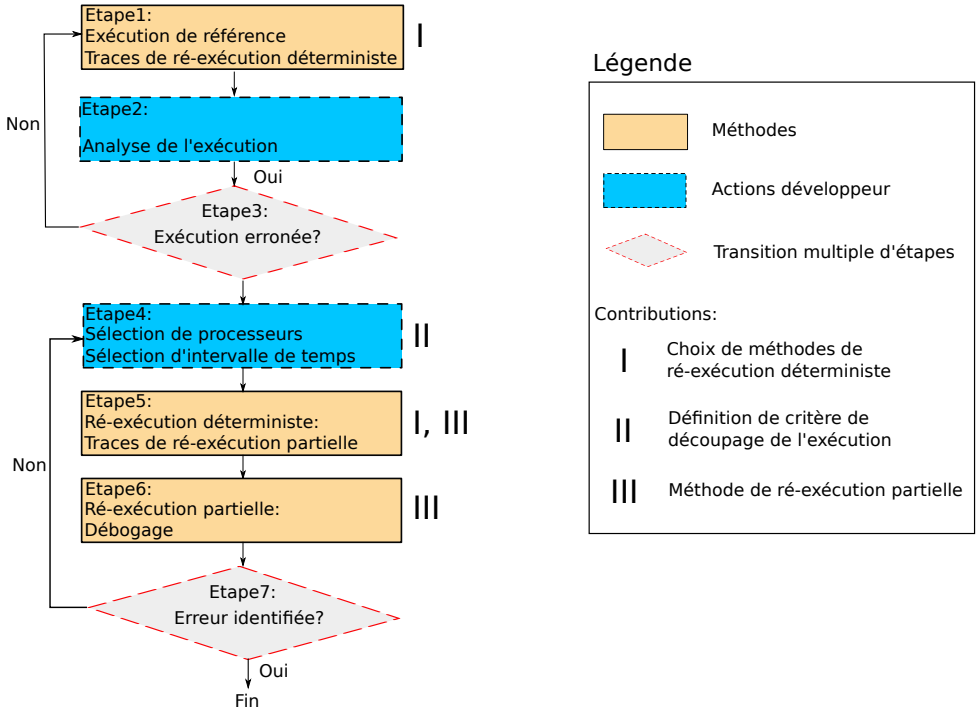


FIGURE 4.2 – Méthodologie de mise au point du logiciel MPSoC.

étapes sont montrées sur la figure 4.2. Cette figure représente pour chaque étape les actions effectuées par les développeurs, les méthodes utilisées, ainsi que nos contributions.

- *Étape 1* : la première étape représente une exécution du logiciel que nous appelons exécution de référence. Durant cette exécution, des traces de ré-exécution déterministe sont collectées. Afin de déterminer le contenu de ces traces, nous avons identifié des sources de non-déterminisme MPSoC, ainsi qu'un ensemble de méthodes de ré-exécution déterministe.

  La proposition de méthodes de ré-exécution déterministe a fait l'objet de notre première contribution présentée dans la section 4.4. Dans ce choix, nous avons rempli l'objectif d'efficacité en ne reproduisons qu'une partie du comportement non-déterministe. Nous maîtrisons ainsi l'intrusion, ce qui signifie que généralement la collecte de données n'empêche pas les occurrences des erreurs.

- *Étape 2* : cette étape consiste à analyser l'exécution de référence afin d'identifier une partie au comportement anormal. Ces analyses sont basées sur les paramètres de sortie de l'exécution et les traces de ré-exécution déterministe. L'analyse d'une exécution pour l'identification d'une partie

fautive repose sur l'expérience du développeur ainsi que sur la connaissance du logiciel et de son comportement. Cette analyse est donc manuelle.

- *Étape 3* : en fonction du résultat de l'étape 2, le développeur a deux possibilités selon qu'il a découvert un comportement anormal ou non. S'il a repéré un problème il continue avec l'étape suivante, sinon il recommence le cycle.

- *Étape 4* : l'objectif de l'étape 4 est de se focaliser sur la partie supposée fautive de l'exécution. Nous avons défini deux critères selon lesquels une partie de l'exécution peut être délimitée : un ensemble de groupes de processeurs et un intervalle de temps. La sélection de la partie supposée fautive est effectuée par les développeurs.

  La définition des deux critères représente notre deuxième contribution qui remplit l'objectif de réduction du nombre d'éléments d'exécution à analyser. Ces éléments sont réduits d'un côté par le nombre de processeurs participants dans l'exécution, et d'un autre côté par sa durée.

- *Étape 5* : dans cette étape, il s'agit de ré-exécuter de manière déterministe l'exécution de référence afin de collecter des traces additionnelles. Ces traces permettent la ré-exécution de la partie sélectionnée lors de l'étape 4. Le mécanisme de collecte de traces fait l'objet de notre troisième contribution, décrite dans la section 4.5.

- *Étape 6* : l'étape 6 consiste à ré-exécuter la partie sélectionnée en utilisant les traces collectées à l'étape précédente. Notre méthode de ré-exécution partielle isole les groupes de processeurs et informe les développeurs lorsque l'intervalle de temps souhaité est atteint. À partir de ce point les développeurs utilisent un débogueur conventionnel pour analyser l'exécution.

- *Étape 7* : lorsque la cause de l'erreur n'est pas identifiée, le cycle est répété à partir de la quatrième étape. Les développeurs ont deux possibilités. La première consiste à sélectionner un nouvel intervalle de temps parmi le même groupe de processeurs. Dans ce cas, la mise au point continue à partir de l'étape 6. La deuxième consiste à faire une nouvelle sélection selon les deux critères et de repartir sur l'étape 5.

4.3 Caractéristiques d'une plate-forme MPSoC

Notre méthodologie doit être applicable pour un ensemble considérable de plates-formes MPSoC. Nous allons donc définir des caractéristiques communs des architectures MPSoC ainsi que des exécutions des logiciels, à partir des plate-formes étudiées (cf. état de l'art, chapitre 2). Nous allons ensuite définir notre méthodologie afin d'être applicable pour toutes les plate-formes ayant ces caractéristiques.

4.3.1 Caractéristiques de l'architecture

Dans le chapitre 2 de l'état de l'art, nous avons montré que les architecture MPSoC intègrent un nombre important de processeurs, de bancs de mémoire et de périphériques. Leur interaction se fait à travers un réseau d'interconnexions. Notamment, nous distinguons les éléments suivants :

- *Groupes de processeurs* : d'un nombre important, les processeurs sont organisés en groupes que nous appelons des nœuds. Dans un nœud les processeurs sont homogènes. Entre les nœuds, les processeurs peuvent être hétérogènes. Par exemple, un nœud peut être spécialisé dans le décodage vidéo alors qu'un autre le sera dans le traitement audio. Les processeurs homogènes de ces nœuds travaillent en parallèle, afin de réduire le temps de traitement audio/vidéo.
- *Mémoire partagée* : dans un nœud, les processeurs interagissent par le biais de mémoire partagée. Tous les processeurs accèdent à un même bloc de mémoire logique partagée.
- *Mémoire distribuée* : la mémoire est distribuée entre les différents nœuds. Cela signifie qu'un processeur d'un nœud ne peut directement accéder à la mémoire d'un autre nœud mais doit passer par le réseau d'interconnexions. Par exemple, un nœud de contrôle devra transférer des données contenant une image par le réseau au nœud respectif du décodage vidéo.
- *Périphériques* : les périphériques permettent d'échanger des données entre les processeurs et l'environnement extérieur. Ils comprennent les caméras, les microphones, les GPS, les claviers, les écrans, les accéléromètres, etc. Les données acquises par ces périphériques sont transmises aux processeurs par la mémoire ou par le réseau.
- *Réseau de communication* : détermine l'organisation hiérarchique de l'architecture en interconnectant les composants matériels (processeurs, mémoire, périphériques).

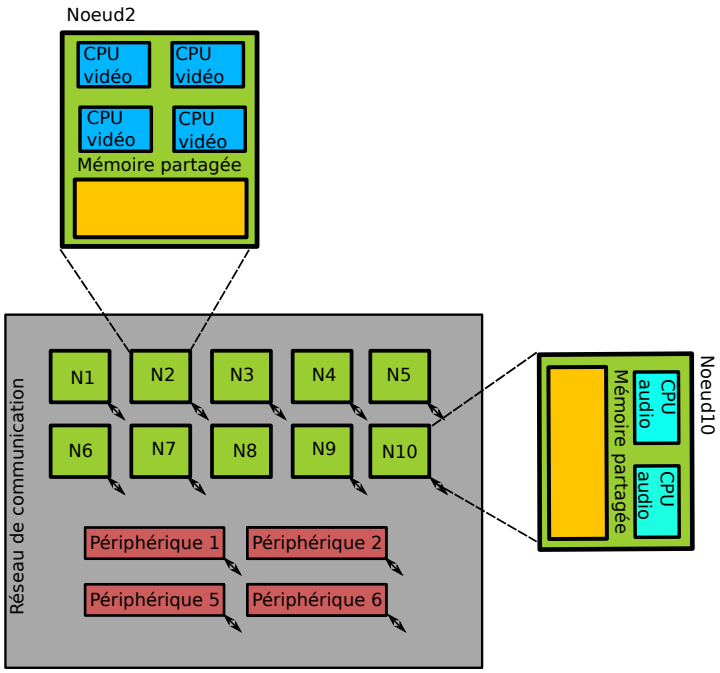


FIGURE 4.3 – Exemple de plate-forme MPSoC.

La figure 4.3 représente un exemple d'architecture qui intègre les éléments proposés. Un réseau interconnecte un ensemble de nœuds à mémoire partagée numérotés de N1 à N10 et des périphériques. Parmi ces nœuds, nous en avons montré deux. Le premier Nœud2, contient 4 processeurs vidéo et un bloc de mémoire partagée. Le deuxième Noeud10 contient deux processeurs audio. Les processeurs de ces nœuds utilisent le réseau d'interconnexion afin d'accéder aux données E/S des périphériques et communiquer avec les processeurs des nœuds distants pour accéder à la mémoire distribuée.

4.3.2 Caractéristiques de l'exécution du logiciel

Nous voyons un logiciel MPSoC en exécution comme un ensemble de flots d'exécution. Ces flots sont ordonnancés sur les nœuds MPSoC. Plus précisément, des sous-ensembles de flots sont ordonnancés au sein d'un nœud. Nous voulons que les flots d'exécution accomplissant des tâches spécifiques nécessitent des nœuds spécifiques. Le logiciel est donc découpé statiquement selon les caractéristiques des nœuds.

Au sein d'un nœud, les flots d'exécution se coordonnent par synchronisation. Ils accèdent à des données partagées (mémoire partagée) à travers des verrous et des sémaphores [KSS96]. Entre les nœuds, les flots d'exécution communiquent par

messages. Cette communication est faite en utilisant des fonctions d'envoi et de réception de données.

Les données des périphériques sont accédées classiquement par un mécanisme d'interruption ou par de l'attente active (polling). L'attente active consiste à interroger un périphérique sur la disponibilité de données. Dans cette attente active, des cycles sont perdus lorsque des données ne sont pas disponibles. Les interruptions permettent d'accéder aux données des périphériques sans perte de cycles, mais nécessitent d'enregistrer et de restaurer l'état d'exécution pour chaque accès.

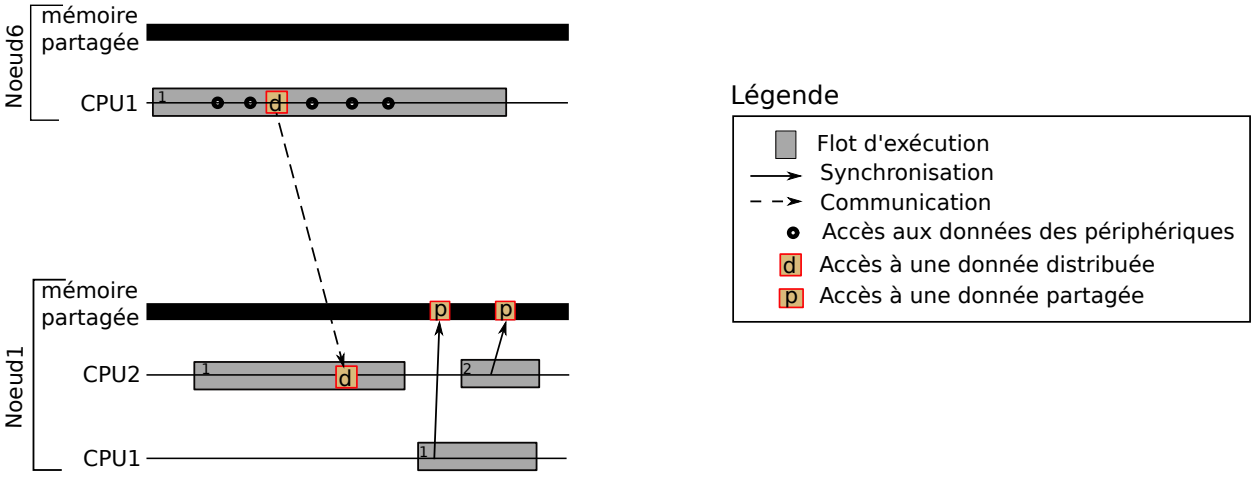


FIGURE 4.4 – Diagramme temporel d'exécution d'un logiciel MPSoC.

Sur la figure 4.4, nous avons représenté un exemple d'exécution d'un logiciel MPSoC. L'exécution est effectuée sur deux nœuds MPSoC : Nœud1 à deux processeurs et Nœud6 à un processeur. Les deux nœuds accèdent respectivement à leur mémoire partagée.

Sur le Nœud1 , deux flots d'exécution sont ordonnancés. Le premier qui est exécuté sur le CPU2, reçoit la donnée d depuis le CPU1 de Nœud6. Cette interaction entre les deux nœuds est montrée par la flèche pointillée. Le flot1 est ensuite ordonnancé sur le CPU1 du même nœud (Nœud1) et accède à la structure partagée p par une opération de synchronisation. Cette opération est montrée par une flèche. Le flot2 est créé et ordonnancé sur le CPU2. Ce flot accède pour une deuxième fois à la structure de synchronisation p.

Sur le Nœud6, le seul flot d'exécution (flot1) est ordonnancé sur le CPU1. Ce flot accède deux fois aux données des périphériques avant d'envoyer la donnée d vers le Nœud1. Finalement, ce même flot accède successivement trois fois aux données des périphériques.

4.4 Ré-exécution déterministe de logiciel MPSoC

Dans cette section, nous étudions les sources de non-déterminisme MPSoC. Chacune de ces sources est analysée et des méthodes de ré-exécution déterministe sont proposées. Dans notre choix, nous sommes guidés par l'objectif d'efficacité. En d'autres termes, nous recherchons le meilleur compromis entre intrusion et complétude. Nous voulons recueillir le maximum d'informations sur l'exécution en perturbant au minimum son déroulement.

Le non-déterminisme dans le logiciel MPSoC vient de ses interactions avec le support d'exécution. Le logiciel accède à ce support en utilisant un environnement de programmation. En analysant les environnements de programmation MPSoC, nous avons identifié quatre sources de non-déterminisme détaillées dans les sections suivantes : accès aux données partagées et synchronisation, communication, entrées/sorties et ordonnancement.

4.4.1 Accès aux données partagées et synchronisation

Les accès aux données partagées par plusieurs flots d'exécution sont sources de non-déterminisme lorsque les deux conditions suivantes sont satisfaites :

1. L'ordre des accès n'est pas spécifié dans le code.
2. Au moins un flot accède aux données en écriture.

En effet, un accès en écriture peut être effectué avant ou après un autre accès en lecture ou en écriture. L'ordre de ces accès est spécifique à une exécution et détermine la valeur de la donnée accédée, d'où le non-déterminisme.

Des exemples sont les verrous, utilisés pour s'assurer qu'une section de code est exécutée par un seul et unique flot d'exécution (celui qui détient le verrou). Les verrous sont utilisés pour synchroniser l'accès aux données partagées. Cependant, cette structure est également partagée. La prise et la libération d'un verrou se traduisent par des écritures sur la structure. La demande de disponibilité se traduit par une lecture. L'ordre des prises des verrous par les flots d'exécution n'est pas explicitement spécifié dans le code. En conséquence, cet ordre dépend de l'ordonnancement, des charges des processeurs lorsque les flots sont exécutés sur des processeurs différents etc. Dans ce cas, les deux conditions sont satisfaites et l'exécution est non-déterministe.

Il est possible de reproduire tous les accès aux structures de données partagées. Cependant, dans [RZ97], les auteurs ont démontré que cette reproduction induirait une intrusion importante sur l'exécution.

Nous préférons considérer, comme dans RecPlay [RDB99], les accès non-synchronisés à la mémoire comme des erreurs et reproduire les accès aux structures de synchronisation. Nous utilisons l'approche proposée dans [LAVC94] et brièvement montrée dans l'état de l'art. Ci-dessous nous donnons une explication plus détaillée.

Collecte de données

Le mécanisme de collecte de données est basé sur des horloges de Lamport pour tracer l'ordre des accès aux structures de synchronisation par les flots d'exécutions.

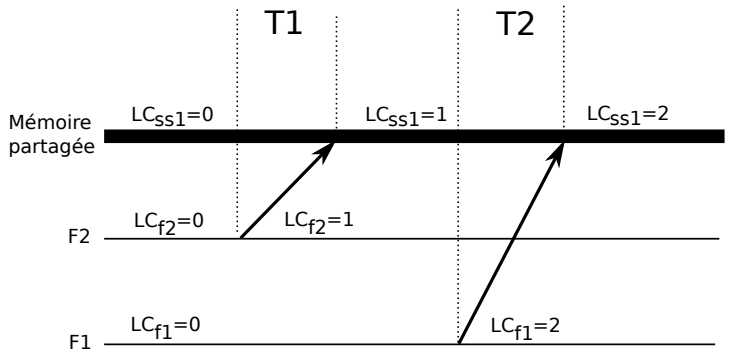


FIGURE 4.5 – Mise à jour des horloges de Lamport.

L'approche utilise des horloges scalaires : une pour chaque structure de synchronisation (LC_{ss}) et une par flot d'exécution (LC_f). Ces horloges avancent lorsqu'un flot d'exécution accède à une structure de synchronisation. L'avancement consiste à mettre à jour les deux horloges respectives LC_{ss} et LC_f en utilisant la règle suivante :

$$LC_f = LC_{ss} = max(LC_f, LC_{ss}) + 1 \tag{4.1}$$

.

Considérons l'exemple présenté sur la figure 4.5 où deux flots d'exécution, $F1$ et $F2$ accèdent à la structure $ss1$. Les horloges respectives sont LC_{f1}, LC_{f2} et LC_{ss1}, initialisées à 0. À l'instant $T1$, $F2$ accède à $ss1$. Les horloges avant et après l'accès sont :

- Avant : LC_{f2}=0 et LC_{ss1}=0.
- Après : LC_{f2}=LC_{ss1}=1.

A l'instant $T2$, $F1$ accède à $ss1$. Les horloges respectives deviennent :

- Avant : LC_{f1}=0 et LC_{ss1}=1.

- Après : LC_{f1}=LC_{ss1}=2.

Lorsque l'horloge d'un flot d'exécution est *enretard* par rapport à l'horloge d'une structure de synchronisation, *i.e* $LC_f < LC_{ss}$, l'ordre des accès à LC_{ss} est non-déterministe. Il s'agit d'une situation où d'autres flots d'exécution ont demandé l'accès à la structure de synchronisation et l'ont obtenue avant. Or, durant une ré-exécution, l'ordre des accès peut changer.

Pour tracer ce cas, un couple contenant l'ancien LC_f et du nouveau LC_f est enregistré. Dans l'exemple proposé, $F2$ enregistre dans sa trace le couple (0,1) pour marquer le premier accès à la structure de synchronisation $ss1$. $F1$ enregistre dans sa trace le couple (0,2) représentant qu'un autre flot à déjà accédé à la structure de synchronisation courante. On peut remarquer que la trace ne contient pas des horloges de structures de synchronisation. Ces horloges sont utilisées uniquement pour détecter des accès non-déterministes.

Reproduction du comportement

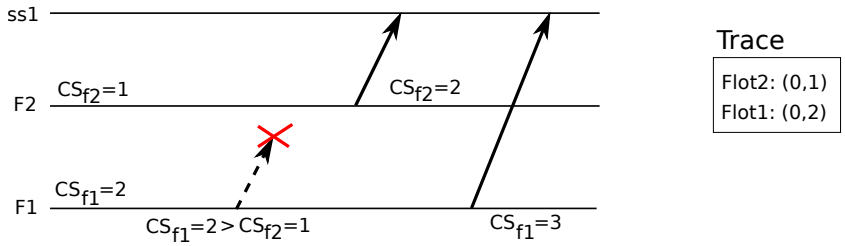


FIGURE 4.6 – Reproduction de l'ordre des accès à une variable de synchronisation.

Afin d'effectuer la ré-exécution déterministe, l'approche proposée utilise un nouveau type de compteurs par flot d'exécution appelé compteur d'incrément (slice counter) CS. Le flot d'exécution ayant la plus faible valeur de ce compteur est autorisé à accéder aux structures de synchronisation.

La mise à jour de ce compteur assure que l'ordre des accès aux structures de synchronisation respecte celui enregistré en utilisant les traces collectées. Cette mise à jour consiste à incrémenter le compteur de 1 lorsqu'un accès est déterministe. Lorsque l'accès est non-déterministe, sa valeur prend la borne supérieure du couple d'horloges de Lamport enregistré. L'accès est non-déterministe lorsque la borne inférieure du couple est égale à CS.

Considérons l'exemple présenté sur la figure 4.6. Dans cet exemple, lors de la ré-exécution, les flots d'exécution essayent d'accéder à la structure de synchronisation $ss1$ dans un ordre différent de celui tracé. Le flot $F1$ essaye d'accéder à la structure

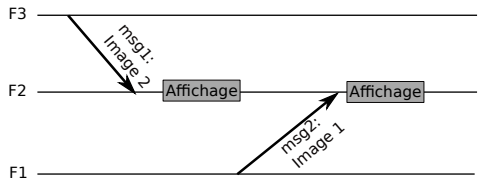
$ss1$ avant le flot $F2$. La méthode présentée bloque le flot $F1$ et attend l'accès par le flot $F2$ avant le déblocage.

Initialement, les compteurs d'incréments des deux flots (CS_{f1} et CS_{f2}) sont incrémentés respectivement de 2 et de 1 (bornes supérieures des couples respectives d'horloges tracées). Le flot $F1$ essaye d'accéder à la structure $ss1$ mais est bloqué puisque la valeur de son compteur n'est pas la plus faible ($CS_{f1}>CS_{f2}$). Lorsque le flot $F2$ accède à $ss1$, son compteur est incrémenté de 1 et devient 2. Le flot $F1$ est alors débloqué et accède à la structure $ss1$ puisqu'il n'existe plus de flot avec une valeur de compteur plus faible.

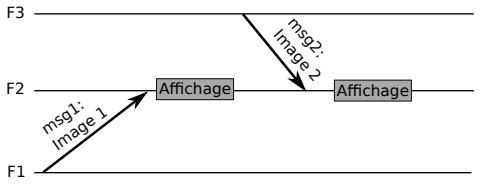
4.4.2 Communication

Une deuxième source de non-déterminisme est la communication par messages, qui peut donner lieu à deux cas. Le premier cas se présente lorsque plusieurs flots d'exécution (sources) envoient des messages vers un même flot (destination). L'ordre de réception de ces messages peut être différent durant deux exécutions lorsque le flot (destination) ne le spécifie pas explicitement. Cet ordre dépend en effet du routage, de la vitesse des liens, de leurs charges etc.

Le deuxième cas de non-déterminisme se produit lors des réceptions non-bloquantes de messages. Il s'agit d'opérations de réception de messages qui se terminent immédiatement, indépendamment de la taille des données disponibles. Les valeurs de retour de ces opérations sont non-déterministes puisqu'elles dépendent des autres flots d'exécution qui envoient les données. Un exemple est la fonction `MPI_Test()` qui fait partie de l'environnement de programmation MPI [GLS99]. Cette fonction retourne $vrai$ lorsque les données à recevoir sont disponibles et $faux$ dans le cas contraire. Durant deux exécutions, le même appel à cette fonction peut retourner une valeur différente selon le nombre d'appels des `MPI_Send()` faits par d'autres flots d'exécution.



(a) Affichage de deux images suite à leur récepiton dans l'ordre suivant: Image2, Image1



(a) Affichage de deux images suite à leur récepiton dans l'ordre suivant: Image1, Image2

FIGURE 4.7 – Exécutions non-déterministes d'un logiciel dues à l'ordre indéterminé de réception de messages.

La figure 4.7 représente deux exécutions différentes à cause de l'ordre indéterminé de réception de messages. Dans la première exécution, montrée sur la

figure 4.7 (a), le flot d'exécution 2 reçoit deux messages, un avec l'image 1 et l'autre avec l'image 2 et les affiche dans l'ordre de la réception. Dans la deuxième exécution, montrée sur la figure 4.7 (b), l'ordre de réception ainsi que d'affichage est inversé.

Parmi les méthodes de ré-exécution déterministe de la communication (chapitre 3), nous avons choisi les moins intrusives. Nous utilisons respectivement Netzer et Miller [NM92] afin de reproduire l'ordre de réception des messages et [CFMR95] pour la réception non-bloquante des messages.

Collecte de données

Nous montrons successivement les méthodes de collecte de données pour reproduire l'ordre de réception de messages et la réception non-bloquante de messages. Nous introduisons d'abord les horloges vectorielles [SK92] nécessaires pour enregistrer l'ordre de réception de messages. Les horloges vectorielles sont utilisées afin de détecter et de tracer uniquement les messages dont l'ordre de réception est indéterminé afin de réduire l'intrusion.

Les horloges vectorielles ont la propriété de refléter la relation de causalité entre les messages envoyés et reçus [CB89]. Cette relation causale capture l'ordre dans lequel les messages sont transférés. Le travail [Fid88] démontre comment utiliser ces horloges afin d'identifier un ordre non-déterministe de transferts de messages. Dans la suite, nous montrons les horloges vectorielles, leur avancement, ainsi que la détection d'ordre non-déterministe de transferts de messages.

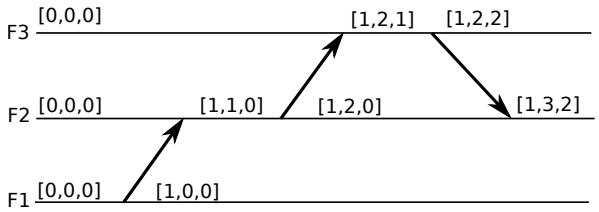


FIGURE 4.8 – Détection de dépendance entre messages.

Chaque flot d'exécution dispose d'une horloge vectorielle. Cette horloge est constituée de n entiers (n est le nombre des flots d'exécution participant dans l'exécution du logiciel). Une horloge date chaque envoi et réception de message en avançant l'horloge. Les messages envoyés sont estampillés en utilisant la valeur courante de l'horloge vectorielle. La réception d'un message permet de synchroniser l'horloge du flot d'exécution courant avec celui qui a envoyé le message.

Pour montrer l'avancement de l'horloge ainsi que la synchronisation, nous utilisons la notation dans [CDK05]. Soit HV_i l'horloge vectorielle du flot d'exécution

i et EV_m l'estampille vectorielle d'un message envoyé m. HV_i est avancée dans les deux cas suivants :

- Lorsqu'un flot d'exécution i envoie un message, $HV_i[i]$ est incrémenté de 1. Ensuite, le message est estampillé de manière à ce que EV_m=HV_i et est envoyé. Par exemple, sur la figure 4.8, lors de l'envoi du message $m1$, [0,0,0] → [1,0,0] ($HV_i[i]++$) et le message envoyé est estampillé avec [1,0,0] (EV_m=HV_i).
- Lorsqu'un flot d'exécution j reçoit un message avec l'estampille EV_m, deux opérations sont effectuées d'une manière atomique. La première consiste à compter les événements locaux en incrémentant $HV_j[j]$ de 1. La deuxième consiste à synchroniser l'horloge reçue avec l'horloge locale en utilisant la règle suivante : ∀ k ≠ j, $HV_j[k]$=max($HV_j[k]$,$EV_m[k]$).

  Par exemple, sur la figure, à la réception du message $m1$, l'horloge du flot 2 est avancée d'abord de [0,0,0] → [0,1,0] ($HV_j[j]++$) et ensuite de [0,1,0] → [1,1,0] (max($HV_j[k]$,$EV_m[k]$)).

Cet avancement des horloges vectorielles leur donne les propriétés suivantes :

- $HV_j[j]$ représente le nombre d'événements sur le flot d'exécution respectif jusqu'à y compris j.
- $HV_j[k]$ ∀ k ≠ j représente le nombre d'événements sur le flot d'exécution respectif qui précèdent causalement j.

La méthode proposée consiste à détecter des réceptions de messages déterministes. En conséquence, tous les autres messages ne sont pas déterministes et sont donc tracés. Deux réceptions de messages sont déterministes lorsqu'entre ces deux réceptions, un message est transféré vers le flot d'exécution qui envoie le deuxième message. Ce transfert de message est effectué toujours après la première réception et avant la deuxième puisque ces trois opérations sont effectuées par le même flot d'exécution. En conséquence, la première réception est toujours effectuée avant la deuxième.

Ces trois opérations définissent l'ordre causal suivant : soit $m_a \rightarrow m_b$ et $m_b \rightarrow m_c$ implique $m_a \rightarrow m_c$. m_a, m_b et m_c sont trois messages et → montre leur ordre de réception.

Un tel cas est présenté sur la figure 4.8 où les messages m_1 et m_3 sont toujours reçus dans le même ordre. En effet, m_2 est envoyé après la réception de m_1, m_3 est envoyé après la réception de m_2 donc m_1 est toujours reçu avant m_3.

Pour détecter la réception déterministe de messages, il suffit de vérifier que l'horloge vectorielle du premier message est propagée jusqu'à l'horloge du troisième message par un deuxième message comme sur l'exemple proposé. Sur cet exemple, on voit que l'horloge de m_1 (HV_1=[1,0,0]) est contenue dans l'horloge de m_3 puisque $HV_3[1] = 1$.

Le traçage des transferts non-déterministes consiste à enregistrer l'identifiant du flot qui a envoyé le message ainsi que la valeur d'un compteur de messages reçus. Ce compteur est utilisé pour distinguer la réception de message déterministe de celle qui ne l'est pas.

Afin de reproduire les communications non-bloquantes, il suffit de tracer la taille des données reçues. Cependant, les réceptions consécutives de messages de la même taille peuvent être regroupées dans une seule trace. Cette trace contient un couple du nombre de réceptions et de la taille des données reçues.

Reproduction du comportement

Pendant la phase de ré-exécution, chaque message doit être envoyé et reçu par le même couple de flots d'exécution que durant l'exécution de référence. D'abord les réceptions de messages déterministes doivent être distinguées de celles qui ne sont pas déterministes. Ensuite pour chaque message non-déterministe, le couple de flots d'exécution (envoi et réception du message) doit être le même que durant l'exécution de référence. Les messages non-déterministes sont détectés lorsque la valeur d'un compteur de réception de messages appartient à la trace. Chaque réception de message non-déterministe est bloquée tant que le message correct n'est pas reçu. Un message est correct lorsqu'il est en provenance du flot d'exécution qui correspond à celui tracé durant l'exécution de référence.

Pour reproduire la réception de messages non-bloquants, la trace est utilisée pour connaître la taille de données à recevoir. La réception est donc bloquée tant que cette taille n'est pas atteinte.

Nous avons utilisé deux approches pour reproduire le non-déterminisme de la communication par messages. Ces deux approches ont été sélectionnées pour leur faible impact sur l'intrusion. La première approche permet de reproduire l'ordre de réception des messages en traçant uniquement les transferts de messages non-déterministes. La deuxième approche permet de reproduire la réception non-bloquante de messages.

4.4.3 Entrées/Sorties

Les environnements de programmation MPSoC fournissent principalement deux mécanismes pour échanger des données entre les processeurs et les périphériques : interruptions et attente active. Dans les deux cas, la disponibilité des données des périphériques dépend d'un contexte extérieur qui n'est, a priori, pas déterministe.

Les interruptions sont levées par les périphériques afin de prévenir le processeur d'un changement d'état. Le processeur interrompt le logiciel en cours et exécute le

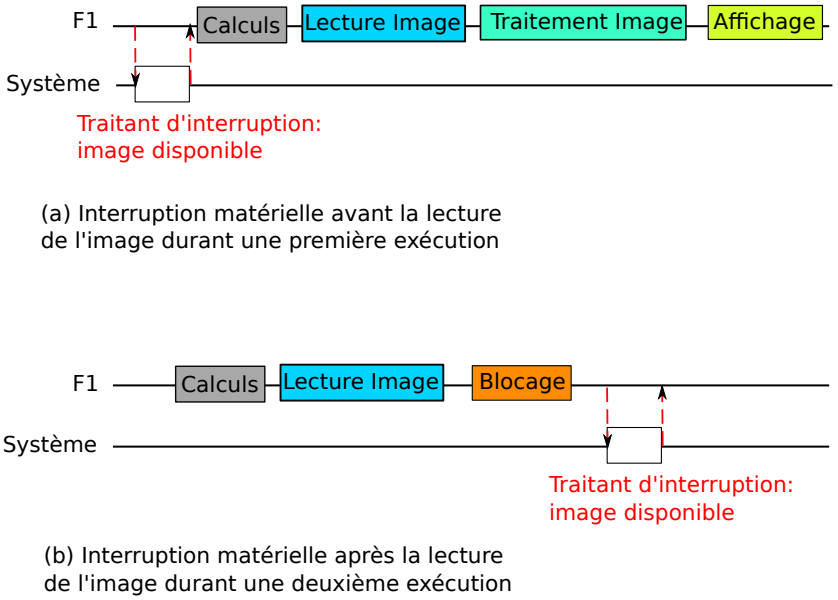


FIGURE 4.9 – Exécutions non-déterministes d'un logiciel dues aux interruptions.

traitant d'interruption destiné à gérer la demande spécifique qui lui est adressée. Lorsque l'interruption est traitée, le processeur reprend l'exécution du logiciel.

Les interruptions créent du non-déterminisme puisque le moment de leur occurrence dépend d'un environnement extérieur et peuvent modifier l'état de l'exécution du logiciel. En effet, le traitant d'interruption peut modifier les registres du processeur ainsi que la mémoire et en conséquence, l'exécution du logiciel après la reprise.

Un exemple d'exécution non-déterministe suite aux interruptions est présenté sur la figure 4.9. Cette exécution effectue des calculs, et ensuite lit une image à partir d'un périphérique. Lorsqu'une image est disponible, elle est traitée et affichée. Lorsque cette image n'est pas disponible, l'exécution est en attente. Sur la figure 4.9 (a), une image est délivrée suite à une interruption avant sa lecture dans le logiciel. Une deuxième exécution en attente est montrée sur la figure 4.9 (b) où l'occurrence de l'interruption se fait après la lecture de l'image.

L'attente active comprend principalement deux opérations pour accéder aux données des périphériques. Une première qui indique la disponibilité de données et une deuxième qui fournit ces données. Le contenu ainsi que la taille de ces données sont non-déterministes puisqu'ils dépendent d'un contexte extérieur. En conséquence, le déroulement de l'exécution est non-déterministe lorsqu'il dépend des données fournies suite à l'attente active.

Nous ne reproduisons pas les interruptions, ce qui limite notre solution pour

des couches logicielles de haut niveau mais réduit l'intrusion. En effet, nous supposons que la complexité des MPSoC nécessiterait d'utiliser un système qui gère les interruptions. Ce système fournit une interface de plus haut niveau comme l'attente active afin d'exporter les données des périphériques aux couches logicielles de plus haut niveau. En reproduisant ces couches, nous nous focalisons sur les entrées/sorties d'une partie des périphériques utilisées par un logiciel ciblé pour la mise au point. Nous évitons donc de tracer toutes les interruptions des périphériques produites au niveau système.

La ré-exécution déterministe de l'attente active nécessite de reproduire les données en entrée suite à chaque requête d'un périphérique. Nous supposons que durant l'exécution de référence, le contenu des données est enregistré par des dispositifs spécifiques. Ces dispositifs sont par exemple, des ports de traces [Man01]. En conséquence, pour reproduire les entrées, il suffit d'enregistrer la taille des données disponibles suite à chaque requête. Durant la ré-exécution, cette taille est consultée à partir de la trace et le contenu est lu à partir du dispositif spécifique.

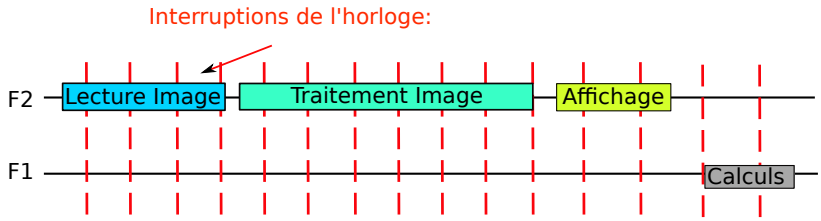
4.4.4 Ordonnancement

L'ordonnancement est un mécanisme système qui donne accès aux ressources systèmes des flots d'exécution. Ce mécanisme interromp périodiquement un flot d'exécution actif qui accède aux ressources, le met dans une queue en attente et active un autre processus. Un ordonnancement peut être effectué dû aux multiples facteurs comme :

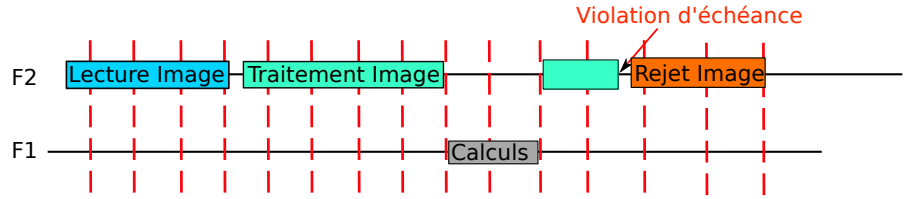
- Partage du temps de calcul d'un processeur parmi plusieurs flots d'exécution.
- Respect des priorités des flots d'exécution. Par exemple, lorsqu'un flot d'exécution avec une priorité plus élevée est créé, il faut l'ordonnancer immédiatement.
- Libération du processeur lorsqu'un flot d'exécution est bloqué.

En général, l'ordonnancement est exécuté suite à une interruption de l'horloge effectuée après l'exécution d'une instruction arbitraire du logiciel. Puisque cette instruction n'est pas toujours la même, l'exécution peut être non-déterministe. Dans un logiciel qui doit respecter des échéances, l'occurrence d'une interruption durant le calcul critique peut retarder son exécution et donc violer ces échéances. Cependant, lorsque l'interruption ne se produit pas durant ce calcul critique, l'exécution peut se dérouler d'une manière normale.

Sur la figure 4.10, nous montrons deux exécutions d'un logiciel qui se déroulent d'une manière différente dû à l'ordonnancement. Sur la figure 4.10 (a), le deuxième flot d'exécution lit, traite et affiche une image avant d'être préempté par le premier flot d'exécution. Sur la figure 4.10 (b), le deuxième flot d'exécution est préempté



(a) Ordonnancement durant une exécution qui permet l'affichage d'une image



(b) Ordonnancement durant une exécution qui ne permet pas l'affichage d'une image

FIGURE 4.10 – Exécutions non-déterministes d'un logiciel dues à l'ordonnanceur.

durant les traitements de l'image. Il ne respecte pas la fréquence d'affichage de l'image et viole l'échéance. En conséquence, l'image est rejetée.

Les ordonnancements sont dus aux interruptions de l'horloge interne qui a une fréquence élevée. En conséquence, le nombre des interruptions est élevé et induit une intrusion importante si on enregistre chaque occurrence. En conséquence, nous avons fait le choix de ne pas reproduire les ordonnancements.

4.5 Méthode de ré-exécution partielle

La ré-exécution partielle concerne une partie des nœuds et un intervalle de temps d'exécution, réduisant la difficulté de mise au point sur deux dimensions. En effet, les développeurs sélectionnent des groupes de processeurs (nœuds) plutôt que les processeurs individuels. Nous avons fait ce choix puisque les développeurs veulent en général mettre au point une partie de leurs logiciels et ces parties sont exécutées sur des nœuds spécifiques. De plus, le nombre de nœuds est moins important que le nombre de processeurs, facilitant le choix des éléments supposés fautifs.

Dans notre méthode, nous introduisons deux phases ainsi que la notion de

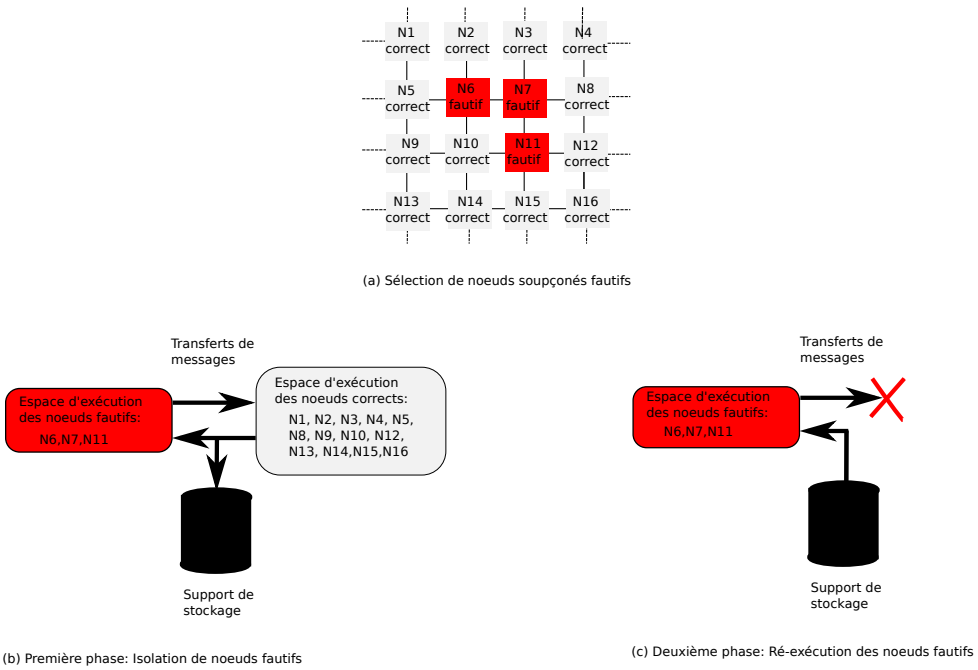


FIGURE 4.11 – Exécutions non-déterministes d'un logiciel dues à l'ordonnanceur.

nœuds fautifs et de *nœuds corrects* montrés sur la figure 4.11. Sur la figure 4.11(a), nous présentons une architecture avec 16 nœuds. Les développeurs supposent que les nœuds N6,N7 et N11 sont fautifs et les sélectionnent. Nous avons introduit une première phase représentée sur la figure 4.11(b). Cette phase enregistre les données reçues par les *nœuds fautifs* et envoyées par les *nœuds corrects*.

Nous avons introduit une deuxième phase. Cette phase, représentée sur la figure 4.11(c), correspond à une ré-exécution déterministe des *nœuds fautifs*. Ces nœuds utilisent les données enregistrées lorsqu'ils ont besoin de données envoyées par les *nœuds corrects*. Par ailleurs, les messages à destination des nœuds corrects ne sont pas envoyés.

Nous introduisons la notion de débogage durant un intervalle de temps. Il s'agit de restreindre l'utilisation du débogueur pendant une période de temps durant l'exécution, délimitée par deux bornes. Nous avons défini ces bornes comme deux événements dans les traces de ré-exécution déterministe, sélectionnées par les développeurs. La détection de ces bornes est effectuée lors des lectures des traces par le mécanisme de ré-exécution déterministe. Lorsqu'une telle lecture corresponde à la première borne, l'exécution est suspendue sous le contrôle du déboguer. Les développeurs utilisent ce débogueur jusqu'à ce qu'une autre lecture corresponde à la deuxième borne.

Une caractéristique importante de notre choix des deux dimensions pour déli-

miter une partie de l'exécution est l'indépendance des plate-formes MPSoC. D'une part, toutes les architectures MPSoC sont caractérisées par des groupes de processeurs. D'autre part, toutes les exécutions non-déterministes produisent des traces qui sont utilisées par les mécanismes de ré-exécution déterministe. Deux lectures des traces de ré-exécution déterministe permettent donc de délimiter un intervalle de temps à déboguer.

4.5.1 Collecte d'information

Nous avons identifié quatre types d'interactions entre les nœuds. Dans chaque cas, nous analysons ces interactions afin de proposer des mécanismes de collecte de traces pour la ré-exécution partielle.

1. Message échangé entre deux *nœuds corrects*.
2. Message échangé entre deux *nœuds fautifs*.
3. Message envoyé par un *nœud correct* et reçu par un *nœud fautif*.
4. Message envoyé par un *nœud fautif* et reçu par un *nœud correct*.

Afin de détecter le type de l'interaction, les nœuds doivent connaître le statut du nœud distant lors de chaque envoi et réception de message. Lorsqu'il s'agit d'une réception, il suffit d'ajouter l'identifiant du nœud distant dans le corps du message. Dans le cas d'un envoi, un message supplémentaire est crée. Ce message est en provenance du nœud distant et contient son identifiant.

Dans les deux premiers cas, la collecte de données n'est pas nécessaire puisque les *nœuds corrects* ne sont pas ré-exécutés et tous les *nœuds fautifs* participent à la ré-exécution.

Dans le troisième cas, les données des messages reçus doivent être sauvegardées durant la première phase puisque les *nœuds corrects* n'existent pas durant la deuxième phase et ne pourront pas envoyer ces messages. Nous traçons donc un couple contenant les données du message ainsi que de la valeur d'un compteur d'opérations de communication que nous utilisons durant la deuxième phase. Ce compteur, spécifique au flot d'exécution est incrémenté après chaque opération de communication. Nous l'utilisons afin de marquer les messages reçus depuis un *nœud correct*.

Dans le quatrième cas, il suffit de marquer que l'envoi du message courant ne doit pas être effectué durant la deuxième phase. Il suffit donc d'enregistrer uniquement la valeur du compteur d'opérations de communication.

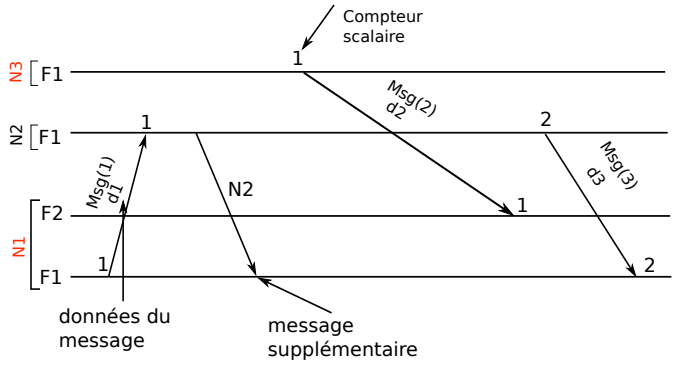


FIGURE 4.12 – Traçage de données pour la ré-exécution partielle des nœuds : Nœud1 et Nœud3.

La figure 4.12 représente les mécanismes de collecte de données lors de la première phase pour la ré-exécution partielle des nœuds 1 et 3. Le message Msg(1) montre les mécanismes mis en place pour le troisième cas de dépendance de données. Dans ce cas, le *nœud fautif* détecte que le message envoyé est reçu par un *nœud correct* en utilisant le message supplémentaire $N2$. Ce message supplémentaire contient l'identifiant (N2) du nœud qui reçoit le message . La valeur courante du compteur d'opérations de communication est donc tracée. Le message Msg(2) représente le cas numéro deux de dépendances de données où aucun traçage n'est nécessaire. Le message Msg(3) représente le cas numéro trois. Dans ce cas, lors de la consommation, le nœud N1 détecte que ce message est envoyé par le nœud N2 en extrayant son identifiant du message. La valeur du compteur, ainsi que les données du message sont tracées.

Les données tracées sont utilisées dans la deuxième phase pour ré-exécuter les *nœuds fautifs* sans avoir besoin des *nœuds corrects*. Nous montrons les mécanismes mis en place dans la section suivante.

4.5.2 Ré-exécution partielle

La deuxième phase consiste à ré-exécuter les *nœuds fautifs* en utilisant la trace à la place des interactions avec les *nœuds corrects* et à détecter les bornes de l'intervalle de temps sélectionné.

Comme durant la première phase, les quatre cas doivent être traités. Cependant, puisque les flots sur les nœuds corrects ne sont pas exécutés, le cas un ne se présenterais pas. Afin de distinguer entre les trois autres cas, nous utilisons à nouveau un compteur d'opérations de communication et comparons sa valeur à celle qui est dans la trace pour chaque opération d'envoi/réception de message. Lorsque la valeur de ce compteur n'est pas dans la trace, indépendamment du

type de l'opération, on est dans le deuxième cas puisqu'aucune information n'est tracée. Lorsque la valeur du compteur appartient à la trace et l'opération en cours est d'envoi de message, le cas numéro quatre se présente. Le message ne doit pas être envoyé puisque les flots ne sont pas exécutés sur le nœud distant. Lorsque la valeur du compteur est tracée mais l'opération en cours est de réception, son contenu est fourni à partir de la trace.

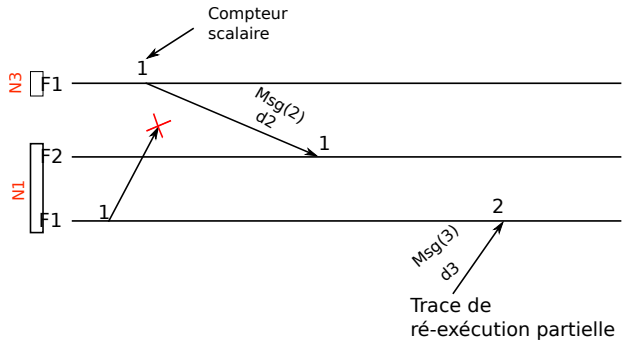


FIGURE 4.13 – Ré-exécution partielle des nœuds : Nœud1 et Nœud3.

Sur la figure 4.13, nous montrons la ré-exécution de deux nœuds (N1 et n3) parmi les trois présentés sur la figure précédente 4.12. Le message Msg(1) n'est pas envoyé puisque la valeur du compteur d'opérations de communication (1) du flot d'exécution 1 appartient à la trace. Le deuxième message est procédé comme durant la première phase puisque les valeurs des compteurs respectifs du nœud 1 et du nœud 2 ne sont pas tracées. Les données du troisième message sont fournies à partir de la trace, puisque la valeur du compteur scalaire (2) appartient à la trace.

Notre mécanisme de détection d'intervalle de temps à déboguer prend en entrée deux événements (bornes de l'intervalle) qui correspondent à deux entrées dans la trace de ré-exécution déterministe. Durant la ré-exécution partielle, les valeurs lues à partir des traces sont comparées aux bornes. Lorsque la première borne est atteinte, l'exécution est suspendue et un débogueur est lancé. Lorsque la deuxième borne est atteinte, la deuxième phase est terminée.

4.6 Synthèse

Dans ce chapitre, nous avons présenté une nouvelle méthodologie pour la mise au point des logiciels MPSoC. Notre méthodologie a deux propriétés principales. Premièrement, cette méthodologie utilise la ré-exécution déterministe pour reproduire les erreurs non-déterministes. Deuxièmement, le logiciel est ré-exécuté partiellement afin de mieux se focaliser sur la partie de l'exécution supposée fautive.

Afin de prendre en compte une large diversité de plate-formes MPSoC, nous avons proposé de sélectionner une partie de l'exécution selon deux critères génériques : une partie des nœuds de l'architecture et un intervalle de temps de l'exécution au comportement supposé anormal.

Nous avons utilisé un ensemble de méthodes afin de reproduire les exécutions non-déterministes du logiciel MPSoC. D'un côté, nous avons identifié les sources de non-déterminisme dans les exécutions. D'un autre côté, nous avons sélectionné un ensemble de méthodes de ré-exécution déterministe qui permettent de faire un compromis entre la complétude et la performance. Nous nous sommes focalisés sur la couche applicative et certaines sources de non-déterminisme ne sont pas reproduites afin de réduire la perturbation. Cependant, nous avons montré dans l'état de l'art que ces sources peuvent être reproduites lorsque l'intrusion n'est pas un facteur déterminant.

La ré-exécution partielle est accomplie en utilisant une nouvelle méthode qui comprend deux phases. Cette méthode prend en entrée les traces de ré-exécution déterministe, un ensemble d'identifiants de nœuds supposés fautifs et deux entrées dans le fichier de traces qui bornent l'intervalle de temps où l'exécution a un comportement anormal. La première phase consiste à collecter des données afin de pouvoir ré-exécuter les nœuds supposés fautifs en isolation. Durant la deuxième phase, seuls ces nœuds sont ré-exécutés. L'intervalle de temps est détecté en comparant les valeurs lues à partir des traces avec les bornes de l'intervalle. La mise au point est effectuée entre ces deux bornes.

Notre méthodologie peut être utilisée pour mettre au point des logiciels MPSoC conçus avec un ensemble large d'environnements de programmation qui ont un ensemble de propriétés communes. Dans le chapitre suivant, nous montrons une implémentation de notre méthode pour mettre au point du logiciel MPSoC conçu en utilisant un tel environnement de programmation.

Chapitre 5

Mise au point d'applications sur MPSoC avec ReDSoC

Dans ce chapitre, nous présentons le prototype que nous avons implémenté afin de valider notre méthodologie de mise au point. Nous l'avons nommé **ReDSoC** pour **Re**cord-Replay for **D**eterministic **SoC**-Debug. Dans la suite, nous présentons d'abord une vue générale du processus de mise au point MPSoC en utilisant ReDSoC. Ensuite, nous introduisons un API MPSoC qui est utilisé par les applications afin d'exploiter l'architecture et de récréer les problématiques de mise au point MPSoC. Finalement, nous présentons chacun des quatre outils de ReDSoC.

5.1 Vue générale

Notre prototype ReDSoC effectue le débogage d'un logiciel MPSoC à partir d'une machine de développement (*hôte*). Cette machine *hôte* est un ordinateur standard disposant d'un clavier pour écrire les commandes de débogage, d'un écran afin de visualiser le comportement du logiciel débogué et d'une connexion avec la plate-forme embarquée afin d'exécuter les commandes de débogage. La machine *hôte* est nécessaire lorsque la plate-forme MPSoC est par exemple une puce dépourvue d'écran, de clavier et/ou de lourde infrastructure de débogage. Surtout, elle permet de déporter une partie du coût de débogage en dehors du MPSoC.

La figure 5.1 représente les quatre outils utilisés dans ReDSoC pour déboguer le logiciel MPSoC. Les commandes de débogage sont exécutées par le débogueur GDB sur la plate-forme *hôte*, basée sur Linux. Nous avons apporté une extension au client GDB qui implémente l'isolation d'un intervalle de temps de notre outil de réexécution partielle. La plate-forme *hôte* exécute également un outil de visualisation de traces qui permet de rechercher un éventuel comportement anormal.

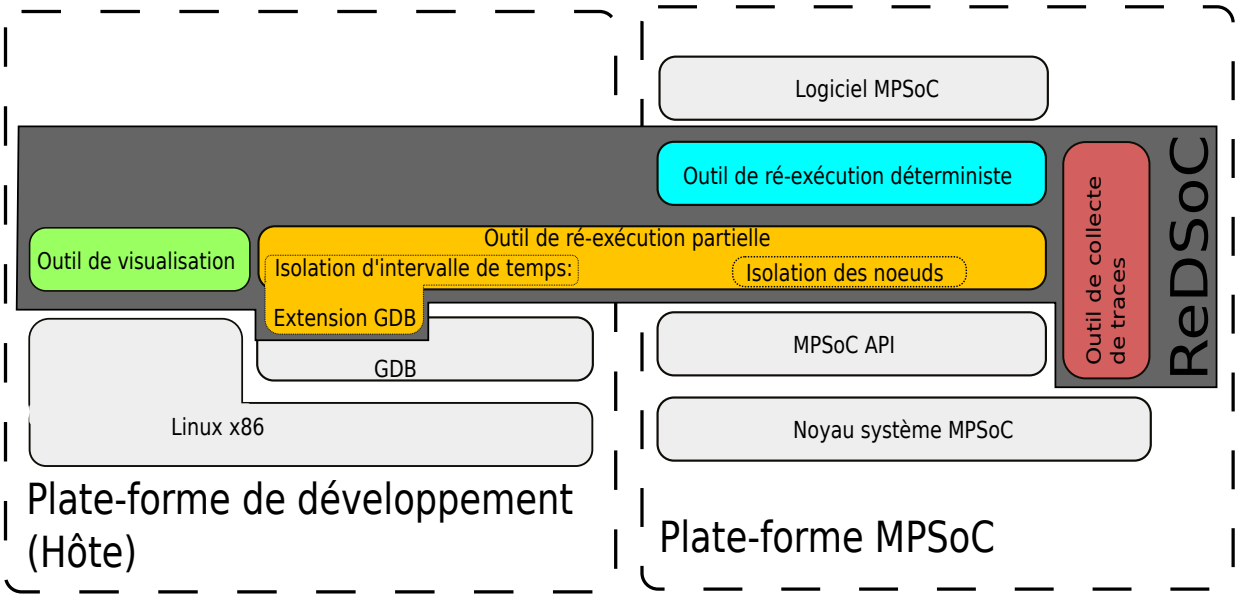


FIGURE 5.1 – Architecture logicielle de ReDSoC.

Sur la plate-forme déboguée, nous supposons que le logiciel MPSoC utilise une interface de programmation (API) (cf. section 5.2) afin d'exploiter l'architecture. Les appels à cette API, sont enregistrés ou modifiés par l'outil de ré-exécution déterministe (cf. section 5.4) et la partie isolation de nœuds de l'outil de ré-exécution partielle (cf. section 5.5). L'enregistrement des données est effectué par un outil de collecte de traces (cf. section 5.3) qui est configuré afin d'utiliser des informations d'un noyau spécifique MPSoC.

Il existent principalement trois stratégies de déploiement de ReDSoC sur l'architecture : une instance pour tous les nœuds, une instance par tranches de nœuds ou encore une instance par nœud. La première stratégie permet d'avoir la plus faible empreinte au niveau de la mémoire. Cependant, cette solution est difficile à mettre en œuvre et peut poser des problèmes au niveau de la contention dû à la gestion des interactions entre ReDSoC et l'application. La deuxième représente un compromis entre la simplicité d'implémentation et l'empreinte mémoire. Dans nos travaux, nous avons déployé ReDSoC en utilisant une instance indépendante pour chaque nœud de l'architecture dû à la simplicité de mise en œuvre et son faible impact sur la contention.

La figure 5.2 représente un exemple de déploiement de ReDSoC sur quatre nœuds. Chaque instance de ReDSoC prend en entrée un fichier de configuration. Ce fichier est spécifique à un nœud et doit être écrit, ainsi que transféré sur le nœud correspondant, par les développeurs. Un fichier de configuration contient les informations suivantes :

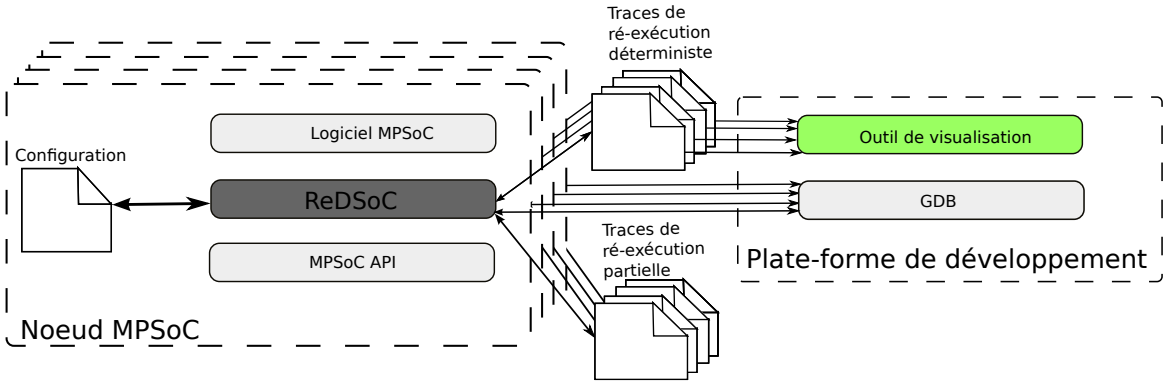


FIGURE 5.2 – Déploiement de ReDSoC sur un nœud MPSoC.

- Un entier qui indique l'identifiant du nœud.
- Un entier qui indique la phase de mise au point (collecte de données, ré-exécution déterministe ou ré-exécution partielle).
- Un ensemble d'entiers indiquant les nœuds supposés fautifs.

ReDSoC produit ou utilise des traces de ré-exécution déterministe ou de ré-exécution partielle en fonction de la phase de mise au point. Ces traces peuvent être enregistrées/lues en utilisant le système de fichiers du nœud correspondant ou exportées/importées en utilisant un support de stockage externe. Il est possible que tous les nœuds n'aient pas accès au système du fichier. Dans ce cas, ReDSoC peut être étendu avec un support qui permet de transférer les traces et les fichiers de configuration de tous les nœuds en utilisant ceux qui ont accès au système de fichiers. Cette solution impliquerait d'utiliser des protocoles de communication comme le *round-robin* avec l'objectif principal de limiter la contention due aux transferts de traces simultanés. Pour éviter la contention, il serait préférable d'utiliser des ressources matérielles dédiées au traçage, mais le coût de la plate-forme serait plus élevé.

Sur la plate-forme de développement, les traces de ré-exécution déterministe de tous les nœuds doivent être transférées afin d'être visualisées par KPTrace. De plus, nous supposons qu'il existe une implémentation spécifique de serveur GDB qui peut transférer des informations entre toutes les instances de ReDSoC sur les nœuds et le client GDB, sur la plate-forme de développement.

5.2 API MPSoC

Plusieurs interfaces de programmation pour les MPSoC ont été proposées dans la littérature(*cf.* chapitre 2). Cependant, aucune n'a pu s'imposer comme standard

de programmation. Elles sont toutes spécifiques à un type d'application ou à une plate-forme. La majorité sont d'une complexité importante et ne représentent pas les problématiques de mise au point dues au non-déterminisme et à la complexité d'exécution. Par conséquent, nous avons choisi de définir notre propre API MPSoC.

Notre objectif est de concevoir un API afin de représenter les sources de non-déterminisme MPSoC (chapitre 4, section 4.4), le passage à l'échelle ainsi que le modèle de la plate-forme MPSoC (cf. chapitre 4, section 4.3.2) que nous avons défini. Notre API doit donc créer une multitude de flots d'exécution (tâches) qui se synchronisent, communiquent et accèdent aux données des périphériques.

Nous avons décidé de baser notre API sur une partie du standard POSIX [SFR04] à cause de sa disponibilité et de son utilisation par une large communauté. Cependant, nous avons apporté des simplifications importantes afin d'accélérer le développement tout en remplissant nos objectifs.

```
/*Gestion de tâches*/
int taskCreate(task_t *task,void *(*funct)(void *),void *args, int node);
int taskJoin(task_t *task);

/*Synchronisation*/
int synCreateMutex/synDestroyMutex(mutex_t *m);
int synCreateCond/synDestroyCond(cond_t *c);
int synLock(mutex_t *m);
int synUnlock(mutex_t *m);
int synCondition(cond_t *c, mutex_t *m);
int synSignal(cond_t *c);

/*Communication*/
int tcpCreate(int s,int p, int type);
int tcpListen(int s);
int tcpAccept(int local_s, int distant_s);
int tcpConnect(int s,node_id num,int p);
int tcpClose(int s);
int tcpSend(int s, char *buff, int size);
int tcpRecv(int s, char *buff, int size);

/*Accès aux données des périphériques et au système de fichiers*/
int pOpen(const char *name);
int pClose(int fd);
int pRead(int fd, void *buf, size_t count);
int pWrite(int fd, void *buf, size_t count);
```

FIGURE 5.3 – Interface de programmation du logiciel MPSoC.

Nous avons défini quatre types de fonctions présentés sur la figure 5.3. L'ensemble de ces fonctions retourne un code d'erreur ou une valeur de succès dépendant de l'implémentation. Dans la suite, nous présentons ces fonctions.

5.2.1 Tâches

Notre API permet de représenter le logiciel MPSoC comme un ensemble de flots d'exécution que nous appelons des tâches. Cette représentation est fréquemment utilisée dans les noyaux systèmes multi-tâches des systèmes embarqués [STMa] [Rivb]. Les tâches sont ordonnancées sur les processeurs des nœuds et exécutent des fonctions. Nous fournissons deux fonctions de gestion des tâches :

- `int taskCreate(task_t *task,void *(*funct)(void *),void *args, int node)` : crée une nouvelle tâche qui s'exécute en concurrence avec la tâche courante. Cette nouvelle tâche exécute la fonction *funct* avec les paramètres en entrée *args* sur un des processeurs du nœud *node*. L'argument *task* est fourni en retour et pointe vers la nouvelle structure qui est créée. La politique d'ordonnancement des tâches sur les processeurs est un choix d'implémentation que nous n'imposons pas.
- `int taskJoin(task_t *task)` : suspende la tâche courante jusqu'à la fin de l'exécution de la tâche *task*.

5.2.2 Synchronisation

L'interface de synchronisation permet de coordonner les tâches au sein d'un nœud. Cette coordination est basée sur deux structures de données couramment utilisées, faisant partie du standard POSIX : les verrous et les conditions.

- `int synCreateMutex(mutex_t *m)` : alloue une structure *$*m$* de type verrou.
- `int synDestroyMutex(mutex_t *m)` : libère une structure *$*m$* de type verrou.
- `int synCreateCond(cond_t *c)` : alloue une structure *$*c$* de type condition.
- `int synDestroyCond(cond_t *c)` : libère une structure *$*c$* de type condition.
- `int synLock(mutex_t *m)` : le verrou *$*m$* est pris. Le code est exécuté d'une manière atomique jusqu'à `synUnlock`.
- `int synUnlock(mutex_t *m)` : le verrou *$*m$* est relâché. La section atomique peut être accédée par une autre tâche.
- `int synCondition(cond_t *c, mutex_t *m)` : relâche le verrou *$*m$* et bloque la tâche courante. Lorsque la condition *$*c$* est signalée, la tâche courante est débloquée et le verrou `*m` est verrouillé.

- `int synSignal(cond_t *c)` : envoie un signal vers une des tâches qui sont bloquées sur la condition `*c`. Lorsque aucune tâche n'est bloquée, cette fonction n'a pas d'effet sur l'exécution.

5.2.3 Communication par messages

La communication par messages entre les tâches des différents nœuds est implémentée en utilisant les *sockets* [CSP04] et fait partie du standard POSIX. Une *socket* représente d'une manière unique les deux extrémités d'un lien de communication entre deux ports. Nous avons défini une interface simplifiée de gestion de *sockets* qui représente les extrémités d'un lien par deux tâches qui appartiennent à 2 nœuds différents. La communication utilise le protocole TCP [Ins81].

Pour communiquer, deux tâches doivent d'abord se connecter. Pour cela, l'une d'entre elles doit exécuter la séquence de fonctions `tcpCreate, tcpListen et tcpAccept`. L'autre doit exécuter `tcpCreate et tcpConnect`.

- `int tcpCreate(int s,int p, int type)` : retourne un entier *s* qui représente une *socket* de numéro de port *p*. L'argument *type* prend deux valeurs, 0 ou 1 qui indiquent respectivement que la *socket* utilise de la communication bloquante ou non-bloquante.
- `int tcpListen(int s)` : met la *socket s* en attente de connexion.
- `tcp_socket_t tcpAccept(int s)` : établit un lien de communication entre la *socket* locale *s* et celle qui est distante, fournie en sortie.
- `int tcpConnect(int s,node_id num,int p)` : établit un lien de communication entre la *socket* locale *s* et une distante qui est mise en attente sur le port numéro *p* du nœud *node_ id*.
- `int tcpSend/tcpRecv(int s, char *buff, int size)` : envoie et reçoit respectivement des données de taille maximale *size* allouées dans la mémoire tampon **buff* par le lien de communication représenté par la *socket s*.
- `int tcpClose(int s)` : libère la structure *s*.

5.2.4 Entrées/Sorties

Les données d'entrée/sortie ainsi que du système de fichiers sont accédées en lecture ou en écriture non-bloquante de façon similaire au système UNIX [Ste92].

- `int pOpen(const char *name)` : retourne un entier qui représente le périphérique de nom **name*.
- `int pClose(int fd)` : l'entier *fd* ne représente plus aucun périphérique.

- `int pRead(int fd, void *buf, size_t count)` : remplit la mémoire tampon **buf* avec des données de taille maximale *count* reçues par le périphérique *fd*.
- `int pWrite(int fd, void *buf, size_t count)` : envoie les données allouées dans la mémoire tampon **buf* de taille *count* vers le périphérique *fd*.

Nous allons interfacer ReDSoC avec cette API afin de mettre au point le logiciel MPSoC. Afin d'utiliser ReDSoC sur une plate-forme MPSoC qui ne fournit pas cette interface, il suffirait d'identifier les mécanismes représentés par notre API et d'effectuer l'interfaçage correspondant.

5.3 Outil de collecte de traces

L'outil de collecte de traces est issu de nos travaux de collaboration sur la conception d'un environnement générique d'observation MPSoC [PRMMG+09, PRMG+09, PMG+09]. Cet environnement est basé sur des composants logiciels afin de fournir des propriétés de généricité, d'observation multi-niveaux et de configurabilité.

La gestion des composants durant l'exécution du logiciel a un surcoût qui accroît l'intrusion. Le surcoût est justifié lorsqu'il y a plusieurs couches logicielles à observer, les éléments à analyser ne sont pas connus etc. Puisque nous voulons observer uniquement les éléments de notre API, nous avons conçu une solution simplifiée qui évite le surcoût des composants.

Notre outil de collecte de traces fournit une interface de quatre fonctions, implémentées selon l'architecture :

- `int getNodeId()` : retourne un entier qui représente l'identifiant du nœud.
- `int getTaskId()` : retourne un entier qui représente l'identifiant de la tâche.
- `unsigned int getTimestamp()` : retourne une estampille temporelle en sortie en microsecondes. Cette estampille est utilisée par l'outil de visualisation pour représenter le temps de l'occurrence d'une entrée dans la trace. Ce temps peut être obtenu typiquement par lecture du registre d'horloge interne ou en utilisant une interface système. Cette horloge interne peut être spécifique à un processeur, à un nœud ou à l'architecture. Les entrées dans la trace sont donc représentées dans autant d'échelles de temps que d'horloges. Par conséquent, l'outil de visualisation doit permettre la représentation de plusieurs échelles de temps.
- `int trace(int type, char *traceData, int size, vect_t Vector)` : génère une entrée dans la trace qui contient de un à cinq champs : Es-

tampille Temporelle, Identifiant du Nœud, Identifiant de la Tâche, Type d'entrée dans la Trace et Données de l'entrée de la Trace ([ET, IN, IT, TT, DT]). L'argument *Vector* est un vecteur de cinq bits qui représente les cinq champs. Un champ est inclus dans la trace lorsque sa valeur respective dans le vecteur est égale à 1. Le type de la trace (*type*) est un entier qui prend les valeurs de 1 à 5 pour distinguer les traces respectives des quatre sources de non-déterminisme et de ré-exécution partielle. L'argument **traceData* est un pointeur vers un tampon mémoire qui contient les données à enregistrer. L'argument *size* représente la taille des données à enregistrer.

L'écriture sur un support de stockage des données peut être effectuée en utilisant un port de traces matériel [MW01] pour réduire l'intrusion ou sur un système de ficher en utilisant une interface système d'entrées/sorties. En effet, la gestion des traces est dépendante de la plate-forme considérée. Par exemple, lorsque cette plate-forme intègre un port de traces pour chaque nœud, les traces peuvent être exportées. Lorsque le port de traces n'est pas disponible et la plate-forme intègre suffisamment de ressources de stockage, les données peuvent être enregistrées sur le système de fichiers local.

5.4 Outil de ré-exécution déterministe

L'outil de ré-exécution déterministe permet de reproduire le comportement non-déterministe dû à la synchronisation, à la communication par messages et aux accès des données des périphériques. Pour cela, nous utilisons les méthodes présentées dans le chapitre précédent (section 4.4). Dans la suite, nous montrons pour chaque type de fonction le comportement non-déterministe, l'implémentation de la collecte de données et la reproduction du comportement.

5.4.1 Synchronisation

Le non-déterminisme d'exécution lié à la synchronisation est dû à l'ordre d'accès à une section critique par plusieurs flots d'exécution. Dans notre API MPSoC, ceci est reflété dans l'utilisation des fonctions *synLock* et *synCondition*. En effet, si deux tâches utilisent *synLock* pour accéder à une section critique dans une exécution, il se peut alors que dans une deuxième exécution leur ordre d'accès change, créant ainsi du non-déterminisme. Pour ré-exécuter de manière déterministe les accès synchronisés des tâches, il est donc nécessaire de reproduire l'ordre des prises des verrous par les tâches dans ces fonctions.

Nous utilisons les horloges de Lamport pour tracer les prises des verrous. Durant la ré-exécution déterministe, les tâches sont directement bloquées et débloquées selon l'ordre enregistré.

Dans la suite, nous détaillons le traitement de `synLock(mutex_t *m)`. La reproduction de `synCondition(cond_t *c, mutex_t *m)` utilise en effet les mêmes mécanismes.

```
1  RRsynLock(mutex_t *m){
2  ...
3     if RecordPhase() {
4        synLock(m);
5        TraceMechanism(m);
6     } else
7        WaitSynchVar();
8        synLock(m);
9        NextSliceCounterValue();
10 ...
11 }
```

FIGURE 5.4 – Interception de synLock.

Afin de reproduire la prise des verrous dans `synLock`, nous introduisons une fonction d'encapsulation `RRsynLock`, présentée sur la figure 5.4. Cette fonction d'encapsulation est appelée à la place de `synLock` dans le logiciel et effectue la ré-exécution déterministe. Durant la phase d'enregistrement, le verrou est pris avec la fonction `synLock` originelle et l'ordre d'accès est tracé par `TraceMechanism`. Durant la phase de ré-exécution, la fonction `WaitSynchVar` bloque la tâche jusqu'à ce que l'ordre enregistré soit respecté. Ensuite, la fonction `NextSliceCounterValue` met à jour les structures de ré-exécution déterministe.

```
1  int TraceMechanism(mutex_t *m) {
2  ...
3     oldEFlowLamport[eFlowID] = eFlowLamport[eFlowID];
4     if (eFlowLamport[eFlowID]<synchVarLamport[synchVarID]) {
5        generateSynchVarTraceData(traceData,oldEFlowLamport[eFlowID],eFlowLamport[eFlowID]);
6        trace(synchVarTraceType,traceData,synchVarTraceSize,synchVarVector);
7     }
8     synchVarLamport[synchVarID] = max(synchVarLamport[synchVarID],eFlowLamport[eFlowID])+1;
9     eFlowLamport[eFlowID] = synchVarLamport[synchVarID];
10  ...
11 }
```

FIGURE 5.5 – Traçage de la synchronisation.

L'extrait de la fonction `TraceMechanism` (figure 5.5) trace l'ordre des prises des verrous en utilisant une horloge de Lamport par tâche et une par verrou. Ces horloges sont représentées par deux tableaux d'entiers $eFlowLamport[eFlowID]$ et $synchVarLamport[synchVarID]$ où $eFlowID$ et $synchVarID$ représentent respectivement les identifiants des tâches et des verrous.

La prise d'un verrou par une tâche est non-déterministe lorsque l'horloge du verrou est en avance par rapport à l'horloge de la tâche (ligne 4). Ce cas se présent par exemple, lorsque deux tâches accèdent successivement à la même section

critique. L'ordre des accès est indéterminé et l'horloge d'une des deux tâches reste en retard par rapport à l'horloge du verrou de la section critique. Dans ce cas, le couple d'ancienne et de nouvelle valeur d'horloge de la tâche sont tracées. La fonction `generateSynchVarTraceData` (ligne 5) assigne à la variable *traceData* ce couple. La fonction `trace` enregistre *traceData* dans un fichier. Le format des entrées dans la trace est le suivant :

- Estampille temporelle.
- Type de la trace.
- Identifiant de tâche : un entier qui représente la tâche qui détecte le non-déterminisme.
- Données d'une entrée de la trace : deux entiers qui représentent un couple d'ancienne et de nouvelle horloge de Lamport de la tâche.

Lors de chaque prise de verrou, l'horloge de la tâche courante ainsi que du verrou sont avancées en utilisant leur maximum incrémenté d'un (ligne 8).

```
1  void WaitSynchVar() {
2  ...
3     while  (!lowestSliceCounterValue()) {
4        blockedTasks++;
5        synCondition(&waitCond,&waitSynchVar);
6      }
7     for (i=0;i<blockedTasks;i++)
8        synSignal(&waitCond);
9     }
10 ...
11 }
```

FIGURE 5.6 – Vérification de l'ordre enregistré des prises des verrous.

La figure 5.6 présente un extrait de la fonction `WaitSynchVar` qui bloque une tâche lorsque l'ordre des prises des verrous n'est pas conforme à celui enregistré. Lorsque cet ordre est conforme, toutes les tâches sont débloquées et le verrou est pris. Nous utilisons un compteur scalaire *eFlowID* par tâche, maintenu dans un tableau partagé *SliceCounter*. L'accès au compteur de la tâche avec l'identifiant *eFlowID* se fait par *SliceCounter[eFlowID]*. Une tâche prend un verrou lorsque son compteur *SliceCounter* a la plus petite valeur. Cette condition est vérifiée par la fonction `lowestSliceCounterValue` qui cherche le plus petit élément dans le tableau *SliceCounter* et vérifie qu'il correspond au compteur de la la tâche courante. Dans ce cas, toutes les tâches sont signalées afin qu'une autre puisse prendre le verrou. Dans le cas inverse, la tâche courante est bloquée en utilisant `synCondition(&waitCond,&waitSynchVar)` (ligne 5).

La figure 5.7 présente un extrait de la fonction `NextSliceCounterValue` qui met à jour le compteur scalaire de la tâche courante à partir des traces lorsque l'accès à un verrou est non-déterministe. Lorsque cet accès est déterministe, le compteur scalaire est incrémenté de un.

```
1  void NextSliceCounterValue() {
2  ...
3    if (SliceCounter[eFlowID] == firstTracedLC) {
4        SliceCounter[eFlowID] = lastTracedLC;
5        GetCoupleFromTrace(&firstTracedLC,&lastTracedLC);
6    }
7    else
8       SliceCounter[eFlowID]++;
9    }
10 ...
11 }
```

FIGURE 5.7 – Mise à jour des structures de ré-exécution.

L'accès à un verrou est non-déterministe lorsque la valeur du compteur scalaire de la tâche courante (`SliceCounter[eFlowID]`) atteint la valeur de la première horloge de Lamport d'un couple enregistré dans la trace (*firstTracedLC*). Dans ce cas, (*SliceCounter[eFlowID]*) prend la valeur de la deuxième horloge de Lamport de ce couple. La fonction `GetCoupleFromTrace` assigne à (*firstTracedLC*) et à (*lastTracedLC*) respectivement les deux horloges de Lamport du prochain couple enregistré dans la trace. (*SliceCounter[eFlowID]*) est incrémenté d'un lorsque sa valeur n'a pas encore atteint la valeur de la première horloge de Lamport d'un couple enregistré.

5.4.2 Communication par messages

La communication par messages peut être source de deux types de non-déterminisme dus à l'ordre de réception de messages et à la taille des données reçues en utilisant des mécanismes non-bloquants. Le premier type de non-déterminisme concerne les fonctions `tcpConnect` et `tcpAccept`. En effet, `tcpAccept` crée une *socket* avec la première tâche d'un nœud quelconque qui exécute `tcpConnect` avec les paramètres adéquats. Cette première tâche peut être différente durant deux exécutions créant ainsi du non-déterminisme.

Le deuxième type de non-déterminisme concerne la fonction `tcpRecv`, lorsque la *socket* est non-bloquante. En effet, la taille des données fournies par `tcpRecv` est soit celle spécifiée dans ces paramètres, soit celle des données disponibles. La taille de ces données peut être différente durant deux exécutions puisqu'elle dépend des *tcpSend* exécutées par d'autres nœuds en concurrence.

Afin de reproduire la première source de non-déterminisme, nous enregistrons l'identité du nœud distant lors de chaque création de *socket* par la fonction `tcpAccept`. La suite de ces identités représente donc l'ordre dans lequel les tâches des différents nœuds se sont connectées à la tâche qui accepte les connexions en utilisant `tcpAccept`. Durant la ré-exécution déterministe, une tâche accepte toutes les connexions mais vérifie si l'identité de la tâche distante respecte l'ordre enre-

gistré. Lorsque l'ordre n'est pas respecté, la *socket* connectée est enregistrée dans une mémoire tampon et la tâche continue d'attendre la *socket* espérée. Les *sockets* enregistrées sont fournies lorsqu'elles respectent l'ordre enregistré.

```
1  int RRtcpConnect(int s,node_id num,int p) {
2  ...
3     ret = tcpConnect(s,num,p);
4     MessageProduction(num,buffer,&size);
5     tcpSend(s,buffer,size);
6     return ret;
7  ...
8  }
```

FIGURE 5.8 – Ré-exécution déterministe de tcpConnect.

La fonction `RRtcpConnect`, représentée sur la figure 5.8 encapsule et reproduit les appels de `tcpConnect`. Le rôle de cette fonction d'encapsulation est d'établir une connexion en exécutant `tcpConnect` et d'envoyer l'identifiant du nœud courant par cette connexion. La fonction `MessageProduction` transforme donc l'entier qui représente le nœud courant en chaîne de caractères (*buffer*) et l'envoie au nœud distant en utilisant `tcpSend`.

```
1  int RRtcpAccept(int s) {
2  ...
3     if RecordPhase() {
4        ret = tcpAccept(s);
5        tcpRecv(s,buffer,size);
6        generateTcpAcceptTraceData(traceData,buffer);
7        trace(TcpAcceptTraceType,traceData,TcpAcceptTraceSize,tcpAcceptVector);
8     } else if (!SearchTcpSocket(s[n],&ret)) {
9        ret = tcpAccept(s[n]);
10       tcpRecv(s[n],buffer,size);
11       while (!compare(getExpectedNodeid(),getRecvNodeid(buffer))) {
12          AddSocket(getRecvNodeid(buffer),s[n],ret,getSocketsArray());
13          ret = tcpAccept(s[n]);
14          tcpRecv(s[n],buffer,size);
15       }
16    }
17    return ret;
18 }
```

FIGURE 5.9 – Ré-exécution déterministe de tcpAccept( ).

Sur la figure 5.9, nous présentons la fonction `RRtcpAccept` qui encapsule et reproduit `tcpAccept`. Durant la phase du traçage, cette fonction d'encapsulation établit une connexion avec une tache distante en utilisant `tcpAccept` (ligne 4), reçoit l'identité du nœud distant `tcpRecv` (ligne 5) et produit une trace en utilisant `generateTcpAcceptTraceData` (ligne 6) et `trace` (ligne 7). Le format de cette trace est le suivant :

- Estampille temporelle.

- Type de la trace.
- Identifiant de tâche : un entier qui représente la tâche qui exécute la fonction `tcpAccept`.
- Données d'une entrée de la trace : un entier qui représente l'identifiant du nœud distant.

Nous n'utilisons pas d'horloges vectorielles pour limiter le surcoût de leur gestion et puisqu'en général la création d'une *socket* n'est pas une opération fréquente. En général, les *sockets* sont créées une fois pour permettre de multiples échanges de données.

Durant la ré-exécution déterministe, la tâche exécutante `tcpAccept`, recherche dans une mémoire tampon la disponibilité de la *socket* attendue par la fonction `SearchTcpSocket` (ligne 8). La *socket* est disponible lorsque l'identifiant du nœud distant de la *socket* correspond à celui lu à partir des traces. Dans ce cas, la fonction `SearchTcpSocket` fournit à partir de la mémoire tampon dans *s[n]* la *socket* correspondante et dans *ret* la valeur de retour de `tcpAccept` correspondante. Lorsque la *socket* n'est pas disponible, `RRtcpAccept` attend la connexion qui correspond à celle enregistrée en exécutant `tcpAccept` (ligne 9). Toutes les *sockets* créées qui ne sont pas conformes à la trace sont mises dans la mémoire tampon par `AddSocket`(ligne 12). Le premier argument de cette fonction correspond à l'identité du nœud distant reçu comme une chaîne de caractères *buffer* par `tcpRecv` (ligne 10) et transformé en entier par la fonction `getRecvNodeid`. Les arguments 2 et 3 représentent respectivement la *socket* créée en utilisant `tcpAccept` (ligne 9) et sa valeur de retour *ret*. Le quatrième argument est la fonction `getSocketsArray` qui fournit la mémoire tampon avec les *sockets* qui ne sont pas conformes à la trace afin d'ajouter la *socket* nouvellement créée.

Pour la ré-exécution déterministe de la fonction `tcpRecv` d'une *socket* non-bloquante, nous traçons un couple contenant la taille de données reçues ainsi que le nombre d'appels à `tcpRecv` qui ont retourné la même taille de données. Le format des entrées dans la trace comprend

- Estampille temporelle.
- Type de la trace.
- Identifiant de tâche : un entier qui représente la tâche qui exécute la fonction `tcpRecv`.
- Données d'une entrée de la trace : deux entiers, un pour représenter la taille de données reçues et l'autre qui représente le nombre de fois consécutives pour lesquelles la même taille de données est reçue.

La ré-exécution déterministe de `tcpRecv` consiste à attendre en boucle jusqu'à ce que la taille enregistrée dans la trace soit disponible.

5.4.3 Fonctions d'accès aux données des périphériques

L'acquisition des données des périphériques est non-déterministe lorsqu'elle est basée sur l'attente active. Notre API permet d'effectuer cette attente active en utilisant la fonction non-bloquante `pRead`. Afin de ré-exécuter d'une manière déterministe cette fonction, la taille ainsi que le contenu des données lues doiventt être reproduits. Pour la taille des données nous utilisons le même mécanisme que `tcpRecv`. Le format de la trace est donc identique.

- Estampille temporelle.
- Type de la trace.
- Identifiant de tâche : un entier qui représente la tâche qui exécute la fonction `pRead`.
- Données d'une entrée de la trace : deux entiers, un pour représenter la taille de données reçues et l'autre qui représente le nombre de fois consécutives pour lesquelles la même taille de données est reçue.

Lorsque le contenu des données n'est pas le même durant deux exécutions, nous supposons qu'il est fourni à partir des périphériques durant le traçage et disponible à partir d'un support de stockage durant la ré-exécution déterministe. En effet, durant cette ré-exécution déterministe, `pOpen` n'est pas exécuté. La taille des données qui doivent être fournies au logiciel par la fonction `pRead` est lue à partir des traces. Leur contenu est lu à partir d'un support de stockage en utilisant une fonction `DataContentRestore(buffer,size)`. Cette fonction utilise des mécanismes spécifiques à la plate-forme pour obtenir les données sauvegardées des périphériques.

5.5 Outil de ré-exécution partielle

Dans cette section, nous commençons par présenter l'implémentation des mécanismes de collecte de données qui permettent d'isoler des nœuds pour la ré-exécution partielle. Ensuite, nous continuons avec l'implémentation de la ré-exécution partielle et l'interaction avec GDB pour le débogage d'un intervalle de temps spécifique.

5.5.1 Collecte d'information

Afin d'isoler une partie des nœuds (supposés fautifs), il est nécessaire de tracer les données modifiées lors des interactions entre ces nœuds-là et les autres nœuds (supposés corrects). Dans notre API, les interactions entre les nœuds s'effectuent

en utilisant quatre fonctions : *tcpConnect*, *tcpAccept*, *tcpSend* et *tcpRecv*. Nous utilisons respectivement quatre fonctions pour tracer les données modifiées lors des interactions entre nœuds. Dans la suite, nous en présentons *tcpConnect* et *tcpRecv*, puisque les mécanismes de traçage respectifs pour les deux autres *tcpAccept* et *tcpSend* sont les mêmes.

Les nœuds considérés fautifs sont spécifiés de manière explicite dans un fichier de configuration fourni par le développeur en charge du débogage. Pour tous ces nœuds-là, nous mettons en place un mécanisme d'enregistrement de données pour la ré-exécution partielle de `tcpConnect`.

Suite à la création d'une connexion par `tcpConnect`, notre mécanisme identifie le nœud distant. Lorsque ce nœud ne fait pas partie de la liste des nœuds supposés fautifs, les deux actions suivantes sont effectuées :

- Traçage de la valeur de retour de `tcpConnect` et de la valeur d'un compteur des appels de toutes les fonctions de communication, initialisé à 0.
- Enregistrement de l'identifiant de la *socket* créée, dans un tableau.

Les valeurs du compteur enregistrées désignent les connexions établies par `tcpConnect` avec des nœuds corrects. Durant la ré-exécution partielle, pour toutes les connexions marquées, l'exécution de `tcpConnect` consiste à retourner la valeur de retour à partir de la trace. Les connexions qui ne sont pas marquées sont ré-exécutées normalement.

Le tableau d'identifiants de *socket* est utilisé pour marquer les échanges de données avec des nœuds corrects. Lorsque `tcpRecv` utilise une *socket* marquée, nous enregistrons les données reçues, la valeur de retour, ainsi que la valeur du compteur. Dans le cas de `tcpSend`, seules la valeur de retour et la valeur du compteur doivent être tracées.

```
1  int PartialRecordTcpConnect(tcp_socket_t s, node_id *num, int retTcpConnect) {

2  MessageProduction(num,buffers,&size);
3  tcpRecv(s,bufferr,size);
4  tcpSend(s,buffers,size);
5  if (CorrectNode(bufferr)) {
6     RegisterNodeSocket(bufferr,s);
7     GeneratePartialTcpConnectTraceData(traceData,CFC,retTcpConnect);
8     trace(PartialTcpConnectTraceType,traceData,PartialTcpConnectTracSize,tcpConnectVector);
9  }
10 CFC++;
11}
```

FIGURE 5.10 – Traçage de tcpConnect().

La fonction `PartialRecordTcpConnect`, présentée sur la figure 5.10 implémente le traçage de `tcpConnect`. D'abord, l'identifiant du nœud distant est reçu

en utilisant `tcpSend` (ligne 3). Lorsque ce nœud est correct (ligne 5), la *socket s* est enregistrée dans un tableau en utilisant `RegisterNodeSocket(bufferr,s)` (ligne 6). Finalement, la valeur de retour de `tcpConnect` (`retTcpConnect`) ainsi que la valeur du compteur d'appels de fonctions de communication (`CFC`) sont tracées (ligne 7,8).

La fonction `tcpAccept` utilise un mécanisme similaire d'enregistrement de données pour la ré-exécution partielle. Il s'agit de tracer sa valeur de retour ainsi que la valeur du compteur et d'enregistrer la socket créée dans un tableau.

```
int PartialRecordTcpRecv(tcp_socket_t s, int retTcpRecv, char *buffer) {

if (registredSocket(s)) {
   GeneratePartialTcpRecvTraceData(traceData,CFC,retTcpRecv,buffer);
   trace(PartialTcpRecvTraceType,traceData,PartialTcpRecvTraceSize,tcpRecvVector);
}
CFC++;
}
```

FIGURE 5.11 – Traçage de tcpRecv().

La figure 5.11 représente la fonction `PartialRecordTcpRecv` qui trace `tcpRecv`. La fonction `registredSocket(s)` recherche la *socket* utilisée par tcpRecv() parmi celles qui définissent un lien de communication avec un nœud correct, enregistrées dans le tableau durant le traçage de `tcpConnect` et de `tcpAccept`. Lorsque le nœud distant est correct, `GeneratePartialTcpRecvTraceData` et `trace` tracent la valeur de retour de `tcpRecv`, les données reçues dans la variable `buffer` et le compteur d'appels (`CFC`).

5.5.2 Ré-exécution d'une partie des unités d'exécution

Afin de ré-exécuter les nœuds supposés fautifs en isolation, nous nous basons sur les données enregistrées. Avant chaque opération de communication (`tcpConnect(), tcpAccept, tcpRecv() et tcpSend()`), nous vérifions si la communication est avec un nœud supposé correct. Si c'est le cas, l'appel n'est pas effectué mais le résultat de l'appel est récupéré dans la trace. Si c'est une communication entre deux nœuds dans l'ensemble qui est débogué, la communication est ré-exécutée en appelant effectivement la fonction correspondante. Pour distinguer les nœuds fautifs des nœuds corrects, nous utilisons la fonction `IsCorrectNode` présentée sur la figure 5.12. Cette fonction utilise le compteur des appels de communication (*CFC*). Quand la valeur du compteur correspond à une valeur tracée (nextCFC), la communication correspondante est faite avec un nœud correct.

```
int IsCorrectNode() {
...
if (CFC == nextCFC) {
  GetNextCommunicaitonCounterFromTrace(&nextCFC);
  CFC++;
  return TRUE;
}
else {
  CFC++;
  return FALSE;
}
}
```

FIGURE 5.12 – Détection de communication avec les *nœuds correct*.

5.5.3 Ré-exécution limitée dans le temps

Pour limiter la ré-exécution de nœuds fautifs dans le temps, nous utilisons un intervalle borné par deux lectures des traces de ré-exécution déterministe. En effet, les développeurs analysent les traces de ré-exécution déterministe et sélectionnent deux entrées dans ces traces qui bornent l'intervalle d'exécution supposé anormal. Le débogage est effectué entre les deux lectures de ces entrées durant l'exécution partielle.

Pour détecter le début et la fin de l'intervalle et permettre le débogage uniquement pendant cet intervalle, nous avons apporté une extension à GDB qui implémente un nouveau type de points d'arrêt de GDB. Nous les appelons des points d'arrêt de ré-exécution (replay-breakpoints), déclenchés lorsqu'une des deux bornes de l'intervalle est atteinte.

L'extension GDB prend en entrée les deux bornes de l'intervalle à déboguer. Ces bornes correspondent à deux entrées dans la trace et sont identifiées d'une manière unique en utilisant le triplet :

- Identifiant de nœud.
- Identifiant de tâche.
- Estampille temporelle.

Durant l'exécution partielle, notre extension reçoit successivement les triplets qui correspondent à chaque appel des fonctions de notre API. Pour cela, nous avons modifié GDB afin de mettre un point d'arrêt sur `gdbNotify(debugTrace)`, appelée suite à chaque exécution de fonction de notre API. L'argument de `gdbNotify` est une structure partagée (*debugTrace*) entre le logiciel et GDB qui contient une description de la dernière fonction qui est ré-exécutée (identifiant de nœud, identifiant de tâche et estampille temporelle). Lorsque ce point d'arrêt est déclenché, nous comparons la structure (*debugTrace*) aux bornes de l'intervalle à déboguer qui donne lieu à trois possibilités :

- *debugTrace* ne correspond par aux bornes : l'exécution se poursuit.
- *debugTrace* correspond à la borne inférieure : l'exécution est suspendue et le débogage peut commencer.
- *debugTrace* correspond à la borne supérieure : l'exécution a atteint la fin de l'intervalle de temps à déboguer. Les développeurs ont la possibilité de sélectionner un nouvel intervalle. Lorsque cet intervalle est après l'ancien, l'exécution continue. Lorsque le nouvel intervalle est avant l'ancien, exécution est relancée.

5.6 Visualisation de traces d'exécution

Afin de visualiser les traces de ré-exécution déterministe, nous nous sommes basés sur le format de traces Pajé [SdKB00, dOSdKM02], ainsi que l'outil KPTrace [STM09a]. Le format Pajé permet d'organiser des données dans une hiérarchie de type arborescence. Les nœuds sont appelés *conteneur* et les feuilles *entités*. Plusieurs types d'entités sont définis dans Pajé mais nous n'en utilisons que deux, appelées *liens* et *événements*.

- *Liens* : sont des actions qui débutent sur un *conteneur* à une date et se terminent sur un autre *conteneur* à une autre date. Par exemple, les tâches peuvent être représentées par des *conteneurs* et deux entrées successives dans une section critique par ces tâches comme un *lien*.
- *Événements* : sont des actions instantanées sur un *container* à une date donnée. Par exemple, un *événement* peut être l'appel d'une fonction par une tâche qui est le *container*.

KPTrace [STM09a] est un outil d'observation du logiciel embarqué s'exécutant sur les systèmes d'exploitation Symbian [Com10] et les distributions Linux de STMicroelectronics [STM09b]. Cet outil fait partie d'une suite d'outils de débogage, traçage, analyse et visualisation de STMicroelectronics appelé STWorkbench [STM10]. KPTrace permet de visualiser dans une base de temps, un format de traces propriétaires qui contient un type d'événement, une estampille, un identifiant du processus ou du thread système et une suite variable d'arguments.

Nous avons conçu un script qui transforme post-mortem chaque entrée dans nos traces sous le le format Pajé. Ensuite, nous avons modifié KPTrace afin de prendre en compte ce format.

Nous avons choisi le format de traces Pajé à cause de sa polyvalence qui lui permet d'organiser un large spectre de données utilisateur. Nous avons défini la sémantique suivante pour représenter nos traces sous le format Pajé :

- *Containers* : le *container* racine de la structure arborescente est l'identifiant de l'exécution du logiciel à mettre au point. Les *containers* fils sont les identifiants des nœuds de la plate-forme qui participent au calcul. Ces *containers* nœuds ont également des fils qui sont les identifiants des tâches.
- *Liens* : nous représentons par des *liens* deux relations de précédence durant l'exécution. La première est entre deux accès successifs par deux tâches à une même variable de synchronisation. La deuxième est entre les tâches qui exécutent `tcpConnect()` et `tcpAccept()` afin de créer une socket.
- *Événements* : nous utilisons les *événements* afin de représenter les lectures des données des périphériques ainsi que la réception de données de communication.

```
Type de l'événement  Date                 Nom  Container  Valeur
20                   8.8830950000000e+06  DP   T1         12288
```

FIGURE 5.13 – Représentation d'une *entité* de type *événement*

La figure 5.13 représente une entrée dans la trace de ré-exécution déterministe sous le format Pajé. Cette entrée correspond à une lecture des données d'un périphérique identifié par la valeur 20. Cet accès est effectué par la tâche *T1* à la date *8.8830950000000e+06*. La taille des données lues est de 12288 Ko.

```
Type début de lien   Date                 Nom   Container   Container fils   Valeur
42                   8.9068630000000e+06  L_N   N1          T3               3456

Type fin de lien     Date                 Nom   Container   Container fils   Valeur
43                   8.9069510000000e+06  L_N   N1          T1               3457
```

FIGURE 5.14 – Représentation d'une *entité* de type *lien*.

Sur la figure 5.14, nous représentons les accès successifs par deux tâches *T3* puis *T1* à une variable de synchronisation. La figure montre les deux dates d'accès, le *container* père de *T1* et de *T3* ainsi que les valeurs des horloges de Lamport.

Nous avons modifié le code de KPTrace afin de visualiser nos traces. Ces modifications consistent notamment à écrire un nouvel analyseur syntaxique pour interpréter le format de traces Pajé. Ensuite, nous avons adapté les fonctions d'affichage KPTrace pour représenter nos traces dans un espace bidimensionnel spécifique à chaque nœud. Nous représentons le temps sur l'abscisse et les tâches sur l'ordonnée. Les *liens* sont représentés par une flèche et les *événements* par un drapeau.

Sur la figure 5.15 nous avons représenté deux *liens* par deux flèches et deux accès aux données d'un périphérique par des drapeaux. Les *liens* correspondent à

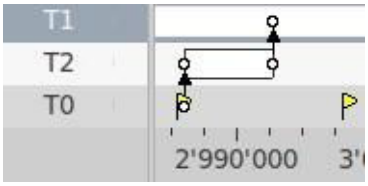


FIGURE 5.15 – Visualisation des *liens* et des *événements* avec KPTrace.

trois accès successifs à une variable de synchronisation par les tâches *T1, T2, T3*. Les deux drapeaux correspondent aux accès des données d'un périphérique. La couleur des drapeaux est spécifique aux périphériques.

5.7 Synthèse

Dans ce chapitre, nous avons présenté ReDSoC, un ensemble de quatre outils qui permettent d'appliquer notre méthodologie de mise au point MPSoC. Ces outils s'interfacent avec un API qui représente les problématiques de mise au point, ainsi que les caractéristiques de la plate-forme MPSoC. Pour mettre au point une MPSoC qui ne dispose pas de cette API, il suffit d'effectuer l'interfaçage correspondant entre ReDSoC et la plate-forme.

L'architecture de ReDSoC que nous avons définie permet de déporter une partie des ressources de débogage sur une plate-forme de développement. En conséquence, nous avons réduit les besoins en ressources de débogage sur la MPSoC.

Le premier outil que nous avons implémenté permet la collecte de traces. Cet outil est une interface de fonctions. Ces fonctions sont configurables pour un contexte spécifique (architecture et logiciel) afin de permettre l'utilisation sur une large diversité de plates-formes MPSoC.

L'outil de ré-exécution déterministe permet de reproduire les accès aux variables de synchronisation, la communication et les accès aux périphériques. Cet outil implémente les méthodes de ré-exécution déterministe choisies.

Nous avons implémenté notre méthode de ré-exécution partielle dans un troisième outil. D'une part, cet outil ré-exécute la communication entre nœuds exécutés et non-exécutés d'une manière transparente. D'une autre part, nous avons implémenté la détection d'un intervalle de temps durant l'exécution en utilisant un nouveau type de points d'arrêt dans GDB (*replay-breakpoints*).

Le dernier outil représente visuellement les traces de ré-exécution déterministe. Cet outil aide au développeur d'avoir une vision globale du comportement du programme et éventuellement de choisir un intervalle de temps où le comportement de

l'application est anormal. D'une part, nous avons implémenté un script qui transforme les traces de ré-exécution déterministe sous le format Pajé. D'autre part, nous avons modifié l'outil de visualisation développé dans notre groupe KPTrace pour prendre en charge ce format Pajé.

Troisième partie

Validation

Chapitre 6

Étude de cas sur architecture embarquée

Dans ce chapitre, nous validons notre prototype ReDSoC en présentant une étude de cas de débogage d'erreur non-déterministe. Pour ce faire, nous travaillons sur une architecture embarquée et étudions une application temps réel.

Dans la suite, nous présentons d'abord l'architecture matérielle, avant de détailler l'implémentation de notre API et le déploiement de nos outils. Nous présentons ensuite le débogage d'une erreur non-déterministe.

6.1 Architecture matérielle MPSoC

L'architecture matérielle que nous avons utilisée est conçue par la société Gumstix [Gum]. La figure 6.1 représente cette architecture, composée d'une carte d'extension (*expansion board*) Stagecoach et de deux nœuds de calcul (*computer-on-module*) Overo FE COM. Chacun de ces nœuds est composé d'un processeur ARM Cortex-A8 cadencé à 600MHz, de 256Mo de mémoire DDR RAM de faible consommation énergétique, de 256Mo de mémoire NAND flash et d'un port d'extension microSD pour rajouter de la mémoire flash.

La carte d'extension comprend 7 ports d'extension (*slots*) permettant de connecter 7 nœuds. Dans la configuration que nous avons utilisée, les deux nœuds occupent respectivement le premier et le troisième port d'extension de la carte. La connexion entre les nœuds est faite en utilisant un commutateur réseau (switch Ethernet) intégré, de capacité 100Mbit/s, qui affecte une adresse IP unique à chaque nœud. La carte d'extension comprend également trois connectiques : un connecteur réseau RJ45, un port USB OTG [OTGH] par nœud et un port USB de console. Le connecteur RJ45 établit un lien entre une carte réseau externe et

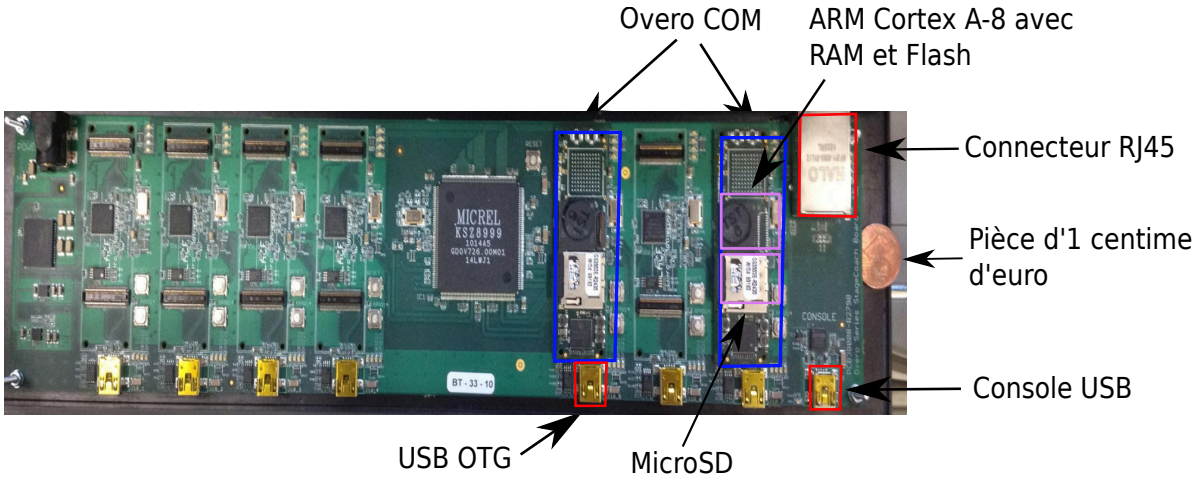


FIGURE 6.1 – Carte Stagecoach avec deux nœuds Overo FE COM

le commutateur de la carte. Il permet d'adresser les différents nœuds en utilisant leurs adresses IP depuis l'extérieur. Le port USB OTG permet de connecter un large ensemble de périphériques USB (clavier, souris, caméra, mémoire flash ...) à un nœud de calcul. Le port de console USB permet d'établir une communication série avec un seul nœud.

Les plates-formes MPSoC basées sur Stagecoach sont destinées à être utilisées dans des domaines embarqués qui exigent une faible consommation énergétique, des dimensions réduites et des performances élevées. Elles sont déjà utilisées par exemple, dans le domaine de l'aéronautique [Sam11] ou dans le projet Sandia's MegaTux qui utilise 196 modules de calcul dans 25 cartes d'extension [proa].

6.2 Implémentation de l'API MPSoC

Le logiciel fourni avec l'architecture embarquée comprend une version spécifique de Linux 2.6 qui intègre la bibliothèque *libc*. Une instance du noyau Linux est utilisée séparément sur chacun des deux nœuds. Nous avons implémenté les fonctions de notre API MPSoC principalement en encapsulant des fonctions de la *libc*.

Sur la figure 6.2, nous présentons la pile logicielle que nous utilisons. Dans la suite, nous présentons quelques différences entre notre API et les interfaces de Linux et *libc*, ainsi que les simplifications que nous avons considérées.

- *Gestion de tâches et synchronisation* : les fonctions de gestion de tâches et de synchronisation entre les tâches encapsulent l'API POSIX threads.

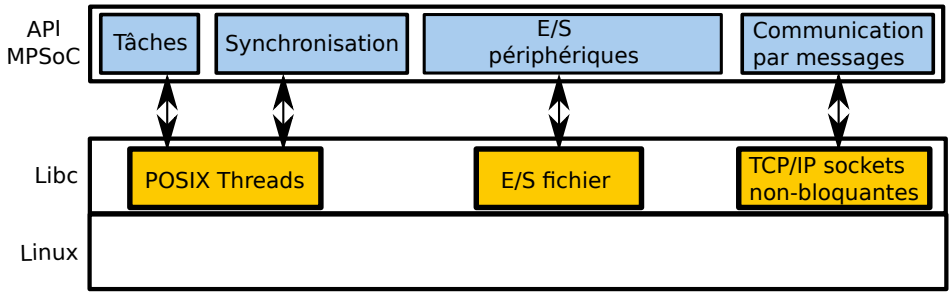


FIGURE 6.2 – Pile logicielle utilisée par l'API MPSoC

Dans la fonction de notre interface `taskCreate(task_t *task,void *(*funct)(void *),void *args, int node)` (cf. chapitre 5, section 5.2), l'argument `node` qui spécifie le nœud sur lequel la tâche s'exécute n'est pas utilisé. En effet, la tâche s'exécute par défaut sur le nœud courant. Cette solution nous permet d'utiliser directement l'API POSIX Threads. Sur chaque nœud, le processus Linux lancé est considéré comme étant la première tâche.

- *E/S périphériques* : les entrées du clavier sont accédées en encapsulant dans les fonctions d'E/S de notre API (`pOpen, pRead, pClose`), les fonctions de lecture/écriture sur un fichier. Ce fichier de périphérique d'entrée par caractère correspond au clavier.

- *Communication par messages* : les fonctions de communication par messages encapsulent les primitives de la *libc* qui permettent la gestion de sockets TCP/IP bloquantes et non-bloquantes. Les principales différences entre nos fonctions de communication et celles fournies par la *libc* sont les identifiants des points communicants. Notre API utilise des identifiants de nœuds à la place d'adresses IP pour établir un lien de communication. Nous avons donc mis en place un tableau de correspondances entre les identifiants des nœuds et leurs adresses IP. Ce tableau est rempli statiquement dans le code des fonctions `tcpListen` et `tcpConnect`.

6.3 Déploiement de la plate-forme

Dans le processus de déploiement, nous avons configuré d'une part, le logiciel afin d'exploiter l'architecture, et d'autre part, notre solution de mise au point ReDSoC. Nous détaillons ces configurations dans la suite.

6.3.1 Configuration de la plate-forme

Ayant eu à notre disposition uniquement le matériel, nous avons commencé par installer un ensemble de logiciels. Nous avons d'abord créé une carte MicroSD amorçable avec l'image d'une distribution Linux pour l'embarqué Angstrom [Dis]. Cette distribution comprend un noyau Linux 2.6, la *libc6*, un système de fichiers racine, client/serveur ssh, etc.

Nous avons utilisé un compilateur croisé de la chaîne d'outils Sourcery CodeBench Lite Edition [Gra] pour produire des exécutables pour ARM Cortex-A8 sur une plate-forme x86. Ces exécutables sont l'application à déboguer, les outils de ReDSoC, et le serveur GDB.

Afin d'optimiser les coûts d'écriture des traces sur un support de stockage, nous avons créé un système de fichiers sur la mémoire vive. Les traces sont écrites sur ce système de fichiers durant l'exécution. Post-mortem, nous les transférons par ssh sur la plate-forme de développement pour leur visualisation.

6.3.2 Configuration de ReDSoC pour la plate-forme

Le déploiement de ReDSoC sur la plate-forme d'exécution nécessite de configurer chacun de nos outils de mise au point.

- *Outil de collecte de traces* : comme nous l'avons présenté dans le chapitre 5 précédent, cet outil fournit une interface de quatre fonctions qui sont implémentées spécifiquement pour une plate-forme donnée. La première fonction `getTimestamp` fournit une estampille temporelle au format du visualiseur KPTrace. Cette estampille est obtenue par lecture du registre de l'horloge du processeur ARM. Ce registre est un compteur de cycles que nous convertissons en millisecondes afin de respecter le format de KPTrace. La deuxième fonction `getNodeId` fournit l'identifiant du nœud courant à partir du fichier de configuration. La fonction `getTaskId` qui fournit l'identifiant de la tâche, utilise `pthread_self`. Cette fonction fournit un identifiant unique pour chaque thread, que nous représentons par des entiers dans un tableau de correspondances. La dernière fonction `trace` enregistre les données de ré-exécution déterministe et partielle respectivement dans deux fichiers différents sur le système de fichiers du nœud.
- *Outil de ré-exécution déterministe* : dans notre modèle de ré-exécution déterministe nous avons supposé que le contenu des entrées de l'application est disponible durant la ré-exécution déterministe. Dans le cas de l'application que nous allons utiliser, les données d'entrée sont les valeurs associées aux touches du clavier et au temps fourni par l'horloge interne. Nous avons donc inclus une fonction qui enregistre la séquence de ces données dans un fichier

spécifique. Durant la ré-exécution déterministe, les données enregistrées sont lues séquentiellement.

- *Outil de ré-exécution partielle* : nous avons intégré notre extension de GDB afin de détecter l'intervalle de temps à mettre au point. Nous avons attaché une instance séparée du serveur GDB sur chacun des exécutables des deux nœuds. Sur une plate-forme de développement, nous avons exécuté GDB qui inclut notre extension afin de se connecter par TCP/IP avec chacun des serveurs GDB. En conséquence, deux instances de GDB sont nécessaires pour mettre au point simultanément les deux nœuds de la plate-forme.
- *Outil de visualisation de traces* : cet outil est exécuté sur la plate-forme de développement. Les traces sont en effet téléchargées à partir des nœuds en utilisant une connexion `ssh` entre les nœuds et la plate-forme de développement.

6.4 Débogage d'un jeu d'arcade

Nous présentons le débogage d'une erreur non-déterministe d'exécution sur une application du jeu de Tetris. Nous présentons successivement l'implémentation de l'application en utilisant notre API MPSoC, l'erreur d'exécution non-déterministe, ainsi que son débogage.

6.4.1 L'application Tetris à 2 joueurs

Le jeu d'arcade que nous avons utilisé est un clone de Tetris [Nil] avec un support pour deux joueurs. Cette application est de taille d'environ 0.7Mo et comprend environ 15000 lignes de code.

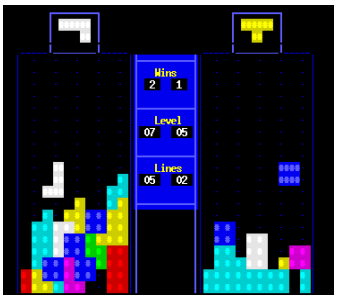


FIGURE 6.3 – Capture d'écran d'un des deux joueurs de Tetris.

La figure 6.3 représente le plateau d'un des deux joueurs de Tetris au cours du jeu. La grille du joueur est à gauche et celle de son adversaire à droite. On

peut notamment voir que le joueur voit en temps réel les déplacements des pièces de l'adversaire. Un joueur peut compliquer le jeu de l'adversaire en supprimant plusieurs lignes simultanément. Cette suppression fait apparaître des pièces sur la grille de l'adversaire. Le perdant est celui dont le plateau déborde plus rapidement.

Chaque instance du jeu est représentée par une tâche (processus Linux). Ces tâches utilisent notre API pour communiquer et pour accéder aux données du clavier. Pour ce faire, nous avons remplacé les appels des fonctions de la *libc* d'accès aux périphériques et de communication par sockets par celles de notre API MPSoC.

La disponibilité d'entrées du clavier est vérifiée en continu en utilisant `pRead`. Les entrées du clavier peuvent déplacer la pièce sur l'horizontale, la retourner ou accélérer son déplacement vertical.

L'horloge interne est utilisée pour respecter la fréquence des déplacements verticaux des pièces. En effet, une itération de la boucle du jeu s'exécute beaucoup plus rapidement que la fréquence de déplacement des pièces. Entre deux déplacements, l'application vérifie donc la disponibilité des entrées du clavier jusqu'à ce que la fréquence soit atteinte.

Afin de ré-exécuter d'une manière déterministe cette application, il est nécessaire de reproduire ses entrées : les lectures des valeurs de l'horloge et du clavier. Nous avons donc modifié l'application afin d'enregistrer dans un fichier la succession de ces valeurs.

Les deux instances du jeu s'échangent les informations sur les touches appuyées pour mettre à jour le plateau de l'adversaire. En effet, chaque touche appuyée dans une des instances du jeu est envoyée par `tcpSend`. La réception, faite par `tcpRecv` ne bloque pas le jeu et les déplacements des pièces du joueur.

6.4.2 Débogage

Nous avons demandé à des ingénieurs dans notre groupe d'introduire artificiellement une erreur dans l'application (correcte) de jeu de Tetris. Le défi a été de voir si nous pourrions la découvrir en utilisant ReDSoC.

Après l'introduction de l'erreur, nous avons observé que de temps en temps le jeu se terminait avec une erreur de division par zéro. En effet, une des deux instances se terminait à cause de cette erreur et l'autre instance était par conséquent déclarée vainqueur.

Afin de mettre au point cette application, nous l'avons exécutée sous le contrôle de ReDSoC jusqu'à l'occurrence de l'erreur que nous avons enregistrée. Dans cette exécution, nous avons appelé les deux nœuds respectivement Nœud1 et Nœud2. Le nœud qui s'est terminé avec l'erreur de division par zéro est Nœud1.

La première étape de la mise au point consiste à analyser l'exécution afin d'identifier un comportement anormal. Nous savons déjà que le Nœud1 se comporte de façon anormale due à l'erreur de division par zéro. Nous allons essayer de mieux localiser l'erreur en visualisant les traces de ré-exécution déterministe.

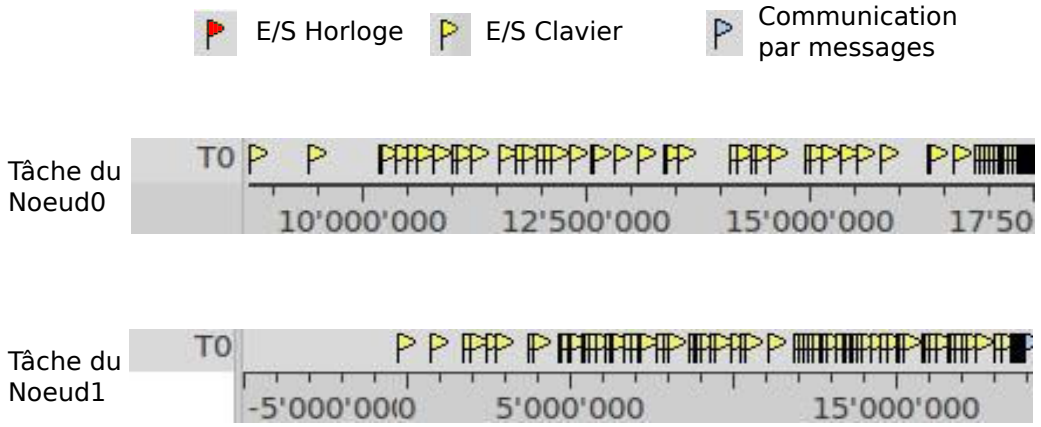


FIGURE 6.4 – Visualisation des traces de ré-exécution déterministe.

La figure 6.4 montre la visualisation par KPTrace des traces de ré-exécution déterministe. Ces traces sont visualisées dans deux référentiels temporels différents. Ces référentiels correspondent respectivement aux deux horloges internes des processeurs ARM dans les deux nœuds. Le drapeau rouge correspond à la lecture de l'horloge interne pour respecter la fréquence des déplacements verticaux des pièces. Le drapeau jaune correspond aux lectures des touches appuyées par les joueurs. Les drapeaux bleus représentent les touches appuyées par l'adversaire et reçues par le réseau. Ces traces sont en effet comprimées pour visualiser toute la durée de l'exécution sur une fenêtre. On a donc l'impression de voir uniquement des drapeaux jaunes mais en agrandissant une région, on peut voir les autres drapeaux.

Puisque l'exécution se termine suite à une erreur, nous analysons les traces en fin d'exécution. Dans une première observation, nous constatons qu'il y a une irrégularité sur une région à la fin de la trace. Cependant, cette application doit avoir un comportement similaire entre deux périodes temporelles courtes qui correspondent au déplacement vertical d'une pièce. Ce comportement attendu ne correspond pas à celui observé sur la trace, où on peut clairement distinguer une modification importante du comportement à partir d'un point proche de la fin de l'exécution. En conséquence, nous agrandissons cette région, montrée sur la figure 6.5 afin de mieux l'analyser. Cette région correspond à onze drapeaux régulièrement espacés. Nous agrandissons la région autour du dernier drapeau afin d'analyser le comportement juste avant l'erreur. Nous observons trois lectures de l'horloge, suivies de trois lectures de touches du clavier et une réception de données.

Au vu des analyses effectuées, nous avons décidé de sélectionner le Nœud1

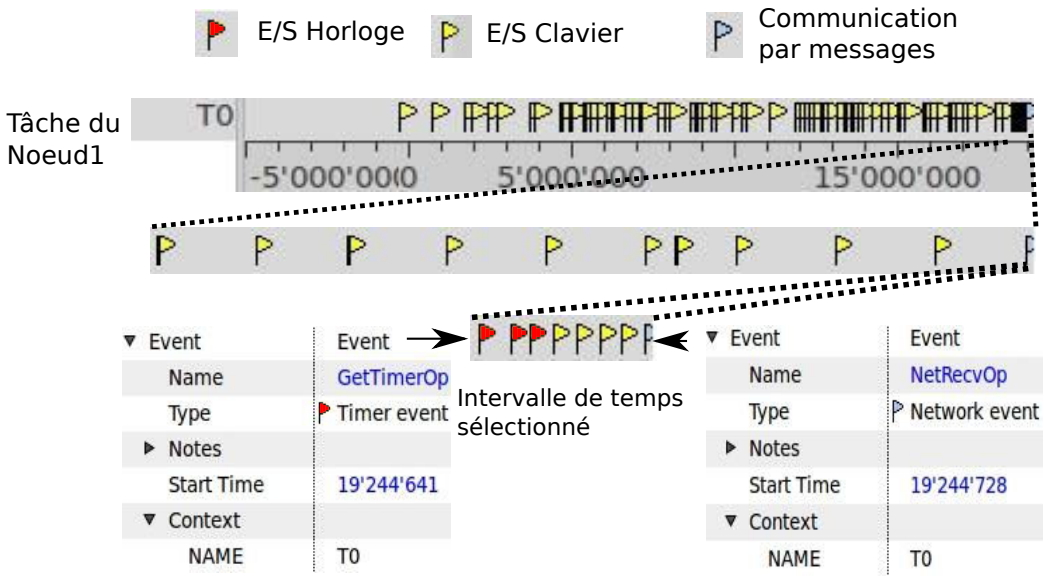


FIGURE 6.5 – Analyse des traces et sélection d'un intervalle de temps à déboguer.

et l'intervalle de temps montré sur la figure 6.5. Cet intervalle est délimité par deux événements qui représentent respectivement une lecture de l'horloge interne à gauche et une réception des données par le réseau à droite. On peut notamment observer les différents champs de l'événement de lecture de l'horloge :

- Name : nom de la fonction `GetTimerOp` qui récupère des valeurs de l'horloge.
- Type : les données récupérées sont de type *Timer event*.
- Start Time : estampille temporelle au moment du traçage.
- Context Name : la tâche *T0* qui représente le conteneur parent dans le format de traces Pajé que nous utilisons.

La sélection du Nœud1 à déboguer consiste à mettre à jour le fichier de configuration de ReDSoC. Dans ce fichier, on indique le numéro du Nœud1 à être exécuté en séparation ainsi qu'une identification des deux événements qui bornent l'intervalle de temps. Cette description comprend le triplet (numéro du nœud, numéro de tâche et estampille temporelle en microsecondes). On peut remarquer que les opérations effectuées par plusieurs processeurs en même temps correspondront à plusieurs entrées dans la trace, qu'on pourrait distinguer par le numéro de la tâche, spécifique à un processeur donné. Ces informations sont montrées sur la fenêtre de propriété des drapeaux KPTrace. Les deux bornes sont respectivement (1, 0, 19.244.641μs) et (1, 0, 19.244.728μs).

Afin d'utiliser GDB pour l'intervalle de temps sélectionné, les outils de ReDSoC préparent l'exécution. D'abord, une ré-exécution déterministe est faite auto-

matiquement afin de collecter les traces de ré-exécution partielle. Ensuite, ReD-SoC lance l'exécution partielle en activant GDB et en définissant deux *replay-breakpoints* correspondants aux deux bornes de l'intervalle. Lorsque l'exécution est suspendue, l'activité de débogage commence.

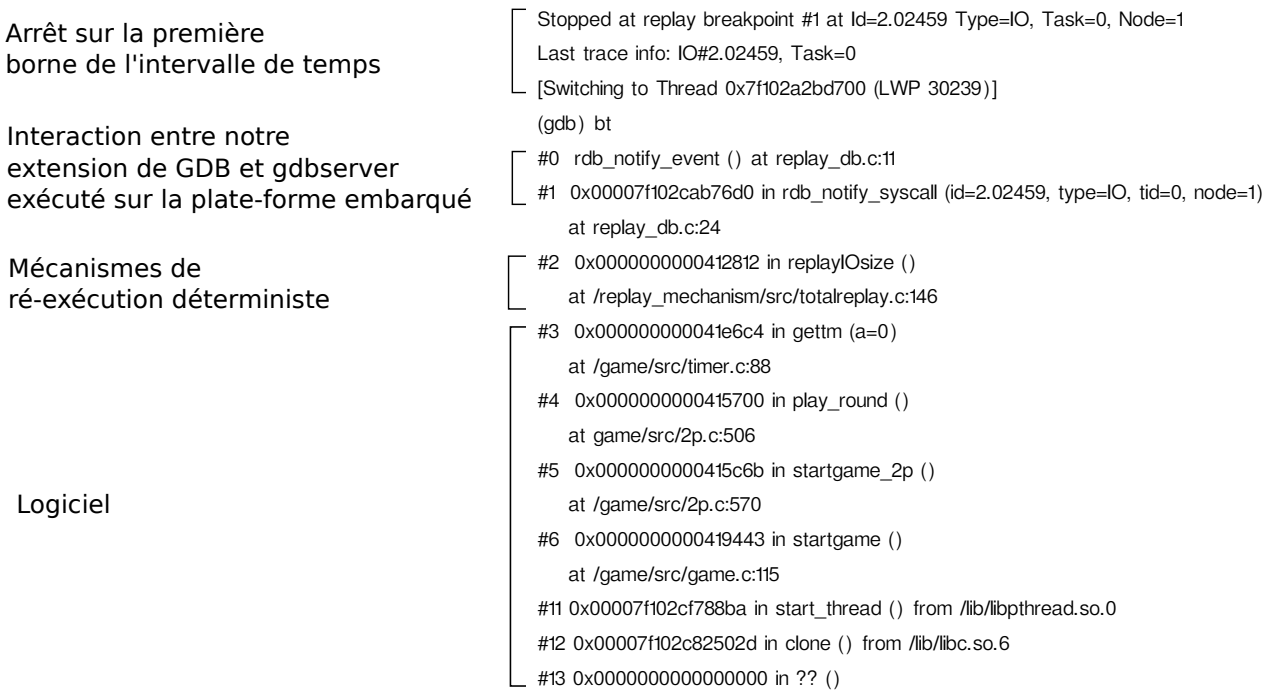


FIGURE 6.6 – Activité de débogage sur une partie de l'exécution.

Sur la figure 6.6, nous montrons une capture d'écran de la suspension de l'exécution lorsque la première borne de l'intervalle est atteinte. La première ligne de cette capture montre l'identifiant de la borne, composé de quatre champs. Le premier champ indique le numéro de l'entrée dans la trace de ré-exécution déterministe (202459). Le deuxième champ représente le type de l'entrée dans la trace (E/S périphérique). Les deux derniers champs correspondent respectivement aux identifiants de la tâche (Task=0) et du nœud (Node=1). Sur la quatrième ligne ((gdb) bt) de la capture d'écran, nous montrons la pile des appels de fonctions lors de la suspension de l'exécution. Dans cette pile, nous observons d'abord l'interaction entre notre extension de GDB et serveur GDB par la fonction `rdb_notify_event`. Cette fonction informe GDB sur l'avancement de la ré-exécution déterministe. Plus particulièrement, on peut observer que notre mécanisme de ré-exécution déterministe reproduit une opération d'entrées (type=IO), de numéro (id=202459) par la tâche (tid=0) et sur le nœud (node=1). Ensuite, nous observons la fonction de ré-exécution déterministe des E/S périphériques (`replayIOsize`) suivie des appels des fonctions du logiciel.

L'analyse du comportement anormal procède d'une manière standard. D'abord nous avons utilisé un point d'arrêt sur la dernière fonction de réception de données avant l'erreur. En analysant l'exécution pas à pas nous avons observé que la valeur reçue est incorrecte (0) d'où l'erreur de division par zéro. Puisque cette valeur erronée est en provenance du nœud Nœud0, nous avons décidé d'analyser sa trace.

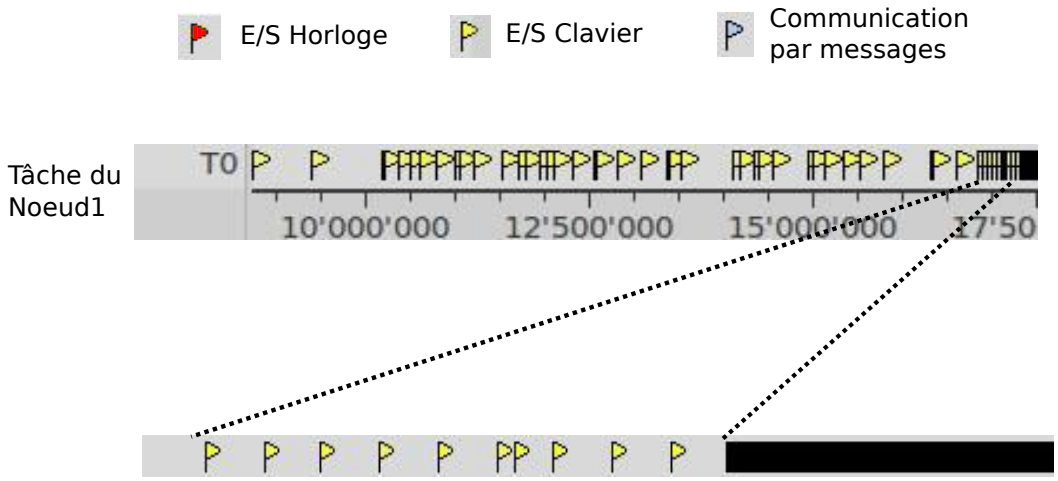


FIGURE 6.7 – Analyse des traces et sélection de deuxième intervalle de temps à déboguer.

La trace du nœud Nœud0 est montrée sur la figure 6.7. Nous observons que vers la fin de l'exécution la régularité de l'exécution n'est plus respectée. En agrandissant, nous découvrons dix drapeaux régulièrement espacés suivis d'un carré noir. Ce carré noir représente en réalité, une multitude de drapeaux de communication ayant une fréquence importante. Cette partie de la trace correspond à la détection de la déconnexion du nœud distant (Nœud1). Puisque nous voulons analyser la valeur erronée de la dernière entrée du clavier nous sélectionnons l'intervalle de temps à partir de l'irrégularité de l'exécution jusqu'à la déconnexion.

Lorsque l'intervalle de temps à déboguer est atteint, nous avons mis un point d'arrêt sur la lecture des entrées du clavier. Ce point d'arrêt s'est activé dix fois. La dixième fois, le tampon en mémoire qui contient les entrées du clavier a été saturé. En conséquence, la valeur de la touche du clavier, envoyée vers le nœud distant à partir du tampon mémoire saturé a été incorrecte.

Nous avons débogué quelques fois cet intervalle de temps et rapidement découvert la cause de l'erreur. En effet, la saturation du tampon mémoire est due à un traitement erroné des événements du clavier lorsqu'une touche est continuellement appuyée. Cet appui continuait à générer des événements du clavier d'une fréquence trop importante, d'où la saturation du tampon mémoire.

6.5 Synthèse

Dans ce chapitre, nous avons montré l'utilisation de ReDSoC pour le débogage d'une application de jeu sur plate-forme embarquée. Nous avons utilisé une version de Tetris à deux joueurs et avons utilisé une architecture basée sur la carte Stagecoach à deux nœuds de calcul.

La mise en place de la plate-forme a présenté un effort technique, notamment à cause de l'adaptation des logiciels standard (noyau Linux, GDB, compilateurs etc.), ainsi que de ReDSoC pour l'architecture ARM. Cependant, le portage du jeu de Tetris qui consistait à interfacer les mécanismes de communication et d'acquisition des entrées avec notre API n'a pas présenté de difficulté particulière.

Nous avons identifié facilement une erreur qui survenait de manière non-déterministe à cause des E/S clavier et des communications réseau. Pour ce faire, la visualisation de traces nous a d'abord permis d'identifier une partie de l'exécution erronée à déboguer. Ensuite, nous avons utilisé nos mécanismes de ré-exécution déterministe et partielle afin de reproduire l'exécution sur chacun des deux nœuds en séparation. Enfin, nous avons utilisé le débogage GDB sur deux périodes spécifiques de l'exécution autour de l'occurrence de l'erreur.

Nous avons validé notre méthodologie de mise au point, ainsi que ReDSoC sur une plate-forme à deux nœuds totalisant deux processeurs. Dans le chapitre suivant, nous étudions notre solution sur une plate-forme à plusieurs nœuds et possédant un nombre plus important de processeurs.

Chapitre 7

Étude de cas sur plate-forme NUMA

Ce chapitre a pour objectif de valider le passage à l'échelle de notre méthodologie de mise au point. Nous expérimentons avec la mise au point d'une application complexe (parallèle, distribué et avec des contraintes temporelles) exécuté sur une plate-forme multi-processeurs. L'application est basée sur les bibliothèques FFMPEG [Tom06] utilisées par une large communauté pour la conception d'applications de traitement de flux multimédia [Prob]. Nous avons défini les caractéristiques d'une plate-forme MPSoC qui permettrait d'exécuter efficacement notre application. Puisque nous n'avons pas eu à notre disposition une telle plate-forme MPSoC, nous en avons virtualisée une à partir d'une architecture NUMA (Non-Uniform Memory Access). Cette architecture a 32 cœurs dans quatre processeurs et est basée sur Linux. Plus particulièrement, nous avons virtualisé les nœuds MPSoC en utilisant des processus Linux que nous avons restreint à exploiter les ressources des nœuds NUMA spécifiques.

Nous commençons par présenter la plate-forme utilisée. Ensuite, nous montrons le déploiement des logiciels sur la plate-forme. Enfin, nous discutons la mise au point d'une erreur non-déterministe.

7.1 De NUMA vers MPSoC

L'indisponibilité de plate-forme MPSoC réelle à grande échelle pour la validation de notre méthodologie de mise au point nous a laissés face à un choix entre deux possibilités. Le premier choix consistait à utiliser une plate-forme MPSoC simulée. Cependant, les solutions existantes comme QEMU [Bel05] ordonnancent les processeurs simulés sur un processeur réel et peuvent donc *cacher* des erreurs non-déterministes.

Le deuxième choix, que nous avons pris, est de virtualiser une plate-forme

MPSoC en utilisant une architecture avec plusieurs processeurs réels. Sur des processeurs réels, le code est exécuté en concurrence et permet donc l'occurrence d'un large ensemble d'erreurs comme les accès non-synchronisés à la mémoire comme les blocages infinis. Nous avons choisi de virtualiser une plate-forme MPSoC, aux caractéristiques annoncées dans le chapitre 4, section 4.3, à partir d'une architecture NUMA afin de profiter des multiples processeurs physiques. Comme les MPSoC ciblées, les architectures NUMA utilisent une organisation hiérarchique des processeurs et des bancs mémoire dans un ensemble de nœuds interconnectés afin d'accroître les performances. Nous avons donc considéré que l'architecture NUMA représentait un choix pertinent afin de valider le passage à l'échelle de notre solution.

Une architecture NUMA est caractérisée par un ensemble de groupes d'unités de calcul (processeurs/cœurs) disposant de leur propre mémoire et, éventuellement, de leurs propres canaux d'E/S. Toutefois, chaque unité de calcul peut accéder à la mémoire associée aux autres groupes, et ce de façon cohérente. Chaque groupe est appelé nœud NUMA. Le nombre d'unités de calcul par nœud NUMA varie en fonction des caractéristiques du matériel. Il est plus rapide d'accéder à la mémoire locale qu'à la mémoire associée aux autres nœuds NUMA [Rib11].

7.1.1 Comment représenter les plate-formes MPSoC en utilisant une architecture NUMA ?

La représentation de la plate-forme MPSoC doit être conforme au modèle que nous avons défini dans le chapitre 4, section 4.3. Nous avons donc représenté les éléments de la plate-forme MPSoC en utilisant une architecture NUMA de la façon suivante :

- *Nœuds MPSoC* : nous avons décidé de faire correspondre les nœuds MPSoC à ceux de l'architecture NUMA. Les processeurs d'un nœud MPSoC sont donc les cœurs d'un nœud NUMA. Le nombre de nœuds MPSoC ne peut pas dépasser celui de l'architecture NUMA. Cependant, cette représentation est réaliste puisqu'elle nous permet d'avoir des nœuds MPSoC physiquement distincts.
- *Mémoire MPSoC* : nous représentons la mémoire d'un nœud MPSoC par le bloc mémoire associé au nœud NUMA correspondant. Dans une architecture NUMA, la mémoire est partagée par définition entre tous les nœuds. Nous devons donc restreindre les processeurs d'un nœud afin qu'ils n'aient uniquement accès à leur propre bloc de mémoire. Les processeurs et donc les cœurs NUMA des différents nœuds devront communiquer afin de s'échanger des données.
- *Périphériques MPSoC* : les périphériques de l'architecture MPSoC sont ceux

disponibles dans l'architecture NUMA. Par définition, dans une architecture NUMA tous les périphériques sont accédés par tous les nœuds. Cependant, dans une architecture MPSoC, il est plus réaliste de définir les accès des périphériques en fonction des besoins en termes d'E/S des nœuds. Par exemple, il est fréquent d'utiliser des processeurs de contrôle qui accèdent à l'ensemble des périphériques et qui transmettent les données d'E/S aux processeurs de calcul intensif.

- *Réseau d'interconnexion* : dans l'architecture MPSoC, nous utilisons le réseau d'interconnexion natif NUMA. La topologie de l'architecture MPSoC est donc représentée par celle de l'architecture NUMA.

Afin d'obtenir la représentation considérée, notre idée consiste à utiliser un processus Linux pour le matériel et le logiciel d'un nœud MPSoC. Ce processus Linux est restreint à exploiter les ressources d'un nœud physique NUMA. Cette représentation nous permettrait d'avoir une vraie indépendance d'exécution entre les nœuds MPSoC puisqu'ils utilisent les composants physiquement différents de l'architecture NUMA.

Nous avons décidé de ne pas réserver la totalité des ressources d'un nœud NUMA pour la représentation d'un nœud MPSoC. Nous réservons, en effet, une partie pour le débogage, ainsi que pour le système Linux.

Nous avons conçu un logiciel pour représenter un nœud MPSoC. En entrée, nous fournissons un fichier de configuration qui spécifie les ressources NUMA à utiliser pour le nœud MPSoC. Durant l'exécution, les ressources spécifiées sont allouées et l'exécution est faite sur le nœud correspondant.

Le fichier de configuration contient les informations suivantes :

- *Identité du nœud* : entier qui indique le nœud physique NUMA.
- *Processeurs* : le nombre de processeurs réservés dans le nœud NUMA.
- *Mémoire* : entier représentant la taille de la mémoire à allouer parmi celle disponible dans le nœud NUMA.

Nous devons également représenter le système de fichiers MPSoC. Un premier choix peut être d'utiliser le système de fichiers Linux natif. Cependant, l'exécution du logiciel MPSoC peut être fortement perturbée lorsque ce système est sollicité par les besoins du noyau. Afin d'éviter ces perturbations, nous avons donc fait un autre choix qui consiste à utiliser une partition spécifique, créée sur une partie de la mémoire non-utilisée d'un des nœuds NUMA.

La figure 7.1 représente les composants d'un nœud MPSoC et les ressources exploitées de l'architecture NUMA. Le processus Linux exécute le logiciel du nœud MPSoC. Ce logiciel utilise quatre cœurs et une partie de la mémoire du nœud NUMA. Une autre partie de la mémoire du nœud NUMA est réservée pour le système de fichiers.

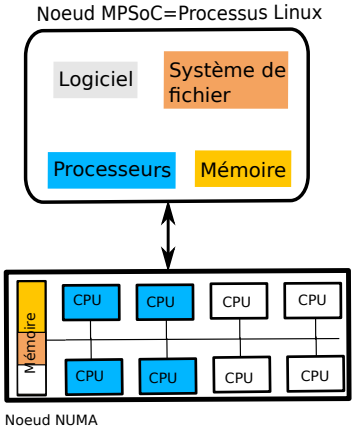


FIGURE 7.1 – Représentation d'un nœud MPSoC par un processus Linux.

7.1.2 Architecture MPSoC considérée

Nous allons représenter une plate-forme MPSoC à partir de l'architecture NUMA que nous avons à notre disposition et qui est montrée sur la figure 7.2. Cette architecture est organisée en un ensemble de nœuds interconnectés. Chaque nœud est composé de 8 processeurs X86 et d'un banc de mémoire de 32Go. Tous les nœuds ont accès à l'ensemble des périphériques du système.

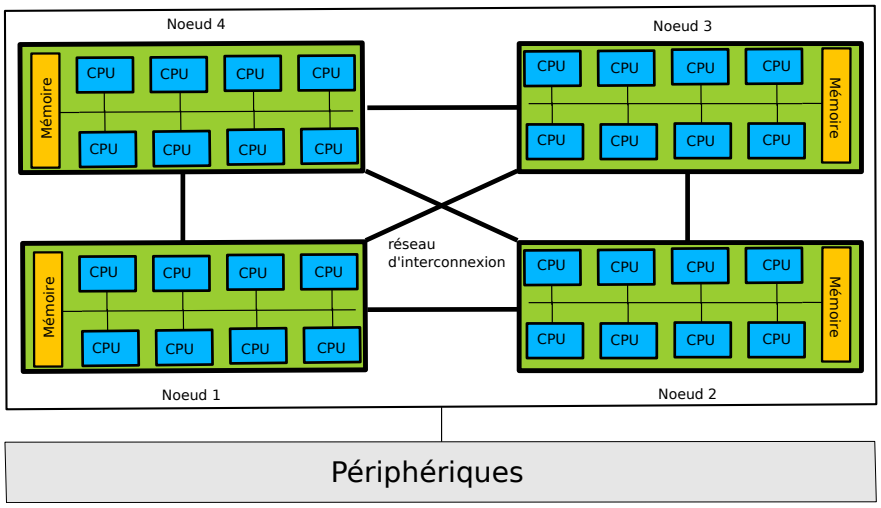


FIGURE 7.2 – Architecture NUMA.

Les logiciels que nous utilisons sur cette architecture afin de représenter une plate-forme MPSoC sont le noyau Linux et la bibliothèque libnuma [Kle05]. Cette bibliothèque encapsule les appels systèmes Linux afin de fournir un API de placement des données sur les bancs de mémoire spécifiques.

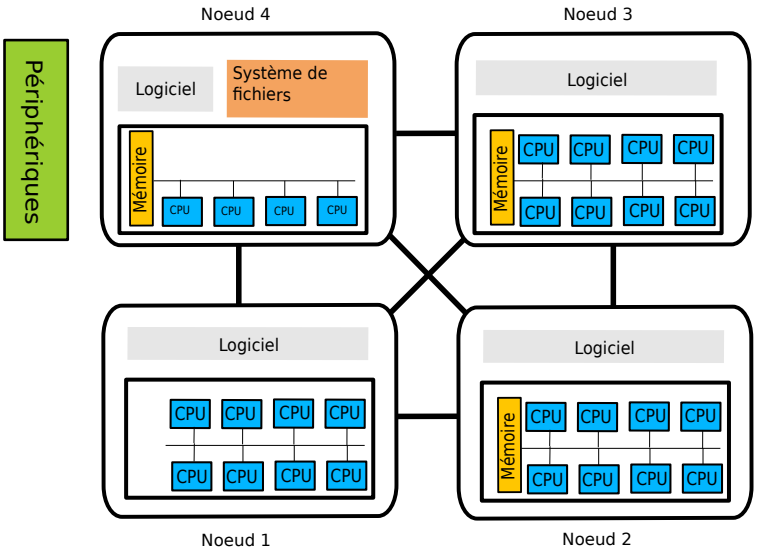


FIGURE 7.3 – Architecture MPSoC considérée.

Sur la figure 7.3, nous avons représenté une plate-forme MPSoC à quatre nœuds qui correspondent à ceux de l'architecture NUMA. Chacun des nœuds MPSoC est représenté par un processus Linux. La partie logicielle sur chaque nœud comprend l'API MPSoC, ReDSoC, ainsi que le logiciel de traitement multimédia, expliqués dans la section suivante. La partie matérielle de MPSoC comprend trois nœuds qui utilisent huit processeurs, et un quatrième qui dispose de quatre processeurs. Les quatre processeurs restants de l'architecture NUMA sont utilisées pour le noyau Linux et le débogage. Nous avons considéré que le nœud à quatre processeurs accède à l'ensemble des périphériques, au système de fichiers et peut transférer les données des périphériques aux autres nœuds de la communication. Ce nœud utilise 4Go pour la mémoire et 16Go pour le système de fichiers des 32Go disponibles. Les trois autres nœuds utilisent 4Go pour la mémoire des 32Go disponibles.

Afin d'utiliser ReDSoC, nous avons besoin d'un support de stockage accessible à partir de chacun des nœuds afin de gérer l'enregistrement et l'utilisation des traces. Un choix réaliste consiste à supposer qu'un port de traces permet d'enregistrer les données, par exemple sur le système de fichiers Linux. Ces traces pourront être récupérées par le nœud de contrôle et transférées vers les autres nœuds durant les ré-exécutions. Cependant, cette approche nécessite des extensions importantes de ReDSoC afin d'implémenter les transferts de traces. Nous avons fait le choix d'utiliser une solution simplifiée, c'est-à-dire d'utiliser comme support de stockage une partition supplémentaire de 4Go sur chacun des nœuds. Cette partition représente un port de trace et facilite la récupération des données enregistrées en utilisant des lectures de fichiers.

Pour mettre au point le logiciel MPSoC, nous exécutons des instances de GDB

qui intègrent notre extension de mise au point. Ces instances, exécutées sous le contrôle de numactl [Kle05], utilisent des ressources de l'architecture NUMA qui ne sont pas réservées pour les nœuds MPSoC. Une instance de GDB débogue le logiciel d'un nœud MPSoC en s'attachant au processus correspondant.

7.2 API MPSoC

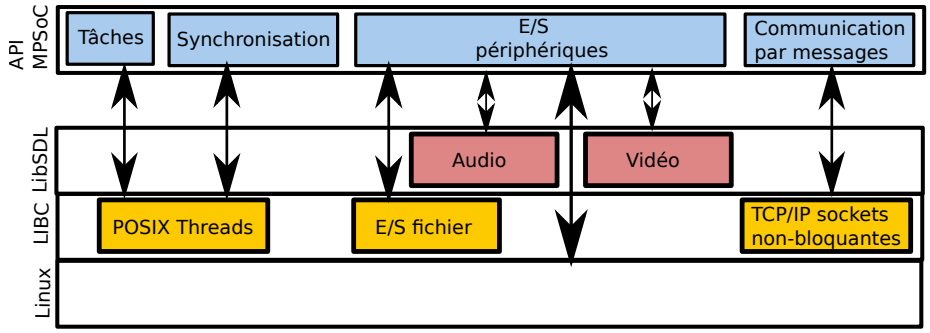


FIGURE 7.4 – Pile logicielle utilisée par l'API MPSoC.

Dans cette étude de cas, nous avons implémenté les quatre types de fonctions de l'API MPSoC en utilisant les trois couches logicielles montrées sur la figure 7.4 : Linux 2.6, la bibliothèque libSDL [Pen03] et la *libc*.

Les fonctions de gestion de tâches et de synchronisation sont basées sur les fonctions POSIX Threads de la *libc*. Les tâches d'un nœud MPSoC correspondent aux threads d'un processus Linux. Les threads sont créés et ordonnancés sur les processeurs de l'architecture NUMA réservés pour le nœud MPSoC en utilisant l'appel système `sched_setaffinity`. Cette fonction, exécutée lors de la création d'un thread, garantit que son exécution est effectué sur un ensemble de processeurs spécifiés en entrée. Les processeurs sont choisis parmi ceux disponibles dans le nœud NUMA, encodés statiquement. Nos fonctions de synchronisation utilisent directement l'interface de gestion des verrous et des conditions de la *libc*.

Nous avons utilisé notre API d'entrées/sorties afin d'accéder au système de fichiers, aux données d'un environnement extérieur par le réseau, à l'écran, à la carte son et à l'horloge. Le système de fichiers est accédé en utilisant les fonctions de lecture/écriture de la *libc*, l'écran et à la carte son en utilisant la libSDL et l'hologe par un registre Linux. Le type du dispositif est spécifié comme paramètre en entrée de la fonction `pOpen`.

La communication par messages est basée sur les fonctions de gestion de sockets de la *libc* pour la communication entre les processus (IPC) qui représentent les nœuds MPSoC.

7.3 Débogage d'une application de mosaïque vidéo

FIGURE 7.5 – Application de mosaïque vidéo.

Dans cette section, nous présentons le débogage d'une erreur non-déterministe d'une application de mosaïque vidéo. La figure 7.5 représente une capture d'écran de l'exécution de cette application qui affiche simultanément sur l'écran trois flux vidéo indépendants.

Comme dans l'étude de cas précédent, une erreur a été introduite par les ingénieurs de notre équipe afin de défier notre prototype ReDSoC. Dans la suite, nous décrivons l'application de mosaïque avant de nous focaliser sur le processus de mise au point. Nous étudions plus particulièrement la capacité de ReDSoC de mettre au point des exécutions effectuées sur plusieurs processeurs.

7.3.1 L'application de mosaïque vidéo

Nous avons créé une application de mosaïque à partir de la suite FFMPEG [Tom06] qui contient plus de 300,000 lignes de code dans 30Mo. Nous avons plus particulièrement utilisé deux composants de cette suite : FFPLAY [AHC09] et FFSERVER [PR11]. Nous présentons d'abord ces deux composants. Ensuite, nous montrons les modifications que nous avons apportées sur ces composants afin de concevoir notre application.

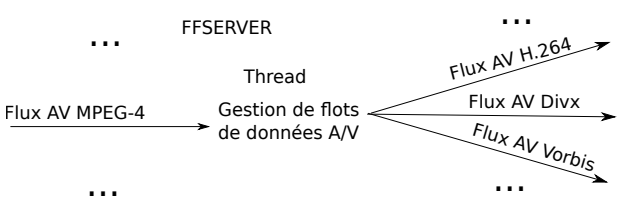


FIGURE 7.6 – Composant FFSERVER.

FFSERVER est un serveur de flots audio/vidéo, représenté sur la figure 7.6). Ce serveur prend en entrée des flux A/V en utilisant des protocoles comme Real-time Transport Protocol [SCFJ96] (RTP), Real-Time Stream Protocol [Sch98] (RTSP) ou HTTP. Pour chaque flux en entrée, plusieurs flux en sortie peuvent être créés aux formats différents comme MPEG-4, H.264, Divx, Vorbis [EDN00], etc. FFSERVER est basé sur une boucle infinie qui vérifie successivement la disponibilité de données en entrée. Les données disponibles sont envoyées vers la sortie correspondante.

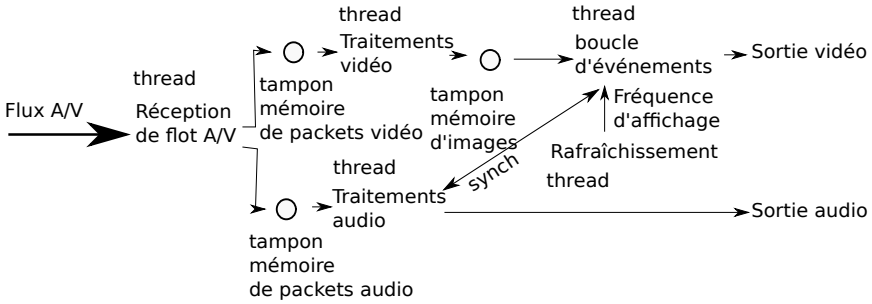


FIGURE 7.7 – Composant FFPLAY.

FFPlay est un lecteur de flux A/V représenté sur la figure 7.7. À la réception d'un flux A/V en entrée, il extrait les paquets de la vidéo et de l'audio et les place dans deux tampons en mémoire différents. Deux threads appliquent des traitements pour reconstituer respectivement les images et les trames audio. Un autre thread est utilisé pour gérer le taux de rafraîchissement, défini par le nombre d'images par seconde (IPS). Un quatrième thread effectue trois actions :

- Traitement des commandes des utilisateurs.
- Synchronisation avec le thread de traitement audio.
- Envoi des données vidéo vers les sorties selon la fréquence d'affichage spécifiée en entrée.

Nous avons utilisé FFSERVER et FFPlay pour créer notre propre application de mosaïque. Cette application affiche trois flux vidéo en même temps sur un écran. Un flux principal d'audio/vidéo qui prend 2/3 de l'écran et deux flux vidéo secondaires qui partagent 1/3 de l'écran. Cette application peut être utilisée dans les équipements embarqués de surveillance pour contrôler plusieurs zones simultanément ou encore dans les récepteurs de télévision pour avoir un aperçu de plusieurs chaînes.

La figure 7.8 montre l'architecture de notre application. Afin d'intégrer notre API MPSoC, nous avons dû apporter d'importantes modifications sur les deux composants FFSERVER et FFPlay. En effet, nous avons d'une part modifié la

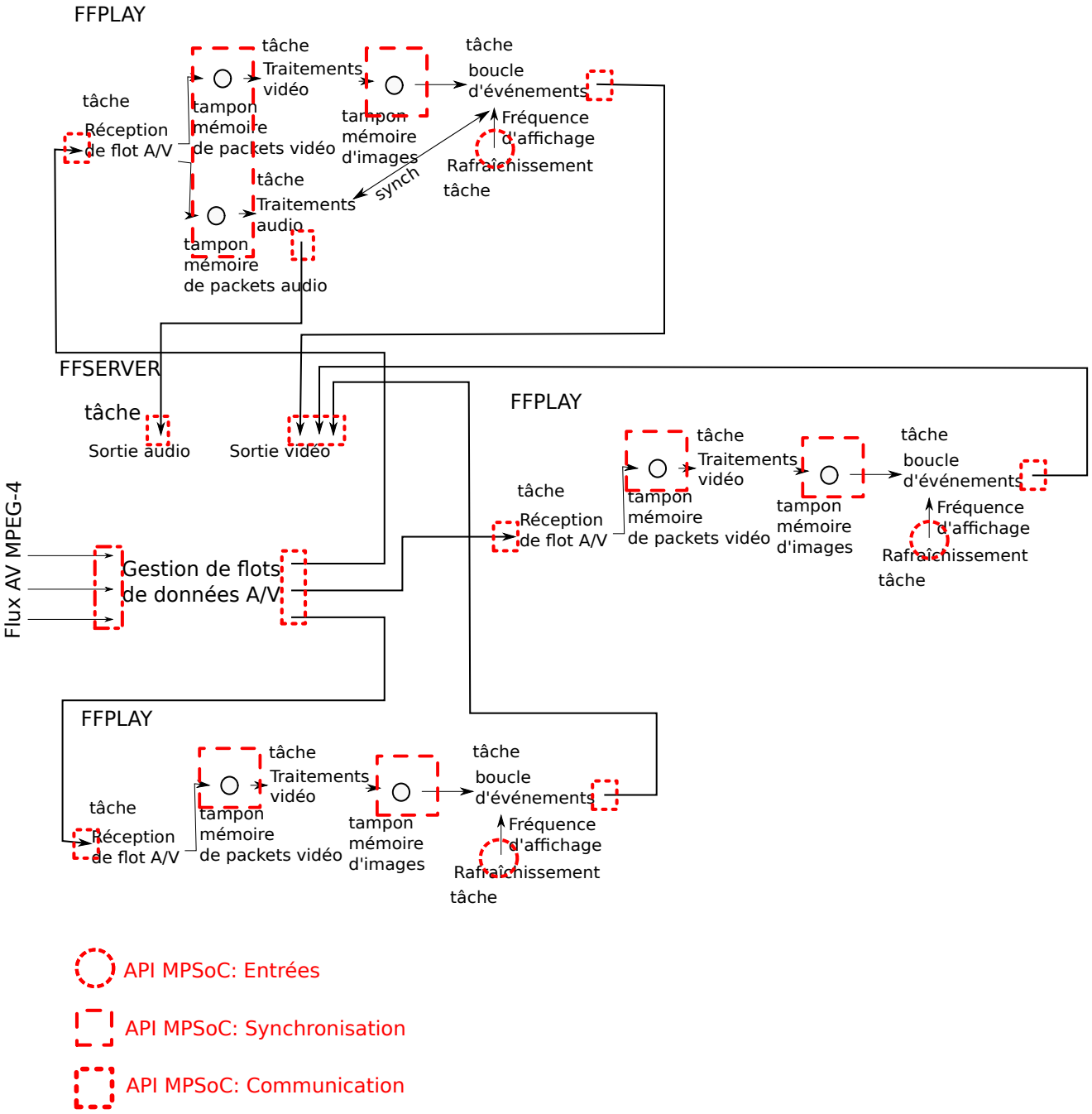


FIGURE 7.8 – Application de mosaïque vidéo MPSoC.

structure des composants. D'autre part, nous avons remplacé tous les appels système Linux par des appels vers notre librairie MPSoC.

En ce qui concerne la structure des composants, nous avons procédé comme suit. Dans FFSERVER, nous avons rajouté un code qui reçoit les trois flux A/V afin de les afficher sur la partie correspondante de l'écran et d'envoyer l'audio vers la sortie correspondante.

Dans FFPLAY, nous avons supprimé les parties qui affichent les images et qui envoient le son. Nous avons également supprimé les traitements audio dans les deux FFPLAY secondaires puisque leur son n'est pas utilisé.

La figure 7.8 représente l'utilisation de notre API dans l'application de mosaïque. En ce qui concerne le remplacement des appels système Linux, par des appels à notre API MPSoC, nous avons procédé comme suit :

- *Gestion de tâches* : nous avons remplacé les fonctions de gestion de threads par des fonctions de gestion de tâches.

- *E/S* : dans FFSERVER, les entrées sont les trois flux A/V reçus depuis un environnement extérieur par le réseau, les sorties sont les trames audio redirigées vers les haut-parleurs et les images affichées sur l'écran par la SDL. Dans FFPlay les entrées sont les valeurs de l'horloge afin de gérer le taux de rafraîchissement et il n'y a pas de sorties. En adaptant le code de l'application, nous avons pu remplacer les fonctions d'E/S natives par celles de notre API.

- *Synchronisation* : dans FFPLAY, trois tampons en mémoire circulaires sont utilisés pour les transferts de données audio et vidéo entre les tâches. Ces tampons en mémoire circulaire sont gérés par un ensemble de verrous et de conditions de la *libc* dans le code natif, replacées par les fonctions de synchronisation de notre API MPSoC.

- *Communication* : tous les transferts de données A/V entre FFSERVER et les trois lecteurs FFPLAY utilisent les sockets non-bloquantes de la *libc*, remplacées par les fonctions de notre API MPSoC. Ces sockets non-bloquantes permettent notamment de gérer la réception de plusieurs flux de données simultanément. En effet, chaque flux est reçu sur un port de communication différent. Une boucle infinie vérifie successivement la disponibilité de données sur chacun des ports.

Pour distinguer entre les tâches des quatre composants de l'application (FFSERVER et trois FFPLAY) nous avons regroupé les tâches de chaque composant dans un nœud différent.

7.3.2 Débogage de l'erreur

Nous lançons les quatre processus Linux qui correspondent aux nœuds MPSoC. D'abord, le processus exécutant FFSERVER est lancé sur le Nœud0. FFSERVER lit trois flux vidéo à partir du système de fichier. Les données de ces flux sont mises dans trois tampons en mémoire en attendant la disponibilité des FFPLAY. Les trois processus exécutant FFPLAY sont lancés successivement sur les nœuds Nœud1, Nœud2 et Nœud3.

Suite à un nombre important d'exécutions du logiciel MPSoC, avec des flux vidéo en entrée différents, nous avons constaté une erreur d'exécution non-déterministe. En effet, certaines exécutions sont correctement effectuées. Dans d'autre exécutions, l'affichage des images correspondant à un ou aux plusieurs flux A/V ne se fait pas. Cependant, lorsque l'affichage d'un flux commence, il continue jusqu'à la fin et se termine correctement.

Nous avons réussi à tracer une exécution où un flot sur les trois n'est pas affiché. Comme il s'agit du flot traité par FFPLAY sur Nœud2, intuitivement nous suspectons donc le comportement de ce nœud comme anormal. Nous poursuivons le débogage par la visualisation des traces de ré-exécution déterministe afin de confirmer notre intuition ou de se focaliser sur un autre nœud et d'identifier un intervalle de temps à déboguer.

La figure 7.9 représente les traces de ré-exécution déterministe collectées sur les quatre nœuds MPSoC. Les traces du Nœud0 (FFSERVER) montrent les points de réception de messages non-bloquants à partir de FFPLAY. Les traces des trois autres nœuds (FFPLAY) montrent une activité de réception de messages (flux de données A/V) au début de leur exécution. Ensuite, l'activité de synchronisation montre les accès aux tampons en mémoire qui représentent les traitements des données A/V.

Nous pouvons remarquer que les traces des trois composantes FFPLAY représentent chacune trois tâches. Cependant, le nombre de tâches réellement utilisées dans chaque composante est de cinq. En effet, la tâche qui décode le son est cachée dans la bibliothèque libSDL qui n'est pas tracée. De plus, la tâche qui effectue le taux de rafraîchissement des images ne produit pas de traces. En conséquence, les traces des composantes FFPLAY comportent uniquement trois tâches.

Sur les traces montrées, nous observons le blocage infini de la tâche T1. Cette tâche bloque également l'exécution des autres tâches. Pour mieux comprendre le comportement de cette tâche, nous avons agrandi la zone autour du blocage.

La figure 7.10 représente le comportement des tâches autour du point de blocage. Nous observons plus particulièrement que la tâche T2 est bloquée suite à la prise d'un verrou qui est repris ensuite par la tâche T0. Afin de comprendre la raison de ce blocage, nous choisissons un intervalle de temps à déboguer entre les

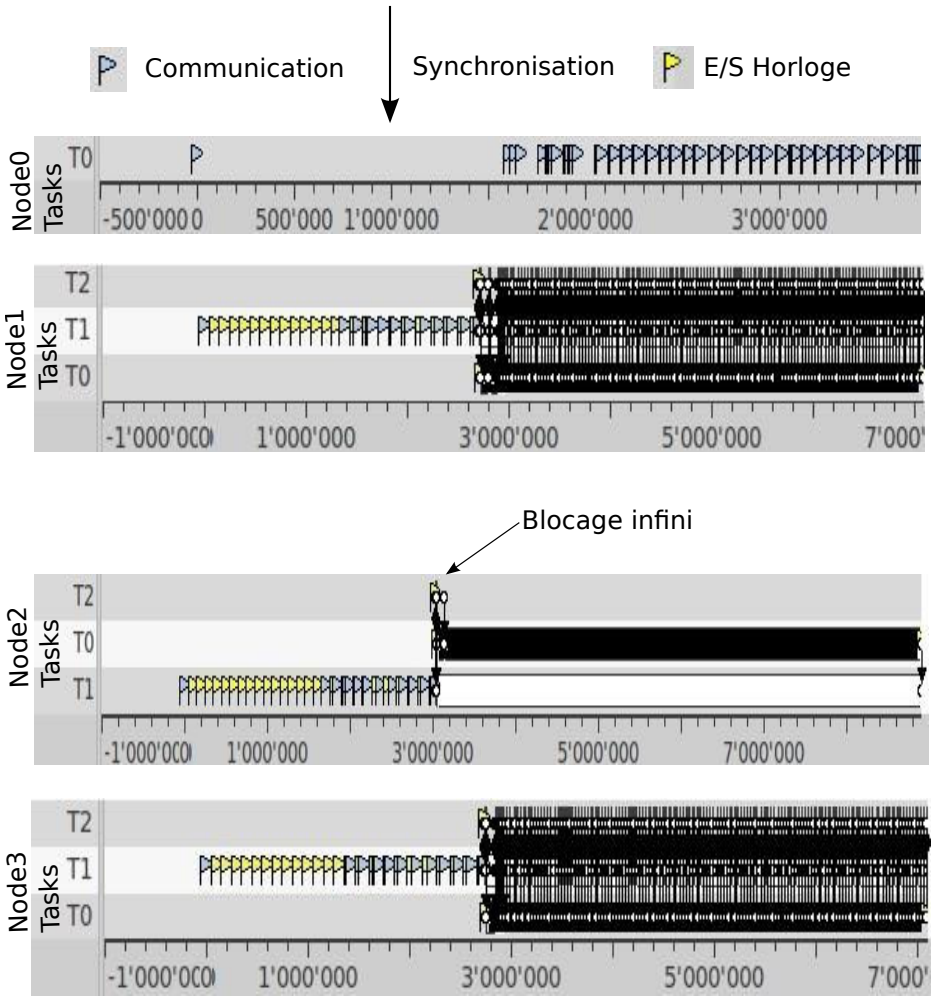


FIGURE 7.9 – Visualisation des traces de ré-exécution déterministe.

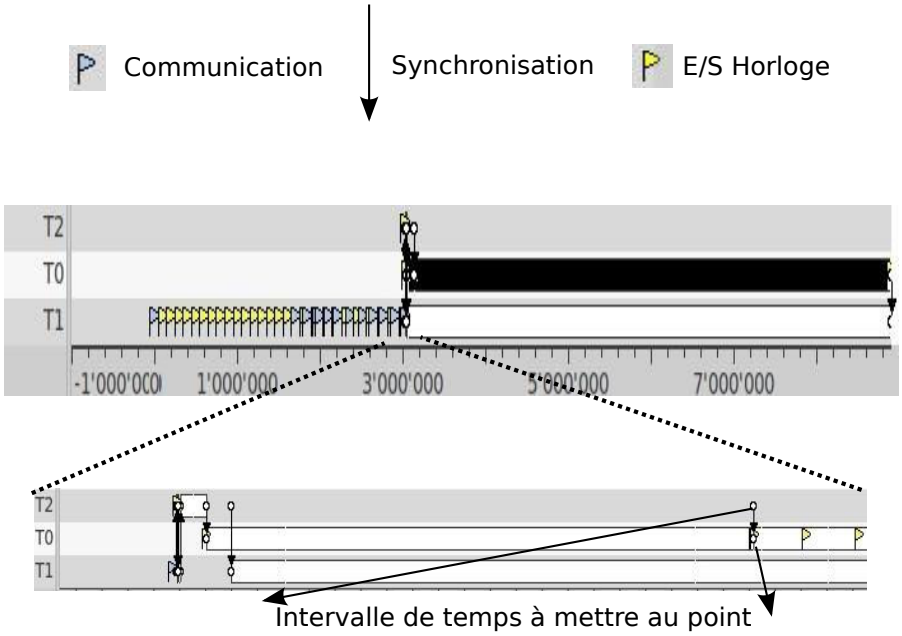


FIGURE 7.10 – Analyse des traces et sélection d'un intervalle de temps.

deux opérations de synchronisation correspondantes.

En utilisant GDB sur la ré-exécution partielle du Nœud2 et durant l'intervalle sélectionné, nous avons effectué les deux observations suivantes :

- La première borne de l'intervalle de temps à déboguer correspond à une attente d'allocation en mémoire pour les données d'une image sur une variable de condition.
- La deuxième borne correspond à l'entrée en section critique qui alloue l'espace pour l'image.

L'erreur correspond à une omission de la part du développeur à signaler que l'allocation de l'espace pour l'image est accomplie. En conséquence, lorsque l'attente de l'allocation (T1) se fait avant l'allocation (T2), l'exécution est bloquée. Lorsque ces deux opérations s'effectuent dans l'ordre inverse, l'exécution se déroule normalement.

La mise au point de cette erreur a nécessité trois ré-exécutions partielles du même nœud et dans le même intervalle de temps. Dans la première ré-exécution nous nous sommes rendu compte qu'une variable de condition n'était jamais signalée. Dans la deuxième ré-exécution, nous avons constaté que la variable de condition permet d'effectuer l'attente d'allocation d'images. Durant la troisième ré-exécution, nous avons découvert que la variable de condition n'est pas signalée après l'allocation d'une image. En terme général, le nombre de ré-exécutions par-

tielles dépendent entièrement des capacités et de l'expertise du développeur à bien cibler et à analyser la partie fautive d'une exécution.

7.4 Synthèse

Dans ce chapitre, nous avons présenté une deuxième étude de cas sur le passage à l'échelle de nos outils de mise au point. Nous avons débogué une exécution sur 18 processeurs répartis dans quatre nœuds MPSoC. Trois processeurs ont été virtualisés en utilisant les trois cœurs parmi les huit disponible d'un processeurs NUMA. Les quinze autres ont été virtualisés en utilisant cinq cœurs sur trois processeurs NUMA.

Le logiciel que nous avons utilisé est une application de mosaïque qui est basée sur la suite FFMPEG, largement utilisée pour la conception de logiciels de traitement multimédia.

Nous avons choisi de représenter une plate-forme MPSoC à partir d'une architecture NUMA à cause de leurs caractéristiques communes qui permettent le passage à l'échelle des performances. Nous avons plus particulièrement implémenté notre API MPSoC de façon à représenter les éléments d'une plate-forme MPSoC comme les nœuds, la mémoire, le système de fichiers, les périphériques et le réseau d'interconnexion en utilisant l'architecture NUMA.

ReDSoC nous a permis de déboguer une erreur non-déterministe de synchronisation en seulement trois ré-exécutions partielles du même nœud et durant le même intervalle de temps. Le nœud et l'intervalle de temps ont été identifiés en se basant sur la sémantique du logiciel et en observant les traces de ré-exécution déterministe. La ré-exécution déterministe de la synchronisation, de la communication et des entrées nous a permis de reproduire l'erreur. La ré-exécution partielle nous a permis de réduire le nombre de processeurs à déboguer de 28 à 8 et de déboguer dans un intervalle de temps entre deux accès successifs d'une section critique.

A cause du passage à l'échelle, les futures MPSoC intégreront plusieurs centaines de processeurs distribués dans une multitude de nœuds. Indépendamment du nombre processeurs dans l'architecture, en déboguant un seul nœud, notre approche permettrait de réduire considérablement le nombre de processeurs. En effet, le nombre de processeurs dans chaque nœud serait limité à quelques unités voire à quelques dizaines d'unités afin d'éviter les problèmes de contention dûs aux accès à la mémoire partagée.

Chapitre 8

Évaluation de la méthodologie de mise au point

Dans ce chapitre, nous évaluons notre méthodologie pour la mise au point des MPSoC. Dans la suite, nous présentons les critères d'évaluation choisis avant de passer à l'évaluation proprement dite.

8.1 Critères d'évaluation

Nous évaluons notre solution en choisissant trois critères qui correspondent aux objectifs que nous nous sommes fixés. Il s'agit notamment :

- *Mise au point d'erreurs non-déterministes* : ce critère évalue la capacité de notre solution à déboguer des erreurs non-déterministes. Il s'agit plus particulièrement de discuter des sources de non-déterminisme prises en compte et de l'impact de cette décision sur le débogage d'erreurs non-déterministes.
- *Passage à l'échelle* : il s'agit d'évaluer notre capacité à mettre au point des logiciels complexes s'exécutant sur des architectures MPSoC à un grand nombre de processeurs. Nous nous intéressons à la valeur ajoutée de notre solution par rapport aux approches classiques de débogage de systèmes embarqués.
- *Performance* : nous évaluons la performance par rapport à deux paramètres : l'intrusion et le délai d'attente de débogage. Nous définissons l'intrusion comme l'impact de notre solution sur les occurrences des erreurs. En effet, il ne faut pas que des erreurs disparaissent ou se produisent à une fréquence différente. Nous définissons le délai d'attente comme le temps d'attente nécessaire avant de pouvoir ré-exécuter et donc déboguer une partie spécifique de l'application. Il faut que ce temps soit minimal afin de permettre une mise

au point rapide.

- *Simplicité d'utilisation* : nous évaluons la démarche qui est utilisée pour mettre au point une exécution MPSoC. Cette démarche est un ensemble d'étapes : collecte de traces de ré-exécution déterministe, identification d'une partie de l'exécution à mettre au point, ré-exécution déterministe etc. Plus particulièrement, nous discutons les possibilités de simplifier ces étapes lorsqu'elles sont effectuées par les développeurs.

8.2 Mise au point d'erreurs non-déterministes

Dans un contexte non-déterministe, la mise au point est difficile puisque les erreurs ne se produisent pas durant toutes les exécutions et il est possible qu'elles ne se produisent pas aux mêmes points de l'exécution. D'un côté, il n'y a pas de garantie que le code erroné sera exécuté durant la phase de mise au point.

Pour corriger les erreurs non-déterministes, notre solution permet la ré-exécution déterministe. Nous reproduisons trois sources de non-déterminisme : les paramètres d'entrée, la synchronisation et la communication par messages. Ceci nous permet de mettre au point les erreurs classiques devenues non-déterministes à cause de ces trois sources.

Nous ne reproduisons pas les accès non-synchronisés à la mémoire. Néanmoins, notre solution peut être facilement couplée avec un outil de détection de ces accès comme erreurs (ERASER [SBN+97], Helgrind+ [JBPT09] ou Intel Thread Checker [BBMP06]. Lorsque cet outil détecte une telle erreur, il s'agit de rendre l'accès synchronisé et de ré-exécuter l'application en utilisant ReDSoC. Lorsque l'erreur continue d'être non-déterministe, on peut en conclure que cet accès n'a pas d'impact sur le comportement erroné. Dans ce cas, il faut répéter cette procédure avec le prochain accès non-synchronisé. Lorsque l'accès qui rend l'erreur non-déterministe est identifié, on peut utiliser ReDSoC afin de reproduire l'exécution erronée.

Une autre source de non-déterminisme que nous ne considérons pas sont les interruptions. Ceci est une limitation importante. En effet, au niveau du système, les interruptions sont une composante essentielle pour la gestion des données de multiples périphériques dans les systèmes embarqués. Il est donc possible d'avoir fréquemment des erreurs non-déterministes dues aux interruptions. Lorsque la cible du débogage est le système, il est donc impératif d'inclure dans la méthodologie un mécanisme de ré-exécution déterministe des interruptions. On peut utiliser une des approches étudiées dans l'état de l'art (chapitre 3, section 3.4.3).

Notre solution pourrait être utilisée afin de déboguer le code applicatif comme nous l'avons fait dans les deux études de cas présentées. En effet, nous pensons que la complexité croissante du logiciel et du matériel MPSoC nécessitera des

couches d'abstraction entre le système et le code applicatif qui cacheront une partie des interruptions par des mécanismes alternatifs comme l'attente active que nous reproduisons. Il serait donc possible de déboguer les erreurs applicatives, lorsqu'elle sont rendues non-déterministes à cause des entrées fournies par un mécanisme d'attente active.

Si nous considérons maintenant les études de cas menées, nous pensons avoir montré que la méthodologie permet la mise au point d'erreurs non-déterministes. Le débogage du logiciel du jeu (chapitre 6) nous a permis de mettre au point une erreur non-déterministe d'E/S en reproduisant deux sources de non-déterminisme :

- Communication par messages : la communication par messages est utilisée pour recevoir les déplacements des pièces effectuées par le joueur adverse et ainsi mettre à jour le plateau du joueur courant. Afin de respecter les contraintes de temps réel, la durée de l'exécution des fonctions doit être prévisible. Les messages sont donc reçus d'une manière non-bloquante et créent du non-déterminisme. La ré-exécution déterministe de la communication par messages nous a permis de reproduire les déplacements des pièces sur la grille qui affiche le jeu adverse.
- Entrées du clavier : pour les mêmes raisons de respect de contraintes de temps réelles, les entrées du clavier qui déterminent les déplacements des pièces sur la grille du joueur sont reçues en utilisant des opérations non-bloquantes. En reproduisant cette source de non-déterminisme, d'une part, les pièces se déplacent comme durant l'exécution erronée. D'autre part, les joueurs physiques ne sont pas nécessaires durant la phase de mise au point.

Sur l'application de mosaïque vidéo, nous avons utilisé ReDSoC afin de déboguer une erreur non-déterministe de synchronisation en reproduisant les deux sources de non-déterminisme suivantes :

- Communication par messages : la communication par messages est non-déterministe à deux points de l'exécution du logiciel. Le premier point est lorsque FFPLAY reçoit le flux A/V encodé à partir de FFSERVER. Cette réception se fait par une boucle qui vérifie périodiquement la disponibilité de données du flux A/V par des messages non-bloquants. Le deuxième point est lorsque FFSERVER reçoit le flux A/V décodé à partir de FFPLAY pour être visualisé. La ré-exécution déterministe nous a permis de reproduire les données des flux A/V reçues entre chaque affichage d'image.
- Synchronisation : le non-déterminisme est dû à l'accès concurrent des tampons mémoire partagés entre les tâches. En effet, les données des flux A/V sont échangées entre les tâches des instances de FFPLAY par un tampon mémoire circulaire. Ce tampon en mémoire est accédé d'une manière atomique en utilisant des fonctions de synchronisation. ReDSoC nous a donc permis de reproduire l'état du tampon mémoire lors de chaque accès.

Dans les deux études de cas, la reproduction du non-déterminisme nous a permis de se focaliser sur une des exécutions erronées. Nous avons donc réussi à déboguer cette exécution en garantissant que la même erreur se produirait durant le débogage.

8.3 Passage à l'échelle

Le passage à l'échelle des MPSoC accroît la complexité des exécutions et en conséquence celle du débogage. En effet, le débogage consiste à effectuer une succession de deux actions : suspension de l'exécution à un point donné et analyse des éléments de l'exécution. Avec l'accroissement des processeurs, le *bon* point de l'exécution à suspendre devient plus difficile à trouver. D'autre part, l'analyse à partir de ce point se complexifie puisque le nombre d'éléments de l'exécution devient plus important. Dans les deux études de cas, nous avons identifié deux difficultés de débogage MPSoC. Il s'agit en utilisant les outils standard, notamment l'analyse de l'exécution sur plusieurs processeurs et les interactions entre les processeurs. Dans la suite, nous présentons ces difficultés ainsi que la valeur ajoutée de notre solution.

8.3.1 Analyse de l'exécution sur plusieurs processeurs

Nous avons constaté qu'il est particulièrement difficile d'analyser un même code exécuté d'une manière répétitive et d'une fréquence élevée, par plusieurs processeurs qui traitent des données différentes. En utilisant les débogueurs classiques, cette analyse consiste à utiliser un point d'arrêt sur ce code. Ce point d'arrêt s'activerait sur chacun des processeurs et à chaque répétition. Lorsqu'un point d'arrêt est atteint, les développeurs doivent analyser l'état de l'exécution pas à pas sur un nombre important de processeurs. Cette analyse représente donc une tâche lourde qui demande un temps considérable. Nous avons retrouvé cette problématique sur les deux applications étudiées.

Sur l'application de mosaïque vidéo, nous avons eu besoin d'analyser un code exécuté d'une manière répétitive par quatre tâches sur trois nœuds. Ce code correspond aux accès à un tampon mémoire circulaire afin de transférer les images d'une tâche à l'autre. En utilisant un débogueur classique, un point d'arrêt sur ce code s'activerait 12 fois (trois nœuds*quatre tâches) par répétition. De plus, en déboguant pas à pas, l'exécution serait suspendue 12 fois par instruction (une par processeur).

Notre méthodologie de mise au point nous a permis d'isoler un seul nœud et un intervalle de temps délimité par deux accès au code cible effectués par deux tâches.

L'erreur a été rapidement identifiée en utilisant seulement deux points d'arrêt et en analysant un code exécuté par deux tâches.

Sur l'application du jeu, nous avons voulu étudier les deux opérations : de réception des données réseau et de traitements des entrées du clavier. Ces deux opérations sont exécutées d'une manière répétitif par les deux instances du jeu sur les deux processeurs. L'analyse de ces deux opérations nécessite deux points d'arrêt correspondantes. Un point d'arrêt s'activera quatre fois par répétition (2 points d'arrêt*2 instances du jeu).

En utilisant notre solution, nous avons isolé successivement deux processeurs et deux intervalles de temps. Ces intervalles de temps correspondent respectivement à une dizaine opérations de réception des données réseau et de traitements des entrées du clavier. Nous avons donc pu identifier l'erreur en utilisant une vingtaine d'occurrences de points d'arrêt et en analysant une seule instance du jeu.

8.3.2 Interactions entre les processeurs

Nous avons observé que les interactions entre les nœuds sont modifiées lorsqu'une partie seulement des nœuds est déboguée. En effet, le débogage peut créer de nouvelles erreurs ou faire disparaître d'autres à cause de la communication entre les nœuds. Ces erreurs dépendent de la sémantique de l'application et des implémentations de l'intergiciel ou du système. Néanmoins, l'apparition ou la disparition des erreurs est due à la suspension du nœud qui reçoit les messages alors que celui qui les envoie est actif ou vise versa. En utilisant notre méthodologie il est possible d'éviter ce cas, puisque la communication par messages entre les nœuds est reconstruite en utilisant des traces à place des mécanismes réels.

Nous avons plus particulièrement observé ce cas en utilisant GDB sur un des deux nœuds de la plate-forme embarquée, pendant que l'autre s'exécute. Le nœud débogué est arrêté sur un point d'arrêt. L'autre nœud continue à s'exécuter et à envoyer des données. Ces données sont accumulées dans le tampon mémoire de réception de données qui ne peut pas être vidé puisque l'exécution est arrêtée. En conséquence, une nouvelle erreur est créée à cause de la saturation du tampon mémoire. La ré-exécution des nœuds en séparation nous a permis de résoudre ce problème. En effet, le tampon mémoire est rempli comme lors de l'exécution de référence à partir des traces de ré-exécution partielle.

8.4 Performance

Nous évaluons les performances de ReDSoC sur les deux plates-formes utilisées dans les études de cas : une première réelle, à deux processeurs dans deux nœuds

et une deuxième à 28 processeurs dans quatre nœuds. Sur la plate-forme réelle nous utilisons deux applications : le jeu de Tetris et un décodeur MJPEG. Sur la plate-forme à quatre nœuds nous utilisons l'application de mosaïque vidéo.

Le décodeur MJPEG traite un flux d'images JPEG [1]. Le décodage consiste à diviser les images en blocs. Ces blocs sont décodés en utilisant un algorithme de Huffman [Huf52], un ré-ordonnancement de pixels et une transformation discrète inverse de cosinus (IDCT). Les blocs sont ensuite réordonnés afin de reconstituer les images originales et les envoyer vers une sortie pour leur affichage.

Afin de porter cette application sur notre plate-forme, nous avons utilisé trois tâches : Fetch, Reorder et IDCT. Fetch découpe les images en blocs et applique l'algorithme de Huffman. IDCT effectue les transformations discrètes. Reorder réordonne et reconstitue les images décodées. Les traitements des données du flux d'images sont synchronisés en utilisant nos structures de synchronisation (conditions). Ces conditions sont utilisées pour implémenter un modèle de producteur-consommateur [Jef93] sur un tampon en mémoire qui contient un ensemble d'images. L'état du tampon en mémoire est non-déterministe. Afin de reproduire une exécution, nous collectons donc les données nécessaires pour reproduire l'activité de synchronisation.

8.4.1 Intrusion

Nous avons utilisé les cinq métriques suivantes pour évaluer l'intrusion :

- *Temps d'exécution natif* : temps d'exécution moyen du logiciel sans utiliser ReDSoC.
- *Temps d'exécution de référence* : temps d'exécution moyen du logiciel en collectant des traces de ré-exécution déterministe.
- *Surcoût :* Surcoût en pourcentage de l'exécution de référence par rapport à l'exécution native.
- *Taille des données collectées* : taille de la trace de ré-exécution déterministe.
- *Nombre d'entrées dans la trace* : nombre d'entrées dans la trace.

Plate-forme embarquée

Sur le tableau 8.1, nous présentons les métriques d'intrusion relevées pour le jeu de Tetris et pour le décodeur MJPEG. L'intrusion du jeu de Tetris est évaluée

1. L'application a été mise en œuvre dans [APcDG05], dans le cadre du développement d'une plate-forme de simulation *cycle-accurate*

Logiciel	Temps d'exécution natif (s)	Temps d'exécution de référence (s)	Surcoût (%)	Taille des données collectées (Ko)	Nombre d'entrées dans la trace
MJPEG					
*Noeud*1	139	144	3.59	2298	45471
Tetris					
*Noeud*1	62	62	<1%	333	877
*Noeud*2	60	60	<1%	201	530

TABLE 8.1 – Intrusivité due à la collecte de données pour la ré-exécution déterministe.

séparément sur les deux nœuds. En effet, sur ces deux nœuds, deux instances du mécanisme de collecte de données sont exécutées séparément.

Afin de comparer les temps d'exécution natifs et de référence, les deux exécutions doivent recevoir les mêmes entrées. Pour l'application du décodage, nous avons utilisé la même vidéo. Pour le logiciel de jeu, nous avons enregistré les touches du clavier durant une exécution. Ensuite, durant l'exécution native et de référence nous avons fourni les entrées enregistrées.

Le surcoût dans l'application MJPEG est d'environ 3.6%, dû à l'utilisation de plusieurs directives de synchronisation pour le décodage de chaque image. Nous n'avons pas remarqué une différence visuelle entre une exécution normale et une qui est tracée. L'intrusion est 1.7X plus importante que la moyenne rapportée dans RecPlay [RD99] qui utilisent la même méthode de ré-exécution déterministe de la synchronisation. Cependant, cette intrusion est justifiée en considérant le nombre important d'entrées dans la trace (45417) sur la durée de l'exécution (144s).

Nous avons considéré que l'intrusion du jeu de Tetris est insignifiante puisque le surcoût du temps dû au traçage est négligent. En effet, les contraintes de temps font que le calcul (déplacement des pièces) est faible par rapport à l'inactivité (attente entre deux déplacements). Ce temps d'inactivité est utilisé pour le traçage. En conséquence, les temps d'exécution native et de référence sont les mêmes.

Le traçage ne dépasse pas le temps d'inactivité pour deux raisons. La première raison est la faible quantité des volumes de traces générées par les deux nœuds (333ko et 201ko respectivement). La deuxième est la régularité de l'exécution qui fait que la collecte des données est régulièrement distribuée sur toute la durée de l'exécution. En conséquence, il n'y a pas de périodes sur l'exécution du logiciel où l'outil de traces est fortement sollicité dépassant le temps d'inactivité et modifiant le comportement de l'exécution.

Durant les exécutions du jeu de Tetris, nous n'avons observé aucune différence du comportement erroné entre une exécution native et de référence. Cette observation confirme la faible intrusion due à la collecte des traces.

Plate-forme NUMA

Pour évaluer les performances de nos outils de mise au point sur l'application de mosaïque vidéo, nous avons effectué plusieurs exécutions avec des paramètres en entrée différents (plusieurs flux A/V de taille et de formats différents). Sur l'intégralité des expériences, l'intrusivité durant la phase de collecte de données est insignifiante (<0.1%). En effet, l'enregistrement des données pour la ré-exécution déterministe utilise des cycles des processeurs d'inactivité entre l'affichage de deux images. Cette inactivité est imposée aux processeurs afin de respecter le taux d'images par seconde.

À titre d'exemple, nous avons exécuté le logiciel avec comme paramètre trois mêmes flux A/V. Ces flux A/V sont au format MJPEG et de taille 57Mo. L'exécution dure 31s et génère 500Ko de traces. Nous n'avons pas observé aucun ralentissement, dégradation du flux A/V ou changement de la fréquence de l'occurrence de l'erreur que nous avons mis au point. De plus, le comportement régulier de l'exécution fait que le surcoût de la collecte de données est équitablement réparti sur toute la durée de l'exécution. Il n'y a donc pas de périodes de l'exécution qui sont susceptibles d'être fortement perturbées. En conséquence, dans ce cas, nous en avons conclu que les mécanismes de collecte de données n'ont pas d'impact sur la capacité de nos outils de mettre au point les exécutions.

L'impact des outils de mise au point sur les occurrences de l'erreur est difficile à mesurer surtout lorsque les erreurs sont non-déterministes. En effet, la fréquence d'occurrence de ces erreurs peut être indéterminée due au non-déterminisme. Cependant, le surcoût en terme de temps d'exécution entre une exécution normale et une qui utilise les outils de mise au point est un bon indicateur de l'intrusion. Lorsque ce temps accroît, le nombre de cycles processeurs utilisés dans les outils de mise au point est plus important durant l'exécution du logiciel. En conséquence, la probabilité de modifier l'occurrence de l'erreur accroît.

Nous devons poursuivre nos travaux sur l'étude de l'intrusion notamment sur un plus grand ensemble de logiciels erronés. Dans les études que nous avons présentées, tous les logiciels ont un comportement régulier. Ce comportement est représentatif pour une grande partie des logiciels embarqués mais n'est pas général. Dans les deux cas de logiciels avec des contraintes temporelles l'intrusion était insignifiante mais on ne peut pas conclure ce résultat dans le cas général.

Un vrai effort doit être fait pour définir des autre critères d'évaluation de l'intrusion que le surcoût en temps d'exécution. Ces critères doivent être appliqués dans plusieurs cas, dans un contexte industriel afin de tirer des conclusions d'une précision élevée.

8.4.2 Ralentissement du débogage

Nous avons utilisé trois métriques pour évaluer le ralentissement du débogage.

- *Temps d'exécution natif* : temps d'exécution moyen du logiciel sans utiliser ReDSoC.
- *Délai d'attente pour déboguer* : ce délai correspond à une ré-exécution déterministe et à une ré-exécution partielle jusqu'à l'intervalle de temps à déboguer. Il s'agit de représenter le surcoût en terme de temps d'attente de notre approche par rapport à un débogage classique sur un même intervalle de temps. Nous avons pris le pire cas de la ré-exécution partielle où l'intervalle de temps à déboguer est à la fin de l'exécution.
- *Taille des données enregistrées* : données nécessaires pour la ré-exécution partielle. Ces données sont enregistrées durant la ré-exécution déterministe.

Logiciel	Temps d'exécution natif (s)	Délais d'attente pour déboguer (s)	Taille des données enregistrées (ko)
MJPEG			
$Node1$	139	149	0
Tetris			
$Node1$	62	161	2.7
$Node2$	60	156	2.9

TABLE 8.2 – Performances de la ré-exécution déterministe et partielle.

Sur le tableau 8.2, nous montrons le délai d'attente pour déboguer ainsi que la taille des données enregistrées pour les deux logiciels qui font l'objet de notre étude. Ces deux métriques sont en effet proportionnelles puisque le délai d'attente augmente lorsque la taille des données enregistrées accroît. Puisque cette taille dépend du nœud sélectionné, nous l'évaluons sur les deux nœuds en séparation pour le jeu de Tetris.

Nous observons que le délai d'attente pour le débogage du Nœud1 (161s) et de Nœud0 (156s) du logiciel de jeu est approximativement 2X plus important que celui de l'exécution native (62s et 60s respectivement). En effet, le logiciel est ré-exécuté deux fois. La première pour collecter les données pour la ré-exécution partielle. La deuxième pour la ré-exécution partielle. Nous pensons qu'un délai d'attente deux fois plus important que celui de l'exécution natif est important. Afin de réduire ce délai, nous envisageons d'utiliser des points de sauvegarde (checkpoints) périodique durant l'exécution de référence. En conséquence, le délai d'attente serait réduit à 2X le temps d'exécution entre deux points de sauvegarde. Cependant, la fréquence

des points de sauvegarde représenterait un compromis entre l'intrusion et le délai d'attente pour le débogage.

Le décodeur MJPEG est exécuté sur un seul nœud. Le délais maximum de débogage correspond donc à une ré-exécution déterministe. Le temps de ré-exécution déterministe (149s) est plus élevé mais acceptable par rapport à celui de l'exécution native (139s). Ce surcoût est dû l'ordonnancement forcé des opérations de synchronisations.

Sur l'application de mosaïque vidéo, les cycles d'inactivité effectués afin de respecter les taux d'affichage d'images sont suffisants pour collecter des données et effectuer la ré-exécution déterministe et partielle. En conséquence, lorsque l'intervalle de temps à mettre au point est au début de l'exécution, l'attente avant le débogage corresponde à une exécution déterministe. Lorsque cet intervalle est à la fin de l'exécution, le développeur doit attendre le temps de la ré-exécution déterministe et partielle. Par exemple, sur l'application de mosaïque vidéo, l'attente maximale avant le débogage pour notre exécution qui dure 31s est de 62s.

Le délai d'attente dépend notamment du temps de ré-exécution déterministe du logiciel, du temps de collecte des données pour la ré-exécution partielle et du positionnement de l'intervalle de temps à déboguer. Lorsque cet intervalle est à la fin de l'exécution, le délai d'attente peut dépasser 3x le temps d'exécution normale. Ce délai est inacceptable lorsque le logiciel d'une longue durée. Cependant, comme pour l'application de Tetris, ce délai peut être réduit en utilisant des points de sauvegarde.

8.5 Simplicité d'utilisation

Dans notre prototype, nous nous sommes préoccupés de mettre en valeur les nouveaux concepts sans tenir compte de la simplicité d'utilisation. Dans cette section, notre objectif n'est pas de mesurer la simplicité d'utilisation mais de discuter les possibilité de rendre automatique les actions manuelles. Les deux actions, de configuration des outils de mise au point, ainsi que d'identification et de sélection d'une partie de l'exécution à mettre au point sont manuelles. Dans la suite, nous discutons les possibilités de simplifier ces actions en les rendant automatiques.

8.5.1 Configuration des outils de mise au point

Notre prototype actuel nécessite la création d'un fichier de configuration par nœud qui est modifié après chaque étape de la mise au point. Nous modifions pour l'instant manuellement ce fichier afin d'indiquer :

- Identifiant du nœud.

- Phase de la mise au point : exécution de référence, ré-exécution déterministe ou ré-exécution partielle.
- Ensemble de nœuds soupçonnés fautifs : cette donnée n'est pas nécessaire lorsque la phase est l'exécution de référence puisque l'occurrence de l'erreur n'a pas encore eu lieu.

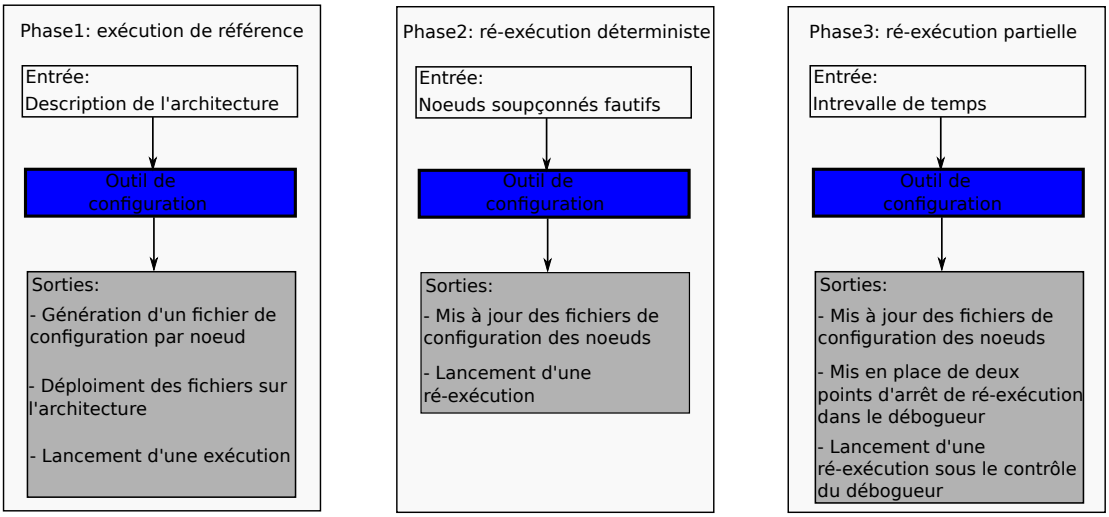


FIGURE 8.1 – Programme de configuration de ReDSoC.

Cette procédure manuelle pourrait être automatisée en utilisant un outil visuel de configuration comme celui montré sur la figure 8.1. Cet outil configure ReDSoC selon la phase de mise au point :

- Phase 1 : L'outil prend en entrée une description de l'architecture qui comprend plus particulièrement les identifiants des nœuds, ainsi que leur topologie. En sortie, l'outil crée un fichier de configuration par nœud et lance l'exécution. Le fichier contient l'identifiant du nœud et indique qu'il s'agit de la première phase de mise au point.
- Phase 2 : Après avoir analysé l'exécution de référence, les développeurs fournissent à l'outil les nœuds soupçonnés fautifs. L'outil modifie la phase 1 en phase 2 et ajoute une liste de nœuds soupçonnés fautifs dans les fichiers de configuration des nœuds avant de relancer le logiciel MPSoC.
- Phase 3 : Les développeurs spécifient en entrée de l'outil, l'intervalle de temps à mettre au point. La phase dans les fichiers de configuration des nœuds est modifiée à 3 avant de relancer l'exécution sous le contrôle du débogueur, fournit avec les bornes de l'intervalle de temps à mettre au point.

Après la phase trois, l'outil peut attendre en entrée la sélection d'une nouvelle partie de l'exécution à déboguer et continuer jusqu'à ce que le développeur indique

que l'erreur est identifiée.

Certains nœuds peuvent ne pas avoir accès à un système de fichiers. Dans ce cas, il faudrait modifier ReDSoC en ajoutant une phase d'initialisation au lancement du programme. Durant cette phase, les nœuds qui ont accès à un système de fichiers vont transférer les informations de configuration aux autres nœuds.

La sélection d'une partie de l'exécution qui consiste à entrer la liste de nœuds soupçonnés fautifs, ainsi que les deux bornes de l'intervalle de temps peut être simplifiée en utilisant l'outil de visualisation. Il s'agit d'encadrer avec le pointeur de la souris, les nœuds et l'intervalle de temps à partir de la visualisation. Lorsque cet encadrement est effectué, l'outil de visualisation fournit automatiquement les entrées correspondantes à l'outil de configuration, selon la phase de mise au point.

8.5.2 Identification d'une partie à mettre au point

Afin d'identifier la partie erronée de l'exécution, nous considérons qu'une exécution est correcte lorsque ces paramètres de sorties sont correctes. Lorsqu'une à plusieurs sorties ne sont pas correctes, les développeurs doivent choisir une partie de l'exécution à déboguer.

Nœuds

Dans le cas général des MPSoC, les sorties erronées sont un indicateur pertinent pour cibler un ensemble de nœuds qui ont exécuté un code potentiellement fautif. En effet, les différentes parties du logiciel sont attentivement déployées par les développeurs sur les différents nœuds durant l'exécution, afin d'optimiser l'exploitation de l'architecture. En conséquence, durant les tests unitaires, lorsqu'un paramètre de sortie est erroné, les développeurs pourront aisément identifier des nœuds impliqués dans cette sortie erronée. Il est donc possible d'ajouter dans les tests unitaires des informations sur le déploiement du logiciel afin de pouvoir identifier automatiquement le nœud concerné, lorsqu'une sortie est erronée. Cependant, cette automatisation nécessite de définir plusieurs formalismes.

Dans les deux études de cas, nous avons pu facilement cibler des nœuds potentiellement fautifs en observant les sorties des exécutions. Dans l'étude sur le jeu d'arcade, deux nœuds exécutent deux instances du jeu. Nous avons facilement ciblé le nœud fautif qui correspondait à celui qui exécutait l'instance du jeu erronée. Cependant, suite à une analyse en utilisant des points d'arrêt, nous avons ciblé l'autre nœud qui était le véritable fautif. Sur le logiciel de mosaïque A/V, nous avons exécuté trois lecteurs et un serveur A/V sur quatre nœuds respectifs. Nous avons facilement ciblé le nœud fautif qui correspondait au lecteur bloqué.

Intervalle de temps

Nous avons fait le choix de sélectionner un intervalle de temps en utilisant des traces de ré-exécution déterministe. Ces traces montrent une partie du comportement de l'exécution durant le temps et peuvent donc représenter un comportement anormal. Parmi les erreurs que ces traces peuvent représenter, nous en avons identifié quatre :

- Blocages : les occurrences des entrées dans la trace de ré-exécution déterministe ne sont plus générées après le moment de blocage.
- Violation de l'ordre des accès aux structures partagées : les traces comportent l'ordre des accès aux structures partagées.
- Communication désordonnées : les traces montrent l'ordre des envois et des réceptions des messages.
- Violation d'échéances temporelles : présence d'entrées dans les traces anormalement éloignées ou reprochées.

Durant le débogage de l'application de mosaïque vidéo, la visualisation des traces a représenté une situation d'inter-blocage. Nous avons pu identifier un intervalle de temps qui correspondait à cette situation et analyser rapidement le comportement anormal afin d'identifier la source de l'erreur. Dans [LPSZ08], les auteurs montrent que 53% des erreurs de concurrence sont dues aux blocages, aux inter-blocages et à un ordre non prévu d'accès aux structures de synchronisation. Puisque ces erreurs représentent un comportement anormal dans les traces, la réduction de l'intervalle de temps peut simplifier le débogage dans un large ensemble de cas.

L'observation de nos traces peut relever également d'autres erreurs comme dans l'étude de cas sur le jeu d'arcade. Dans cette étude, nous nous sommes attendu d'avoir une trace régulière comme le comportement du logiciel. Cependant, nous avons observé des entrées dans la trace, anormalement agrégées avant l'occurrence de l'erreur. Cette anomalie a été un indicateur de l'erreur et nous a permis de réduire l'intervalle de temps à déboguer.

Un problème pertinent lors de l'analyse de traces pour identifier un comportement anormal est le nombre important d'entrées dans la trace et leur affichage. En effet, ces entrées sont agrégées pour être affichées. Lorsque les développeurs s'intéressent à un intervalle de temps spécifique la zone agrégée peut être agrandie. Cependant, l'examen de toutes les entrées est souvent impossible et des erreurs peuvent rester cachées. Un approche prometteuse [LCH+09, CW10, ZXHW10] exploite des possibilités d'utiliser de la fouille de données [KZ02] pour trouver des motifs erronés dans les traces. Ces patterns peuvent être utilisés pour automatiquement sélectionner un intervalle de temps à mettre au point.

8.6 Synthèse

Dans ce chapitre, nous avons évalué notre méthodologie de mise au point MPSoC selon quatre critères, notamment la mise au point d'erreurs non-déterministes, le passage à l'échelle, la simplicité d'utilisation et la performance.

Les mécanismes de ré-exécution déterministe que nous avons mis en place nous ont permis de déboguer des erreurs non-déterministes dans les études de cas des applications de jeu et de mosaïque vidéo. En effet, nous avons montré que ces mécanismes ont permis de reproduire le comportement de l'exécution erronée. Notre solution a donc simplifié la mise au point en permettant d'analyser le comportement anormal de cette exécution erronée avec un débogueur.

Nous avons montré que la ré-exécution partielle simplifie la complexité de la mise au point dû au passage à l'échelle des MPSoC. Dans l'étude de cas sur la plateforme réelle, nous avons constaté que le débogage d'un nœud durant l'exécution de l'autre perturbe l'exécution et peut créer de nouvelles erreurs. Sur la plateforme MPSoC à 16 processeurs nous avons constaté qu'il est difficile d'analyser un code exécuté d'une manière répétitive par plusieurs processeurs. En ré-exécutant une partie des nœuds et dans un intervalle de temps, notre solution a simplifié le débogage en réduisant le nombre de processeurs ainsi que de répétitions.

Concernant la simplicité d'utilisation, l'identification de nœuds potentiellement fautifs ainsi que d'intervalle de temps au comportement anormal nécessitant des connaissances de la plate-forme, du déploiement du logiciel sur la plate-forme durant l'exécution, de la structure du logiciel ainsi que des efforts d'analyse de la part des développeurs. Néanmoins, dans les deux cas de mise au point présentés, l'identification d'une partie soupçonnée fautive n'a pas posé de difficultés particulières.

Concernant les performances, nous avons jugé l'intrusion dans l'intégralité des études de cas satisfaisante (0%-3,6%). Cependant, nous avons besoin de généraliser nos résultats sur un plus large ensemble de logiciels. Dans notre approche, les délais de débogage sont importants. L'utilisation de points de sauvegarde peut réduire ces délais mais augmentera l'intrusion. Des études devront être effectuées afin de trouver le meilleur compromis entre l'intrusion et le délai de débogage.

Chapitre 9

Conclusion et perspectives

Les plates-formes embarquées, utilisées dans les domaines du multimédia et de la télécommunication, nécessitent de plus en plus de puissance de calcul mais doivent garantir une consommation énergétique faible. Le progrès incroyable de la technologie des semi-conducteurs a permis le passage à l'échelle des plates-formes avec un impact minimal sur la consommation énergétique. Les architectures MPSoC qui en résultent intègrent des centaines de processeurs, des blocs de mémoire et des périphériques interconnectés par un réseau complexe. Le logiciel est exécuté par de multiples groupes d'entités qui utilisent des mécanismes d'ordonnancement, de communication et de synchronisation afin d'exploiter au mieux l'architecture. Ces mécanismes créent des interactions complexes qui posent de nouveaux défis pour les outils de mise au point.

Nous avons identifié deux problèmes majeurs pour le débogage du logiciel MPSoC : les erreurs non-déterministes et le passage à l'échelle. Les erreurs non-déterministes peuvent apparaître durant une exécution normale et disparaître pendant le débogage. Ce comportement est dû aux interactions entre les processeurs et aux accès des entrées d'un environnement extérieur. En effet, l'ordre des interactions entre processeurs peut changer entre deux exécutions puisque ces processeurs exécutent le code indépendamment. En ce qui concerne les entrées reçues, il est rare que l'on puisse effectuer plusieurs exécutions dans le même environnement d'exécution.

Dans une exécution erronée, qui est caractérisée par multiples entités d'exécution en interaction, il peut être complexe d'identifier un point de départ et d'effectuer les analyses de l'erreur pas à pas. D'une part, il est possible qu'un point du code est atteint d'une manière répétitif par plusieurs entités d'exécution. D'autre part, chaque pas de l'exécution peut être effectué par une entité différente et faire partie d'une des multiples interactions. Il est donc particulièrement difficile de se focaliser sur la partie erronée de l'exécution et de l'analyser.

Nos objectifs sont donc de concevoir des mécanismes qui permettent d'une part, de reproduire les exécutions MPSoC erronées afin de pouvoir les déboguer. D'autre part, nous devons pouvoir se focaliser sur une partie des entités d'exécution afin de permettre le passage à l'échelle. De plus, les mécanismes que nous proposons doivent tenir compte des contraintes des applications embarquées, dont la plus importante est la maîtrise de l'intrusion.

Contribution : méthodologie de mise au point MPSoC

Dans cette thèse, nous avons introduit une nouvelle méthodologie de mise au point qui simplifie le débogage du logiciel MPSoC. Dans cette méthodologie, un cycle de mise au point permet aux développeurs de sélectionner et de déboguer successivement différentes parties d'une exécution erronée.

Nous avons contribué en identifiant les sources de non-déterminisme MPSoC et en sélectionnant des méthodes de ré-exécution déterministe, afin de reproduire une exécution erronée. Nous avons également contribué en définissant deux critères qui identifient une partie de l'exécution à circonscrire et guident une méthode de ré-exécution partielle. Nous avons conçu un prototype ReDSoC qui permet d'effectuer le cycle de mise au point. Finalement, nous avons validé notre méthodologie, ainsi que ReDSoC sur deux études de cas qui montrent la mise au point d'erreurs non-déterministes sur des plate-formes MPSoC. Dans la suite, nous détaillons nos contributions sur la ré-exécution déterministe et la ré-exécution partielle.

- *Ré-exécution déterministe* : nous avons sélectionné des mécanismes qui permettent de reproduire la couche applicative d'un large ensemble de logiciels MPSoC en maîtrisant l'intrusion. Plus particulièrement, nous avons identifié quatre sources de non-déterminisme dans les MPSoC : la communication réseau, les accès aux données partagées, l'ordonnancement et les accès aux entrées/sorties.

  Afin de reproduire la communication, nous avons utilisé deux méthodes. La première méthode est basée sur des horloges vectorielles afin de reproduire l'ordre des transferts de messages. Cette méthode détecte et trace uniquement les transferts non-déterministes afin de réduire l'intrusion. La deuxième méthode reproduit la taille des données reçues lors des transferts de messages.

  Nous avons reproduit la synchronisation en utilisant des horloges de Lamport. Ces horloges nous ont permis de détecter quand l'ordre des accès aux structures de synchronisation est non-déterministe. Le traçage, effectué suite à cette détection, concerne donc qu'une partie des accès, réduisant ainsi l'intrusion. Nous n'avons pas reproduit les accès non-synchronisés à la mémoire

dû à leur important impact sur l'intrusion. En effet, dans le cas général, la fréquence d'occurrence des accès non-synchronisés à la mémoire peut être très importante. Leur traçage peut donc perturber les logiciels avec contraintes temporelles.

Nous avons reproduit le mécanisme d'attente active, fréquemment utilisé dans la couche applicative afin d'acquérir les entrées du logiciel. Nous avons plus particulièrement reproduit les opérations non-bloquantes de lecture des donnes qui sont la source de non-déterminisme dans l'attente active. Nous n'avons pas reproduit les interruptions, qui sont également utilisées pour acquérir les entrées, mais sont utilisés dans la couche système qui est de bas niveau et souvent ne concerne pas la couche applicative. Pour la même raison, nous ne reproduisons pas l'ordonnancement.

Nous avons implémenté les méthodes de ré-exécution déterministe proposés dans notre prototype ReDSoC. Nous avons appliqué ces méthodes sur des mécanismes de communication, de synchronisation et d'entrées sorties largement utilisés, notamment basés sur le standard POSIX.

Dans les chapitres 6, 7, 8, nous avons montré que la sélection de méthodes de ré-exécution déterministe nous a permis de reproduire intégralement les exécutions de trois applications effectuées sur deux plate-formes MPSoC. Nous avons montré que notre solution induit une intrusion maîtrisée sur l'exécution du logiciel qui nous a permis d'appliquer notre solution sur des logiciels de traitement multimédia qui tolèrent une faible perturbation due aux contraintes d'échéances.

- *Ré-exécution partielle* : nous avons défini deux critères qui permettent d'identifier et de cibler la partie supposée fautive de l'exécution, ainsi qu'une méthode qui permet de circonscrire cette partie. Le premier critère concerne les entités d'exécution d'un sous-ensemble des nœuds de l'architecture. Nous avons choisi ce critère suite à nos observations sur les inspirations des développeurs. Dans le cas du jeu de Tetris par exemple, les développeurs ont soupçonné le nœud exécutant l'instance du jeu qui a échouée. Durant la mise au point du logiciel de mosaïque vidéo, les développeurs ont voulu se concentrer sur le nœud exécutant le lecteur vidéo bloqué.

  Le deuxième critère permet de considérer un intervalle de temps durant l'exécution. Ce critère permet de bien restreindre la quantité d'informations à considérer et la durée de débogage. Par exemple, dans l'étude de cas sur le logiciel de mosaïque vidéo, une situation d'inter-blocage a été instantanément observée en visualisant les traces. L'intervalle de temps sélectionné délimitait les deux accès à une structure de synchronisation, responsables de cet inter-blocage.

  Nous avons conçu une nouvelle méthode de ré-exécution partielle qui est indépendante de la plate-forme et du logiciel. Cette méthode prend en en-

trée les traces de ré-exécution déterministe, un ensemble d'identifiants de nœuds et deux entrées de traces qui bornent un intervalle de temps de l'exécution. Dans une première phase, des données sont collectées pour pouvoir ré-exécuter en isolation les nœuds choisis. Durant une deuxième phase, seuls ces nœuds sont ré-exécutés. L'intervalle de temps est détecté en comparant les traces durant leur ré-exécution avec les bornes de l'intervalle. Le débogage est effectué entre ces deux bornes.

A part l'implémentation directe des mécanismes définis dans notre méthodologie, nous avons défini un nouveau type de point d'arrêt - *replay-breakpoint*. Ces *replay-breakpoints* s'activent lors de lectures spécifiques des traces de ré-exécution déterministe. Nous avons fourni une extension à GDB qui utilise ces points d'arrêt afin de détecter un intervalle de temps à mettre au point.

Perspectives

La conception de la solution de mise au point MPSoC que nous avons proposée a de multiples perspectives. Nous en présentons une première à court terme et une deuxième à plus long terme.

- *Déporter ReDSoC sur la plate-forme de développement* : il serait plus judicieux de déporter une plus grande partie des outils de ReDSoC sur la plate-forme de développement afin de réduire l'utilisation de ressources de débogage sur la MPSoC. Plus particulièrement, il s'agit de déporter les mécanismes de ré-exécution déterministe et partielle dans la partie client du débogueur.

  Un point de départ est d'étudier les possibilités d'utiliser des points d'arrêt sur des événements qui concernent la ré-exécution déterministe et partielle. Lorsque ces points sont atteints, il faudrait définir les données à transférer entre la partie client et la partie serveur du débogueur.

- *Automatiser l'identification d'une partie anormale de l'exécution* : La recherche d'une partie anormale de l'exécution pourrait être automatisée en utilisant de la fouille de donnée sur les traces. La fouille de donnée serait utilisée afin d'identifier automatiquement des motifs anormaux parmi les données collectées. Par exemple, un tel motif peut être détecté lorsqu'une tâche sort d'une barrière de synchronisation avant l'entrée de toutes les autres. Dans ce cas, la partie de l'exécution sélectionnée pourrait être délimitée par le nœud exécutant la barrière et par l'intervalle de temps entre la première entrée dans la barrière et la sortie inattendue.

Table des figures

Bibliographie

[AHC09] Y. Ahn, Y.S. Hwang, and K.S. Chung. Flexible framework for dynamic management of multi-core systems. In *SoC Design Conference (ISOCC), 2009 International*, pages 237–240. IEEE, 2009.

[AL94] K.M.R. Audenaert and L.J. Levrouw. Interrupt replay : a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10) :601–612, 1994.

[APcDG05] Ivan Augé, Frédéric Pétrot, François. Donnet, and Pascal Gomez. Platform-Based Design From Parallel C Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12) :1811–1826, December 2005.

[ARM] ARM. Cortex-a9 processor. `http://bit.ly/bHOomu`.

[AS09] G. Altekar and I. Stoica. Odr : output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206. Citeseer, 2009.

[BBMP06] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling data race detection in the intel thread checker. In *First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), March*, volume 26, 2006.

[Bel05] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.

[BG91] D.F. Bacon and S.C. Goldstein. Hardware-assisted replay of multiprocessor programs. *ACM SIGPLAN Notices*, 26(12) :206, 1991.

[BHMC10] U.D. Bordoloi, H.P. Huynh, T. Mitra, and S. Chakraborty. Design space exploration of instruction set customizable mpsocs for multimedia applications. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 170–177. IEEE, 2010.

[CB89] B. Charron-Bost. Combinatorics and geometry of consistent cuts : Application to concurrency theory. *Distributed algorithms*, pages 45–56, 1989.

[CDK05] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems : concepts and design*. Addison-Wesley Longman, 2005.

[CFMR95] C. Clemencon, J. Fritscher, M. Meehan, and R. Rühl. An implementation of race detection and deterministic replay with mpi. *EURO-PAR'95 Parallel Processing*, pages 155–166, 1995.

[CM93] S.J. Clarke and J.A. McDermid. Software fault trees and weakest preconditions : a comparison and analysis. *Software Engineering Journal*, 8(4) :225–236, 1993.

[Com10] The Symbian Foundation Community. Symbian operating system, 2010. `http://www.symbian.org/`.

[Con] ObjectWeb Consortium. Cecilia framework. `http://fractal.ow2.org/cecilia-site/current/`.

[CRV94] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. Testtube : A system for selective regression testing. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 211–220. IEEE, 1994.

[CSP04] Inc Cisco Systems and Cisco Networking Academy Program. *Cisco Networking Academy Program : CCNA 1 and 2 companion guide*. Cisco Networking Academy Program series. Cisco Press, 2004.

[CW96] M. Campione and K. Walrath. *The Java tutorial : object-oriented programming for the Internet*, volume 201634546. Addison-Wesley, 1996.

[CW10] P.H. Chang and L.C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 607–612. IEEE, 2010.

[DBL03] Open multimedia platform for next-generation mobile devices. In *PATMOS*, page 196, 2003.

[Dis] The Angstrom Linux Distribution. `http://www.angstrom-distribution.org/`.

[dOSdKM02] B. de Oliveira Stein, J.C. de Kergommeaux, and G. Mounié. Pajé trace file format. Technical report, ID-IMAG, Grenoble, France, 2002. `http://sourceforge.net/projects/paje/`.

[EDN00] T. Ebrahimi, F. Dufaux, and Y. Nakaya. Multimedia Systems, Standards, and Networks. In *Lecture Notes in Computer Science*. Marcel Dekker Publishing, 2000.

[ESL01] M. El Shobaki and L. Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Quality Electronic Design, 2001 International Symposium on*, pages 56–61. IEEE, 2001.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[FKH+08] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S.S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 189–198. ACM, 2008.

[GASS06] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*, pages 27–27. USENIX Association, 2006.

[GCRDB04] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec : a portable record/replay environment for multi-threaded java applications. *Software : Practice and Experience*, 34(6) :523–547, 2004.

[Gdb] Gdbserver. `http://www.delorie.com/gnu/docs/gdb/gdbserver.1.html`.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Springer, 2002.

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message passing interface*. 1999.

[Gra] Mentor Graphics. Sourcery codebench lite edition. `http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/`.

[Gum] Gumstix. Development of advanced computer-on-module products. `http://www.gumstix.com/about.html`.

[GWT+08] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M.F. Kaashoek, and Z. Zhang. R2 : An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 193–208. USENIX Association, 2008.

[Hil] Green Hills. Multi integrated development environment. `http://www.ghs.com/products/MULTI_IDE.html`.

[HMC+09] D.R. Hower, P. Montesinos, L. Ceze, M.D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Communications of the ACM*, 52(6) :93–100, 2009.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9) :1098 –1101, sep. 1952.

[Hus02] J. Huselius. Debugging parallel systems : A state of the art report. *MRTC Report no*, 63, 2002.

[HV99] M. Henning and S. Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Ins] Texas Instruments. Omap5912. `http://www.ti.com/product/omap5912`.

[Ins81] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http ://rfc.sunsite.dk/rfc/rfc793.html.

[JBPT09] A. Jannesari, K. Bao, V. Pankratius, and W.F. Tichy. Helgrind+ : An efficient dynamic race detector. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–13. IEEE, 2009.

[Jef93] K. Jeffay. The real-time producer/consumer paradigm : A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing : states of the art and practice*, pages 796–804. ACM, 1993.

[JTA] JTAG. Joint test action group. `http://www.siliconfareast.com/jtag.htm`.

[KK93] L.V. Kale and S. Krishnan. Charm++ : a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[Kle05] A. Kleen. A numa api for linux. *Novel Inc*, 2005.

[Kra00] D. Kranzlmüller. Event graph analysis for debugging massively parallel programs. *Institute of Graphics and Parallel Processing, Johannes Kepler Universitat Linz, Austria*, 2000.

[KSC00] R. Konuru, H. Srinivasan, and J.D. Choi. Deterministic replay of distributed java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 219–227. IEEE, 2000.

[ksld] Linux kernel source level debugger. `http://kgdb.linsyssoft.com/`.

[KSS96] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads*. SunSoft Press, 1996.

[KZ02] W. Klosgen and JM Zytkow. Handbook of data mining and knowledge discovery. *Recherche*, 67 :02, 2002.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[LAVC94] L.J. Levrouw, K.M.R. Audenaert, and J.M. Van Campenhout. A new trace and replay system for shared memory programs based on lamport clocks. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 471–478. IEEE, 1994.

[LCH+09] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection : a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 557–566, New York, NY, USA, 2009. ACM.

[LMC87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, 100(4) :471–482, 1987.

[LPSZ08] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes : a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[LWV+10] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P.M. Chen, and J. Flinn. Respec : efficient online multiprocessor replayvia speculation and external determinism. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 77–90. ACM, 2010.

[Man01] D. Mann. Processor including a combined parallel debug and trace port and a serial port, January 16 2001. US Patent 6,175,914.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[MCL89] J.M. Mellor-Crummey and T.J. Leblanc. A software instruction counter. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 78–86. ACM, 1989.

[MKSR09] J.C. Maeng, J.I. Kwon, M.K. Sin, and M. Ryu. Rt-replayer : a record-replay architecture for embedded real-time software debugging. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1670–1675. ACM, 2009.

[MP96] E. Maillet and B. Plateau. Le traçage logiciel d'applications parallèles : conception et ajustement de qualité. 1996.

[MSP07] D. Mangano, A. Scandurra, and C. Pistritto. Relieving physical issues in new noc-based socs. In *Proceedings of the 2nd international conference on Nano-Networks*, page 10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[MV08] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2) :S10, 2008.

[MW01] D.P. Mann and C.K. Wakeland. Software debug port for a microprocessor, February 6 2001. US Patent 6,185,732.

[NBK99] M. Neyman, M. Bukowski, and P. Kuzora. Efficient replay of pvm programs. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 676–676, 1999.

[Net91] R.H.B. Netzer. *Race condition detection for debugging shared-memory parallel programs.* PhD thesis, University of Wisconsin, 1991.

[Net93] R.H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR*

workshop on Parallel and distributed debugging, pages 1–11. ACM, 1993.

[Nil] Victor Nilsson. Vitetris. `http://www.victornils.net/tetris/`.

[NM92] R.H.B. Netzer and B.P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 502–511. IEEE Computer Society Press, 1992.

[OTGH] USB On-The-Go and Embedded Host. `http://www.usb.org/developers/onthego/`.

[Pen03] B. Pendleton. Game programming with the sdl. *Linux Journal*, 2003(110) :1, 2003.

[PMG+09] Carlos Prada, Vania Marangozova, Kiril Georgiev, Jean-François Mehaut, and Miguel Santana. Towards a Component-based Observation of MPSoC. In *Proceedings of 4th IEEE International Symposium on Embedded Multicore Systems-on-Chip*, sep 2009.

[PPB02] P.G. Paulin, C. Pilkington, and E. Bensoudane. Stepnp : A system-level exploration platform for network processors. *Design & Test of Computers, IEEE*, 19(6) :17–26, 2002.

[PPL+04] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor soc platform applied to high-speed traffic management. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53. ACM, 2004.

[PPS+10] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay : a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[PR11] A. Pura and CV Raghu. Design of a wireless adapter for multimedia projectors. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, pages 1–4. IEEE, 2011.

[PRMG+09] Carlos Prada-Rojas, Vania Marangozova, Kiril Georgiev, Jean-François Méhaut, and Miguel Santana. Observation de systèmes

embarqués : une approche à base de composants. In *Conférence Française sur les Systèmes en Exploitation (CFSE)*, Toulouse, France, sep 2009.

[PRMMG+09] Carlos Hernan Prada Rojas, Vania Marangonzova-Martin, Kiril Georviev, Jean-François Méhaut, and Miguel Santana. Towards a Component-based Observation of MPSoC. Rapport de recherche RR-6905, INRIA, 2009.

[proa] Sandia's MegaTux project. Embedded systems conference (esc) in san jose. `http://www.linuxfordevices.com/c/a/News/Sandia-StrongBox-and-Gumstix-Stagecoach/`.

[Prob] FFmpeg-Based Projects. `http://ffmpeg.org/projects.html`.

[PS04] G. Palermo and C. Silvano. Pirate : A framework for power/performance exploration of network-on-chip architectures. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 521–531, 2004.

[PW01] J. Pedersen and A. Wagner. Correcting errors in message passing systems. *High-Level Parallel Programming Models and Supportive Environments*, pages 122–137, 2001.

[PZX+09] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K.H. Lee, and S. Lu. Pres : probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, October*, pages 11–14. Citeseer, 2009.

[RD99] Michiel Ronsse and Koen De Bosschere. Recplay : a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2) :133–152, 1999.

[RDB99] M. Ronsse and K. De Bosschere. Recplay : a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2) :133–152, 1999.

[RDBC+03] M. Ronsse, K. De Bosschere, M. Christiaens, J.C. de Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9) :62–67, 2003.

[RDd99] M. Ronsse, K. De Bosschere, and J. Chassin de Kergommeaux. Efficient Execution Replay for ATHAPASCAN-0 Parallel Programs. *INRIA*, Research raport(3635), 1999.

[Rib11] Christiane Pousa Ribeiro. *Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms*. PhD thesis, 2011.

[Riva] Wind River. Jtag debugging. `http://www.windriver.com/products/JTAG-debugging/`.

[Rivb] Wind River. Vxworks embedded system kernel. `http://www.windriver.com/products/vxworks/`.

[RZ97] M. Ronsse and W. Zwaenepoel. Execution replay for treadmarks. In *Proceedings of EUROMICRO Workshop on Parallel and Distributed Processing*, 1997.

[Sam11] J.R. Samson. Implementation of a dependable multiprocessor cubesat. In *Aerospace Conference, 2011 IEEE*, pages 1 –10, march 2011.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser : A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4) :391–411, 1997.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Real-time transport protocol. *RFC1899*, 1996.

[Sch98] H. Schulzrinne. Real time streaming protocol (rtsp). 1998.

[SdKB00] Benhur Stein, Jacques Chassin de Kergommeaux, and Pierre-Eric Bernard. Pajé, an interactive visualization tool for tuning multithreaded parallel applications. *Parallel Computing*, 26 :1253–1274, 2000.

[SE] ST-Ericsson. Snowball. `http://www.igloocommunity.org/`.

[SFR04] W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network Programming : The sockets networking API*. Addison-Wesley professional computing series. Addison-Wesley, 2004.

[SGS10] J.E. Stone, D. Gohara, and G. Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3) :66, 2010.

[SK92] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1) :47–52, 1992.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM systems Journal*, 39(1) :175–193, 2000.

[Ste92] W Stevens. *Advanced programming in the UNIX environment*. Addison-Wesley Pub. Co, Reading, Mass, 1992.

[STMa] STMicroelectronics. Os21 embedded system kernel. `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/CD17358306.pdf`.

[STMb] STMicroelectronics. Spear1340. `http://www.st.com/internet/mcu/product/251211.jsp`.

[STM09a] STMicroelectronics. Dynamic Kernel Tracing with KPTrace. Website, 2009. `http://www.stlinux.com/devel/traceprofile/kptrace`.

[STM09b] STMicroelectronics. STLinux Distribution. Website, 2009. `http://www.stlinux.com/`.

[STM10] STMicroelectronics. STWorkbench 4.01. Website, 2010. `http://www.stlinux.com/node/737`.

[TBS+11] Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini. An introduction to multi-core system on chip - trends and challenges. In *Multiprocessor System-on-Chip*, pages 1–21. 2011.

[TCR+10] S.V. Tota, M.R. Casu, M.R. Roch, L. Rostagno, and M. Zamboni. Medea : a hybrid shared-memory/message-passing multiprocessor noc-based architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 45–50, 2010.

[Thaa] H. Thane. Monitoring, testing and debugging of distributed real-time systems. *Department of Machine Design*, page 128.

[Thab] H. Thane. Monitoring, Testing and Debugging of Distributed Real-Time Systems.

[TNP+02] W.T. Tsai, Y. Na, R. Paul, F. Lu, and A. Saimi. Adaptive scenario-based object-oriented test frameworks for testing embedded systems. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 321–326. IEEE, 2002.

[Tom06] S. Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146) :10, 2006.

[Too] Lauterbach Development Tools. Jtag debugger. `http://www.lauterbach.com/frames.html?home.html`.

[TSHP03] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[vdWdKH+04] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors : an interface-centric approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217. ACM, 2004.

[WLW+10] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–206. ACM, 2010.

[WWW06] W. Wolf, W. Wolf, and W. Wolf. *High-performance Embedded Computing : Architectures, Algorithms, and Applications*. Morgan Kaufmann, 2006.

[XBH03] M. Xu, R. Bodik, and M.D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 122–133. IEEE, 2003.

[XLW+09] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz : Subgroup reproducible replay of mpi applications. *ACM SIGPLAN Notices*, 44(4) :251–260, 2009.

[ZXHW10] J. Zou, J. Xiao, R. Hou, and Y. Wang. Frequent instruction sequential pattern mining in hardware sample data. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 1205–1210. IEEE, 2010.

Résumé

Les plates-formes MPSoC permettent de satisfaire les contraintes de performance, de flexibilité et de consommation énergétique requises par les systèmes embarqués émergents. Elles intègrent un nombre important de processeurs, de blocs de mémoire et de périphériques, hiérarchiquement organisés par un réseau d'interconnexion. Le développement du logiciel est réputé difficile, notamment à cause de la gestion d'un grand nombre d'entités d'exécution (tâches/threads/processus). L'exécution concurrente de ces entités permet d'exploiter efficacement l'architecture mais complexifie le processus de mise au point et l'analyse des erreurs. D'une part, des exécutions peuvent être non-déterministes, c'est à dire qu'elles peuvent se dérouler d'une manière différente. En conséquence, il n'est pas garanti qu'une erreur se produirait durant la phase de mise au point. D'autre part, la complexité de l'architecture et de l'exécution peut rendre trop important le nombre d'éléments à analyser afin d'identifier une erreur. Il pourrait donc être difficile de se focaliser sur des éléments potentiellement fautifs. Un des défis majeurs du développement logiciel MPSoC est donc de réduire le temps de la mise au point.

Dans cette thèse, nous proposons une méthodologie de mise au point qui aide le développeur à identifier les erreurs dans le logiciel MPSoC. Notre premier objectif est de déboguer une même exécution plusieurs fois afin d'analyser des sources d'erreurs potentielles. Nous avons donc identifié les sources de non-déterminisme MPSoC et proposé des mécanismes de ré-exécution déterministe adaptés. Notre deuxième objectif vise à minimiser les ressources utilisées à reproduire une exécution afin de satisfaire la contrainte MPSoC de maîtrise de l'intrusion. Nous avons donc utilisé des mécanismes efficaces de ré-exécution déterministe et considéré qu'une partie du comportement non-déterministe. Le troisième objectif est de permettre le passage à l'échelle, c'est à dire de déboguer des exécutions caractérisées par un nombre d'éléments de plus en plus croissant. Nous avons donc proposé une méthode qui permet de circonscrire et de déboguer qu'une partie de l'exécution. De plus, cette méthode s'applique aux différents types d'architectures et d'applications MPSoC. Nous montrons la valeur ajoutée de notre méthodologie par le biais de différentes études de cas qui utilisent plusieurs configurations matérielles et logicielles.

Mots-clés : Débogage, ré-exécution déterministe, ré-exécution partielle, MPSoC.

Abstract

MPSoC platforms provide high performance, low power consumption and flexibility as required by the emerging embedded systems. They incorporate many processing units, memory blocs and peripherals, hierarchically organized by an interconnection network. The software development is known to be difficult, namely due to the management of multiple execution entities (tasks/threads/processes). The concurrent execution of these entities allows to efficiently exploit the architecture but complicates software debugging. Indeed, the software may be non-deterministic, *i.e.* perform differently at each execution. Consequently, there is no guarantee that an error will occur during the debugging activity. On the other hand, the number of elements to be analyzed during the debugging process is constantly increasing. As a result, it can be difficult to concentrate on the potentially faulty elements. Therefore, one of the most important challenges in the development process of MPSoC software is to reduce the debugging time.

In this thesis, we propose a new debugging methodology for MPSoC software. Our first objective is to be able to debug the same execution several times in order to analyze potential error sources. To do so, we have identified the sources of non-determinism in MPSoC software and proposed the most appropriate record-replay methods. Our second objective is to reduce the execution overhead of the record-replay mechanisms. To accomplish this objective, we have chosen to focus on a part of the non-deterministic behaviour and have selected efficient record-replay methods. The third objective is to provide a scalable solution, *i.e.* to be able to debug executions, characterized by an increasing number of elements. We have propose a partial replay method which allows to isolate and debug a fraction of the execution elements. This method applies to different types of MPSoC architectures and applications. We present the added value of our methodology by using different case studies using several software and hardware configurations.

Keywords : Debugging, deterministic record-replay, partial replay, MPSoC.