Support matériel pour la communication inter-processus dans un système multi-coeur

Maxime France-Pillois

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES

Arrêté ministériel : 25 mai 2016

Présentée par

Maxime FRANCE PILLOIS

Thèse dirigée par **Frédéric ROUSSEAU**, UGA

préparée au sein du **Laboratoire CEA/LETI**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

Support matériel pour la communication inter-processus dans un système multi-coeur.

Hardware support for inter-process communication in multiprocessor system.

Thèse soutenue publiquement le **27 septembre 2018**,
devant le jury composé de :

Monsieur FREDERIC ROUSSEAU
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse
Monsieur JEAN-PHILIPPE DIGUET
DIRECTEUR DE RECHERCHE, CNRS DELEGATION BRETAGNE PAYS DE LOIRE, Rapporteur
Monsieur GAEL THOMAS
PROFESSEUR, TELECOM SUDPARIS, Rapporteur
Monsieur TANGUY RISSET
PROFESSEUR, INSA LYON, Président du jury
Monsieur JEROME MARTIN
INGENIEUR DE RECHERCHE, CEA GRENOBLE, Co-directeur de thèse
Monsieur BENOIT DUPONT DE DINECHIN
INGENIEUR DE RECHERCHE, KALRAY S.A. - MONTBONNOTSAINT-MARTIN, Examinateur

Remerciements

Je souhaite en premier lieu remercier M. Jean-Philippe Diguet et M. Gaël Thomas d'avoir accepté la fonction de rapporteur à l'égard de mon travail et à ce titre, de m'avoir fait profiter de leurs remarques et commentaires précieux. Ces remerciements s'adressent également à M. Benoît Dupont De Dinechin pour la relecture attentive de cette thèse, me faisant bénéficier, ainsi, de son expertise. Je tiens aussi à remercier toutes les personnes qui ont accepté de se déplacer pour participer à ce jury de thèse ; et tout particulièrement M. Tanguy Risset qui en assuré la présidence.

Ma reconnaissance se destine tout particulièrement à mes deux encadrants qui m'ont accompagné tout au long de cette étude.

Frédéric Rousseau, mon directeur de thèse qui a toujours été disponible pour moi. Grâce à sa bonne compréhension du sujet, il a su m'orienter dans ma recherche, alors même que nous ne partagions pas, au quotidien, nos espaces de travail. Il m'a donné l'encadrement dont j'avais besoin dans les moments de flottement. Merci Frédéric pour tes relectures nombreuses et rapides aussi bien du manuscrit de thèse, que des articles scientifiques. Merci de m'avoir permis d'apprendre à structurer ma pensée à travers tes remarques pertinentes.

Toute ma gratitude à Jérôme Martin, mon encadrant au CEA, qui m'a apporté le conseil et l'expertise technique dont je manquais parfois. Sa perspicacité et sa rapidité d'appréhension des problèmes rencontrés m'ont étonné, à plus d'une reprise. Merci pour m'avoir incité à la persévérance dans les pistes de recherches initiées. Merci Jérôme pour avoir veillé à maintenir notre point hebdomadaire de suivi, malgré l'augmentation de tes responsabilités. Merci pour tes relectures et ton souci du détail qui m'ont permis de corriger aussi bien mon anglais que mon français.

Merci enfin, à vous deux pour la sagesse de votre encadrement : me laissant la liberté nécessaire pour gérer ma recherche selon mon entendement, tout en assurant le cadre requis pour la bonne progression de cette thèse.

Je remercie vivement le CEA Leti qui a financé mon travail pendant ces trois années, et qui m'a permis d'avoir accès à du matériel performant sans lequel ce travail n'aurait jamais pu prendre sa finalité actuelle.

Ma gratitude s'adresse aussi à mes deux laboratoires d'accueil.

Premièrement le LISAN au sein duquel j'ai passé l'essentiel de mon temps, merci pour votre accueil chaleureux.

Je voudrais spécialement remercier Éric Guthmuller et César Fuguet Tortolero qui m'ont apporté leur aide et leurs expertises techniques au moment où j'en avais besoin. Merci aussi à Baptiste Angelloz-Nicoud pour son aide précieuse concernant la mise en œuvre de l'émulateur.

Merci à Julie Dumas qui m'a montré la voie et m'a accompagné tout au long de ma

thèse. Merci pour tes conseils pratiques, pour tes astuces concernant l'enseignement, et pour nos discussions plus informelles. Ce fut toujours agréable de partager un petit moment à discuter ensemble.

Je tiens aussi à remercier celui qui a partagé mon bureau (pendant presque 3 ans!), François Dolique. Malgré quelques dissensions quant au partage de l'espace pour étendre nos jambes, tu as toujours été présent pour échanger quelques mots lorsque j'avais besoin de me changer les idées.

Mes remerciements envers l'équipe du LISAN ne seraient pas complets si je ne mentionnais pas Jean Durupt qui par ses lumières a permis indéniablement, d'élever le niveau de culture de notre box de travail.

Merci aussi à mon second laboratoire de rattachement, le TIMA, dont l'équipe s'est toujours montrée accueillante malgré ma présence plus sporadique. Je souhaite tout particulièrement remercier mon compagnon de thèse, Arief Wicaksana. Merci pour ta gentillesse et ton ouverture d'esprit.

Enfin, je voudrais remercier mon épouse, Minna, qui m'a soutenu et accompagné pendant toutes ces années : merci pour tous tes efforts en vue de rendre ma vie "hors laboratoire", des plus agréables.

Merci aussi à ma belle famille, et à leurs regroupements familiaux qui m'ont permis de réaliser qu'en fin de compte l'openspace du CEA n'était pas si bruyant que ça...

Un remerciement particulier à ma sœur, Léa, sans qui je ne serais pas la personne que je suis aujourd'hui.

Mes derniers remerciements, et non des moindres, sont pour mes parents. Merci à vous pour votre soutien indéfectible et pour avoir toujours cru en moi. Merci de m'avoir toujours encouragé et d'avoir été à mes côtés dans tous mes choix de vie. Merci du fond du cœur.

Enfin, merci au Seigneur des univers sans qui nul mouvement n'est possible. Si ma réflexion est juste c'est que Dieu m'a inspiré, si je me suis trompé cela vient de ma nature faillible.

Table des matières

Liste des figures

Liste des tableaux

Introduction

Des téléphones portables aux commandes de vol des avions, en passant par l'Internet des objets (IoT) ou encore les micro-serveurs, les systèmes embarquées font maintenant partie intégrante de notre quotidien. Tous ces systèmes ont pour point commun d'être constitués de ce que l'on nomme un système sur puce (*SoC, System on Chip* en anglais).

Les SoC sont le fruit de la miniaturisation de l'électronique. En effet, grâce à la réduction de la taille des transistors (loi de Moore), des systèmes complets (cœurs de calcul, mémoires, réseaux,...) ont pu être intégrés dans une même puce, annonçant par la même occasion l'essor des systèmes embarqués.

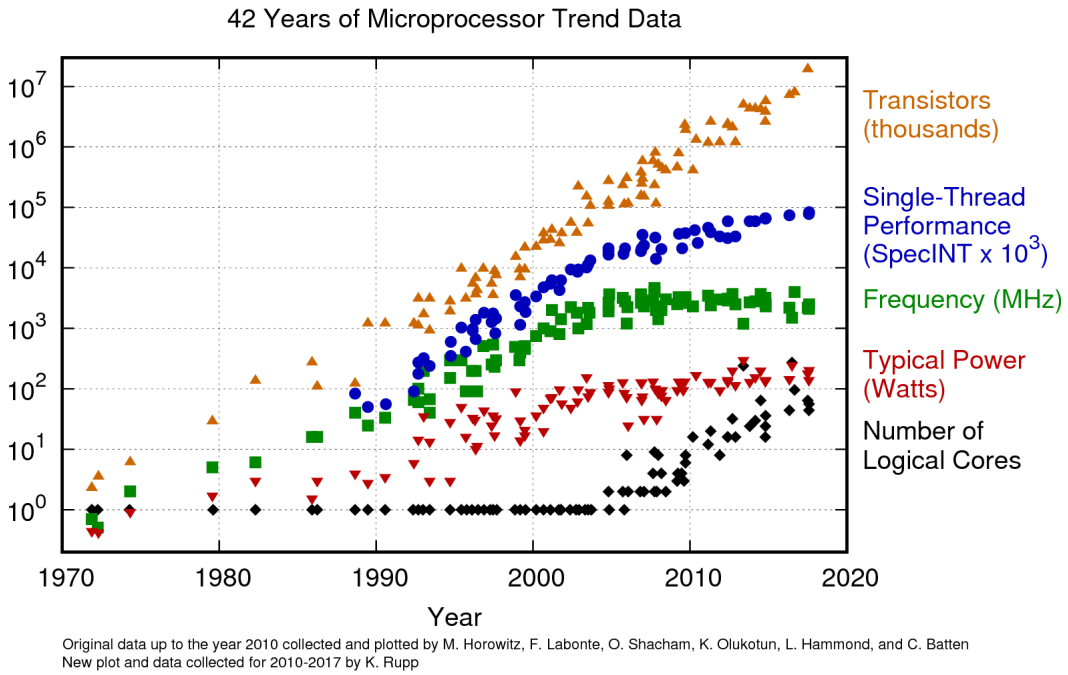

FIGURE 1 – Évolution de la fréquence et du nombre de cœurs des processeurs dans le temps[Rup]

Afin de satisfaire les besoins des utilisateurs, les systèmes embarqués doivent fournir des performances de calcul de plus en plus élevées. Comme représenté sur la figure 1, ce gain en performance était initialement obtenu par l'augmentation de la fréquence d'horloge des processeurs. Depuis une quinzaine d'années, cette technique a atteint sa limite en raison de restrictions technologiques, et de préoccupations relatives à la consommation d'énergie. En effet, plus la fréquence de l'horloge est élevée plus le système consomme de l'énergie.

La course aux performances a dès lors pris un virage, et la parallélisation des tâches s'est avérée être la stratégie adoptée pour fournir plus de puissance de calcul. Dorénavant, les systèmes embarqués intègrent plusieurs cœurs de calcul dans une seule et même

puce. Or, si les tâches (processus) d'une même application s'exécutent en parallèle sur différents cœurs, il y a tout lieu de croire que ces tâches doivent s'échanger des données afin d'accomplir leurs traitements.

Afin de procéder à cet échange de données, dans la majorité des cas, les tâches doivent se synchroniser entre elles. Nous pouvons alors prendre conscience que les mécanismes permettant ces synchronisations entre tâches sont des éléments déterminants pour les performances globales de ces systèmes composés de plusieurs unités de calcul.

En effet, en réduisant les temps nécessaires pour établir ces synchronisations, nous pourrons prétendre à nous rapprocher des performances idéales théoriques, à savoir qu'une application s'exécutant de manière séquentielle en x secondes, s'exécute en $\frac{x}{n}$ secondes sur un système parallèle composé de n cœurs sur lesquels s'exécute la dite application. Sachant qu'en réalité, ce temps idéal ne dépend pas uniquement des synchronisation mais de divers facteurs : parallélisation équilibrée des tâches, communication entre tâches, ...

Pour les systèmes disposant d'une mémoire partagée entre tous les cœurs de calcul, les échanges d'informations entre les processus d'un même système sont réalisés au moyen de variables partagées. Cependant, afin de garantir l'intégrité des données, les accès à ces variables partagées doivent être réalisés avec précautions. Aussi, les concepteurs logiciels sont amenés à définir des sections critiques au sein desquels les variables partagées sont accédées. Des services de synchronisation permettent alors de garantir qu'à un instant donné, un seul et unique processus puisse modifier les variables partagées accédées dans cette section critique.

En ce qui concerne la manière de garantir les accès aux sections critiques, deux grandes stratégies d'implémentation peuvent être mises en place.

La première est dite pessimiste car les exécutions des section critiques sont sérialisées de sorte qu'un seul processus puisse réaliser sa section critique à un instant donnés. Les autres processus, quant à eux, restent bloqués sur un verrou d'entrée tant que l'accès aux données partagées n'a pas été autorisé explicitement par le processus ayant exécuté sa section critique. Ces synchronisations sont dites pessimistes du fait que les accès à la section critique sont sérialisés dans tous les cas, qu'il y ait ou non collision d'accès mémoires (accès simultanés aux mêmes variables mémoires).

Il existe principalement deux manières de mettre en place ce type de synchronisation :

— L'utilisation d'instructions matérielles spécifiques, appelées instructions atomiques, permettant de lire et d'écrire une variable mémoire en une seule instruction. Ce type d'instruction garantit qu'aucune modification de la variable n'a pu être réalisée par un autre processus entre la lecture et l'écriture réalisées par l'instruction atomique. L'implémentation d'un verrou de protection d'une section critique peut alors être mis en place comme décrit dans le chapitre 2.

— La délégation de l'exécution de la section critique à un serveur. En simplifiant, cette méthode appelée délégation, ou *combining* en anglais, consiste à ce qu'un processus fasse exécuter sa section critique par un autre processus, appelé « serveur », en charge d'exécuter toutes les sections critiques accédant à un ensemble défini de variables mémoires partagées. Comme exposé par Lozi *et al.* [LDT+16], l'intérêt de cette méthode est double : elle permet d'éviter l'exécution d'instructions atomiques dans le chemin critique, ainsi que d'adresser le problème de la localité spatiale (mémoires caches) des données partagées accédées dans une section critique. En effet,

une section critique accédant à des variables partagées doit ramener, par un défaut de cache (*cache miss*), les données partagées dans sa mémoire cache locale, et cela à chaque fois qu'un processus s'exécutant sur un cœur différent obtient l'autorisation d'exécuter sa section critique. En regroupant les exécutions des sections critiques de différents processus sur un serveur s'exécutant sur un seul cœur, le besoin de rapatriement de données partagées dans la mémoire cache locale est réduit.

Le seconde stratégie de sécurisation des accès aux variables partagées est l'approche optimiste. Elle consiste en l'exécution a priori des sections critiques contenant les accès aux variables partagées. Un mécanisme matériel de surveillance permet de vérifier les accès concurrents aux variables partagées. Si aucun accès concurrent n'est à déplorer, alors l'exécution du logiciel suit son cours normal. Par contre, si un ou des accès concurrents ont eu lieu, les résultats d'exécution d'une section critique sont gardés, alors que la ou les autres sections critiques concurrentes seront rejouées. L'implémentation de cette stratégie repose en règle générale sur le principe de mémoire transactionnelle.

L'approche optimiste se révèle intéressante en terme de performance pour des systèmes au sein desquels le nombre de collisions est faible. Cependant, lorsque le nombre de cœurs de calcul augmente, le nombre de collisions augmente aussi. Dans ce cas, le coût (temps et consommation) nécessaire à la ré-exécution des sections critiques devient important, ce qui rend cette méthode moins performante pour les systèmes avec un nombre de cœurs élevé comparativement aux approches pessimistes [RSS+18]. De plus, en pratique, il s'avère que l'utilisation des mémoires transactionnelles implique un surcoût non-négligeable aux accès mémoires. Par ailleurs, cette approche nécessite une modification des habitudes de programmation, du fait que le dimensionnement des sections critiques doit dès lors prendre en compte le risque de ré-exécution fréquentes de celles-ci en cas de mauvais dimensionnement. Ces raisons expliquent le fait que les systèmes embarqués actuels utilisent plus souvent des stratégies pessimistes au détriment de l'approche optimiste.

Un des objectifs de ce travail de thèse est d'améliorer les performances temporelles des mécanismes de synchronisation pour les systèmes embarqués composés d'un nombre élevé de cœurs (plusieurs dizaines, voire plusieurs centaines). Nous avons donc choisi de focaliser notre étude sur les approches pessimistes. Le passage à l'échelle étant une motivation importante de notre travail, il nous a semblé que la méthode de délégation et sa centralisation des exécutions des sections critiques sur un serveur se prêtait plus difficilement à cette exigence. Aussi, nous avons choisi de nous concentrer sur les méthodes pessimistes à base de verrous, couramment utilisées dans les systèmes embarqués actuels.

Plan de thèse

Le présent manuscrit vise à présenter le travail de thèse effectué au sein des laboratoires CEA Leti et Tima de Grenoble, dont l'objectif est l'optimisation des mécanismes de synchronisation pour les systèmes many-cœurs. Ce document est organisé de la manière suivante.

Le chapitre 1 définit les notions de systèmes embarquées multi et many-cœurs, ainsi que leurs spécificités. Ce chapitre sera aussi l'occasion de définir en détails le type de système visé par ce travail de recherche.

Le chapitre 2 revient sur les motivations, ainsi que sur les problématiques que nous

cherchons à résoudre.

Le chapitre 3 expose l'état de l'art des les travaux de recherches et les propositions d'optimisation concernant les mécanismes de synchronisation.

Le chapitre 4 présente la chaine de mesures précise et non-intrusive que nous avons conçue afin d'étudier les mécanismes de synchronisation sur les systèmes embarqués multi-coeurs.

Le chapitre 5 propose une approche pratique pour l'optimisation logicielle des mécanismes de synchronisation. Ce chapitre constitue aussi une opportunité de revenir plus en détails sur notre méthodologie d'optimisation exploitant la chaine de mesures conçue.

Dans le chapitre 6, nous proposons une solution de verrou mobile optimisé pour le passage à l'échelle.

Enfin, nous conclurons cette thèse en exposant les solutions proposées aux problématiques soulevées dans le chapitre 2, ainsi qu'en proposant quelques perspectives de poursuite des travaux.

Chapitre 1

Les sytèmes multi et many-coeurs

Sommaire

1.1 Architecture des MPSoC

Concernant les systèmes embarqués, nous entendons parfois parler de *multi-cœur*, parfois de *many-cœur* ou alors même de *MPSoC* mais que renferment réellement ces termes ?

En général, nous parlons de systèmes *multi-cœurs* lorsque ces derniers intègrent moins de dix cœurs de calcul. Au dessus de cette limite, nous parlons plutôt de *many-cœurs*.

Les systèmes *many-cœurs* intègrent de plus en plus de cœurs, plusieurs dizaines, voir plusieurs centaines à l'exemple des processeurs MPPA de Karlay [kal18] composés de plus de 200 unités de traitement.

L'acronyme MPSoC pour *Multi-Processor System on Chip* est, quant à lui, utilisé pour désigner, de manière indifférenciée, des systèmes embarqués intégrant plusieurs processeurs (cœurs de calcul) dans un même composant. En raison de sa portée générale, nous avons choisi d'utiliser cette appellation dans la suite de ce manuscrit.

Dans cette section, nous allons revenir sur les caractéristiques principales de ce type de systèmes afin de définir clairement le type d'architecture ciblée par notre étude.

1.1.1 Les réseaux sur puces

Lorsque les systèmes sur puce étaient composés d'une seule unité de traitement (mono-cœur) ou de quelques unités (multi-cœur), les communications entre les dif-

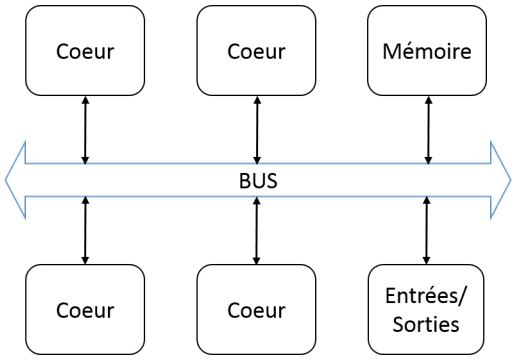

FIGURE 1.1 – Système sur puce communicant via un bus

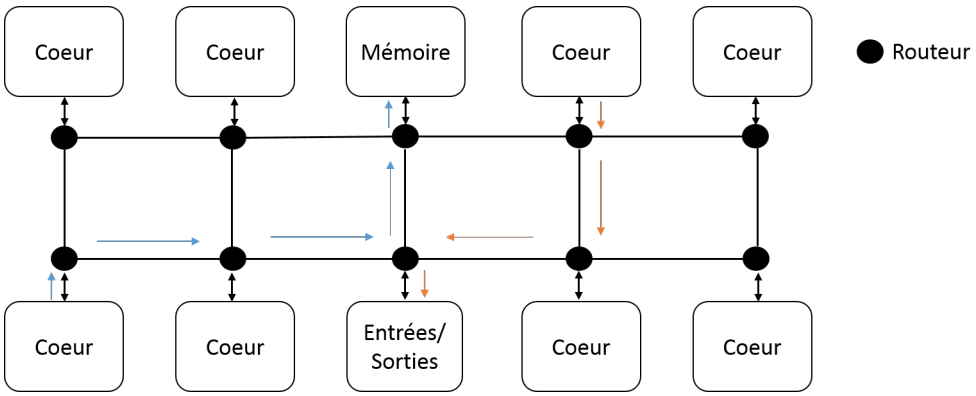

FIGURE 1.2 – Système sur puce communicant via un NoC

férents éléments du systèmes (unité(s) de traitement, mémoire(s), entrées/sorties,...) étaient réalisées au moyen d'un ou de plusieurs bus de communication, comme cela est représenté sur la figure 1.1.

Néanmoins, avec l'apparition de systèmes intégrant de plus en plus d'unités de calcul, les bus de communication ne peuvent plus fournir la bande passante nécessaire. De nouveaux *media* de communication doivent être mis en œuvre afin de supporter, et d'exploiter le parallélisme des *MPSoC*. Les réseaux sur puces (**N**etworks **o**n **C**hips, **NoC**) sont les *media* de communication répondant le mieux à ces nouvelles exigences. En effet, les réseaux sur puces permettent le transfert simultané de plusieurs messages à travers les routeurs composant le réseau. Ainsi, plusieurs communications peuvent s'établir en parallèle, ce qui permet de préserver la bande passante de l'infrastructure de communication lorsque le nombre d'éléments du système augmente. La figure 1.2 représente un système communicant via un réseau sur puce, ainsi que l'établissement des communications en parallèle.

Les réseaux sur puces sont des éléments clés des *MPSoC* nécessaire au passage à l'échelle.

1.1.2 Paradigme mémoire

Les *MPSoC* se divisent en deux grandes catégories en fonction du paradigme mémoire qu'ils implémentent, à savoir : systèmes à mémoire partagée ou systèmes à mémoire privée.

Mémoire partagée Il s'agit des systèmes au sein desquels un même segment de mémoire peut être accédé par un ensemble d'éléments du système (cœurs, périphériques,

accélérateurs matériels, ...). Lorsque nous parlons de mémoire partagée, nous faisons référence en réalité à une mémoire partagée cohérente. Cela signifie qu'une couche de service associée à ce segment de mémoire partagée permet notamment :

— de garantir que l'ensemble des éléments aient la même vue de la mémoire (cohérence mémoire),

— d'assurer un déterminisme lors de l'accès à la mémoire.

Mémoire privée À l'opposé de ces systèmes à mémoire partagée, nous avons les systèmes ne partageant aucun segment mémoire entre les éléments du système. Ces systèmes sont dits « à mémoire privée ».

Notre étude se focalise sur les systèmes à mémoire partagée. Bien que le passage à l'échelle des *MPSoC* à mémoire partagée cohérente est un challenge de taille, nous pensons que la fin des systèmes à mémoire partagée n'est pas imminente. En effet, la programmation efficace des systèmes à mémoire privée renferme une complexité importante quant à la conception logicielle : explicitation systématique des échange de données, systèmes d'exploitation différents, ... Aussi, la maitrise des paradigmes de programmation pour des systèmes à mémoire privée demande un tel niveau d'expertise qu'il n'est pas possible, à l'heure actuelle, d'imposer ce type de système comme solution unique pour les systèmes embarqués hautes performances. Ainsi, nous pensons que les architectures à mémoire partagée ont encore de beaux jours devant elles, et justifie pleinement l'effort de recherche afin d'en améliorer les performances.

1.1.3 Communications entre cœurs

Au sein des *MPSoC*, les communications entre cœurs peuvent être réalisées de deux manières :

— Les échanges de données sont réalisés en exploitant la mémoire partagée. En effet, des variables mémoires partagées (ou structures de données) permettent aux processus s'exécutant sur différents cœurs de calculs de s'échanger des informations. Dans ce cas, des mécanismes de synchronisation doivent être mis en œuvre afin d'éviter les écritures simultanées, de garantir l'ordre entre lectures et écritures, ou encore de synchroniser différentes tâches au moyen d'un rendez-vous.

— Les échanges de données sont effectués par l'établissement de communications explicites : envoi et réception de messages explicites.

Le premier type d'échanges repose sur la mémoire partagée ce qui induit plus de facilité de programmation. Cependant, il ne peut être élaboré que pour des systèmes dits à mémoire partagée.

Le second type d'échange est quant à lui plus général, et peut être mis en œuvre quelque soit le paradigme mémoire du système : mémoire privée ou mémoire partagée. Cependant, sa programmation impose plus de contraintes au programmeur qui se voit obligé d'expliciter tous les échanges de données.

En règle générale, lorsqu'une mémoire partagée est disponible, elle sera préférée pour effectuer les échanges de données pour des raisons d'efficacité.

1.1.4 Architecture en grappe

Les *MPSoC* actuels, et plus particulièrement les systèmes composés de nombreux cœurs de calcul, sont souvent organisés en grappes (*clusters*). Une grappe est un groupement d'éléments : CPU, DMA, IO, IRQ manager, ... autour d'un bloc mémoire. Ce groupe d'éléments ayant accès à un segment de mémoire partagée peut aussi être appelé nœud (*node*) ou tuile (*tile*).

Dans la majorité des cas, les éléments d'une grappe sont inter-connectés par un réseau dit « crossbar » reliant chaque élément en point à point. Les connexions entre grappes sont réalisées grâce aux réseaux sur puce.

1.2 Quelques architectures de MPSoC commercialisés

Dans cette section nous nous proposons de décrire brièvement deux architectures de MPSoC disponibles dans l'industrie, avant de présenter notre architecture dans la section suivante, mettant ainsi en exergue les similarités et différences entre ces architectures.

1.2.1 Le MPPA de Kalray

Le MPPA de kalray [kal18] est une référence dans l'univers des systèmes many-cœurs. En effet, son architecture composée de 256 cœurs de calcul fait actuellement de lui un des processeurs avec le plus grand nombre de cœurs sur le marché.

Le figure 1.3 illustre l'architecture de ce processeur. Il s'agit d'une architecture organisée sous forme de grappes ('C' sur la figure). Les 256 cœurs de l'architecture sont regroupés en 16 grappes (*clusters*) de 16 cœurs. Chaque cœur est formé d'un CPU 32 bits qui dispose de sa mémoire cache L1 privée (instructions et données). De plus, chaque grappe inclut une mémoire partagée entre les éléments de la grappe. Une grappe intègre aussi divers autres éléments tels qu'un contrôleur DMA (*Direct Memory Access*) par exemple.

Les grappes sont inter-connectées grâce à un réseau sur puce de topologie TORE 2D.

1.2.2 Le TILE-Gx72 de Tilera

Parmi les processeurs many-cœurs de référence, nous avons aussi le TILE-Gx72 de Tilera [Til12]. La figure 1.4 expose un schéma de l'architecture de ce processeur.

Le TILE-Gx72 est composé de 72 cœurs 64 bits inter-connectés par un réseau sur puce de type *mesh* 2D. Chaque cœur dispose de ses propres mémoires caches L1 et L2. Une mémoire cache cohérente de troisième niveau L3 est partagée par tous les cœurs du système.

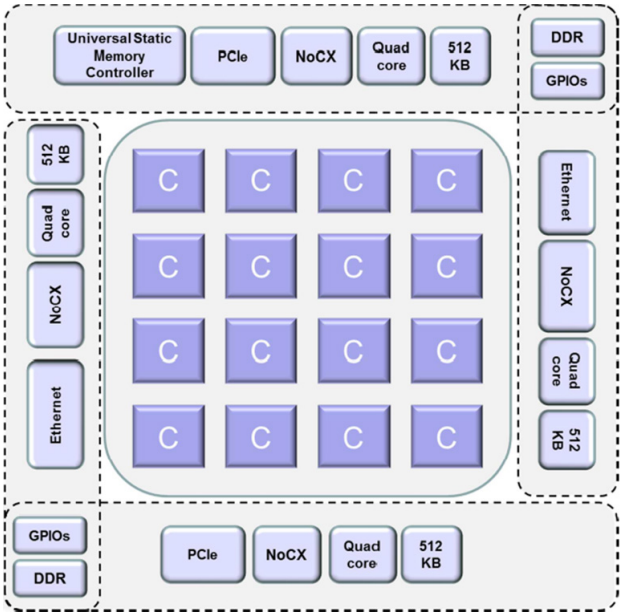

FIGURE 1.3 – Architecture *MPPA*[kal18]

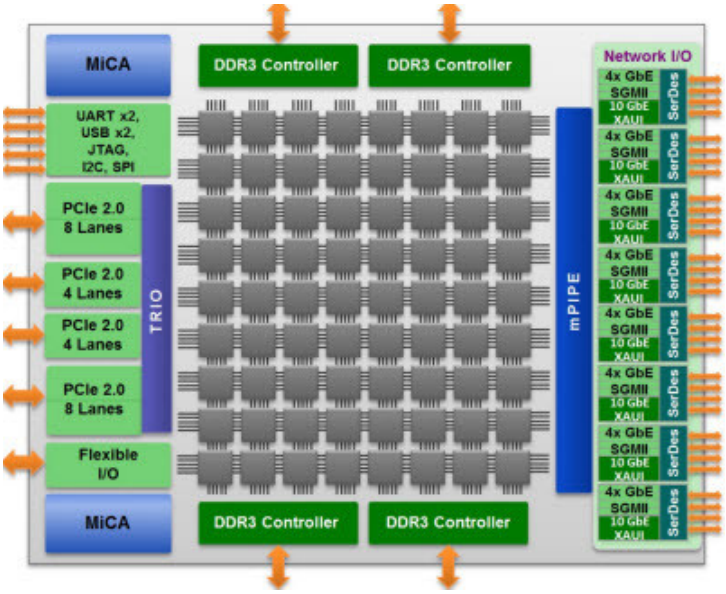

FIGURE 1.4 – Architecture *TILE-Gx72*[Til12]

1.3 *TSAR* : notre plateforme de travail

1.3.1 L'architecture *TSAR*

Dans le cadre de notre étude, nous avons décidé de travailler sur l'architecture *TSAR* (*T*era-*S*cale *AR*chitecture) [TSA18] développée par le **L**aboratoire d'**I**nformatique de **P**aris 6 (**LIP6**). Nous avons choisi ce système en raison de sa bonne représentativité des MPSoC actuels. En effet, *TSAR* est un système many-cœur organisé sous forme de grappes autour d'un réseau sur puce pouvant intégrer une centaine de cœurs de calcul.

La figure 1.5 représente une vue d'ensemble de l'architecture *TSAR*.

Une architecture en grappe

L'architecture *TSAR* est organisée sous forme de grappes comme illustré sur la figure 1.5. Chaque grappe ou *cluster* est principalement composée de :

— quatre cœurs MIPS32 : chaque cœur dispose d'une mémoire cache L1

— une mémoire cache L2

— un gestionnaire d'interruption (**I**nterrupt **C**ontroller **U**nit, **ICU**)

— un contrôleur de réseau (**N**etwork **I**nterface **C**ontroller, **NIC**)

La mémoire est cohérente et partagée par l'ensemble du système. Chaque mémoire L2 cache une plage d'adresses qui lui est propre. Les données cachées dans le L2 d'une grappe peuvent être accédées par n'importe quel cœur du système. Il s'agit donc d'une mémoire partagée distribuée, avec un temps d'accès à celle-ci non-uniforme (**N**on **U**niform **M**emory **A**ccess, **NUMA**).

Nous remarquons alors que l'architecture sur laquelle nous avons mené notre étude se trouve au croisement des chemins entre le MPPA et le TILE-Gx72.

En effet, à l'instar du MPPA notre architecture est regroupée en grappes réunissant différents composants : CPU, mémoires,... Ce type de regroupement autorise l'établissement d'autres types de communications entre éléments, ainsi que le partage de certaines ressources (mémoires,...) permettant la mise en place de coopérations optimisées entre les éléments d'une grappe.

Cependant, tout comme le TILE-Gx72, notre architecture dispose d'une mémoire cohérente partagée par l'ensemble des éléments du système. Grâce à cette mémoire partagée, nous pouvons prétendre à exécuter un système d'exploitation unique sur l'ensemble du système (tel que Linux), et par la même occasion viser des applications standards (*general purpose*).

Le réseau sur puce

Le réseau sur puce implémenté dans l'architecture *TSAR* est un réseau *DSPIN*[1]. Celui-ci implémente une stratégie de routage dite *cut-through*. Cette stratégie implique le fait que les mots constituant chaque paquet (**FLITS** ou ***FL**ow control dig**ITs***) ne sont pas mis en mémoire tampon (bufferisés) dans les routeurs, mais transmis directement au routeur suivant.

L'algorithme de routage implémenté dans *TSAR* est un algorithme de type XY, ce qui signifie que la trame effectue d'abord tous ses déplacements sur l'axe des X (abscisses)

1. Réseau propriétaire de topologie *mesh* 2D développé par le LIP6

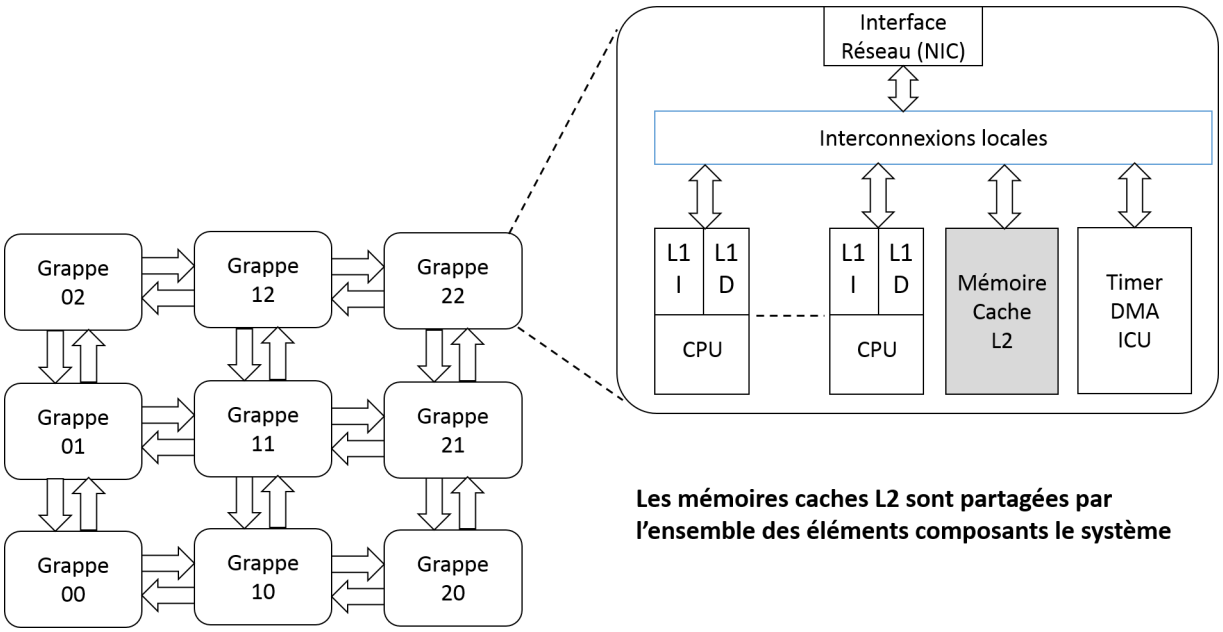

FIGURE 1.5 – Architecture *TSAR*

avant de réaliser ceux en Y (ordonnées), comme cela est représenté sur le figure 1.6. Cette stratégie de routage a l'avantage d'être simple, et de garantir qu'il n'y ait pas d'interblocage entre paquets. Elle est donc bien appropriée pour des systèmes généralistes (*general purpose*) dont le trafic réseau n'est pas connu a priori, comme cela est le cas pour *TSAR*.

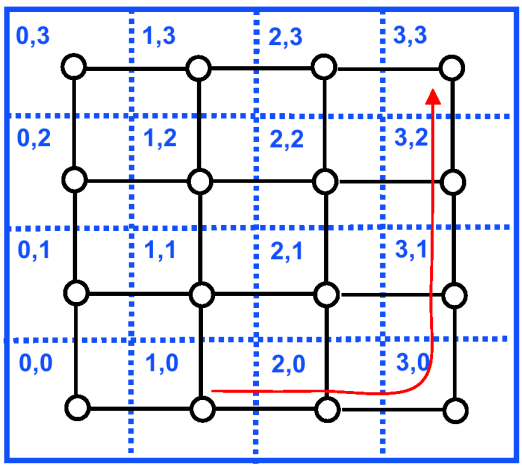

FIGURE 1.6 – Illustration d'un routage XY d'une trame sur un NoC en provenance du nœud (1,0) et à destination du nœud (3,3)

1.3.2 Plateforme d'expérimentation

Dans le cadre de ce travail de thèse, nous avons eu la chance de pouvoir utiliser un émulateur Veloce 2 Quattro [Men18] afin de mettre en place notre plateforme d'expérimentation.

De manière simplifiée, l'émulation permet de simuler à très grande vitesse le modèle RTL (*Register Transfer Level*) d'un système avec une précision au cycle près. Ainsi, nous avons pu émuler un modèle RTL du système *TSAR* afin de mener à bien notre étude.

En ce qui concerne le logiciel, nous avons utilisé la version 4.6 du noyau Linux portée sur notre architecture par le laboratoire d'informatique de Paris 6 (LIP6) et le CEA Leti. De même, nous avons pu bénéficier d'un portage de la µClibc. Cela nous a permis de mettre en place un environnement d'évaluation réaliste mettant en œuvre un système d'exploitation complexe (Linux) ainsi que des applications utilisateurs standards (µClibc).

La version de GCC utilisée pour compiler les applications est la version 4.8.2.

Chapitre 2

Problématique

Sommaire

2.1 La synchronisation inter-processus

En introduction, nous avons présenté le besoin grandissant de mécanismes de synchronisation performants accompagnant l'apparition de MPSoC intégrant de plus en plus de cœurs de calcul.

Dans cette section, nous tenterons de donner une vision d'ensemble sur les synchronisations inter-processus et les mécanismes associés. Avant toutes choses, arrêtons nous sur la signification de l'expression « synchronisation inter-processus ».

Commençons par définir ce que nous désignons par « processus ». En général, dans le domaine informatique, le mot processus est utilisé pour désigner une ensemble d'instructions (*flot* d'exécution) et son espace d'adressage. Deux processus ne partagent pas le même espace d'adressage. Le terme processus s'oppose au *thread* (ou processus léger) qui ne dispose pas d'un espace d'adressage propre. En effet, deux *threads* s'exécutent dans le même espace d'adressage. Par l'expression « synchronisation inter-processus », nous désignons les synchronisations entre processus, mais aussi entre *threads*. Nous englobons donc ces deux types d'éléments dans la notion de processus telle que nous l'utilisons dans ce manuscrit.

2.1.1 Les synchronisations, quel besoin ?

La synchronisation inter-processus est un procédé crucial dans les systèmes parallèles. En effet, les échanges de données entre processus nécessitent que ces derniers puissent se synchroniser afin de garantir l'intégrité des données.

Prenons l'exemple de transactions financières sur un compte bancaire. La figure 2.1 illustre les transactions effectuées sur un compte bancaire lorsqu'aucun mécanisme de synchronisation n'est utilisé.

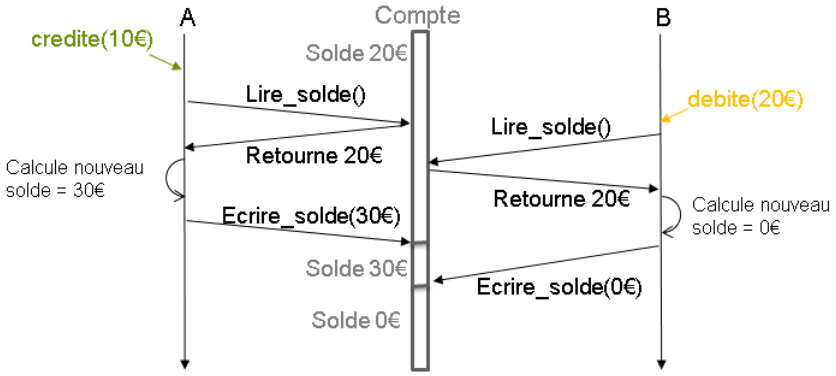

FIGURE 2.1 – Séquence d'opérations concurrentes sans synchronisation sur un compte bancaire

Le schéma représente deux processus "A" et "B" effectuant des opérations simultanées sur le même compte bancaire "Compte". Le solde initial du compte est de 20€. Un ordre de créditer le compte de 10€ est émis par un utilisateur au processus "A". Un ordre de débit de 20€ est émis par un utilisateur au processus "B". Le processus "A" lit le solde du compte puis le processus "B" lit à son tour le solde du compte avant que le processus "A" ait modifié la valeur du solde. Ensuite le processus "A" calcule le nouveau solde du compte suite au crédit de 10€. Une fois le calcul terminé, le processus "A" va écrire le résultat (soit 30€) comme nouveau solde du compte. En parallèle, le processus "B" va lui aussi calculer le nouveau solde du compte débité de 20€ à partir de la valeur qu'il avait lu avant que "A" n'ait effectué son calcul. Le nouveau solde du compte calculé par "B" (soit 0€) est alors écrit en tant que solde du compte bancaire.

Nous observons alors que par manque de synchronisation entre les opérations réalisées par les 2 processus sur la même variable (ici le solde du compte bancaire), nous obtenons un résultat final erroné, car le solde restant sur le compte à la suite des opérations doit être de 10€ et non pas de 0€. Nous remarquons donc que l'utilisation de mécanismes de synchronisation est nécessaire pour garantir l'intégrité du solde du compte bancaire lors de la réalisation de plusieurs transactions simultanées.

La figure 2.2 représente les mêmes transactions bancaires que le schéma 2.1 réalisées par deux processus "A" et "B", mais cette fois, celles-ci sont protégées par un mécanisme de synchronisation. Le mécanisme de synchronisation utilisé permet l'exclusion mutuelle lors de l'accès à la valeur du compte bancaire. Il s'agit d'un verrou garantissant qu'un seul processus a le droit d'accès à la valeur du compte en banque à un instant donné. Lorsque le verrou est à 0, le processus reçoit l'autorisation d'accès à la variable protégée (ici le solde du compte bancaire) et place la valeur du verrou à 1 pour signifier que l'accès est déjà pris. Lorsque le verrou est à 1, le processus sait alors que l'accès est déjà pris, et attend la libération du verrou (état du verrou passé à la valeur 0) avant d'accéder à la variable protégée.

Sur le schéma 2.2 nous pouvons voir que le processus “A" demande l'accès exclusif au compte bancaire en interrogeant la valeur du verrou avant de réaliser son opération de crédit sur le compte bancaire. Obtenant l'accès, le processus “A" effectue directement son opération.
Le processus “B" effectue lui aussi une demande d'accès exclusif au compte bancaire. L'accès étant déjà pris par le processus “A", le processus “B" attend la libération de l'accès, et effectue son opération de débit qu'une fois le verrou relâché par “A".

Nous remarquons que le solde final du compte bancaire est bien de 10€ comme attendu. Le mécanisme de synchronisation a donc permis de garantir l'exactitude de la valeur du compte bancaire lors de la réalisation de plusieurs transactions simultanées.

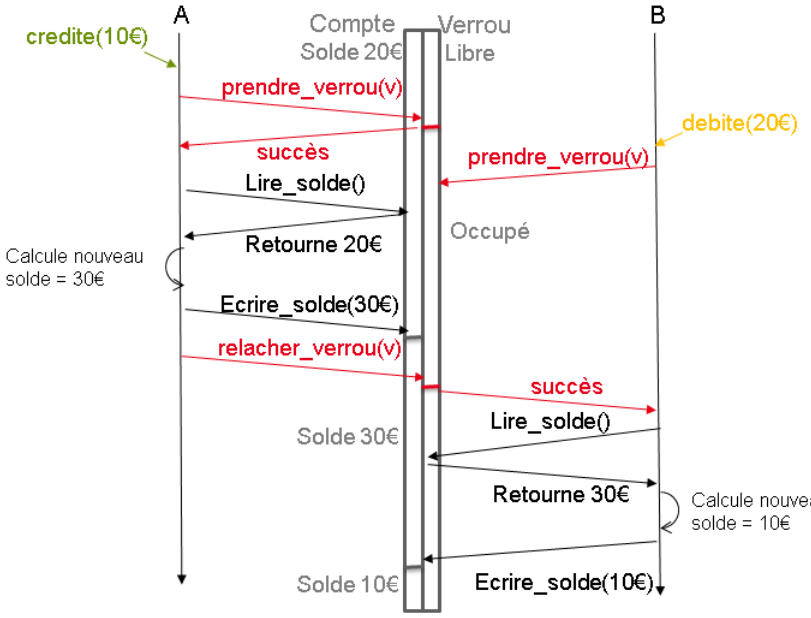

FIGURE 2.2 – Séquence d'opérations concurrentes sur un compte bancaire avec verrou

2.1.2 Mécanismes de synchronisation

Les besoins des utilisateurs (programmeurs) en matière de synchronisation sont multiples. Aussi, de nombreux mécanismes ont été développés afin de répondre à leurs attentes. Nous pouvons citer pour les principaux :

- l'exclusion mutuelle ou *mutex* (***mut****ual* ***ex****clusion*) ;
- le sémaphore ;
- le paradigme producteur/consommateur ;
- le barrière de synchronisation (rendez-vous).

Il s'agit en réalité plus de services fournis aux programmeurs que de mécanismes foncièrement différents, car les primitives sur lesquels ils reposent ont tendances à être similaire. Ainsi, un service peut être implémenté par un autre, et vice versa.

Cela étant dit, une primitive adaptée au service ciblé permet une implémentation plus efficace. Nous avons alors trouvé deux types de primitives revêtant un attrait particulier :

- le décompte du nombre d'appel à l'élément de synchronisation : utile pour l'implémentation des barrières de synchronisation par exemple.
- le verrouillage de l'élément de synchronisation (*lock/unlock*) : à la base de l'exclusion mutuelle

Ces deux primitives font intervenir des spécificités différentes, et nécessite une étude détaillée de chacune d'entre elles. Si ces deux primitives venaient à être optimisées, l'ensemble des autres mécanismes pourraient alors en profiter. Notre travail consiste donc en l'étude de ces deux primitives essentielles aux mécanismes de synchronisation.

Dans les systèmes à mémoire partagée, les synchronisations reposent sur le principe d'instructions atomiques. Il s'agit d'instructions processeur permettant de lire et de modifier une variable mémoire en une seule et unique instruction. Le *test&set* ou encore le *fetch&add* sont des instructions atomiques. Elles permettent : de lire et passer une variable mémoire à 1 pour le *test&set*; de lire et d'incrémenter une variable en une seule instruction pour le *fetch&add* .

Initialement, les deux primitives de base, à savoir le verrou et la barrière, étaient implémentées à partir de ces instructions dans de nombreux systèmes comme présenté par les algorithmes 1 et 2.

Algorithme 1 verrou *test&set*

```
// prise du verrou
while test&set(x) ≠ 0 do
    nop()
end while
// section critique
...
// libération du verrou
x = 0
```

Algorithme 2 barrière *fetch&add*

```
Require: Les variables nb_processes et wake_up sont partagées par l'ensemble des
  processus (i.e variables globales)
  wake_up = 0
  if fetch&add(nb_processes) == (EXPECTED_NB_PROCESSES − 1) then
    nb_processes = 0
    wake_up = 1
  end if
  while wake_up == 0 do
    nop()
  end while
```

L'algorithme 1 présente une fonction basique de gestion d'un verrou utilisant l'instruction atomique *test&set*.

Dans cet algorithme, la variable x représente l'état du verrou : 0 le verrou est libre, 1 le verrou est occupé. La prise du verrou est alors réalisée par une boucle d'attente sur le valeur de x. L'instruction *test&set* lit la valeur de x et la passe à 1 en une seule instruction (atomiquement) afin de garantir qu'aucun autre processus n'a pu lire la valeur de x dans l'intervalle de temps. Lorsque la valeur retournée par l'instruction *test&set* est égale à 0 cela signifie que le verrou était libre. La valeur étant immédiatement positionnée à 1 par l'instruction atomique, le processus peut accéder à sa section critique de manière exclusive. Si la valeur retournée par l'instruction *test&set* est égale à 1, le verrou n'est alors pas disponible, et le processus continue d'itérer dans la boucle d'attente jusqu'à la libération du verrou.

L'algorithme 2 est un exemple de barrière exploitant l'instruction atomique *fetch&add*. Dans cet algorithme, les variables *nb_processes* et *wake_up* sont partagées par tous les processus participant à la barrière. *nb_processes* représente le nombre de processus ayant atteint la barrière. *wake_up* permet de signaler aux processus en attente qu'ils peuvent continuer leur exécution. Lorsque cet indicateur est passé à 1, cela signifie que les processus peuvent dépasser la barrière. Si cet indicateur est à 0, les processus doivent attendre. *EXPECTED_NB_PROCESSES* est une constante contenant le nombre de processus total participant à la barrière. Le compteur du nombre de processus en attente est incrémenté de manière atomique grâce à l'instruction *fetch&add*. Cette instruction permet de lire la valeur de la variable passée en paramètre, puis de l'incrémenter de 1 de manière atomique (sans qu'aucun autre processus ne puisse lire ou écrire la valeur de la variable entre temps). Si l'ensemble des processus attendus n'est pas encore arrivé, un processus attend dans la boucle "while" l'arrivée de la totalité des processus avant de continuer son exécution. Une fois que l'ensemble des processus attendus a atteint la barrière, le compteur de processus est remis à 0, en prévision de le prochaine barrière, et l'indicateur *wake_up* est passé à 1 afin de relâcher l'ensemble des processus. Lorsque l'indicateur *wake_up* est passé à 1, les processus en attente quittent la boucle d'attente et reprennent leur exécution.

Il s'agit là des algorithmes de base, non optimisés, des mécanismes de *mutex* et *barrière de synchronisation*. Ces algorithmes ne présentant pas de bonnes performances ont, par la suite, été optimisés.

2.1.3 Mémoire cache et cohérence

En raison de la latence d'accès aux mémoires, tous les MPSoC à mémoire partagée implémentent des niveaux de cache. Le système de mémoires caches permet de rapprocher les données et les instructions des éléments de calcul afin de réduire le temps d'accès à celles-ci. Ainsi, le cœur de calcul perd moins de temps à attendre les données (ou instructions), et la vitesse de calcul globale du système est augmentée.

Cependant, la hiérarchie des mémoires caches implique de garantir la cohérence des valeurs correspondant à la même adresse mémoire dans les différents caches du système, et cela afin de garantir la validité des données avec lesquelles travaillent les CPUs. Le maintient de cette cohérence de cache requiert des exigences particulières et entraine certains problèmes, comme expliqué par Hennessy et Patterson dans [HP11].

Au cours des dernières années, des efforts importants ont été déployés afin d'améliorer les protocoles de cohérence de cache, et de permettre leur passage à l'échelle. Cependant, comme cela a été relevé par Baumann et *al.* [BBD+09], le nombre de messages échangés afin de permettre la cohérence des caches reste encore très élevé.

Par exemple, l'implémentation basique des systèmes de synchronisation à partir des instructions atomiques tel que *test&set* entraîne l'échange d'un nombre important de messages de cohérence de cache. Le fait que ces instructions atomiques modifient la variable ciblée quelque soit son état, nécessite l'exclusivité sur la ligne de cache. Il s'agira dans notre cas de la ligne cachant la variable du verrou.
Typiquement, le *test&set* écrit la valeur 1 dans la variable cible, et cela même si cette dernière était déjà à 1. L'obtention de l'exclusivité, en vue d'une écriture, sur une ligne de cache implique l'invalidation des autres mémoires caches possédant cette ligne. Cela im-

plique donc des échanges de messages entre les mémoires caches afin de gérer ces invalidations. Une implémentation simple des synchronisations se contente de boucler sur les variables de synchronisation en effectuant une opération atomique à chaque tour de boucle et cela jusqu'à l'obtention du droit d'accès (cf algorithme 1). Ce type d'implémentation entraîne donc une quantité de messages entre caches très importante si le nombre d'itérations est élevé.

Pour cette raison, les mécanismes de synchronisation cherchent dorénavant à utiliser les mémoires caches de manière intelligente afin de limiter les surcoûts des protocoles de cohérence de cache. Ainsi, nous avons vu l'apparition de nouvelles instructions atomiques telles que : *Test-Test-and-set* ou encore *Load Linked / Store Conditionnal (ll/sc)*.

L'idée de base de ces instructions est de séparer de manière sûre les phases de lecture et d'écriture de la variable afin que l'exclusivité sur la ligne de cache ne soit demandée qu'en cas de besoin, c'est à dire uniquement lors d'un changement effectif de la valeur de la variable ciblée. La complexité de ces instructions vient du fait qu'elles doivent garantir que la valeur de la variable n'ait pas pu changer entre les phases de lecture et d'écriture.

L'instruction *Test-Test-and-set* rajoute un test avant que l'instruction *test&set* classique ne soit exécutée. Ainsi, la variable cible (dans notre cas le verrou) ne sera écrite à 1 que si son ancienne valeur était à 0.

La paire d'instructions *Load Linked / Store Conditionnal (ll/sc)* est plus sophistiquée. En effet, il s'agit de deux instructions distinctes. Cependant elles doivent nécessairement être utilisées conjointement pour obtenir le résultat souhaité. L'instruction *Load Linked (ll)* permet de charger une variable mémoire tout en gardant un « traceur » sur cette variable. L'instruction *store conditionnal (sc)* permet d'écrire une variable mémoire si et seulement le « traceur » de cette variable indique que cette dernière n'a pas été accédée depuis l'exécution de l'instruction *ll*. Une manière basique de gérer le mécanisme associé à ces instructions est la suivante :

- L'adresse mémoire de la variable chargée par l'instruction *ll* est stockée dans un registre ou une table.
- Si l'adresse mémoire est accédée, ou si un changement de contexte à lieu, l'adresse est supprimée du registre ou de la table.
- Lorsqu'une écriture est tentée par l'instruction *sc* :
  - si l'adresse ciblée est toujours stockée dans le registre ou la table cela signifie que la variable n'a pas été accédée depuis l'instruction *ll*. L'écriture est alors réalisée. L'instruction renvoie un statut « succès ».
  - si l'adresse ciblée n'est plus dans le registre ou la table, cela signifie que la variable a probablement été accédée depuis l'instruction *ll*. L'écriture n'est pas réalisée. L'instruction renvoie un statut « échec ».

Par ailleurs, un autre avantage de la paire d'instruction *ll/sc* est qu'elle permet, du fait de sa séparation complète entre les instruction de lecture et d'écriture, d'implémenter toutes les autres instructions atomiques : *test-and-set*, *fetch-and-add*, *swap*,...

Une manière simple d'implémenter un verrou à partir de cette instruction est détaillée dans l'algorithme 3.

La pré-condition de cet algorithme est que le registre *R1* contienne l'adresse du verrou.

La première instruction “R2 ← *ll*(R1)", permet de charger la valeur contenue à l'adresse mémoire pointée par le registre *R1* dans le registre *R2*. Dans notre cas, *R2* contiendra l'état du verrou. En plus de ce chargement, comme expliqué précédemment, cette instruction permet de surveiller les accès sur cette case mémoire (à l'aide de ce que nous avons nommé « traceur »).

Ensuite, un test sur l'état du verrou contenu dans le registre *R2* est effectué. Si le verrou est déjà pris (valeur différente de 0) alors le programme reboucle directement sur le label *label_lock* tant que le verrou n'est pas libre. Nous pouvons remarquer que dans cette boucle, aucune tentative de modification de la valeur du verrou n'est effectuée. Ainsi, aucune demande d'accès exclusif à la ligne de cache n'est réalisée.

Si le verrou était libre (valeur égale à 0), le registre *R2* se voit affecter la valeur 1, puis une tentative d'écriture en mémoire de la valeur contenue dans *R2* à l'adresse pointée par *R1* est réalisée grâce à l'instruction *sc*.

Si la case mémoire pointée par le registre *R1* n'a pas été modifiée par une autre processus depuis l'instruction *ll*, alors l'instruction *sc* réussie, la valeur de *R2* est écrite dans la case mémoire pointée par *R1*, et l'instruction *sc* retourne la valeur 1 dans le registre *R2* (indiquant le succès de l'opération).

Si la case mémoire pointée par le registre *R1* a été modifiée depuis l'instruction *ll*, l'instruction *sc* échoue, la valeur de *R2* n'est pas écrite en mémoire, et l'instruction *sc* retourne la valeur 0 dans *R2* (indiquant l'échec de l'opération).

Enfin, un dernier test est réalisé afin de vérifier si la prise de verrou a réussi, c'est à dire si l'instruction *sc* a réussi. Si le registre *R2* contient la valeur 0, cela signifie que l'instruction *sc* a échoué. Le programme reboucle alors sur le label *label_lock* afin de tenter à nouveau la prise du verrou. Si le registre *R2* contient la valeur 1, alors la prise de verrou a réussi. La tâche peut alors rentrer dans sa section critique.

Algorithme 3 Verrou *ll*&*sc*

Require: Le registre R1 contient l'adresse du verrou
label_lock :
R2 ← *ll* (R1) // chargement de la valeur du verrou dans R2 + surveillance des accès à l'adresse contenue dans R1
if R2 ≠ 0 **then**
go to label_lock
end if
R2 ← 1
sc R2,(R1) // tentative d'écriture de la valeur de R2 à l'adresse pointée par R1, ensuite R2 reçoit le statut de l'opération
if R2 == 0 **then**
go to label_lock
end if

Le couple d'instructions *ll/sc* est aujourd'hui disponible dans de nombreux processeurs : ARM, MIPS ou encore Power PC, et est largement utilisé pour implémenter les mécanismes de synchronisation [lls18].

Historiquement, l'instruction atomique sur laquelle reposait les mécanismes de synchronisations sur les processeurs ARM était « SWP » (swap) qui permettait d'échanger une valeur en mémoire de manière atomique. Depuis les architectures ARMv6 cette instruction a été remplacée par les instructions « LDREX » (Load-Exclusive) et « STREX » (Store-

Exclusive) correspondant respectivement aux instructions *Load Link* et *Store Conditional*. Le document [ARM09] revient sur ces instructions et présente un example d'implémentation de *mutex*.

En ce qui concerne les processeurs Power PC les instructions atomiques « lwarx » et « stwcx » correspondent aux instructions *Load Link* et *Store Conditional*.

Cette évolution des mécanismes de synchronisation en raison de l'influence des mémoires caches et des protocoles de cohérence de caches met en évidence le caractère essentiel que revêt la prise en compte de la hiérarchie des mémoires caches ainsi que de l'architecture matérielle des MPSoC afin de proposer des mécanismes performants.

Comme nous venons de le voir dans ce paragraphe, l'influence de la cohérence de cache a déjà été adressée, et des propositions adaptées ont été mise en place. Cependant, nous avons vu dans le chapitre 1 qu'une des caractéristiques importantes des architectures modernes des MPSoC était l'organisation des éléments du système sous forme de grappe (cluster). Aussi, une question se pose : **Est-il possible de tirer profit de la propriété d'organisation en grappe des architectures MPSoC lors de l'établissement des synchronisations inter-processus ?** Nous tenterons de répondre à cette question dans ce manuscrit.

2.1.4 Illustrations des problèmes liés aux mécanismes de synchronisation

Malgré l'utilisation de ces instructions optimisées, des problèmes persistent dans les mécanismes de synchronisation. Ces problèmes se traduisent par l'apparition de délais dans l'établissement de la synchronisation. Dans cette partie, nous nous proposons d'illustrer quelques problèmes typiques pouvant survenir.

Lorsque nous parlons de synchronisation, nous considérons qu'il existe deux sources de ralentissement de la vitesse d'exécution d'une application :

— une première catégorie de ralentissements due à l'application proprement dite. En effet, lors de l'établissement d'une barrière par exemple, il est aisé de comprendre que s'il y a un déséquilibre au niveau du temps de calcul entre les différentes tâches, la première tâche arrivée devra attendre la dernière. Un délai est alors introduit par cette attente. Il s'agit de délais intrinsèques à l'application ;

— une deuxième catégorie de ralentissements résultants directement des mécanismes de synchronisation. En effet, les algorithmes mis en place pour la réalisation des synchronisations peuvent générer eux même des pertes de temps, à l'exemple des problèmes de contention illustrée figure 2.4. Ces problématiques seront expliquées plus en détail dans la prochaine sous section.

Les ralentissements intrinsèques à l'application sont hors du champ de notre étude. Nous nous concentrerons donc sur les problèmes résultants des mécanismes de synchronisation.

Problèmes liés aux verrous

La figure 2.3 illustre un problème pouvant survenir lorsque plusieurs processus, s'exécutant sur plusieurs cœurs, demandent l'accès à un verrou simultanément. Les axes temporels représentent respectivement le *thread 1* (ou processus 1) pour l'axe nommé *T1* sur la figure, le *thread 2* (ou processus 2) pour l'axe nommé *T2*, le verrou pour l'axe nommé *Lock*.

Dans le système *TSAR*, les tentatives d'acquisition du verrou sont réalisées par l'instruction atomique *compare&swap*. Cette instruction compare la valeur contenue à l'adresse mémoire ciblée avec l'un de ses paramètres. En cas d'égalité, une valeur passée en paramètre de l'instruction est écrite dans la variable mémoire. Sinon, aucune écriture ne se produit.

1. Sur le schéma, nous pouvons voir que *T1* fait une demande d'acquisition du verrou. Le verrou étant libre, il est instantanément passé à l'état « pris » et *T1* peut exécuter sa section critique protégée par le verrou.
2. Ensuite, *T2* fait, à son tour, une demande d'acquisition du verrou. Or ce dernier est déjà pris par *T1*. L'acquisition échoue donc et *T2* s'endort.
3. Une fois sa section critique achevée, *T1* libère le verrou et réveille le prochain processus en attente, dans notre cas *T2*.
4. Les étapes de réveil de *T2* sont représentées en bas du schéma dans le cercle bleu. Ce dernier doit passer par trois phases successives avant de reprendre son exécution de manière effective. En raison de cela, le temps de réveil peut être très long.
5. Le schéma représente ensuite *T1* qui continue son exécution sur son cœur de traitement, essayant d'acquérir à nouveau le verrou. Du fait que *T2* n'a pas encore pu renouveler sa demande du verrou dans l'intervalle de temps en raison des étapes de réveil, *T1* acquiert à nouveau le verrou et peut entrer dans sa section critique.
6. Une fois *T2* effectivement réveillé, il tente d'acquérir en vain le verrou déjà repris par *T1*. *T2* s'endort à nouveau en attente du verrou.

Ce schéma illustre un cas où la politique d'acquisition du verrou est injuste car les processus ne sont pas servis dans leur ordre d'arrivée. De plus, le comportement décrit peut se reproduire indéfiniment et mener à un phénomène de famine. Enfin, nous pouvons remarquer que même en cas de réussite, un temps non négligeable (réveil + *compare&swap*) est nécessaire entre le moment où le verrou est libéré, et le moment où il est effectivement repris par un autre *thread*.

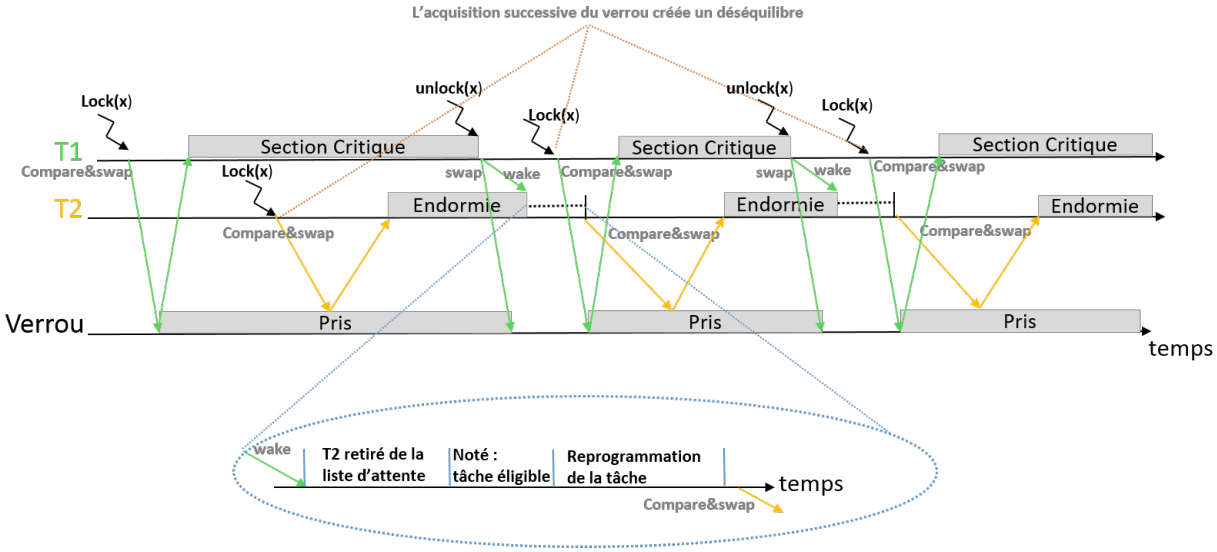

FIGURE 2.3 – Illustration du problème lié aux verrous

Problèmes liés aux barrières de synchronisation

Le schéma 2.4 illustre deux problèmes de contention liés à l'établissement d'une barrière.
Les variables relatives à l'état de la barrière sont stockées dans le cache L2 de la grappe en haut à gauche (petit verrou noir). La détermination du cache L2 dans lequel ces informations sont cachées résulte de l'adresse mémoire de ces variables. En effet, comme expliqué précédemment en section 1.3, chaque mémoire cache L2 couvre une plage d'adresses distincte de la mémoire, c'est ce que nous avions appelé « gestion distribuée » de la mémoire.
Lors de la réalisation d'une barrière, si les différentes tâches participant à la barrière sont bien équilibrées (charge de calcul identique), l'ensemble des tâches tentent de signaler leur arrivée à la barrière quasiment au même instant. Cela peut engendrer des phénomènes de contention à différents endroits de l'architecture. Le schéma 2.4 illustre certaines de ces contentions :

— au niveau des routeurs, comme illustré en bas à gauche du schéma 2.4. Le routeur ne pouvant traiter qu'une trame à la fois, lorsque que plusieurs trames arrivent au même instant, certaines seront mises en attente du passage des autres, ce qui introduit des délais.

— au niveau du contrôleur mémoire, comme illustré en haut à gauche du schéma 2.4. En effet, ce dernier ne peut gérer qu'un seul accès la fois. Lorsque plusieurs demandes d'accès concurrentes arrivent à la mémoire, le contrôleur les sérialise et les traite les unes après les autres. Un délai sur les traitements des requêtes apparaît donc en raison de cette sérialisation.

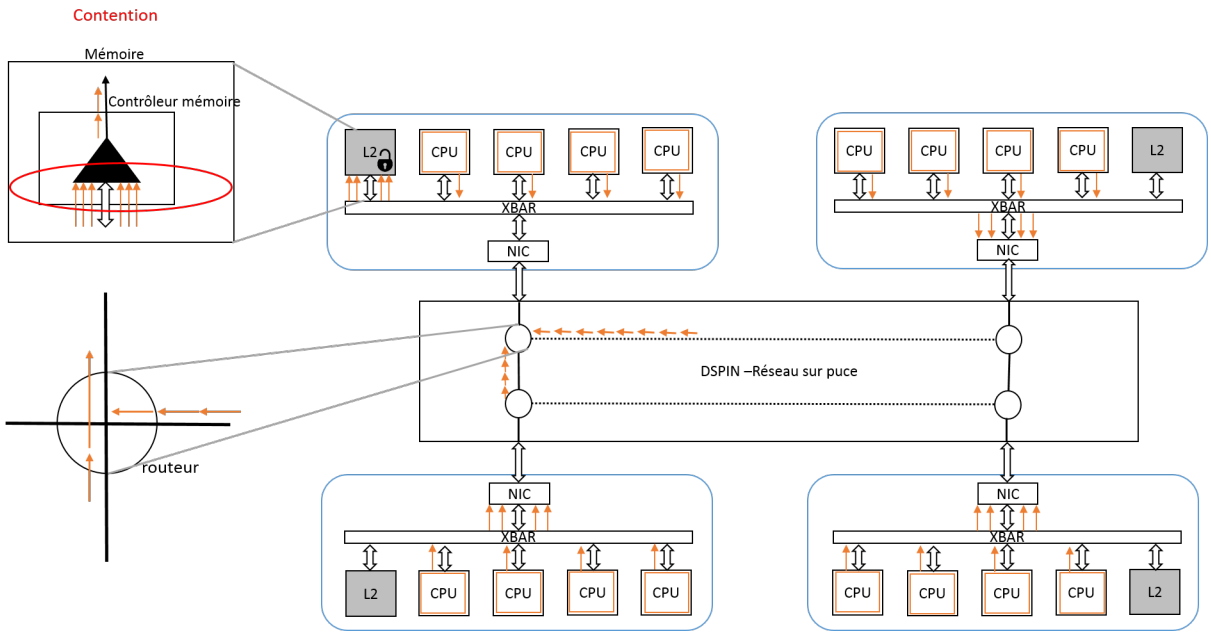

FIGURE 2.4 – Illustration du problème lié aux barrières de synchronisation

Un autre problème peut survenir lors du relâchement des processus. Dans le système *TSAR* comme dans de nombreux MPSoC, le réveil des différents processus, à la suite d'une barrière de synchronisation, est réalisé de manière séquentielle. En effet, le dernier processus arrivé a la charge de réveiller un à un tous les processus en attente. Cela

peut prendre un temps non négligeable, et générer beaucoup de trafic réseau lorsque le nombre de participants à la barrière est élevé (1 trame par processus).

2.2 Du matériel au logiciel

2.2.1 Entre logiciel et matériel

Comme nous avons pu le percevoir depuis le début de ce manuscrit, la synchronisation inter-processus est une problématique répartie entre le logiciel et le matériel.

En effet, ce type de mécanisme, à la base logiciel, est fortement lié avec le matériel du fait que les synchronisations peuvent être garantie uniquement grâce à l'utilisation d'instructions atomiques fournies par le support matériel. Par ailleurs, la couche logicielle peut aussi renfermer une certaine complexité dans l'implémentation des services demandés par les programmeurs (sémaphores, barrières, ...).

En raison de la dualité de ces mécanismes, deux optiques de traitement de la problématique de synchronisation sont possibles :

1. Développer un mécanisme de synchronisation reposant sur une partie matérielle minimaliste (instructions atomiques), optimisé grâce à une partie logicielle performante. Dans ce cas, le matériel n'est utilisé que pour garantir les accès atomiques à une variable partagée, l'optimisation étant dans la mise en œuvre d'algorithmes sophistiqués.
2. Développer un mécanisme de synchronisation optimisé grâce à un support matériel performant. Dans ce cas, la partie logicielle est utile uniquement pour faire le lien entre l'application utilisateur et le module matériel. Cette stratégie propose donc la délégation partielle ou complète de la gestion des synchronisations à un bloc matériel alors que le logiciel permet uniquement de communiquer avec celui-ci.

2.2.2 L'importance de l'environnement de fonctionnement

Sachant que les problèmes de synchronisation englobent des aspects matériels mais aussi logiciels, nous comprenons l'importance d'évaluer les performances de ces mécanismes dans leurs environnements réels de fonctionnement. Ce n'est que de cette manière que les vrais problèmes et ralentissements dus à ces mécanismes peuvent être mis en exergue, et qu'une solution adaptée et efficace peut être mise en place. En effet, il est important de noter que dans ces types de mécanismes, le comportement logiciel vient influer sur les performances du matériel, tout comme le matériel affecte le déroulement du logiciel.

En raison de ces interactions matérielles/logicielles, un certain nombre de mécanismes de synchronisation proposés par le monde académique s'avèrent difficilement exploitables dans de vrais systèmes, et notamment dans des systèmes industriels.

Par ailleurs, de nombreuses solutions proposées ne prennent que peu en compte la facilité de mise en place de leur mécanisme dans un système réel, ainsi que l'acceptabilité de celle-ci de la part des programmeurs.

Aussi, à ce stade, une question peut émerger : **Comment fournir un mécanisme permettant d'accélérer les applications utilisateurs (à haut-niveau), tout en restant simple à mettre œuvre ?** Nous essaierons de proposer une réponse à cette question dans ce manuscrit.

2.3 Observabilité des synchronisations

2.3.1 Identification des problèmes et des ralentissements

Lorsque nous cherchons à optimiser un système de synchronisation, il est nécessaire d'avoir une cartographie précise des temps passés dans chaque section de ce mécanisme. En effet, c'est à partir de ce rapport des délais mesurés que nous pouvons identifier les éventuelles fonctions sous-optimales et proposer des contre-mesures adaptées.

Ainsi, nous comprenons l'importance de disposer d'un système nous permettant d'obtenir des informations temporelles précises sur le déroulement du logiciel.

Par ailleurs, nous venons de voir dans la section précédente 2.2 l'impact du matériel sur le logiciel et notamment sur les mécanismes de synchronisation. Il devient alors essentiel de prendre en compte le support matériel ainsi que ses limitations lors de l'étude des ralentissements des mécanismes de synchronisation.

2.3.2 Méthodes d'évaluation temporelle

En ce qui concerne la mesures des temps (délais) passés dans les fonctions d'une application logicielle s'exécutant sur un MPSoC deux grandes familles de méthodes peuvent être identifiées : celles reposant sur l'exécution native du programme, et celles exploitant des plateformes de simulation.

Exécution native

La première méthode permettant d'évaluer les performances d'une application logicielle en prenant en compte l'impact du matériel ciblé, est d'exécuter le programme directement sur le système cible. C'est ce que nous appelons exécution native. L'exécution sur ce type de cible est très rapide. Aussi, il est possible d'exécuter des applications complexes mettant en œuvre les comportements réels des mécanismes étudiés dans leur environnement de fonctionnement usuel.

Malheureusement, ce type de système s'apparente généralement à une boîte noire, de laquelle les informations temporelles sont difficiles à extraire. La méthode permettant d'obtenir ce type d'information consiste alors en l'ajout, au niveau logiciel, d'instructions dites d'instrumentation, traçant le temps à des points stratégiques du programme. Néanmoins, des inconvénients sont à déplorer pour cette méthode :

— L'ajout d'instructions d'instrumentation au code initial insère des délais supplémentaires qui peuvent impacter le flot d'exécution nominal du processus, transformant ou masquant, par effet de bord, les sources des ralentissements du mécanisme. En raison de cela, les points de mesure ne pourront pas être très nombreux et très précis, afin de limiter l'intrusion de ces mesures dans le déroulement de l'application.

— L'accès au code source de l'application est nécessaire afin de pouvoir rajouter les instructions d'instrumentation. Dans le cas où le code ne serait pas disponible, il est possible de contourner cette limitation en ajoutant les instructions de mesures temporelles au niveau des appels systèmes, mais cela induit une mesure peu précise.

Plateforme de simulation

Afin de palier le problème d'intrusion lors de la mesure des délais, il est possible d'utiliser des mesures par canaux cachés. Le principe de ce type de mesures est d'utiliser l'observation du matériel (notamment le compteur ordinal, *program counter*) afin d'en déduire le comportement du processus logiciel. Pour se faire, il est alors nécessaire de pouvoir observer le support matériel au cours de l'exécution du logiciel. Les plateformes de simulation permettent ce genre de mesures par canaux cachés, à l'exemple du simulateur Gem5 [gem18] qui offre la possibilité de récupérer le compteur ordinal annoté avec le numéro de cycle associé. Grâce à ces informations, il devient alors possible d'extraire des informations temporelles de manière non intrusive avec un grain de mesure fin. L'étude précise du mécanisme ciblé est alors possible.

Cependant, l'inconvénient de cette méthode est sa vitesse d'exécution. En effet, le temps nécessaire pour effectuer l'évaluation temporelle dépend directement de la vitesse de la plateforme de simulation.

Lorsque nous parlons de simulation, nous devons raisonner en terme de niveaux de précision du support matériel simulé. En effet, trois grandes familles de niveau d'abstraction du matériel peuvent être identifiées :

— Le niveau algorithmique : ce niveau d'abstraction permet uniquement de valider le fonctionnement général d'une application, et ne permet pas d'effectuer des mesures temporelles représentatives.

— Le niveau transactionnel : ce niveau d'abstraction descrit les transactions entre les éléments du système simulé. Initialement, ces modèles ne permettent pas de faire de mesures de délais. Cependant, des modèles enrichis, tel que celui proposé par Cheung *et al.* [CHB09], permettent de faire des mesures temporelles. Néanmoins, la précision du modèle du support matériel reste généralement trop grossier pour mettre en évidence l'influence du matériel sur le comportement réel des applications.

— Le niveau cycle : Ce niveau garantit une précision au cycle près. À ce niveau de précision, bien qu'il s'agisse toujours d'un modèle de la plateforme ciblée, l'étude du comportement temporel d'une application logicielle en fonction des spécificités du support matériel peut être réalisée. Cependant, la simulation à ce niveau de précision est très lente. Par example, le démarrage d'un noyau Linux sur un MPSoC de 16 cœurs simulé avec un environnement SystemC[Sys18] « allégé » (SystemCass [BG07]) peut mettre plusieurs jours. Aussi, il n'est pas envisageable de mener une campagne de mesures mettant en jeu des applications réelles complexes avec de tels temps de simulation.

En résumé, bien que cette seconde méthode de mesures non intrusive semble la plus adaptée pour effectuer une campagne de mesures précises dans le but d'évaluer les problèmes des mécanismes de synchronisation, cette dernière est soumise à compromis inhérent aux plateformes de simulation, à savoir de choisir entre vitesse de simulation ou précision du modèle simulé. De plus, elle nécessite de disposer d'un modèle de simulation du support matériel ciblé.

À ce stade, nous prenons conscience que la mise en évidence des problèmes des mécanismes de synchronisation n'est pas triviale. En effet, aucune des méthodes existantes ne permet de répondre à nos attentes à la fois en précision de définition du matériel, mais aussi dans la vitesse de mesures permettant d'évaluer les mécanismes dans leurs conditions normales d'utilisation. Nous pouvons alors nous poser la question suivante :

Comment pouvons nous observer de manière précise le comportement d'applications logicielles complexes sur MPSoC ? Dans ce manuscrit, nous tenterons d'apporter une réponse à cette question.

2.4 Conclusion

Dans ce chapitre, nous avons présenté l'utilité de synchroniser des processus afin que ceux-ci puissent coopérer lors de l'accomplissement d'une ou plusieurs tâches.

En effet, c'est grâce à ces synchronisations qu'un échange de données cohérentes entre processus peut être réalisé. Nous comprenons alors l'importance des synchronisations dans l'univers actuel des systèmes embarqués, où l'augmentation des performances passe par la parallélisation des tâches sur différentes unités de traitement. Aussi, la rapidité d'établissement de la synchronisation est déterminante pour le temps d'exécution global d'une application sur les systèmes fortement parallèles que sont les *many-cœurs*.

Par ailleurs, nous avons montré que les mécanismes de synchronisation peuvent introduire des délais pour différentes raisons, et notamment à cause du phénomène de contention. Il apparaît crucial de fournir des mécanismes de synchronisation optimisés afin d'améliorer les performances des MPSoC.

Dans le chapitre précédent, nous avions défini les types d'architectures de MPSoC ciblées, à savoir un système à mémoire partagée regroupé sous forme de grappes communicant entre elles à travers des réseaux sur puce.

À partir de ces hypothèses, nous avons soulevé les interrogations suivantes :

— Est-il possible de tirer profit de la propriété d'organisation en grappe des architectures MPSoC lors de l'établissement des synchronisations inter-processus ?

— Comment fournir un mécanisme permettant d'accélérer les applications utilisateurs (à haut niveau), tout en restant simple à mettre œuvre ?

— Comment pouvons nous observer de manière précise le comportement d'applications logicielles complexes sur MPSoC ?

Les chapitres 4, 5 et 6 de ce manuscrit ont pour objectif de répondre à ces questions.

Chapitre 3

État de l'art

Sommaire

Dans ce chapitre, nous nous proposons de présenter les travaux majeurs ayant trait aux mécanismes de synchronisation dans les MPSoC. Nous commencerons par un bref aperçu historique des mécanismes de synchronisation. Ensuite, nous nous recentrerons sur les solutions proposées pour améliorer leurs performances dans les MPSoC. Enfin, nous conclurons sur les tendances observées à travers l'étude bibliographique et sur les opportunités d'amélioration.

3.1 Un peu d'histoire

L'étude des problèmes liés aux mécanismes de synchronisation n'est pas récente. Elle débuta avant l'apparition des MPSoC, lorsque les processeurs ont commencé à être multi-processus. Les premières propositions d'optimisation remontent ainsi aux années 1980.

Depuis, de nombreuses solutions d'amélioration de ces mécanismes ont été proposées, principalement pour les plateformes à mémoire partagée. Nous n'allons pas ici refaire tout l'historique de ces propositions, nous nous contenterons de citer quelques unes d'entre elles qui nous semblent les plus notables.

Les mécanismes de synchronisation étant à la frontière entre le logiciel et le matériel, deux approches d'optimisation sont envisageables :

— logicielle : visant l'amélioration des performances des mécanismes de synchronisation par une optimisation algorithmique ;

— matérielle : implémentant un ou des bloc(s) matériel(s) spécifique(s) dédié(s) à l'établissement des synchronisations.

Nous nous efforcerons de respecter cette division dans le suite de cette section afin de clarifier le plus possible l'optique dans laquelle les solutions d'optimisation ont été proposées.

3.1.1 Les solutions logicielles

Comme évoqué dans le chapitre 2, l'un des problèmes des mécanismes de synchronisation est la contention résultant des accès simultanés à une variables partagées indiquant l'état de la synchronisation. À cela s'ajoute les problématiques de cohérence de caches qui apparaissent lors de tentatives d'accès à la même variable par des processus s'exécutant sur différents cœurs de calculs.

Ces problèmes étant exacerbés par l'interrogation périodique d'une variable de synchronisation unique durant l'attente active, l'algorithme **MCS** proposé par **M**ellor-**C**rummey et **S**cott [MCS91] a pour objectif de les réduire en implémentant une file logicielle des processus en attente active d'obtention du droit d'accès à un verrou. Grâce à cette file, l'information n'est plus centralisée en un seul point, mais chaque processus reste en attente sur une variable d'état qui lui est propre. L'algorithme **MCS**, et sa file de processus en attente, permet de passer le verrou directement d'un processus à un autre sans repasser par un point central. En effet, lorsqu'un processus libère le verrou, il donne le droit d'accès au processus suivant dans la liste en venant modifier directement la variable d'état associée au processus successeur.

Cet algorithme, implémenté dans le système Linux depuis le noyau 3.15, est le plus adapté pour des verrous très demandés, subissant donc une forte contention. Cependant, le fait de créer et de maintenir cette file logicielle engendre une légère pénalité d'accès au verrou lorsque celui est libre, comparé aux solutions basiques centralisées utilisant l'instruction *test-and-set* par exemple.

Les environnements d'exécution des mécanismes de synchronisation sont très variables. Deux verrous d'un même programme ne sont pas soumis à la même contention. De la même manière, la contention subie par un verrou peut varier dans le temps. Cette observation a conduit Lim et son équipe à proposer un verrou adaptatif [LA94]. Celui-ci est capable d'adapter son algorithme automatiquement en fonction de la contention subite par le verrou. Lorsqu'un verrou est soumis à une forte contention l'algorithme **MCS** est utilisé. Par contre, lorsque le verrou est peu victime de contention, une solution basique et centralisée, exploitant l'instruction *test-test-and-set*, est mise en œuvre.

Cette idée du verrou adaptatif a été reprise, par la suite, et développée dans des travaux plus récents à l'instar du *smart lock* de Eastep *et al.* [EWSA10] qui étend le principe d'adaptabilité à d'autres paramètres du verrou, tels que la politique d'attribution du verrou par exemple. En effet, grâce à l'exploitation du principe d'apprentissage automatique ou *machine learning*, ils proposent de déterminer la politique d'attribution du verrou en fonction du temps d'exécution des sections critiques des différents processus, en favorisant, par exemple, les sections critiques courtes.

L'adaptabilité des mécanismes de synchronisation représente un avantage certain car il devient alors possible de bénéficier des meilleurs algorithmes quelque soit l'environnement de fonctionnement. Cependant, la définition d'un modèle d'utilisation d'un verrou de manière précise, et cela en cours d'exécution, n'est pas chose aisée. Des algorithmes précis d'apprentissage et de prise de décision renferment une certaine complexité pou-

vant ajouter un surcoût sur le chemin critique d'acquisition du verrou. À l'opposé, des algorithmes trop minimalistes ne sont pas en mesure de prendre des décisions pertinentes. Aussi, un compromis complexe est à trouver, ce qui explique probablement le fait que l'utilisation ce type de solutions ne se soit pas généralisée dans les systèmes multiprocessus.

Une autre possibilité permettant de lutter contre la contention mémoire consiste à ne plus faire d'attente active sur la variable de synchronisation, mais de réaliser une attente passive réduisant drastiquement les accès mémoires, et donc la probabilité des problèmes liés à ces accès.

En effet, dans le cas de l'attente passive, le processus n'interroge plus périodiquement la variable mémoire durant la totalité de sa période d'attente mais ne l'interroge qu'une seule fois. Si l'accès lui est refusé, alors le processus s'endort (i.e. sort de la liste des processus éligibles par l'ordonnanceur pour l'exécution). Ce processus est alors réveillé par le système uniquement lorsque l'élément de synchronisation est à nouveau disponible (libération du verrou, de la barrière, ...).

L'attente passive permet ainsi de réduire le phénomène de contention, mais aussi de libérer le cœur de calcul pour l'exécution d'autres tâches plus utiles en lieu et place de l'interrogation périodique. Cela nécessite néanmoins de stocker l'identifiant du processus dans une liste avant son endormissement afin qu'il puisse être réveillé par la suite.

Une implémentation logicielle optimisée de ce type de mécanisme est réalisée à travers le principe des ***FUTEX*** Linux (**F**ast **U**ser space mu**TEX**) [FRK02]. Il s'agit d'un mécanisme de verrou implémentant l'attente passive dont le but principal est la réduction du nombre de basculements du système d'exploitation du mode d'exécution utilisateur au mode noyau.

En effet, dans les système d'exploitation tels que Linux, une ségrégation est réalisée entre espace utilisateur et espace noyau. Ainsi, certains services du système d'exploitation, et certaines adresses mémoire, ne sont accessibles que depuis l'espace noyau, et cela à des fins de sécurité. Si une application utilisateur souhaite accéder à ces services privilégiés, elle doit réaliser un appel système. Celui-ci permettra de basculer de l'espace utilisateur à l'espace noyau, puis exécutera le service, avant de revenir dans l'espace utilisateur. Ce basculement d'un espace à l'autre implique une sauvegarde du contexte d'exécution qui peut s'avérer coûteuse en nombre de cycles. Afin de garantir de bonnes performances, le mécanisme de *futex* cherche donc à réduire le plus possible le nombre de basculements vers le mode noyau. Ainsi, lorsque le verrou est disponible, le processus peut l'acquérir directement depuis le mode utilisateur grâce au mappage virtuel de la mémoire (*virtual memory mapping*). Le basculement en mode noyau est réalisé uniquement lorsque le verrou est déjà occupé, afin de procédé à l'endormissement du processus en attente d'accès. Le principe de fonctionnement est le suivant : tous les processus indexent une même structure *futex* (contenant l'état du verrou) située dans une mémoire partagée dans leurs espaces utilisateurs. Lorsqu'un processus souhaite acquérir le verrou, il effectue une opération atomique (*atomic_decrement*) sur la variable stockant l'état du verrou dans l'espace utilisateur :

— Si le verrou est disponible alors le verrou sera dorénavant noté comme occupé, et le processus peut accéder à la section protégée.

— Si le verrou était déjà occupé, alors le processus effectue un appel système. Une fois passé en mode noyau, le processus est mis dans une file d'attente logicielle associée au *futex*, puis le processus s'endort.

Lors de la libération du verrou, le processus va modifier (incrémenter) la variable mémoire de l'espace utilisateur relative à l'état du verrou. Si des processus sont en attente d'accès au verrou, alors le processus relâchant l'élément de synchronisation procédera à un appel système afin de réveiller le prochain processus dans la file d'attente noyau.

Le *futex*, implémenté dans le noyau linux depuis la version 2.5.7, permet donc l'implémentation logicielle optimisée du principe d'attente passive tout en limitant le nombre de basculements du mode utilisateur au mode noyau.

3.1.2 Les solutions matérielles

L'ajout de matériel spécifique dédié aux synchronisations est le second axe d'optimisation envisageable. Au sein même de cet axe, nous pouvons identifier deux tendances : l'ajout d'une mémoire tampon dédiée, et l'ajout d'une infrastructure de communication réservée aux synchronisations.

Une mémoire tampon dédiée pour la gestion des synchronisations

Il s'agit de l'ajout d'une mémoire tampon (*buffer*) ainsi que son contrôleur gérant les demandes de synchronisation. L'idée est de créer une file matérielle des processus en attente de synchronisation. La proposition de Monchiero *et al.* [MPSV06] s'inscrit dans cette perspective. Il s'agit d'une mémoire dédiée capable de gérer deux types de requêtes : *lock* et *event*. Lorsqu'un processus émet une requête pour l'obtention du verrou (requête *lock*), le *synchronization-operation buffer* de Monchiero *et al.* intercepte cette demande puis vérifie l'état du verrou. Si le verrou est libre, alors l'adresse mémoire de celui-ci est stockée dans la mémoire tampon dédiée, le verrou est noté comme « pris », et le processus demandeur reçoit le droit d'accès. Si un processus demande l'accès à un verrou déjà pris, alors le *synchronization-operation buffer* enregistre l'identifiant du processus dans une file d'attente associée au verrou, puis renvoie un signal de refus d'accès au processus demandeur qui peut alors s'endormir. Lorsque que le verrou est libéré par un processus, le *synchronization-operation buffer* envoie directement un message au processus suivant dans la file de ce verrou afin de lui accorder le droit d'accès.

Le *synchronization-operation buffer* implémente un principe similaire pour la requête *event*. Les processus s'enregistrent auprès du système comme demandeur d'un événement (typiquement le fait qu'une variable atteigne une valeur déterminée). Celui-ci va alors stocker l'identifiant du processus demandeur dans une file mémoire correspondante à l'événement. Lorsque l'événement se réalise, le *synchronization-operation buffer* réveille l'ensemble des processus en attente de son occurrence. Ce type de requête permet l'établissement des barrières de synchronisation.

La solution proposée permet de gagner :

— en temps global d'exécution : le fait que le processus en attente puisse s'endormir et soit réveillé directement par le *synchronization-operation buffer* permet de libérer le processeur afin qu'il puisse effectuer des tâches utiles.

— en trafic réseau : un processus n'est plus obligé d'interroger périodiquement la mémoire afin de connaître l'état du verrou, trois trames maximum par processus suffisent pour l'établissement la synchronisation.

Cependant, cette solution implique la centralisation de la gestion de l'ensemble des éléments de synchronisation en un seul point du MPSoC, ce qui rend difficile le passage à l'échelle en raison des problématiques de contention. Par ailleurs, ce type de solution ne

propose qu'un nombre d'éléments de synchronisation limitées. De ce fait, lorsque l'ensemble des éléments offerts sont exploités, l'utilisateur doit utiliser des mécanismes logiciels moins performants.

Une infrastructure de communication réservée aux synchronisations

La seconde tendance d'optimisation matérielle ayant vu le jour consiste en l'ajout d'une infrastructure de communication réservée aux synchronisations. Cela permet de se prémunir contre le risque de contention au niveau du médium de communication résultant des autres types échanges (données, ...). En effet, Leiserson *et al.* [LAD+92] propose dès 1992 d'ajouter un réseau dédié pour les trames de contrôle (typiquement les synchronisations) afin que ces trames ne se collisionnent pas avec les trames associées aux données. Cependant, l'ajout d'un réseau complet dédié aux synchronisations peut avoir un coût matériel conséquent.

En règle générale, les solutions matérielles permettent d'obtenir de meilleures performances que les optimisations logicielles en raison de la lourde pénalité temporelle que représentent les couches logicielles. Cependant, un ensemble de services logiciels optimisés (bibliothèque logiciel, pilote du module matériel, ...) doit accompagner la solution matérielle afin de permettre son utilisation par les programmeurs d'application logicielles. De plus, ces solutions sont souvent moins flexibles, et proposent uniquement un nombre limité de mécanismes de synchronisation, à l'instar des solutions implémentant une mémoire dédiée, lorsque celle-ci est remplie, elle n'est plus capable de gérer des synchronisations supplémentaires. Ainsi, la majorité des solutions matérielles se retrouvent limitées par le nombre d'éléments de synchronisation qu'elles offrent. La technique généralement utilisée pour palier cette limitation est de « basculer » sur des synchronisations logicielles, moins performantes donc, lorsque que la limite matérielle est atteinte.

3.2 Optimisations des mécanismes de synchronisation pour les MPSoC

Dans la section précédente, nous avons tenté de donner un aperçu global de l'historique des propositions d'optimisation des mécanismes de synchronisation avant l'essor des MPSoC.

Dans cette section, nous revenons plus en détail sur les travaux de recherche menés dans le but d'optimiser les synchronisations pour des plateformes MPSoC tels que nous les avons définis dans le chapitre 1.

Là encore, l'option logicielle ou l'option matérielle peuvent être envisagées pour optimiser ces mécanismes. Nous garderons donc cette séparation lors de la présentation de l'état de l'art.

3.2.1 Les solutions logicielles

L'augmentation du nombre de cœurs de calcul soulève de nouvelles problématiques notamment quant à la cohérence de cache. Certaines études telles que celles menées par Signhal *et al.* [Sin93] ou Kuo *et al.* [KCK99], ou plus récemment Xiao *et al.* [XWG+16] partent du principe que le coût de la cohérence de caches est trop élevé (principalement en nombre de messages échangés) pour concevoir des mécanismes de synchronisation

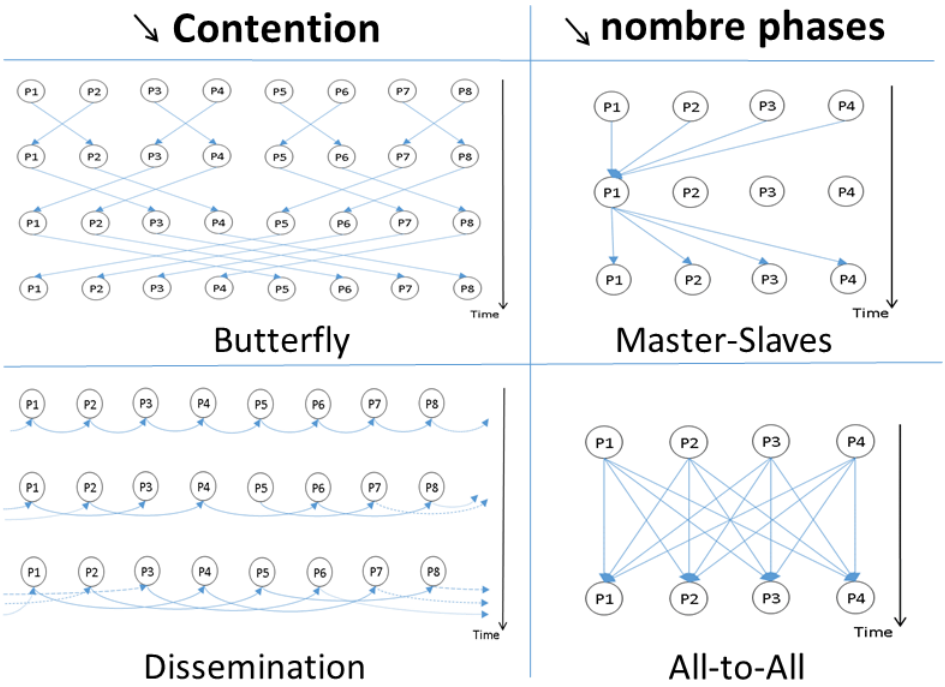

FIGURE 3.1 – Algorithmes d'échange de messages des barrières de synchronisation

performants, passant à l'échelle. Ils proposent alors d'implémenter un mécanisme de verrou au dessus d'échanges explicites de messages. La solution proposée par Kuo *et al.* [KCK99] prévoit l'élection d'un cœur comme étant le serveur du mécanisme de synchronisation. Ce dernier gère un ensemble de verrous, et peut :

— soit les garder dans sa mémoire locale et accorder le droit d'accès au processus demandeur en fonction de la disponibilité du verrou demandé,

— soit transférer physiquement un verrou (i.e. sa représentation mémoire) dans la mémoire d'un autre cœur tout en gardant une trace du cœur dans lequel est stocké le verrou. Ainsi, lorsqu'une demande explicite d'acquisition du verrou lui est envoyée, il peut rediriger la demande vers le cœur stockant physiquement le verrou. Cette seconde approche permet d'être plus efficace en cas de réacquisition du verrou par un même processus. En effet, le verrou étant déjà stocké dans la mémoire du cœur, le processus n'aura pas à émettre de requête par message explicite vers le serveur, mais pourra acquérir directement le verrou dans sa mémoire locale.

Concernant les barrières de synchronisation, en raison de leur utilisation intensive dans l'univers du **HPC** (**H**igh **P**erformance **C**omputing), des efforts importants d'optimisation de leurs algorithmes d'échange de messages pour des infrastructures de communication parallèles ont été menés avant l'avènement des MPSoC et des réseaux sur puce.

Comme illustré sur la figure 3.1, ces algorithmes ont été développés dans deux objectifs distincts. Une partie d'entre eux cherche à réduire la contention afin qu'un nœud ne reçoive pas plusieurs messages simultanément lors d'une même phase (ou ronde). Dans ce domaine, les algorithmes les plus aboutis sont : le *butterfly* et la *dissemination*. Le principe de l'algorithme *butterfly* (en haut à gauche sur la figure 3.1) est le suivant :

— Les processus sont appairés deux à deux avec leur voisin le plus proche. Lorsque qu'un processus atteint la barrière, il envoie son information d'arrivée à sa paire.

— Le processus attend ensuite l'information d'arrivée de sa paire.

— Une fois qu'un processus a connaissance du fait que lui même et sa paire sont arrivés à la barrière, la première phase de l'algorithme est finie pour ce processus, qui passe alors à la seconde phase.

— Dans la seconde phase, un processus est appairé avec une nouvelle paire (un voisin plus éloigné). Le processus envoie l'information de sa propre arrivée à barrière et de l'arrivée de sa paire de la phase 1 au processus paire de la phase 2. Ainsi, le processus paire de la phase 2 a la connaissance de l'arrivée à la barrière de 4 processus : sa propre arrivée, l'arrivée de sa paire phase 1, l'arrivée de sa paire phase 2 et l'arrivée de la paire phase 1 de son actuelle paire phase 2.

— Une fois qu'un processus à reçu les informations d'arrivée provenant de sa paire phase 2, la seconde phase est finie, et le processus peut commencer sa phase 3 avec une nouvelle paire.

— Ce procédé se répète jusqu'à ce que chaque processus ait connaissance de l'arrivée de l'ensemble des autres processus à la barrière. Une fois cet objectif atteint, la barrière est achevée, et le processus peut continuer son exécution.

Le principe de fonctionnement de l'algorithme *dissemination* (en bas à gauche sur le schéma 3.1) est similaire à celui du *butterfly* mis à part le fait que les processus ne sont plus appairés deux à deux, mais l'information d'arrivée d'un processus est envoyée au *x-ème* voisin avec *x* le numéro de phase. Cela permet de s'affranchir des problèmes liés au nombre de processus, nécessairement en puissance de 2 pour l'algorithme *butterfly*.

Les algorithmes du second type cherchent, quant à eux, à réduire le nombre de phases nécessaires à l'établissement de la barrière, au détriment de la contention. L'algorithme *maître-esclave* par exemple nécessite seulement 2 phases : une de récolte des informations d'arrivées, et une de libération des processus. L'algorithme *all-to-all* permet d'établir la barrière en une seule phase.

Le principe de l'algorithme *maître-esclave* (en haut à droite sur la figure 3.1) est d'attribuer des rôles aux processus participant à la barrière. Un processus est désigné en tant que maître de la barrière, il a pour mission de collecter toutes les informations d'arrivées à la barrière envoyées par les esclaves, et de signaler la fin de la barrière à tous les esclaves.

Dans l'algorithme *all-to-all* (en bas à droite sur la figure 3.1), tous les processus envoient leur information d'arrivée à la barrière à l'ensemble des autres processus. Ainsi tous les processus ont connaissance de l'état de la barrière, et sont capables de reprendre leur exécution lorsque la barrière est terminée.

En 2008, Villa *et al.* ont proposé une analyse du comportement de ces algorithmes sur les MPSoC implémentant un réseau sur puce [VPS08]. Celle-ci s'inscrit comme le début de l'intérêt des travaux de recherche pour l'exploitation des réseaux sur puces comme moyen d'optimisation des mécanismes de synchronisation. L'étude évalue les performances de différents algorithmes de barrière de synchronisation pour diverses topologies réseau (*mesh*, *anneau*, *tore*, ...), et conclut sur le fait que l'implémentation matérielle des algorithmes ayant le moins de phases d'établissement (typiquement *All-to-All* figure 3.1) exploite mieux le parallélisme intrinsèque des réseaux sur puce que pour les algorithmes composés de nombreuses phases (tels que l'algorithme *butterfly* 3.1). L'analyse montre aussi que la topologie la plus chère (typiquement *tore*) ne garantit pas forcement de meilleures performances qu'une topologie plus économique, car les algorithmes des mécanismes de synchronisation n'exploitent pas tout le parallélisme disponible.

Par la suite, en 2012, Gauthier *et al* [LFK+12] ont cherché à réduire le nombre de *hops* réalisés par les messages relatifs aux barrières de synchronisation, sur des MPSoC communiquant par un réseau sur puce de topologie *mesh* 2D. Le nombre de *hops* est la distance de Manhattan [1] entre deux processeurs exécutant deux processus participant à la barrière.

La proposition de Gauthier *et al* consiste à placer les processus sur la grille réseau en fonction de leur rôle dans l'algorithme de barrière, et cela dans le but de réduire la distance parcourue par les messages. Par exemple, comme représenté sur la figure 3.2 pour un algorithme de type *maître-esclaves*, le fait de placer le maître au milieu de la grille réseau permet de réduire le nombre de *hops* global nécessaire à l'établissement de la synchronisation.

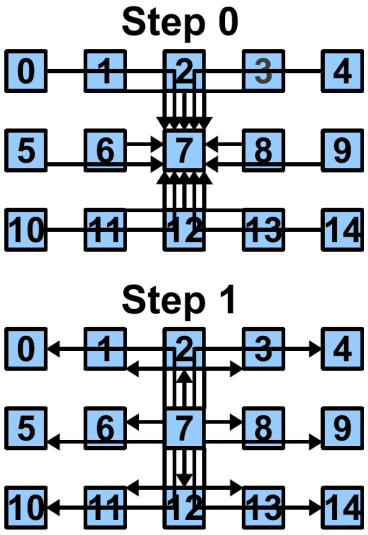

FIGURE 3.2 – Algorithme de barrière maître-esclaves : placement optimisé pour la réduction du nombre de *hops*[LFK+12]

Les objectifs motivant le fait de réduire la distance parcourue par les messages sont :

1. réduire le risque de contention au niveau des routeurs,
2. réduire la latence.

Au cours des dernières années, les travaux d'optimisation logicielle des barrières n'ont pas réellement proposé de solutions en rupture. Les solutions proposées tentent plutôt d'ajuster les techniques existantes à une architecture MPSoC donnée, en mixant les algorithmes existants [RNPL15], [WLSY15] ou encore [THL16]. L'idée ici est de diviser le MPSoC en sous-couches au sein desquelles un algorithme de barrière est mis en place. En ce qui concerne les échanges d'information entre sous-couches, un algorithme différent est utilisé. Grâce à cette stratégie, il devient plus facile de trouver un compris entre latence et contention lorsque le nombre de cœurs participant à la barrière devient élevé.

3.2.2 Les solutions matérielles

Au niveau matériel, quelques solutions d'optimisation des mécanismes de synchronisation exploitant les caractéristiques particulières des réseaux sur puce ont été proposées.

1. il s'agit de la distance la plus courte reliant 2 points d'un réseau ou d'un quadrillage, en ne passant que par les liens de ce réseau/quadrillage.

Dans cette section, nous présenterons les plus pertinentes.

Concernant le mécanisme de verrou, Chen *et al.* [CLJC10] propose d'implémenter des mémoires dédiées aux synchronisations réparties dans l'ensemble du MPSoC. En effet, comme exposé dans le chapitre 1, les cœurs des MPSoC de grande taille sont de plus en plus souvent regroupés en grappes (*clusters*). Le *synchronization handler* de Chen *et al.* prévoit d'exploiter ce principe architectural en répartissant leur système dans l'ensemble des grappes composant le MPSoC, comme représenté sur la figure 3.3 où chaque *PM* représente une grappe.

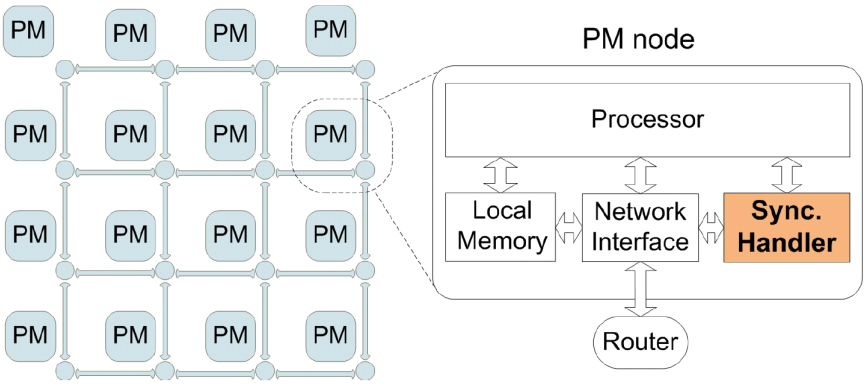

FIGURE 3.3 – Architecture du *synchronization handler*[CLJC10]

Chaque module *synchronisation handler* gère un ensemble de verrous dont l'état est stocké dans une mémoire dédiée. Chaque verrou dispose d'une adresse globale, et peut être adressé par n'importe quel processus du système. Lorsqu'un processus désire obtenir un verrou, il envoie une requête explicite vers le *synchronization handler* qui possède ce verrou. Le module est capable de traiter les trames qu'il reçoit et d'y répondre en fonction de l'état du verrou correspondant. Le système est composé de deux mémoires tampons physiques (*buffers*) qui sont gérées selon la méthode des canaux virtuels. Un canal virtuel correspondant à un verrou. Lorsqu'une requête d'acquisition d'un verrou arrive et que le verrou est déjà pris, celle-ci est stockée dans le canal virtuel associé au verrou jusqu'à la libération de ce dernier. Ce système permet donc l'attente passive du processus qui, une fois la requête émise, peut s'endormir dans l'attente de la réponse du mécanisme de synchronisation. Si les mémoires tampons sont remplies alors le système renvoie un message d'erreur au processus, et la requête n'est pas stockée dans le module.

Une force de cette solution est d'exploiter le principe de localité. Les processus peuvent favoriser la prise de verrous locaux, ou dans des grappes proches de leur localisation, et cela afin d'éviter, ou de limiter, les transferts de messages sur le réseau, réduisant ainsi le temps de latence et les risques de contention réseau.

Concernant l'ajout d'infrastructures de communication dédiées aux synchronisations, des solutions minimalistes ont vu le jour. Bien moins coûteuses matériellement que la solution historique de Leiserson [LAD+92] de duplication complète du réseau, celles-ci sont sensées faciliter le passage à l'échelle de ce type de solutions. La proposition la plus notable est celle de Abellan *et al* qui présente le *g-lock* [AFA11] et la *g-barrier* [AFA12] permettant respectivement d'implémenter un verrou et une barrière de synchronisation à partir de *g-line*, des liens à faible latence permettent de transmettre jusqu'à six signaux en un cycle d'horloge. Le *g-lock* a été implémenté comme représenté sur le schéma 3.4, sur un système communiquant par un réseau sur puce de topologie *mesh*.

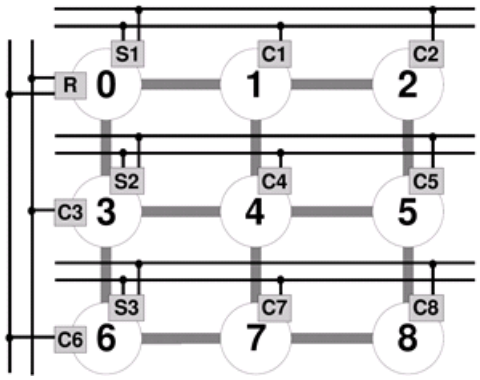

FIGURE 3.4 – Architecture d'un *g-lock*[AFA11]

Une *g-line* est allouée à chaque nœud du système à l'exception du nœud racine (*root*). La proposition d'Abellan *et al.* prévoit deux types de contrôleur :

— les contrôleurs locaux (représentés par la lettre 'c' sur la figure). Ces derniers permettent d'envoyer une requête de demande du verrou, et de recevoir la réponse.

— les contrôleurs de gestion (représentés par les lettres 's' et 'r' sur la figure). Le réseau de synchronisation *g-lock* est implémenté de manière hiérarchique. Les contrôleurs de gestions 'r' permettent de récupérer les requêtes d'une ligne du système et de les transmettre à la racine. La racine 'r' centralise l'ensemble des requêtes et attribue le verrou à un processus demandeur en fonction de sa disponibilité.

Les solutions de ce type sont très performantes car elles résolvent les problèmes de contention et de latence. Cependant, elle sont peu extensibles. En effet, elles ne permettent pas la gestion de demandes de synchronisation (verrou, barrière) de plusieurs processus s'exécutant sur un même processeur. Par ailleurs, lorsque le nombre de processeurs augmente, l'infrastructure de communication doit elle aussi augmenter. Un *g-lock* nécessite N−1 *g-links* pour implémenter un verrou sur un système composé de N processeurs. Pour disposer d'un verrou supplémentaire, l'ensemble du système *g-lock* doit être répliqué.

Aussi, même si les solutions proposées par Abellan *et al.* ou d'autre du même type [SSH+15] tentent de réduire le matériel dédié aux synchronisations comparativement à la solution de Leiserson [LAD+92], ces solutions rencontrent néanmoins toujours des difficultés à passer à l'échelle en raison de l'ajout de matériel spécifique.

L'essor des réseaux sur puce dans les MPSoC a ouvert le champ des opportunités d'optimisation des mécanismes de synchronisation. Dans ce contexte, Chen et Chen [CC11] ont proposé un système matériel permettant de réduire la contention réseau lors de l'établissement d'une barrière de synchronisation.

La solution, qui exploite l'algorithme *maître-esclaves*, implémente deux fonctionnalités :

— La fusion (*merge*) des messages des esclaves signalant au maître leur arrivée à la barrière. En effet, grâce à l'ajout d'un dispositif matériel au sein des routeurs du réseau, lorsque plusieurs messages d'arrivée à la barrière traversent un routeur au même instant, ces messages sont fusionnés en un seul et unique message indiquant le nombre de processus signalant leur arrivée.

— La diffusion (*broadcast*) du message de relâchement des processus en attente sur la barrière. Le message de réveil est alors envoyé à tous les éléments du MPSoC, ce qui

permet au bloc matériel gérant les barrières de ne pas avoir besoin de mémoriser les identifiants des processus en attente.

Cette proposition se construit autour de l'ajout de deux types de blocs matériels différents :

— Les modules feuilles : qui se situent dans les routeurs et permettent de réaliser la fusion et la diffusion des messages sur le réseau.

— Le module racine : qui gère de manière centralisée les barrières, et envoie le message de relâchement des processus lorsque ces derniers ont tous atteint l'élément de synchronisation.

Grâce à ces deux fonctionnalités, le nombre de messages transitant sur le réseau est réduit : la fusion réduisant le nombre de messages remontant vers le maître, et la diffusion remplaçant les n messages que l'algorithme de base *maître-esclaves* impose au maître d'envoyer à ses n esclaves, par un seul et unique message diffusé à l'ensemble des éléments composants le MPSoC.

Cependant, nous pouvons nous interroger sur la pertinence d'une fusion des messages telle que proposée par Chen et Chen [CC11]. En effet, la probabilité que deux trames signalant l'arrivée à la barrière de processus traversent une même routeur au même instant semble faible.

Cette observation a poussé Tseng *et al.* [THL16] a étendre cette idée de fusion des messages d'arrivées à la barrière. En effet, dans la solution qu'ils proposent, les messages ne sont plus seulement fusionnés lorsque ceux-ci traversent en même temps un routeur, mais les trames sont bloquées et fusionnées dans un routeur tant que le nœud local au routeur (cf schema 3.5) n'a pas encore émis sa trame d'arrivée à la barrière. Ainsi, le nombre de trames circulant sur le réseau est encore réduit.

Ce principe de fusion permet ainsi de mieux adresser le problème du trafic réseau. Néanmoins, cette solution nécessite que le système soit informé du nombre de processus participant à la barrière sur chaque nœud, afin de gérer le blocage des trames incidentes en conséquence.

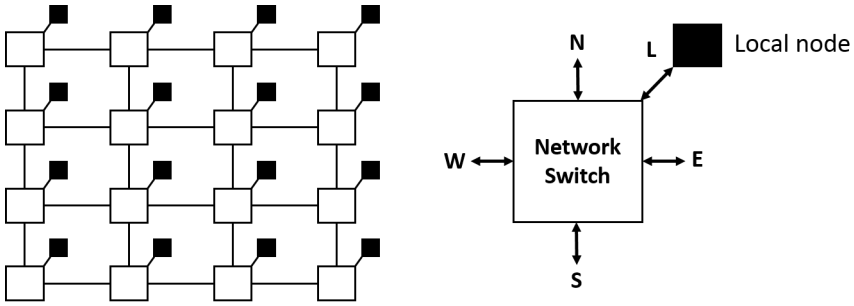

FIGURE 3.5 – Illustation du principe de nœud locale à un routeur [THL16]

3.3 Conclusion

Nous avons pu voir au cours de cet état de l'art que la problématique des mécanismes de synchronisation n'est pas récente. Depuis plus d'une trentaine d'années, et l'apparition des applications multi-processus, des travaux ont été menés afin d'optimiser ces mécanismes. Les optimisations proposées suivent deux optiques différentes :

— logicielle : l'amélioration provient d'un algorithme logiciel optimisé reposant sur un support matériel minimaliste (instructions atomiques simples).

— matérielle : l'amélioration vient d'un support matériel haute performance dédié à la gestion des synchronisations. Ces solutions intègrent un logiciel minimaliste réalisant le lien entre l'application utilisateur et le module matériel.

L'étude des solutions proposées pour chacune des stratégies nous a permis de voir que les solutions matérielles offrent de meilleurs performances car elles ne subissent pas les pénalités des couches logicielles affectant les performances de leurs pendants logicielles. Cependant, les solutions logicielles connaissent plus de succès dans les plateformes industrielles en raison de leur plus grande facilité de mise en œuvre.

Plus récemment, les réseaux sur puce et leur parallélisme intrinsèque ont ouvert de nouvelles problématiques, et de nouvelles perspectives quant à l'optimisation des mécanismes de synchronisation. En effet, ces *media* de communication fortement parallèles ont permis l'augmentation du nombre de cœurs des MPSoC, ainsi que l'apparition de nouveautés architecturales, et notamment le principe de grouper les éléments du système en grappes (*cluster*). Ces spécificités des MPSoC modernes ont permis un renouveau des travaux d'optimisation des synchronisations. L'état de l'art dressé présente quelques propositions d'optimisation reposant sur les caractéristiques particulières des MPSoC communicant via des réseaux sur puce, du point de vue matériel, comme logiciel. Ces solutions comportant avantages et inconvénients, nous pensons que les architectures actuelles des MPSoC peuvent encore offrir des possibilités d'optimisation non exploitées.

Par ailleurs, notre étude bibliographique, nous a permis de prendre conscience que peu d'articles exposent de manière détaillée et quantifiée les problèmes rencontrés par les mécanismes de synchronisation dans les MPSoC implémentant des réseaux sur puce. Afin de pouvoir confirmer ou infirmer leurs hypothèses de travail au sein un environnement réaliste, il nous a semblé important de commencer notre étude par la caractérisation précise des problèmes apparaissant lors de l'établissement des synchronisations dans des cas d'utilisation réels. Le méthode de caractérisation mise en place sera l'objet du chapitre 5.

Chapitre 4

Une chaine de mesure précise et adaptée pour la détection des problèmes de synchronisation

Sommaire

Lorsque nous projetons d'étudier les performances d'un service logiciel, il est important de définir précisément ce que nous souhaitons observer et comment nous pouvons obtenir ces informations. Ainsi, nous reviendrons dans ce chapitre sur la notion d'observabilité des mécanismes de synchronisations ainsi que sur les limites des outils de mesure existants et leur non applicabilité à notre problématique, pour enfin proposer une solution alternative reposant sur la technique de l'émulation. Enfin, nous présenterons une chaine de mesure développée dans l'objectif de répondre à nos exigences d'observabilité.

4.1 Évaluation des mécanismes de synchronisation

L'optimisation d'un mécanisme mettant en œuvre du logiciel (et en particulier les mécanismes de synchronisations) passe par l'étude préalable des faiblesses de ces mécanismes afin de pouvoir apporter les solutions appropriées. Dans cette section, nous tenterons de définir les informations nécessaires à l'étude des performances du mécanisme, ainsi que les moyens permettant de les obtenir.

4.1.1 Étude réaliste des performances d'un mécanisme logiciel

Pour mener une étude la plus complète et la plus précise possible, certaines exigences quant à la méthodologie de mesure doivent être spécifiées.

Premièrement, comme nous l'avons évoqué préalablement en section 2.2, les mécanismes de synchronisation se situent à la frontière entre le logiciel et le matériel. Aussi, il est utile d'envisager une étude englobant ces deux aspects.

En ce qui concerne le matériel, le phénomène de contention (réseau + mémoire) est la principale cause de détérioration des performances d'une application logicielle. Sachant cela, une étude complète des performances d'un processus logiciel doit être en mesure d'évaluer le taux de contention matérielle généré par le mécanisme étudié.

Pour ce qui est du logiciel, il est important de pouvoir réaliser des mesures temporelles précises de manière non intrusive. Disposer d'un grain de mesure fin permet d'avoir une image précise du déroulement du processus. Afin de répondre à cette attente, l'obtention d'un arbre d'appels aux fonctions, avec le temps passé dans chacune d'entre elles, peut représenter un atout précieux. Ce type d'information est doublement utile :

1. L'arbre d'appel dévoile le flot d'exécution du mécanisme. En effet, identifier le flot d'exécution d'un processus logiciel n'est pas trivial lorsque nous considérons un mécanisme faisant appel au noyau Linux composé de plus de 23 000 fichiers C.
2. L'annotation de chaque appel de fonction avec le nombre de cycles passés dans la fonction permet de repérer les sections de code les plus chronophages, afin de focaliser notre travail d'optimisation sur celles-ci.

Cependant, cela ne suffit pas pour mener à bien une étude complète. En effet, notre objectif est le traitement d'applications et de mécanismes s'exécutant sur des MPSoC, et mettant en œuvre différents cœurs de traitement. Cela implique a fortiori que de la coopération entre cœurs intervient. Aussi, il est important de pouvoir visualiser et quantifier des événements transversaux à différents cœurs de calcul.

4.1.2 Quels outils de mesures ?

Comme énoncé dans le chapitre 2, à l'heure actuelle les deux techniques utilisées pour l'étude temporelle de systèmes logiciels sur MPSoC sont : l'instrumentation du code et l'extraction des informations temporelles par canaux cachés.

L'instrumentation du code

Cette technique consiste en l'ajout d'instructions spécifiques destinées à extraire des données temporelles sur l'application en cours d'exécution. L'avantage de cette technique est qu'elle peut être réalisée directement sur le MPSoC ciblé, ce qui permet une exécution rapide. Cependant, son inconvénient majeur est qu'elle introduit des instructions supplémentaires dans le code initial, ce qui modifie le flot d'exécution nominal de l'application, et par cela, peut modifier ou supprimer des sources de ralentissement.

Ainsi apparait la question de la granularité des mesures. En effet, une faible granularité serait très intrusive, alors qu'une grosse granularité ne permettrait pas d'avoir des données suffisamment précises pour l'étude des sources de ralentissements.

Les canaux cachés

L'obtention d'information sur le flot d'exécution du logiciel par canaux cachés est réalisée grâce à l'observation du support matériel exécutant le logiciel. Ainsi, le déroulement

du logiciel n'est aucunement affecté. Les mesures sont alors réalisées de manière transparente d'un point de vue logicielle, d'où l'appellation « canaux cachés ».

La principale technique permettant d'obtenir ce type de mesures consiste en l'utilisation de simulateurs. En effet, la majorité des simulateurs permettent d'extraire au cours de la simulation certaines données issues de la plateforme matérielle simulée. Parmi les données disponibles : le compteur ordinal (*program counter*) des CPU ainsi que le numéro de cycle d'horloge associé à chaque valeur de ce compteur. Ces deux informations permettent alors de reconstruire le flot d'exécution du logiciel, ainsi que les temps passés dans chaque section de code. Les informations temporelles sont alors obtenues de manière non intrusive avec un grain de mesure très fin (au niveau de l'instruction).

Néanmoins, la réalisation de ce type de mesures temporelles implique de simuler une plateforme avec une précision au cycle près. Or, les simulations au cycle près se révèlent excessivement lentes, et ne permettent pas la mise en place d'un environnement réaliste (système d'exploitation, applications utilisateurs, ...).

Une autre manière d'observer le matériel afin d'obtenir des informations sur le déroulement du logiciel est d'utiliser des modules dédiés au debug insérés dans certains processeurs industriels récents, tels que ceux proposés par les processeurs ARM [deb01]. Un exemple d'exploitation de ces modules est le travail présenté par Zheng *et al.* [ZPG+16] qui propose une solution permettant de tracer la pile d'appels aux fonctions logicielles, de manière non intrusive, à l'aide des données récoltées via le module de debug des processeurs ARM.

À ce stade, il est utile de noter que la vitesse d'exécution est affectée par l'extraction des données vers un PC hôte. En effet, les auteurs de [ZPG+16] évoquent une augmentation du temps d'exécution de 74x comparée à une version sans surveillance. Néanmoins, ce ralentissement reste acceptable, et permet d'exécuter un environnement de fonctionnement réaliste. L'inconvénient majeur de cette technique réside dans sa granularité de mesure qui ne permet pas de mener une étude temporelle très précise.

Ainsi, aucune des techniques actuellement utilisées pour la caractérisation temporelle ne correspond à nos attentes de mesures précises d'un mécanisme logiciel dans un environnement de fonctionnement réaliste. Pour cette raison, nous avons développé notre propre outil de mesure exploitant le principe de l'émulation.

4.2 L'émulation, une alternative aux limites de la simulation

Afin de contourner les limitations temporelles rencontrées par la simulation précise, des plateformes d'émulation ont vu le jour. Dans cette section, nous reviendrons brièvement sur le principe de l'émulation et sur son utilisation habituelle.

4.2.1 Notions d'émulation

L'émulation consiste en l'utilisation d'un support matériel reconfigurable capable d'exécuter le modèle d'un système ou bloc matériel. Dans les faits, les émulateurs sont composés d'un ensemble de FPGA (*Field Programmable Gate Array*) configurés à partir de la description RTL (*Register Transfert Level*) de la plateforme matérielle à émuler.

L'utilisation d'un support matériel de ce type permet l'accélération du temps d'exécution comparativement à l'approche classique mettant en œuvre un simulateur.

Cependant, le fait de ne plus exécuter le modèle de notre MPSoC directement sur un ordinateur (simulateur) mais de l'exécuter sur un support matériel indépendant rajoute une couche de complexité quant à l'observation du comportement du modèle au cours de l'émulation. Bien que le support matériel intègre quelques services permettant l'observation du système émulé, ceux-ci ne permettent généralement pas d'observer le système avec la même granularité qu'en simulation (à l'instruction près), ou alors au détriment des performances. En effet, la récupération des informations extraites du système émulé requiert la synchronisation entre le support matériel (FPGA) et la partie logicielle de gestion de l'émulation (ordinateur hôte). Ces échanges d'information impliquent la suspension de l'émulation jusqu'à ce que la synchronisation soit établie, ce qui nuit à la rapidité de l'émulation. Aussi, si l'envoi d'informations est récurent, les performances de l'émulation chutent, et le temps d'exécution du système se rapproche alors de celui de la simulation. Il ne devient alors plus possible d'envisager l'émulation d'un environnement réaliste en un temps raisonnable.

L'utilisation de l'émulation soulève donc à nouveau la question de la granularité des mesures temporelles. Nous présenterons la manière dont nous avons traité cette problématique dans la section 4.3.

4.2.2 L'émulation, un outil de vérification éprouvé

Le principe de l'émulation n'est pas nouveau, au point que la majorité des sociétés d'aide à la conception des circuits électroniques proposent des plateformes d'émulation : Mentor Graphics[vel], Synopsys[syn], Cadence[cad],...

Dans la plupart des cas, l'émulation est utilisée pour la validation fonctionnelle de systèmes dont la complexité rend leur simulation difficile (notamment en raison du temps de simulation requis) comme cela est illustré dans [AS15] et [GJSS16] par exemple. Les émulateurs proposés par les sociétés d'aide à la conception sont très puissants, mais aussi très chers. En raison de cela, des projets de conception d'émulateurs, moins puissants, mais aussi moins chers ont vu le jour à l'instar de celui présenté par Nava *et al.*[NBT+05] conçu pour la validation fonctionnelle des composants *ST Microelectronics*.

Bien que les plateformes d'émulation étaient historiquement utilisées pour la validation fonctionnelle, ces dernières années les sociétés d'aide à la conception commencent à étendre leurs offres de services, et proposent désormais des services d'aide au debug logiciel. Un bon exemple de ce type de service est l'outil *Codelink* proposé par Mentor Graphics[cod]. Cet outil offre des services se rapprochant de ceux proposés par *GDB* à la différence près que dans le cas de *Codelink*, les informations concernant le logiciel sont post-traitées à partir des traces générées par la plateforme d'émulation. En effet, après une phase de configuration complexe de la plateforme, un utilisateur peut émuler un MPSoC tout en récupérant durant l'émulation des traces des signaux des CPU. Une fois l'émulation terminée, les traces peuvent alors être traitées par *codelink* qui reconstruit, à partir de celles-ci, le flot d'exécution logiciel. L'utilisateur peut alors rejouer l'exécution du logiciel pas à pas tout en observant les registres des CPU ainsi que les piles d'appels aux fonctions.

Codelink ainsi que les autres outils du même genre sont très puissants et répondent en partie à nos problématiques d'observabilité des mécanismes logiciels. Cependant, le fonctionnement de ces outils nécessite le stockage de l'ensemble des valeurs des signaux relatifs aux CPU, ce qui représente une quantité de données importantes dès lors que le

mécanisme logiciel à observer est complexe. Même si certains émulateurs, comme celui de *Mentor Graphics*, offrent des mécanismes permettant la sauvegarde rapide des données tracées (tampon matériel de stockage dans le cas de *Mentor*), en raison de la quantité importante de données à stocker, ces mécanismes ne permettent pas la sauvegarde à grande vitesse de l'ensemble des données nécessaires à une étude complexe. L'émulateur doit alors se synchroniser à l'ordinateur hôte afin de lui transférer au fur et à mesure les données enregistrées. Ces synchronisations, durant un temps non négligeable, ralentissent l'émulation. Par ailleurs, ces outils permettent uniquement l'observation du flot logiciel de manière indépendante sur chaque CPU, et ne permettent pas de réaliser d'études corrélées entre le logiciel et le matériel, ni d'observer des événements se produisant entre différents CPU.

N'ayant pas trouvé d'outil de mesures de performances répondant à nos attentes en terme d'observabilité, nous avons décidé de concevoir notre propre chaine de mesures.

4.3 Une chaine de mesures non intrusive pour l'identification des ralentissements des mécanismes de synchronisation

Afin de pouvoir identifier les problèmes existants dans les mécanismes de synchronisation ainsi que leurs origines selon les critères d'observabilité que nous avons définis, nous avons conçu notre propre chaine de mesure précise et non intrusive reposant sur l'émulation.

Une des caractéristiques importantes de notre chaine de mesure est son minimalisme. En effet, nous avons fait le choix de sauvegarder un nombre restreint de données, configurable en fonction du type d'analyse que nous désirons mener (cf section 4.3.3). Grâce à cette spécificité, le nombre de synchronisations nécessaires entre l'émulateur proprement dit et le PC hôte est réduit, ce qui permet de maintenir une vitesse d'émulation élevée.

Dans cette section nous présenterons d'abord l'architecture générale de notre chaine de mesure. Ensuite, nous discuterons des choix d'implémentation mis en œuvre. Enfin, nous présenterons brièvement la méthodologie permettant son utilisation.

4.3.1 Une architecture en deux parties

Notre chaine de mesure se compose de deux parties représentées sur la figure 4.1 : une partie « acquisition » en charge de l'extraction des données durant l'émulation (à gauche sur le schéma) ; une partie « traitement » (à droite sur le schéma) responsable du traitement des valeurs extraites par la partie « acquisition ». L'échange entre les deux parties repose sur le principe de co-émulation permettant de mettre en place une communication entre la plateforme d'émulation et un PC hôte pilotant l'émulateur.

Acquisition des données

La partie d'acquisition des données permet d'extraire du MPSoC, en cours d'émulation, les valeurs des signaux utiles à notre analyse. Une fois récupérées, ces données sont directement envoyées vers le PC hôte. En raison de la nature des données que nous souhaitons analyser, il n'est pas envisageable d'extraire l'ensemble des signaux du système émulé, et cela principalement pour deux raisons :

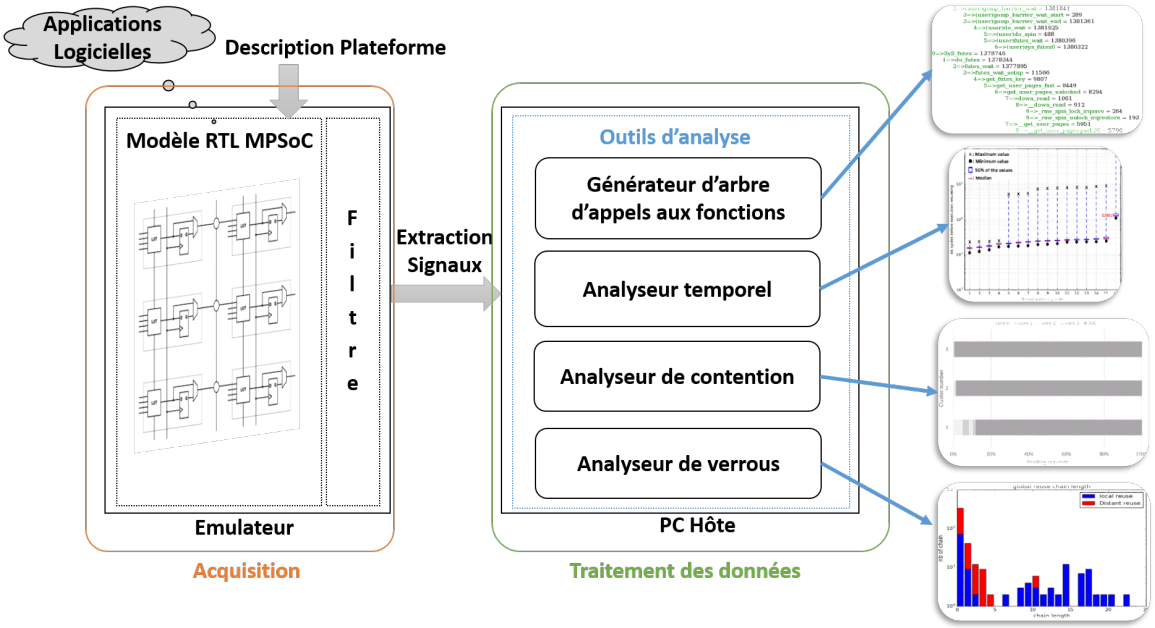

FIGURE 4.1 – Architecture de notre chaine de mesure non intrusive pour MPSoC

— la quantité des données extraites. En effet, nous souhaitons évaluer des mécanismes dans un environnement réaliste composé d'un système d'exploitation et d'applications utilisateur complexes. Dans ce contexte, l'extraction de l'ensemble des signaux représenterait un nombre de données très important, dont le traitement et l'enregistrement seraient problématique.

— la vitesse d'émulation. Comme évoqué précédemment (section 4.2.1), le transfert de données de la plateforme d'émulation vers le PC hôte nécessite une synchronisation entre les deux parties impliquant la suspension de l'émulation pendant ce laps de temps.

Ainsi, nous avons décidé de filtrer les données que nous transmettons au PC hôte grâce à l'ajout d'une fonction de filtrage représentée sur la figure 4.1. Cette dernière permet d'envoyer uniquement les données nécessaires pour le type d'évaluation que nous souhaitons mener. Par exemple, si nous voulons élaborer un arbre d'appels aux fonctions, nous avons uniquement besoin des signaux suivants :

— Le compteur ordinal de chaque CPU afin d'être informé du déroulement du programme

— La valeur du registre d'instruction afin de pouvoir déterminer le type d'instruction exécuté (typiquement un saut (*jump*) vers une fonction)

— Le nombre de cycles d'horloge

Le filtrage nous permet alors de limiter l'envoi des données à ces quelques signaux, ce qui réduit largement le nombre d'information à transférer. Par ailleurs, cette fonction permet aussi de mettre en place un filtrage plus complexe, autorisant l'envoi des données uniquement lorsque que le compteur ordinal atteint une adresse prédéfinie, ou que la valeur du registre d'instruction est égale à une certaine valeur par example. Ainsi, nous pouvons imaginer envoyer des données uniquement durant l'exécution d'une fonction bien déterminée (compteur ordinal ayant atteint l'adresse de début de la fonction), et d'envoyer seulement les appels de fonctions explicites (registre d'instruction égal aux codes d'opération —*opcodes*— des instructions de saut avec liaison : « jal » et « jalr » du MIPS32).

Grâce à cette stratégie, nous pouvons palier les deux points soulevés précédemment concernant le nombre de données transmises. Par ailleurs, le fait de réduire le nombre de données enregistrées présente un autre avantage, qui est celui du passage à l'échelle. En effet, lorsque nous voulons proposer une étude complète d'un mécanisme, l'ensemble des signaux utiles doit être extrait. Or, dans une architecture MPSoC, cela signifie que les données de l'ensemble des cœurs, mémoires,... de toutes les grappes doivent être récupérées. Nous comprenons donc que si aucun effort n'est mené pour réduire le nombre de données extraites, la quantité de données peut vite exploser et rendre impossible le support d'architectures composées de nombreux CPU.

Traitement des données

Le traitement des données est réalisé par un ensemble d'outils logiciels conçus pour analyser les données envoyées par la partie « acquisition », afin d'en tirer des informations utiles. Ces outils s'exécutent sur le PC hôte. Ils permettent de calculer et de présenter les comportements et les sources de ralentissement des mécanismes étudiés. Dans le cadre de notre étude des mécanismes de synchronisation, nous avons principalement développé quatre outils complémentaires représentés sur la figure 4.1.

Le générateur d'arbre d'appels aux fonctions

Cet outil permet de tracer la pile d'appels aux fonctions logicielles effectués au cours de l'exécution du mécanisme étudié, et d'annoter chaque fonction avec le nombre de cycles passés dans celle-ci. L'outil génère un fichier de sortie par CPU. La figure 4.2 illustre une pile d'appels générée par l'outil. Chaque ligne se compose des informations suivantes :

— Un préambule contenant un numéro représentant la profondeur de la pile d'appels. Nous avons choisi d'écrire explicitement cette profondeur afin de faciliter la lecture.

— (à droite de la flèche) Le nom de la fonction. Dans certains cas ce nom est précédé de la mention « (user) » indiquant qu'il s'agit d'une fonction de l'espace utilisateur de Linux.

— (en fin de ligne) Le nombre de cycles passés dans la fonction.

Par ailleurs, une surcouche de l'outil permet de comparer hiérarchiquement deux à deux des arbres appels.

L'analyseur temporel

Cet outil réalise des analyses temporelles, visant à mettre en exergue les sources de ralentissement dans les mécanismes de synchronisation. Le principe de celui-ci est d'effectuer des corrélations entre différents événements s'étant produits lors de l'émulation afin d'exposer les délais passés dans les différentes phases du mécanisme étudié. Dans le cas d'une barrière de synchronisation cet outil permettra par exemple de calculer les délais des phases de relâchement des processus, ou encore les délais de réveil des processus.

Une des forces de cet outil est aussi sa capacité à calculer des résultats moyennés sur plusieurs itérations. En effet, dans un environnement matériel/logiciel complexe, de nombreuses sources peuvent venir parasiter les mesures :

— matérielles : défauts de caches, prédictions de branchements, ...

— logicielles : ordonnancement des tâches, interruptions, ...

Aussi, il est important de pouvoir s'affranchir des mesures singulières en réalisant plusieurs mesures du même phénomène.

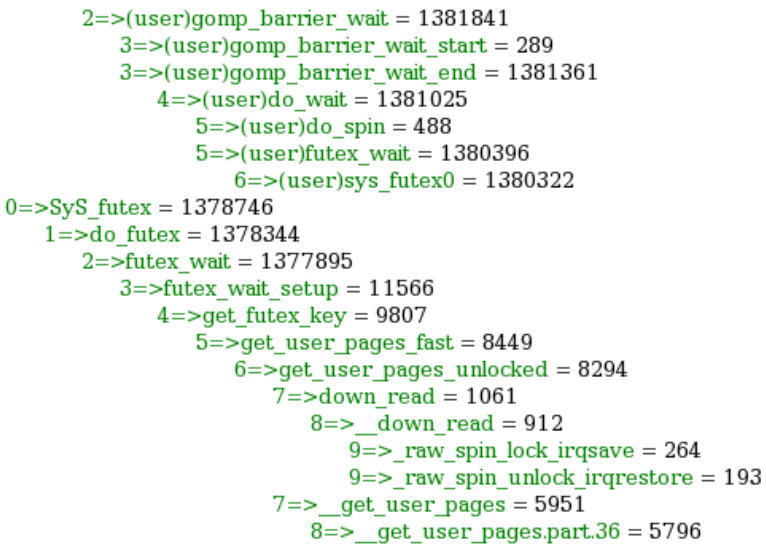

FIGURE 4.2 – Exemple d'un arbre d'appels aux fonctions généré par notre chaine de mesure

L'analyseur de contention

Cet outil vise à fournir des informations sur la contention réseau apparaissant lors de l'exécution du mécanisme étudié. Il présente des informations quantifiées sur le taux de charge aux points clés du système de communication entre éléments du MPSoC (routeurs, interfaces réseau,...).

L'analyseur de verrous

Cet outil permet de tracer l'utilisation des verrous de synchronisation au cours de l'exécution d'une application complexe. Il permet notamment de connaitre le déplacement des verrous au cours du temps en fonction du processus ayant acquis l'autorisation d'accès. Nous verrons dans le chapitre 6 comment ces informations nous ont permis de concevoir une solution de verrous innovants.

4.3.2 Choix d'implémentation

La section précédente présente l'architecture générale de notre chaine de mesure. Dans cette section, nous revenons un peu plus en détails sur les choix stratégiques de son implémentation.

La figure 4.3 reprend les grandes lignes de l'architecture présentées dans la figure 4.1 en ajoutant les détails de nos choix d'implémentation.

Nous retrouvons sur la figure 4.3 les deux phases constituant notre chaine de mesure : la phase d'acquisition et la phase de traitement des données. La première phase est dite *on-line*, c'est à dire qu'elle est réalisée de manière concomitante avec l'émulation. La seconde phase quant à elle est dite *off-line*, ce qui signifie qu'il s'agit d'un post-traitement des données, effectué une fois l'émulation terminée. La communication entre ces deux phases est réalisée au moyen de fichiers de *logs*. La phase d'acquisition écrit les données extraites dans ces fichiers, tandis que la phase de traitement vient par la suite lire les données contenues dans ces fichiers afin de les traiter.

Ce choix d'implémentation résulte du fait que l'analyse temporelle requiert la connaissance des événements présents et passés, donc l'enregistrement de l'historique des événements est nécessaire.

Par ailleurs, un de nos objectif lors de la conception de cette chaine de mesure est

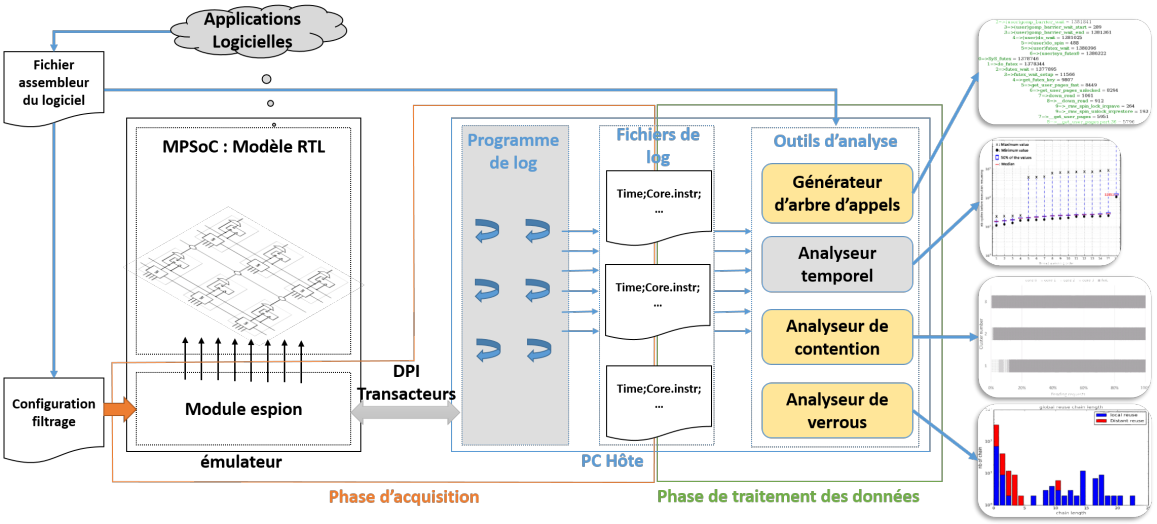

FIGURE 4.3 – Diagramme des choix d'implémentation de notre chaine de mesure non intrusive pour MPSoC

de maintenir le temps d'émulation le plus réduit possible. Or, comme énoncé précédemment, le principe de la co-émulation, sur lequel se base notre chaine de mesure, implique que l'émulation est suspendue durant toute la période pendant laquelle la partie logicielle (sur le PC hôte) a la main. Ainsi, le fait de retarder le traitement des données permet de réduire le temps passé dans la partie logicielle, et de ce fait de ralentir le moins possible l'émulation.

Un autre avantage à réaliser le traitement en différé, à partir des fichiers de *logs*, est de permettre d'effectuer plusieurs traitements différents sur le même ensemble de données, sans avoir besoin de relancer l'émulation.

Phase d'acquisition de données

Comme illustré sur la figure 4.3, la phase d'acquisition des données est divisée entre la plateforme d'émulation à proprement parler, et le PC hôte.

Sur l'émulateur, l'acquisition des données est réalisée par un module appelé *module espion*. Celui-ci, écrit en langage *SystemVerilog*, est connecté au modèle du MPSoC. Il permet d'extraire des signaux du système émulé à différents niveaux de la hiérarchie du modèle. Pour ce faire, ce module définit des « pointeurs » vers les signaux ciblés à partir des chemins complets vers ceux-ci. La variation des valeurs des signaux peut alors être monitorée grâce à ces pointeurs. Les valeurs de ces signaux sont ensuite envoyées vers le PC hôte grâce à l'interface de co-émulation *Direct Programming Interface* (DPI) et au principe de transacteurs. Les transacteurs sont un service de la plateforme d'émulation permettant l'envoi et la réception de données entre le support matériel d'émulation et le PC hôte pilotant l'émulation (cf [Viv13]). Grâce à ce service, les transferts de données entre ces deux parties peuvent être réalisés de manière efficace.

En plus de cette fonction d'extraction de signaux, le *module espion* est aussi en charge de la fonction de filtrage évoquée à la section précédente. En effet, celui-ci envoie des données vers le PC hôte via les transacteurs DPI uniquement lorsque certaines préconditions sont remplies. Un exemple simple de filtre est d'envoyer le compteur ordinal annoté du numéro de cycle d'horloge uniquement lorsque la valeur de ce compteur ordinal a évolué, et non pas à chaque cycle, cela dans le but d'éviter les envois redondants

lorsque que le compteur ordinal est bloqué. Un exemple de filtre un peu plus complexe est d'envoyer le compteur ordinal accompagné du numéro de cycle uniquement lorsque celui-ci atteint des adresses prédéfinies. Pour un filtrage de ce type, les valeurs des conditions de filtrage sont amenées à varier en fonction du programme logiciel exécuté. Aussi, nous avons choisi de définir ces valeurs de filtrage dans un fichier texte externe, noté *configuration filtrage* sur la figure 4.3. Grâce à ce fichier de configuration externe, il est possible de modifier les valeurs des conditions de filtrage sans avoir besoin de synthétiser à nouveau l'ensemble de la plateforme.

En ce qui concerne la partie sur le PC hôte, nous avons conçu un logiciel en langage C++ capable de récupérer les données transmises via les transacteurs DPI, et de les écrire dans les fichiers de logs : soit dans un format ASCII compact, soit dans un format binaire ultra-compact permettant ainsi de réduire l'impact mémoire. Sachant que l'émulation est interrompue tant que les fonctions DPI s'exécutent sur le PC hôte, nous avons conçu ce logiciel afin de passer le moins de temps possible dans ces fonctions. Pour cela, nous lisons en parallèle les données provenant des transacteurs grâce à l'instantiation de plusieurs processus (threads) dédiés à cette tâche. De plus, les données lues à partir des transacteurs ne sont pas directement écrites dans les fichiers de logs, mais sont stockées temporairement dans des mémoires tampons. Cela permet de relâcher la main plus rapidement car l'écriture dans des fichiers peut durer un temps non négligeable. L'écriture dans les fichiers de logs est alors réalisée par d'autres processus indépendants des fonctions DPI.

Ce *module espion* permet donc l'acquisition d'informations temporelles sur l'exécution du logiciel par canaux cachés. En effet, s'agissant d'un greffon matériel d'espionnage, le flot d'exécution logiciel n'est nullement impacté par ses mesures. Bien qu'il soit possible que la fréquence d'horloge de la plateforme dépende en partie de celui-ci, représentant environ 10% des LUT de la plateforme pour un MPSoC TSAR 16 cœurs, le nombre de cycles nécessaires aux opérations du programme logiciel ne sera pas affecté, du fait que, ni l'architecture du MPSoC, ni le code source du programme n'aient été modifiés. Aussi, les mesures resteront précises, sans distorsion du flot d'exécution initial du logiciel.

Phase de traitement de données

La phase de traitement des données est réalisée *off-line*, une fois les données écrites dans les fichiers de logs. Les différents outils composant cette phase vont alors lire ces données et les traiter afin d'en tirer des informations utiles pour l'analyse des performances des mécanismes de synchronisation.

Générateur de pile d'appels aux fonctions

Il s'agit d'un script en langage python conçu afin d'extraire les instructions de saut (*jump and link*) et de retours (*return*) des fichiers de logs. En effet, ces instructions permettent de savoir lorsqu'un appel à une fonction est réalisé (« jal » et « jalr ») et lorsque nous sortons d'une fonction (« jr »). Ainsi, en utilisant la valeur du compteur ordinal et celle du registre d'instruction annotées avec le numéro de cycle, il devient alors possible de connaitre avec précision le temps (nombre de cycles) passé dans chaque fonction. De plus, en croisant les adresses du compteur ordinal avec les adresses des fonctions contenues dans les fichiers désassemblés du logiciel, nous sommes en mesure de retrouver le nom de chaque fonction. À partir de ces informations, nous pouvons alors tracer les

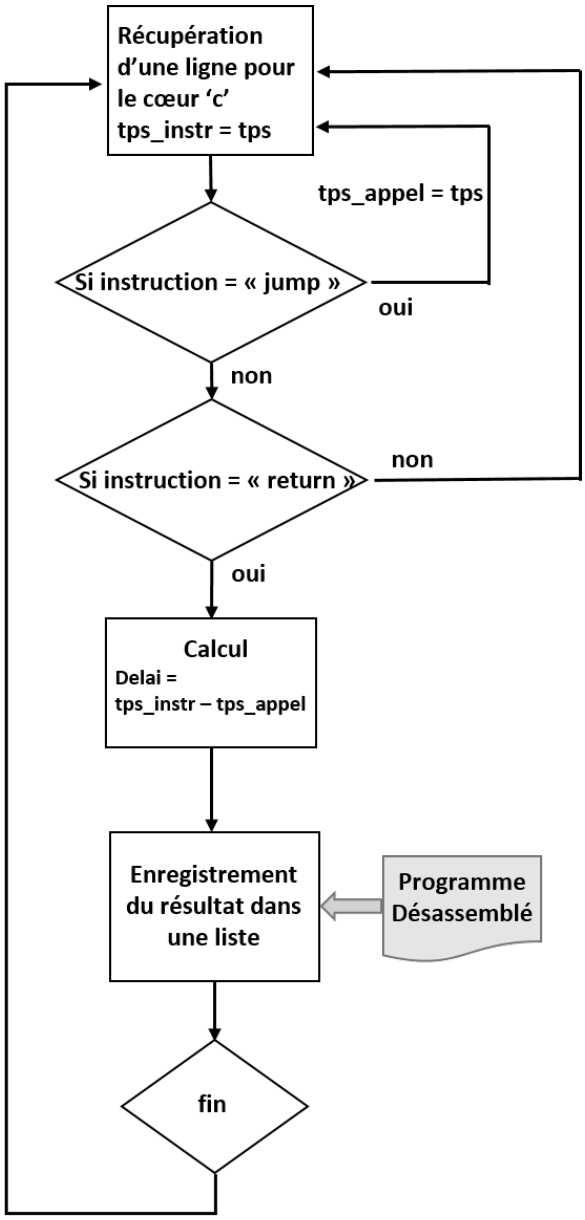

FIGURE 4.4 – Étapes de l'algorithme de construction de l'arbre d'appel aux fonctions

arbres d'appels aux fonctions comme celui illustré figure 4.2.

L'algorithme mis en œuvre pour tracer l'arbre d'appels est illustré sur la figure 4.4. Afin d'en simplifier la représentation, cette figure présente le flot de traitement de données relatives à un CPU, sachant que chaque CPU est analysé de manière indépendante. La première étape est la récupération d'une ligne de données à partir du fichier de *logs* d'entrée. Si la ligne correspond au cœur en cours de traitement, le numéro de cycle de la ligne est stocké en mémoire et nous passons à l'étape suivante. Sinon nous sautons la ligne puis lisons la ligne suivante, et cela jusqu'à atteindre une ligne correspondante au cœur ciblé. Le seconde étape permet de vérifier le type d'instruction exécutée : si le code d'opération de l'instruction courante correspond à une instruction d'appel de fonction (« jal » ou « jalr »), nous sauvons le temps de l'instruction en tant que temps d'appel, puis nous rappelons la fonction principale de l'algorithme de manière récursive. S'il s'agit d'une instruction de retour de fonction (*return*, « jr »), nous passons à l'étape suivante en charge du calcul du délai passé dans la fonction. Le nom de la fonction associé à une adresse est ensuite déduit à partir du fichier désassemblé du logiciel, puis le résultat est

stocké dans une liste temporaire. Cette liste est ensuite inversée lors de l'écriture dans le fichier de sortie afin d'obtenir un arbre d'appels correspondant à l'ordre des appels aux fonctions. Cet algorithme est alors exécuté jusqu'à ce que la dernière ligne du fichier de *logs* soit traitée.

Il s'agit en réalité du principe de base du traitement. Dans la pratique, des sauts impromptus résultants des routines de traitement des interruptions viennent interrompre le flot linéaire appel-retour de fonctions, et fausser l'estimation du temps nécessaire à leur exécution. Nous avons alors dû mettre en place un traitement particulier pour prendre en compte le traitement des interruptions (IRQ). Les étapes supplémentaires dédiées au traitement des interruptions sont représentées en orange sur le figure 4.5. Le processus de traitement est le même que celui exposé précédemment à l'exception de trois spécificités :

— Si l'instruction courante n'est pas reconnue comme étant un saut à une fonction (« jal » ou « jalr ») ou un retour de fonction (« jr »), un test supplémentaire est réalisé avant l'appel récursif à la fonction principale. Si l'instruction exécutée est un saut vers une routine d'interruption, nous l'enregistrons en tant que tel, et nous sauvons aussi le niveau de profondeur actuelle de la pile d'appels, c'est à dire la profondeur à laquelle a eu lieu l'interruption (IRQ).

— Lorsqu'une instruction de retour est rencontrée, si nous étions dans une routine d'IRQ, alors le délai de l'IRQ est calculé en lieu et place du calcul nominal de délai. Le résultat est ensuite stocké en mémoire. Par ailleurs, la trace de cette routine d'interruption est ajoutée à liste d'appels aux fonctions en vue de l'écrire dans le fichier de sortie.

— Lorsque que le délai d'une fonction régulière vient d'être calculé, le niveau de profondeur d'appel de cette fonction est comparé au niveau auquel est apparu l'IRQ : si l'IRQ est intervenu pendant l'exécution de la fonction (profondeur de l'IRQ inférieur à la profondeur courante), le temps passé dans l'IRQ est alors soustrait au temps passé dans la fonction.

La figure 4.6 illustre un exemple pratique du processus de génération de la pile d'appels de fonctions. En haut à gauche est représenté le fichier d'entrée (fichier de *logs*) produit par la phase d'acquisition. Ce fichier contient les valeurs du compteur ordinal (PC), le code d'opération de l'instruction exécutée (codeOP), le registre « rs » qui contient l'adresse vers laquelle l'instruction *jump* (« jalr ») saute, et enfin le numéro de cycle d'horloge d'occurrence de cette instruction (temps). À gauche de ce fichier, nous avons représenté, le flot d'exécution du programme en pseudo-code afin d'en faciliter la compréhension. En haut à droite de la figure, nous avons le fichier désassemblé du logiciel. Ce fichier contient l'adresse mémoire de chaque instruction du programme. Au milieu de la figure se trouve une représentation symbolique de la mémoire et des variables nécessaires aux traitements des données. En bas de la figure, nous avons le fichier de sortie généré par l'outil.

Comme cela est représenté sur la figure, la première étape est la lecture d'une ligne dans le ficher d'entrée. Dans notre cas, une instruction d'appel (*jump*) peut être identifiée grâce au code d'opération de l'instruction. L'adresse de la fonction appelée *fctA* est contenue par le registre « rs ». Cette adresse est alors stockée dans la mémoire avec le numéro du cycle associé.

Ensuite, les lignes du fichier d'entrée sont lues jusqu'à ce que nous rencontrions l'adresse de la première instruction de la routine de traitement d'interruption. Nous stockons alors cette information avec son numéro de cycle dans la mémoire.

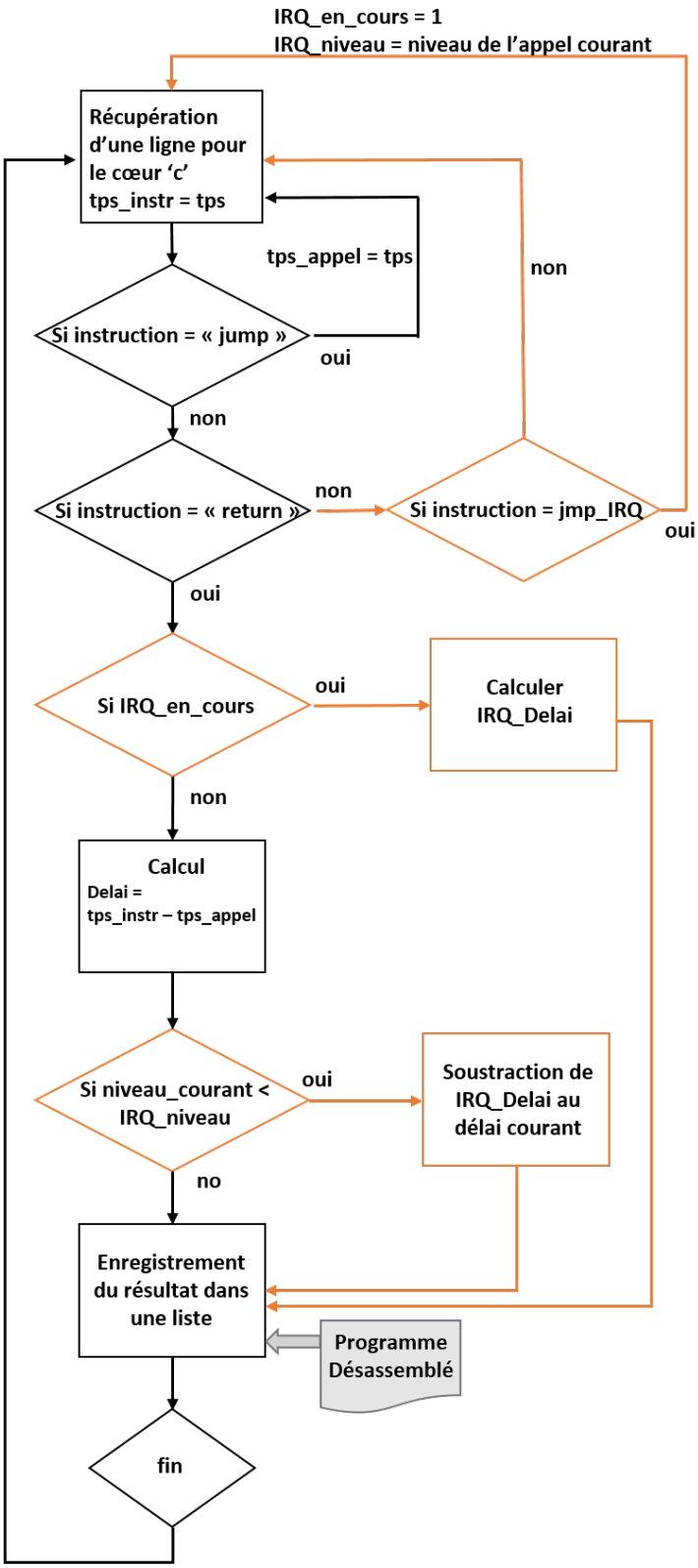

FIGURE 4.5 – Étapes de l'algorithme de construction de l'arbre d'appel aux fonctions avec la gestion des IRQ

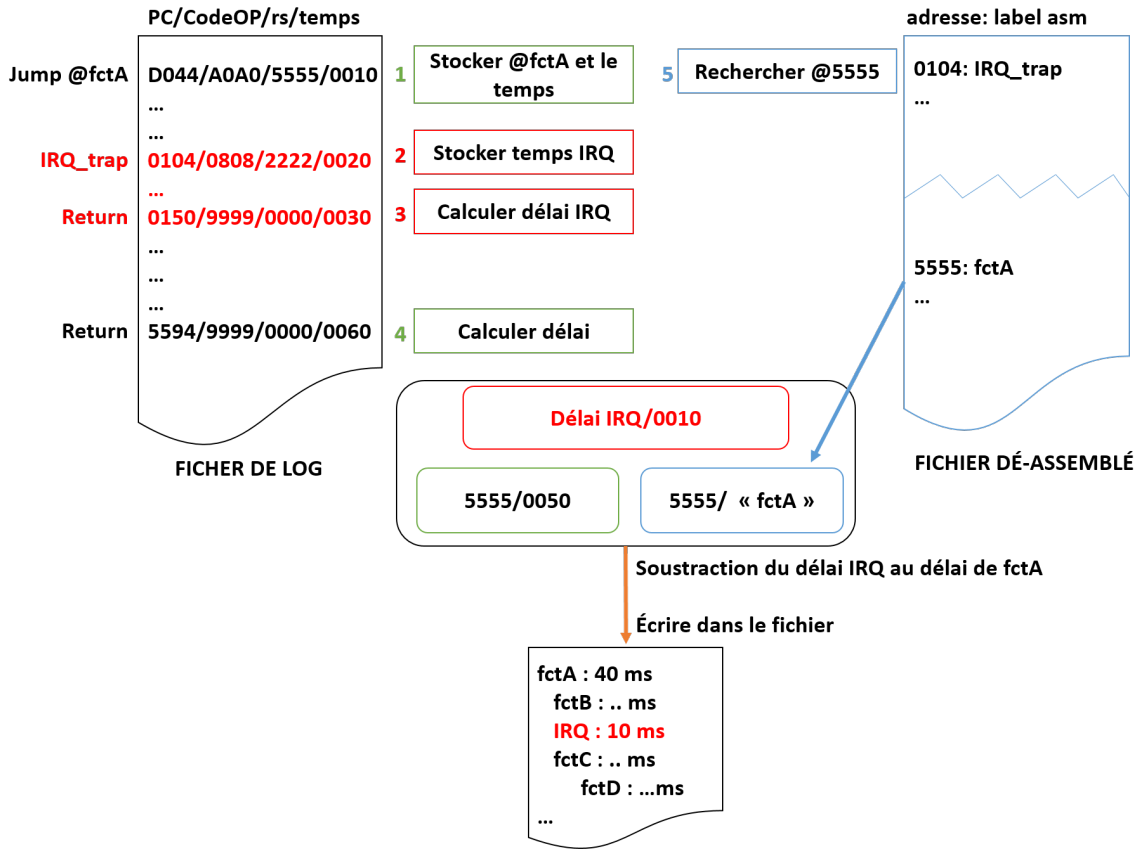

FIGURE 4.6 – Exemple de génération de l'arbre d'appel aux fonctions

Après cela, les lignes du fichier d'entrée sont lues jusqu'à ce que nous trouvions l'instruction de retour de la routine d'interruption. Nous calculons alors la durée de cette interruption, et stockons le résultat en mémoire.

Ensuite, les lignes du fichier d'entrée sont lues jusqu'à ce que nous lisions la ligne contenant l'instruction de retour de la fonction *fctA*.

C'est alors que nous déduisons, à partir du fichier désassemblé, le nom de la fonction *fctA* grâce à son adresse. Une fois le nom retrouvé, nous calculons le temps total passé dans cette fonction (dans notre exemple 50 cycles). Cependant, étant donnée qu'une interruption s'est produit pendant l'exécution de cette fonction, nous soustrayons la durée de traitement de l'interruption au temps passé dans la fonction *fctA*. Enfin, nous écrivons le résultat final dans le fichier de sortie.

Analyseur temporel

Dans notre chaine de mesure, l'analyse temporelle des mécanismes de synchronisation est réalisée au moyen d'un programme en langage C++. Le principe général de cet outil est de calculer les délais entre les adresses d'instructions spécifiques définies dans le fichier de filtrage, et cela quel que soit le cœur exécutant l'instruction. Ensuite, les délais calculés sont associés aux processus ayant réalisé l'opération, ainsi qu'au mécanisme de synchronisation en jeu. Ces résultats sont alors présentés à l'utilisateur qui, à partir de ceux-ci, peut connaitre la durée d'événements particuliers relatifs à un mécanisme de synchronisation (temps de relâchement,...).

Afin de pouvoir mener une étude inter-CPU, nous avons besoin de connaitre les identifiants des processus légers (*threads*) s'exécutant sur les différents CPU. Par ailleurs, nous avons aussi besoin de pouvoir identifier les mécanismes de synchronisation utilisés par

le logiciel afin de mesurer les délais relatifs à chacun d'entre eux.

D'un point de vue logiciel, l'obtention de ces informations est assez directe, cependant notre étude non-intrusive nous impose d'obtenir ces informations à travers le support matériel, ce qui se révèle beaucoup moins trivial. En effet, les identifiants des processus ne sont pas directement stockés dans des registres matériels que nous pourrions observer. Ainsi, afin de contourner ce problème, nous avons trouvé qu'un processus peut être identifié grâce à ses pointeurs de pile et de tas qui sont différents pour chaque processus. Ces pointeurs étant stockés dans des registres matériels, nous avons alors étendu notre *module espion* (phase d'acquisition) afin d'extraire leurs valeurs.

En ce qui concerne l'identification des mécanismes de synchronisation, étant donné que tous les mécanismes de synchronisation de notre plateforme reposent sur le principe de verrou, nous utilisons l'adresse de ce verrou comme identifiant du mécanisme. Afin de retrouver cette valeur, nous lisons l'adresse sur laquelle le mécanisme réalise son opération atomique. Cette adresse correspond alors à l'adresse du verrou. L'exécution de l'instruction atomique est retrouvée grâce à son code d'opération contenu dans le registre d'instruction. Pour ce qui est de l'adresse ciblée, elle est contenue dans un registre du CPU que nous avons ajouté à la liste des registres extraits par le *module espion*.

Analyseur de contention

Afin de pouvoir quantifier la contention réseau, nous avons besoin d'avoir des informations sur les requêtes envoyées à la mémoire, et cela à différents points du réseau reliant les cœurs de calcul aux mémoires. Pour ce faire, le *module espion* est étendu afin d'extraire les signaux des canaux de communication réseau au niveau des routeurs et des interfaces réseau. L'analyseur de contention est alors un script python permettant de suivre une requête mémoire sur le chemin vers la mémoire, et de calculer le temps passé dans chaque section réseau à partir des informations contenues dans les fichiers de *logs*. Grâce à cela, nous pouvons identifier les endroits où des pertes de temps sont observées, ainsi que leurs origines : contention réseau, mémoire, ... Le suivi des trames de requêtes mémoire est réalisé grâce aux identifiants de trames contenus dans leur en-tête.

Analyseur de verrou

Cet outil est un script python permettant d'analyser le comportement des verrous à partir de leur identifiant (adresse du verrou), ainsi que du cœur ayant réalisé la requête. Grâce à cela, nous pouvons connaitre le modèle d'attribution des verrous aux différents CPU du MPSoC, en fonction des processus ayant obtenus les droit d'accès. L'intérêt de ce type d'information sera discuté dans le chapitre 6.

En plus de cela, l'observation des requêtes mémoire de demande de verrous permet de déterminer l'endroit de stockage physique de la variable mémoire associée à un verrou, et par conséquent de savoir si les cœurs réalisent des accès à des mémoires proches ou distantes.

D'autres informations concernant l'utilisation des verrous telles que le temps d'utilisation, ou alors le nombre de demandes en attente, peuvent aussi être obtenues.

4.3.3 Utilisation de la chaine de mesure

Les étapes nécessaires pour l'utilisation de notre chaine de mesure peuvent être résumées de la manière suivante :

1. Définir le mécanisme que nous souhaitons étudier

2. Choisir le type d'étude que nous désirons mener, en fonction du mécanisme ciblé, et de notre niveau de connaissance de celui-ci. Par exemple, si nous voulons avoir une vue générale du flot d'exécution du mécanisme, nous nous orienterons plutôt vers l'étude de la pile d'appels aux fonctions, alors que si nous souhaitons mesurer des délais inter-CPU particuliers nous utiliserons plutôt l'analyseur de délais.
3. Déterminer les signaux à extraire du système en fonction du type d'étude ciblée.
4. Configurer le *module espion* et la fonction de filtrage en fonction des signaux préalablement définis.
5. Émuler le système exécutant le logiciel à étudier. L'acquisition des données est réalisée pendant cette étape.
6. Exécuter l'outil de traitement de données désiré.
7. Analyser les résultats.

Il est important de noter que les étapes 3 et 4 concernant la détermination des signaux à extraire, ainsi que la configuration du *module espion* requièrent une connaissance certaine du MPSoC sur lequel s'exécute le mécanisme. En effet, la configuration du *module espion* nécessite de créer des « pointeurs » vers les signaux désirés. Pour cela, les chemins exacts des signaux dans la hiérarchie du MPSoC émulé doivent être indiqués depuis l'interface de haut niveau jusqu'au nom précis des signaux dans le(s) module(s) final(aux) (CPU par exemple). Conscient de la difficulté que représente ce besoin d'expertise pour un utilisateur, la facilitation de cette étape de configuration est une des perspectives d'amélioration de la chaine de mesure, comme mentionné ci-après.

Les chapitres suivants présenteront plus en détails les configurations et les résultats obtenus grâce à cette chaine de mesure.

4.4 Conclusion et perspectives

Après avoir clairement défini les contraintes d'observabilité des mécanismes de synchronisation à respecter afin de pourvoir mener une étude complète et précise révélant les sources de ralentissements de ce type de mécanismes dans un environnement réaliste, nous avons vu que les techniques utilisées habituellement pour observer les caractéristiques temporelles d'un logiciel MPSoC ne correspondaient pas à nos attentes. Pour cette raison, nous avons conçu une chaine de mesure en adéquation avec nos exigences, reposant sur l'émulation.

Composée de deux phases, cette chaine de mesure permet de mener des études de performance d'un logiciel englobant des problématiques matérielles et logicielles, et cela de manière non intrusive. Un ensemble d'outils de traitement permet de générer des sorties de différents types mettant en évidence les sources des ralentissements dans les mécanismes de synchronisation.

Parmi nos perspectives d'amélioration de cette chaine de mesure figure l'assistance à la configuration. En effet, cet outil est certes très configurable, mais nécessite une certaine connaissance du MPSoC ciblé et de la chaine de mesure afin de configurer cette dernière en fonction de l'étude désirée. Nous souhaiterions alors développer une surcouche logicielle permettant de faciliter, et d'automatiser un peu plus la configuration de la chaine de mesure. Par ailleurs, nous pourrions aussi envisager d'étendre cette chaine de mesure à d'autres mécanismes logiciels de base afin de proposer un outil plus complet permettant de localiser les faiblesses d'un logiciel quelconque.

Chapitre 5

Méthodologie d'optimisations du logiciel au service des mécanimes de synchronisation

Sommaire

L'identification précise des problèmes et des sources de ralentissements des mécanismes de synchronisation est l'étape préalable à leur optimisation. Dans le chapitre précédent, nous avons décrit la chaine de mesure précise et non intrusive conçue dans cette optique. Dans ce chapitre, nous reviendrons sur la méthodologie mise en place pour l'optimisation des mécanismes de synchronisation. Nous présenterons ensuite deux optimisations sur les mécanismes de barrière de synchronisation de la bibliothèque OpenMP réalisées grâce à l'application de cette méthodologie.

5.1 L'optimisation de la pile logicielle, un compromis efficace

Nous avons pu voir à travers l'état de l'art dressé chapitre 3 qu'il existe deux stratégies d'optimisation des mécanismes de synchronisation : l'optimisation logicielle et l'optimisation matérielle. Nous avons aussi vu que les solutions logicielles sont généralement plus facile à utiliser que les propositions matérielles.

Une des problématiques ciblées par cette thèse est de fournir un mécanisme de synchronisation permettant d'accélérer les applications utilisateurs (à haut niveau), tout en

restant simple à mettre en œuvre. Aussi, nous avons opté pour l'étude précise de la pile logicielle utilisée actuellement dans les applications parallèles afin de détecter la présence éventuelle de sources de ralentissements qui n'auraient pas encore été solutionnées.

5.1.1 Parallélisation d'un programme logiciel

En règle générale, la parallélisation d'un programme logiciel peut être réalisée de deux manières :

— la première méthode consiste en la création et la gestion des processus (et notamment les problématiques de synchronisation entre processus) « à la main » en s'appuyant sur des bibliothèques relativement « bas niveau » tel que *Pthread* (*Posix*).

— la seconde méthode, quant à elle, consiste en l'utilisation de bibliothèques « haut niveau » aidant à la parallélisation du code. Ces dernières gèrent de manière semi-automatique de la création des processus et des synchronisations à partir de directives haut niveau insérées par le programmeur dans son code.

En ce qui concerne les bibliothèques d'aide à la parallélisation le standard *de facto* est OpenMP, qui gère les problématiques relatives au code parallèle à partir de directives du pré-processeur. Les directives sont interprétées par le compilateur lors des phases de compilation. Le compilateur étend alors le code initial, et le transforme en un code parallèle : processus, synchronisation,...

En raison de la complexité induite par la parallélisation d'un programme « à la main » avec les bibliothèques bas niveau, les bibliothèques d'aide à la parallélisation sont de plus en plus utilisées, avec le standard OpenMP en tête de cet essor.

Nous avons alors décidé d'étudier les mécanismes de synchronisation de ce standard, et plus particulièrement l'implémentation GNU de cette bibliothèque. Il existe en effet de nombreuses implémentations du standard OpenMP : GNU, Intel, LLVM,... ; nous avons choisi l'implémentation GNU en raison de sa grande utilisation dans des systèmes réels, ainsi que de son aspect « logiciel libre » (*open source*).

Un exemple d'utilisation de la bibliothèque GNU OpenMP

Dans cette section, nous nous proposons de présenter le principe général de fonctionnement de la bibliothèque GNU OpenMP à travers un exemple de programme simple 5.1.

Listing 5.1 – Exemple de code C parallélisé avec OpenMP

```
1  #include <omp.h>
2  #define TAB_SIZE 1000

3  int main (void){
4    unsigned int n=0;
5    unsigned int sinTable[TAB_SIZE];
6    omp_set_num_threads(16);

7    #pragma omp parallel for shared (sinTable)
8    for(n=0; n<TAB_SIZE; n++)
9      sinTable[n] = n*2;

10   print_table(sinTable);
11   return 1;}
```

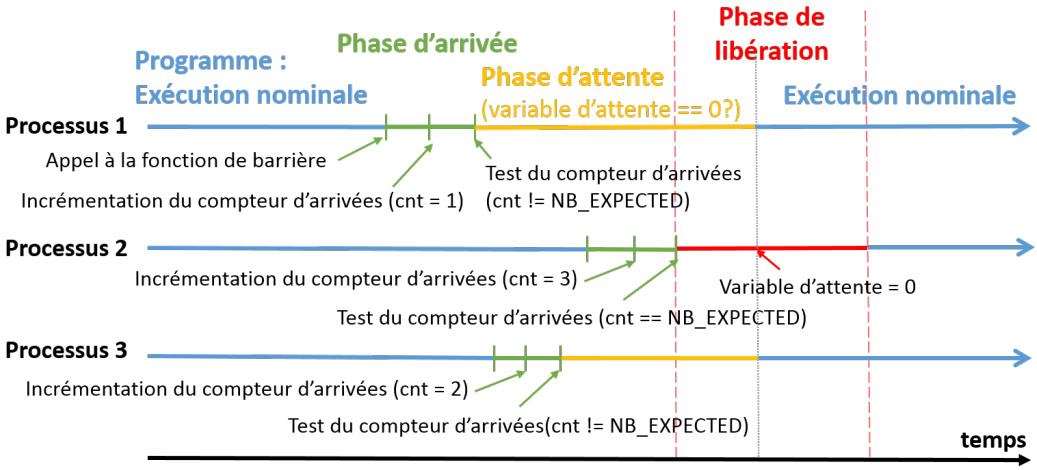

FIGURE 5.1 – Chronogramme de l'exécution d'une barrière de synchronisation compteur central pour 3 processus

Le but de ce programme est de calculer un tableau d'éléments (ligne 9) puis de l'afficher (ligne 10). La calcul de ce tableau est parallélisé, et est effectué sur 16 processus grâce aux directives OpenMP ligne 6 et 7. C'est au moment de la compilation que ces directives sont interprétées par le compilateur. Le compilateur ajoute alors au code initial les instructions nécessaires à la création, et la gestion des processus. Dans notre exemple, la boucle *for* est remplacée par la création des 16 processus légers (*threads*). Chaque processus est en charge du calcul d'un seizième du tableau. Par ailleurs, la bibliothèque OpenMP va aussi insérer une barrière de synchronisation à la fin de la section parallèle (entre les lignes 9 et 10). Cette barrière de synchronisation est nécessaire afin de garantir la terminaison du calcul de l'ensemble des processus avant de passer à l'étape suivante. La section parallèle s'achève avec la fin de la boucle *for*. L'affichage du tableau (ligne 10) est alors réalisé par un seul et unique processus (le processus principal).

À travers cet exemple simple, nous remarquons que la barrière de synchronisation est le mécanisme de synchronisation au cœur des performances de la bibliothèque OpenMP du fait qu'une barrière est ajoutée par défaut à la fin de toutes les sections parallèles.

Au vu de cela, nous avons choisi d'orienter notre recherche d'optimisation des mécanismes de synchronisation de la bibliothèque OpenMP vers le service de barrière.

La barrière de synchronisation GNU OpenMP

Lors de la présentation de l'état de l'art (chapitre 3), nous avons vu qu'il existe plusieurs algorithmes permettant d'implémenter le service de barrière de synchronisation. La bibliothèque GNU OpenMP, comme de nombreuses bibliothèques de synchronisation standard, a opté pour un algorithme relativement simple : le compteur central.

Le principe de fonctionnement de cet algorithme est illustré sur la figure 5.1 qui représente le déroulement de 3 processus synchronisés par une barrière. L'idée de base de cet algorithme est d'utiliser une variable centrale (notée *cnt* sur la figure) comptant le nombre de processus ayant atteint la barrière. Cette variable partagée par l'ensemble des processus est protégée par un mécanisme d'accès exclusif (typ. un verrou). Lorsqu'un processus fait appel à la fonction de barrière, il entre dans ce que l'on a nommé la **phase d'arrivée**. Durant cette phase, le processus signale au système de barrière son arrivée en incrémentant le compteur partagé. Ensuite, il va tester la valeur de ce compteur afin de savoir si l'ensemble des processus attendus est arrivé sur la barrière ou non. Si le nombre de

processus attendus n'est pas atteint, le processus passe dans sa **phase d'attente**, pendant laquelle il va attendre l'arrivée des autres processus. Ce cas est illustré par les processus 1 et 3 sur la figure 5.1. Sinon, si le nombre de processus attendus est atteint, le dernier processus a atteindre la barrière (dans notre cas le processus 2) s'aperçoit qu'il est le dernier arrivé, grâce au test du compteur. Il débute alors la **phase de libération** durant laquelle il est en charge de libérer l'ensemble des processus en attente, avant de reprendre son flot d'exécution nominale. Une fois sortis de leur phase d'attente, les autres processus peuvent reprendre leur exécution nominale.

À ce stade, il est utile de rappeler que les délais introduits par les mécanismes de synchronisation sont de deux sortes :

— Les délais dépendant de la parallélisation de l'application en elle même, qui sont la plupart du temps dus à un mauvais équilibre de charge entre les processus. Si on applique ce type de délai au cas des barrières de synchronisation, il s'agit du temps pendant lequel les processus arrivés sur la barrière doivent attendre le processus le plus lent (dont la phase de calcul est la plus longue) avant de continuer leur exécution.

— Les délais intrinsèques aux mécanismes synchronisation. Ces délais résultent de l'implémentation du mécanisme de synchronisation à proprement parler.

Lorsque que nous parlons d'optimisation des synchronisations, nous visons le second type de délais. En effet, le premier type de délai est de la responsabilité du programmeur, il est donc hors de notre champ d'action potentiel.

La phase d'attente introduite par tous les mécanismes de synchronisation peut être réalisée de deux façons :

— L'**attente active** qui consiste à interroger périodiquement une variable partagée (*variable d'attente* sur la figure 5.1) en attendant qu'un autre processus (processus libérateur) change l'état de celle-ci, permettant ainsi la libération. Ce type d'attente est très efficace en termes de vitesse de reprise d'exécution, cependant l'interrogation périodique monopolise le processeur alors que celui-ci pourrait exécuter d'autres tâches plus utiles pendant ce laps de temps. De plus cette interrogation périodique augmente la consommation d'énergie.

— L'**attente passive** qui implique l'endormissement des processus en attente. Lorsqu'un processus débute sa phase d'attente, il va appeler une fonction spéciale qui lui permettra de s'endormir (i.e. rendre la tâche non planifiable par l'ordonnanceur). C'est alors le processus réalisant la fonction de libération qui le réveillera en temps voulu. Cette stratégie a l'avantage de libérer le processeur pour l'exécution d'autres tâches, cependant, le réveil d'un processus peut parfois être long, comme nous le verrons dans la suite de ce manuscrit.

La politique d'attente choisie par la bibliothèque GNU OpenMP est une combinaison des deux précédentes : un processus rentrant dans sa phase d'attente va commencer par une attente active, et cela jusqu'à ce qu'un crédit de temps prédéfini soit écoulé. Une fois ce temps écoulé, le processus bascule en attente passive, et s'endort jusqu'à sa libération, qui met fin à sa phase d'attente. Il s'agit là de la politique d'attente choisie par défaut par GNU OpenMP. Néanmoins des directives permettent de forcer un type d'attente particulier (active ou passive) en fonction des besoins utilisateur.

En ce qui concerne les phases d'arrivée et de libération, des sources de ralentissement peuvent potentiellement provenir de chacune d'entre elles. Aussi, une étude plus approfondie doit être réalisée pour ces phases. L'étude de ces phases suivant notre méthodologie d'analyse est présentée dans la section 5.3.

5.2 Méthodologie d'analyse et d'optimisation des mécanismes logiciels

Lors de l'étude de l'état de l'art (chapitre 3), nous avions mis en évidence une faiblesse récurrente de nombreux papiers quant aux difficultés rencontrées pour étudier les mécanismes de synchronisation dans leur environnement réel de fonctionnement. Cette observation, nous a amené à concevoir une chaine de mesures permettant d'évaluer les mécanismes en environnement réel (chapitre 4). Dans cette section, nous présentons la méthodologie d'analyse mise en place pour l'étude et l'optimisation des mécanismes de synchronisation. Les quatre étapes de cette méthodologie exploitant notre chaine de mesure sont les suivantes :

Étape 1 : Une fois le mécanisme ciblé déterminé, la première étapes consiste en l'étude des zones potentiellement critiques grâce à l'*analyseur temporel* de notre chaine de mesure.

Configuration Le *module espion* et le fichier de filtrage de la phase d'acquisition de la chaine de mesure doivent être configurés afin de délimiter la zone d'étude (fonction étudiée), ainsi que les événements particuliers que nous souhaitons mesurer (exécution de certaines instructions particulières). Pour cela, le fichier de filtrage doit être renseigné avec les adresses des points de mesures (instructions ou fonctions) auxquels nous souhaitons obtenir des informations temporelles. En ce qui concerne le *module espion*, il doit être configuré afin d'extraire les signaux voulus, et d'envoyer leurs valeurs lorsque les conditions d'envoi sont remplies (par exemple : lorsque le compteur ordinal atteint une adresse définie dans le fichier de filtrage).

Exécution Ensuite, dans un second temps, nous exécutons la plateforme de mesures mettant en place un environnement réel d'exécution (système d'exploitation, application utilisateur de test,...). Les valeurs des signaux de la plateforme sont extraites et écrites dans les fichiers de *logs* à cette occasion.

Traitement Une fois les fichiers de *logs* obtenus, nous pouvons procéder à leur traitement grâce aux outils de la chaine de mesure prévus à cet effet. En l'occurrence, nous utilisons pour cette étape l'*analyseur temporel*.

Analyse Enfin, l'analyse des résultats produits par l'outil peut être effectuée par l'utilisateur afin de déduire des informations pertinentes en vu d'optimiser le mécanisme.

Étape 2 : Une fois la ou les zone(s) critique(s) identifiée(s), l'étape suivante consiste en la localisation de la source du ralentissement. Pour ce faire, nous utilisons le *générateur*

d'arbre d'appels afin d'avoir une vision très précise des délais nécessaires à l'exécution de chaque fonction.

Afin de mener cette étude, les quatre sous-étapes du point précédent (configuration, exécution, traitement, analyse) doivent être à nouveau effectuées.

Étape 3 : Les deux étapes d'analyse précédentes permettent d'identifier avec précision la ou les sources de ralentissement du mécanisme dans un environnement de fonctionnement réaliste. À partir de ces informations, nous pouvons désormais proposer une solution d'optimisation adaptée.

Étape 4 : La dernière étape consiste en l'évaluation de la solution proposée. Pour cela, nous mesurons à nouveau les délais passés dans les sections critiques ainsi que la durée d'exécution de l'application afin d'évaluer l'apport par notre solution.

Une étape d'analyse supplémentaire peut être ajoutée en fonction des besoins. Il s'agit de l'étude comportementale du matériel au cours de l'exécution du mécanisme, et plus particulière de la contention réseau et mémoire qui peuvent impacter les performances du logiciel. S'agissant d'une étape d'analyse, les quatre phases d'étude (configuration, exécution, traitement, analyse) doivent être réalisées. L'étape de configuration a alors pour but d'étendre le champ d'acquisition des données au delà des signaux des CPU, afin de permettre l'extraction d'autres signaux de la plateforme matérielle, tels que les signaux d'interface des contrôleurs de mémoire ou des routeurs. Le traitement des données est réalisé grâce à l'*analyseur de contention*.

5.3 Cas d'application : la barrière de synchronisation de la bibliothèque GNU OpenMP

Cette section présente l'étude du mécanisme de barrière de synchronisation de la bibliothèque GNU OpenMP selon la méthodologie décrite précédemment (section 5.2).

5.3.1 Environnement d'expérimentation

L'étude du mécanisme de barrière GNU OpenMP a été réalisée sur la plateforme MPSoC TSAR décrit au chapitre 1. En ce qui concerne l'environnement matériel, l'émulateur utilisé comme élément de base de notre plateforme de mesure est un émulateur *Veloce2 Quattro*[Men18].

Pour ce qui est de la partie logicielle, nous avons pu profiter du portage sur la plateforme TSAR du noyau Linux 4.6 réalisé par le laboratoire d'informatique de Paris 6 (LIP6) et le CEA Leti. La version de GCC utilisée pour compiler le noyau Linux et les applications est la version 4.8.2. Il est utile de noter que la bibliothèque GNU OpenMP est partie intégrante du compilateur GCC, aussi leur version est identique.

Au vu de la relative ancienneté de la version de GCC utilisée, nous pouvons nous interroger sur la représentativité de mener une étude sur une bibliothèque OpenMP désuète. Cependant, la source de ralentissement détectée, et la solution proposée est toujours applicable à la version 7.2 de GCC, car aucune amélioration de cette fonction de la barrière

GNU OpenMP n'a été réalisée depuis.

Notre travail de recherche s'inscrit dans l'amélioration des performances des MPSoC. Aussi, lors de notre évaluation nous avons choisi d'affecter un processus (*thread*) par cœur de calcul. il s'agit d'une méthode couramment utilisée dans les applications parallèles lors de la recherche de performances élevées, afin de limiter les délais induits par la migration de tâches.

5.3.2 La phase d'arrivée, une fausse problématique

Nous avions conclu la section 5.1.1, qui présentait le principe de fonctionnement de la barrière de synchronisation implémentée par GNU OpenMP, sur le fait que deux phases de l'algorithme implémenté pouvaient être potentiellement problématiques : la phase d'*arrivée* et celle de *libération*.

En effet, la phase d'arrivée peut, en théorie, représenter un risque de ralentissement induit par la contention mémoire occasionnée par les arrivées concomitantes de plusieurs processus sur la barrière, qui auraient pour conséquence la multiplication des tentatives d'accès concurrents au compteur partagé. Dans ce cas, les accès au compteur doivent être sérialisés, ce qui génère un délai.

Cependant, comme cela a été avancé par Wei *et al.* [WLSY15], ce cas ne se produit que dans des cas extrêmement rares. En effet, les processus n'arrivent en réalité (presque) jamais au même instant en raison de légers déséquilibres entre processus dus aux interférences du système : accès mémoire, défauts de cache, gestion des entrées/sorties,...

Afin de nous assurer de la validité de cette hypothèse de travail, nous avons décidé de mesurer l'intervalle de temps s'écoulant entre deux arrivées sur la barrière.

Pour cela, nous exécutons le code simple présenté précédemment (programme 5.1), sur notre plateforme d'évaluation pour un système composé de 16 cœurs (16 processus affectés aux 16 cœurs). En effet, ce programme simple a l'avantage de permettre à OpenMP de bien équilibrer la charge de calcul entre les différentes tâches, ce qui représente le pire cas pour ce qui est de la contention lors de la phase d'arrivée. Afin de nous assurer de la reproductibilité des mesures effectuées, nous avons exécuté le micro-code 400 fois.

La figure 5.2 expose les intervalles de temps s'écoulant entre deux appels consécutifs à la fonction de barrière réalisés par des processus différents. L'axe des abscisses représente le nombre de cycles entre deux appels. L'axe des ordonnées représente le nombre d'occurrences. Si nous prenons l'intervalle de temps $60-80$ cycles par exemple, la figure informe que nous avons mesuré 16 délais entre deux appels à la fonction de barrière compris dans cet intervalle de temps.

Ainsi, nous pouvons voir sur cette figure que le nombre de fois pour lequel l'intervalle de temps était inférieur à 60 cycles est vraiment très faible, ce qui confirme, par la pratique, l'hypothèse sur le faible risque de contention pendant la phase d'arrivée.

Par ailleurs, le temps perdu lors de cette phase d'arrivée n'est pas forcement très problématique du point de vue des performances globales de l'application. En effet, dans la majorité des cas, les processus rentrent ensuite dans leur phase d'attente. Ainsi, les délais pouvant apparaitre lors de la phase d'arrivée sont dans de nombreux cas masqués par la phase d'attente suivante.

Sur la base de ces éléments, nous concluons sur le fait qu'en pratique la criticité de cette phase d'arrivée ne nécessite pas une attention particulière en vue de l'optimiser. Nous avons alors décidé de focaliser notre étude sur la phase de libération. Cependant,

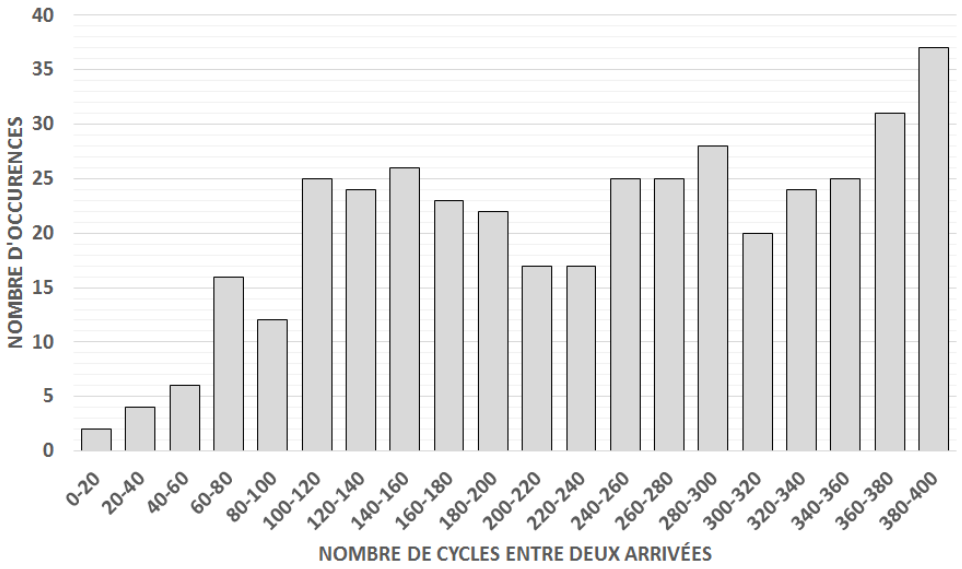

FIGURE 5.2 – Intervalles de temps entre deux appels consécutifs à la fonction barrière pour 16 cœurs

comme nous avons pu l'évoquer dans la section 5.1.1, la procédure de libération diffère en fonction des modes d'attente dans lesquels étaient les processus logiciels :

— l'attente active, qui nécessite uniquement l'affectation de la variable d'attente à l'état de libération ;

— l'attente passive, qui nécessite le réveil des processus endormis.

Pour cette raison, la suite de notre analyse de la barrière de synchronisation OpenMP est divisée en deux études distincts en fonction du type d'attente.

5.3.3 Optimisation de l'attente active

Cette sous-section présente l'étude, ainsi qu'une proposition d'optimisation, de la phase de libération dans le cas d'une attente active.

Évaluation temporelle de la phase de libération

En adéquation avec la méthodologie d'étude mise en place (section 5.2), la première étape de l'étude de la phase de libération est l'évaluation du temps passée dans cette section critique.

Pour cela, nous avons à nouveau exécuté le micro-code 5.1 sur une plateforme TSAR à 16 cœurs, puis moyenné les résultats des 400 exécutions afin d'avoir des résultats reproductibles.

Le but de cette évaluation est de mesurer le temps entre le début de la phase de libération (initiée par le dernier processus arrivé sur la barrière), et le moment où les processus quittent la fonction de barrière pour reprendre leur activité nominale. Pour ce faire, nous utilisons l'*analyseur temporel*.

La figure 5.3 montre les résultats obtenus. L'axe des ordonnées représente le nombre de cycles entre l'instant ou le dernier processus prend conscience qu'il est le dernier et débute la phase de libération, et le moment où un processus reprend son exécution nominale. L'axe des abscisses représente les différents processus dans l'ordre de libération. À savoir que la première colonne par exemple représente le premier processus à avoir repris

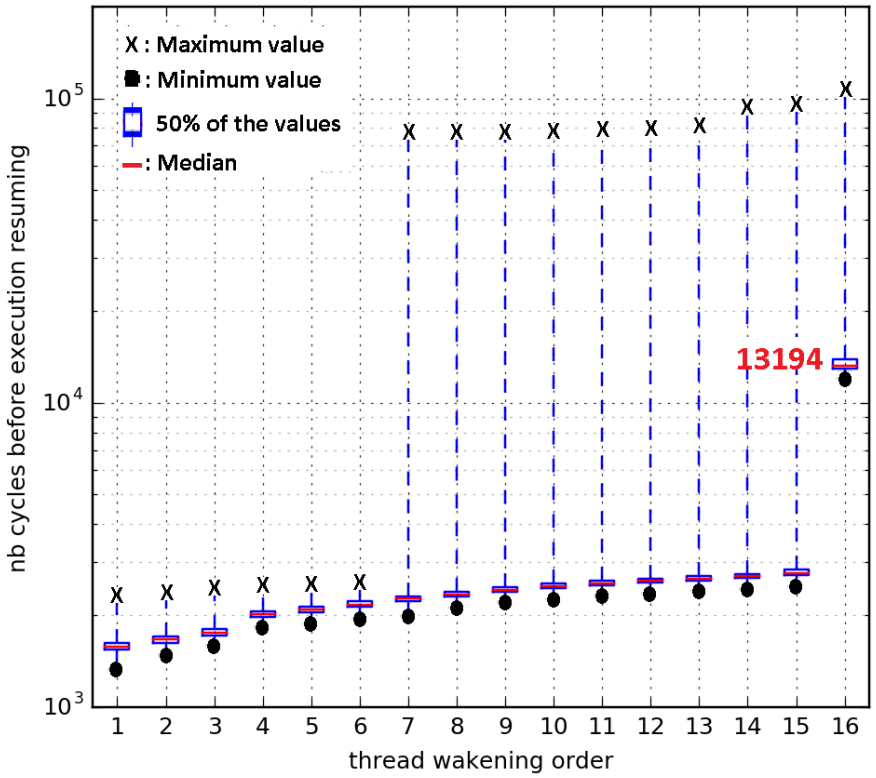

FIGURE 5.3 – Durée de la phase de libération pour 16 processus attachés à 16 cœurs pour 400 appels à la barrière.

son exécution quel que soit son identifiant. La figure représente des *boites à moustaches* dont la ligne rouge est la médiane et le rectangle bleu contient 50% des valeurs. La valeur minimale est représentée par un point, et la valeur maximale par un 'X'.

Nous pouvons voir sur cette figure que les boites bleues sont bien regroupées autour des valeurs médianes, cependant la valeur maximale peut parfois être bien plus élevée. Ces valeurs singulières sont dues à la gestion des routines d'interruption ou à l'ordonnanceur du système qui parasitent l'exécution des processus utilisateur, et génèrent des délais importants de manière occasionnelle. Par conséquent, étant donné que ces valeurs résultent de parasites, nous ne les prenons pas en considération lors de notre étude, et nous nous focalisons sur la valeur médiane et les boites bleues de chaque processus.

Nous pouvons noter sur cette figure que le temps nécessaire à la reprise (libération) de l'ensemble des processus est d'environ 13,200 cycles. Nous remarquons aussi qu'un processus est largement retardé par rapport aux autres. Afin de déterminer la cause de ce retard, nous passons à la second étape de notre méthode d'analyse, permettant la localisation des sources du ralentissement.

Localisation de la source du ralentissement

Afin de localiser l'origine de cet important retard d'un processus par rapport aux autres, nous avons décidé de tracer la pile d'appels aux fonctions grâce à notre chaine de mesure.

Cette pile d'appels aux fonctions nous permet d'analyser avec plus de précision le flot d'exécution au sein même de la phase de libération. C'est ainsi que nous avons remarqué que la bibliothèque OpenMP réalise de manière non dissociée un appel à la fonction en charge du réveil des processus, qu'un processus soit réellement endormi sur la barrière ou non.

En effet, la politique d'attente mise en œuvre par GNU OpenMP implique la possibilité d'avoir sur la même barrière certains processus en attente active alors que d'autre sont endormis, et cela en fonction du temps depuis lequel un processus attend. Dans ce cas, le processus libérateur doit libérer les processus en attente active en changeant l'état de la *variable d'attente*, ainsi que réveiller les processus endormis (attente passive). Cependant, lorsque la charge de travail est bien répartie entre les différents processus, ce qui est le cas la plupart du temps avec la politique d'équilibrage de charge d'OpenMP, la durée de la phase d'attente des processus est réduite. Ainsi aucun processus ne bascule en attente passive, et par conséquent aucun processus n'a besoin d'être réveillé. Notre micro-application 5.1 représente ce cas d'application pour lequel la charge est bien équilibrée. Ainsi, la phase d'attente des processus se limite uniquement à de l'attente active.

Par ailleurs, l'annotation temporelle de notre arbre d'appels aux fonctions nous apprend qu'environ 12900 cycles sont passés dans cette fonction de réveil (approximativement 97% du temps total de libération pour 16 processus) alors qu'aucun processus n'a besoin d'être réveillé dans les faits. Cette observation nous a donc mené à envisager une contre-mesure afin de réduire le temps total nécessaire à la libération des processus lorsque l'ensemble de ceux-ci réalise une attente active.

Proposition d'optimisation

À partir des observations précises sur le fonctionnement de la barrière de synchronisation GNU OpenMP, nous proposons de mémoriser le mode d'attente des différents processus en attente sur une barrière. Ainsi, si des processus sont effectivement endormis sur la barrière (attente passive), nous réalisons l'appel à la fonction de réveil, sinon nous sautons cet appel et sortons directement de la phase de libération.

Pour ce faire, nous avons implémenté une liste chainée contenant les « barrières à réveiller ». Dès qu'un processus bascule de l'attente active vers l'attente passive, l'identifiant de la barrière est ajouté à cette liste. Cet ajout est réalisé par le processus lui même avant de s'endormir.

Ensuite, lorsque le dernier processus débute la phase de libération, il recherche l'identifiant de la barrière courante dans la liste des « barrières à réveiller ». Si l'identifiant s'y trouve, cela signifie qu'au moins un processus est réellement endormi sur la barrière. Dans ce cas, il supprime l'identifiant de la liste puis appelle la fonction de réveil des processus. Sinon, si l'identifiant n'est pas trouvé dans la liste, alors aucun processus n'est endormi. L'appel à la fonction de réveil n'est donc pas réalisé.

Cette optimisation est minimale et très peu intrusive : sa longueur est de 50 lignes de code impactant 2 fichiers seulement.

Évaluation de la solution

Une fois la solution implémentée, nous pouvons passer à la dernière étape de la méthodologie d'optimisation : l'évaluation de la solution.

Il s'avère que sur notre plateforme à 16 cœurs, cette optimisation apporte un gain significatif. Le figure 5.4 expose les durées de libération des processus pour 16 processus s'exécutant sur 16 cœurs avec l'optimisation proposée. Le type de représentation choisi est le même que celui décrit précédemment pour la figure 5.3

Nous pouvons remarquer sur cette figure que le premier processus à reprendre son exécution est plus rapide que le premier de la version non optimisée (figure 5.3). En effet, le premier processus à reprendre son exécution est désormais celui réalisant la libération des processus. Dispensé de l'appel à la fonction de réveil, ce processus est maintenant

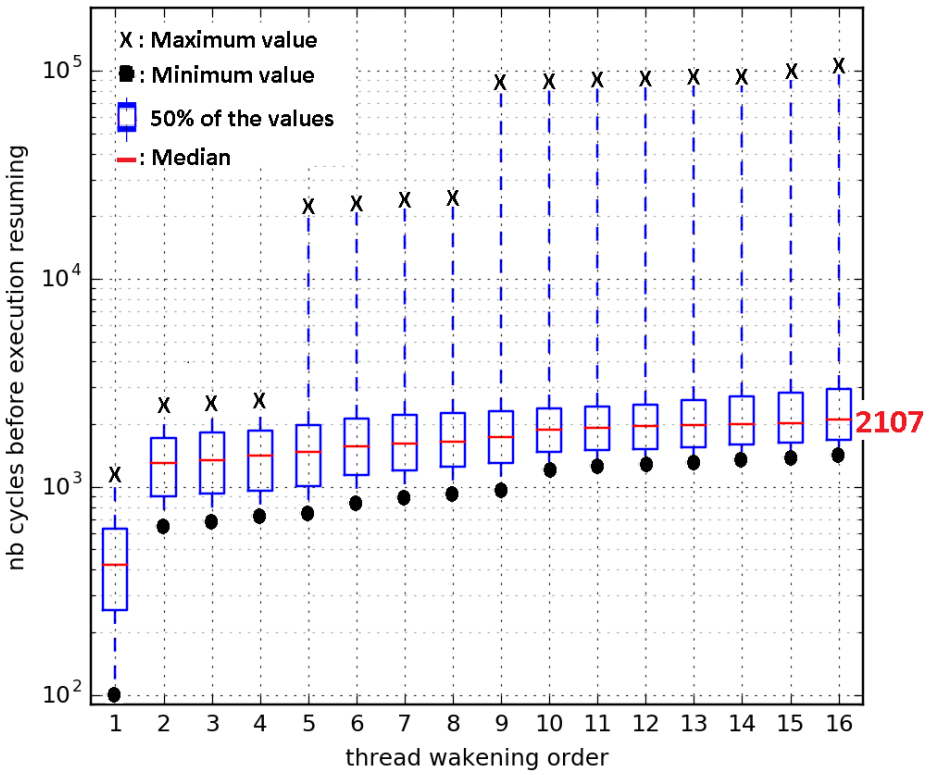

FIGURE 5.4 – Durée de la phase de libération pour 16 processus attachés à 16 cœurs pour 400 appels à la barrière avec l'optimisation proposée

en mesure de changer l'état de la variable d'attente puis de retourner à son exécution nominale avant tous les autres processus.

Par ailleurs, nous voyons sur la figure que l'ensemble des processus reprennent leur exécution légèrement plus rapidement qu'avec la version non optimisée. Cela est dû à un effet de bord de l'optimisation. En effet, la fonction de réveil des processus est une fonction complexe faisant beaucoup d'accès à la mémoire partagée. En supprimant l'exécution de cette fonction lorsqu'elle n'est pas nécessaire, de nombreux accès à la mémoire partagée sont supprimés par la même occasion, ce qui a pour effet de réduire la contention générale dans le système. Dès lors, chaque processus peut accéder un peu plus rapidement à la mémoire, permettant ainsi de réduire son temps d'exécution.

Enfin, la figure 5.4 expose le fait que la libération des 16 processus dure maintenant approximativement 2100 cycles (valeur médiane), à comparer avec les 13194 cycles précédents. L'optimisation proposée permet de gagner près de 84% du temps nécessaire à la phase de libération pour une plateforme de 16 cœurs.

Une fois l'évaluation de notre solution d'optimisation achevée pour une plateforme de 16 cœurs, nous avons répété cette évaluation pour des plateformes TSAR composées de 8 et 24 cœurs. Le tableau 5.1 présente les gains obtenus pour ces différentes plateformes.

Ces résultats montrent que lorsque la taille de la plateforme augmente, le gain diminue. En effet, mis à part les éventuels problèmes de contention mémoire, le temps passé dans la fonction de réveil lorsqu'aucun processus n'a besoin d'être réveillé est constant. Par contre, le temps nécessaire à la libération des processus augmente avec le nombre de processus. C'est pour cela que le gain relatif apporté par notre optimisation diminue lorsque que le nombre de processus augmente.

TABLEAU 5.1 – Gains obtenus grâce à l'optimisation de la phase de libération de la barrière de synchronisation GNU OpenMP pour différentes tailles de plateformes TSAR

Nombre de processus	Durée totale de la phase de libération sans l'optimisation	Durée totale de la phase de libération avec l'optimisation	Gain
8 sur 8 cœurs	11662 cycles (médiane)	1481 cycles (médiane)	87%
16 sur 16 cœurs	13194 cycles (médiane)	2107 cycles (médiane)	84%
24 sur 24 cœurs	14039 cycles (médiane)	7975 cycles (médiane)	43%

TABLEAU 5.2 – Résultats de l'optimisation de la bibliothèque GNU OpenMP sur l'application IS (classe « S ») du Benchmark NAS (20 exécutions avec l'optimisation, et 20 sans optimisation) pour 16 processus

	Durée totale des phases de libération	Durée totale d'exécution de l'application
IS with optimization	93×10^6 cycles	1069×10^6 cycles
IS without optimization	162×10^6 cycles	1228×10^6 cycles
gain with optimization	42.5%	12.9%

Afin d'évaluer le gain obtenu par notre optimisation sur une application réelle, nous avons exécuté l'application « Integer Sort » (classe « S ») du benchmark *NAS Parallel Benchmark*[nas].

Le temps total passé dans les phases de libération, ainsi que le temps total d'exécution de l'application, ont été mesurés grâce à notre chaine de mesure pour une plateforme de 16 cœurs. Afin d'obtenir des résultats les plus représentatifs possibles, nous avons exécuté 20 fois l'application avec et sans la solution proposée.

Les résultats obtenus sont présentés dans le tableau 5.2. Nous pouvons noter que le gain moyen pour la phase de libération est "seulement" de 42,5%. Nous expliquons ce gain, plus faible que celui mesuré pour notre micro-application, par la présence résiduelle de parasites du systèmes (ordonnanceur, interruption, ...) affectant les résultats, et cela en dépit du nombre d'exécutions.

Par ailleurs, nous pouvons noter un gain important sur le temps global d'exécution de l'application : 12,9%. Ce gain s'explique par le temps économisé dans les barrières de synchronisation elles-mêmes, mais aussi grâce à la réduction de la contention mémoire résultant de la suppression de la fonction de réveil, comme expliqué précédemment.

Grâce à l'étude précise selon la méthodologie proposée, nous avons pu concevoir une optimisation simple, ciblée, et efficace de la phase de libération des processus en attente active.

Les autres implémentations du standard OpenMP (LLVM, Intel, ...) n'étant malheureusement pas disponibles sur notre plateforme, nous n'avons pu comparer nos résultats avec celles-ci. Néanmoins, l'étude a posteriori du code source de la bibliothèque libre accès LLVM a révélé qu'une stratégie similaire à la nôtre, appelant la fonction de réveil uniquement lorsque cela est nécessaire, y est implémentée.

5.3.4 Optimisation de l'attente passive

Cette sous-section vise à l'optimisation de la phase de libération de la barrière GNU OpenMP à la suite d'une attente passive. Afin de bien différencier cette procédure de libération de celle traitée dans la sous-section précédente, nous nommons cette phase *phase de réveil*.

Avant de réfléchir à l'optimisation de la phase de réveil, quelques précisions quant à la procédure de réveil des processus de Linux sont nécessaires.

Dans le portage Linux que nous utilisons, la procédure de réveil repose sur l'envoi d'interruptions inter-processeur (Inter-Processor Interrupts, IPI). Lorsqu'un processus *A* doit réveiller un processus *B*, *A* déplace, au sein de la mémoire partagée du noyau, la structure représentant le processus *B* de la file des processus inactifs vers la file des processus actifs associée à un cœur de calcul désigné par le système d'exploitation pour exécuter ce processus. Ensuite, *A* envoie une *IPI* au cœur en charge du processus *B* afin de demander qu'une procédure de ordonnancement soit réalisée. Cette procédure recherche alors en mémoire partagée la liste des processus actifs qui lui est associée afin de planifier l'exécution du processus *B*.

Analyse de la phase de réveil

Connaissant déjà la phase critique que nous souhaitons étudier : la phase de réveil, nous avons choisi de passer directement à la seconde étape de notre méthodologie d'étude des mécanismes logiciels, à savoir l'étude détaillée du flot d'exécution du mécanisme.

Pour ce faire, une directive OpenMP permettant de forcer l'attente passive a été ajouté à la micro-application 5.1. Ensuite, nous exécutons cette micro-application sur la plateforme de mesure afin de tracer la pile d'appels aux fonctions de la phase de réveil.

L'analyse de cette pile d'appels met en avant un déroulement non attendu : La même fonction permettant l'envoi des IPI est appelée autant de fois qu'il y a de cœurs dans le système, et cela en dépit du fait que cette fonction soit codée de sorte à pouvoir prendre en paramètre une liste contenant les identifiants des processeurs destinataires des IPI. En effet, le noyau Linux construit une liste d'un élément seulement (l'identifiant du cœur), puis appelle la fonction d'envoi des IPI. Cette dernière récupère donc la liste qui lui est passé en paramètre, puis la parcourt pour finalement n'envoyer qu'une seule IPI. Ce procédé est ensuite répété $n-1$ fois, n étant le nombre de cœurs sur lesquels s'exécutent des processus à réveiller.

Proposition d'optimisation logicielle

L'observation précédente nous a amené à optimiser la procédure d'envoi des IPI. Notre proposition consiste à regrouper toutes les requêtes d'envoi d'IPI dans une seule liste, afin d'envoyer toutes les IPI en un seul appel à la fonction d'envoi, supprimant ainsi des gestions de listes chainées coûteuses.

Afin de mesurer l'impact de notre optimisation, nous mesurons la durée totale de la phase de réveil de la barrière, c'est à dire le temps entre l'atteinte de la barrière par le dernier processus et la reprise de l'exécution nominale de l'ensemble des processus. Afin d'assurer la reproductibilité des mesures, nous moyennons de nouveau les résultats sur 400 exécutions de la micro-application 5.1.

Au début de l'évaluation de la solution, les résultats pour une plateforme de 16 cœurs étaient encourageants (gain d'environ 25%). Cependant, l'optimisation c'est révélée inefficace pour une plateforme de 64 cœurs, n'apportant aucune réduction de la durée de la phase de réveil.

Au regard de l'optimisation, la suppression de gestion de listes chainées ne peut qu'améliorer les performances de la procédure. Cependant, dans les faits, le regroupement de l'envoi des IPI induit le rapprochement temporel de l'exécution des procédures d'ordonnancement des processus, qui dès lors se déroulent quasiment au même moment sur chaque cœur.

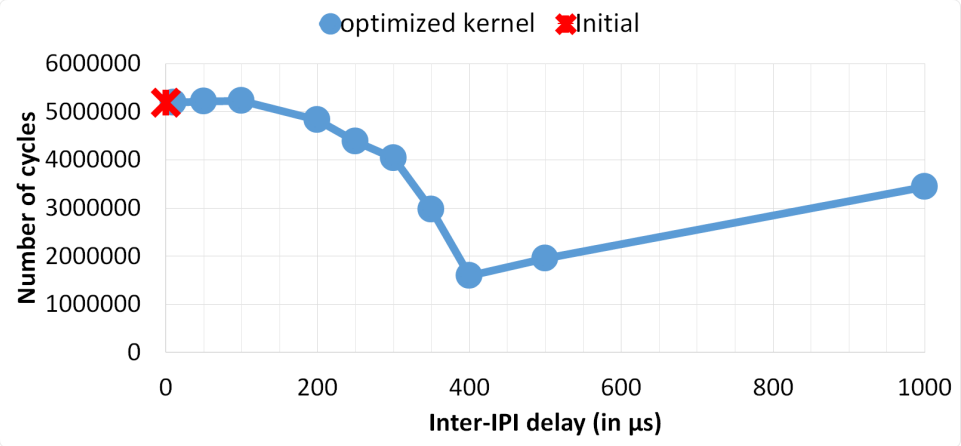

FIGURE 5.5 – Durée de la phase de réveil pour 64 processus en fonction du délai introduit entre IPI

L'étude théorique de la fonction d'ordonnancement révèle l'occurrence d'accès à la mémoire afin de récupérer la liste des processus éligibles, comme expliqué dans le sous-section 5.3.4. Le rapprochement des exécutions de cette fonction par les différents cœurs de calcul induit alors des accès concurrents à la mémoire. Nous avons alors intuité l'apparition de contention matérielle au niveau de la mémoire, celle-ci n'étant pas capable des traiter toutes les requêtes en même temps. Ainsi, nous avons suggéré d'appliquer une méthode utilisée habituellement dans le monde du HPC (*High Performance Computing*), consistant à introduire un léger délai entre deux envois d'IPI, ce qui implique par la même occasion un léger décalage des procédures d'ordonnancement réduisant ainsi la contention mémoire.

Évaluation de la solution logicielle

La figure 5.5 présente les durées des phases de réveil de 64 processus attachés à 64 cœurs en fonction du délai introduit entre les IPI.

L'axe des abscisses représente le délai introduit entre deux envois d'IPI en microsecondes. L'axe des ordonnées représente la durée de la phase de réveil en nombre de cycles. La croix rouge montre le durée initiale de la phase de réveil sans optimisation. Les points bleus exposent les résultats obtenus sur notre plateforme à 64 cœurs avec l'optimisation.

Les résultats obtenus confirment notre hypothèse quant au problème de contention. En effet, la durée de la phase de réveil diminue lorsque que le délai entre IPI augmente (jusqu'à 400µs) grâce à la réduction de la contention. Au dessus de 400µs de délai entre IPI, la durée de réveil croit linéairement en suivant le délai inséré. Cela s'explique par le fait qu'à partir de ce point, le délai introduit devient supérieur au gain apporté par la réduction de la contention.

Par ailleurs, ces résultats montrent qu'au point optimal (délai de 400µs), la durée totale nécessaire au réveil des 64 processus est d'environ 1 681 200 cycles. Sans optimisation, la durée de la phase de réveil était d'environ 5 196 200 cycles. L'optimisation logicielle permet alors un gain substantiel d'environ 67%.

Une fois l'évaluation de notre solution d'optimisation effectuée pour une plateforme de 64 cœurs, nous l'avons évaluée pour des plateformes de 16, 24 et 36 cœurs (soit 4, 6 et 9 grappes de 4 cœurs).

Cette évaluation a révélé une tendance générale quant à l'évolution de la durée de réveil en fonction de la variation des délais entre IPI. Trois phases distinctes ont pu être

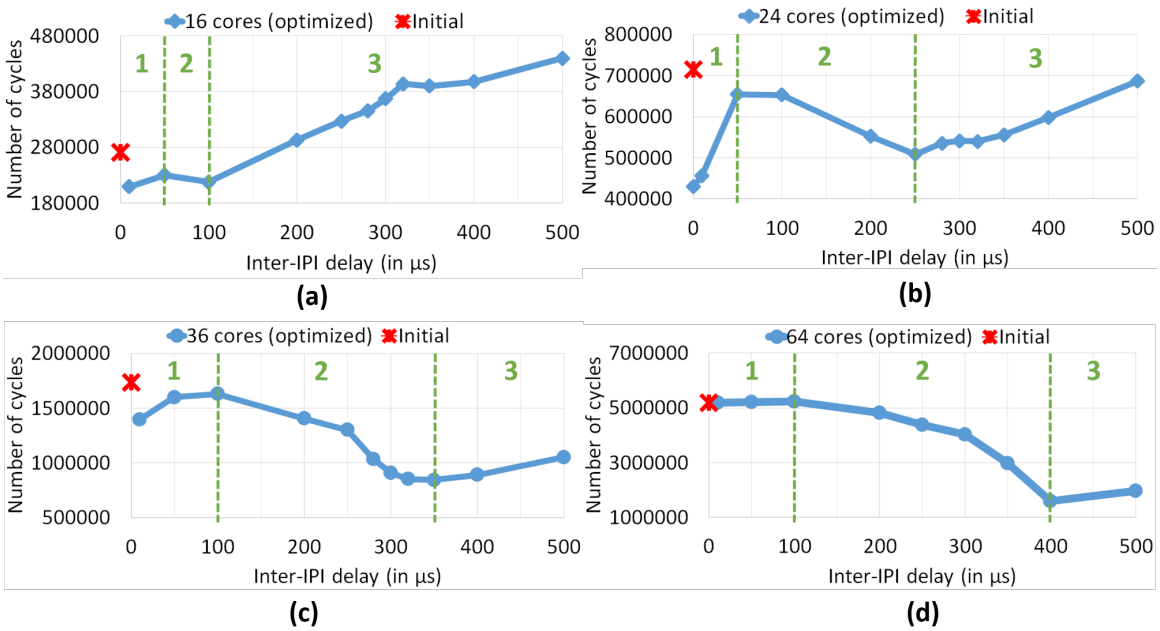

FIGURE 5.6 – Durées des phases de réveil en fonction du délai introduit entre deux envois d'IPI : (a) pour 16 cœurs, (b) pour 24 cœurs, (c) pour 36 cœurs, (d) pour 64 cœurs

identifiées :

1. Une première phase durant laquelle la durée de réveil augmente tant que le délai entre IPI est inférieur à un seuil.
2. Lorsque le seuil est dépassé, la durée de la phase de réveil diminue rapidement jusqu'à ce que le point d'équilibre soit atteint. Il s'agit du point pour lequel le gain provenant de la réduction de la contention est annulé par le coût résultant de l'ajout du délai entre deux envois d'IPI.
3. Après ce point, la durée de la phase de réveil augmente linéairement en fonction du délai entre IPI ajouté.

Ces trois phases sont mises en évidence sur la figure 5.6 représentant les durées de réveil en fonction du délai introduit entre IPI pour chaque plateforme : 16, 24, 36 et 64 cœurs. La première phase d'augmentation de la durée de la phase de réveil peut être expliquée par le fait que les procédures d'ordonnancement des processus exécutées par les différents cœurs ne sont pas suffisamment décalées dans le temps pour éviter les problèmes de contention mémoire. Cependant, l'ajout du délai entre IPI vient s'ajouter au ralentissement provenant de la contention, ce qui engendre une augmentation de la durée totale de la phase de réveil. Une fois le seuil passé, la contention mémoire chute, ce qui entraine la réduction de la durée de la phase de réveil. Enfin, après le point d'équilibre, le délai entre IPI devient surdimensionné ce qui implique l'augmentation de la durée nécessaire au réveil de l'ensemble des processus.

Les points d'équilibre contention/délai entre IPIs pour les plateformes évaluées sont présentés figure 5.7. Le point d'équilibre est lié à la contention apparaissant sur la plateforme. Sachant que la contention augmente avec le nombres de cœurs procédant à l'ordonnancement des tâches, le délai entre IPI du point d'équilibre d'une plateforme augmente avec le nombre de cœurs.

Le tableau 5.3 présente le gain au point d'équilibre pour chaque plateforme évaluée. Nous remarquons que les gains réalisés par cette optimisation sont importants pour chaque plateforme, avec une tendance à la hausse pour les plateformes subissant des fort

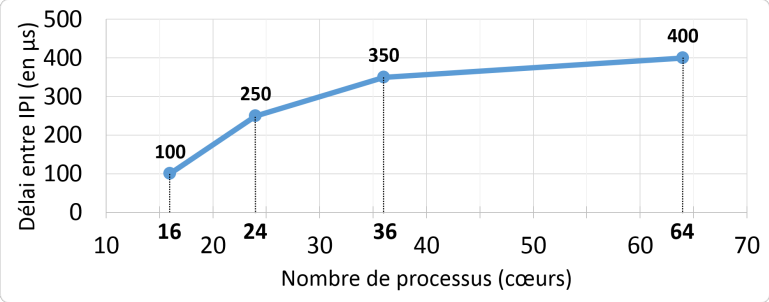

FIGURE 5.7 – Points d'équilibre entre la contention et le délai introduit entre IPI en fonction du nombre de processus

TABLEAU 5.3 – Gain observé au point d'équilibre contention/délai entre IPI pour différentes plateformes TSAR

Nombre de processus	16	24	36	64
Gain	20%	29%	51%	67%

problèmes de contention. Il est par ailleurs intéressant de noter, comme cela est visible sur les figures 5.6.a et 5.6.b, que la campagne de mesures a révélé que pour les plateformes de 16 et 24 cœurs les problèmes de contention ne sont suffisamment important pour tirer profit du décalage de l'envoi d'IPI. En effet, pour les plateformes avec une contention relativement faible, c'est lorsque le délai entre IPI est nul (0 µs) que la phase de réveil est la plus rapide. Pour ces plateformes, le délai entre IPI optimal n'est donc plus le délai au point d'équilibre, quand bien même le gain obtenu au point d'équilibre reste significatif.

Les résultats obtenus mettent en évidence le fait que notre proposition d'optimisation est intéressante pour différentes tailles de plateformes, et plus spécifiquement pour celles subissant une forte contention. Cependant, une caractérisation théorique du point d'équilibre n'est pas chose aisée, car ce point dépend directement de la contention dans le système, qui résulte quant à elle d'un grand nombre d'éléments (architecture matérielle, activité logicielle, ...).

Afin d'évaluer l'impact de notre proposition d'optimisation sur une application réelle, nous mesurons le temps d'exécution de notre application de référence, l'application « Integer Sort » (classe « S ») du benchmark *NAS*, pour une plateforme de 64 cœurs. Nous avons dans un premier temps mesuré le temps d'exécution de cette application sans optimisation, puis nous avons réitéré la mesure avec notre proposition d'optimisation au point optimal, c'est à dire pour un délai entre IPI de 400µs. Les résultats obtenus sont présentés dans le tableau 5.4. Nous remarquons un gain conséquent de 8.8% qui montre que notre optimisation permet d'améliorer les performances globales d'une application réelle.

TABLEAU 5.4 – Application « IS » du benchmark *Nas Parallel* avec et sans optimisation pour une platforme de 64 coeurs

Application	Noyau non optimisé	Noyau optimisé	Gain
IS (classe S)	2110×10^6 cycles	1924×10^6 cycles	8.8%

Analyse des faiblesses matérielles

La section précédente a montré que les performances du mécanisme de barrière de synchronisation GNU OpenMP sont très fortement impactées par les restrictions matérielles, et notamment les phénomènes de contention. Aussi, nous avons voulu poursuivre notre étude par l'analyse de la contention matérielle se produisant lors de la phase de réveil.

La première phase de notre méthodologie d'étude consiste à déterminer les sections critiques. Dans le cas présent, il s'agit plutôt de déterminer les éventuelles faiblesses matérielles afin de les étudier plus en détail.

Sur l'architecture TSAR, l'étude théorique du chemin de données vers la mémoire met en évidence une faiblesse potentielle au niveau de la politique de gestion des priorités de l'arbitre du *crossbar* interne aux grappes. Au sein de ce *crossbar*, toutes les requêtes à destination de la mémoire locale sont routées vers une seule destination : N vers 1, avec N représentant le nombre d'initiateurs dans le *crossbar*. La politique d'arbitrage alloue le même temps de communication pour chaque initiateur du *crossbar* grâce à une stratégie de type *round robin*. Or, l'ensemble des requêtes provenant des cœurs distants (hors de la grappe implémentant la mémoire ciblée) passe à travers l'interface réseau de la grappe destination (NIC sur la figure 1.5). Cette interface réseau est vue comme étant un unique initiateur sur le *crossbar* de la grappe, et ne dispose donc que d'un seul créneau de communication. Ainsi, lorsque tous les cœurs essaient d'accéder simultanément à la mémoire d'une grappe, les cœurs distants sont pénalisés, et de nombreuses requêtes provenant de ces cœurs se retrouvent bloquées au niveau de l'interface réseau dans l'attente de l'obtention du droit d'accès au *crossbar*.

Une potentielle faiblesse matérielle a donc été identifiée par l'étude théorique du système. Nous pouvons désormais passer à la seconde étape de la méthodologie d'étude consistant en l'évaluation pratique de cette faiblesse. Pour cela, la chaine de mesure est configurée afin de réaliser l'extraction des signaux relatifs à l'arbitre du *crossbar*. Ensuite nous exécutons la micro-application 5.1 sur une plateforme de 64 cœurs. Enfin, l'*analyseur de contention* permet de traiter les données extraites.

La figure 5.8 présente le nombre de requêtes en attente d'accès au *crossbar* pour les 16 grappes (clusters) composant la plateforme 64 cœurs. L'axe des abscisses représente l'identifiant de la grappe (*cluster)*. L'axe des ordonnées représente le cumul du nombre de requêtes en attente du droit d'accès au *crossbar* au cours de l'exécution de la phase de réveil de la barrière de synchronisation.

Nous pouvons noter sur cette figure, que les grappes 1, 2 et 16 subissent beaucoup plus de contentions que les autres, ce qui indique que les mémoires de ces trois grappes sont beaucoup plus sollicitées. Cela signifie que quelques bancs mémoires contiennent la majorité des données et instructions accédées pendant la phase de réveil. À partir de cette observation, nous pouvons déduire que l'allocation mémoire du noyau Linux ne répartit pas correctement les données/instructions de la phase de réveil sur les différentes mémoires du MPSoC, et cela en dépit du fait que l'option de réplication du code noyau (« *replication of the kernel code* ») ait été positionnée lors de la compilation du noyau. L'amélioration de la politique d'allocation mémoire du noyau Linux pour les architectures MPSoC serait alors une source d'amélioration potentielle des performances logicielles grâce à la réduction de la contention mémoire. Cependant, cette modification complexe étant hors du cadre défini pour ce travail de thèse, nous n'avons pas poursuivi dans cette voie.

Par ailleurs, la figure 5.8 montre aussi la répartition des requêtes en fonction de leur

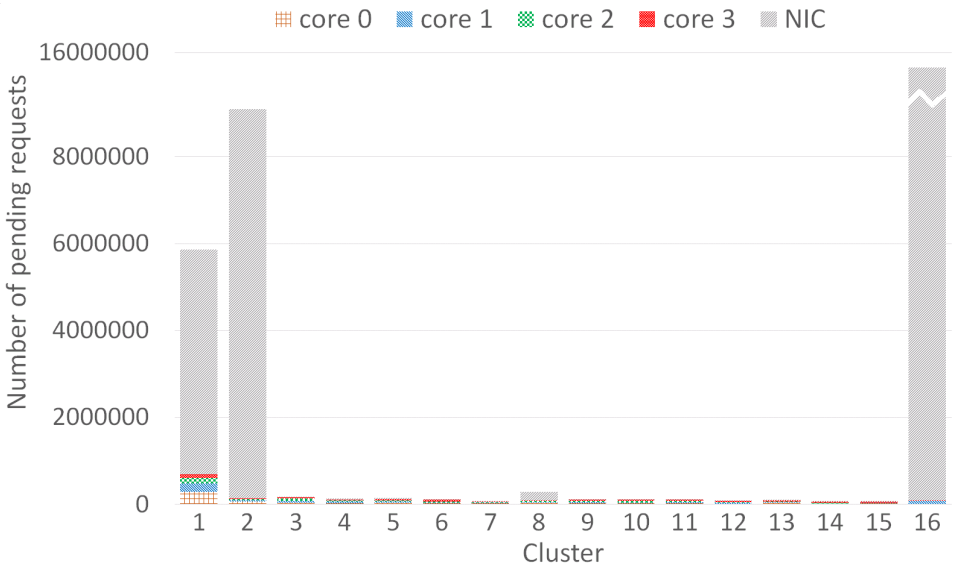

FIGURE 5.8 – Nombre de requêtes en attente pour chaque grappe de la plateforme de 64 cœurs au cours de la phase de réveil d'une barrière de synchronisation

initiateur. Nous remarquons que la majorité des requêtes en attentes sont celles provenant de l'interface réseau (NIC), ce qui confirme notre hypothèse quant à la faiblesse de la politique d'arbitrage du *crossbar*.

Proposition d'optimisation matérielle

Une source du ralentissement a donc été identifiée. Nous pouvons passer à la troisième phase de notre méthodologie, en proposant une optimisation ciblée.

Afin d'améliorer les restrictions de la politique d'arbitrage du *crossbar*, nous proposons de modifier celle-ci de sorte d'accorder 4 fois plus de temps d'accès pour les requêtes extérieures (NIC) que pour les requêtes locales en cas de contention. Cela signifie que lorsque des requêtes sont en attente sur le NIC, ce dernier pourra envoyer quatre requêtes au cours d'une ronde d'arbitrage alors que les cœurs locaux ne pourront en envoyer qu'une seule.

Évaluation de la solution matérielle

Une fois l'optimisation matérielle mise en œuvre, nous pouvons débuter la phase d'évaluation des performances de la solution proposée. Pour cela, nous mesurons la durée de la phase de réveil pour 64 processus lors de l'exécution de la micro-application 5.1 sur une plateforme de 64 cœurs. Afin de garantir la reproductibilité des résultats, seule la mesure médiane de 400 exécutions est étudiée.

Les résultats obtenus pour différents délais entre IPI sont représentés par les triangles oranges sur la figure 5.9. Lorsque les délais entre IPI sont petits, c'est à dire lorsque que la contention est élevée, les résultats obtenus avec la nouvelle politique du crossbar montrent un gain important d'environ 30%. En ce qui concerne les cas ou la contention est réduite, le nombre de requêtes bloquées en attente d'accès au *crossbar* est aussi réduit, ce qui explique le fait que les résultats avant et après la modification de la politique d'arbitrage du *crossbar* soient semblables. Par ailleurs, nous pouvons remarquer que l'optimisation réalisée ne permet pas de réduire suffisamment la contention mémoire pour affecter le point de fonctionnement optimal (durée de la phase de réveil la plus petite).

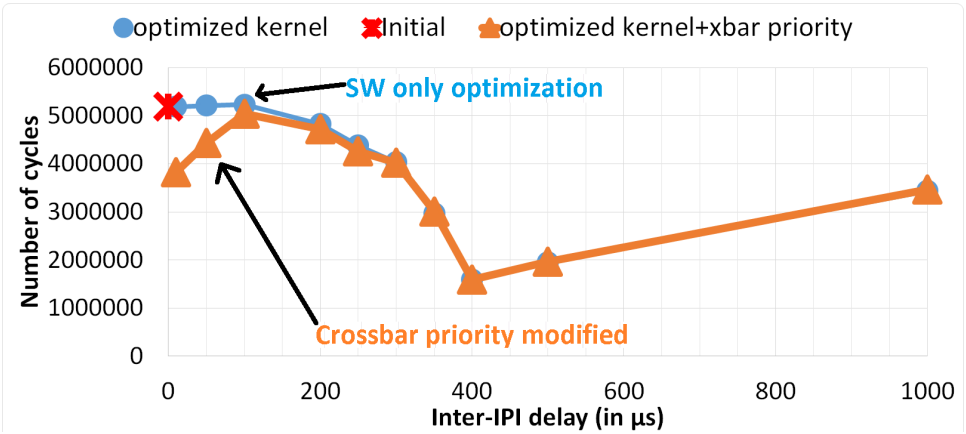

FIGURE 5.9 – Durée de la phase de réveil pour 64 processus en fonction du délai introduit entre IPI (arbitre du *crossbar* modifié)

Cette étude met en avant le fait que la mémoire cache L2 représente le vrai goulot d'étranglement à l'origine de l'apparition de la contention, car cette dernière n'est pas en mesure de traiter à temps toutes les requêtes incidentes. Aussi, l'augmentation de la vitesse de traitement de ces requêtes au niveau du cache L2 permettrait d'améliorer les performances de la phase de réveil de la barrière de synchronisation, ainsi que les performances du logiciel de manière générale. Le CEA Leti est actuellement entrain de refondre le cache L2 de la plateforme TSAR afin de paralléliser le service des requêtes, permettant ainsi d'accélérer la vitesse de traitement du cache.

5.4 Conclusion

Au cours de ce chapitre, nous avons présenté notre méthodologie d'optimisation des mécanismes de synchronisation reposant sur une observation précise de leur comportement dans un environnement de fonctionnement réaliste. Grâce à cette méthodologie, nous avons pu étudier le mécanisme de barrières de la bibliothèque GNU OpenMP. Cette étude a donné lieu à deux optimisations améliorant les performances de ce mécanisme :

— la première située dans la bibliothèque GNU OpenMP vise le procédé de libération de l'attente active ;

— la seconde, au niveau du noyau Linux, permet d'améliorer le procédé de réveil de multiple processus sur une plateforme MPSoC.

Ces deux optimisations engendrent des gains substantiels sans modifier les habitudes des utilisateurs, et en garantissant une mise en place simple nécessitant uniquement la mise à jour du logiciel (bibliothèque ou noyau). Ces optimisations, ainsi que la méthodologie qu'elles mettent en œuvre, permettent d'apporter des éléments de réponse à la question soulevée dans le chapitre 2, à savoir : Comment fournir un mécanisme permettant d'accélérer les applications utilisateurs (à haut niveau), tout en restant simple à mettre œuvre ? En effet, l'optimisation des couches logicielles suite à une étude précise en conditions réelles semble être une bonne option.

Par ailleurs, la méthodologie mise en place avec l'outil de mesure ouvre de nombreuses perspectives quant à l'étude et à l'optimisation d'autre mécanismes de synchronisation, voire même d'autres mécanismes logiciels critiques avec une extension des outils d'analyse proposés par la chaine de mesure.

Chapitre 6

Lockality, un verrou mobile optimisé pour les architectures MPSoC

Sommaire

Dans ce chapitre nous présentons *Lockality*, une solution innovante de verrou mobile pour les MPSoC facilitant le passage à l'échelle. Pour ce faire, nous débuterons par exposer les motivations nous ayant incité à proposer ce type de solution. Ensuite, nous présenterons le principe général de notre solution ainsi que ses avantages. Enfin, nous évaluerons les performances obtenues avec *Lockality* sur la plateforme MPSoC TSAR. Nous conclurons sur les limites et les perspectives d'évolutions.

6.1 Utilisation des verrous dans les applications parallèles

6.1.1 Principe de localité spatiale des verrous

Au cours de notre étude bibliographique, un trait caractéristique sur l'utilisation des verrous de synchronisation a éveillé notre curiosité. Certains travaux tels que ceux de Kuo *et al.* [KCK99] ou Rutgers *et al.* [RBS12] affirment que les verrous libérés par un processus s'exécutant sur un cœur de calcul 'C' seront dans de nombreux cas repris par un processus s'exécutant sur le même cœur de calcul 'C'. Nous dirons alors que le verrou est *réutilisé* par le cœur de calcul ou processeur. Nous parlerons de *chaine de réutilisation* pour désigner le nombre de fois consécutives où un verrou libéré par un processus s'exécutant sur un cœur d'une grappe de calcul est acquis par le même cœur ou un cœur de la même grappe.

Ces travaux évoquent une deuxième spécificité quant à l'évolution temporelle de la réutilisation des verrous au fil du temps. En effet, les chaines de réutilisation d'un verrou ne sont pas l'apanage d'un cœur ou d'une grappe à l'exception des autres, ces chaines de réutilisation peuvent se produire sur différents cœurs (grappes) au cours de l'exécution du logiciel. Cela a pour conséquence que certaines chaines de réutilisation doivent faire des accès répétés à un verrou localisé dans une mémoire distante (mémoire associée à une autre grappe), impactant de ce fait les performances du logiciel en raison d'une latence d'accès plus longue ainsi que d'une augmentation du trafic réseau.

Ainsi, nous pouvons comprendre l'importance de différencier deux types d'accès à un verrou en fonction de sa localisation physique en mémoire :

— accès distants : accès à un verrou dont la variable de synchronisation est située dans une mémoire distante (mémoire localisée dans une autre grappe que le cœur à l'origine de la requête)

— accès locaux : accès à un verrou dont la variable de synchronisation est située dans une mémoire locale (mémoire localisée dans la même grappe que le cœur à l'origine de la requête)

Afin de tirer profit du principe de localité spatiale, les deux travaux précédemment cités proposent de maximiser le nombre d'accès locaux grâce au déplacement de la variable mémoire du verrou dans la mémoire associée au processeur ayant acquis le verrou. De la sorte, un processeur peut acquérir un verrou directement depuis sa mémoire locale en cas de réutilisation.

6.1.2 Vérification expérimentale de la réutilisation des verrous dans les MPSoC

Afin de confirmer les hypothèses énoncées quant aux caractéristiques d'utilisation des verrous, et d'étudier le comportement des verrous sur un MPSoC, nous avons évalué le taux de réutilisation des verrous sur l'architecture TSAR. Grâce à la chaine de mesure développée (cf chapitre 4), nous avons pu mesurer les taux de réutilisation des verrous pour trois applications caractéristiques issues du benchmark SPLASH2 [WOT+95] [Son14] : Cholesky, Water-NSquared et Barnes, en utilisant le patch *splash2 CAPSL*[cap] pour la parallélisation du code avec la bibliothèque *Posix Pthread*.

Le tableau 6.1 expose les résultats de mesure du taux réutilisation des verrous pour nos trois applications de référence sur une plateforme TSAR composée de 64 cœurs de calculs regroupés en 16 grappes.

Les mesures effectuées confirment la première hypothèse concernant la forte réutilisation des verrous par cœurs, avec des verrous réutilisés à plus de 60%, ce qui signifie que

TABLEAU 6.1 – Taux de réutilisation des verrous pour 64 processus s'exécutant du 64 cœurs (16 grappes)

Application	Taux de réutilisation par cœur	Taux de réutilisation par grappe
Choleski	~63%	~66%
Barnes	~67%	~75%
Water-Nsquared	~68%	~75%

plus d'une fois sur deux un verrou est acquis par le dernier cœur l'ayant relâché. Les résultats obtenus montrent aussi que le taux de réutilisation par grappe est encore plus élevé que celui mesuré par cœur. Il s'agit du ratio pour lequel un verrou libéré par un cœur est acquis la fois suivante par un cœur de la même grappe.

Concernant la seconde hypothèse, nous exploitons l'outil *analyseur de verrous* de notre chaine de mesure afin de caractériser la répartition des chaines de réutilisation des verrous selon les accès locaux ou distants.

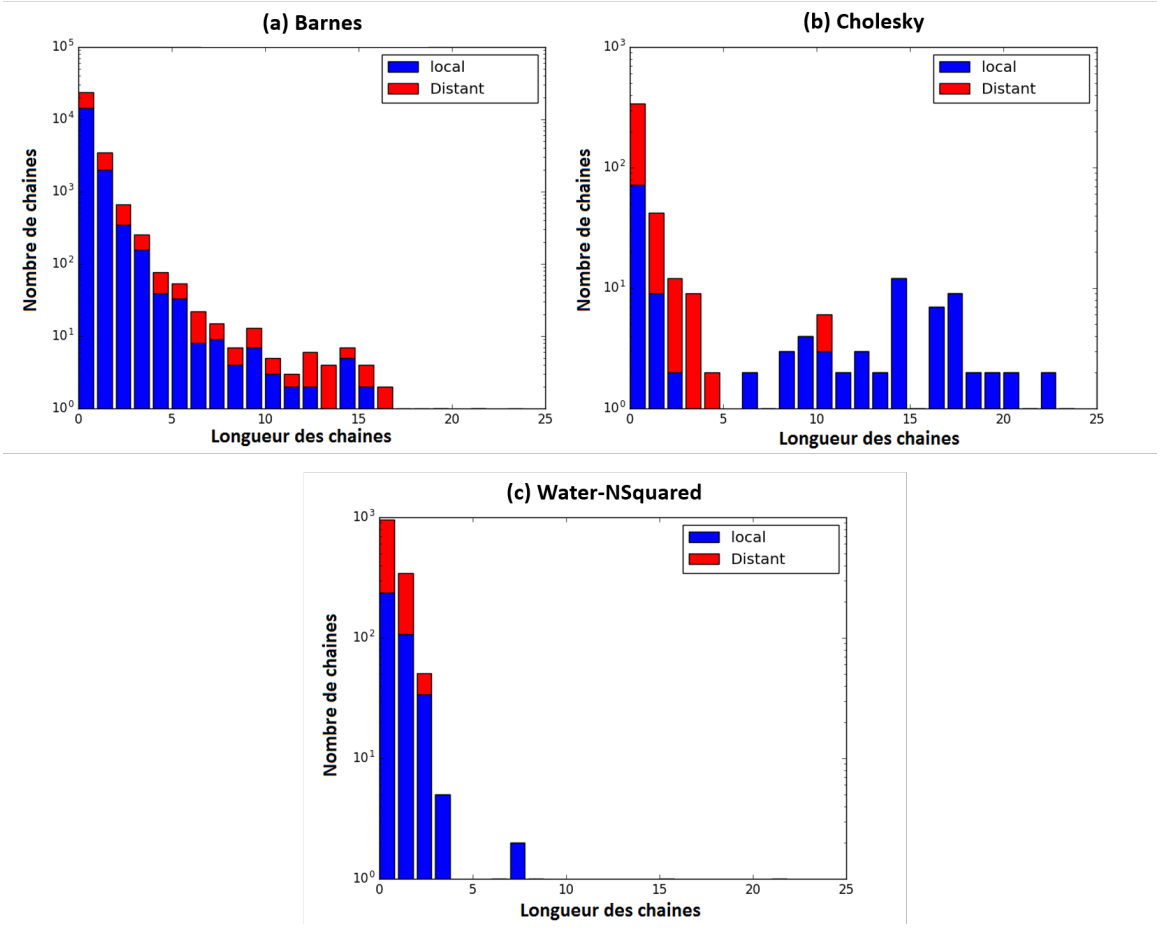

FIGURE 6.1 – Longueurs des chaines de réutilisation par grappe pour 64 cœurs (16 grappes) pour les applications, (a) Barnes, (b) Cholesky, (c) Water-nsquared

Les graphes de la figure 6.1 présentent la longueur des chaines de réutilisation pour nos trois applications de référence. L'axe des abscisses représente la longueur des chaines (i.e. le nombre d'accès consécutifs par des processus s'exécutant sur la même grappe de calcul). L'axe des ordonnées représente le nombre d'occurrence des chaines au cours de l'application pour une longueur donnée. Si nous prenons en exemple le graphe 6.1.a,

nous pouvons voir qu'approximativement 200 chaines de réutilisation de longueur 3 se sont produites lors de l'exécution de l'application Barnes (c'est à dire 3 réacquisitions successives au sein de la même grappe. i.e. la même grappe obtient 4 fois de suite l'accès au même verrou). Les chaines de longueur 0 représentent les cas où deux accès successifs au verrou proviennent de deux grappes différentes.

Les types d'accès relatifs à chaque chaine sont aussi représentés sur la figure 6.1 : les chaines accédant à un verrou situé dans la grappe du CPU exécutant le processus (accès locaux) sont représentées en bleu, les chaines accédant à un verrou situé dans une autre grappe (accès distant) sont représentées en rouge.

Les informations sur le type d'accès local/distant mettent en évidence le nombre non négligeable des deux types d'accès. Cette répartition laisse à penser qu'une instanciation intelligente du verrou, c'est à dire allouer le verrou de manière statique dans la mémoire de la grappe faisant le plus d'accès, laisserait un nombre d'accès distants importants.

Ces résultats mettent donc en exergue le fait qu'une exploitation judicieuse du principe de localité spatiale des verrous peut améliorer les performances globales de ces mécanismes. À savoir que si un verrou venait a être relocalisé dans la mémoire proche du dernier cœur ayant acquis le verrou, nous pouvons supposer que l'accès suivant proviendra de la même grappe, et pourra être réalisé de manière locale, réduisant non seulement la latence d'accès, mais aussi le trafic réseau.

En effet, même si dans un cas sans contention matérielle, le surcoût de latence d'accès au niveau matériel à une mémoire distante peut sembler limité (~6 cycles + 3 cycles par routeur traversé sur notre plateforme TSAR) par rapport à la durée totale (logiciel + matériel) d'acquisition d'un verrou (quelques centaines de cycles, comme nous le verrons section 6.4.1), lors de l'apparition de contention, cette latence d'accès s'accroit rendant la durée d'un accès local bien plus attractive.

De plus, la relocalisation des verrous œuvre aussi dans le sens de l'amélioration des performances globales de l'application du fait qu'elle réduit le nombre de trames sur le réseau, diminuant ainsi le trafic réseau et le risque de contention.

6.1.3 Relocalisation dynamique des verrous

À ce stade, deux optiques peuvent être envisagées pour réaliser ces déplacements de verrous : logicielle ou matérielle.

En ce qui concerne l'approche logicielle, nous pouvons imaginer à minima deux stratégies :

— La réallocation logicielle dynamique du verrou dans une plage d'adresses couverte par la mémoire locale au processus ayant acquis un verrou. Cependant cela implique un protocole relativement complexe afin de garantir qu'aucun accès au verrou ne soit réalisé durant sa réallocation.

— La gestion des déplacements par messages explicites à l'instar des solutions proposées par Kuo *et al.* [KCK99] et Rutgers *et al.* [RBS12].

Néanmoins, ces types de solutions pâtissent du surcoût des couches logicielles, qui les rendent non concurrentielles par rapport aux mécanismes de verrous classiques reposant sur les protocoles de cohérence de cache.

Ainsi, afin de déplacer les verrous dans un délai du même ordre de grandeur que les temps d'accès à une mémoire distante, l'optique matérielle est la seule envisageable en vue de proposer une solution performante. Il s'agit alors de développer un support matériel capable de déplacer les verrous d'une grappe à une autre de manière sûre, c'est à dire

en garantissant à chaque instant la cohérence, au niveau système, des états associés à un verrou.

6.1.4 Le passage à l'échelle

Dans un contexte d'augmentation du nombre de cœurs de calcul, le passage à l'échelle du système de déplacement des verrous se révèle nécessaire. Dans les articles [KCK99] et [RBS12], les solutions proposées reposent sur un gestionnaire central mémorisant la localisation physique des verrous à chaque instant, afin de pouvoir soit en informer le demandeur d'un verrou [RBS12], soit transférer directement la demande vers l'endroit de stockage du verrou [KCK99]. Néanmoins, en plus de la latence ajoutée sur le chemin d'acquisition du verrou, cette gestion implique l'augmentation du nombre de trames réseau, ainsi que la création d'un point chaud centralisant l'ensemble des requêtes.

Nous comprenons alors l'importance de proposer une solution permettant la gestion des verrous et de leurs déplacements de manière décentralisée, afin de supporter le passage à l'échelle.

En vue de permettre la gestion décentralisée des verrous, nous fixons une hypothèse de travail concernant les plateformes MPSoC ciblées. Considérant que l'augmentation du nombre de cœurs de calcul dans les MPSoC est tributaire de l'exploitation des réseaux sur puce en tant que *media* de communication entre grappes, nous supposons que ces réseaux sur puce implémentent la fonctionnalité de diffusion (*broadcast*) des trames permettant la décentralisation de la gestion des verrous comme nous le verrons dans la section 6.2.2.

6.2 *Lockality*, un verrou mobile décentralisé

6.2.1 Principe de fonctionnement

Afin d'exploiter le principe de localité spatiale des verrous, nous proposons un mécanisme matériel assurant le déplacement d'un verrou dans une mémoire proche du cœur l'ayant acquis, et cela de manière automatique.

La figure 6.2 illustre le principe de fonctionnement de la solution proposée. Les traits pleins verticaux représentent la vie des processus A et B. Les traits en pointillés représentent les zones mémoires associées au verrou. LockBuf(A) est la mémoire proche du processus A, LockBuf(B) est la mémoire proche du processus B. Le processus A crée un nouveau verrou (*Create*) ce qui entraine son instanciation dans la mémoire locale associée au processus A. Le processus A fait alors une demande de prise de verrou (*Lock*) vers sa mémoire locale. Le verrou est présent dans la mémoire locale et il est libre, l'accès est donc accordé au demandeur. Par la suite, le processus A relâche le verrou (*Unlock*). Aucune demande extérieure n'est encore parvenue au verrou, le verrou reste donc physiquement localisé dans la mémoire proche du processus A. Ensuite, le processus A fait à nouveau une demande d'accès au verrou. La demande est envoyée vers la mémoire proche qui est toujours responsable du verrou et ainsi peut directement accorder l'accès au processus A. Ce dernier relâche ensuite le verrou. À ce moment, c'est au tour du processus B de demander le verrou. Le processus émet une demande vers sa mémoire locale qui ne possède pas le verrou. Cette mémoire se charge alors de transmettre la demande (*Request lock*) à la mémoire possédant le verrou (la mémoire proche de A). Le verrou étant libre, la mémoire renvoie le droit d'accès, ainsi que la responsabilité du verrou à la mémoire

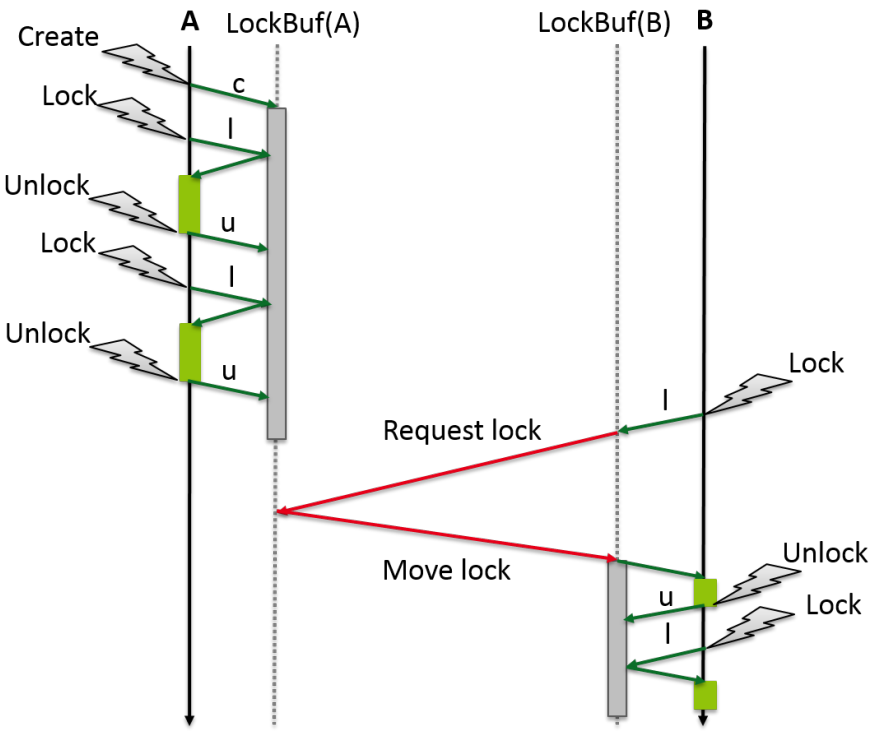

FIGURE 6.2 – Schéma de principe de la solution *Lockality*

proche de B (*Move lock*). Le verrou devient alors localisé dans la mémoire locale au processus B. La mémoire locale du processus B lui accorde alors le droit d'accès au verrou. Le processus B relâche ensuite le verrou en envoyant un ordre de relâchement à sa mémoire locale. Par la suite, le processus B demande à nouveau le droit d'accès au verrou à sa mémoire locale. Étant donné qu'aucune requête distante n'a été reçue entre temps, le verrou est toujours localisé dans la mémoire locale à B, qui peut alors directement lui donner le droit d'accès.

6.2.2 Une solution décentalisée

La solution que nous proposons pour assurer le déplacement des verrous d'une grappe à l'autre en fonction de l'acquéreur du verrou repose sur des blocs matériels communicants entre eux afin de garantir la cohérence des états des verrous à chaque instant. Ces modules matériels permettent le stockage des verrous ainsi que la gestion de leurs déplacements. Chaque module est responsable des verrous qui sont physiquement hébergés dans sa grappe. Ainsi, notre solution ne dépend pas d'un seul module centralisant les informations des verrous, mais ces informations sont réparties dans le système.

Cette décentralisation de l'information impose que chaque module connaisse le module responsable d'un verrou afin de pouvoir lui adresser ses requêtes. Pour ce faire, nous exploitons le principe de diffusion (*broadcast*) des réseaux sur puce afin de communiquer cette information, en une seule trame, à l'ensemble des modules lors d'un déplacement. Cette décentralisation de la gestion permet d'obtenir l'information sur l'état du verrou en seulement 2 trames réseau, contre potentiellement 3 avec la proposition de kuo *et al.* [KCK99], et 4 avec la proposition de Rutgers *et al.* [RBS12] pour un verrou distant.

Une autre force de notre solution est de permettre le transfert de responsabilités du verrou en même temps que la réponse à sa demande d'acquisition (en cas de verrou libre).

Ainsi, le transfert du verrou n'est pas (ou peu) coûteux par rapport au temps initial d'accès à la mémoire distante, accès nécessaire même dans le cas d'une solution classique.

6.2.3 Étude théorique du gain escompté

L'étude théorique de l'apport de notre solution sur des cas d'utilisation a été menée pour nos trois applications de référence du benchmark SPLASH2 : Cholesky, barnes et WaterN-squared.

Le système *Lockality* devient intéressant d'un point de vue performances si la formule suivante est respectée :

$$Nb_{local} \times T_{local} + Nb_{distant} \times T_{distant} > Nb\_Lockality_{local} \times T\_Lockality_{local} + Nb\_Lockality_{dep} \times T\_Lockality_{dep} \quad (6.1)$$

Nb_{local} : Nombre d'accès locaux avec un verrou classique
T_{local} : Temps d'accès à un verrou local
$Nb_{distant}$: Nombre d'accès distants
$T_{distant}$: Temps d'accès à un verrou distant avec un verrou classique
$Nb\_Lockality_{local}$: Nombre d'accès au verrou *Lockality* local
$T\_Lockality_{local}$: Temps d'accès à un verrou *Lockality* local
$Nb\_Lockality_{dep}$: Nombre de déplacements de verrou *Lockality*
$T\_Lockality_{dep}$: Temps de déplacement d'un verrou *Lockality*

Les blocs matériels *Lockality* gérant les accès aux verrous étant semblables à des mémoires locales, nous pouvons faire l'hypothèse suivante :

$$T_{local} \approx T\_Lockality_{local} \quad (6.2)$$

De plus, le transfert de responsabilité du verrou (déplacement) ayant lieu en même temps que le réponse à la demande de verrou, nous avons :

$$T_{distant} \approx T\_Lockality_{dep} \quad (6.3)$$

Ainsi, l'équation 6.1 peut être réécrite :

$$Nb_{local} \times T_{local} + Nb_{distant} \times T_{distant} > Nb\_Lockality_{local} \times T_{local} + Nb\_Lockality_{dep} \times T_{distant} \quad (6.4)$$

Par ailleurs, nous avons :

$$Nb_{local} + Nb_{distant} = Nb\_Lockality_{local} + Nb\_Lockality_{dep} \quad (6.5)$$

Aussi, si le nombre de déplacements est moindre que le nombre d'accès distants initiaux, le système *Lockality* devrait permettre d'obtenir un gain de temps.

Dans la section 6.1.2, nous présentions les mesures réalisées quant à l'évaluation de la longueur des chaines de réutilisation pour les applications de référence (figures 6.1). À partir de ces informations, nous déduisons le nombre de déplacements nécessaires avec le système *Lockality*. En effet, le verrou doit migré vers une autre grappe à chaque fin de chaine. Ce nombre de déplacements est alors à comparer au nombre d'accès distants de l'implémentation initiale. Le tableau 6.2 présente cette comparaison pour un MPSoC TSAR à 64 cœurs. Il expose, en effet, les types d'accès aux verrous réalisés lors de l'exécution de nos trois applications de référence. Les colonnes 2 et 3 représentent la répartitions

TABLEAU 6.2 – Comparaison du nombre d'accès aux verrous par type (MPSoC TSAR, 64 coeurs)

Applications	Nombre d'accès locaux	Nombre d'accès distants	*Lockality* Nombre d'accès locaux	*Lockality* Nombre de déplacements
Barnes	22547	16160	10235	28472
Cholesky	1687	1014	2226	475
Water-NSquared	3309	1679	3622	1366

entre accès locaux et distants dans le cas d'une solution classique (*Posix*). Les colonnes 4 et 5 estiment les types d'accès lors de l'utilisation de la solution *Lockality*, à savoir le nombre d'accès locaux (colonne 4), ainsi que le nombre de déplacements escomptés (colonne 5).

Nous pouvons voir que pour les applications Cholesky et Water-NSquared, les nombres de déplacements *Lockality* sont inférieurs aux nombres d'accès distants de la solution standard. Seule l'application Barnes présente un nombre de déplacements supérieur. Ces chiffres indiquent que pour les applications Cholesky et Water-NSquared, la solution *Lockality* devrait apporter un gain en performance.

À ce stade, il est important de noter que le nombre de déplacements calculés est en réalité un chiffre majoré. En effet, l'implémentation de *Lockality* prévoit une politique d'attribution des verrous favorisant les attributions locales aux déplacements, tout en se prémunissant contre le phénomène de famine. Aussi, en réalité, les nombres de déplacements devraient être réduits par rapport à ceux exposés dans le tableau 6.2. De ce fait, le gain induit par la solution devrait être supérieur, en supposant que les effets de bord au niveau applicatif de la modification de l'ordre d'exécution des sections critiques ne compensent pas ce gain.

6.3 Implémentation du système *Lockality*

Cette section donne un aperçu de la mise en oeuvre de *Locality* sur une architecture MPSoC tel que TSAR.

6.3.1 Principe

Le système *Lockality* se compose de deux parties : un pilote logiciel et un module matériel.

Le pilote logiciel est en charge de l'interface entre le module matériel et l'utilisateur, ainsi que de la gestion de quelques fonctions essentielles à l'implémentation des mécanismes de verrous (cf sous-section 6.3.2).

Le module matériel s'intègre dans une architecture MPSoC au niveau grappe. Chaque grappe implémente un module appelé *Lock Manager (LM)* en charge de gérer les verrous physiques (attributions, déplacements, ...). Un *Lock Manager* est composé d'une mémoire stockant les états des verrous, ainsi que d'une logique de contrôle en charge de la gestion de ceux-ci.

Le système *Lockality* permet de garantir les propriétés principales suivantes :

— un verrou est possédé par un seul et unique *Lock Manager* à un instant donné,

— la gestion d'un verrou est effectuée par le *Lock Manager* possédant le verrou.

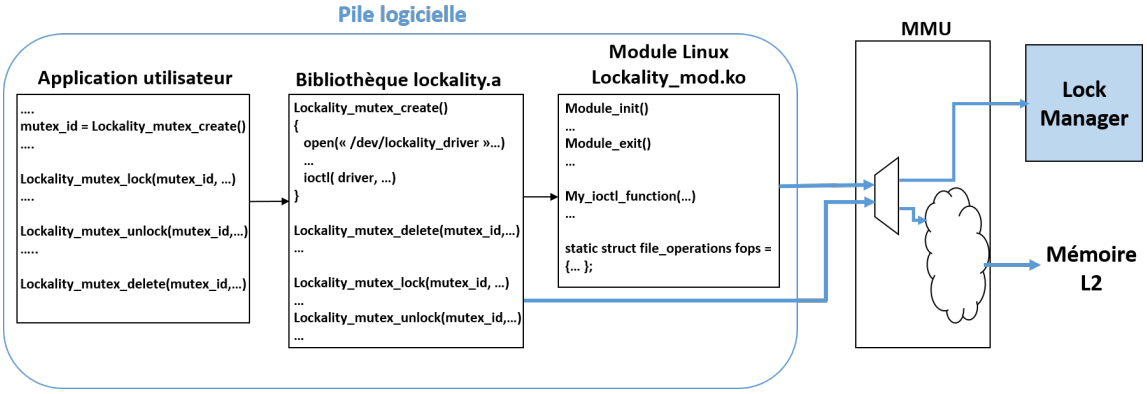

FIGURE 6.3 – Vue d'ensemble du système *Lockality*

6.3.2 Le pilote logiciel

Afin de fournir un système complet, utilisable par un développeur logiciel, nous avons développé un pilote logiciel pour le système *Lockality*. Celui-ci permet de gérer l'interface entre les applications utilisateur désireuses d'utiliser des verrous *Lockality*, et le module matériel gérant les verrous.

Le pilote a été développé pour s'intégrer dans un environnement Linux. Il se compose de deux éléments : une bibliothèque dans l'espace utilisateur, et un module noyau, comme l'illustre la figure 6.3 qui représente l'ensemble de la pile logicielle de l'application utilisateur jusqu'au module matériel *Lock Manager*.

À gauche de la figure 6.3 est représentée une application utilisateur faisant appel la bibliothèque *Lockality*. La bibliothèque *Lockality* (« lockality.a ») expose principalement quatre fonctions accessibles par les utilisateurs :

— **Lockality_mutex_create** : Cette fonction gère la création d'un verrou *Lockality*. Elle garantit l'unicité de l'identifiant associé au verrou. Pour cela, une image logicielle de la table des verrous présents dans les *Lock Managers* est gardée dans la bibliothèque. Cette table est protégée par un mutex logiciel (*Posix*) afin de se prémunir des accès concurrents. En cas de débordement du nombre de verrous matériels disponibles, le bibliothèque bascule automatique sur un mutex logiciel *Posix* en informant l'utilisateur de cela.
Remarque : L'utilisation d'un verrou *Posix* standard dans une implémentation de verrou optimisée peut sembler surprenante. Néanmoins, la création des verrous a lieu pendant une phase d'initialisation, et non pas dans les sections de code critiques fortement parallèles. Nous estimons alors que l'utilisation d'un verrou classique dans une phase non critique n'est pas problématique.

— **Lockality_mutex_delete** : Cette fonction gère la destruction du verrou dans l'ensemble du système.

— **Lockality_mutex_lock** : Cette fonction gère la prise du verrou. Si le verrou ne peut pas être pris, le processus est alors mis en attente active.

— **Lockality_mutex_unlock** : Cette fonction libère le verrou (matériel ou logiciel).

L'API (interface de programmation) a délibérément été choisie la plus proche possible du standard *Posix* afin de faciliter et de minimiser l'adaptation des programmes utilisateur désireux d'utiliser de notre système.

Pour des raisons de performances, les fonctions de verrouillage (*lock*) et de déverrouillage (*unlock*) font directement appel au matériel sans passer par le noyau grâce au

TABLEAU 6.3 – Structure mémoire d'un verrou

Owner (1 bit)	Next cluster (log2(NB_CLUSTER) bits)	Pointer (log2(NB_CLUSTER) bits)	Waiting (1 bit)	Old (1 bit)	Fairness counter (4 bits)

mapping mémoire du module *Lock Manager* directement dans l'espace utilisateur. En effet, la pénalité induite par les appels système (passage par le noyau) rend ce type d'appels impossibles pour des fonctions critiques, bien que plus corrects d'un point de vue conceptuel.

Néanmoins, la création et la destruction des verrous dont les performances ne sont pas critiques sont réalisés grâce au module Linux « Lockality_mod.ko », comme représenté sur la figure.

En plus de l'interface entre les applications utilisateur et le module matériel, le pilote logiciel assure aussi quelques fonctions de base relatives à la gestion des processus (*thread*) Linux. En effet, c'est au niveau de la bibliothèque *Lockality* que la mise en attente des processus est réalisée.

6.3.3 L'architecture matérielle

La gestion des verrous mobiles est assurée par des blocs matériels appelés *Lock Managers*, qui implémentent le protocole défini dans la sous-section 6.3.4. Les communications entre les *Lock Managers* sont établies à travers des réseaux sur puces dédiés initialement à la cohérence de cache sur l'architecture TSAR.

Ces modules matériels sont composés d'un segment de mémoire et d'une logique de contrôle chargée du fonctionnement et de la cohérence globale du système, notamment par l'échange de messages entre *Lock Managers*.

En ce qui concerne la mémoire, il s'agit d'une table mémorisant les données relatives à chaque verrou. Chaque verrou est défini par la structure illustrée par le tableau 6.3.

— Le champ *Owner* informe si le verrou est possédé par le *Lock Manager* courant.

— Le champ *Next Cluster* permet de stocker l'identifiant du prochain *Lock Manager* en liste d'attente pour l'acquisition du verrou.

— Le champ *Pointer* contient l'identifiant du *Lock Manager* propriétaire du verrou.

— Le champ *Waiting* informe sur le fait qu'un processus associé à ce *Lock Manager* est déjà en attente de l'obtention du verrou. Ce champ permet de ne pas réitérer la demande distante si un nouveau processus associé au même *Lock Manager* vient à demander à son tour le verrou.

— Le champ *Old* permet de sauver l'information que le *Lock Manager* était le dernier possesseur du verrou. Il s'agit d'un état intermédiaire après le déplacement d'un verrou, et avant que le transfert de responsabilité soit effectif. Cet état permet garantir qu'une demande de verrou obtient dans tous les cas une réponse, et cela même si le verrou est en cours de déplacement.

— Le champ *Fairness counter* permet de compter le nombre d'attributions locales consécutives du verrou afin d'assurer l'équité dans la politique d'attribution du verrou.

La figure 6.4 illustre l'intégration du module « Lock Manager » au sein d'une grappe de l'architecture TSAR.

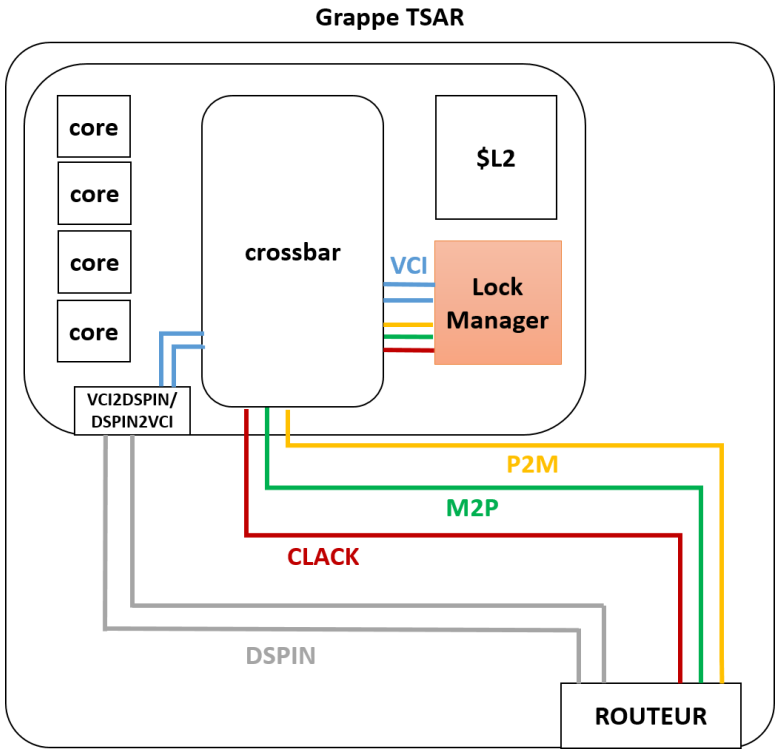

FIGURE 6.4 – Schéma d'implémentation de module *Lock Manager* dans une grappe TSAR

6.3.4 Le protocole de communication

Le protocole de communication entre les *Lock Managers* est la pièce maitresse du système *Lockality*. En effet, c'est lui qui permet l'établissement du principe d'exclusion mutuelle sous-jacente au mécanisme de verrou, en assurant les déplacements des verrous et la cohérence de leurs états à tout instant. Afin d'y parvenir, le protocole garantit les exigences suivantes :

Exigence 1 : Le protocole garantit qu'un seul *Lock Manager* soit propriétaire d'un verrou à un instant donné.

Éléments d'implémentation : Seul le propriétaire complet d'un verrou peut transférer sa propriété.

Exigence 2 : Le protocole garantit que chaque *Lock Manager* a une information cohérente et consistante de l'état de chaque verrou du système.

Éléments d'implémentation : Le propriétaire unique est le seul informé de l'état réel du verrou. Les autres *Lock Managers* disposent uniquement de l'information sur le fait que le verrou n'est pas en leur possession.

Exigence 3 : Le protocole garantit qu'un seul processus au plus ait obtenu le droit d'accès au verrou à un instant donné.

Éléments d'implémentation : Une fois la trame de transfert de propriété émise sur le réseau, l'ancien propriétaire n'est plus autorisé à accorder la droit d'accès au verrou.

Exigence 4 : Le protocole garantit l'absence d'inter-blocage (*dead-lock*) lors d'échange de trames sur le réseau *mesh* 2D routage *X-first*.

Éléments d'implémentation : Trois canaux de communication différents sont mis en œuvre afin de prévenir le système des inter-blocages. Dans le cas de notre implémentation, il s'agit de trois canaux virtuels (notés « CLACK », « M2P » et « P2M » sur la figure 6.4).

Exigence 5 : Le protocole garantit que toute requête d'acquisition d'un verrou obtient une réponse (absence de *live-lock*).

Éléments d'implémentation : Pendant le transfert d'un verrou, l'ancien propriétaire informe le nouveau propriétaire des demandes d'accès au verrou qu'il a reçues.

Exigence 6 : Le protocole garantit l'absence de famine dans l'attribution des verrous.

Éléments d'implémentation : Les demandes locales d'attribution d'un verrou sont privilégiées dans la limite du raisonnable. Lorsque le nombre d'attributions locales dépasse un seuil prédéfinit alors que des demandes distantes sont en attente, la propriété du verrou est obligatoirement transférée au *Lock Manager* suivant.

Exigence 7 : Le protocole garantit l'attribution équitable des verrous.

Éléments d'implémentation : Une liste distribuée des *Lock Managers* en attente d'obtention du verrou est implémentée selon une stratégie proche de celle présentée par Xiao *et al.* dans [XWG+16].

6.4 Évaluation de la solution

6.4.1 Évaluation du gain unitaire

Une fois l'étude des gains théoriques réalisée, et l'implémentation du système *Lockality* achevée, nous avons pu débuter l'évaluation pratique des performances de la solution proposée. La première étape de celle-ci consiste en la comparaison du gain unitaire apporté par notre solution par rapport à la solution standard.

Les procédures d'acquisition de verrou

De nos jours, le standard *de-facto* lorsque nous parlons de verrou (*mutex*) dans les MPSoC est la bibliothèque *Pthread* (*Posix*). Nous avons alors décidé de comparer notre solution avec celle-ci.

Sur la plateforme TSAR, l'implémentation des verrous est réalisée grâce aux instructions *ll* et *sc*, comme expliqué dans le chapitre 2. Cela signifie que le verrou n'est réellement acquis qu'au terme de l'instruction *sc*, comme cela est présenté par l'algorithme 3 du chapitre 2. Cette implémentation requiert donc deux accès à la mémoire tandis que notre solution acquiert un verrou en une seule requête, comme illustré sur la figure 6.5 qui détaille les temps d'accès à un verrou selon les deux stratégies.

La figure 6.5.a représente le découpage temporel de l'acquisition d'un verrou disponible distant avec une stratégie standard tel que *Pthread*. Les traits noirs verticaux représentent la vie des CPU, L1/TLB (*Translation Lookaside Buffer*) et mémoire cache L2. Lorsque le CPU exécute l'instruction *ll*, celle-ci est transmise vers la TLB qui analyse le type de requête (durée *t1*), puis envoie une demande de lecture sur le réseau à destination de la mémoire L2 distante (durée *t_ntwd*). La mémoire L2 distante effectue alors la lecture (durée *t2*) avant de répondre à la requête (durée *t_ntwd*). Une fois la réponse reçue, le CPU va tester la réponse afin de prendre sa décision de continuer le processus d'acquisition (durée *t_cpu*). Après cela, le CPU exécute l'instruction *sc* qui est analysée par la

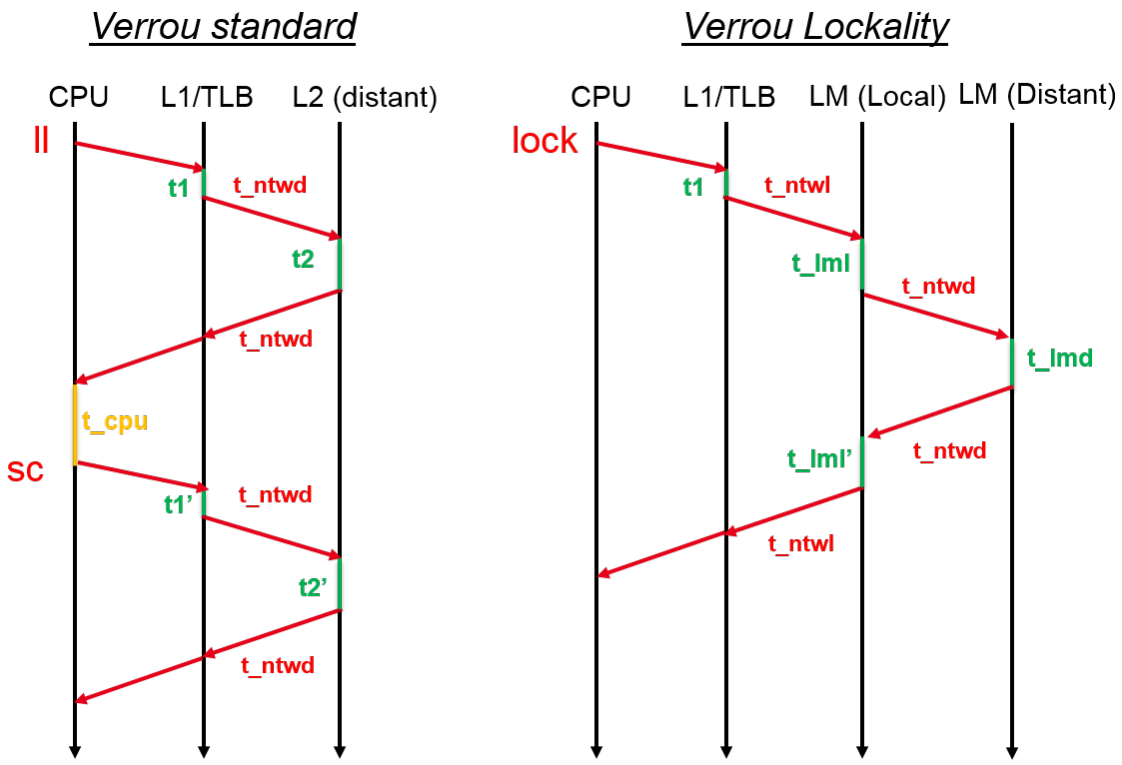

FIGURE 6.5 – Temps d'accès verrou avec et sans le système *Lockality*

TLB (durée *t1'*), puis une requête d'écriture conditionnelle est envoyée sur le réseau (durée *t_ntwd*). La mémoire L2 traite la demande (durée *t2'*) puis répond à la requête (durée *t_ntwd*).

La figure 6.5.b représente le découpage temporel de l'acquisition d'un verrou disponible distant, avec une stratégie *Lockality*. Lorsque le CPU exécute l'instruction de verrouillage *lock*, une demande est adressée à la TLB qui l'analyse (durée *t1*). Une requête à destination du *Lock Manager* local à la grappe est émise (durée *t_ntwl*). Le *Lock Manager* local traite la requête (durée *t_lml*) puis envoie une demande d'acquisition du verrou au *Lock Manager* distant possédant le verrou (durée *t_ntwd*). Le *Lock Manager* distant traite la requête (durée *t_lmd*), puis envoie sa réponse au *Lock Manager* ayant fait la demande (durée *t_ntwd*). Le *Lock Manager* local gère la réponse (durée *t_lml'*), puis répond au CPU (durée *t_ntwl*).

Afin de comparer les temps d'acquisition réels d'un verrou, nous avons mesuré le temps entre l'exécution de l'instruction *ll* et la réponse à l'instruction *sc* pour la stratégie *Pthread*. Pour la solution *Lockality*, nous avons mesuré le temps entre l'exécution de l'instruction de verrouillage *lock* et sa réponse. Ces temps seront désignés par *temps d'acquisition physique du verrou* dans la suite de ce manuscrit.

Mesures du gain unitaire

Afin de mesurer les temps unitaires d'acquisition des verrous locaux et distants, nous avons écrit un programme simple constitué de deux processus légers (*threads*) prenant, puis relâchant cinq verrous (*mutex*) de manière désynchronisée, afin de ne pas rencontrer de contention sur les verrous, et de réaliser toujours une demande sur un verrou disponible. Les processus sont judicieusement assignés à des cœurs de calculs afin de forcer les

TABLEAU 6.4 – Délais médians d'acquisition physique des verrous *Pthread* et *Lockality*

Implémentation	Verrou local	Verrou distant
Pthread	48 cycles	136 cycles
Lockality	2 cycles	34 cycles

TABLEAU 6.5 – Délais médians d'exécution des fonctions de verrouillage *Pthread* et *Lockality* sur la plateforme MPSoC TSAR

Implémentation	Verrou local	Verrou distant
Pthread : *pthread_mutex_lock*	373 cycles	377 cycles
Lockality : *lockality_mutex_lock*	115 cycles	121 cycles

demandes locales et distantes de manière déterministes. Le même programme a été écrit une fois en utilisant les *mutex* de la bibliothèque *Pthread*, et une fois en utilisant ceux de la bibliothèque *Lockality*.

Cette micro-application est la seule application exécutée sur le MPSoC pendant la période de mesure, afin d'éviter toute contention réseau ou mémoire.

Les mesures ont été menées sur une plateforme MPSoC TSAR composée de 16 cœurs regroupés en 4 grappes.

Afin de garantir la reproductibilité des mesures, nous avons exécuté le même programme de tests dix fois consécutives pour chaque version (*Pthread* et *Lockality*).

Le tableau 6.4 représente les temps médians d'acquisition physique des verrous locaux et distants sur notre plateforme TSAR pour les deux stratégies évaluées. Ces résultats montrent que notre solution de bloc matériel dédié à la gestion des verrous permet une prise de verrou plus rapide dans le cas local comme dans le cas distant comparée à l'acquisition du verrou en mémoire par une procédure de type *ll*/*sc*.

Ces délais issus des mesures sur la plateforme viennent modifier l'hypothèse émis dans les formules 6.2 et 6.3. En effet, nous avons en pratique :

$$T_{local} > T\_Lockality_{local} \qquad T_{distant} > T\_Lockality_{dep} \tag{6.6}$$

T_{local} : Temps d'accès à un verrou local
$T_{distant}$: Temps d'accès à un verrou distant avec un verrou classique
$T\_Lockality_{local}$: Temps d'accès à un verrou *Lockality* local
$T\_Lockality_{dep}$: Temps de déplacement d'un verrou *Lockality*

Il s'agit donc d'un cas favorable pour notre solution qui doit permettre de réaliser des gains au niveau applicatif même lorsque le nombre de déplacements *Lockality* n'est pas strictement inférieur au nombre d'accès distants d'une solution classique.

Le tableau 6.5 expose les temps médians nécessaires pour accomplir les fonctions de verrouillage : *pthread_mutex_lock* et *lockality_mutex_lock*.

Le tableau 6.6 expose les temps médians nécessaires pour accomplir les fonctions de déverrouillage : *pthread_mutex_unlock* et *lockality_mutex_unlock*.

Nous pouvons alors remarquer que les délais nécessaires pour les opérations de verrouillage (*lock*) et de déverrouillage (*unlock*) sont moindre avec notre solution *Lockality*, que cela soit pour des verrous locaux ou distants. Ce large gain de plusieurs centaines de cycles s'explique par les meilleures performances d'acquisition physique du verrou, mais pas uniquement. En effet, le gain de l'implémentation ne compte que pour une partie

TABLEAU 6.6 – Délais médians d'exécution des fonctions de déverrouillage *Pthread* et *Lockality* sur la plateforme MPSoC TSAR

Version	Verrou local	Verrou distant
Pthread : *pthread_mutex_unlock*	726 cycles	795 cycles
Lockality : *lockality_mutex_unlock*	207 cycles	251 cycles

TABLEAU 6.7 – Temps d'exécution médians de 5 exécutions des application Cholesky et Water-NSquared avec et sans l'utilisation de 16 verrous *Lockality* sur la plateforme MPSoC TSAR

Application	Cholesky	Water-NSquared
Sans *Lockality*	$30\,160 \times 10^3$ cycles	$209\,146 \times 10^3$ cycles
Avec *Lockality*	$26\,987 \times 10^3$ cycles	$205\,481 \times 10^3$ cycles
Gain	~10,5%	~1,7%

(entre 50 et 100 cycles comme cela est mentionné dans le tableau 6.4). Le gain supplémentaire s'explique par les différences d'implémentation des pilotes logiciels. En effet, les fonctions de verrouillage et de déverrouillage de la bibliothèque *Pthread* sont plus complexes que les nôtres en raison de leur capacité à pouvoir gérer différents types de mutex (lectures multiples, ...). Cette complexité plus importante se paie donc sur le nombre de cycles nécessaires pour l'acquisition (ou le relâchement) d'un simple verrou.

Par ailleurs, nous remarquons que les écarts de délais pour une même option entre accès local et distant ne sont pas une projection directe de la différence de délais d'acquisition physique (tableau 6.4). En effet, la complexité des couches logicielles du noyau Linux gérant les accès au matériel interfèrent de sorte qu'une correspondance directe ne peut pas être déduite.

6.4.2 Gain réel sur des applications complexes

Afin d'évaluer l'impact de la solution proposée sur des applications utilisateur complexes, nous mesurons les temps d'exécution des applications Cholesky et Water-NSquared du benchmark SPLASH2 sur une plateforme MPSoC TSAR de 16 cœurs (4 grappes).

La première difficulté rencontrée lors de cette évaluation fut la sélection des verrous bénéficiant de la solution *Lockality*. En effet, ces applications implémentent plusieurs centaines de verrous. Cependant, les verrous *Lockality*, à l'instar des mécanismes de synchronisation matériels, sont en nombre limité. Dans notre cas, nous avons mené notre évaluation avec 16 verrous *Lockality*. Les verrous applicatifs n'étant pas tous utilisés avec la même intensité, il y a fort à parier que certains se prêtent mieux à la solution proposée que d'autre. Néanmoins, n'ayant pas une connaissance assez fine des applications, nous nous sommes contenté d'affecter les premiers verrous crées par les applications à la solution *Lockality*, puis d'assurer les verrous suivants avec une implémentation *Pthread* classique.

Dans l'objectif de passer outre les résultats de mesures singulières dues aux interférences de la plateforme (interruptions, ...), nous exécutons cinq fois chaque application, puis nous gardons uniquement les temps d'exécution médians, présentés dans le tableau 6.7.

Les résultats de mesure des temps d'exécution exposent les gains réalisés grâce à l'utilisation de notre solution.

TABLEAU 6.8 – Coût matériel du bloc *Lock Manager* 16 verrous comparée au coût d'une grappe de calcul TSAR

Bloc	Surface (22 nm)	Surface (28 nm)	Surface (normalisée)	Nombre de cellules
Grappe TSAR (sans mémoires)	—	799 960 μm	559 972 μm	1 180 837 cellules
Lock Manager	12 571 μm	—	12 571 μm	53 122 cellules
Impact	—	—	2,24%	4,50%

Ces gains sont dus en partie à la réduction des temps d'exécution des fonctions de verrouillage et de déverrouillage, mais pas uniquement. Effectivement, la modification du temps passé dans les mécanismes de synchronisation, ou encore la modification de la politique d'attribution des verrous (comme cela est le cas avec *Lockality* qui favorise les accès locaux), affectent le flot d'exécution du logiciel. Cela entraine une modification de l'ordre d'accès aux mémoires, ce qui influe sur le nombre de défauts de caches, impactant par conséquent le temps d'exécution du logiciel. Aussi, il est difficile, voire impossible de pouvoir lier un gain au niveau applicatif à une seule source d'optimisation. Nous pouvons uniquement dire que l'utilisation de notre solution *Lockality* permet de manière concrète de réduire les temps d'exécution des applications utilisateur complexes.

6.4.3 Impact matériel de la solution

Dans cette section nous évaluons l'impact matériel du module *Lock Manager* sur une grappe TSAR. Pour ce faire, nous avons synthétisé le *Lock Manager* en technologie 22 nm FDSOI, en utilisant le flot de synthèse standard utilisé dans le laboratoire de rattachement au sein du CEA Leti. Le tableau 6.8 présente les résultats quant à la surface et au nombre de cellules d'un *Lock Manager* implémentant 16 verrous. La première ligne du tableau expose les résultats de synthèse d'une grappe de la plateforme TSAR. Ce module avait été synthétisé dans la technologie 28 nm FDSOI, flot de synthèse qui n'était plus utilisable au moment de la synthèse du *Lock Manager*. Par ailleurs, la synthèse d'une grappe en technologie 22 nm FDSOI n'est pas chose aisée en raison de la présence de bancs mémoires nécessitant un placement minutieux. L'objectif ici étant uniquement d'avoir un ordre de grandeur de l'impact du module *Lock Manager* sur la logique d'une grappe TSAR, nous n'avons pas re-synthétisé la grappe, mais nous nous sommes contenté d'appliquer un coefficient rectificateur, à savoir que la technologie 28 nm FDSOI représente un surcoût en surface d'environ 30% par rapport à la technologie 22 nm FDSOI. C'est ce coefficient que nous avons appliqué dans le tableau 6.8 pour comparer l'estimation de la surface d'une grappe TSAR hors mémoire en 28 nm FDSOI à celle du *Lock Manager*, dans la colonne « Surface (normalisée) ». En effet, les grappes TSAR intègrent des mémoires relativement grandes, nous avons donc décidé de les exclure de notre comparaison afin d'être plus équitables. Nous trouvons alors que la surface d'un *Lock Manager* composé de 16 verrous représente seulement ~2% de la surface de la logique d'une grappe TSAR.

Le nombre de cellules quant à lui ne varie pas en fonction de la technologie utilisée, et peut être directement comparé.

Cette évaluation met en avant le faible coût matériel résultant de l'ajout d'un module *Lock Manager* dans une grappe TSAR. Aussi, l'implémentation de la solution *Lockality* reste envisageable et fortement attractive au vue de son faible impact matériel compte tenu des gains en performances générés.

6.5 Conclusion : limites et perspectives

Au cours de ce chapitre, nous avons présenté notre solution innovante de gestion non centralisée de verrous mobiles pour MPSoC.

Notre proposition repose sur l'étude du comportement des verrous sur des plateformes MPSoC. Grâce à l'exploitation des caractéristiques particulières des MPSoC modernes (grappe, réseau sur puce), cette solution propose un renouveau du mécanisme de verrou, et permet d'apporter des éléments de réponse à la question soulevée dans le chapitre 2, à savoir : Est-il possible de tirer profit de la propriété d'organisation en grappe des architectures MPSoC lors de l'établissement des synchronisations inter-processus ?
L'implémentation de notre solution dans le MPSoC TSAR, nous a permis d'évaluer en pratique le gain généré par celle-ci. Nous avons alors comparé la solution *Lockality* à la solution classique *Pthread*, et cela à trois niveaux différents.

Nous avons premièrement mis en avant le fait que le mécanisme permet de réduire de quelques dizaines de cycles le temps d'acquisition physique d'un verrou, que ce dernier soit localisé dans la grappe locale ou dans une grappe distante.

Ensuite, nous avons montré que la solution *Lockality* réduit les temps de verrouillage et de déverrouillage des verrous de quelques centaines de cycles par rapport à la solution *Pthread*. Ce gain provient de la réduction du temps d'acquisition physique du verrou, mais aussi de l'implémentation minimaliste de notre bibliothèque d'exploitation du mécanisme.

Enfin, nous avons vu à travers deux applications réelles issues du benchmark SPLASH2 que notre solution permet de réduire le temps d'exécution globale d'applications utilisateur.

Au cours de ce chapitre, nous avons présenté une solution de mécanisme de verrou performante et facile d'utilisation grâce à une interface calquée sur celle de *Posix*. Cependant, deux limitations du système ont pu être identifiées.

La première concerne la migration de tâches. En effet, le protocole actuel ne permet pas la migration des tâches (processus) d'un cœur à un autre (ou plutôt d'un cœur associé à un *Lock Manager* à un autre cœur qui serait associé à un autre *Lock Manager*), et cela du fait que le relâchement d'un verrou doit être réalisé sur le *Lock Manager* qui a permis son acquisition. Afin de permettre de relâcher le verrou sur un *Lock Manager* différent de celui ayant permis son acquisition, des trames supplémentaires permettant d'informer le propriétaire du verrou que celui-ci a été libéré par un autre *Lock Manager* sont nécessaires. L'assignation d'un processus à un cœur étant chose courante lors de la recherche de performances pour l'exécution d'un programme parallèle, en raison du coût de migration, cette limitation semble tolérable, même si l'extension du protocole serait un plus incontestable.

Le seconde limitation est relative à la sécurité du système. Actuellement aucun mécanisme n'a été mis en place pour la sécurisation des verrous matériels offerts par le système *Lockality*. En effet, le système *Lockality* ne contrôle pas l'identifiant du processus réalisant des opérations sur un verrou. Aussi, nous pourrions imaginer qu'un logiciel malveillant puisse interférer, et nuire à l'utilisation des verrous *Lockality*. Afin de palier ce problème, le système *Lockality* doit être étendu de la manière suivante :

— L'identifiant du processus doit être envoyé avec chaque requête émise par un CPU vers un *Lock Manager*. En réalité, cet identifiant est déjà envoyé dans notre implémentation sur la plateforme TSAR (il s'agit d'une donnée présente dans le protocole de communication VCI utilisé [All01]).

— Lors de la création d'un verrou dans les tables des *Lock Managers*, l'identifiant du processus auquel le verrou est rattaché doit être stocké.

— Lorsqu'un *Lock Manager* reçoit une demande d'opération sur un verrou de la part d'un CPU, l'identifiant du processus contenu dans la requête est comparé à celui associé au verrou. Le résultat de cette comparaison conditionne le traitement de la requête.

L'ajout du service de sécurisation des verrous n'ajoute qu'une complexité minime sur le chemin critique d'acquisition d'un verrou, qui ne devrait pas excéder un seul cycle pendant lequel la comparaison des identifiants sera effectuée. Cette amélioration du système *Lockality* ne remet pas donc pas en cause les gains apportés par cette solution par rapport à une solution classique.

Les principales perspectives d'évolution de la solution proposée concernent la limitation du nombre de verrous matériels offerts. En effet, le système *Lockality* actuel définit statiquement à la synthèse une zone mémoire de taille fixe dédiée à la gestion d'un nombre déterminé de verrous. Afin de relâcher la contrainte quant à la définition statique du nombre de verrous matériels désirés, nous pourrions envisager de coupler notre *Lock Manager* avec la mémoire cache de la grappe. De cette manière, il serait possible d'allouer de manière dynamique une section de mémoire dédiée au verrou dans la mémoire cache de la grappe en fonction des besoins utilisateur.

Même s'il peut être bien dimensionné par rapport aux besoins d'une ou plusieurs applications, le nombre de verrous matériels offerts par *Lockality* sera toujours limité par la capacité mémoire. Par ailleurs, certains types d'utilisation de verrous se prêtent plus à l'utilisation de verrous mobiles que d'autres, et cette propriété est susceptible d'évoluer dans le temps. Aussi, nous pourrions combiner la solution *Lockality* avec un mécanisme de verrous adaptatifs. En exploitant le principe d'apprentissage, il serait alors possible de définir, à la volée, les verrous les plus enclins à être implémentés matériellement avec le système *Lockality*. Ainsi, en cas de limitation mémoire, les verrous pouvant le plus bénéficier de la solution mobile seraient gérés par le matériel, alors que les autres seraient gérés au niveau logiciel, et cela afin de garantir les meilleures performances au niveau applicatif.

Conclusion

Influencé par les besoins provenant du calcul haute performance, et plus particulièrement par l'univers des micro-serveurs, cette thèse visait l'amélioration des performances des mécanismes de synchronisation sur les MPSoC.

Nous avons pu voir au travers de notre étude bibliographique (chapitre 3) que les mécanismes de synchronisation ont déjà fait l'objet de nombreuses propositions d'optimisation depuis plus de 30 ans. Cependant, la récente multiplication du nombre des cœurs intégrés dans les MPSoC ainsi que l'évolution des architectures de ces systèmes redistribuent les cartes. L'objectif de ce travail était alors d'étudier les nouvelles opportunités d'optimisation offertes par ces récentes évolutions des MPSoC.

Cette démarche nous a permis de prendre conscience de l'importance de disposer d'informations temporelles précises sur chaque mécanisme que nous nous proposons d'optimiser. Nous avons alors été amenés à traiter préalablement cette problématique avant de pouvoir débuter l'optimisation des mécanismes de synchronisation.

Dans le chapitre 2 nous avions soulevé trois interrogations qui ont sous tendu nos travaux de recherche. Des éléments de réponses à ces questions ont été proposés tout au long de ce manuscrit, nous résumons ici les conclusions auxquelles nous avons pu aboutir.

Comment pouvons nous observer de manière précise le comportement d'applications logicielles complexes sur MPSoC ? Le chapitre 4 de ce manuscrit permettait de revenir sur une notion essentielle à l'étude d'un mécanisme logiciel, à savoir la notion d'observabilité. C'est à dire définir l'ensemble des informations/données dont nous souhaitons disposer, ainsi que la manière de les obtenir.

Nous avions alors défini une série d'exigences nécessaires à l'optimisation des mécanismes de synchronisation sur MPSoC. Nous présentions les deux stratégies actuellement disponibles pour obtenir des informations temporelles sur ce type de systèmes : l'instrumentation du code en exécution native, et l'extraction de données par canaux cachés sur les plateformes de simulation, ou par l'exploitation des modules de surveillance insérés dans certains processeurs industriels.

Cependant, aucune de ces stratégies ne répondant pleinement à nos exigences, nous avons proposé une chaine de mesure adaptée reposant sur l'émulation et le principe de co-émulation.

La chaine de mesure a été conçue afin de permettre l'évaluation temporelle non intrusive et précise (au cycle près) des mécanismes de synchronisation s'exécutant sur MPSoC. Constituée de deux phases (phase d'acquisition des données et phase de traitement des données), elle permet d'identifier avec précision les sources de ralentissement dans les mécanismes de synchronisation.

La phase d'acquisition est réalisée de manière simultanée à l'émulation. L'acquisition des données repose sur un module spécifique connecté à la plateforme émulée afin d'ex-

traire les données utiles à notre analyse. Cette phase exploite le principe de co-émulation afin d'envoyer les informations extraites vers un PC hôte en charge d'écrire les données reçues dans des fichiers de *logs*.

La phase de traitement des données est quant à elle réalisée une fois l'émulation terminée, sur la base des fichiers de *logs* produits par la phase d'acquisition. Cette phase consiste en l'utilisation d'une série d'outils conçus pour déduire les informations utiles des données extraites de la plateforme : pile d'appels aux fonctions, durée d'exécution d'un événement, taux de contention, ...

Grâce aux résultats générés par cette chaine de mesure, il devient possible d'identifier les comportements et les causes de ralentissement des mécanismes de synchronisation, permettant ainsi d'entrevoir les possibilités d'amélioration.

Par ailleurs, l'utilisation d'une plateforme d'émulation couplée à une politique minimaliste d'extraction de données permet le passage à l'échelle de cette chaine de mesure, mais elle permet surtout l'exécution de logiciels complexes : système d'exploitation (Linux), applications utilisateur, ... Grâce à cette propriété, il devient alors possible d'étudier les mécanismes de synchronisation dans un environnement d'exécution réaliste, permettant ainsi de cibler les problèmes rencontrés dans les applications réelles.

En conclusion de ce chapitre 4 nous avions mentionné la simplification de l'étape de configuration de la chaine de mesures en tant qu'amélioration utile.

Comment fournir un mécanisme permettant d'accélérer les applications utilisateurs (à haut niveau), tout en restant simple à mettre œuvre ? Nous avons vu dans ce manuscrit que le besoin de mécanismes de synchronisation résulte de la parallélisation des applications logicielles. Dans le chapitre 5, nous présentions les deux stratégies de parallélisation de code : une manuelle (typiquement *Pthread*) et une semi-automatique reposant sur des bibliothèques d'aide à la parallélisation du code telles que *OpenMP*.

Dans un tel contexte, nous énoncions l'attrait que revêt l'optimisation des couches logicielles, bibliothèque et système d'exploitation, afin de proposer une optimisation facile à mettre en œuvre.

Aussi, nous avons opté pour l'étude de la bibliothèque *GNU OpenMP* en raison de sa forte utilisation et de son aspect « logiciel libre ».

Bénéficiant de la chaine de mesure conçue, nous avons présenté dans le chapitre 5 l'étude et l'optimisation du mécanisme de barrière de synchronisation de la bibliothèque *GNU OpenMP*. L'étude menée cible les deux types d'attente offerte par cette bibliothèque à savoir l'attente active et passive.

L'analyse de la phase de relâchement des processus en attente active sur la barrière mettait en évidence un appel non nécessaire à une fonction de réveil des processus. Nous avons alors proposé une contre-mesure permettant de réaliser l'appel à cette fonction coûteuse en temps et en accès mémoire uniquement lorsque cela est réellement nécessaire. Cette amélioration, somme toute assez simple, permet de réaliser des gains en performance importants, de l'ordre de 80% de réduction du temps nécessaire pour le libération de 16 processus sur notre plateforme MPSoC TSAR.

En ce qui concerne l'attente passive, c'est au niveau du noyau Linux que notre chaine de mesure a permis de mettre en exergue un flot d'exécution sous-optimal impliquant une gestion coûteuse de listes chainées. La proposition d'optimisation formulée en pre-

mier lieu s'est vue fortement impactée par les problèmes de contention mémoire, affectant sérieusement ses performances. À la suite d'une mise au point itérative, nous avons finalement pu proposer une solution optimale réduisant au maximum le phénomène de contention, et permettant ainsi de réduire la phase de réveil de 64 processus d'environ 60%. Par ailleurs, la contention mémoire étant la pierre d'achoppement de l'optimisation proposée, nous avons pu évaluer ce phénomène, et proposer quelques pistes d'optimisation afin de la réduire.

En conclusion, cette chaine de mesure qui permet d'étudier les mécanismes de synchronisation de manière précises dans un environnement réaliste, a rendu possible la proposition d'optimisations ciblant les problèmes réellement rencontrés par ces mécanismes lors de leur utilisation par des programmes utilisateur s'exécutant au dessus d'un système d'exploitation. Ce type d'optimisation traitant les sources réelles de ralentissement offre des gains en performance significatifs, tout en restant simple à mettre en œuvre car une simple mise à jour des couches logicielles (bibliothèque ou système d'exploitation) permet de bénéficier de l'amélioration, sans que l'utilisateur ait besoin de modifier son application.

Est-il possible de tirer profit de la propriété d'organisation en grappe des architectures MPSoC lors de l'établissement des synchronisations inter-processus? Le chapitre 6 présentait une solution innovante de verrou de synchronisation mobile. Orienté par des hypothèses formulées dans des articles académiques, nous décidions d'étudier grâce à notre chaine de mesure les comportements des verrous de synchronisation sur les MPSoC, et notamment leur taux de réutilisation par le même cœur, ou la même grappe. Les résultats obtenus confirmant le fort taux de réutilisation, nous proposions une solution matérielle permettant de relocaliser automatiquement les verrous dans la mémoire de la grappe en ayant obtenu l'accès. La stratégie de gestion non centralisée des verrous imaginée rend possible le passage à l'échelle de cette solution.

La comparaison de la solution proposée avec une solution standard de type *Pthread* présentait des résultats intéressants, avec une réduction de plusieurs dizaines de cycles pour les accès locaux comme distants (c'est à dire lors d'un déplacements dans le cas de notre solution *Lockality*). Au niveau applicatif, la comparaison avec la solution classique *Pthread* nous a permis de mesurer des gains substantiels de plusieurs milliers de cycles, suite à l'utilisation de 16 verrous *Lockality* lors de l'exécution de deux applications du benchmark SPLASH2 (Cholesky et Water-Nsquared) sur une plateforme MPSoC TSAR composé de 16 cœurs.

La solution de verrous mobiles proposée permet alors de profiter de manière efficace de l'organisation sous forme de grappe des MPSoC actuels. Le déplacement et la gestion décentralisée des verrous permet la réduction du nombre de messages réseaux relatifs aux verrous, et diminue par cela les problèmes de contention.

En conclusion du chapitre 6 nous notions tout de même deux limitations à la solution proposée ainsi que des améliorations envisageables afin d'y remédier. La première limitation concernait la sécurité des verrous, car pour l'heure aucune vérification du processus effectuant des opérations sur un verrou n'est réalisée. La mémorisation de l'identifiant du processus créateur du verrou dans les modules matériels, suivie d'une vérification matérielle à chaque opération, devrait résoudre le problème. La seconde limitation implique l'évolution du protocole de communication entre les modules de gestion des verrous afin

de permettre la migration d'un processus ayant acquis un verrou *Lockality* d'un processeur à un autre.

Pour aller plus loin

Ce travail de thèse a permis de répondre à des questions soulevées par la recherche d'optimisation des mécanismes de synchronisation pour des plateformes MPSoC.

Néanmoins, nous avons réalisé, au cours de notre démarche, que des problématiques connexes à notre recherche ne pouvaient pas être dissociées de la quête d'optimisation des performances des MPSoC, et cela du fait que les performances d'une application utilisateur s'exécutant sur ce type de système dépend de nombreux paramètres indissociables. La modification du temps passé dans un mécanisme de synchronisation implique des changements dans le flot d'exécution de l'application qui eux aussi vont influer sur les performances. Dans le chapitre 6 nous commencions déjà à aborder la question de ces effets de bord. En effet, la modification de la politique d'attribution d'un verrou, la réduction du temps passé dans un mécanisme de synchronisation, ou encore la réduction de la contention réseau entrainent la modification de l'ordre des accès mémoire. Celle-ci se traduit ensuite par une augmentation ou une diminution du nombre de défauts de cache affectant les performances de l'application dans un effet boule de neige. Dès lors, il devient impossible d'ignorer ces phénomènes qui empêchent qu'une réduction de x% du temps nécessaire à l'établissement d'une synchronisation entraine un gain systématique de $nombre\_utilisations \times x$% du temps d'exécution d'une application. En effet, la naïveté de cette affirmation a été illustrée dans le chapitre 5 avec l'optimisation de l'attente passive de la barrière de synchronisation.

Ainsi, nous ne prétendons pas que le travail mené au cours de cette thèse permet de garantir un gain chiffré sur tout type d'applications MPSoC, en effet, nous avons conscience d'avoir proposé des solutions traitant une partie des problèmes existant sans pour au tant englober l'ensemble des effets de bord engendrés. Cependant, nous n'envisageons pas notre étude, et notamment le système *Lockality*, comme une finalité en soi, mais plutôt comme une travail amont ouvrant de nouvelles perspectives d'optimisations tant logicielles que matérielles.

Ainsi, la chaine de mesure non intrusive pour MPSoC décrite au chapitre 4 conjuguée avec la méthodologie d'étude des mécanismes de synchronisation présentées dans le chapitre 5 ouvre des perspectives quant à l'optimisation poussée d'autres mécanismes bas niveau s'exécutant sur des plateformes MPSoC, et cela grâce à une compréhension fine des problèmes rencontrés par ces mécanismes, que ces problèmes proviennent d'implémentations logicielles sous-optimales ou de limitations matérielles.

La solution de verrous mobiles proposée, forte de sa gestion décentralisée, résout en partie la problématique du passage à l'échelle de ce type de mécanismes, et permet de réduire l'impact en nombre de messages sur le réseau. Exploitant le principe de localité spatiale des verrous, nous pouvons imaginer utiliser cette solution afin de favoriser de manière explicite les exécutions sur la même grappe ou le même cœur des sections critiques faisant appellent à un même jeu de données/instructions et cela afin de limiter les défauts de cache. Ainsi, il serait possible d'allier les avantages des mécanismes de délégation (*combining*) avec ceux des solutions à base de verrous.

Par ailleurs, en raison de la difficulté de prédictions des conditions d'utilisation des mécanismes de synchronisation, il nous semble intéressant pour de futur travaux de recherche d'étudier à nouveau, et de re-penser les mécanismes adaptatifs reposant sur le principe d'apprentissage. En effet, en raison des récentes recherches sur les réseaux neu-

ronaux, les systèmes d'apprentissage ont pu bénéficier de larges améliorations. Nous pensons qu'une utilisation judicieuse des techniques d'apprentissage plus rapide et moins coûteuses issues de ce domaine pourront bénéficier aux mécanismes de synchronisation afin de pouvoir ajuster, en cours d'exécution, la solution garantissant les meilleures performances.

Publications & Brevet

Publications dans des conférences internationales

— Maxime France-Pillois, Jérôme Martin, Frédéric Rousseau. Accurate MPSoC prototyping platform and methodology for the studying of the Linux synchronization barrier slowdown issues. 29th IEEE International Symposium on Rapid System Prototyping, (RSP 2018) IEEE

— Maxime France-Pillois, Jérôme Martin, Frédéric Rousseau. Linux synchronization barrier on MPSoC : hardware/software accurate study and optimization. Short paper. *International Conference on Application-specific Systems, Architectures and Processors (ASAP 2018)* IEEE

— Maxime France-Pillois, Jérôme Martin, Frédéric Rousseau. Optimization of the GNU OpenMP Synchronization Barrier in MPSoC. *International Conference on Architecture of Computing Systems (ARCS 2018)* Springer

Poster

— Maxime France-Pillois, Jérôme Martin, Frédéric Rousseau. Support matériel reposant sur les réseaux sur puce pour l'optimisation des synchronisations interprocessus dans les systèmes manycores. *Journées Nationales du Réseau Doctoral en Micro-nanoélectronique (JNRDM 2016)*

Brevet

— Maxime France-Pillois, Jérôme Martin, Frédéric Rousseau, Éric Guthmuller. Lockality, un verrou d'exclusion mutuelle mobile pour les architectures multi-cœurs. 2018

Bibliographie

[AFA11] J.L. Abellan, J. Fernandez, and M.E. Acacio. GLocks : Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 893–905, 2011.

[AFA12] J.L. Abellan, J. Fernandez, and M.E. Acacio. Efficient Hardware Barrier Synchronization in Many-Core CMPs. *IEEE Transactions on Parallel and Distributed Systems*, 23(8) :1453–1466, 2012.

[All01] VSI Alliance. Virtual Component Interface Standard, version 2, April 2001. http://home.mit.bme.hu/~feher/MSC_RA/VCI/VCI.pdf.

[ARM09] ARM. Arm synchronization primitives, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf.

[AS15] M. AbdElSalam and A. Salem. Soc verification platforms using hw emulation and co-modeling testbench technologies. In *2015 10th International Design Test Symposium (IDT)*, pages 14–19, Dec 2015.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel : A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[BG07] R. Buchmann and A. Greiner. A fully static scheduling approach for fast cycle accurate systemC simulation of MPSoCs. In *2007 Internatonal Conference on Microelectronics*, pages 101–104, 2007.

[cad] Cadence emulation platform. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-xp.html.

[cap] The Modified SPLASH-2 Home Page. http://www.capsl.udel.edu/splash/.

[CC11] Xiaowen Chen and Shuming Chen. DSBS : Distributed and Scalable Barrier Synchronization in Many-Core Network-on-Chips. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1030–1037, November 2011.

[CHB09] E. Cheung, H. Hsieh, and F. Balarin. Fast and accurate performance simulation of embedded software for MPSoC. In *2009 Asia and South Pacific Design Automation Conference*, pages 552–557, January 2009.

[CLJC10] Xiaowen Chen, Zhonghai Lu, A. Jantsch, and Shuming Chen. Handling shared variable synchronization in multi-core Network-on-Chips with distributed memory. In *SOC Conference (SOCC), 2010 IEEE International*, pages 467–472, September 2010.

[cod] Codelink. https://www.mentor.com/products/fv/codelink/.

[deb01] ARM On-Chip Debug Hardware, October 2001. http://www.electronicdesign.com/dsps/arm-chip-debug-hardware.

[EWSA10] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks : Lock Acquisition Scheduling for Self-aware Synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 215–224, New York, NY, USA, 2010. ACM.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks : Fast userlevel locking in linux. *AUUG Conference Proceedings*, page 85, 2002.

[gem18] gem5. gem5, 2018. http://gem5.org/.

[GJSS16] S. Gopikrishna, M. Jha, S. Sreekanth, and G. Savithri. A multiprocessor system on chip verification on hardware accelerator and software emulation. In *2016 IEEE International Conference on Advances in Electronics, Communication and Computer Technology (ICAECCT)*, pages 423–428, Dec 2016.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[kal18] kalray. kalray, 2018. http://www.kalrayinc.com/kalray/products/.

[KCK99] Chen-Chi Kuo, J. Carter, and R. Kuramkote. MP-LOCKs : replacing H/W synchronization primitives with message passing. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 284–288, January 1999.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 25–35, New York, NY, USA, 1994. ACM.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 272–285, New York, NY, USA, 1992. ACM.

[LDT+16] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Fast and portable locking for multicore architectures. *ACM Trans. Comput. Syst.*, 33(4) :13 :1–13 :62, January 2016.

[LFK+12] Lovic Gauthier, Farhad Mehdipour, Koji Inoue, Shinya Ueno, and Hiroshi Sasaki. Efficient barrier synchronization for 2d meshed NoC-based many-core processors. In *The 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*, Oita, Japan, March 2012.

[lls18] Load-link/store-conditional, 2018. https://en.wikipedia.org/wiki/Load-link/store-conditional.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1) :21–65, 1991.

[Men18] Mentor. Veloce2 Emulator, 2018. https://www.mentor.com/products/fv/emulation-systems/veloce.

[MPSV06] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient Synchronization for Embedded On-Chip Multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(10) :1049–1062, October 2006.

[nas] NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[NBT+05] M. D. Nava, P. Blouet, P. Teninge, M. Coppola, T. Ben-Ismail, S. Picchiottino, and R. Wilson. An open platform for developing multiprocessor SoCs. *Computer*, 38(7) :60–67, July 2005.

[RBS12] J.H. Rutgers, M.J.G. Bekooij, and G.J.M. Smit. An efficient asymmetric distributed lock for embedded multiprocessor systems. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 176–182, July 2012.

[RNPL15] Andrey Rodchenko, Andy Nisbet, Antoniu Pop, and Mikel Luján. Effective barrier synchronization on intel xeon phi coprocessor. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015 : Parallel Processing*, pages 588–600, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[RSS+18] Sven Rheindt, Andreas Schenk, Akshay Srivatsa, Thomas Wild, and Andreas Herkersdorf. Cacao : Complex and compositional atomic operations for noc-based manycore platforms. In Mladen Berekovic, Rainer Buchty, Heiko Hamann, Dirk Koch, and Thilo Pionteck, editors, *Architecture of Computing Systems – ARCS 2018*, pages 139–152, Cham, 2018. Springer International Publishing.

[Rup] Karl Rupp. 42 Years of Microprocessor Trend Data | Karl Rupp. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[Sin93] M. Singhal. A Taxonomy of Distributed Mutual Exclusion. *J. Parallel Distrib. Comput.*, 18(1) :94–101, May 1993.

[Son14] Stacey Son. splash2 : Splash 2 Benchmarks, October 2014. https://github.com/staceyson/splash2.

[SSH+15] T. Soga, H. Sasaki, T. Hirao, M. Kondo, and K. Inoue. A flexible hardware barrier mechanism for many-core processors. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pages 61–68, 2015.

[syn] Synopsys emulation platform. https://www.synopsys.com/verification/emulation.html.

[Sys18] SystemC. SystemC, 2018. http://accellera.org/downloads/standards/systemc.

[THL16] Y. L. Tseng, K. H. Huang, and B. C. C. Lai. Scalable mutli-layer barrier synchronization on NoC. In *2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, April 2016.

[Til12] Tilera Corporation. tile processor architecture overview for the tile-gx series release 1.1, May 2012. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf.

[TSA18] TSAR. Tsar, 2018. https://www-soc.lip6.fr/trac/tsar/wiki/AtomicOperations.

[vel] Veloce Emulation Platform. https://www.mentor.com/products/fv/emulation-systems/.

[Viv13] Lawrence Vivolo. Transaction-based Verification And Emulation Combine For Multi-megahertz Verification Performance, June 2013. http://www.electronicdesign.com/eda/transaction-based-verification-and-emulation-combine-multi-megahertz-verification-performance.

[VPS08] O. Villa, G. Palermo, and C. Silvano. Efficiency and scalability of barrier synchronization on NoC based many-core architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 81–90. ACM, 2008.

[WLSY15] Z. Wei, P. Liu, R. Sun, and R. Ying. TAB barrier : Hybrid barrier synchronization for NoC-based processors. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 409–412, 2015.

[WOT+95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs : characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[XWG+16] Hao Xiao, Ning Wu, Fen Ge, T. Isshiki, H. Kunieda, Jun Xu, and Yuangang Wang. Efficient Synchronization for Distributed Embedded Multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2) :779–783, 2016.

[ZPG+16] Chengyu Zheng, M. D. Preda, J. Granjal, S. Zanero, and F. Maggi. On-chip system call tracing : A feasibility study and open prototype. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 73–81, October 2016.

Résumé

La forte parallélisation des applications MPSoC accroit le besoin d'optimisation des mécanismes de synchronisation, primordiaux pour l'échange sûr d'informations entre processus. En effet, les délais qu'ils introduisent impactent les performances globales des MPSoC. L'objet de cette thèse est d'étudier puis d'optimiser les performances temporelles de ces mécanismes de synchronisation.

La complexité croissante des MPSoC impose l'étude précise des mécanismes ciblés dans un environnement réaliste mettant en exergue les spécificités logicielles et matérielles. Les outils de mesures disponibles ne répondant pas à nos exigences de précision conjuguée à la vitesse d'analyse, nous avons conçu notre propre chaine de mesure non intrusive reposant sur une plateforme d'émulation.

Appliquée à l'étude de l'implémentation *GNU* du mécanisme de barrière de synchronisation offert par la bibliothèque d'aide à la parallélisation de code *OpenMP*, notre chaine de mesure a mis en évidence deux faiblesses d'implémentation, aboutissant à la mise en place d'optimisations logicielles et matérielles réduisant de manière significative les délais de ce mécanisme.

La chaine de mesure développée nous a également permis de vérifier une hypothèse structurante pour l'optimisation : un verrou, bien qu'utilisé par plusieurs cœurs de différentes grappes au cours de l'application, est très souvent repris par le dernier cœur l'ayant libéré. Sur la base de ce constat, nous proposons une solution innovante assurant, de manière totalement décentralisée, la relocalisation dynamique des verrous dans la mémoire proche du cœur ayant obtenu l'accès. Cela permet de réduire la latence d'accès et le trafic réseau lors de la réutilisation d'un verrou par une même grappe.

Abstract

High parallelism of MPSoC applications increase the need of optimization for the synchronization mechanisms, essential to ensure consistent data exchanges between threads. Delays inserted by them impact the whole performances of the system. This thesis work aims to analyze and reduce delays of synchronization mechanisms for MPSoC architectures.

The growing complexity of MPSoCs requires assessment of proposed optimizations against hardware and software specifics in real-life environment. Since usual tools to perform measurements do not fulfill required accuracy with sufficient evaluation speed, we have designed a custom non-intrusive tool-chain based on an emulation platform.

The study of the *GNU* OpenMP library implementation of the synchronization barriers, carried out with our tool-chain, has revealed two weaknesses. Our proposed hardware and software optimizations achieve significant reduction of the delays introduced by the synchronization barrier.

The designed tool-chain has also allowed us to confirm a fundamental hypothesis for the optimization of the lock mechanism : although during the run time a lock may be used by various cores belonging to different clusters, it is often reused by the last core which has released it. Based on this observation, we propose an innovative decentralized solution to manage dynamic re-homing of locks in memory close to the last access-granted core, thus reducing access latency and network traffic in case of reuse of the lock by the same cluster.

ISBN : 978-2-11-129248-2