PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Abstraction de la synchronisation dans une stratégie de co-design logiciel/matériel sur une plateforme (SOC) multiprocesseur
Auteur: Author:	Patrick Samson
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Samson, P. (2006). Abstraction de la synchronisation dans une stratégie de co-design logiciel/matériel sur une plateforme (SOC) multiprocesseur [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7831/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7831/
Directeurs de recherche: Advisors:	Guy Bois, & El Mostapha Aboulhamid
Programme: Program:	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

ABSTRACTION DE LA SYNCHRONISATION DANS UNE STRATÉGIE DE CO-DESIGN LOGICIEL/MATÉRIEL SUR UNE PLATEFORME (SOC) MULTIPROCESSEUR

PATRICK SAMSON
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2006

© Patrick Samson, 2006.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-25574-2
Our file Notre référence
ISBN: 978-0-494-25574-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

ABSTRACTION DE LA SYNCHRONISATION DANS UNE STRATÉGIE DE CO-DESIGN LOGICIEL/MATÉRIEL SUR UNE PLATEFORME (SOC) MULTIPROCESSEUR

présenté par: SAMSON Patrick

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

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

M. ROY Robert, Ph.D., président

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. ABOULHAMID Mostapha, Ph.D., membre et codirecteur de recherche

MME. NICOLESCU Gabriela, Doct., membre

Un condensé des leçons apprises au cours des présents travaux de recherche.

« The road of life twists and turns and no two directions are ever the same. Yet our lessons come from the journey, not the destination. » - Don Williams, Jr.

« The greatest danger in modern technology isn't that machines will begin to think like people, but that people will begin to think like machines. » - Anonyme

REMERCIEMENTS

Il convient d'abord de remercier les gens qui m'ont offert un support remarquable tout au long de ma maîtrise: ma famille proche et ma copine. Mario, Ginette et Karine, sans vos encouragements et votre aide, cette maîtrise n'aurait peut-être jamais vue le jour. Il va de soi qu'une présence régulière offre une contribution marquée dont l'essence perdure au fil des années. C'est pourquoi, dans le même ordre d'idée, je tiens à remercier mes confrères de travail. Ces partenaires de recherche et amis ont toujours su apporter conseils, support moral et divertissements dans les moments de gloire et dans les moments plus difficiles. Merci à Simon, Jean-François, François, Francis, Mortimer et au CIRCUS.

Je ne peux poursuivre avant de remercier mon directeur et mon co-directeur de recherche, Guy Bois et Mustapha Alboulhamid. Leur expérience et leurs judicieux conseils m'ont permis d'amorcer mon projet dans une avenue novatrice et de lui donner l'orientation qu'il connaît aujourd'hui.

Un remerciement final s'applique aussi au personnel technique (Alex) et administratif qui a facilité mon travail, mon développement et mon apprentissage durant mon séjour à l'École Polytechnique de Montréal.

RÉSUMÉ

Un nombre croissant de systèmes embarqués sont aujourd'hui implantés sur une même puce électronique. De tels systèmes prennent alors l'appellation systèmes-sur-puce ou SoC. La conception et l'implantation de SoC reposent sur les technologies des semi-conducteurs qui permettent des niveaux d'intégration en constante croissance. Bien que ces systèmes soient plus compacts, consomment une faible puissance et offrent des performances inégalées en comparaison à leurs ancêtres, ils sont plus complexes à concevoir. Cette augmentation de la complexité est amplifiée par l'avènement des MPSoCs, les systèmes-sur-puce multiprocesseurs.

La plateforme de conception SPACE cherche à faciliter la conception et l'implantation de systèmes-sur-puce. SPACE opère à un niveau d'abstraction supérieur au niveau RTL plus conventionnel: le niveau système (ESL). SPACE repose sur l'assemblage de composants IP afin de créer un système électronique. Dans l'ensemble, SPACE se veut une solution plus rapide et plus économique à la création de SoCs performants. SPACE cherche à étendre ses capacités afin de supporter la création de MPSoCs. À ce titre, les plateformes multiprocesseurs requièrent des fonctionnalités additionnelles. La bibliothèque SISSMA développée dans le cadre de cette recherche intègre des services de synchronisation pour MPSoC dans SPACE.

La bibliothèque SISSMA sert d'abord d'analyse initiale à l'intégration de supports de synchronisation génériques, indépendants de la plateforme matérielle utilisée. Son attrait tient aussi du fait qu'elle est aisément configurable afin de tenir compte d'un facteur de conception propre à l'ESL: le partitionnement logiciel/matériel des tâches du système. Ainsi, fidèle à la méthodologie de conception au niveau système et à la plateforme SPACE, une application n'a pas à être personnalisée suite à son partitionnement logiciel/matériel, puisque SISSMA assure les changements grâce à ses options de configuration. Enfin, il est montré que les performances de la bibliothèque SISSMA sont négligeables lors de synchronisations dites locales. Dans

le cas de synchronisations dites à distance, elle requiert moins de deux fois le temps d'exécution requis par une synchronisation faite par le système d'exploitation temps-réel MicroC/OS-II.

ABSTRACT

A growing number of embedded systems today are manufactured on a single electronic chip. In such a case, a system is called a system-on-chip or SoC. The design and implementation of SoCs rely on semiconductor technologies which nowadays increase in terms of integration levels. Considering that such systems are more compact, operate with lower power and offer unprecedented performances in comparison to their ancestors, they are more complex to design. This increase in complexity is amplified by the introduction of MPSoCs, the multiprocessor systems-on-chip.

SPACE, a design platform, is meant to facilitate the design and implementation of systems-on-chip. SPACE operates at an abstraction level higher than the more conventional RTL level: the electronic system level (ESL). SPACE relies on IP component assembly in order to create an electronic system. Overall, SPACE offers a faster and cheaper solution to creating efficient SoCs.

SPACE seeks to expand its services in order to support the creation of MPSoCs. Multiprocessor platforms require additional functionalities. The SISSMA library developed in this research integrates synchronization services for MPSoC into SPACE. The SISSMA library first serves as an initial analysis to the integration of generic synchronization support, independent of the hardware platform used. Its configurability makes it attractive in considering a key factor of ESL design: hardware/software partitioning of system tasks. Hence, in complete harmony with electronic system level methodology and with the SPACE platform, an application does not require custom tuning following its hardware/software partitioning since SISSMA ensure the changes through its configuration options. To conclude, it is demonstrated that the performance overhead of the SISSMA library are negligible during local synchronizations. For remote synchronizations, they demand less than twice the synchronization time required by the MicroC/OS-II real-time operating system.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ACRONYMES

API	**A**pplication **P**rogrammer **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASIP	**A**pplication **S**pecific **I**nstruction-set **P**rocessor
BRAM	**B**lock **R**andom **A**ccess **M**emory
BCA	**B**us **C**ycle **A**ccurate
CC-NUMA	**C**ache **C**oherent **N**onuniform **M**emory **A**rchitecture
CMP	**C**hip **M**ulti**P**rocessors
CAF	**C**o-**A**rray **F**ortran
CAS	**C**ompare-**A**nd-**S**wap
CLB	**C**onfigurable **L**ogic **B**lock
CMOS	**C**omplementary **M**etal **O**xide **S**emiconductor
DCM	**D**igital **C**lock **M**anager
DSP	**D**igital **S**ignal **P**rocessor
E/S	**E**ntrée/**S**ortie
EDA	**E**lectronic **D**esign **A**utomation
ESL	**E**lectronic **S**ystem **L**evel
FF	**F**lip-**F**lop
FIFO	**F**irst-**I**n-**F**irst-**O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HAL	**H**ardware **A**bstraction **L**ayer
IC	**I**ntegrated **C**ircuit
IDE	**I**ntegrated **D**evelopment **E**nvironement
I/F	**I**nterface
IP	**I**ntellectual **P**roperty
IPCP	**I**mmediate **P**riority **C**eiling **P**rotocol

IPIF	**IP** InterFace
LMB	**L**ocal **M**emory **B**us
LUT	**L**ook **U**p **T**able
OS	**O**perating **S**ystem
MCS	**M**ellor-**C**rummey et **S**cott
MPI	**M**essage **P**assing **I**nterface
MPSoC	**M**ulti**P**rocessor **S**ystem-**O**n-**C**hip
NOC	**N**etwork-**o**n-**C**hip
NUCA	**N**o**n**uniform **C**ommunication **A**rchitecture
NUMA	**N**o**n**uniform **M**emory **A**rchitecture
OPB	**O**n-chip **P**eripheral **B**us
QOSB	**Q**ueue **O**n **S**ync **B**it
PA	**P**in **A**ccurate
RAM	**R**andom **A**ccess **M**emory
RoC	**R**otator **O**n **C**hip
RTU	**R**eal **T**ime **U**nit
SISSMA	**S**PACE eng**I**ne for **S**ynchronization of **S**ystemC **M**ultiprocessor/ Multithreaded **A**pplications
SLD	**S**ystem **L**evel **D**esign
SLIC	**S**ystem **L**ink and **I**nterrupt **C**ontroller
SMP	**S**ymmetric **M**ulti**P**rocessing
SMT	**S**imultaneous **M**ulti**T**hreading
SoC	**S**ystem-**O**n-**C**hip
SoCLC	**S**ystem-**O**n-**C**hip **L**ock **C**ache
SoCSU	**S**ystem-**O**n-**C**hip **S**ynchronization **U**nit
SPIN	**S**calable **P**rogrammable **I**nterconnection **N**etwork

SPMD	**S**ingle **P**rogram **M**ultiple **D**ata
SMT	**S**imultaneous **M**ultithreading
RTOS	**R**eal **T**ime **O**perating **S**ystem
TAS	**T**est-**a**nd-**S**et
TATAS	**T**est-**a**nd-**T**est-**a**nd-**S**et
TF	**T**imed **F**unctionnal
TLM	**T**ransactional **L**evel **M**odel
TTL	**T**ask **T**ransaction **L**evel **I**nterface
UPC	**U**nified **P**arallel **C**
UTF	**UnT**imed **F**unctional
VLIW	**V**ery **L**ong **I**nstruction **W**ord
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
UPC	**U**nified **P**arallel **C**

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

Cette section sert d'introduction au thème des systèmes-sur-puce. Il traite des aspects impliqués dans leur conception et des méthodes prédominantes de conception et de raffinements de ces derniers.

1.1 Les systèmes embarqués d'aujourd'hui

Les systèmes embarqués sont omniprésents dans notre environnement. On les retrouve dans une multitude de domaines d'application, autant dans notre environnement quotidien (micro-onde, machine à laver, thermostats numériques, téléphone cellulaire) qu'au travail et dans les transports (assistant numérique personnel, lecteur de musique MP3 portatif, système de GPS portatif). De nos jours, une automobile contient en moyenne de 20 à 80 ordinateurs [5]. Dans cette optique, il n'est pas surprenant de constater que plus de 90 % des microprocesseurs vendus sur le marché, incluant les microcontrôleurs, soient destinés à des applications embarquées [6], plutôt qu'à des ordinateurs de travail (PC) ou des serveurs.

Le marché de l'électronique impose de constantes pressions aux concepteurs de systèmes embarqués. Les systèmes à concevoir doivent fournir un nombre croissant de fonctionnalités tout en offrant des performances améliorées. Il suffit d'observer l'évolution d'un système embarqué très populaire, le téléphone cellulaire. Les téléphones cellulaires ont non seulement diminué de format de génération en génération, mais offrent constamment plus de services, au-delà de la fonction d'émettre ou de recevoir des appels téléphoniques: carnet d'adresses, mini plateformes

de jeux, lecteur de musique numérique, prise de photos et enregistrement de vidéos numériques, etc.

Il est important de noter qu'une telle demande a un impact marqué sur la complexité des systèmes embarqués. Les systèmes embarqués de dernières générations comportent maintenant plusieurs processeurs, souvent de nature hétérogène, et de la logique personnalisée en plus de devoir répondre à des contraintes de performance. Alors que le terme performance faisait jadis référence au temps d'exécution d'une application ou à son débit, il prend désormais une toute autre signification. Malgré que ce dernier aspect de la performance soit toujours présent, d'autres qualificatifs sont venus s'y greffer. Les termes puissance et énergie font aujourd'hui partie de la définition de performance. Il n'est pas rare de constater que les systèmes embarqués opèrent à partir de batteries. Il est donc requis qu'ils performent correctement sur une longue période de temps sans demander le remplacement constant des piles utilisées.

Dans le même ordre d'idée, la dissipation de puissance reliée à un système embarqué prend de plus en plus d'importance. Cette dissipation d'énergie affecte directement le coût du système à concevoir [7], car il requiert l'utilisation d'un système de refroidissement et l'encapsulation du circuit intégré.

Il y a un besoin, dans le domaine de la conception de systèmes embarqués, pour de nouveaux outils. Ces outils devront nécessairement attaquer de face les complexités identifiées précédemment. Plus particulièrement, une des problématiques actuelles se situe au niveau de la productivité des méthodes utilisées. Essentiellement, les méthodes de conception utilisées depuis les dix dernières années ne parviennent plus à soutenir la cadence requise.

Bref, la conception et la fabrication de systèmes embarqués est un monde complexe où interagissent plusieurs domaines d'application: informatique, microélectronique,

réseautique et télécommunication, pour n'en nommer que quelques-uns.

1.1.1 Loi de Moore

Cette loi, qui représente plutôt une observation résultant de données empiriques, fut formulée en avril 1965 par Gordon E. Moore, co-fondateur de la compagnie Intel. Dans sa formulation la plus connue, la loi fait état de l'avancement de la technologie et constate que le nombre de transistors des circuits intégrés double tous les 18 mois. Il est indiqué de mentionner que la formulation désirée de la loi fait référence à une période de 24 mois, plutôt qu'à 18, avant que le nombre de transistors ne double [8].

La Loi de Moore, bien qu'elle se soit révélée correcte dans les années suivant sa formulation, est en fait devenue un objectif de performance pour l'ensemble de l'industrie de la microélectronique. Effectivement, l'industrie de semi-conducteurs s'est fixée comme objectif de pouvoir suivre cette tendance formulée à travers cette loi.

Toutefois, bien que les technologies de fabrication des circuits intégrés rendent disponibles tous ces transistors, il en résulte que les concepteurs ne parviennent pas à suivre cette cadence exposée dans la Loi de Moore. Bref, les systèmes à concevoir augmentent en termes de contenu (quantité de transistors) et en termes de performance (puissance, bande passante, latence, etc.). Par contre, les méthodes courantes de conception et de vérification ne permettent pas une telle progression. En somme, il en résulte la tendance montrée à la Figure 1.1.

Ainsi, il est possible d'observer que chaque nouvelle génération de technologie des semi-conducteurs creuse le fossé entre la productivité des concepteurs et le potentiel offert au niveau matériel par tous ces transistors. Un premier besoin se fait entendre afin de disposer de nouveaux outils permettant d'accroître la productivité des

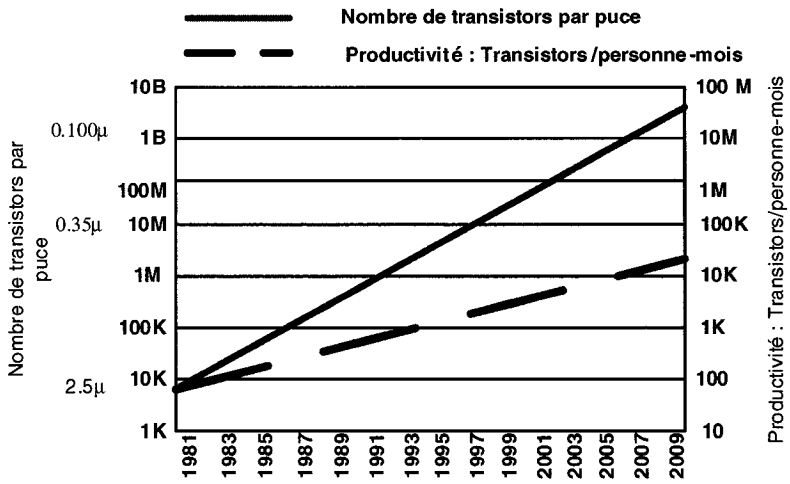


Figure 1.1 Écart de la productivité

concepteurs. À titre d'exemple, aucun design, basé sur la technologie 90-nanomètres, n'excédait 50 millions de transistors d'utilisation, alors qu'ils disposaient du double de ressources [9].

1.1.2 Loi de Wirth

Dans un même ordre d'idée, il n'y a pas que la productivité dans le domaine matériel qui élargit ce fossé. Il apparaît que la conception logicielle requiert elle aussi un gain de productivité. Le domaine de la conception logicielle a fort à gagner à utiliser de nouvelles stratégies de vérification et de génération de code. D'une manière analogue à la Loi de Moore, Wirth a formulé en 1995 sa loi exposant une partie du problème dans le domaine logiciel. Il affirme ainsi que le « logiciel ralentit plus rapidement que le matériel accélère » [10]. Bref, Wirth remarque que bien que le matériel devienne de plus en plus rapide, cela ne signifie pas que le travail à accomplir se fait plus rapidement. En fait, les programmes conçus croissent et deviennent de plus en plus complexes, mais finalement, requièrent le même temps pour s'exécuter.

1.1.3 Loi de Rock

Cet aspect touchant le développement logiciel prend de l'ampleur lorsque l'on considère l'aspect économique de la fabrication de circuits intégrés. Effectivement, chaque nouvelle technologie des semi-conducteurs est plus coûteuse que la précédente. Arthur Rock, de par sa Loi (« Rock's Law »), arrive aux mêmes conclusions: « le coût de fabrication d'une nouvelle fabrique de semi-conducteurs double à tous les 4 ans » [11].

En somme, il est nécessaire d'adapter les méthodes de conception actuelles, tant matérielles que logicielles, afin de pallier ces difficultés.

Bien que différentes embûches se présentent afin de mieux exploiter les ressources offertes par l'industrie des semi-conducteurs, fidèles jusqu'à maintenant à la Loi de Moore, des systèmes des plus en plus compacts et performants sont produits. L'ère des systèmes-sur-puce débute.

1.2 Problématique

Les concepteurs de systèmes électroniques font face à plusieurs défis de taille. Non seulement sont-ils d'abord soumis à de fortes pressions afin de produire des systèmes plus avancés et plus complexes à chaque génération d'un produit, mais ils doivent accomplir cette tâche avec l'aide d'outils qui n'offrent pas la productivité nécessaire. À cette complexité propre au matériel s'ajoute une vérification aussi complexe ainsi qu'un contenu logiciel qui augmente constamment avec le temps [5].

De surcroît, il est essentiel d'accélérer et de faciliter la conception et l'implantation de systèmes-sur-puce multiprocesseurs, des plateformes qui gagnent en popularité, mais qui sont, elles aussi, négligées par les outils de conception actuels. Les approches

utilisées présentement parviennent de peine et de misère à développer de telles solutions. Leur contenu logiciel demande plusieurs milliers de lignes de code. Ce développement logiciel à bas niveau d'abstraction, où tous les détails sont pris en compte, demande beaucoup de temps [12]. La situation est similaire au niveau du matériel: concevoir au niveau RTL en plus de s'assurer du bon fonctionnement d'une plateforme contenant plusieurs processeurs demande d'énormes ressources, tant temporelles que financières.

D'une manière similaire, la conception d'une application et de sa plateforme associée se doit d'être dissociée à part entière de son implantation finale. Ainsi, les méthodes actuelles doivent être abstraites afin d'éliminer du processus de conception les dépendances envers le partitionnement possible du système et de ses détails de bas niveau.

Globalement, les systèmes conçus sont bel et bien validés. Toutefois, cette validation a lieu une fois l'intégration de tous les composants complétée. Une telle approche ne peut suffire face aux courts temps d'accès au marché et au nombre croissant de fonctionnalités à valider [13].

La bibliothèque SystemC adresse plusieurs de ces problèmes. SystemC permet d'abstraire un grand nombre de détails propres à l'implantation d'un système-sur-puce, mais superflus quant à son design. La plateforme de conception SPACE [14], qui se base sur SystemC lui sert de complément tout en adressant des aspects négligés par SystemC. Ainsi, l'approche supportée par SPACE quant à la conception d'un produit électronique au niveau système vise la réduction des temps de conception et de vérification en plus de faciliter l'optimisation de ce dernier.

1.3 Objectifs

L'objectif principal de ce travail consiste à augmenter les fonctionnalités de la plateforme SPACE avec des services de synchronisation propices à une plateforme multiprocesseur. Ceux-ci se veulent complémentaires aux services de communication existants, qui reposent sur la transmission de messages. Ces services de synchronisation doivent être indépendants de la plateforme matérielle choisie tout en étant aisément configurables afin de ne pas restreindre les capacités de SPACE à déplacer des composants entre les couches logicielle et matérielle sans effort. Cette indépendance de la plateforme matérielle gagne en importance lorsque des composants, ne présentant pas des services particulièrement adaptés pour une solution multiprocesseur, telle l'utilisation de cache, cohérente ou pas, ou d'architecture mémoire particulière, sont utilisés afin de constituer une plateforme multiprocesseur. Cette importance implique que les services de synchronisation fournis doivent offrir un compromis satisfaisant entre flexibilité et performance.

1.4 Méthodologie

Pour débuter, une familiarisation avec la plateforme SPACE, sans oublier la bibliothèque SystemC, est de mise. Puisque SPACE repose sur le concept de modélisation au niveau système et de modélisation transactionnelle, il est essentiel d'être familier avec ces approches.

Par la suite, l'interface de programmation, à travers laquelle les services de synchronisation sont disponibles, sera conçue. Cette interface, bien que développée et implantée dans le langage C++ avec la complicité de SystemC, se doit d'être utilisable tant par des tâches logicielles que matérielles, tel que requis par la plateforme SPACE.

Ensuite, les mécanismes architecturaux génériques, référés sous le nom du moteur de synchronisation, seront conçus. La conception de ce bloc matériel comprendra deux étapes. En premier lieu, un modèle transactionnel (TLM) de ce dernier sera conçu et par la suite optimisé. Une fois le modèle optimisé au niveau transactionnel, une version synthétisable sera implantée avec l'aide du langage de description matérielle VHDL.

Enfin, l'ensemble des composants définis, à savoir l'interface de programmation ainsi que son support matériel générique, sera utilisé afin d'exécuter la multiplication de deux matrices. Pour ce faire, un système-sur-puce multiprocesseur (MPSoC) sera conçu et le tout exécutera à partir d'une plateforme matérielle basée sur une puce reprogrammable (FPGA).

1.5 Contributions

Ce travail a d'abord permis d'explorer la littérature existante quant aux méthodes de synchronisation, développées pour les systèmes multiprocesseurs, et des applications dans un contexte de conception au niveau système pour des systèmes-sur-puce. Ainsi, une interface de programmation générique est désormais disponible dans SPACE afin de synchroniser des tâches. Ces tâches peuvent être implantées en logiciel comme en matériel et s'exécuter sur un même processeur tout comme être partitionnées sur des processeurs indépendants.

Ces interfaces utilisent leurs propres mécanismes, basés sur un module matériel générique, si nécessaire, et ne dépend donc aucunement de l'architecture choisie pour le système-sur-puce implanté. L'interface permet par conséquent d'abstraire tous les détails de synchronisation facilitant la conception au niveau système et la portabilité du code source créé.

Les services de synchronisation offerts tiennent aussi compte des contraintes possibles en temps réel et offrent des optimisations favorisant de meilleures performances: différents mode de synchronisation et d'éveil, un ordonnancement FIFO aux ressources de synchronisation et concurrence.

De plus, cette interface ainsi que son support matériel est entièrement intégrée dans SPACE, permettant de nouvelles possibilités quant à la conception de plateforme multiprocesseur, pavant la voie à de futurs travaux sur la modélisation de plateformes multiprocesseurs.

Enfin, les travaux effectués offrent la base requise afin de pouvoir extraire des métriques sur le système simulé par rapport à la synchronisation entre ses composants.

1.6 Organisation de ce mémoire

Ce mémoire est constitué de cinq chapitres. Le présent chapitre introduit le monde des systèmes embarqués et de la conception de systèmes électroniques. Le second chapitre introduit les systèmes-sur-puce multiprocesseurs. Il présente aussi d'une manière plus générale tous les aspects propres à leur conception. Enfin, il conclut avec une introduction aux différentes méthodes de conception au niveau système. Le chapitre 3 constitue une revue de littérature sur les techniques de synchronisation développées pour des plateformes multiprocesseurs. L'intégration de ces protocoles de synchronisation dans un environnement de codesign logiciel/matériel est ensuite décrit dans le chapitre 4. Ce chapitre décrit les objectifs ainsi que la conception de la bibliothèque de synchronisation SISSMA, dédiée à la conception au niveau système dans la plateforme SPACE. Le chapitre 5 présente les performances de cette bibliothèque de synchronisation. Enfin, une conclusion résume les travaux de recherche accomplis et propose des travaux futurs.

CHAPITRE 2

SYSTÈME-SUR-PUCE: DÉFINITION, PROBLÉMATIQUE ET CONCEPTION

Les technologies de conception de circuits intégrés offrent des densités d'intégration en constante augmentation. Au rythme actuel, il sera possible, d'ici une dizaine d'années, d'intégrer un milliard de transistors sur une même puce d'une taille raisonnable [15]. Il est essentiel d'identifier un parallélisme afin de tirer profit de toutes ces ressources.

Traditionnellement et encore aujourd'hui, il relève du microprocesseur d'exploiter le parallélisme inhérent à un programme. Les microprocesseurs disposent de nombreux mécanismes afin d'optimiser l'utilisation parallèle de leurs ressources afin d'augmenter leur performance. Ainsi, ils tirent profit du parallélisme au niveau du jeu d'instruction (« Instruction Level Parallelism (ILP) »), du pipeline d'exécution ou de la prédiction de branche, pour nommer quelques techniques disponibles.

Toutefois, l'utilisation d'un seul fil d'exécution limite le parallélisme disponible pour de nombreuses applications. En fait, il devient de plus en plus coûteux, tant en complexité qu'en surface et en temps de conception ou de vérification, d'extraire ce parallélisme [15].

Depuis quelques années, les plateformes multiprocesseurs ont gagné en intérêt. Cet intérêt renouvelé pour de telles plateformes vient principalement des nouvelles technologies de la microélectronique, qui permettent leur intégration sur une même puce.

Le présent chapitre constitue une présentation générale des plateformes multipro-

cesseurs. Une courte description de leur début est d'abord donnée. Elle sera suivie par une description des différents types de plateformes multiprocesseurs ainsi que leurs distinctions. Dans un second volet, une revue littéraire de plusieurs plateformes multiprocesseurs commerciales est montrée.

2.1 Définitions: Soc et MPSoC

2.1.1 Système-sur-Puce (SoC)

Tout d'abord, il convient de définir ce qu'est un système-sur-puce, communément référé sous l'appellation « System-on-Chip » ou SoC dans la littérature. Un système-sur-puce est un circuit intégré qui implante la majorité ou l'ensemble des fonctions d'un système électronique complexe. De façon générale, un système-sur-puce est principalement constitué de logique numérique. Il peut contenir un processeur de jeu d'instructions, de la mémoire, des bus de communication, de la logique spécialisée ou d'autres fonctions numériques. Les systèmes-sur-puce se retrouvent dans différentes catégories de produits, tel le domaine de la télécommunication par leur utilisation dans les téléphones cellulaires [7].

Des besoins croissants en termes de performance ont amené les concepteurs à concevoir des systèmes-sur-puce multiprocesseurs.

2.1.2 Système-sur-Puce Multiprocesseur (MPSoC)

Les systèmes-sur-puce multiprocesseurs, ou « Multiprocessor System-on-Chip (MP-SoC) », constituent une simple extension au système-sur-puce. Ainsi, un système-sur-puce multiprocesseur est simplement un système-sur-puce qui utilise plusieurs

processeurs. Les MPSoCs sont complexes à concevoir en partie dû aux fonctions sophistiquées qu'ils implémentent et en partie à cause de la grande variété de technologies impliquées afin d'accomplir le travail [5].

Enfin, bien que les systèmes-sur-puce multiprocesseurs soient relativement récents, ce n'est pas le cas des systèmes multiprocesseurs. La section qui suit donne un bref historique des systèmes multiprocesseurs. L'expérience acquise au cours des 45 dernières années avec ces systèmes peut maintenant être transmise aux concepteurs de MPSoC.

Toutefois, bien que ce transfert de connaissance ne soit pas nouveau en tant que soi, il reste néanmoins que beaucoup de connaissances acquises au niveau des systèmes de traitement parallèle relèvent d'applications scientifiques, de bases de données ou d'autres applications qui diffèrent beaucoup des systèmes embarqués. Deux nouveaux aspects notables apportés par les systèmes-sur-puce multiprocesseurs sont [16]:

1. Les opérations en temps réel
2. Les opérations à faible consommation d'énergie et de puissance.

2.1.2.1 Origine des systèmes multiprocesseurs

Les systèmes multiprocesseurs ont fait leur apparition au début des années 1960 [17]. L'intérêt principal pour de tels systèmes émanait du fait qu'ils étaient fiables et redondants, qu'ils couvraient plusieurs gammes de prix et de performance, et ce, pour un nombre réduit de designs différents, et qu'ils supportaient des mises à jour incrémentales [17]. Toutefois, il fut déjà identifié à cette époque que tous ces avantages des systèmes multiprocesseurs étaient contrebalancés par un certain nombre d'inconvénients. Les systèmes multiprocesseurs demandent notamment des temps de conception allongés, leur extensibilité (« scalability ») est limitée et l'obsolescence

rapide de certains composants matériels limite l'application de mises à jour. De plus, un manque flagrant de supports logiciels, particulièrement au niveau des systèmes d'exploitation et des modèles de programmation, affecte négativement les systèmes multiprocesseurs par rapport aux systèmes monoprocesseurs existants [17].

Gorden Bell et Catharine Van Ingen présentent un résumé de l'évolution des systèmes multiprocesseurs, depuis leur tout début jusqu'en 1999, dans [17].

2.2 Complexité de la conception de systèmes-sur-puce au niveau système

La conception et l'implantation d'un système-sur-puce (SoC), multiprocesseur ou non, englobent de nombreux aspects. La section suivante identifie et donne une description de chacun de ces aspects. Cette section a pour objectif de donner une vue globale de ce qu'un tel design implique.

2.2.1 Synthèse logicielle et matérielle

Tout d'abord, tous les systèmes électroniques se composent d'une partie matérielle et d'une partie logicielle, aussi simple peut-elle être. Le processus de génération de ces deux couches porte le nom de synthèse logicielle et de synthèse matérielle. La génération logicielle implique la création de l'application qui s'exécutera sur le SoC. La génération matérielle implique la création de la plateforme qui supportera la couche logicielle. Il est essentiel de distinguer la synthèse matérielle du processus de synthèse numérique qui peut être utilisé pour générer automatiquement cette couche matérielle. Toutefois, la synthèse numérique n'est pas la seule avenue disponible pour y arriver. D'autres approches sont décrites dans la section 2.3.4, notamment l'utilisation de plateformes préconçues ou de plateformes constituées par l'assemblage

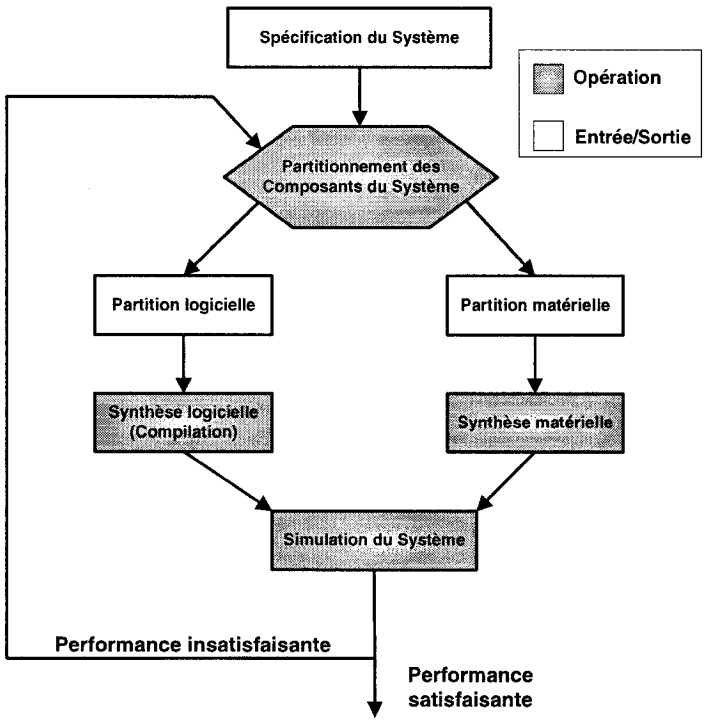


Figure 2.1 Processus de partitionnement logiciel/matériel

de composants existants.

2.2.2 Architecture système et partitionnement logiciel/matériel

L'étape de génération des couches logicielles et matérielles ne constitue que la conclusion du processus de conception et l'implantation d'un SoC. De prime abord, il est primordial de concevoir ce système afin qu'il réponde aux requis tant en termes de fonctionnalités que de performance.

Le partitionnement logiciel/matériel est défini comme le processus décisionnel à travers lequel l'ensemble des fonctionnalités du système est partagé entre les couches logicielles et matérielles. La Figure 2.1 illustre ce processus.

Tel qu'illustré, le processus de partitionnement logiciel/matériel cherche la solution optimale répondant aux contraintes du système. Les contraintes typiques à respecter peuvent être techniques (énergie requise, puissance dissipée, surface consommée, temps d'exécution, bande passante, consommation de mémoire) ou économiques. Économiquement, il est généralement moins dispendieux de développer des composants logiciels comparativement à leur équivalent matériel. Toutefois, ils sont aussi moins performants et requièrent l'utilisation d'un processeur adapté à leurs besoins (processeur d'usage général, microcontrôleur, processeur VLIW, DSP ou ASIP).

2.2.3 Performance des communications

Bien que le partitionnement logiciel/matériel identifie quels composants constitueront les couches logicielle et matérielle, il n'indique pas nécessairement comment ils y sont organisés. D'ailleurs, l'architecture d'un SoC identifie non seulement la nature des composants présents, mais surtout leurs interconnexions. Le réseau de communication qui unit tous les composants du SoC est un facteur déterminant de ses performances et de la surface requise pour son implantation. Traditionnellement, ce réseau reposait sur l'utilisation d'un bus de communication. Depuis quelques années, une variété de réseaux-sur-puce ont fait leur apparition. Il y a notamment les réseaux basés sur une topologie en anneau (Token Ring, Rotator-on-Chip [18][19]), ceux basés sur des arbres (SPIN [20]) ou encore ceux prenant une topologie en maille (Hot Potato [21]) ou le typique crossbar. Bref, la gestion des communications constitue un autre aspect important de la conception d'un SoC.

2.2.4 Synchronisation

Un aspect qui gagne en importance avec l'avènement des systèmes-sur-puce multiprocesseurs (MPSoC) est la synchronisation. La synchronisation englobe l'ensemble des aspects relatifs à l'exécution correcte de multiples fils d'exécution, tel dans une plateforme multiprocesseur ou avec une application multitâche, ainsi que la protection des ressources partagées. De plus, l'exécution des comportements désirés de l'application au moment approprié relève de la synchronisation.

2.2.5 La nature hétérogène des composants

La conception d'un SoC constitue l'agglomération de plusieurs composants, souvent de nature hétérogène, augmentant par le fait même la complexité de ce processus. Des exemples de SoC sont donnés à la section 2.3.4.3. Il est possible de remarquer que ces plateformes sont souvent constituées d'un processeur à usage général en plus d'un processeur spécifique (DSP, processeur réseau, accélérateurs spécifiques), sans oublier le support d'une gamme de périphériques distincts, des mémoires, un réseau de communication et des unités de traitement spécifiques (« custom logic »).

2.2.6 La pile logicielle

La couche logicielle d'un SoC peut prendre différentes saveurs en fonction des besoins requis. Ainsi, bien que cette couche semble n'abriter que l'application s'exécutant en coopération avec les services fournis dans la couche matérielle, celle-ci cache en fait une pile complète de services. Les principaux éléments de cette pile sont montrés à la Figure 2.2.

Typiquement, le premier niveau implante les comportements de l'application. Ils

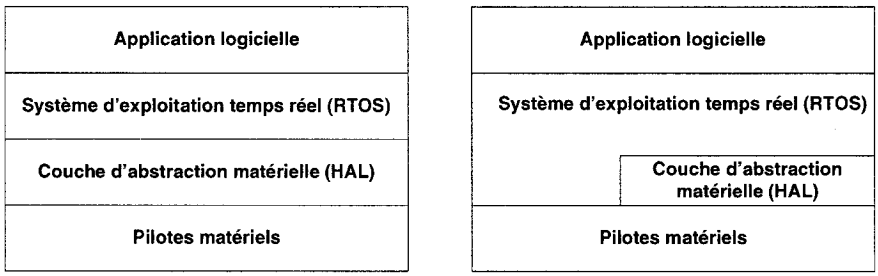


Figure 2.2 Les différents éléments de la couche logicielle

sont typiquement modélisés dans un langage de programmation tel le C/C++ ou même en assembleur. À l'autre extrême de cette pile, les pilotes (ou « drivers ») servent d'interface vers la couche matérielle. Les pilotes sont propres aux composants de la couche matérielle. Ils permettent l'interaction entre une application et un composant matériel. Par exemple, une minuterie, un contrôleur d'interruption, un contrôleur Ethernet et un UART possèdent chacun leur propre pilote spécifique à leurs fonctionnalités.

Bien que ces deux éléments de la pile logicielle peuvent répondre à tous les besoins d'une application destinée à un SoC, deux autres couches peuvent être présentes afin d'offrir des services additionnels.

Il est possible qu'une application requière les services d'un système d'exploitation (« Operating System, OS ») ou d'un système d'exploitation temps réel (« Real Time Operating System, RTOS »). D'une manière générale, le système d'exploitation est essentiellement un programme logiciel qui gère les ressources logicielles et matérielles d'un système. De la même façon, un système d'exploitation temps réel constitue une classe spécialisée des systèmes d'exploitation. Toutefois, ceux-ci sont destinés à des applications temps-réel. Bref, ils sont conçus afin d'être déterministes et ainsi répondre à des contraintes de temps dans des échéances prescrites. eCos [22], INTEGRITY [23], QNX [24], MicroC/OS-II [25], Nucleus [26], RTEMS [27], RTLinux

[28], ThreadX [29] et VxWorks [30] sont quelques exemples de systèmes d'exploitation temps réel.

La couche d'abstraction du matériel, communément appelée « Hardware Abstraction Layer » ou HAL, sert d'interface entre l'application ou le RTOS et les pilotes. Bref, cette couche permet une séparation entre les détails de bas niveau requis pour interagir avec le matériel et l'application logicielle [13]. Il est possible que cette couche fasse partie du système d'exploitation, ce qui explique les deux représentations de la Figure 2.2.

Les systèmes embarqués, nonobstant les SoC, étant des systèmes répondant à des besoins précis, il n'est pas inhabituel qu'une combinaison de ces différentes couches se retrouve dans la pile logicielle en fonction des besoins exprimés par l'application. Ces différentes couches favorisent la modularité et la réutilisation des éléments logiciels conçus. Il est à noter que la couche logicielle sert parfois de facteur différentiateur entre deux produits qui, sous toutes apparences, utilisent une même plateforme matérielle. La couche logicielle prend de plus en plus d'ampleur avec le gain de popularité des plateformes programmables, où un SoC peut contenir plusieurs millions de lignes de code [31].

Cette section a identifié les principales étapes présentes dans le processus de conception d'un système-sur-puce. La section suivante présente l'évolution des méthodologies de conception.

2.3 Conception de systèmes-sur-puce multiprocesseurs

Alors que l'intégration de multiples processeurs sur une même puce n'était qu'une tendance il y a quelques années [12], de tels systèmes font maintenant partie de la

réalité. Toutefois, les méthodes de conception traditionnelles se doivent d'évoluer au même rythme que les besoins qu'ils rencontrent. Les sections qui suivent présentent l'évolution des méthodes de conception de systèmes embarqués et de systèmes-sur-puce, à partir de l'approche plus traditionnelle de conception jusqu'à la conception au niveau système.

2.3.1 Approche traditionnelle à la conception

Le point de départ de tout système implique **la définition de ses requis et spécifications**. Cette première étape cible le fonctionnement du système: ses fonctionnalités et son mode d'opération. Une fois le système défini, les concepteurs traversent une phase d'**exploration architecturale**. Cette phase identifie, pour chaque module, s'il sera implanté dans la couche logicielle ou dans la couche matérielle du système. Le processus décisionnel de ce partitionnement logiciel/matériel repose sur les contraintes du système, à savoir la surface qu'il occupe, ses restrictions au niveau de la consommation d'énergie, sa puissance d'opération et les performances d'exécution qui lui sont imposées ou requises. Par la suite, une fois les spécifications définies et l'architecture choisie, une séparation se produit : **le développement de la couche logicielle et de la couche matérielle** débute de façon parallèle, mais séparée [32]. Néanmoins, il se trouve que les équipes logicielles et matérielles reposent sur la qualité et l'exactitude des spécifications initiales, souvent rédigées dans un langage informel.

La faiblesse principale d'une telle approche repose sur le fait que la décision du partitionnement logiciel/matériel se fait a priori, sans expérimentation concrète sur le système cible. De plus, la solution implantée, étant fortement dépendante de la partition choisie, il devient coûteux et complexe en temps de conception de changer de partitions par la suite [33]. Enfin, l'approche traditionnelle au partitionnement

consiste souvent à chercher une solution offrant le plus de contenu logiciel possible. Dans l'éventualité où les performances ne sont pas satisfaisantes, une partie des fonctionnalités est alors transférée en matériel.

Bref, l'approche traditionnelle de la conception peut tirer profit d'un environnement unifié où le logiciel et le matériel sont représentés. Cet environnement permet la vérification complète du système en plus de clairement démarquer la séparation logicielle/matérielle. Également, le choix préétabli de la partition mène à des designs sous-optimaux. La décision de partitionnement doit reposer sur des métriques réelles, vérifiables, applicables aux systèmes en cours de conception. Somme toute, un flot de conception mieux défini offrirait de meilleures garanties quant à la complexité du système à concevoir ainsi qu'au temps d'accès au marché.

2.3.2 Codesign logiciel/matériel

Devant les faiblesses de l'approche traditionnelle, le codesign logiciel/matériel a pris forme. Cette approche se veut une évolution de l'approche traditionnelle. Voici un bref aperçu de ce qu'est le codesign logiciel/matériel [34].

Les méthodologies de codesign cherchent d'abord à apporter une solution aux concepteurs pris avec des méthodes ad hoc quant au partitionnement logiciel/matériel de leur système. L'objectif du codesign logiciel/matériel est d'identifier un flot de conception dirigée permettant d'optimiser le processus de partitionnement logiciel/matériel et de réduire les coûts de conception et le temps d'accès au marché. Par le fait même, ce flot de conception évolué réduit les coûts directs du produit, facilite l'intégration du système et génère automatiquement les interfaces entre le logiciel et le matériel.

La Figure 2.3 illustre le flot de conception suggéré par la méthodologie de codesign logiciel/matériel.

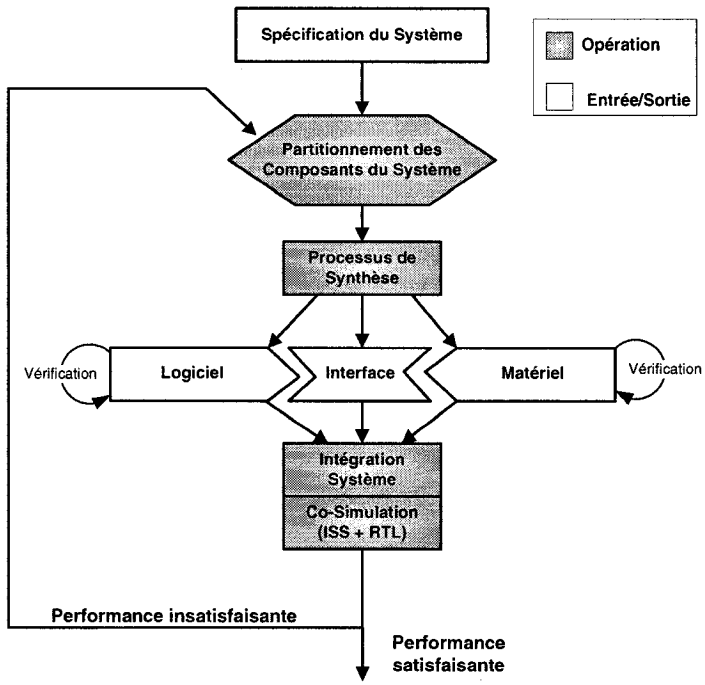


Figure 2.3 Flot de conception du codesign logiciel/matériel

Le point de départ reste le même : la spécification du système. Toutefois, une variante apparaît. La modélisation du système requiert un processus itératif plus efficace afin d'identifier le partitionnement logiciel/matériel optimal. Le codesign utilise un environnement intégré de cosimulation où le logiciel et le matériel interagissent. Cet environnement permet un retour plus rapide sur les décisions de la conception initiale du système comme suite aux informations obtenues durant la cosimulation.

La faiblesse principale de la méthodologie de codesign logiciel/matériel, telle que décrite plus tôt, tient d'abord du temps de développement qu'elle impose. La décision de partitionnement repose sur les métriques de performance obtenues lors de la cosimulation des composants logiciels et matériels. D'une part, les composants matériels sont modélisés dans un langage de description matériel (HDL) tel que VHDL ou Verilog. Parallèlement, les composants logiciels sont typiquement implantés dans

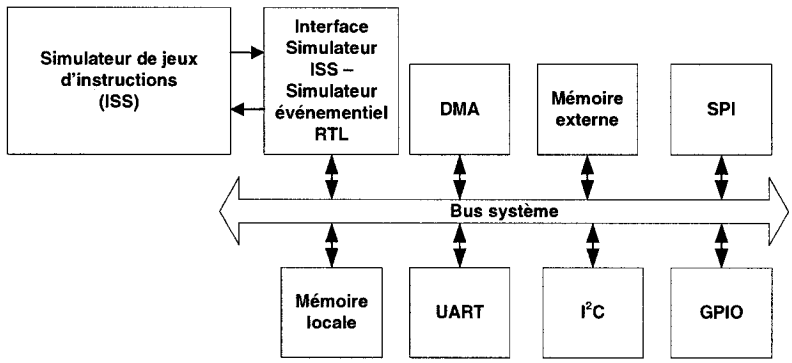


Figure 2.4 Configuration d'une co-simulation logicielle/matérielle entre un simulateur de jeux d'instructions et une partition matérielle

un langage de programmation tel C/C++ ou en assembleur. Durant la simulation, l'application logicielle s'exécute sur un simulateur de jeux d'instructions, ou ISS pour « Instruction Set Simulator ». Ce simulateur de jeux d'instructions interagit avec les blocs matériels afin d'émuler les comportements du système conçu. Cet arrangement est présenté à la Figure 2.4.

Bien qu'un tel environnement procure un niveau de précision très élevé, il requiert aussi un long temps de simulation. Effectivement, les blocs matériels, étant implantés au niveau RTL, ou « Register Transfer Level », et le simulateur de jeux d'instructions, offrant le même niveau de précision, prennent beaucoup de temps à s'exécuter dû à la grande quantité de détails disponibles durant la simulation du système. À cela, s'ajoute un temps de conception relativement long des composants du système qui sont implantés avec le maximum de détails et de précision. Bref, cela affecte directement le temps de développement, principalement en ce qui a trait au processus d'identification de la partition optimale.

La méthodologie de codesign logiciel/matériel contient tous les éléments pertinents à la conception d'un système électronique optimal. Toutefois, un nouvel intérêt se dessine pour une approche améliorée de cette méthodologie : la conception au niveau

système.

2.3.3 Conception au niveau système

La conception au niveau système, référée sous l'appellation ESL, ou « Electronic System Level Design », adresse le problème de productivité auquel les concepteurs font face dû à l'augmentation de la complexité des nouveaux processus des circuits intégrés. La conception au niveau système représente une avenue prometteuse dans le monde de l'EDA (« Electronic Design Automation ») [9]. À la base, la conception au niveau système modélise d'abord le système à un niveau d'abstraction plus élevé que le niveau RTL [35] afin d'obtenir un **prototype virtuel** de l'application.

En opérant à un niveau d'abstraction plus élevé, non seulement il est désormais possible d'accélérer la simulation du système en entier [36], mais il est aussi plus rapide de modéliser l'ensemble des composants du système [37], puisqu'il requiert moins de détails a priori, ceux-ci étant abstraits. Globalement, l'élévation du niveau d'abstraction se traduit par un cycle modélisation-partitionnement-simulation accélérée tout en limitant le compromis fait au niveau de la précision des opérations.

De plus, cette approche de conception permet la vérification au niveau système. Une telle assurance est de plus en plus nécessaire, considérant que la durée de la fenêtre d'accès au marché diminue et qu'une nouvelle fabrication d'une puce (« respin ») est très dispendieuse [38].

Ces modèles à hauts niveaux d'abstraction sont aussi profitables au développement parallèle du système. Alors qu'il n'était pas rare de voir le cycle de développement d'un système débuté par son implantation matérielle, puisque les développeurs logiciels dépendent de la plateforme cible dans leur tâche, le développement logiciel n'est plus retardé : un modèle exécutable de la plateforme matérielle est disponible.

De plus, l'utilisation d'interfaces entre les couches logicielle et matérielle favorise le développement parallèle de ces dernières [12]. Ces mêmes prototypes virtuels peuvent aussi servir de modèle de référence lors de la vérification de la solution finale : les modèles haut-niveaux et RTL se doivent de produire les mêmes résultats.

2.3.3.1 TLM

Une stratégie qui gagne en popularité afin d'élever le niveau d'abstraction des designs consiste à modéliser le système au niveau transactionnel. Ce niveau prend aussi le nom de niveau TLM, où TLM tient lieu de « Transactional Level Modeling ». Le TLM est complémentaire et fait partie de la conception au niveau système; ils ne sont pas orthogonaux.

Une transaction correspond à un échange de données entre deux composants ou à un événement. En fait, la synchronisation du système simulé se déroule à travers des transactions échangées par le biais d'un canal de communication abstrait. Ce canal sert de séparation entre le comportement des composants du système et la communication entre ces derniers. Ainsi, les comportements d'un composant ne se trouvent pas couplés à son interface ce qui favorise sa réutilisation.

L'objectif de la modélisation transactionnelle repose d'abord sur sa simplicité d'utilisation, puisque plusieurs détails sont abstraits dans une transaction. De plus, la simulation reposant sur moins d'événements, qui ont été groupés en une transaction, peut s'exécuter plus rapidement.

SystemC et SystemVerilog gagnent en popularité pour la conception au niveau système et transactionnelle.

SystemC

SystemC[1] [9][39] est une bibliothèque C++, et non un langage. SystemC permet la modélisation de composants matériels, tels les signaux et les événements concurrents, aussi retrouvés dans les langages de description matérielle (HDL). SystemC a l'avantage de disposer d'un standard IEEE 1666 [40] de modélisation au niveau transactionnel (TLM) qui uniformise ses pratiques d'utilisation, facilitant entre autres l'échange de noyaux IP. La bibliothèque SCV (« SystemC Verification ») est aussi disponible pour la vérification de modèles de systèmes et de composants électroniques.

Le lecteur intéressé est invité à consulter [41], un article qui présente les résultats d'un sondage quant à l'utilisation de SystemC en industrie.

SystemVerilog

SystemVerilog [42][43] est une extension au langage de description matérielle (HDL) Verilog. Il relève aussi du standard IEEE 1364 [9]. Verilog, tout comme son homologue VHDL, est d'abord conçu pour la modélisation de composants matériels. Une partie de ce langage est synthétisable.

SystemVerilog, dans sa dernière version (3.1), se veut une évolution de Verilog vers la modélisation à plus haut niveau. Il dispose désormais de plusieurs nouveaux attributs :

1. Des classes et des objets de communication et de synchronisation (sémaphore, boîtes aux lettres);
2. Des processus dynamiques;
3. Une interface directe avec le langage C;

[1] Un résumé de l'architecture d'un modèle SystemC est présenté dans l'annexe II.

4. Un support intégré pour la vérification, basé sur les assertions;

5. Des interfaces.

Autres langages

Pour la conception au niveau système, les langages prédominants présentement sur le marché sont C/C++, incluant SystemC, et SystemVerilog. Toutefois, d'autres langages se font connaître. SysML [44], un sous-ensemble d'UML, est un langage destiné aux ingénieurs systèmes pour la spécification, l'analyse, la conception et la vérification de systèmes complexes de tous types, incluant, entre autres, le matériel, le logiciel ou des données. D'autres langages, tels Real-time Java [45], B# [46] ou PERC Pico [47], visent spécifiquement la conception et l'implantation de systèmes embarqués et temps réel, sans nécessairement opérer au niveau système.

2.3.3.2 Raffinement progressif

La conception au niveau système utilise plutôt une approche de conception allant du haut vers le bas (de l'anglais Top-Down), contrairement à la méthodologie de conception traditionnelle qui est plus fidèle à l'approche de bas vers le haut (de l'anglais Bottom-Up) [48]. La méthodologie de conception au niveau système se veut une approche par raffinement progressif. Ainsi, les modèles conçus initialement sont développés de façon itérative afin d'atteindre leur implantation finale exécutable. Ce raffinement progressif peut traverser différentes étapes. Elles sont résumées à la Figure 2.5.

Un premier niveau, le niveau UTF ou « **Untimed Functional** », est purement fonctionnel et ne comporte aucune notion de temps. Son objectif est de permettre la validation de l'exécution du système sans distinction logicielle/matérielle. Cette première spécification fonctionnelle du système est indépendante de la technologie.

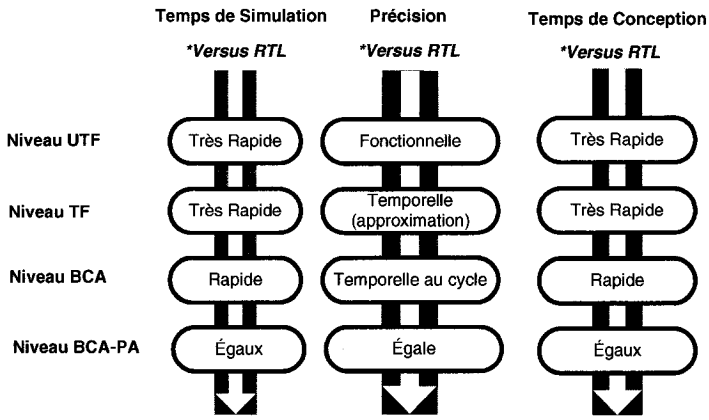


Figure 2.5 Raffinement progressif

De nos jours, il est probable que cette représentation soit en SystemC [35].

Par la suite, cette représentation est raffinée. Ainsi, les composants du système sont partitionnés. La distinction logicielle/matérielle est donc maintenant présente. De plus, les opérations de chaque composant sont typiquement annotées en fonction du temps qu'elles requièrent pour s'exécuter. Il est donc possible d'évaluer leur performance sans imposer de longs temps de simulation. Ces métriques servent aussi de guide aux outils de partitionnement. Ce niveau porte le nom de TF, ou « **Timed Functional** ».

La prochaine étape du raffinement consiste à modéliser les composants du système de façon à ce qu'ils soient précis au cycle près. Cette étape se nomme CA, pour « **Cycle-Accurate** » ou BCA, pour « **Bus Cycle-Accurate** ». La simulation du système à ce troisième niveau de raffinement est donc fidèle à la solution finale en termes de cycle d'exécution. Toutefois, les interfaces demeurent abstraites : les concepts de ports ou de canaux de communication s'appliquent toujours ici.

Enfin, en dernier lieu, les modèles des composants du système sont adaptés afin de prendre leur représentation finale, où tous leurs signaux d'entrée et de sortie sont

présents. Ce niveau porte l'appellation PA, pour « **Pin-Accurate** ».

L'attrait pour chaque niveau de raffinement repose principalement sur la nature et la qualité, en termes de précision, des métriques qu'il est possible d'extraire des simulations [35].

La section suivante présente les alternatives principales présentement utilisées en industrie ayant pour but de raffiner les modèles haut-niveaux.

2.3.4 Alternatives de conception au niveau système

L'approche préconisée afin de concevoir un système à un niveau d'abstraction élevé et de le raffiner vers sa plateforme exécutable finale varie. Présentement, trois approches se distinguent [12][49]. Voici une présentation de chacune.

2.3.4.1 Synthèse système

La synthèse système est essentiellement une approche de conception de haut vers le bas, à savoir qu'elle vise la génération automatique de l'architecture et des modèles logiciels d'un système à partir de ses spécifications [12][50]. Contrairement à la synthèse RTL, déjà utilisée depuis plus de 10 ans, la synthèse au niveau système s'apparente davantage à la synthèse comportementale. Ainsi, les spécifications du système peuvent se présenter sous une forme algorithmique, contrairement à la représentation RTL où l'ensemble des détails est présent. L'objectif de la synthèse comportementale est donc d'ajouter automatiquement les détails matériels à une description algorithmique d'un bloc matériel modélisé en C++, par exemple, afin de générer un modèle précis au cycle (« cycle-accurate ») et à la broche (« pin-accurate ») [36]. Par le fait même, automatiser cette conversion permet aux concepteurs

d'explorer plusieurs alternatives de conception, puisque la synthèse simplifie l'effort et le temps requis pour évaluer différentes variantes à un design [51].

Afin de parvenir à synthétiser un système dans son ensemble, il est primordial que ce dernier puisse être représenté, tant dans son contenu logiciel que dans son contenu matériel, dans un même langage.

Le langage SpecC [52], issu de travaux de recherche à l'Université de Californie, à Irvine, est un langage de conception et une méthodologie en soi au niveau système. Sa méthodologie tend vers l'approche de raffinement progressif présenté précédemment. En fait, il partage beaucoup de similitudes avec SystemC au niveau des concepts offerts (signaux, canaux de communication, interface de communication)[2]. SpecC est aussi un surensemble du langage C. SpecC, avec ses outils de synthèse logicielle et matérielle, répond bien aux objectifs de la synthèse système [53].

Un désavantage des langages propriétaires ou moins établis, tel SpecC, repose sur leur manque d'utilisation et sur le grand bagage de code existant en d'autres langages. Le langage C/C++ et la bibliothèque SystemC constituent une autre avenue prometteuse à la conception au niveau système dû au grand bassin de population et d'outils qui les supportent.

Dans cette optique, d'autres outils viennent répondre à ce besoin par leur capacité de synthétiser des composants exprimés dans les langages C/C++ ou SystemC : CatapultC [54] de Mentor, Agility Compiler et DK Design Suite [55] de Celoxica, SystemCrafterSC [56] de SystemCrafter, ou Cynthesizer [57] de Forte. Certaines solutions de la synthèse au niveau système demeurent pour le moment limitées. De façon générale, seul un sous-ensemble du C/C++ ou de SystemC est synthétisable. De plus, la synthèse est souvent orientée vers la génération de matériel. Bref, l'utilisateur

[2]SystemC a hérité certains concepts, dont ceux de canaux et d'interfaces, de SpecC

peut aussi se voir imposer des contraintes d'implantation afin d'utiliser certains outils.

2.3.4.2 Conception basée sur l'utilisation de noyaux IP

Contrairement à la méthode de raffinement basée sur la synthèse système, la conception basée sur l'utilisation de noyaux IP (« Component-based Design ») suit une approche de conception de bas vers le haut. Le système à concevoir est donc assemblé à partir de composants préexistants souvent appelés noyaux IP, pour « Intellectual Property ». En général, ces composants offrent une interface standard afin de s'interconnecter. Voici quelques exemples de standards utilisés :

- La suite CoreConnect [58] d'IBM qui repose sur l'utilisation de trois bus de communication différents : PLB (« Processor Local Bus »), OPB (« On-chip Peripheral Bus ») et DRC (« Device Control Register Bus »).
- Les bus AMBA [59] de Arm: les bus AMBA AHB (« Advanced High-Performance Bus ») et APB (« Advanced Peripheral Bus »).
- Le réseau-sur-puce Network [60] de Sonics.
- Le bus WishBone [61], dont la spécification est ouverte (code source libre).

Cette méthode a l'avantage d'offrir une très grande flexibilité quant au choix et à l'arrangement des blocs qui constituent le système. Elle favorise aussi la réutilisation, une solution enviable afin de réduire les coûts reliés au développement et à la vérification, puisque chaque noyau IP est préalablement vérifié.

La Figure 2.6 illustre le principe sous-jacent à cette approche appliquée au bus OPB de la suite CoreConnect. Dans la figure, une interface est utilisée afin d'utiliser un bloc qui n'a pas la compatibilité voulue avec le protocole de communication choisie.

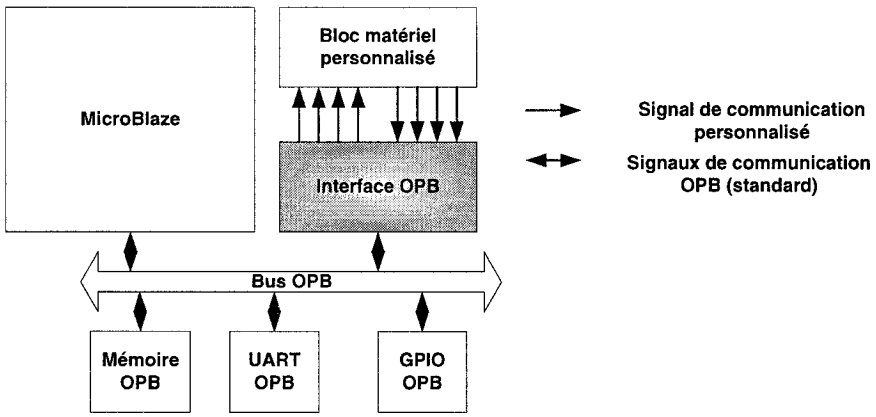


Figure 2.6 Plateforme conçue à partir de la méthodologie de conception basée sur l'assemblage de noyaux IP

Les plateformes SPACE [14], Rose [12], StepNP [62] ou celle de CoWare [63] reposent sur cette méthodologie afin de concevoir la plateforme d'un système-sur-puce.

La plateforme de conception au niveau système SPACE

SPACE [14] est un environnement de conception au niveau système. SPACE repose sur la modélisation transactionnelle et constitue une extension aux fonctionnalités de SystemC. La plateforme permet la modélisation complète d'un système tant dans le contenu de sa couche matérielle et de sa couche logicielle, incluant les appels à un système d'exploitation temps réel. Cet environnement facilite aussi le raffinement successif du modèle d'un système par l'utilisation de trois niveaux d'abstraction dédiés à la cosimulation logicielle/matérielle.

SPACE se distingue principalement grâce à :

1. Sa modélisation des comportements du logiciel par l'encapsulation des fonctionnalités d'un système d'exploitation temps réel (RTOS) dans une interface SystemC/RTOS;

2. Sa capacité à déplacer des composants du système entre les partitions logicielle

et matérielle du système, sans requérir de modification au code de la part du concepteur;

3. Son interface SystemC/RTOS et son mécanisme de communication.

La plateforme SPACE définit sa propre infrastructure de communication. C'est cette dernière qui assure un lien entre les composants matériels et logiciels d'un système-sur-puce. Les communications propres à un système conçu à partir de SPACE reposent sur la transmission de messages. SPACE distingue les composants *maîtres*, ceux qui peuvent émettrent des requêtes, des composants *esclaves*, dont la responsabilité consiste à répondre aux requêtes reçues. Un processeur correspond à un bloc du premier type alors qu'une mémoire est du second. De plus, deux composants maîtres peuvent communiquer entre eux.

Une requête dispose aussi d'un synchronisme inhérent: elle peut être bloquante ou non-bloquante. Dans le cas d'une requête bloquante, le composant qui émet la requête ne poursuivra son exécution que lorsqu'une réponse sera reçue, ce qui n'est pas le cas avec une requête non-bloquante. Cette infrastructure de communication générique permet l'ajout de composants personnalisés pratiquement indépendants du protocole de communication utilisé. La Figure 2.7 illustre l'organisation architecturale propre à SPACE. Chaque bloc s'annexe à un canal de communication avec l'aide d'une interface dédiée.

La plateforme de conception au niveau système StepNP

Tout comme la plateforme SPACE, StepNP repose sur le langage C++ et la bibliothèque SystemC. De par sa nature, la plateforme StepNP offre la possibilité de modéliser rapidement des blocs matériels, l'abstraction entre les communications et leur implémentation via les canaux, et la réutilisation de nombreux outils de développement C++. StepNP se compose de trois environnements de travail

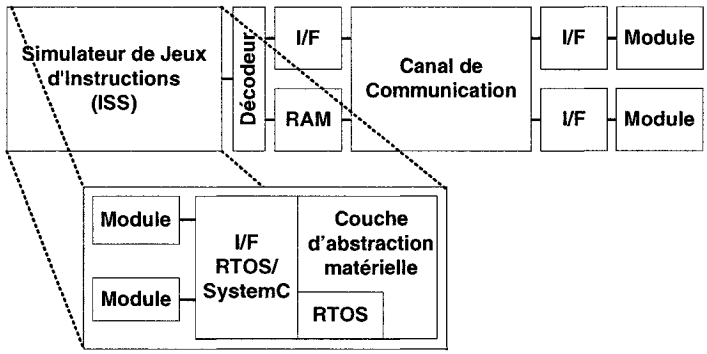


Figure 2.7 Architecture de la plateforme SPACE

complémentaires [64]. Ces environnements sont:

1. L'environnement de développement matériel permet la description des différents blocs de la plateforme multiprocesseur. On y retrouve entre autre des modèles des processeurs ARM et PowerPC, des canaux de communication et des modèles de mémoires.

2. L'environnement de développement logiciel vient préciser l'application qui sera exécutée par les blocs matériels.

3. Des outils fournissent le support requis afin de générer, déverminer, contrôler et analyser la plateforme conçue.

2.3.4.3 Conception basée sur une plateforme

La conception basée sur une plateforme (« Platform-based Design ») sert de solution intermédiaire entre les deux approches exposées précédemment. La plateforme présente une architecture fixe. Ici, chaque composant de la spécification fonctionnelle du système est associé à un élément de la plateforme sélectionnée. Cette plateforme est conçue à l'avance et, généralement, offre une certaine flexibilité afin de convenir à différentes solutions. La réutilisation d'une même plateforme à travers plusieurs

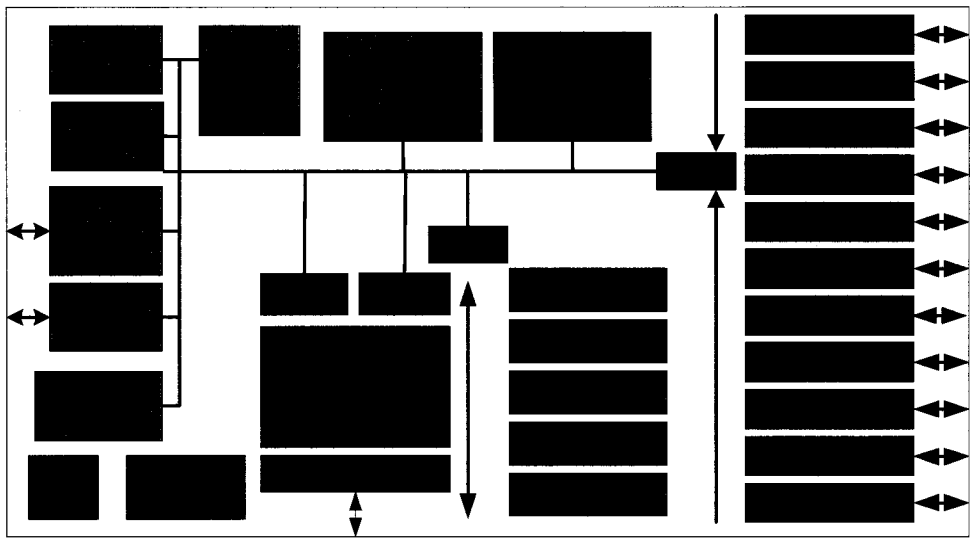


Figure 2.8 Architecture de la plateforme Nomadik (STMicroelectronics)

générations d'un même produit permet aussi de réduire les coûts non récurrents associés à sa conception [65]. Ces solutions sont toutefois souvent propres à un même domaine d'applications, tel le multimédia. STMicroelectronics, avec la plateforme Nomadik [66], Texas Instruments, avec la plateforme OMAP (« Open Multimedia Application Platform ») [67], ou la plateforme Nexperia [68] de Philips, constituent des exemples de plateformes destinées à ce mode de conception. Les Figures 2.8, 2.9 et 2.10 illustrent respectivement le contenu de ces plateformes. Il est observable que ces SoCs présentent une composition très variée de composants en plus de processeurs et d'accélérateurs matériels dédiés à leur domaine d'application.

Cette dernière section conclut le premier chapitre qui se veut une introduction aux concepts pertinents à la conception de systèmes-sur-puce multiprocesseurs.

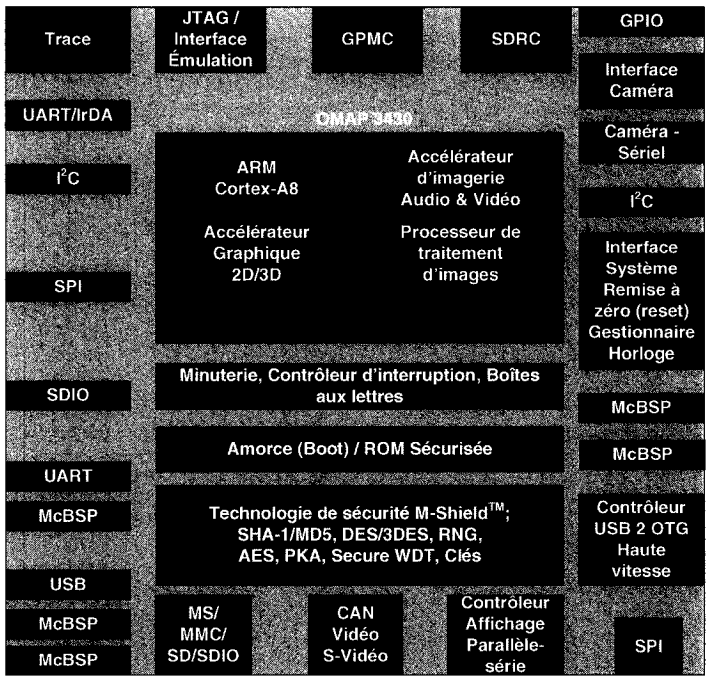


Figure 2.9 Architecture de la plateforme OMAP (Texas Instruments)

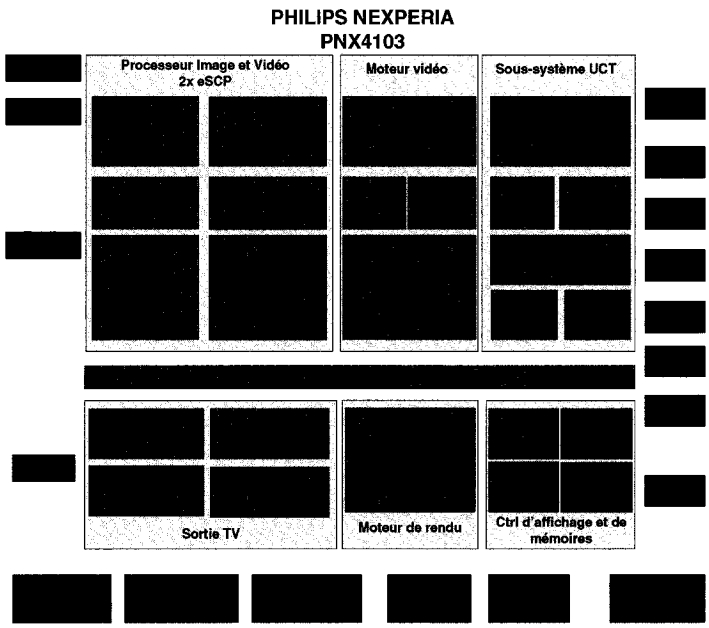


Figure 2.10 Architecture de la plateforme Nexperia (Philips)

CHAPITRE 3

LA SYNCHRONISATION DANS UN SYSTÈME MULTIPROCESSEUR

Le présent chapitre expose le problème de la synchronisation dans une plateforme multiprocesseur. Il présente par la suite les solutions qui y sont apportées. L'accent est principalement orienté vers le modèle de mémoire partagée.

3.1 La synchronisation : un besoin

Les applications destinées à des plateformes multiprocesseurs sont similaires à des applications multitâches quant à leurs besoins pour des services de synchronisation. Tout comme ces dernières, les applications conçues pour une solution multiprocesseur utilisent plusieurs processus ou tâches[1]. Afin d'assurer leur bonne exécution, ces tâches requièrent des services de synchronisation leur permettant un séquencement correct. De la même manière, dans un environnement où l'espace d'adressage est global, ou du moins commun et partagé par plusieurs tâches, il est requis que ces tâches disposent de services permettant un accès sécuritaire à des emplacements-mémoires partagés, le cas échéant, où l'accès doit être **mutuellement exclusif** [69]. Une **section critique** est définie comme une séquence d'instructions qui ont un accès exclusif à une ressource partagée [70].

Les plateformes monoprocesseurs multitâches ou multiprocesseurs offrent ces services de synchronisation qui apparaissent pratiquement équivalents aux développeurs

[1]Les termes « tâches » et « processus » sont considérés comme équivalent dans ce travail.

logiciels quant à leur utilisation (dans l'optique que le système repose sur le modèle de mémoire partagée). Toutefois, les méthodes utilisées pour les implanter sur le système cible et les rendre ainsi disponibles varient. Un système monoprocesseur peut simplement offrir des services de synchronisation à travers l'utilisation d'un système d'exploitation temps réel (RTOS) alors qu'une plateforme multiprocesseur se doit de disposer de mécanismes additionnels, souvent matériels, afin de permettre des accès atomiques à certaines ressources. L'accès atomique est crucial, puisque l'opposé risquerait de compromettre l'intégrité de la ressource accédée, que ce soit une donnée ou un périphérique. Cette atomicité est souvent garantie par (1) la désactivation des interruptions et (2) la désactivation de l'ordonnanceur d'un RTOS sur une plateforme monoprocesseur. Ces solutions seules ne conviennent pas à éliminer une **concurrence critique** (« race condition ») dans une plateforme multiprocesseur entre une tâche et une routine d'interruption [71].

3.2 La synchronisation dans les systèmes multiprocesseurs

3.2.1 Protocoles de synchronisation

L'accès à une ressource partagée est contrôlé par un protocole de synchronisation. Un protocole de synchronisation peut être bloquant, c'est-à-dire qu'une tâche est bloquée jusqu'à ce qu'elle soit éveillée afin d'indiquer qu'elle peut procéder. Le terme « positive wakeup » est aussi utilisé dans la littérature pour décrire les protocoles de synchronisation bloquants [70]. Ce type de protocole est bénéfique dans un environnement multitâche, puisqu'il permet à d'autres tâches de s'exécuter pendant que la première attend. Aussi, une tâche en attente ne crée aucune contention sur le réseau d'interconnexion du système. Toutefois, un protocole bloquant risque d'imposer un long temps de synchronisation. Ce long temps de synchronisation relève

du temps requis aux changements de contexte et à l'envoi et au traitement du signal d'éveil, souvent communiqué via une interruption matérielle.

Une seconde catégorie de protocoles de synchronisation repose sur une **attente active** (« busy waiting » ou « spin »). Ainsi, le protocole scrute continuellement la ressource demandée jusqu'à son obtention. Si cette approche n'est pas correctement contrôlée, elle peut avoir des conséquences négatives sur le système. Non seulement garde-t-elle la tâche active durant le **processus de scrutation** (« polling »), mais aussi elle risque d'engendrer un afflux de requêtes sur le réseau de communication, ralentissant par conséquent les autres éléments dans leurs opérations. C'est une approche qui peut aussi consommer plus d'énergie. Néanmoins, c'est l'approche souvent favorisée dans les systèmes basés sur le modèle de mémoire partagée puisqu'elle permet l'échange très rapide de variables de synchronisation [70].

La section qui suit présente les problèmes typiques de la synchronisation multiprocesseur. Par la suite, les principaux algorithmes de synchronisation basés sur une attente active pour les systèmes multiprocesseurs sont introduits.

3.2.2 Problèmes typiques de la synchronisation multiprocesseur

3.2.2.1 Contention

D'une manière générale, la contention se définit comme un accès compétitif ou concurrent à une ressource. Dans le contexte d'un système-sur-puce, l'accès à une mémoire centrale par plusieurs processeurs crée de la contention au niveau de la mémoire elle-même et sur le bus système, en supposant un tel réseau d'interconnexion.

Dans un contexte de synchronisation pour des systèmes-sur-puce multiprocesseurs, la contention se produit généralement au niveau des ressources partagées dont l'accès

exclusif est requis. Ainsi, si un accès atomique est requis à une plage-mémoire partagée où est emmagasiné l'état d'un objet de synchronisation[2], il est fort probable que de la contention ait lieu lorsque plusieurs processeurs cherchent à accéder à cette même ressource-mémoire.

La contention résulte généralement de l'incapacité du système de paralléliser certaines opérations. La sérialisation d'opérations influence négativement les performances d'un système. En somme, tout concepteur d'un SoC tente de minimiser ce phénomène de contention.

3.2.2.2 Point chaud (« Hot spot »)

Un point chaud dans un système résulte suite à une contention élevée sur une même ressource. Ainsi, un grand nombre d'opérations se trouvent sérialisées vers un même service ou un même composant. Un point chaud se produit dans un MPSoC lorsqu'un grand nombre de processeurs cherche à accéder à un même objet de synchronisation placé en mémoire centrale partagée. Il a été démontré que la présence de points chauds dégrade les performances de l'ensemble du trafic du réseau d'interconnexion et non seulement le trafic relié à la synchronisation des processeurs [72].

Un point chaud est donc très nuisible aux performances globales d'un système. Ils constituent un obstacle majeur à l'obtention de hautes performances dans un système basé sur un bus ou un réseau d'interconnexion multiniveaux [73][74][72]. C'est pourquoi différentes optimisations sont utilisées afin d'éviter la contention d'une manière générale et, d'une manière plus spécifique, les points chauds. À titre

[2]Dans le contexte de ce mémoire, un sémaphore et un objet de synchronisation (« lock » ou « synchronization lock ») sont considérés équivalents. Bien que dans certaines implantations, un sémaphore a des fonctionnalités spécialisées, ici les deux assurent simplement la protection d'une section critique.

d'exemples, dans les réseaux-sur-puce (NoC, « Network-on-Chip »), l'utilisation de mémoire cache[3] représente une telle optimisation qui élève le parallélisme potentiel d'un MPSoC.

3.2.2.3 Préemption

La préemption est associée à une application multitâche. La préemption a lieu lorsque l'ordonnanceur du système d'exploitation, temps réel ou non, décide qu'une tâche différente de celle qui est en cours d'exécution est plus prioritaire. Ainsi, la tâche en cours d'exécution est préemptée et une nouvelle tâche la remplace en tant que tâche active sur le processeur en question.

En lien avec la synchronisation, la préemption peut avoir deux conséquences néfastes dans une application multitâches qui repose sur un système multiprocesseur [75][76]. Celles-ci sont analogues au problème d'inversion de priorité.

1. Si une tâche est préemptée pendant qu'elle est en possession d'un objet de synchronisation, cette tâche en garde la possession pendant une durée indéterminée au détriment d'autres tâches, locales ou non, qui peuvent attendre après cette même ressource.

2. Si un objet de synchronisation est alloué à une tâche qui est prête à rouler, mais non active à ce moment, cette tâche n'est pas en mesure d'entreprendre l'exécution de sa section critique immédiatement. Par conséquent, elle conserve l'objet de synchronisation sur une période allongée inutilement.

Lorsque les objets de synchronisation reposent sur un protocole d'attente active, le premier élément se résout par l'inhibition de la préemption pendant qu'une tâche est

[3]Un résumé sur les hiérarchies de mémoire et des protocoles de cohérence de mémoire cache est présenté dans l'annexe I.

en contrôle d'un objet de synchronisation. Ce type de solutions porte le qualificatif préemption-sécuritaire (« preemption-safe ») [77].

En réponse au second problème, certaines structures de données, implantées d'une manière non-bloquante (« lock-free »), garantissent qu'au moins un processus réussira à mettre à jour, dans un temps limité, une structure de données peu importe l'état des autres processus. Ce type d'algorithme ne requiert pas d'interface particulière avec le noyau du système d'exploitation. Toutefois, il requiert généralement que le système cible supporte une des primitives de synchronisation universelles, telles que *CompareAndSwap* (CAS) ou la paire d'instructions *LoadLinked/StoreConditional* (LL/SC) [77]. Aussi, les performances de ces algorithmes varient beaucoup en fonction de la nature des structures de données utilisées [76].

3.2.3 Algorithmes de synchronisation par attente active

Puisqu'assurer l'exclusion mutuelle à l'aide d'une solution purement logicielle est très coûteux [78], la majorité des plateformes multiprocesseurs à mémoire partagée dispose d'un support matériel afin de permettre l'accès exclusif à une structure de données partagées [4]. Généralement, ce support prend la forme d'une ou de plusieurs instructions spécialisées atomiques afin de lire, modifier ou écrire en mémoire. C'est à partir de telles instructions que les protocoles de synchronisation sont construits. Une description sommaire des instructions spécialisées atomiques les plus communes est présentée dans l'annexe VI.

3.2.3.1 Objet de synchronisation basé sur le spin (*Test-and-Set*)

L'algorithme le plus simple exécute l'instruction *Test-and-set*(TAS) à répétition jusqu'à l'obtention de l'objet de synchronisation. Plus le nombre de processeurs

en spin augmente, plus les performances du système se dégradent devant le trafic grandissant généré. Non seulement ce trafic ralentit les autres processeurs non impliqués dans l'accès d'une section critique, mais le processus qui détient présentement le sémaphore est aussi en compétition avec les processeurs en spin [4].

Tableau 3.1 Pseudo code de l'opération Test-and-Set (TAS)

Initialisation	lock := CLEAR;
Acquisition	while (TestAndSet(lock) = = BUSY);
Libération	lock := CLEAR;

3.2.3.2 Objet de synchronisation basé sur le spin avec lecture (*Test-and-Test-and-Set*) (« Snooping Lock »)

Afin de réduire le coût du processus de spin ainsi que son impact sur le réseau de communication du système, il fut proposé [79] de simplement lire la valeur de l'objet de synchronisation (lecture non atomique) et, seulement lorsque celui-ci devient libre, d'effectuer l'instruction *Test-and-Set*. Cette approche réduit le trafic sur l'ensemble du système si les mémoires caches sont cohérentes. De cette manière, le processus de spin n'opère que sur la mémoire cache. Une fois l'objet de synchronisation libre, les caches sont mises à jour ou invalidées engendrant une lecture. Comme suite au changement de valeur de l'objet de synchronisation, le processus en attente peut exécuter une instruction Test-and-set. La mise à jour des caches en plus des requêtes *Test-and-set* de la part des processus en attente génère plusieurs requêtes-mémoires qui influent sur les autres processeurs. À cela s'ajoute le fait que l'instruction *Test-and-Set* est considérée comme une écriture et, par conséquent, invalide la donnée en mémoire cache, générant un trafic additionnel. Dans le pire cas où la politique de cohérence de cache « write update » est utilisée, $O(n^2)$ requêtes sont émises, où n représente le nombre de processus en attente de l'objet de synchronisation. La quantité de trafic varie en fonction du nombre de processus en attente et du protocole choisi pour la

cohérence des caches [4].

Tableau 3.2 Pseudo code de l'opération Test-and-Test-and-Set [4]

Initialisation	lock := CLEAR;
Acquisition	while(lock = = BUSY or TestAndSet(lock) = = BUSY);
Libération	lock := CLEAR;

3.2.3.3 Objet de synchronisation basé sur l'évitement des collisions

Dans l'objectif de réduire le nombre d'instructions *Test-and-Set* infructueuses, Anderson [4] proposa d'insérer un délai différent pour chaque processus, avant de revérifier l'état d'un objet de synchronisation pour ensuite tenter de l'acquérir. Ce délai est aussi appelé **recul** (« backoff »).

Le délai entre chaque nouvelle tentative d'acquisition peut être statique ou dynamique. Anderson a démontré qu'une augmentation exponentielle du délai est préférable à une augmentation linéaire, permettant au nouveau processus en spin de s'adapter plus rapidement au délai optimal. Aussi, il note que le délai ne devrait pas être augmenté lorsque l'objet de synchronisation est occupé, mais plutôt lorsqu'il est libre et que son acquisition subséquente (TAS) échoue [1].

Certaines variations aux protocoles présentés existent. Ces variantes sont particulières à l'architecture du système cible (présence ou non de mémoires caches, cohérence des caches, choix du protocole de cohérence des caches). Par exemple, il est possible d'ajouter des délais entre chaque référence à la mémoire centrale [70].

3.2.3.4 Objet de synchronisation basé sur un ticket « Ticket Lock » [1][2]

Ce type d'objet de synchronisation repose sur l'utilisation de deux compteurs globaux. Le premier identifie le ticket présentement servi et le second, le prochain ticket disponible. Lorsqu'un processus désire acquérir l'objet de synchronisation, une instruction atomique de type *FetchAndIncrement* permet de lire le compteur du prochain numéro disponible, ce qui attribut un ticket au processus, et d'incrémenter ce compteur global. Par la suite, il entre en spin jusqu'à ce que la valeur de son ticket et le numéro du ticket présentement servi, indiqué par le second compteur global, correspondent.

Afin de réduire la contention sur le réseau d'interconnexion, il est possible d'utiliser la différence entre le ticket obtenu et le numéro du ticket présentement servi afin de déterminer un recul avant le prochain spin.

3.2.3.5 Objet de synchronisation MCS

Mellor-Crummey et Scott [80] ont proposé un algorithme de synchronisation qui assure aussi un accès FIFO à l'objet de synchronisation. Leur solution est extensible aux systèmes qui ne disposent pas de mémoires caches cohérentes, mais requiert une mémoire locale à chaque processeur qui est aussi accessible globalement. La Figure 3.1 reflète le mode d'opération de cet algorithme.

Chaque objet de synchronisation garde une liste de processeurs en attente. Cette queue d'attente est une liste chaînée. Chaque processeur en attente alloue dans sa mémoire locale un nœud de cette liste chaînée. Chaque nœud de la liste contient un lien vers le prochain processeur en attente dans la queue en plus d'un fanion d'attente sur lequel le processeur attend en spin. L'objet de synchronisation

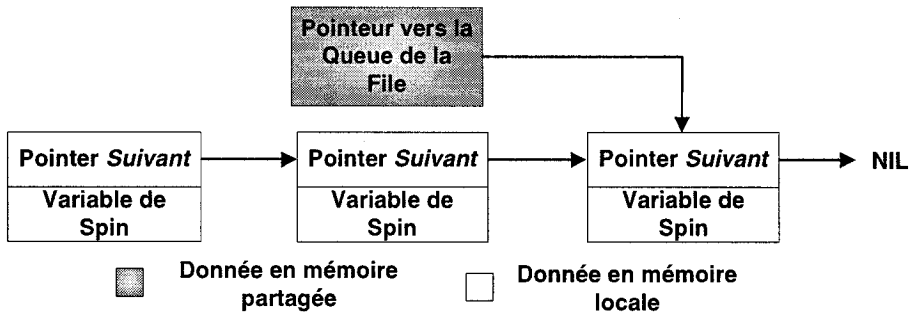


Figure 3.1 Organisation de la mémoire des objets de synchronisation MCS à exclusion mutuelle

contient un pointeur vers l'élément en queue de file. À l'aide de l'instruction atomique *FetchAndStore*, un processeur peut déterminer si l'objet est libre et se mettre en attente, dans le cas contraire, en y inscrivant l'adresse de son propre nœud. Lorsqu'un processeur libère l'objet de synchronisation, seul le fanion d'attente du prochain processeur en liste est signalé. Cette dernière étape peut se faire à l'aide de l'instruction atomique *CompareAndSwap* ou *FetchAndStore*, en fonction des ressources disponibles.

3.2.3.6 Analyse des performances des algorithmes de synchronisation présentés

Selon les expériences présentées dans [1][4][80], les algorithmes de synchronisation reposant sur une attente active offrent généralement des performances satisfaisantes, en comparaison aux algorithmes qui disposent de queues, lorsque le nombre de processeurs actifs est faible, c'est-à-dire inférieur à 10. Les stratégies qui profitent d'un recul, principalement le recul exponentiel ou basé sur un ticket, sont plus performantes. Il se trouve aussi que les approches utilisant des queues sont plus exigeantes en temps d'exécution et donc moins efficaces pour un petit nombre de processeurs; bref, lorsqu'il y a moins de contention ou que l'objet de synchronisation

est libre. Or, la situation s'inverse lorsque le nombre de processeurs augmente : les protocoles de synchronisation qui tirent profit des mémoires locales et des mémoires caches afin de réduire le trafic performent mieux.

3.2.3.7 Autres supports matériels pour la synchronisation de plateformes multiprocesseurs

Réseau combinant (« Combining Networks »)

Ce type de réseau d'interconnexion améliore les accès parallèles à un même emplacement-mémoire [81]. Essentiellement, des accès multiples concurrents à un même emplacement-mémoire sont combinés par le réseau en une seule requête-mémoire. Les algorithmes de synchronisation présentés antérieurement tirent profit de cette fonctionnalité. Par exemple, deux requêtes d'une instruction *TestAndSet* peuvent être combinées et seule une des deux requêtes retourne une valeur positive permettant par conséquent au processeur à la source de cette requête de poursuivre son exécution immédiatement. Encore une fois, la haute complexité associée à cette fonctionnalité augmente non seulement le coût du réseau, mais peut aussi affecter négativement sa latence [82]. Un compromis se présente de nouveau.

Puce SLIC

La puce SLIC [83], où SLIC tient lieu de « System Link and Interrupt Controller », est associée à chaque composant principal (processeur, contrôleur-mémoire, contrôleur E/S) du système multiprocesseur *Sequent Balance* [84]. Cette puce fournit un support additionnel pour la distribution des interruptions, pour l'exclusion mutuelle à bas niveau et pour le contrôle de la configuration et des erreurs du système. Toutes les instances de la puce SLIC sont reliées par un réseau parallèle à celui du système. La puce requiert 6000 portes logiques dans une technologie CMOS (« CMOS gate array

component ») [85].

Chaque puce SLIC implémente 64 sémaphores binaires, appelés *portes* (« gates »), et supporte un ensemble de commandes afin d'exécuter des instructions *TestAndSet* atomiques dessus. Ces 64 sémaphores sont communs à toutes les puces SLIC du système et chacune en possède donc une copie.

Le système d'exploitation DYNIX, une saveur d'UNIX utilisée par Sequent, implémente trois types d'objets de synchronisation à partir de ces *portes* de la puce SLIC : des objets de synchronisation par attente active (« spin locks »), des sémaphores et un accès direct aux *portes*. Les sémaphores de DYNIX utilisent les *portes* de la puce SLIC afin d'accéder à une mémoire partagée qui contient l'état du sémaphore en question.

La puce SLIC, comparativement à des instructions *TestAndSet* atomiques opérant sur la mémoire centrale, réduit les coûts et la complexité des autres composants principaux du système : bus système, contrôleurs-mémoires, mémoires caches des processeurs. À cela s'ajoute une réduction du trafic sur le bus, puisque les puces SLIC communiquent entre elles à travers leur propre bus.

Accélérateur matériel SoCSU/SOCLC

D'autres travaux de recherche antérieurs par un groupe à l'Institut de Technologie de Géorgie (« Georgia Institute of Technology ») ont proposé le support des services de synchronisation à travers un composant matériel, le « System-on-Chip Synchronization Unit » ou SoCSU [86]. Leur solution consiste à implanter en matériel l'ensemble de la logique de contrôle responsable des objets de synchronisation. Chaucun de ces objets est représenté par un bit identifiant leur état : libre ou occupé. Le SoCSU est conçu pour synchroniser l'accès à de courtes sections critiques seulement. Leur solution limite le trafic généré par la mise en attente d'un processeur

qui se voit refuser l'accès à un objet de synchronisation. Ce processeur attend ainsi en boucle jusqu'à la réception d'un signal d'interruption lui permettant ensuite de poursuivre son exécution.

Dans une seconde version du SoCSU, qui fut alors renommé SoCLC (« System-on-Chip Lock Cache »), les auteurs ont ajouté un support pour des sections critiques longues, en plus du support déjà existant pour les sections critiques courtes [87]. Ils utilisent la marque des 1000 cycles pour distinguer entre une section critique courte et une longue. La variante présente dans cette seconde version se situe au niveau du comportement logiciel lorsqu'une tâche se voit refuser l'accès à un objet de synchronisation. Alors que pour une courte section critique, une tâche attend en boucle, l'attente afin d'accéder à une section critique longue est bloquante : une tâche peut donc être préemptée par le système d'exploitation local.

L'organisation architecturale de SoCSU/SoCLC est montrée dans la Figure 3.2.

La solution proposée par le SoCLC a l'avantage de libérer le système de supports matériels additionnels requis pour implanter des instructions atomiques, telles *LL/SC*, *TestAndSet* ou *CompareAndSwap*, par exemple. Elle a aussi l'avantage d'offrir un minimum de configuration quant au protocole d'éveil utilisé afin d'éveiller une tâche en attente. Il est donc possible d'utiliser le protocole FIFO ou un protocole basé sur la priorité des tâches.

De plus, dans une troisième version du SoCLC, un support additionnel pour contrer l'inversion de priorité est présent [88]. Leur solution repose sur l'utilisation du protocole IPCP (« Immediate Priority Ceiling Protocol »). L'inversion de priorité correspond à une situation où une tâche de moindre priorité possédant une ressource partagée s'exécute pendant qu'une tâche de plus haute priorité est en attente suite à l'accès à cette même ressource [89]. Il est donc profitable d'éviter ce phénomène qui

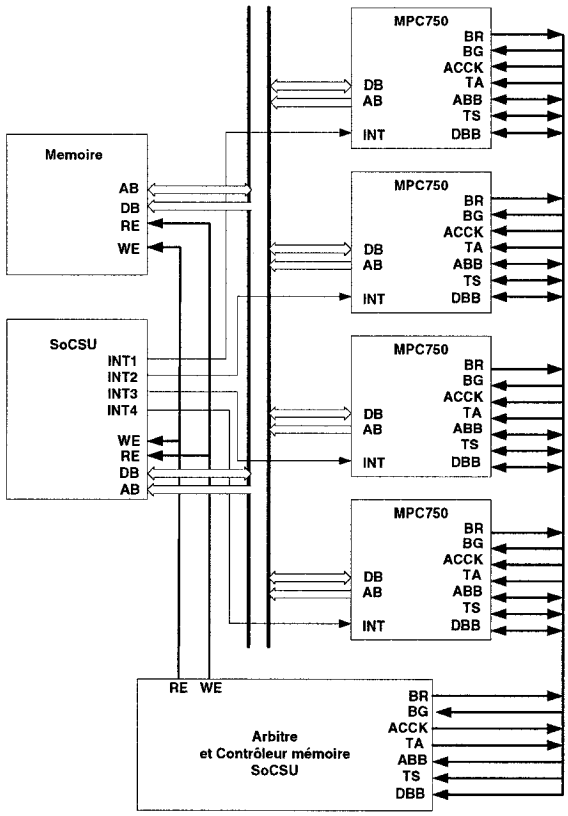


Figure 3.2 Architecture matérielle d'un système à 4 processeurs utilisant le composant SoCSU/SoCLC

affecte les performances temps-réel d'un système.

La faiblesse majeure du SoCLC tient à son mécanisme d'éveil des processeurs en attente d'accès pour un objet de synchronisation protégeant une section critique courte. Dans leur solution, les auteurs ont modifié la routine d'interruption utilisée afin que seules les opérations requises soient exécutées : une tâche qui exécutait une boucle vide d'attente peut reprendre son exécution et procéder avec l'exécution de sa section critique. Toutefois, l'ensemble des opérations génériques, propre à toutes routines d'interruption génériques, est éliminé, telle la sauvegarde de l'état des registres du processeur. Bien que l'élimination de ce code améliore grandement les performances de leur solution, elle n'est aucunement valable pour tous les systèmes où le processeur utilisé ne dispose que d'une seule ligne d'interruption. Bref, en plus des changements requis au niveau du système d'exploitation utilisé, la portabilité de cette approche est questionnable. Une solution générique, c'est-à-dire qui ne limite pas le comportement des interruptions traitées, cherche à éviter d'éveiller une tâche en attente par l'envoi d'un signal d'interruption, dont le temps de traitement avoisine le temps d'acquisition de l'objet de synchronisation.

Enfin, il est possible d'observer sur la Figure 3.2 l'organisation architecturale requise pour relier le SoCSU/SoCLC à une plateforme multiprocesseur. Une contrainte additionnelle à l'utilisation d'un tel composant se trouve au niveau de sa dépendance au contrôleur de mémoire, qui ne peut être générique. Bref, ce contrôleur aussi doit être modifié afin de supporter ce composant matériel pour la synchronisation, modifications qui ne sont pas toujours possibles.

À titre indicatif, des travaux de recherche similaires au SoCLC ont donné lieu à une unité matérielle responsable de l'ensemble des tâches du kernel d'un système d'exploitation temps-réel. Cette unité s'appelle RTU (« Real Time Unit ») [90]. Le RTU prend en charge le processus d'ordonnancement, la gestion des interruptions et

la gestion d'événements, incluant la synchronisation.

3.2.4 Optimisations apportées aux algorithmes existants

La littérature sur les protocoles de synchronisation pour plateformes multiprocesseurs renferme des versions optimisées des algorithmes présentés plus tôt. Ces algorithmes sont des variantes offrant un plus grand potentiel de parallélisme ou tout simplement des fonctionnalités additionnelles pour les concepteurs.

3.2.4.1 Objet de synchronisation de type rédacteur/lecteur

Un objet de synchronisation de type rédacteur/lecteur assouplit la condition d'exclusion mutuelle afin de permettre à plusieurs processus d'accéder à une donnée partagée tant qu'aucun processus ne tente de modifier cette dernière au même moment [91]. Essentiellement, les opérations sont séparées en deux classes. Une première classe est constituée de rédacteurs. Ils ont un accès exclusif à l'objet de synchronisation, puisqu'ils ont la capacité de modifier la donnée protégée. La seconde classe comprend les processus qui se limitent à la lecture de la donnée partagée. Ceux-ci peuvent accéder à l'objet de synchronisation d'une manière concurrente, puisqu'ils ne modifient pas la donnée protégée.

Mellor-Crummey et Scott [92] ont repris leur algorithme d'exclusion mutuelle, présenté dans la section 3.2.3.5, et en ont fait une version de type rédacteur/lecteur. Ils présentent en fait plusieurs saveurs qui possèdent chacune un critère d'égalité différent : priorité aux lecteurs, priorité aux rédacteurs ou priorité au premier processus (FIFO) sans distinction du type d'accès désiré. Les deux premières variantes peuvent résulter en la famine de certains processus alors que la dernière assure l'accès à l'objet de synchronisation au détriment d'un parallélisme limité, puisqu'un

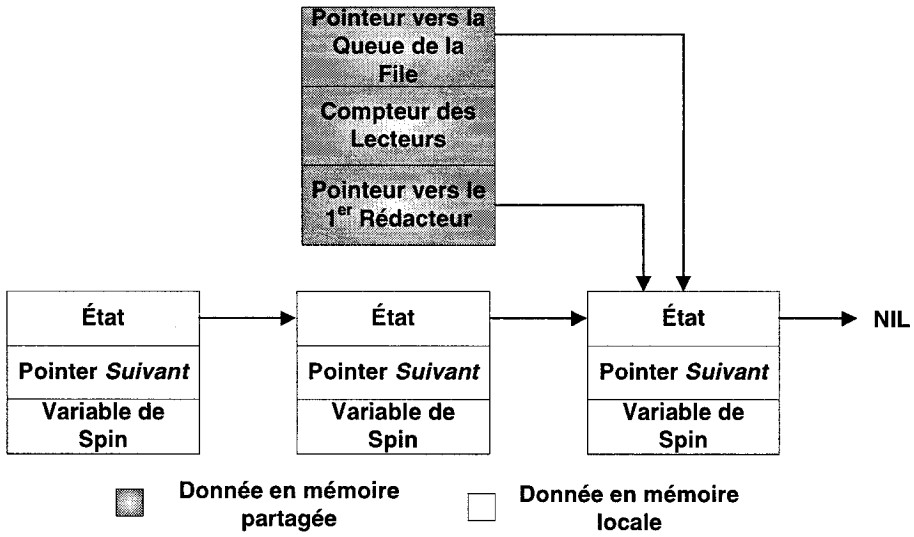


Figure 3.3 Organisation de la mémoire des objets de synchronisation MCS de type rédacteur/lecteur

rédacteur peut temporairement faire attendre certains processus lecteurs. Favoriser les processus lecteurs maximise le débit du système alors que favoriser les processus rédacteurs empêche les processus lecteurs de lire d'anciennes données.

Tout comme la version originale de l'algorithme de synchronisation MCS, cette évolution offre la possibilité qu'un processus en attente entre en spin sur une variable locale et non en mémoire centrale. Dans ce cas, c'est l'organisation montrée à la Figure 3.3 qui prévaut afin d'implanter l'algorithme. Chaque processeur emmagasine un nœud d'une liste chaînée dans sa mémoire locale.

Dans les situations où une mémoire locale accessible globalement n'est pas disponible, Mellor-Crummey et Scott proposent une version de cet algorithme de synchronisation basée sur le spin en mémoire centrale. L'aspect pertinent de cette dernière réside dans l'intégration de l'algorithme de synchronisation basé sur un ticket afin d'ordonner les lecteurs et les rédacteurs. Cet algorithme repose sur quatre compteurs:

1. Compteur des processus lecteurs servis
2. Compteur des processus lecteurs en attente
3. Compteur des processus rédacteurs servis
4. Compteur des processus rédacteur en attente

L'accès à ces compteurs se fait à travers des instructions atomiques. Le code source de cet algorithme est reproduit ci-dessous [92].

Protocole de synchronisation basé sur les tickets

```
type counter = unsigned integer

const RC_INCR    = 0x10000
const WC_INCR    = 0x1
const W_MASK     = 0xFFFF

const WC_TOPMASK          = 0x8000
const RC_TOPMASK          = 0x80000000

type lock = record
        requests : unsigned integer := 0
        completions : unsigned integer := 0

procedure start_write(L : ^lock)
        unsigned integer prev_processes :=
                        fetchClearThenAdd(&L->requests,
                                          WC_TOPMASK,
                                          WC_INCR)
        repeat until completions = prev_processes

procedure start_read(L : ^lock)
        unsigned integer prev_writers :=
                                fetchClearThenAdd(&L->requests,
                                                  RC_TOPMASK,
                                                  RC_INCR) & W_MASK
        repeat until (completions & W_MASK) = prev_writers

procedure end_write(L : ^lock)
        clearThenAdd( &L->completions,
                      WC_TOPMASK,
                      WC_INCR)

procedure end_read(L : ^lock)
```

```
clearThenAdd( &L->completions ,
              RC_TOPMASK,
              RC_INCR)
```

Enfin, il convient de mentionner que l'utilisation d'une liste doublement chaînée par Krieger, Stumm, Unrau et Hanna [93] permet de réduire à un seul accès-mémoire l'algorithme de synchronisation MCS de type rédacteur/lecteur.

3.2.4.2 Objet de synchronisation avec temps d'arrêt

Dans certaines situations, il peut être profitable ou encore même requis qu'un processus puisse cesser d'attendre après un objet de synchronisation une fois un certain temps d'arrêt (« timeout ») écoulé. Scott et Scherer III [94] ont ajouté cette fonctionnalité à deux algorithmes de synchronisation basée sur l'utilisation d'une queue d'attente, incluant celui de Mellor-Crummey et Scott (MCS) (section 3.2.3.5).

Cette fonctionnalité de temps d'arrêt peut s'avérer nécessaire dans les situations suivantes [94] :

1. Un processus temps-réel dispose d'un délai d'attente maximal pour un objet de synchronisation. Une fois ce délai expiré, le processus doit annoncer une erreur et poursuivre son exécution.

2. Dans un environnement où un système d'exploitation exécute une application multitâche, il est possible qu'un processus soit préempté alors qu'il est propriétaire d'un objet de synchronisation. Ce temps d'arrêt permet à un autre processus, en attente de cet objet de synchronisation, de cesser temporairement son attente, comme suite à l'expiration du temps d'arrêt, afin de laisser le processeur à une autre tâche. Il pourra alors essayer de nouveau d'acquérir l'objet de synchronisation ultérieurement.

3. Dans un système de bases de données parallèles, le délai d'attente sert de solution au rétablissement comme suite à un interblocage. Ainsi, un processus qui attend « trop longtemps » peut supposer qu'un interblocage s'est produit, annuler son opération courante, et répéter cette opération par la suite.

3.2.5 Variantes des algorithmes basés sur le spin avec délais adaptatifs

Dans un système dont les processeurs ne disposent pas de mémoires caches ou dans les situations où les variables de synchronisation ne sont pas utilisables à partir d'une mémoire cache, il est essentiel d'utiliser un mécanisme d'attente plus évolué dans l'élaboration d'une stratégie de synchronisation. Telle qu'illustrée auparavant, une stratégie d'attente qui repose sur un spin continuel n'est pas performante dans ce genre d'environnement. Ces stratégies d'attente reposent sur l'utilisation d'un recul (« backoff ») entre chaque spin. Différents protocoles d'attente sont présentés dans la littérature quant au calcul du recul voulu.

Dans une première catégorie, exception faite du blocage immédiat, se trouvent les mécanismes d'attente dits passifs dont l'identification du recul est déterministe : elle ne dépend pas de données obtenues dynamiquement au fur et à mesure que l'application exécute. Cette catégorie comprend le recul constant, linéaire et exponentiel [95], introduit dans la section 3.2.3.3.

En complément à la première catégorie, une seconde catégorie comprend les algorithmes d'attente avec **recul adaptatif**. Ceux-ci utilisent l'information disponible à l'exécution afin de calculer le temps de recul avant une nouvelle tentative d'acquisition d'un objet de synchronisation [96]. Lorsque l'algorithme de synchronisation repose sur l'utilisation d'un ticket, chaque processus connaît sa position dans la file d'attente (le numéro de son ticket) ainsi que le nombre de processus qui le devance (le numéro du ticket présentement servi). Cette différence peut servir de recul.

3.2.5.1 Protocoles adaptatifs

Afin de combiner les avantages des protocoles de synchronisation par attente active et bloquant, les protocoles adaptatifs, aussi appelés réactifs, utilisent des données dynamiques du système afin de modifier, à l'exécution, leur comportement. Ainsi, ces protocoles cherchent d'abord à utiliser l'attente active, où des processus entrent en spin, jusqu'à ce qu'une certaine limite soit atteinte. Par la suite, ils favoriseront une stratégie bloquante. Les algorithmes présentés proviennent des travaux de Karlin, Li, Manasse et Owicki [97].

Délai de spin fixe

L'approche de base consiste à exécuter un nombre fixe de spins avant de procéder à une attente bloquante. Typiquement, le nombre de spins correspond à une fraction du nombre de spins équivalant à un changement de contexte. Le coût de la synchronisation est plus faible lorsque la limite de spin est placée à la moitié du nombre de spins équivalant à un changement de contexte, contrairement à en être égal.

Algorithme en ligne optimal

Un algorithme dit *en ligne* (« online ») recueille les données utilisées dans son processus décisionnel durant l'exécution de l'application en question. Par opposition, un algorithme *hors ligne* (« offline ») connaît à l'avance les comportements de l'application. Ce dernier sert simplement de référence aux fins de comparaison en tant que solution idéale.

L'algorithme en ligne optimal utilise une approximation de la distribution des temps d'attente sur les objets de synchronisation, en termes de nombre de spins. À partir de cette distribution, il est possible d'ajuster dynamiquement la limite sur le nombre

de spins requis avant une attente bloquante. Bien que cette approche offre une très grande précision, elle est impraticable dû à la surcharge d'exécution qu'elle demande.

Algorithme en ligne : les trois derniers temps d'attente (« Last three samples »)

Une solution qui repose sur cette même distribution consiste à utiliser le temps d'attente des trois derniers accès afin d'estimer la distribution des temps d'attente sur l'objet de synchronisation. La distribution est alors approximée comme étant la moyenne des distributions sur les trois derniers accès.

Algorithme en ligne : marche aléatoire (« random walk »)

L'algorithme de la marche aléatoire repose sur la supposition que la probabilité que le temps d'attente dépasse le temps d'un changement de contexte est élevée si le temps d'attente de la tâche précédente est supérieur à celui d'un changement de contexte. Chaque fois qu'une tâche attend sur l'objet de synchronisation pour une durée supérieure au temps d'un changement de contexte, la limite de spins est abaissée afin que les tâches successives se mettent en attente bloquante plus tôt.

Une analyse comparative de sept algorithmes d'attente avec recul adaptatif est présentée dans [97]. Ces travaux concluent que les solutions adaptatives sont plus performantes que les solutions non-adaptatives ou passives. Aussi, le blocage immédiat (attente bloquante) est rarement avantageux, en comparaison à une solution hybride. Des travaux complémentaires ont aussi analysé la limite optimale à utiliser en fonction de la distribution des temps d'attente [98].

Spin : global ou local

Lim et Agarwal [95] proposent une variante aux algorithmes adaptatifs présentés

jusqu'ici. Leur algorithme adaptatif choisit entre une attente active basée sur un spin vers une mémoire globale avec l'instruction atomique *TestAndSet* (TAS) et un spin en mémoire locale, tel que proposé par l'objet de synchronisation MCS avec queue. Leur implantation repose sur une variable qui indique quel mode de synchronisation utiliser. Le mode passe de TAS à MCS lorsqu'un nombre limite de spins est atteint. Cette solution performe bien tant que la fréquence de changement de mode est basse. Autrement, la surcharge associée au changement de mode domine et affecte négativement les performances.

3.3 Conclusion

Enfin, il est possible de constater qu'il existe différentes variations quant aux protocoles de synchronisation disponibles. En général, le choix du protocole dépend fortement de l'architecture cible et du type d'application. La présence de mémoires locales et de mémoires caches ainsi que le type de protocole de cohérence utilisé pour les caches affectent directement les performances du protocole de synchronisation. Les services matériels disponibles, tels les instructions atomiques ou des sémaphores matériels, influencent aussi ce choix. Une présentation du coût matériel et des facteurs qui influencent l'implantation d'instructions atomiques (CAS, FetchAndAdd, ...) est donnée dans [99]. Il va de soi que le niveau de contention, un facteur dynamique et fonction de l'application, affecte aussi le choix de l'algorithme de synchronisation. Certains offrent une surcharge opérationnelle plus faible en présence de contention dans le système, mais à un coût plus élevé en temps d'exécution en absence de contention. De la même façon, le choix d'un protocole d'attente est influencé par le temps d'attente des processus et donc la durée d'utilisation des objets de synchronisation. En somme, l'identification d'un algorithme de synchronisation universel et optimal n'est pas possible et requiert qu'un compromis soit fait.

Il est aussi pertinent de mentionner la forte dépendance de la majorité des modèles de programmation existant aux systèmes multiprocesseurs SMT et aux applications dédiées à des fins scientifiques (voir annexe III). Dans une optique de conception à un haut niveau d'abstraction, telle la conception au niveau système, une solution où le code source est annoté (OpenMP) et où un support matériel propre à certaines architectures est requis (OpenMP, MPI) pose des contraintes contrevenantes à certaines solutions.

CHAPITRE 4

SISSMA : SYNCHRONISATION MULTIPROCESSEUR DANS LA PLATEFORME VIRTUELLE SPACE

4.1 Terminologie

Par souci d'uniformité, il convient de définir certains termes utilisés dans le présent ouvrage afin d'assurer une compréhension uniforme.

Processus et tâche

Le domaine des systèmes temps réel distingue un processus (« process ») d'une tâche (« task » ou « thread »). De cette manière, un processus (« process ») est souvent associé à un fil d'exécution qui possède sa propre mémoire d'exécution, alors qu'une tâche (« task » ou « thread »), qui prend l'allure d'un processus léger, peut partager son espace mémoire avec d'autres tâches de la même nature. Cette distinction entre processus et tâche est sans conséquence dans le contexte de la bibliothèque SISSMA. Ainsi, ces deux termes sont considérés équivalents : les deux désignent un fil d'exécution.

Aussi, puisque les présents travaux s'appliquent dans un environnement de codesign logiciel/matériel au niveau système, une tâche relève autant d'une exécution logicielle, comme sur un processeur d'usage général, que d'une implantation matérielle, par exemple en tant que module matériel dédié.

Objet de synchronisation et sémaphore

Bien que plusieurs systèmes d'exploitation, temps réel ou non, distinguent les

sémaphores, les objets de synchronisation et les *mutex*, dans le contexte de la bibliothèque SISSMA, ils sont tous considérés équivalents. Bref, tous permettent l'exclusion mutuelle. Ainsi, il est possible de protéger l'accès à une ressource partagée par l'utilisation en paire d'un sémaphore de synchronisation plutôt qu'avec le mutex. Aussi, l'envoi comme la réception d'un signal de synchronisation peut se faire indépendamment par deux tâches différentes, par exemple dans le but de synchroniser ces dernières. Enfin, l'utilisation du terme *mutex*, comme objet de synchronisation assurant une exclusion mutuelle, sera évitée afin d'éviter toutes confusions avec ce type d'objets dont les fonctionnalités diffèrent d'un RTOS à un autre, par exemple au niveau des options et fonctionnalités (priorité d'attente, héritage de priorité, etc.).

4.2 SISSMA : introduction et objectifs

Fondamentalement, SISSMA (« **S**PACE eng**I**ne for **S**ynchronization of **S**ystemC **M**ultiprocessor/Multithreaded **A**pplications ») consiste en une bibliothèque responsable d'offrir des services de synchronisation. Ces services de synchronisation sont donc utilisables autant par les tâches logicielles que par les tâches matérielles. Les objectifs de SISSMA sont de :

1. Favoriser le développement d'application sur MPSoC

   Les systèmes d'exploitation temps réel (RTOS) permettent la synchronisation d'applications multitâches sur un système monoprocesseur. SISSMA cherche à étendre ces fonctionnalités vers les systèmes multiprocesseurs (voir Figure 4.1).

2. Permettre la synchronisation dans une stratégie de conception au niveau système (ESL).

   SISSMA se veut une bibliothèque indépendante du partitionnement du système. La conception au niveau système implique des décisions de partitionnement

après la conception du système, tant dans son contenu logiciel que matériel. Dans cette optique, il est profitable, comme suite à un nouveau partitionnement du système, d'éviter les modifications à l'application partitionnée. De cette façon, les services de synchronisation peuvent être utilisés sans se soucier de l'organisation finale de l'application.

3. Offrir des services de synchronisation indépendants de la plateforme matérielle

   Tel qu'illustré antérieurement, la majorité des protocoles de synchronisation pour des plateformes multiprocesseurs utilise des fonctionnalités propres à la plateforme cible dans leur opération telle que les instructions atomiques spécialisées, les mémoires caches et leur protocole de cohérence associé, les mémoires locales accessibles globalement, etc. La conception de MPSoCs basée sur l'assemblage de composants IP offre un potentiel inégalé en termes de flexibilité et de variantes dans la composition du système. Par exemple, il est possible de créer une plateforme multiprocesseur par l'interconnexion de multiples processeurs simples. Toutefois, dans l'éventualité où ces derniers ne disposent pas des améliorations architecturales propres aux MPSoCs, SISSMA fournit alors le support logiciel et matériel afin de permettre l'utilisation d'un protocole de synchronisation multiprocesseur.

4. Raffiner la granularité de la synchronisation dans SPACE

   SPACE, dont la technologie repose sur la bibliothèque logicielle SystemC, permet la modélisation d'un SoC (ou MPSoC) par l'assemblage de modules SystemC. Une restriction de SPACE se présente au niveau de l'implantation des modules : chaque module ne peut contenir qu'une seule tâche (« process »). Aussi, SPACE possède actuellement un mode de communication et de synchronisation basé sur la transmission de messages. Ces échanges ont la possibilité d'être bloquants ou non.

   SISSMA vient complémenter SPACE à l'aide de services de synchronisation

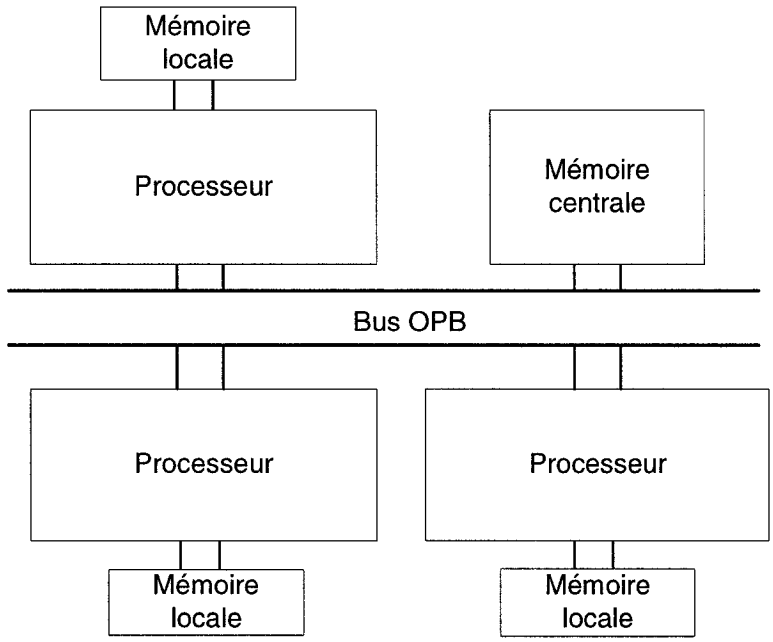


Figure 4.1 MPSoC typique ciblé par les présents travaux

utilisables par différentes tâches, d'un même module ou non. Il sert aussi de complément dans son mode de synchronisation permettant l'utilisation de mémoires partagées, par opposition à la transmission de messages.

4.3 Définition : SISSMA

4.3.1 Services disponibles

Les services de synchronisation offerts par SISSMA couvrent deux volets complémentaires. Tout d'abord, la synchronisation intertâche constitue une première fonctionnalité. Cette fonctionnalité permet l'envoi et la réception de signaux de synchronisation similaires aux événements de SystemC. Par analogie, cette fonctionnalité offre l'équivalent des sémaphores binaires d'un système d'exploitation.

Une tâche en attente d'un signal de synchronisation n'est pas autorisée à poursuivre son exécution tant que le signal n'est pas reçu.

Tel que mentionné précedemment, l'utilisation explicite du sémaphore SISSMA permet la protection de tous types de ressources, matérielles comme logicielles. De plus, SISSMA fournit des objets plus évolués afin de permettre le partage de mémoire entre différentes tâches. À cette fin, il est possible de partager tous les types de données de base supportés par le langage de programmation C++ : formats de données prédéfinies (*int*, *short*, *float*, *double*, ...) en plus des types définis par le développeur (*structure* et *objet*). Les tableaux unidimensionnels et bidimensionnels sont présentement supportés.

4.3.2 Configuration : synchronisation locale ou à distance

La configuration de chaque objet de synchronisation de la bibliothèque SISSMA dépend des tâches qui interagissent sur cet objet.

Dans un système multiprocesseur, deux situations peuvent se produire. D'abord, il est possible que deux tâches qui exécutent sur un même processeur cherchent à se synchroniser entre elles. Dans ce cas, il est plus performant d'utiliser les services de synchronisation du système d'exploitation local à ce même processeur, puisque ces deux tâches dépendent l'une de l'autre. Si une première tâche est bloquée en attente d'un sémaphore, il va de soi que nulle autre que la seconde tâche ne peut la libérer et ainsi permettre à la première de poursuivre. Ce type de synchronisation est référé comme une **synchronisation locale**.

Toutefois, s'il se trouve que ces deux tâches en question exécutent sur deux processeurs distincts, les services des systèmes d'exploitation individuels ne suffisent plus, puisque l'état du sémaphore commun aux deux tâches doit être partagé. Un accès atomique

à un emplacement-mémoire est donc requis afin d'implanter un des protocoles de synchronisation présentés au chapitre précédent. Les processeurs utilisés afin de construire des MPSoCs dans la plateforme SPACE n'offrent pas les mécanismes atomiques requis par ces algorithmes (instructions atomiques spécialisées ou mémoires caches cohérentes). La solution utilisée par la bibliothèque SISSMA consiste en l'intégration, à l'architecture de la plateforme multiprocesseur, d'un composant matériel, nommé *moteur de synchronisation SISSMA*, afin d'assurer l'accès atomique aux variables partagées utilisées par le protocole de synchronisation associé aux sémaphores SISSMA. Ce type de synchronisation est appelé **synchronisation à distance**.

Jusqu'ici, il n'y a que des tâches logicielles qui sont considérées. Il convient d'étendre ce raisonnement aux différentes combinaisons de synchronisation impliquant des tâches logicielles et matérielles.

Identification du mode d'opération d'un objet de synchronisation SISSMA

Synchronisation locale (voir Figure 4.2):

a) La synchronisation locale est permise seulement lorsque l'ensemble des tâches logicielles impliquées dans la synchronisation s'exécute sur un même processeur. Ces tâches ne peuvent pas migrer vers un autre processeur au cours de leur exécution : pas d'équilibrage de charge (« load balancing »). (Figure 4.2 - Cas 1)

b) Par extension, la synchronisation locale est aussi permise lorsque l'ensemble des tâches matérielles impliquées dans la synchronisation exécute sur un même module matériel. (Une implantation matérielle du système d'exploitation ou une solution équivalente est toutefois requise.) (Figure 4.2 - Cas 2)

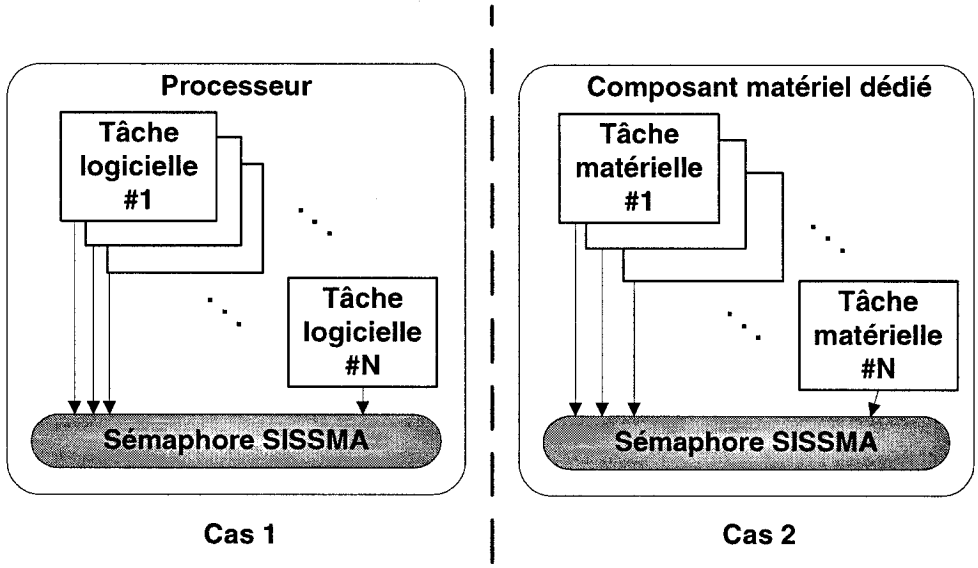


Figure 4.2 Cas d'utilisation de la synchronisation locale lors de la configuration d'un sémaphore SISSMA

Synchronisation à distance (voir Figure 4.3):

a) La synchronisation implique plusieurs tâches logicielles dont au moins deux sont associées à des processeurs différents. (Figure 4.3 - Cas 1)

b) La synchronisation implique une ou plusieurs tâches logicielles en plus d'une ou plusieurs tâches matérielles. (Figure 4.3 - Cas 2)

c) La synchronisation implique plusieurs tâches matérielles implantées dans des composants matériels distinctifs et non hiérarchiques. (Figure 4.3 - Cas 3)

4.3.3 Optimisations dans un contexte temps-réel

À la base, la bibliothèque SISSMA utilise un protocole de synchronisation simple qui repose sur une attente active périodique. Le délai d'attente entre chaque nouvelle tentative d'acquisition du sémaphore est constant et configurable. Ce protocole, bien que fonctionnel, performe mieux lorsqu'il est optimisé.

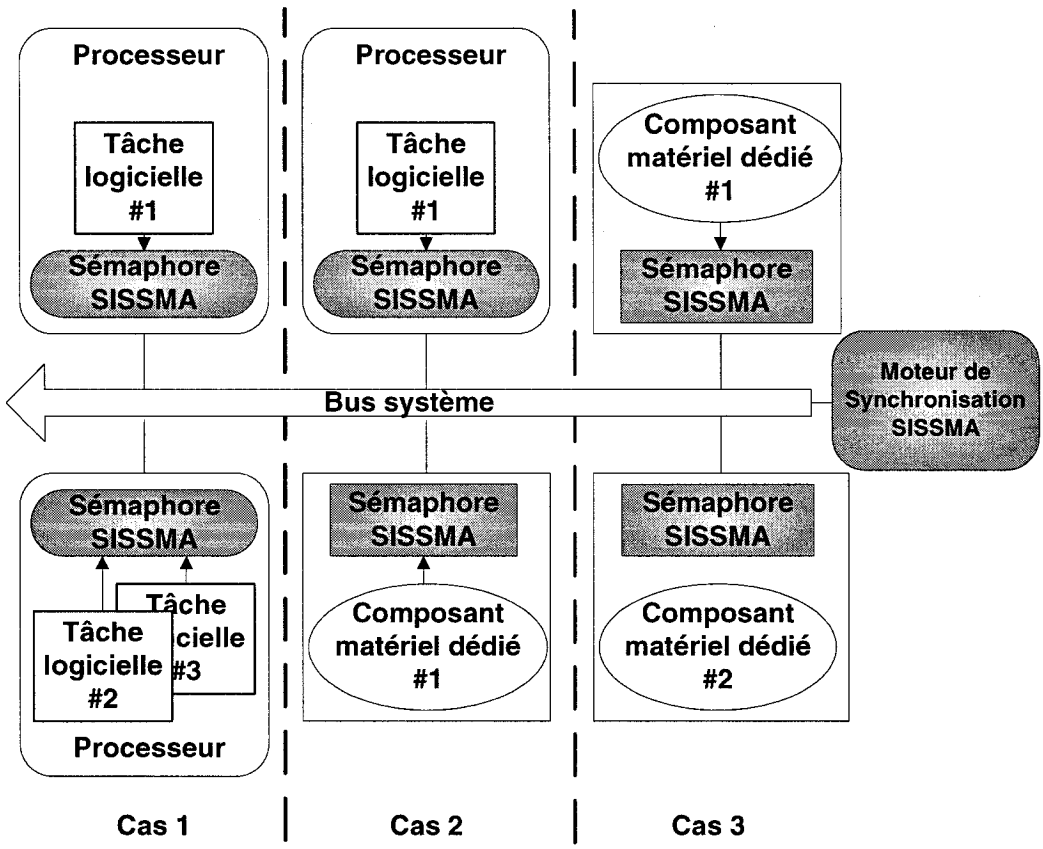


Figure 4.3 Cas d'utilisation de la synchronisation à distance lors de la configuration d'un sémaphore SISSMA

Puisque la synchronisation sur une plateforme multiprocesseur se doit d'être performante, la bibliothèque SISSMA offre plusieurs options permettant de réduire le coût associé au processus de synchronisation. Ces optimisations s'appliquent dans le cas d'une synchronisation à distance seulement. Lors d'une synchronisation locale, ce sont les caractéristiques des sémaphores du système d'exploitation local qui prévalent. La section 4.5 clarifie ce dernier point.

4.3.3.1 Optimisation du temps de blocage : recul adaptatif

Par défaut, seul un accès séquentiel (FIFO) à un objet de synchronisation SISSMA est permis. Dans l'éventualité où cet objet serait déjà acquis par une tâche autre, la nouvelle tâche entre en attente active basée sur le spin. Afin de limiter la contention sur le réseau d'interconnexion du système, la bibliothèque SISSMA met à profit le recul adaptatif. L'algorithme de synchronisation dispose de deux données dynamiques afin de déterminer le recul voulu entre deux spins consécutifs : (1) la moyenne des durées des deux dernières sections critiques courtes (la distinction courte et longue d'une section critique est décrite dans la section 4.3.3.2) sur ce sémaphore et (2) le nombre de tâches plus prioritaires en attente sur ce même sémaphore[1]. L'algorithme d'attente active peut donc utiliser cette information et calculer son recul en fonction de la formule suivante :

$$\text{Recul} = \text{Rang} * \text{DuréeMoyenneSC} / 2 \quad \text{cycles}$$

où *Rang* correspond au nombre de tâches plus prioritaires en attente du même sémaphore et *DuréeMoyenneSC* correspond à la moyenne de la durée des deux dernières sections critiques courtes sur ce même sémaphore. Le recul, en nombre de cycles de l'horloge système, est ensuite divisé de nouveau en fonction de l'implémentation de l'attente active utilisée. Cette implémentation correspond en général à une boucle vide[2]. La division additionnelle par deux tient du fait que la variance sur la moyenne de la durée des sections critiques n'est pas connue. Ainsi, une section critique plus longue pourrait compromettre cette moyenne au risque de faire attendre inutilement une tâche.

[1] Dans son implantation actuelle, obtenir cette seconde donnée requiert une lecture additionnelle sur le moteur de synchronisation SISSMA.

[2] Nous considérons que l'exécution d'une boucle vide requiert 4 instructions en langage assembleur.

4.3.3.2 Optimisation de la concurrence : section critique longue et promotion

Une option additionnelle concerne la longueur des sections critiques. Si une tâche possède une section critique d'une longue durée, il est plus profitable que les tâches en attente bloquent temporairement. Cette attente bloquante permet d'améliorer la concurrence dans l'application, en supposant qu'elle soit multitâches. Une section critique est considérée longue si sa durée excède le temps requis pour deux changements de contexte, puisque la tâche mise en attente sera éventuellement chargée de nouveau sur le processeur en question. La responsabilité est déléguée aux développeurs de spécifier si l'acquisition d'un sémaphore servira à accéder à une section critique longue ou courte. Par défaut, la section critique courte est privilégiée, particulièrement lorsqu'un simple accès à une donnée partagée est requis.

L'utilisation de l'attente bloquante est aussi mise à contribution dans l'utilisation du concept de promotion[3]. La promotion permet de transformer une requête pour une section critique courte en une requête pour une section critique longue. Ce type de promotion a lieu lorsque la queue d'un sémaphore contient une tâche en attente pour une section critique longue. Dans ce cas, il est inutile que les autres tâches qui lui succèdent dans la file attendent à l'aide du spin, ce qui serait un gaspillage de ressources. La promotion doit être autorisée par le développeur.

4.3.3.3 Optimisation de la concurrence : accès rédacteur/lecteur

Tel que présenté dans la section 3.2.4.1, certaines ressources peuvent, dans certaines circonstances, être accédées de façon non exclusive. Par exemple, des données

[3]Les travaux présentés dans [95][98] montrent une approche à deux phases. Une première phase utilise une attente active. Lorsqu'un certain délai est écoulé, l'algorithme de synchronisation change de mode et utilise alors une attente bloquante.

partagées peuvent être lues par plusieurs tâches à la fois, mais modifiées par une seule à un moment donné. Dans un tel cas, il est profitable de permettre des accès concurrents à cette dernière. L'option rédacteur/lecteur permet des accès concurrents à une même ressource. Lorsqu'un objet de synchronisation de base est utilisé (classe *SissmaSync*), il est de la responsabilité du développeur de spécifier le type d'accès demandé : lecteur ou rédacteur. Toutefois, lorsque des objets plus évolués sont utilisés pour le partage de mémoire (*SissmaData*, *SissmaDataArray* et *SissmaDataArray2d*), le type de synchronisation est implicite : une lecture ou une écriture sera protégée en tant que lecteur et rédacteur, respectivement, tel qu'indiqué par la nature de l'objet. L'algorithme utilisé assure une égalité d'accès (« fair lock ») au sémaphore; les lecteurs ou les rédacteurs ne sont pas privilégiés. Si l'option est désactivée, tous les accès sont mutuellement exclusifs.

4.4 Conception des composants SISSMA

4.4.1 Réutilisation du concept de tickets

L'utilisation de tickets dans le protocole de synchronisation permet l'ordonnancement implicite des tâches cherchant à contrôler un sémaphore. Le ticket (section 3.2.3.4), bien qu'il requière un compteur afin d'identifier l'état d'un sémaphore binaire, en remplacement à un seul bit d'état, élimine l'utilité des files d'attente pour les tâches en attente active. Désormais, les files d'attente sont seulement requises pour les tâches en attente bloquante (ou promues), diminuant par conséquent la profondeur des files. Les files d'attente sont aussi simplifiées par l'utilisation du ticket, puisque seul l'accès FIFO est requis. Le ticket facilite aussi l'attente active adaptative. L'assignation, l'utilisation et la transmission de la priorité de chaque tâche ne sont alors plus pertinentes non plus, dans le contexte de la bibliothèque de synchronisation SISSMA,

la priorité d'une tâche n'étant pas pertinente.

4.4.2 API SISSMA : modélisation système et conception logicielle

4.4.2.1 Structure

L'interface de programmation (API) offerte par la bibliothèque SISSMA permet l'utilisation de ses services lors de la conception au niveau du système. Les services de synchronisation sont supportés par plusieurs classes C++. Chaque classe offre des services plus évolués, d'où sa structure hiérarchique. Le diagramme de classe simplifié de l'API SISSMA est montré dans la Figure 4.4.

Afin de minimiser la contention vers un même moteur de synchronisation SISSMA, il est possible de partitionner l'ensemble des sémaphores SISSMA entre plusieurs de ces modules matériels. La classe *SissmaBase* permet de localiser le moteur de synchronisation SISSMA associé au sémaphore en question en plus de ses options. Les options disponibles sont :

1. Utilisation de l'attente bloquante avec protocole d'attente (présentement limité au mode FIFO)

2. Utilisation d'accès de type rédacteur/lecteur

Les classes *SissmaSync* et *SissmaSyncRw* permettent la synchronisation intertâches. Ainsi, elles implantent les fonctionnalités d'un sémaphore. Lorsque les fonctions *lock* et *unlock* de ces classes sont utilisées en paire, il est possible d'assurer l'exclusion mutuelle à une ressource. La classe *SissmaSyncRw* se distingue de sa classe parent par son support à des accès de type rédacteur/lecteur, d'où le suffixe « Rw » (Reader/Writer).

Les trois dernières classes, *SissmaData*, *SissmaDataArray* et *SissmaDataArray2d*,

permettent le partage d'une donnée, d'un tableau unidimensionnel ou d'un tableau bidimensionnel, respectivement. Elles offrent un éventail de méthodes d'accès aux données partagées accessibles en lecture et en écriture.

4.4.2.2 Configuration du mode de synchronisation

La bibliothèque SISSMA utilise les génériques (« templates ») du langage C++. Pour le partage de données, un premier paramètre du générique identifie le type de données partagées. Un paramètre distinct identifie le mode de synchronisation (local vs à distance) utilisé pour cet objet. Un exemple de configuration est montré ci-dessous.

Exemples de déclaration d'un sémaphore SISSMA

```
// synchronisation locale
SissmaData<int, SISSMA_DATA_LOCAL>
                sissmaSyncLocale(SISSMA_SYNC_SEM_ID_A,
                                 SISSMA_DEVICE_ID_BASE_ADDR);

// synchronisation à distance
SissmaData<double, SISSMA_DATA_REMOTE>
                sissmaSyncDistance(SISSMA_SYNC_SEM_ID_B,
                                   SISSMA_DEVICE_ID_BASE_ADDR);
```

4.4.3 Moteur de synchronisation SISSMA : conception matérielle

4.4.3.1 Modèle SystemC

Le modèle SystemC du moteur de synchronisation SISSMA repose sur les services de la plateforme SPACE afin de communiquer avec d'autres composants du système. Il est conçu au niveau transactionnel (TLM) afin d'améliorer ses performances de

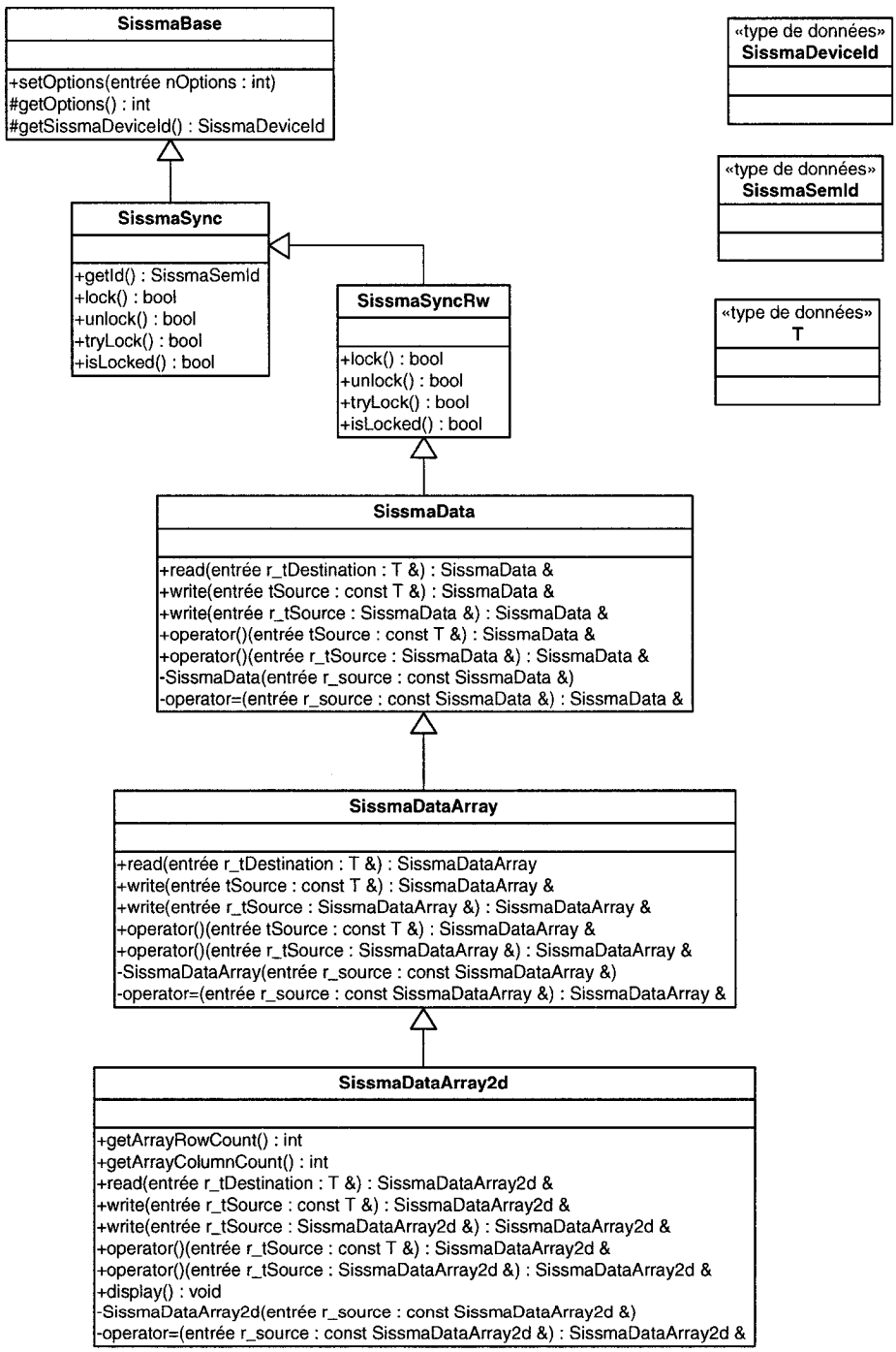


Figure 4.4 Diagramme de classe simplifié de l'API SISSMA

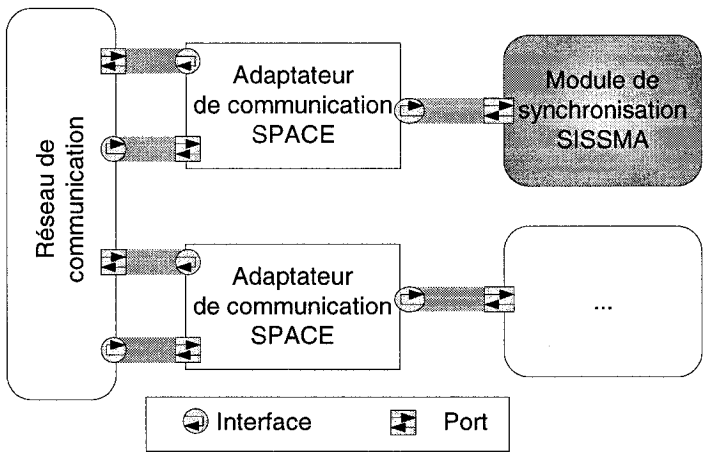


Figure 4.5 Organisation architecture du module SISSMA dans la plateforme SPACE

simulation. Ce modèle a ensuite servi de guide lors de l'implantation de son équivalent VHDL.

Communication

La plateforme SPACE possède ses propres infrastructures de communication. Tout module communiquant interagit avec cette interface de communication, propre à SPACE afin d'envoyer ou de recevoir un message. Cette organisation architecturale est montrée à la Figure 4.5.

L'utilisation de ports et d'interfaces SystemC permet l'émission de requêtes entre un module et son adaptateur de communication respectif. Ces transactions abstraient les détails du protocole de communication utilisé.

Structure

Conceptuellement, SPACE distingue les composants maîtres (« module ») et composants esclaves (« device »). Un composant maître a la capacité d'envoyer et de recevoir des requêtes tandis qu'un composant esclave est simplement adressé

par un maître afin de répondre à ses demandes. Un device ne peut pas émettre une requête par lui-même.

Le moteur de synchronisation SISSMA est un module SPACE qui contient une seule tâche. Le rôle de cette tâche est de vérifier, sur une base périodique, si une nouvelle requête lui est destinée. Si tel est le cas, cette tâche exécute la requête reçue et répond au module émetteur. Les types de requêtes possibles sont résumés dans le Tableau 4.1.

Tableau 4.1 Type de requêtes supportées par le modèle SystemC du moteur de synchronisation SISSMA

Type de requête	**Description**
Acquisition	La tâche désire acquérir un sémaphore ou attend un signal de synchronisation. (Comparable aux fonctions Pend() ou P() d'un sémaphore traditionnel.) La tâche est bloquée jusqu'à l'obtention du sémaphore ou à la réception du signal de synchronisation.
Libération	La tâche désire relâcher un sémaphore ou envoyer un signal de synchronisation. (Comparable aux fonctions Post() ou V() d'un sémaphore traditionnel.)
Tentative d'acquisition	Ce type de requête est identique à une acquisition, mais il n'est pas bloquant lors d'une réponse négative (sémaphore occupé).
État	La tâche désire connaître l'état du sémaphore.

Toutes requêtes vers le module SISSMA se concluent par une réponse, peu importe si elle est positive ou négative. Dans le cas d'une requête bloquante, la tâche se met en attente bloquante sur la réception d'un message provenant du module SISSMA. Lorsque ce dernier détermine que cette tâche doit être éveillée, il lui transmet un message qui, à sa réception, débloque la tâche en question.

Le diagramme de classe simplifié du modèle TLM du moteur de synchronisation SISSMA est illustré à la Figure 4.6. Tous les services du module ne sont pas utilisés en même temps. En fait, ceux-ci dépendent de la configuration du module. Ainsi,

les services des classes *DataTicketLockRw* et *SissmaDeviceRwLockHelper* sont utiles seulement si l'optimisation rédacteur/lecteur est active. Il en va de même pour la classe *SissmaDeviceTaskQueue* et l'optimisation permettant les attentes bloquantes. Le Tableau 4.2 résume le rôle de chaque classe.

Tableau 4.2 Rôle des classes du modèle TLM du moteur de synchronisation SISSMA

Classe	Rôle
SpaceBaseModule	Classe de base d'un composant maître dans SPACE.
SissmaDevice	Le moteur de synchronisation SISSMA.
SissmaDeviceBasicHelper SissmaDeviceRwLockHelper	Classes de support dans le traitement des requêtes.
DataTicketLock DataTicketLockRw	Registre d'état de chaque sémaphore basé sur l'utilisation de tickets d'accès.
SissmaDevicePayload SissmaDeviceResponse	Type de données utilisées lors d'une requête et d'une réponse, respectivement.
SissmaDeviceTaskQueue	File d'attente pour un sémaphore.
SissmaDeviceWaitingTaskInfo	Structure de données utilisées dans chaque élément d'une file d'attente.

4.4.3.2 Modèle VHDL

Le moteur de synchronisation implanté en VHDL correspond à l'implantation personnalisée de la bibliothèque SISSMA. Cette implantation se démarque principalement par son mode de communication indépendant de la plateforme SPACE (voir l'annexe IV pour plus de détails).

Communication

Le moteur de synchronisation est fait de deux composants principaux. Son noyau constitue le bloc central. Il implante l'ensemble de sa logique d'opération. Son interface assure son interaction avec les autres composants du système. La Figure 4.7 montre cet arrangement.

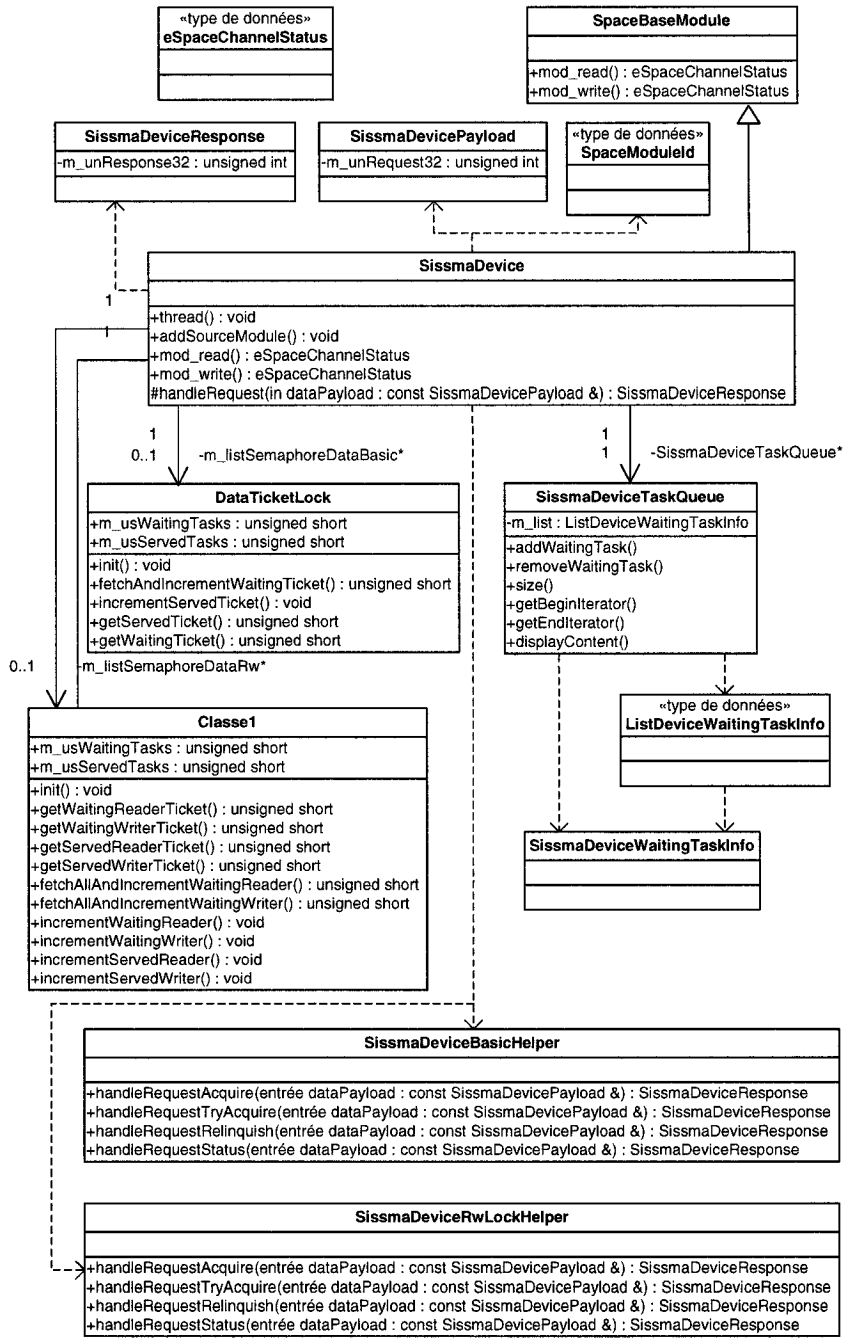


Figure 4.6 Diagramme de classe simplifié du modèle TLM du moteur de synchronisation SISSMA

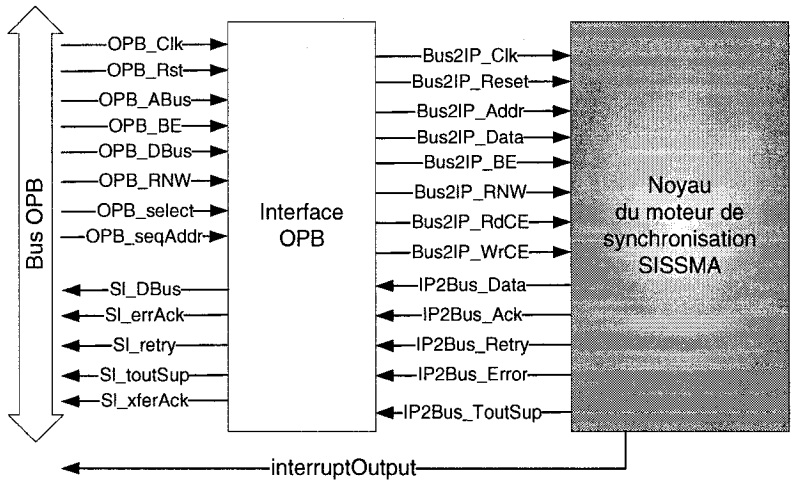


Figure 4.7 Interface de communication et noyau logique du moteur de synchronisation SISSMA

L'interface OPB utilisée permet au module de communiquer avec un bus OPB (« On-chip Peripheral Bus ») de la suite CoreConnect d'IBM. Cette séparation entre l'interface de communication et le noyau permet la réutilisation de ce dernier avec d'autres interfaces de communication. L'interface OPB est un bloc IP fourni avec l'outil *Platform Studio* de Xilinx. Il porte le nom d'*IPIF*.

Afin d'être atomiques, toutes requêtes vers le module SISSMA se font par le biais d'une lecture. C'est la plage d'adresse spécifiée par la requête qui identifie l'opération désirée. L'encodage de chaque type de requêtes est décrit en détail dans l'annexe V.

Structure du noyau

Le contrôleur principal du noyau du moteur de synchronisation est sensible à tous les événements sur ses signaux d'entrée. À chaque coup de l'horloge système, qui lui est transmis via le signal *Bus2IP_Clk*, il échantillonne les signaux *Bus2IP_RdCE* ou *Bus2IP_WrCE* qui, lorsqu'actifs, lui indiquent respectivement qu'une requête de lecture ou d'écriture est reçue. La commande de la requête est encodée dans

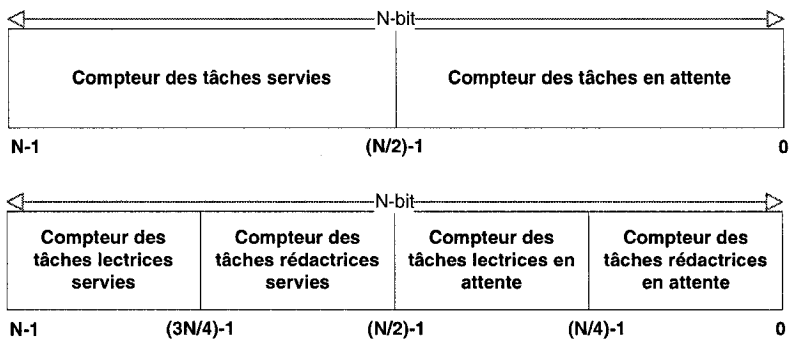


Figure 4.8 Structure du registre d'état d'un sémaphore du moteur de synchronisation SISSMA

l'adresse de la requête. Cette adresse correspond au signal *Bus2IP_Addr* qui occupe 32 bits. Un décodeur est utilisé afin d'extraire la commande et ses paramètres de la requête. Les commandes se distinguent à l'aide de la plage adresse lue du moteur de synchronisation.

Un fichier de registres emmagasine l'état de chaque sémaphore. Chaque sémaphore possède un registre dont la structure est illustrée à la Figure 4.8.

Le registre d'état est partitionné en deux lorsque les sémaphores de base sont utilisés. Chaque partie du registre correspond à un compteur. Celui de droite correspond au compteur des tâches en attente, soit le distributeur de tickets. Le second compteur marque le ticket de la dernière tâche qui a pris possession du sémaphore. Lorsque l'optimisation rédacteur-lecteur est active, chaque compteur du registre d'état d'un sémaphore est subdivisé de nouveau en deux afin de distinguer les compteurs pour les tâches lectrices et rédactrices. Cette organisation est requise afin d'utiliser les tickets pour la synchronisation. Ces compteurs assurent un accès ordonné au sémaphore.

Un composant constitué de multiplexeurs utilise l'identificateur de sémaphores, un paramètre présent dans chaque requête, afin de sélectionner le bon élément du fichier de registre d'état des sémaphores. Le contenu de ce registre est ensuite transmis au

contrôleur qui l'utilise pour répondre à une commande.

Un tableau contenant une série de files d'attente sert lors de la mise en attente bloquante de tâches. Ces files d'attente sont contrôlées par le contrôleur principal. Il en va de même pour le tableau des registres de spins. Ces registres servent à sauvegarder la durée des dernières sections critiques de chaque sémaphore. Le contrôleur se sert d'un compteur afin d'évaluer la durée de chaque section critique.

Enfin, un générateur d'interruption est responsable de l'émission des signaux d'interruption. Ces signaux sont utilisés pour éveiller les tâches en attente bloquante sur un sémaphore SISSMA. Ce générateur utilise les informations présentes dans les files d'attente et dans le fichier de registres d'état des sémaphores.

À ce titre, l'implantation VHDL du module SISSMA supporte une opération supplémentaire propre à son mode de communication différent de son modèle TLM compatible avec SPACE. Cette commande sert à identifier l'instance de l'objet de synchronisation SISSMA sur laquelle la tâche à débloquer est en attente.

Le mécanisme d'attente bloquante repose sur l'utilisation d'un sémaphore local au module et aussi local à l'instance SISSMA[4]. Ce sémaphore sert à bloquer une tâche en attente sur un sémaphore SISSMA. Sur chaque module, un tableau statique enregistre tous les objets de synchronisation SISSMA utilisés par ledit module. Comme suite à la réception du signal d'interruption utilisé pour éveiller une tâche bloquée, la routine d'interruption interroge le module SISSMA afin de connaître l'identificateur de l'instance SISSMA qui renferme le sémaphore bloquant à signaler. Une fois ce sémaphore local signalé, la tâche retrouve son état actif et peut poursuivre avec

[4]L'utilisation d'un sémaphore local unique pour chaque instance d'un objet de synchronisation SISSMA ne peut garantir l'éveil de la tâche appropriée si plusieurs tâches sont bloquées en attente sur un même sémaphore SISSMA à la fois. Toutefois, cette solution est temporaire et sera remplacée par les infrastructures de communication de SPACE à moyen terme, tel qu'expliqué dans l'annexe IV.

l'exécution de sa section critique.

La Figure 4.9 présente les principaux blocs qui composent ce module matériel.

Certains composants sont automatiquement présents dans le module SISSMA alors que d'autres sont facultatifs. Cette génération conditionnelle dépend de la configuration de ce dernier.

Configuration

Afin de minimiser la surface et la consommation d'énergie, l'implantation VHDL du module SISSMA utilise les clauses « generic » du langage de description matérielle VHDL. Ces clauses permettent la génération conditionnelle de composants matériels. Les options de configuration de ce module utilisent donc ces clauses. Le Tableau 4.3 liste les paramètres de configuration disponibles.

Le nombre de sémaphores, la largeur de leur registre d'état ainsi que la profondeur des files d'attente sont tous des paramètres variables et configurables. La largeur du registre d'état est fonction du nombre maximal de tâches qui requiert un même sémaphore à un même instant.

Le nombre de signaux d'interruption émis correspond au nombre de composants maîtres qui émettent des requêtes vers le module SISSMA, indépendamment du nombre de tâches que chaque composant maître contient.

En fonction de la valeur du paramètre C_ENABLE_OPTION_RWLOCK qui active ou non l'optimisation rédacteur-lecteur, un contrôleur central différent est généré. Aussi, dans l'objectif de réduire la surface matérielle requise, les files d'attente ne sont générées que si les attentes bloquantes sont activées. Il en va de même pour les registres de spin.

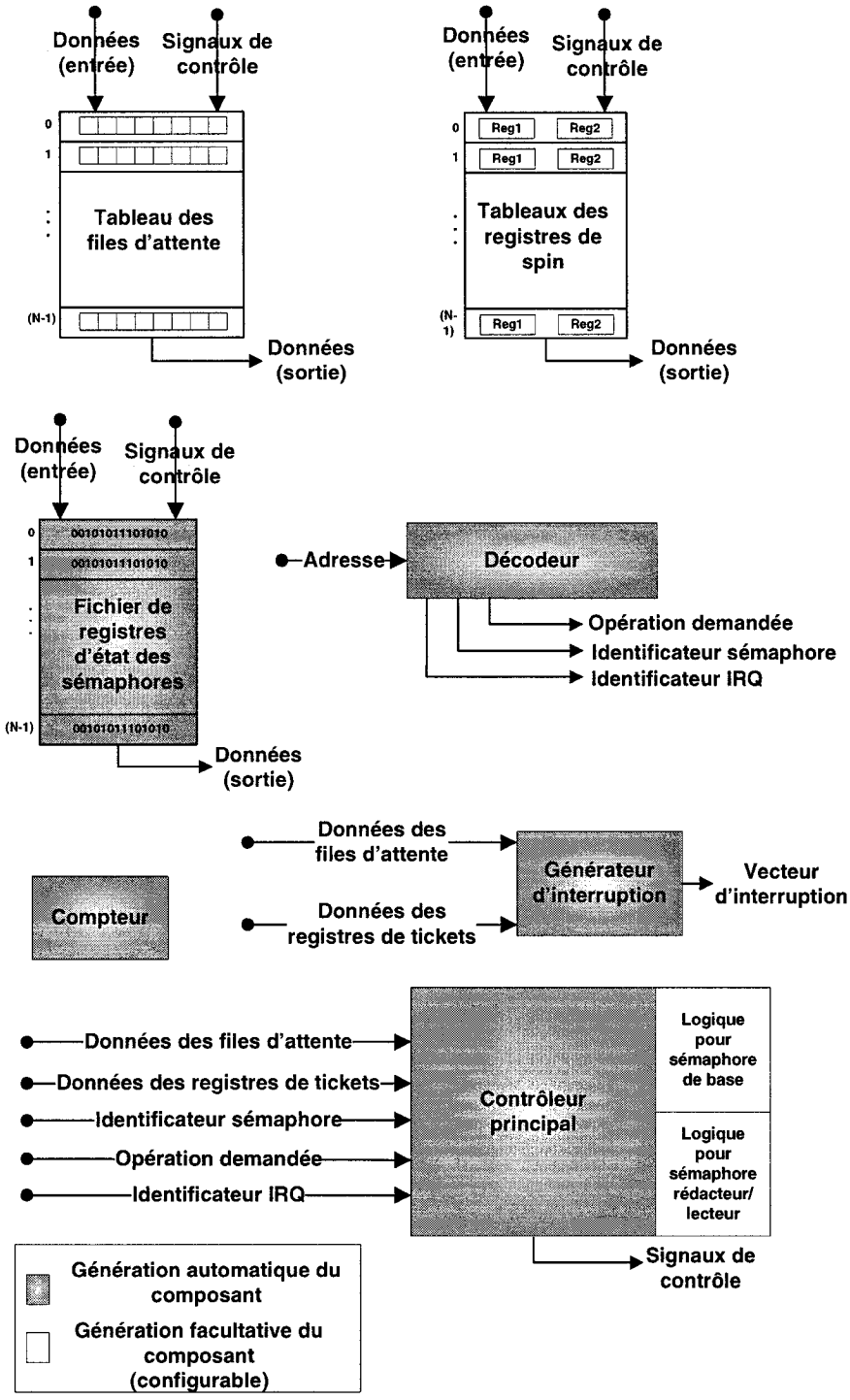


Figure 4.9 Composition matérielle du moteur de synchronisation SISSMA

Tableau 4.3 Description des paramètres de configuration du module SISSMA en VHDL

Paramètre	Valeur	Description
C_LOCK_COUNT	Entier positif	Nombre de sémaphores
C_LOCK_COUNTER_WIDTH	Entier positif	Largeur des registres d'état
C_LOCK_QUEUE_DEPTH	Entier positif	Profondeur des files d'attente
C_MAX_IRQ_LEVEL	Entier positif	Nombre de signaux d'interruption
C_ENABLE_OPTION_RWLOCK	{0, 1}	Activation de l'optimisation rédacteur-lecteur
C_ENABLE_OPTION_LONG_CS	{0, 1}	Activation de l'optimisation pour l'attente bloquante
C_ENABLE_SPINDELAY_REGISTERS	{0, 1}	Activation des registres de spins
C_SPINDELAY_CLOCK_DIVIDER	Entier positif; puissance de 2	Diviseur de l'horloge pour le compteur des registres de spins

4.5 Intégration dans la plateforme SPACE

La bibliothèque SISSMA dans son ensemble repose sur la plateforme SPACE et sur la bibliothèque SystemC. Le bon fonctionnement de SISSMA dépend de différents services offerts par SPACE. SISSMA tire profit des mécanismes de communication bloquants et non-bloquants supportés par SPACE. Ainsi, une attente active utilise les communications non-bloquantes, alors qu'une attente bloquante utilise les communications du même type. Bref, la bibliothèque SISSMA est fondée sur SPACE mais SPACE en reste indépendant.

La synchronisation locale fournie par SISSMA repose sur l'interface de programmation SystemC/RTOS de SPACE. De cette façon, même si la bibliothèque SISSMA utilise en apparence les services de sémaphores de SystemC, c'est le comportement logiciel

de RTOS sélectionné qui exécute. Il reste toutefois primordial que les sémaphores du RTOS sélectionné correspondent avec l'ordre d'accès FIFO établi dans la bibliothèque SISSMA. Dans le cas contraire, un changement de mode de synchronisation (local vs à distance) pourrait présenter une séquence d'exécution différente des tâches impliquées dans la synchronisation. C'est le cas entre autres des sémaphores natifs de SystemC qui ne sont aucunement déterministes quant à la tâche éveillée comme suite à la libération d'un sémaphore [39].

La synchronisation à distance de SISSMA requiert l'infrastructure de communication de SPACE. La communication entre une tâche et le moteur de synchronisation SISSMA, un module SPACE, se fait via la transmission de messages. Une tâche en attente active utilise des communications non bloquantes, alors qu'une tâche qui utilise l'attente bloquante se sert de la communication bloquante. Bref, la synchronisation à distance assurée par la bibliothèque SISSMA repose sur les infrastructures de communication de SPACE.

CHAPITRE 5

ÉVALUATION DES PERFORMANCES ET ANALYSE

Ce chapitre présente les performances de la bibliothèque SISSMA. Les performances de son contenu logiciel et matériel sont évaluées. D'abord, l'environnement d'analyse est introduit. Ensuite, les métriques statiques ainsi que les métriques dynamiques concluent. Ces dernièrs sont recueillies à partir d'une application effectuant la multiplication de deux matrices.

5.1 Environnement d'analyse

L'ensemble des données recueillies relève de l'exécution de multiples programmes d'évaluation sur une plateforme monoprocesseur ou multiprocesseur. Toutes les données proviennent de l'exécution réelle de ces systèmes-sur-puce. Bref, aucun résultat de simulation n'est rapporté ici. Les outils utilisés pour créer ces plateformes, leur architecture et leur configuration sont détaillés.

5.1.1 Outils et plateforme cible

Les outils *EDK* et *Platform Studio* en plus d'*ISE Foundation* de la compagnie Xilinx servent à la création des systèmes-sur-puce utilisés. *ISE Foundation* est dédié à la conception de composants électroniques. Il permet en autre la conception, la simulation, l'assignation des entrées/sorties et l'analyse de puissance ou temporelle. *EDK* quant à lui sert à la construction et à la conception d'un SoC embarqué. C'est d'abord un environnement où l'assemblage de multiples composants IP permet

la création d'une plateforme matérielle. Il offre aussi des outils de développement logiciel : compilateur et débogueur.

La version 8.1 de ces deux outils fut utilisée afin de concevoir le moteur de synchronisation SISSMA et les deux SoCs décrits ultérieurement. *XFlow* est l'outil de synthèse numérique intégré à l'outil ISE.

Un circuit intégré reprogrammable de type FPGA [100][101] constitue la plateforme cible où les SoCs exécutent. La carte AP130 de la compagnie Amirix sert de plateforme de prototypage et d'analyse. Cette carte électronique contient un FPGA de type Virtex-II Pro, le VP30. Cette puce FPGA dispose de 13696 slices[1] et de 2448 kbits de mémoire locale, ou BRAM.

Tableau 5.1 Caractéristiques de la puce FPGA utilisée

Élément	**Valeur**
Famille du FPGA	Virtex-II Pro
Type	VP30
Grade	6
Emballage (« packaging »)	ff896

5.1.2 Architecture et configuration des systèmes-sur-puce d'analyse

Bien que la puce FPGA de la carte électronique AP130 dispose de deux processeurs PowerPC 405, les systèmes-sur-puce créés reposent sur le processeur présynthétisé MicroBlaze. Ce processeur utilise donc les ressources configurables du FPGA lors de son instanciation.

Une première plateforme, celle-ci monoprocesseur, est présentée à la Figure 5.1.

[1]Une slice du FPGA VP30 contient deux registres et deux tables de vérité à quatre entrées. Ces tables sont configurables et peuvent servir aussi en tant que registre à décalage ou d'unité mémoire.

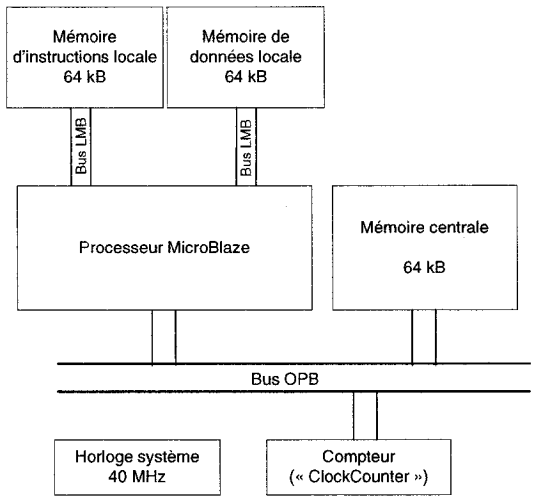


Figure 5.1 Architecture de la plateforme d'analyse monoprocesseur utilisée

Le processeur MicroBlaze interagit avec l'ensemble du système à l'aide d'un bus OPB. Ce bus donne accès à une mémoire centrale de 64 kB. Le MicroBlaze a aussi un accès direct avec deux autres mémoires, celles-ci de 64kB chacune, à partir de bus LMB dédiés. Ces mémoires locales au processeur sont plus rapides d'accès qu'une mémoire centrale disposée sur le bus OPB. Ces deux mémoires locales emmagasinent le programme et ses données lors de l'exécution. Les trois mémoires présentes sont créées à partir de mémoire BRAM, locales à la puce FPGA. Ainsi, aucun accès en mémoire externe n'est requis.

La seconde plateforme contient les mêmes éléments. Toutefois, le groupe qui contient le processeur et ses deux mémoires locales est dupliqué afin d'obtenir une solution multiprocesseur. La Figure 5.2 montre cette seconde architecture.

Un module additionnel est présent dans ces deux architectures: le compteur (« ClockCounter »). Ce module est un périphérique utilisé afin d'évaluer le temps d'exécution d'un algorithme sur la plateforme. Ce compteur compte le nombre de coups d'horloge qui a lieu depuis le moment où il est mis à zéro. Le contenu de ce

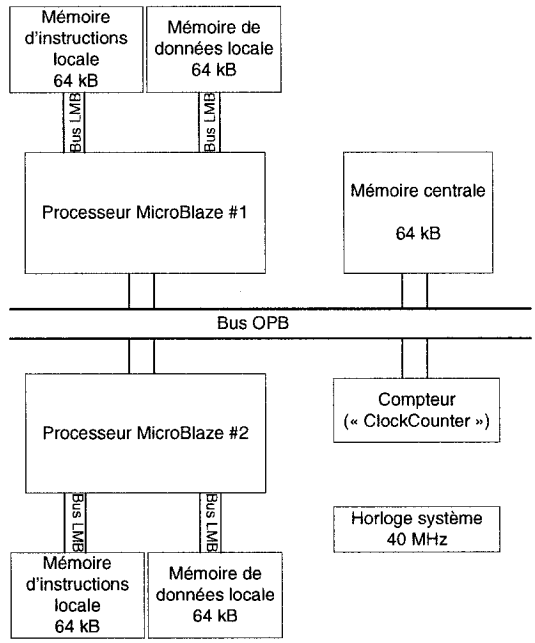


Figure 5.2 Architecture de la plateforme d'analyse multiprocesseur utilisée

compteur se limite à huit registres de 32 bits, tel qu'illustré à la Figure 5.3. Ces huit registres constituent en fait un seul registre de 256 bits. Ils sont séparés afin d'être lus à l'exécution, le bus OPB étant limité à des requêtes de 32 bits à la fois.

Ces deux SoCs utilisent une horloge système cadencée à 40 MHz. L'horloge provient d'un oscillateur présent sur la carte électronique AP130. Ce signal est ensuite transmis à un DCM présent sur la puce FPGA, dont le rôle est de distribuer le signal d'horloge à

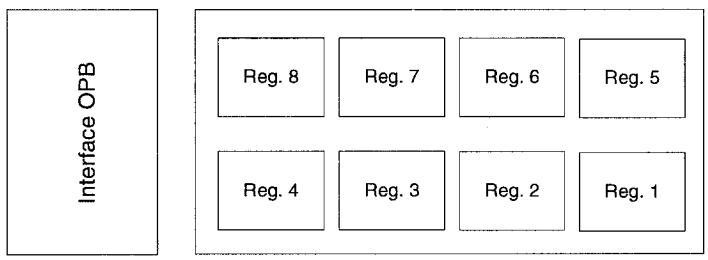


Figure 5.3 Organisation du compteur du temps d'exécution

travers la puce FPGA. Le même signal d'horloge sert pour l'ensemble des composants du système.

Le processeur MicroBlaze dispose d'options de configuration. Ces options affectent ses unités arithmétiques et permettent d'accélérer certaines opérations à faible coût en termes de surface matérielle. Les processeurs MicroBlaze sont configurés afin de disposer de matériels additionnels optimisant la multiplication, la division, les opérations de comparaison et les décalages. Un décalage sur un nombre variable de bits requiert désormais une seule instruction assembleur, une optimisation profitable à l'API SISSMA qui l'utilise lors du formatage d'une requête destinée au moteur de synchronisation SISSMA. Lorsque ces optimisations sont activées, il va de soi que le compilateur de MicroBlaze doit être configuré de la même manière afin de générer les nouvelles instructions supportées.

Les processeurs MicroBlaze ne disposent pas de mémoire cache. Toutefois, leurs mémoires locales de données et d'instructions offrent un temps d'accès équivalent. Aussi, les processeurs MicroBlaze ne peuvent pas recevoir d'interruption dans ces deux configurations.

5.2 Métriques statiques

Les métriques statiques correspondent aux données qui qualifient les performances des composants SISSMA. Ces métriques ne dépendent pas d'une application particulière. Les métriques logicielles correspondent au temps d'exécution des différents services de synchronisation tandis que les métriques matérielles concernent la consommation de surface.

5.2.1 Métriques logicielles

5.2.1.1 Méthode d'évaluation

Un programme de test sert à l'évaluation de tous les services de synchronisation de l'API SISSMA en fonction des différentes optimisations. Le temps d'exécution rapporté est en nombre de cycles d'horloge. L'horloge utilisée correspond à l'horloge principale du SoC cadencée à 40 MHz.

Afin d'obtenir une meilleure précision, les programmes d'analyse utilisent une boucle qui exécute dix millions d'itérations. Ainsi, l'opération évaluée est exécutée dix millions de fois. Ce nombre élevé d'itérations rend négligeable le temps requis pour la lecture des compteurs affichant le temps d'exécution du test. Un échantillon d'un programme de test utilisé pour l'évaluation des fonctions OSSemPend et OSSemPost du système d'exploitation temps réel MicroC/OS-II est montré ci-dessous. Tous les autres programmes d'analyse se basent sur une structure similaire.

```
resetExecutionTimeCounter();                    // Reset Clock Counter
for (unsigned int i = 0 ; i < METRIC_REPETITION_COUNT; i++) {
    OSSemAccept(pSemaphoreMicroC);
    OSSemPost(pSemaphoreMicroC);
}
getExecutionTime();
```

5.2.1.2 Analyse des métriques statiques

Les résultats présentés pour chaque opération incluent l'ensemble du traitement requis pour envoyer une requête de synchronisation, c'est-à-dire l'initialisation de la requête, son traitement et enfin son post-traitement (gestion des erreurs).

Les services de synchronisation offerts par la bibliothèque SISSMA s'appliquent

non seulement dans un contexte de conception au niveau système, mais aussi afin de permettre des plateformes multiprocesseurs qui ne disposent pas d'un support particulier pour la synchronisation. Dans cet ordre d'idée, les performances de synchronisation de la bibliothèque sont comparées avec celle d'un système d'exploitation temps réel, le RTOS MicroC/OS-II. Le Tableau 5.2 présente les performances de ce RTOS. L'opération *accept* se comporte comme un *pend* à l'exception que si le sémaphore est utilisé, la fonction l'indique par sa valeur de retour sans bloquer la tâche qui émet la requête. La fonction *query* quant à elle permet de connaître l'état d'un sémaphore.

Tableau 5.2 Performance de synchronisation offerte par le RTOS MicroC/OS-II

Opérations	**Nbre de cycles d'exécution (10M itérations)**	**Marge d'erreur (10M itérations)**	**Nbre de cycles par itération**
Pend/Post	1430000010	4	143
Accept/Post	1250000010	4	125
Query (État: libre)	1030000011	4	103
Query (État: occupée)	1030000011	4	103

Les données complètes pour chaque type de synchronisation sont incluses en annexe VII. Seuls les tableaux comparatifs sont inclus dans les sections qui suivent.

Synchronisation locale sans optimisation

La synchronisation locale de la bibliothèque SISSMA repose sur l'utilisation d'un API SystemC/RTOS. Ce sont donc les mêmes fonctions de MicroC, appelées indirectement par la bibliothèque SISSMA. Ces performances sont illustrées dans le Tableau 5.3 où tous les cas d'acquisition d'un sémaphore sont inclus: section critique courte et longue, avec ou sans promotion possible vers une section critique longue. Outre le fait que la bibliothèque SISSMA exécute des vérifications additionnelles, la surcharge d'exécution est faible considérant le court temps d'exécution.

Tableau 5.3 Tableau comparatif de la synchronisation locale sans optimisation

Opérations MicroC	CPI*	Opérations SISSMA	CPI*	% Surcharge
Pend/Post	143	Lock/Unlock	193	34.97
Accept/Post	125	Trylock/Unlock	169	35.2
Query (État: libre)	103	isLocked(État: libre)	115	11.65
Query (État: occupée)	103	isLocked(État: occupée)	115	11.65

* Nombre de cycles par itération

Synchronisation locale de type rédacteur/lecteur

L'analyse des performances de la synchronisation locale configurée avec l'optimisation rédacteur/lecteur démontre l'intérêt d'utiliser un RTOS local dont le mode d'attente est compatible avec celui de la bibliothèque SISSMA.

Effectivement, le RTOS MicroC utilise un mode d'attente basé sur la priorité des tâches, tandis que SISSMA est présentement implanté pour le support du mode FIFO. Afin de rendre les deux protocoles d'attente compatible, il est requis d'ajouter des mécanismes de contrôle par-dessus les simples sémaphores de SystemC, qui sont en fait des sémaphores MicroC. Ces mécanismes assurent l'éveil des tâches du bon type (rédacteur ou lecteur). Toutefois, ceux-ci sont temporaires et servent seulement à assurer la vérification fonctionnelle de la bibliothèque SISSMA. Bref, ces mécanismes additionnels imposent une grande surcharge opérationnelle, tel que reflété dans le Tableau 5.4.

Synchronisation à distance sans optimisation

La synchronisation à distance implique le formatage d'une requête qui est ensuite transmise au moteur de synchronisation SISSMA. Ce dernier retourne une réponse qui est ensuite traitée par l'API SISSMA. La majorité de la charge d'exécution se situe au niveau du formatage et du traitement de la réponse. La transmission de la

Tableau 5.4 Tableau comparatif de la synchronisation locale avec l'optimisation rédacteur/lecteur

Opérations MicroC	CPI*	Opérations SISSMA	CPI*	% Surcharge
Pend/Post	143	Lock/Unlock (Courte, Promo: Non)	532	272.03
Pend/Post	143	Lock/Unlock (Longue, Promo: Non)	477	233.57
Pend/Post	143	Lock/Unlock (Courte, Promo: Oui)	533	272.73
Pend/Post	143	Lock/Unlock (Longue, Promo: Oui)	477	233.57
Accept/Post	125	Trylock/Unlock (Courte)	533	326.4
Accept/Post	125	Trylock/Unlock (Longue)	477	281.6
Query (État: libre)	103	isLocked(État: libre)	198	92.23
Query (État: occupée)	103	isLocked(État: occupée)	198	92.23

* Nombre de cycles par itération

requête et son traitement par le moteur matériel ne requièrent qu'une opération de lecture. Sur le bus OPB, une lecture requiert quatre cycles d'horloge.

Tel qu'illusté au Tableau 5.5, pour les opérations d'acquisition et de libération d'un sémaphore, une synchronisation à distance demande environ 270 cycles d'horloge, en fonction des options utilisées. Cela représente un temps d'exécution légèrement inférieur au double d'une synchronisation locale. Pour les opérations moins fréquentes, par exemple celles de type *Accept* ou *Query*, la solution proposée affiche une accélération, d'où la surcharge négative.

Synchronisation à distance de type rédacteur/lecteur

Lorsque l'optimisation rédacteur/lecteur est activée, les performances restent sensiblement les mêmes (Figure 5.6): l'acquisition et la libération demandent en moyenne dix cycles de plus lors du formatage de la requête alors que les autres opérations ne

Tableau 5.5 Tableau comparatif de la synchronisation à distance sans optimisation

Opérations MicroC	CPI*	Opérations SISSMA	CPI*	% Surcharge
Pend/Post	143	Lock/Unlock (Courte, Promo: Non)	261	82.52
Pend/Post	143	Lock/Unlock (Longue, Promo: Non)	272	90.21
Pend/Post	143	Lock/Unlock (Courte, Promo: Oui)	271	89.51
Pend/Post	143	Lock/Unlock (Longue, Promo: Oui)	276	93.01
Accept/Post	125	Trylock/Unlock (Courte)	79	-36.8
Accept/Post	125	Trylock/Unlock (Longue)	90	-28
Query (État: libre)	103	isLocked(État: libre)	47	-54.37
Query (État: occupée)	103	isLocked(État: occupée)	47	-54.37

* Nombre de cycles par itération

sont pas affectées.

5.2.2 Métriques matérielles

Les métriques matérielles propres au moteur de synchronisation SISSMA sont présentées dans cette section. La surface qu'il occupe dans le puce FPGA est montrée en plus du nombre de portes logiques équivalentes. Des données détaillées quant au nombre de *slices*, de *slices FF*[2] et de *LUT* à quatre entrées sont incluses. Enfin, la période minimale et la fréquence maximale d'opération possible pour la puce FPGA utilisée sont aussi incluses. Pour des raisons d'espaces, toutes les métriques sont reproduites dans l'annexe VIII. *Synplify* est l'outil de synthèse utilisé pour recueillir ces résultats. *Synplify* est configuré afin d'optimiser le circuit en vitesse d'opération

[2]Une slice peut prendre différentes configurations. Lorsqu'elle est utilisée pour ses registres, elle prend le qualificatif de slice FF. Aussi, il est possible de combiner les deux LUT à deux entrées d'une même slice afin de créer une LUT à quatre entrées.

Tableau 5.6 Tableau comparatif de la synchronisation à distance avec l'optimisation rédacteur/lecteur

Opérations MicroC	CPI*	Opérations SISSMA	CPI*	% Surcharge
Pend/Post	143	Lock/Unlock (Courte, Promo: Non)	277	93.71
Pend/Post	143	Lock/Unlock (Longue, Promo: Non)	275	92.31
Pend/Post	143	Lock/Unlock (Courte, Promo: Oui)	286	100
Pend/Post	143	Lock/Unlock (Longue, Promo: Oui)	287	100.7
Accept/Post	125	Trylock/Unlock (Courte)	92	-26.4
Accept/Post	125	Trylock/Unlock (Longue)	92	-26.4
Query (État: libre)	103	isLocked(État: libre)	47	-54.37
Query (État: occupée)	103	isLocked(État: occupée)	49	-52.43

* Nombre de cycles par itération

(fréquence cible: 120 MHz).

La surface requise pour implanter le moteur de synchronisation SISSMA varie en fonction des cinq paramètres et des deux options de configuration. Pour les quatre combinaisons possibles des options de configuration, seuls les paramètres les plus probables de varier sont évalués afin de réduire le nombre de combinaisons possiblement très élevé. Enfin, la surface consommée correspond aux ressources du noyau logique de moteur de synchronisation. Ces métriques n'incluent pas la surface demandée par l'interface de communication, l'IPIF OPB.

Configuration: désactivation de l'optimisation rédacteur/lecteur et des sections critiques longues

Cette configuration est la plus simple en termes de complexité. Cette simplicité se traduit par une consommation plus faible en ressource matérielle quant aux registres

requis et à la logique de contrôle simplifiée. L'absence de file d'attente contribue aussi à réduire la surface. Le cas le plus simple permet une cadence excédant 120 MHz. De l'autre côté du spectre, l'utilisation de tickets de 8 bits (LockCounterWidth vaut alors 16 bits) et de 32 objets de synchronisation permet au module matériel d'opérer à plus de 80 MHz.

Configuration: désactivation de l'optimisation rédacteur/lecteur et activation des sections critiques longues

L'activation des sections critiques longues se traduit par l'ajout des files d'attente à la logique du moteur de synchronisation. Chaque sémaphore matériel dispose donc d'une file ayant la profondeur voulue. La surface consommée est alors grandement fonction de la capacité-mémoire requise par ces registres. Le chemin critique est aussi affecté par l'ajout de logique. Un contrôleur et un ensemble de comparateurs doivent assurer la gestion de ces files d'attente en plus du signalement de tâches en attente afin de les éveiller. En comparaison avec la configuration précédente, l'ajout des files d'attente et de cette logique de contrôle additionnelle représente environ l'équivalent de 1000 portes logiques. La fréquence d'opération maximale chute à environ 105 MHz.

Configuration: activation de l'optimisation rédacteur/lecteur et désactivation des sections critiques longues

L'optimisation rédacteur/lecteur affecte directement l'implantation du contrôleur principal du composant matériel. Toutefois, il n'affecte pas la consommation de mémoire (nombre de registres) utilisée. Les registres utilisés conservent leur largeur, mais chaque ticket est subdivisé en deux, tel qu'expliqué dans la section 4.4.3.2. Cette subdivision diminue de moitié le nombre de tâches qui peuvent attendre sur un même sémaphore à un moment donné. Cette logique de contrôle offre tout de même des performances très similaires à la première configuration à un coût très faible: une

centaine de portes logiques équivalentes.

Configuration: activation de l'optimisation rédacteur/lecteur et des sections critiques longues

Cette dernière configuration résulte de la combinaison des deux dernières. Elle requiert la plus grande surface. Dans le pire des cas évalués, les plateformes conçues pour opérer à une fréquence de 40 MHz ne sont nullement restreintes par le chemin critique du moteur de synchronisation SISSMA.

5.3 Métriques dynamiques

5.3.1 Application d'analyse: multiplication de matrices

La multiplication de matrices sert d'application d'analyse afin d'évaluer les performances d'une plateforme multiprocesseur et des services de synchronisation offerts par la bibliothèque SISSMA.

Le code source de cet algorithme est montré ci-dessous.

```
template<class T>
void
thread_ComputeMatrixMultiplication_Standalone (
        const char* pStrThreadName,
        SissmaDataArray2d<T, PARTITIONING_DATA_ACCESS>* pMatrixOperandA,
        SissmaDataArray2d<T, PARTITIONING_DATA_ACCESS>* pMatrixOperandB,
        SissmaDataArray2d<T, PARTITIONING_DATA_ACCESS>* pMatrixResult,
        unsigned short  usTotalThreadCount,
        unsigned short  usThreadId)
{
  // Should be a square matrix
  const unsigned int unMatrixSizeX =
                    pMatrixResult->get_array_row_count();
  // Number of rows processed by this thread
  const unsigned int unRowSubsetCount =
                    unMatrixSizeX/usTotalThreadCount;
  // Base row where to start computation
```

```
  const unsigned int unLowerBound =
                    (usThreadId) * unRowSubsetCount;

  // Compute Matrix multiplication to local data Matrix
  int i, j, k;

  // l is used to index tempArray only if the local copy of
  //   the tempArray is used for the partial result
  int l = ( usTotalThreadCount > 1
      ? unLowerBound
      : 0);

  T tOpA, tOpB, tOpResult;
  for( int i = unLowerBound ;
       i < (usThreadId+1) * unRowSubsetCount ;
       i++, l++)
  {
    for( j = 0 ; j < unMatrixSizeX ; j++ )
    {
      tOpResult = 0;
      for( k = 0 ; k < unMatrixSizeX ; k++ )
      {
        pMatrixOperandA->read(tOpA, i, k, NULL, false);
        pMatrixOperandB->read(tOpB, k, j, NULL, false);

        tOpResult += tOpA * tOpB;
      }
      (*pMatrixResult)(tOpResult, l, j, NULL, false);
    }
  }
}
```

Chaque processeur porte un identificateur unique. Cet identificateur est utilisé afin de partitionner l'ensemble des rangées de la matrice *résultante.* Bref, chaque processeur traite une portion du résultat. Tous les processeurs impliqués dans l'opération mettent ensuite leur résultat en commun.

La mise en commun des résultats permet la distinction de deux approches différentes. Dans la première approche, appelée « copy back », chaque processeur utilise une matrice temporaire située dans une mémoire locale au processeur afin d'y emmagasiner les résultats de ses calculs. Lorsqu'ils ont terminé de traiter les cellules dont ils sont responsables, les processeurs copient une portion de leur matrice

locale dans la matrice *résultante* qui est globale et partagée. Le processeur de synchronisation requis afin d'obtenir accès à la matrice *résultante* ne se fait qu'une seule fois. L'approche « write through » n'utilise pas de matrice temporaire. Aussitôt qu'une donnée est prête à être inscrite dans la matrice résultante, le processeur en question procède à cette écriture. Ces deux approches varient donc le nombre d'accès en mémoire centrale ainsi que le nombre d'opérations de synchronisation requises par la bibliothèque SISSMA.

5.3.2 Accélération par l'utilisation de plusieurs processeurs

Dans l'objectif de quantifier la charge d'exécution de la bibliothèque SISSMA, une comparaison des performances de l'exécution de la multiplication de matrices sur une plateforme à processeur unique (Figure 5.1) et sur la plateforme multiprocesseur (Figure 5.2), toutes deux décrites antérieurement, est de mise.

Afin de respecter le format des mémoires locales et globales utilisées, des matrices de format 10x10 sont utilisées. Aussi, différentes structures de données sont utilisées lors de l'exécution de l'application: *MatrixShared* et *SissmaDataArray2d*. *MatrixShared* est une classe qui encapsule une matrice bidimensionnelle alors que *SissmaDataArray2d* contient le même type de matrice, mais bénéficiant du support de synchronisation de la bibliothèque SISSMA. Puisque la surcharge de ces deux structures de données diffère, il est requis de les distinguer.

Les Figures 5.7 et 5.8 présentent le temps d'exécution requis lorsque différents types de synchronisation et de modes d'opération sont utilisés afin de multiplier deux matrices sur le SoC monoprocesseur et multiprocesseur respectivement.

Dans le premier scénario (cas 1), aucune synchronisation n'est faite sur les données en entrée, puisqu'elles ne sont que lues. Dans un second scénario (cas 2), une

Tableau 5.7 Temps d'exécution de la multiplication de deux matrices sur la plateforme monoprocesseur

Synchronisation	Format des matrices	Type de matrice	Nbre de cycles d'exécution
Aucune	10x10	MatrixShared	65075
Aucune	10x10	SissmaDataArray2d	147854

Tableau 5.8 Temps d'exécution de la multiplication de deux matrices sur la plateforme multiprocesseur

Synchronisation*	Format des matrices	Type de matrice	Cas 1 (NCE**)	Cas 2 (NCE**)
SissmaSync (Copy-Back)	10x10	MatrixShared	45129	339880
SissmaSync (Write-Through)	10x10	MatrixShared	73785	339792
SissmaDataArray2d (Copy-Back)	10x10	SissmaDataArray2d	89693	369630
SissmaDataArray2d (Write-Through)	10x10	SissmaDataArray2d	90525	369381

* Synchronisation à distance seulement
** Nombre de cycles d'exécution

synchronisation est faite à chaque lecture de ces données à l'aide des services de synchronisation des classes *SissmaSync* et *SissmaDataArray2d* (section 4.4.2.1). Clairement, ces deux scénarios illustrent la nécessité d'optimiser une application afin de minimiser le nombre d'opérations de synchronisation, peu importe son temps d'exécution. Autrement, le temps de synchronisation domine sur le temps de calcul utile.

Le Tableau 5.9 compare les gains d'accélération des deux types de matrice. Le mode d'exécution *CopyBack* sert lors des comparaisons.

Dans une situation idéale, l'utilisation d'un deuxième processeur procure un gain

Tableau 5.9 Tableau comparatif des gains d'accélération par l'utilisation d'une plateforme multiprocesseur pour la multiplicaiton de deux matrices.

Type de matrice	Gain d'accélération
MatrixShared	1,44
SissmaDataArray2d	1,65

d'accélération de deux. Bref, l'ensemble des opérations requiert la moitié du temps initial afin de se compléter. Ainsi, cela correspond à un temps d'exécution idéal de 32538[3] cycles. Bien entendu, la situation idéale ne se retrouve pas dans un système réel, puisque nombre de contraintes sont exclues de l'analyse.

Le premier cas, où la synchronisation *SissmaSync* (*CopyBack*) est utilisée de pair avec des matrices de type *MatrixShared.* Il apparaît que 9039[4] cycles sont requis afin de copier le tableau temporaire du résultat de la mémoire locale au processeur jusqu'à la mémoire centrale, incluant le temps requis pour synchroniser l'accès à cet emplacement-mémoire partagé. En éliminant cette surcharge d'opération propre à la plateforme multiprocesseur, le nouveau temps de calcul correspond à 36090[5] cycles d'horloge. Ce temps d'exécution, dont l'environnement est considéré similaire à la situation idéale, donne un gain d'accélération de 1.80.

Enfin, il convient de tenir compte d'une dernière série de facteurs dont l'impact diminue le gain d'accélération attendu. D'abord, un temps non nul est requis afin que les deux processeurs synchronisent leur exécution. Aussi, bien que chaque processeur dispose de mémoires locales, la contention présente par des accès aux mêmes données centrales ralentit leur exécution respective.

[3]Dans une situation idéale, le temps requis sur une plateforme monoprocesseur est divisé par deux lorsque le travail est divisé entre deux processeurs identiques: $65075/2 = 32538$.

[4]Le temps requis pour la synchronisation est obtenu suite à l'exécution de l'algorithme de multiplication de matrices avec et sans synchronisation, incluant le transfert du résultat en mémoire centrale.

[5]Résulte de la somme du temps d'exécution idéal et de la surcharge suite à la synchronisation et au transfert du résultat en mémoire centrale.

5.4 Conclusion

La bibliothèque SISSMA représente une solution de synchronisation pour un environnement de conception au niveau système. Les analyses montrées identifient certains facteurs clés. L'architecture d'une plateforme a une influence importante sur les performances globales. D'une manière plus spécifique, le type de synchronisation et la fréquence d'utilisation de ce type de service influent aussi directement sur les performances d'une application en particulier, tel qu'illustré dans le cas de la multiplication de deux matrices. Une stratégie de conception au niveau système se doit de tenir compte de ces facteurs lors de son optimisation.

CONCLUSION

Les systèmes embarqués sont aujourd'hui omniprésents. Les concepteurs de systèmes embarqués sont soumis à des pressions constantes afin de concevoir les systèmes-sur-puce qui fournissent un nombre croissant de fonctionnalités, tout en offrant des performances améliorées. Les systèmes-sur-puce multiprocesseurs répondent à ces besoins, tout en imposant une plus grande complexité et des coûts de conception et de fabrication plus élevés. De surcroît, le gain en importance du contenu logiciel de ces systèmes demande que ce développement logiciel débute plus tôt dans le flot de conception afin de rencontrer les courtes fenêtres d'accès au marché.

De nouvelles méthodes de conception préconisent à l'utilisation de plateforme virtuelle. Ces plateformes permettent la création d'un modèle du système. Ce modèle permet, non seulement de débuter la conception logicielle avant la disponibilité de la couche matérielle du système, mais offre aussi une meilleure conception globale du système, particulièrement son partitionnement logiciel/matériel. La plateforme SPACE, développée par le laboratoire de systèmes temps-réel et embarqués de l'École Polytechnique de Montréal permet, entre autres, la création de plateformes virtuelles dans le but de concevoir des systèmes-sur-puce au niveau système.

La méthodologie de conception ESL de la plateforme SPACE repose sur l'assemblage de composants IP. Cette approche offre une très grande flexibilité. Toutefois, afin de concevoir des MPSoCs, l'assemblage pur et simple de plusieurs processeurs reliés par un réseau d'interconnexion ne suffit pas. Essentiellement, des services de synchronisation sont aussi requis afin d'assurer l'exécution cohérente de tels MPSoC, puisque les composants IP utilisés ne les offrent pas.

Une analyse en profondeur de la littérature développée pour les systèmes multiprocesseurs a permis de mettre en évidence certains éléments d'implantation:

- Un grand nombre de composants d'un système peuvent être impliqués dans le processus de synchronisation d'une plateforme multiprocesseur.
- L'implantation optimale des services de synchronisation varie grandement en fonction du support matériel et logiciel fourni par la conception d'une plateforme multiprocesseur.
- L'architecture logicielle et matérielle influence le choix d'un algorithme de synchronisation optimal.
- La recherche d'une solution générique applicable dans le contexte de la plateforme SPACE appelle à un compromis flexibilité vs performance optimale.

Conformément à ses objectifs, la bibliothèque de synchronisation SISSMA permet l'assemblage d'une plateforme multiprocesseur à l'aide de composants IP ne diposant pas d'un tel support. Il est montré que l'utilisation de processeurs MicroBlaze dans un MPSoC, un processeur qui ne dispose pas de supports spécialisés pour un environnement multiprocesseur, en conjonction avec la bibliothèque SISSMA offre une solution fonctionnelle. Cette bibliothèque répond au besoin de SPACE de permettre la synchronisation lors de la conception de l'application au niveau système. Il est pertinent de noter l'approche nouvelle utilisée pour déterminer le mode d'attente d'une tâche à l'aide du concept de promotion. De surcroît, la mise en application de concepts établis, tels l'utilisation du ticket et des accès rédacteur/lecteur, améliore les performances globales du système. De plus, puisque la bibliothèque conçue dépend uniquement de son module matériel et de son API logicielle, elle est indépendante du support matériel de la plateforme cible. Le moteur de synchronisation SISSMA offre un nombre d'options de configuration de permettent d'optimiser sa surface tout en réduisant la puissance consommée.

La bibliothèque SISSMA se présente sous deux versions. Elle est d'abord utilisable afin de concevoir un système avec l'aide de la plateforme SPACE. Tous les services

offerts, l'API et le moteur synchronisation, sont implantés dans le langage C++ à partir de la bibliothèque SystemC. Une version personnalisée est aussi disponible pour une utilisation dans une implantation physique, par exemple sur une puce de prototypage de type FPGA. L'analyse des performances des services de synchronisation SISSMA montre que la synchronisation locale est minimalement affectée par la couche logicielle supplémentaire de SISSMA. Quant à la synchronisation à distance, elle requiert moins du double du temps d'une synchronisation locale lorsque comparé au système d'exploitation temps-réel MicroC/OS-II.

Travaux Futurs

Les apprentissages faits au cours des travaux de recherche mettent en lumière certains travaux futurs qui pourraient en découler.

Dans sa version actuelle, la configuration des objets de synchronisation de la bibliothèque SISSMA se fait manuellement par les développeurs. Il serait avantageux que cette configuration du type de synchronisation (locale vs à distance) puisse se faire d'une façon automatisée. Cette approche consisterait à intégrer cette fonctionnalité dans *SpaceStudio*[6], puisque c'est ce dernier outil qui connaît le résultat du partitionnement logiciel/matériel, résultat qui guide la configuration des objets SISSMA.

Toujours afin de faciliter l'utilisation de cette bibliothèque, une nouvelle piste de développement consisterait à évaluer le potentiel des patrons de synchronisation dans le contexte de la conception de systèmes électroniques à haut niveau. Ces patrons de conception représentent des algorithmes dont la solution répond à un problème rencontré fréquemment en informatique. Il existe de tels patrons dédiés

[6] *SpaceStudio* est un environnement de conception (IDE) basé sur une interface graphique. Il permet la création de modules en SystemC et la conception d'un système-sur-puce. Essentiellement, il constitue la porte d'accès à une utilisation graphique de la plateforme SPACE.

aux systèmes multiprocesseurs et distribués [102]. La nouvelle piste de recherche se solderait probablement par une intégration des patrons à la plateforme SPACE dont SISSMA assure la synchronisation à l'interne.

Les fondations établies par la bibliothèque SISSMA offrent des nouvelles opportunités en termes de protocoles de synchronisation. Ainsi, il est possible d'étendre les services de synchronisation présentement disponibles. La synchronisation de base de SISSMA peut servir à protéger un emplacement-mémoire partagé accédé par un protocole de synchronisation non supporté par le moteur de synchronisation SISSMA. Par exemple, un protocole de synchronisation reposant sur l'utilisation de queues d'attente priorisées peut emmagasiner celles-ci en mémoire partagée et synchroniser son accès à travers SISSMA. Dans le même ordre d'idée, les fondations actuelles offrent un environnement fonctionnel afin d'identifier de nouveaux protocoles adaptatifs.

Bien que l'implantation des protocoles de synchronisation se doit d'être générique dû à son contexte d'utilisation dans la plateforme SPACE, il serait bénéfique que celle-ci puisse tirer profit de mécanismes matériels ou architecturaux de la plateforme cible s'ils sont disponibles. Par exemple, si un processeur dispose d'instructions atomiques spécialisées (ex.: LoadLinked/Store), les performances de synchronisation ne pourraient qu'être améliorées par son utilisation. Il faut toutefois que la bibliothèque SISSMA soit en mesure de les utiliser. De la même façon, une architecture, dont les processeurs disposent d'une mémoire locale (cache) accessible globalement, serait en mesure d'implanter le protocole de synchronisation proposé par Mellor-Crumey et Scott (voir section 3.2.3.5). À ce titre, un projet pourrait évaluer les performances d'une telle architecture et son intégration dans la bibliothèque SISSMA. La Figure 5.4 montre un arrangement possible qui repose sur l'utilisation de deux processeurs MicroBlaze. Une mémoire locale à deux ports permettrait un accès tant par son MicroBlaze que par le reste du système.

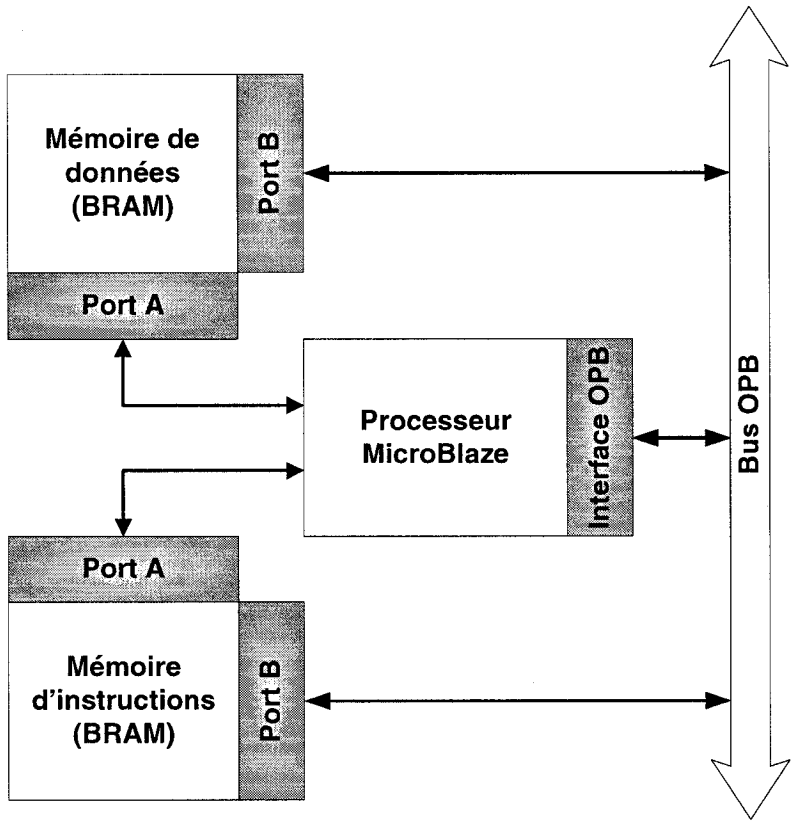


Figure 5.4 Organisation architecturale simplifiée d'un processeur MicroBlaze utilisant des mémoires locales accessibles globalement.

De plus, la bibliothèque SISSMA n'est pas exempte d'interblocage. Il est possible qu'une tâche en attente active sur un sémaphore détenu par une tâche qui exécute sur le même processeur attende indéfiniment si le RTOS local utilise un ordonnancement basé sur les priorités, tel MicroC/OS-II. À ce titre, plusieurs solutions sont possibles. Notamment, empêcher la préemption d'une tâche qui possède un sémaphore de synchronisation est une avenue prometteuse. Ce type de support requiert toutefois une interface générique à l'API SystemC/RTOS de la plateforme SPACE.

Considérations pour le futur

Bien qu'il procure un gain d'accélération, le rôle principal du moteur de synchronisation SISSMA est d'assurer la présence d'un support matériel générique utilisable dans le plus grand nombre possible de variantes architecturales. La tendance actuelle n'est pas l'implantation matérielle des protocoles de synchronisation. En fait, elle consiste plutôt à utiliser des services de synchronisation de base simples afin d'implanter des opérations de synchronisation plus complexes à l'aide d'algorithmes logiciels qui utilisent ces services de base [103]. Les instructions atomiques spécialisées *LoadLink/Store* ou *StoreConditional* représentent de tels services.

RÉFÉRENCES

[1] G. Graunke and S. Thakkar. Synchronization algorithms for shared memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[2] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 199–206, 1995.

[3] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an iterface-centric approach. *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 206–217, 2004.

[4] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[5] G. Hellestrand. The engineering of supersystems. *Computer*, 38(1):103–105, 2005.

[6] A. Corsaro and D. C. Schmidt. The design and performance of real-time Java middleware. *Parallel and Distributed Systems, IEEE Transactions on*, 14(11):1155–1167, 2003.

[7] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-On-Chips.* Morgan Kaufmann Publishers, 2004.

[8] Intel. Moore's Law [En ligne], 2006. www.intel.com/technology/silicon/mooreslaw/ (page consultée le 26 août 2006).

[9] F. Bacchini, G. Martin, P. Paulin, R. A. Bergamaschi, and R. Pawate. System level design: six success stories in search of an industry. *Design Automation Conference, 2004. Proceedings. 41st*, pages 349–350, 2004.

[10] L. Böszörményi, J. Gutknecht, and G. Pomberger. *The School of Niklaus Wirth: The Art of Simplicity.* Morgan Kaufmann Publishers, 2000.

[11] M. Kanellos. Soaring costs of chipmaking recast industry, 2006 2003. news.com.com (page consultée le 31 août 2006).

[12] W. O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Yoo Sungjoo, A. A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor SoC platforms: a component-based design approach. *Design & Test of Computers, IEEE*, 19(6):52–63, 2002.

[13] Yoo Sungjoo, M. W. Youssef, A. Bouchhima, A. A. Jerraya, and M. Diaz-Nava. Multiprocessor SoC design methodology using a concept of two layer hardware-dependent software. volume 2, pages 1382–1383, 2004.

[14] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. M. Aboulhamid. A SystemC Refinement Methodology for Embedded Software. *Design and Test of Computers, IEEE*, 23(2):148–158, 2006.

[15] B. A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[16] W. Wolf. The future of multiprocessor systems-on-chips. *Design Automation Conference, 2004. Proceedings. 41st*, pages 681–685, 2004.

[17] G. Bell and C. van Ingen. DSM perspective: another point of view. *Proceedings of the IEEE*, 87(3):412–417, 1999.

[18] F. Deslauriers. *Modélisation d'un Réseau Intégré sur Puce Basé sur une Architecture en Anneau.* PhD thesis, École Polytechnique de Montréal, 2005.

[19] F. St-Pierre. *Modélisation à bas niveau et analyse d'une réseau intégré sur puce.* PhD thesis, École Polytechnique de Montréal, 2006. (mémoire en préparation).

[20] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. pages 70–73 suppl., 2003.

[21] I. Newman and A. Schuster. Hot-potato algorithms for permutation routing. *Parallel and Distributed Systems, IEEE Transactions on*, 6(11):1168–1176, 1995.

[22] eCos. site web [En ligne], 2006. ecos.sourceware.org/ (page consultée le 26 août 2006).

[23] GreenHills. INTEGRITY [En ligne], 2006. www.ghs.com (page consultée le 26 août 2006).

[24] QNX. QNX Neutrino Realtime Operating System [En ligne], 2006. www.qnx.com (page consultée le 26 août 2006).

[25] Micrium. MicroC/OS-II [En ligne], 2006. www.micrium.com (page consultée le 26 août 2006).

[26] Accelerated Technologies. Nucleus [En ligne], 2006. www.acceleratedtechnology.com/ (page consultée le 26 août 2006).

[27] RTEMS. RTEMS Real-time Operating Systems [En ligne], 2006. www.rtems.com (page consultée le 26 août 2006).

[28] FSM Labs. RTLinux, 2006. www.rtlinux.com (page consultée le 26 août 2006).

[29] Express Logic. ThreadX [En ligne], 2006. www.threadx.com (page consultée le 26 août 2006).

[30] WindRiver. VxWorks [En ligne], 2006. www.windriver.com (page consultée le 26 août 2006).

[31] J. Henkel. Closing the SoC design gap. *Computer*, 36(9):119–121, 2003.

[32] A. A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[33] POLIS. A Framework for Hardware-Software Co-Design of Embedded Systems [En ligne], 2006. embedded.eecs.berkeley.edu/Research/hsc/abstract.html (page consultée le 26 août 2006).

[34] C. Serra. Hardware/Software Codesign Overview [En ligne], 2006. www.cs.uvic.ca/ mserra/HScodesign.html (page consultée le 26 août 2006).

[35] J. Horgan. ESL Chapter 1, 2004. www.edacafe.com (page consultée le 1er septembre 2006).

[36] B. Bowyer. The 'what' and 'why'of transaction level modeling [En ligne], 2006. www.eetimes.com (page consultée le 1er septembre 2006).

[37] H. J. Schlebusch, G. Smith, D. Sciuto, D. Gajski, C. Mielenz, C. K. Lennard, F. Ghenassia, S. Swan, and J. Kunkel. Transaction based design: another buzzword or the solution to a design problem? *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 876–877, 2003.

[38] M. Barry. ESL - Bridging the Gap Between Design and Architectural Development [En ligne]. *Cdn Users*, 23 mars 2006 2006.

[39] SystemC. SystemC User's Guide [En ligne], 2006. www.systemc.org (page consultée le 26 août 2006).

[40] IEEE. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference. Technical report, IEEE, 2005.

[41] J. Horgan. Who Is Using ESL and Why?, 2005. www.edacafe.com (page consultée le 26 août 2006).

[42] IEEE. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2005*, 2005.

[43] W. Rosenstiel, S. Swan, F. Ghenassia, P. Flake, and J. Srouji. SystemC and SystemVerilog: Where do they fit? Where are they going? *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 122–127, 2004.

[44] R. Goering. System level design language arrives [En ligne], 2006. www.soccentral.com (page consultée le 26 août 2006).

[45] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[46] Deep Object Knowledge. B-Sharp, 2006. www.bsharplanguage.org (page consultée le 26 août 2006).

[47] Aonix. PERC Pico [En ligne], 2006. www.aonix.com/perc.html (page consultée le 26 août 2006).

[48] F. Leens. Electronic system-level development: Finding the right mix of solutions for the right mix of engineers, 2005. www.byteparadigm.com/Documentation.html (page consultée le 1er septembre 2006).

[49] S. Leibson. Will the real ESL please stand up?, 2006. www.end.com (page consultée le 1er septembre 2006).

[50] C. Sullivan, A. Wilson, and S. Chappell. Using C based logic synthesis to bridge the productivity gap. pages 349–354, 2004.

[51] R. Cravotta. Automate Your Acceleration. *EDN*, pages 55–64, 2004. www.edn.com (page consultée le 26 août 2006).

[52] M. Fujita and H. Nakamura. The standard SpecC language. *Proceedings of the 14th International Symposium on System Synthesis*, pages 81–86, 2001.

[53] D. Gajski. *SpecC Specification Languaga and Methodology.* Kluwer Academic, Norwell, Mass., 2000.

[54] Mentor. Catapult C Synthesis [En ligne], 2006. www.mentor.com (page consultée le 26 août 2006).

[55] Celoxica. site web:, 2006. www.celoxica.com/ (page consultée le 26 août 2006).

[56] SystemCrafter. SystemCrafterSC [En ligne], 2006.

[57] Forte. Cynthesizer [En ligne], 2006. www.forteds.com (page consultée le 26 août 2006).

[58] IBM. The CoreConnect Bus Architecture, 1999. www.chips.ibm.com (page consultée le 26 août 2006).

[59] D. Flynn. AMBA: enabling reusable on-chip designs. *Micro, IEEE*, 17(4):20–27, 1997.

[60] D. Wingard. Micronetwork-based integration for SOCs. *Proceedings of the 38th conference on Design automation*, page 677, Las Vegas, Nevada, United States, 2001.

[61] OpenCores. SoC Interconnection: Wishbone [En ligne], 2006. www.opencores.org (page consultée le 26 août 2006).

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

[63] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1256–1261, 2004.

[64] S. Provost. *Accélération d'une plateforme d'encodage MPEG-4 à l'aide de processeurs configurables.* PhD thesis, École Polytechnique de Montréal, 2006.

[65] P. G. Paulin. DATE panel chips of the future: soft, crunchy or hard? *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 844–849, 2004.

[66] ST. Nomadik Multimedia Processor [En ligne], 2006.

[67] J. Song, T. Shepherd, Chau Minh, A. Huq, L. Syed, S. Roy, A. Thippana, K. Shi, and U. Ko. A low power open multimedia application platform for 3G wireless. *Proceedings IEEE International Systems-on-Chip (SOC) Conference*, pages 377–380, 2003.

[68] Philips. Highly integrated, programmable system-on-chip - Nexperia [En ligne], 2006. www.semiconductors.philips.com (page consultée le 26 août 2006).

[69] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, 1988.

[70] A. Dinning. A survey of synchronization methods for parallel computers. *Computer*, 22(7):66–77, 1989.

[71] S. Dharmasanam. Using an RTOS to Implement Symmetric Multiprocessing [En ligne], 2006. www.qnx.com/download/whitepapers (page consultée le 1er septembre 2006).

[72] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, 1985.

[73] M. Ariyamparambath, D. Bussaglia, B. Reinkemeier, T. Kogel, and T. Kempf. A highly efficient modeling style for heterogeneous bus architectures. *Proceedings. International Symposium on System-on-Chip*, pages 83–87, 2003.

[74] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 752–757, 2004.

[75] S. Pasricha. Transaction Level Modeling of SoC with SystemC 2.0. *Synopsys User Group Conference (SNUG 2002)*, 2002.

[76] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization. *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 134–143, 1997.

[77] M. M. Michael and M. L. Scott. Relative performance of preemption-safe locking and non-blocking. *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 267–273, 1997.

[78] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1), 1987.

[79] Z. Segall and L Rudolph. Dynamic decentralized cache schemes for an MIMD parallel processor. *Proc. 11th Annu. Intern. Symp. Comput. Architect.*, pages 340–347, 1984.

[80] J. M. Mellor-Crummey and M. L. Scott. Synchronization Without Contention. *Proceedings of the Forth international Conference on Architectural Support For Programming Languages and Operating Systems*, pages 269–278, 1991.

[81] G. Pfister and V. Norton. Hot-spot contention and combining in multistage interconnection networks. *ACM Trans. Comput. Syst.*, 3(4), 1985.

[82] E.D. Brooks III. The shared memory hypercube. *Parallel Computing*, volume 6, pages 235–245, 1988.

[83] B. Beck, B. Kasten, and S. Thakkar. VLSI Assist For A Multiprocessor. *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 10–20, 1987.

[84] S. S. Thakkar, P. Gifford, and G. Fielland. Balance: A Shared Memory Multiprocessor. *Proceedings, 2nd Int. Conf On Supercomputing*, Santa Clara, 1987.

[85] T. Lovett. A CAE Case History. *VLSI Design*, 1984.

[86] B. E. Saglam and III Mooney, V. J. System-on-a-chip processor synchronization support in hardware. *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 633–639, 2001.

[87] B. E. S. Akgul, L. Jaehwan, and V. J. Mooney III. A System-on-a-Chip Lock Cache with Task Preemtion Support. *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*, pages 149–157, 2001.

[88] B. E. S. Akgul, V. J. Mooney III, H. Thane, and P. Kuacharoen. Hardware Support for Priority Inheritance. *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 246–255, 2003.

[89] Q. Li and Caroline Yao. *Real-Time Concepts for Embedded Systems.* CMP Books, 2003.

[90] J. Adomat, j. Furunäs, L. Lindh, and J. Stärner. Real-Time Kernel in Hardware RTU: A step toward deterministic and high performance real-time systems. *8th Euromicro Workshop on Real-Time Systems*, L´Aquila, Italy, 1996.

[91] P. J. Courtnois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[92] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, 1991.

[93] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A Fair Fast Scalable Reader-Writer Lock. *Proceedings of the 1993 International Conference on Parallel Processing*, pages II–201–II–204, 1993.

[94] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 44–52, 2001.

[95] B. H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 25–35, San Jose, California, United States, 1994. ACM Press.

[96] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. *Proceedings of the 16th Annual international Symposium on Computer Architecture*, pages 396–406, 1989.

[97] A. R. Karlin, Li Kai, M. S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. *Proceedings of the13th ACM Symposium on Operating Systems Principle (SIGOPS)*, 1991.

[98] B. H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. Comput. Syst.*, 11(3):253–294, 1993.

[99] M. M. Michael and M. L. Scott. Implementation of Atomic Primitives on Distributed Shared-Memory Multiprocessors. *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 222–231, 1995.

[100] B. Zeidman. All about FPGAs, 23 mars 2006 2006.

[101] D. Maliniak. Basics of FPGA Design, 18 août 2006 2003.

[102] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2. Wiley & Sons, 2000.

[103] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: methodology and performance. *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 23–34, 1999.

[104] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3 edition, 2003.

[105] I. H. Khan. Manging Complexity with SystemC, 13 mars 2006 2006.

[106] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[107] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *Proceedings of the Third international Conference on Architectural Support For Programming Languages and Operating Systems*, pages 64–75, 1989.

[108] Wang Cai-Dong, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real time systems. pages 70–76, 1996.

[109] T. Johnson and K. Harathi. A prioritized multiprocessor spin lock. *Parallel and Distributed Systems, IEEE Transactions on*, 8(9):926–933, 1997.

[110] P. Stenström, E. Hagersten, D. J. Lilja, M. Martonosi, and M. Venugopal. Trends in shared memory multiprocessing. *Computer*, 30(12):44–50, 1997.

[111] B. Zeidman. Using software synthesis for multiprocessor OS and software development, 2006. www.embedded.com (page consultée le 1er septembre 2006).

[112] Alan Wallcraft. Co-Array Fortran, 2006. www.co-array.org/ (page consultée le 31 août 2006).

[113] Alan Wallcraft. Co-Array Fortran vs MPI, 2006. www.co-array.org/cafvsmpi.htm (page consultée le 31 août 2006).

[114] MPI. The Message Passing Interface (MPI) standard, 2006. www-unix.mcs.anl.gov/mpi/ (page consultée le 31 août 2006).

[115] W. Gropp, E. Lusk, N. Doss, and A Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[116] RICE University. Parallel Computing, 2006. www.owlnet.rice.edu/ comp422/resources/ (page consultée le 31 août 2006).

[117] Openmp. OpenMP Specification [En ligne], 2006. www.openmp.org (page consultée le 31 août 2006).

[118] The High Performance Computing Laboratory. Unified Parallel C, 2006. upc.gwu.edu/ (page consultée le 31 août 2006).

[119] T. El-Ghazawi. Programming in UPC, 2006. www.gwu.edu/ upc/tutorials.html (page consultée le 31 août 2006).

ANNEXE I

HIÉRARCHIE DE MÉMOIRE

Cette section présente un résumé du concept de hiérarchie de mémoires. Il aborde principalement le sujet des mémoires caches tout en résumant les principaux protocoles qui assurent leur cohérence. Le lecteur intéressé est invité à consulter [104] pour plus d'informations.

Une hiérarchie de mémoires tient un rôle important devant l'augmentation de performance des processeurs. Alors que ces derniers opèrent de plus en plus rapidement, les temps d'accès des mémoires n'affichent pas la même évolution. Afin de palier à cette différence en performance, des mémoires de plus petite capacité, mais plus rapides sont utilisées. Les mémoires caches jouent ce rôle. Elles servent d'intermédiaire entre la mémoire centrale et les registres du processeurs. La Figure I.1 illustre cet arrangement.

Dans une hiérarchie de mémoires, le contenu d'une mémoire se retrouve aussi dans la mémoire de niveau supérieur de la hiérarchie. Ainsi, le contenu de la mémoire cache se trouve aussi dans la mémoire centrale qui à son tour est une copie d'une partie du contenu du disque dur.

Dans le contexte d'une plateforme multiprocesseur, l'utilisation de mémoires caches implique généralement l'utilisation d'un protocole de cohérence. Ce protocole est responsable de l'intégrité des données en mémoire. Bref, il veille à ce qu'aucune donnée ne soit écrasée ou supprimée d'une mémoire cache avant d'avoir été copiée en mémoire centrale. Aussi, le protocole de cohérence assure la mise à jour des données en mémoire cache afin qu'une donnée utilisée par un processeur n'est pas été modifiée

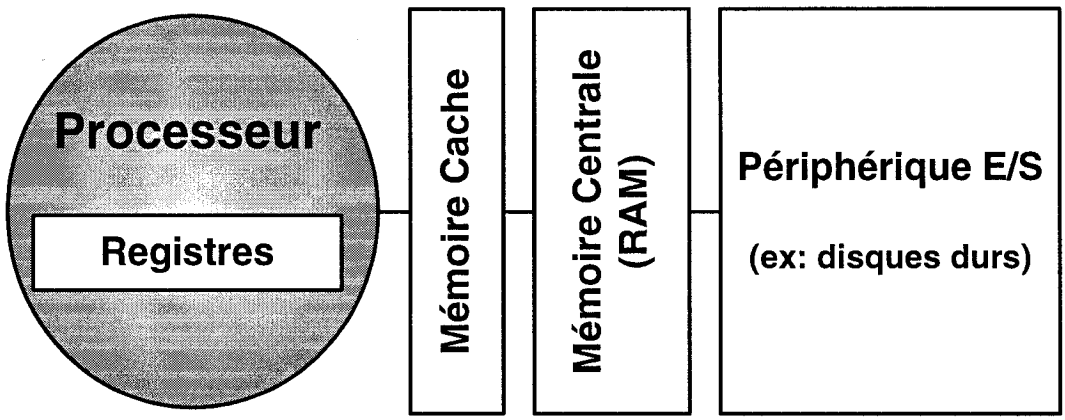


Figure I.1 Hiérarchie de mémoire dans un système embarqué typique

entre temps.

Il existe plusieurs protocoles de cohérence des mémoires caches. Voici un résumé du mode d'opération des plus populaires.

Cohérence basée sur l'utilisation d'un répertoire (« Directory-based Coherence »): Une approche consiste à utiliser un répertoire central. Ce répertoire sert de référence sur l'état d'une donnée lue lorsqu'un processeur désire lire une nouvelle donnée de la mémoire centrale. Par la suite, lorsque cette dernière est modifiée, les mémoires caches qui disposent d'une copie de cette donnée en sont informées par le répertoire.

Cohérence basée sur la surveillance de trafic (« Snooping Coherence »): Une alternative au répertoire requiert que chaque contrôleur d'une mémoire cache surveille les transactions qui ont cours sur le bus système. Le contrôleur peut alors déterminer si une transaction affecte une donnée contenu dans sa mémoire cache afin de s'assurer de contenir une valeur à jour.

Afin de conserver la mémoire centrale dans un état consistant, une mémoire cache se doit de propager les écritures (« **write through** »). Ainsi, une écriture dans la

mémoire cache implique aussi une écriture en mémoire centrale.

Une alternative à cette approche consiste à ne pas mettre à jour la mémoire centrale immédiatement. Chaque emplacement modifié dans la mémoire cache est marqué. Par la suite, lorsque cet emplacement sera évacué de la mémoire cache, par exemple suite à l'écriture d'une nouvelle donnée en mémoire cache, un emplacement marqué comme étant modifié est alors copié en mémoire centrale avant son écrasement. Cette seconde approche, nommée « **write-back** » réduit le nombre d'écritures en mémoire centrale si une même donnée est modifiée plusieurs fois en mémoire cache.

De surcroît, il existe une approche par invalidation des écritures qui consiste à invalider une entrée de la mémoire cache lorsque celle-ci contient une donnée modifiée par un autre processeur. L'accès à cette dernière demandera alors de la mettre à jour avant d'y accéder. Ce protocole se nomme aussi « **write invalidate** ».

D'une manière complémentaire à l'approche « write invalidate », un procotole de cache basé sur une approche « write update » met systématiquement à jour une donnée contenue dans sa mémoire cache lorsqu'elle est modifiée par un autre processeur.

ANNEXE II

SYSTEMC: COMPOSITION D'UN MODÈLE

Cette section présente un résumé de la composition d'un modèle typique implanté à l'aide de la bibliothèque SystemC. Dans l'ensemble, cette section définit les composants principaux, et leurs interactions, d'un tel modèle . Pour de plus amples informations, veuillez consulter [75][105][36][106] ou *www.systemc.org*.

La Figure II.1 illustre l'organisation typique de composants à l'intérieur d'un modèle SystemC. Ces composants sont analogues aux classes du langage C++: ils se doivent d'être cohérents afin d'accomplir une fonction particulière.

Les fonctionnalités du modèle sont définies dans des tâches. Ces tâches prennent aussi le nom de processus (ou « process »). Le groupement cohérent d'un ou plusieurs processus résulte en un *module* SystemC. Un module représente une entité distincte ayant son propre rôle. Il est aussi possible de créer des modules hiérarchiques. Ainsi, un module contient plusieurs autres modules.

Typiquement, un module d'un modèle transactionnel répond à des stimulis. Ceux-ci sont transmis via des événements sur des interfaces aux différents modules. Un module se connecte à une interface particulière à l'aide d'un port compatible avec l'interface utilisée. Ce port permet l'échange d'informations et la transmission d'événements entre modules.

La bibliothèque SystemC fournit différents types de port prédéfinis. Par exemple, il est possible d'utiliser un signal, l'équivalent d'un simple fil électrique. Un signal peut aussi modéliser un bus fait de plusieurs fils (ex.: signal de 8 bits). Enfin,

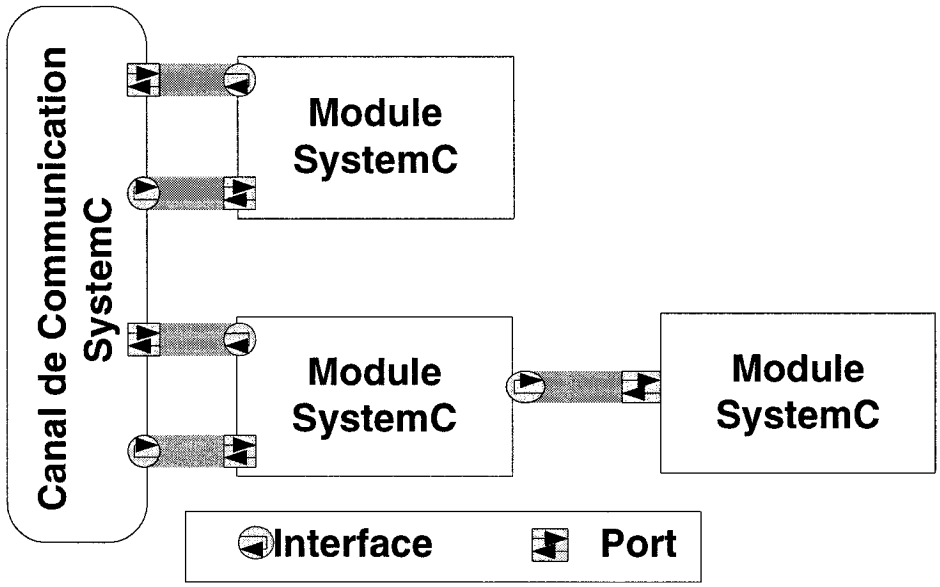


Figure II.1 Structure architecturale des composants SystemC

des structures de communication plus évoluées sont possibles. De cette manière, la création de canaux de communication, des modules dont le rôle est d'assurer l'échange d'information, est aussi envisageable. À ce titre, un concepteur peut créer un canal qui implante les fonctionnalités d'un protocole de communication spécialisé (ex.: AMBA [59] ou CoreConnect [58]) et offrir l'interface voulue.

La bibliothèque SystemC inclut son propre moteur de simulation. Ce moteur opère d'une manière analogue à celui d'un langage des descriptions matérielles (HDL) comme VHDL ou Verilog. De par sa sémantique, ce moteur permet de simuler un environnement dans lequel exécutent des processus concurrents.

ANNEXE III

COMPLÉMENT À LA REVUE DE LITTÉRATURE

Cette section sert de complément à la revue de littérature incluse dans le corps de ce mémoire.

III.1 Objects de synchronisation: complément

III.1.1 Objet de synchronisation de type tournoi (« Tournament Lock »)

Afin de réduire la contention sur le réseau d'interconnexion, ce type d'objet de synchronisation utilise une structure en arbre ayant une base B et une hauteur H. L'arbre forme un tournoi où le gagnant de chaque concours entre les nœuds terminaux devient un participant pour le niveau suivant. Le gagnant du nœud racine peut procéder dans sa section critique. La contention est réduite puisque le nombre de processus en compétition à la fois pour acquérir l'objet de synchronisation du niveau auquel ils participent est réduit de B à chaque niveau. Toutefois, dans le pire cas, il faut s'attendre à ce que la latence associée à l'acquisition de l'objet de synchronisation atteigne H fois celle de l'acquisition optimiste d'un objet de synchronisation basé sur le spin avec lecture [1].

III.1.2 Objet de synchronisation basé sur une queue (« Queueing Lock »)

Anderson[4] et Graunke et Thakkar[1] ont indépendamment proposé un algorithme de synchronisation qui repose sur le concept de queue d'attente. L'idée de base est la

suivante : une structure de donnée centrale, la queue, contient tous les processus en attente sur cet objet de synchronisation. Un processus en attente entre toutefois en spin sur un fanion local. Lorsque l'objet est libéré, c'est ce fanion propre au prochain processus dans la queue d'attente qui est modifié. Idéalement, ce fanion est alloué dans un bloc de mémoire cache. Sur modification de ce fanion, le bloc-mémoire est automatiquement mis à jour par le protocole de cohérence de la cache.

L'algorithme proposé par Anderson requiert seulement une instruction atomique, l'instruction *ReadAndIncrement*, tandis que la solution de Graunke et Thakkar demande l'exécution de plusieurs instructions consécutives protégées par un sémaphore matériel.

Ce type d'objet de synchronisation a un coût supérieur lorsqu'il n'y a pas de contention ou que l'objet est libre. De plus, sa contention est réduite par l'utilisation du protocole de cohérence de cache « Distributed-Write ». Cette approche perd tout son attrait si les mémoires caches du système cible ne sont pas cohérentes. Si le protocole de cohérence repose sur l'invalidation, il est alors requis que chaque fanion d'attente se trouve dans un bloc de cache différent afin qu'une seule mise à jour ait lieu lors de la libération de l'objet de synchronisation.

III.1.3 Synchronisation QOSB

QOSM[107] ou « Queue On Sync Bit » est une approche qui repose sur l'utilisation de mécanismes matériels afin d'assurer l'exclusion mutuelle à une ressource. Elle assure un ordonnancement FIFO. L'utilisation de mécanismes matériels permet de réduire le nombre d'accès à travers le réseau d'interconnexion en mémoire globale tout en favorisant un plus grand parallélisme des requêtes de synchronisation.

Des bits de synchronisation, appelés *syncbits*, sont utilisés afin d'identifier si un objet de synchronisation est libre ou non. Les auteurs, dans leur implémentation de cette approche, proposent d'associer un *syncbit* avec chaque ligne de la mémoire partagée. Ici, une ligne correspond à une unité alignée de mémoire. Bien que cette solution permette l'implantation du protocole de synchronisation à travers une extension du protocole de cohérence de la mémoire cache, il est nécessaire de s'assurer que deux structures de données qui utilisent la synchronisation offerte par ce mécanisme des *syncbits* n'apparaissent pas dans une même ligne-mémoire. Dans le cas où l'objet de synchronisation est occupé, une transaction a lieu sur le réseau d'interconnexion afin d'ajouter un processeur dans la queue d'attente en mémoire centrale partagée. Par la suite, le processeur entre en spin sur le *syncbit* local approprié.

III.1.4 Objet de synchronisation priorisé et héritage de priorité

Afin d'assurer un comportement temps-réel correct, d'autres travaux de recherche ont permis d'ajouter l'héritage de priorité à la liste des fonctionnalités des objets de synchronisation. C'est le cas des travaux de Wang, Takada et Sakamura [108] qui reposent sur un objet de synchronisation avec une queue d'attente. Cette queue d'attente est parcourue afin d'identifier le prochain processus propriétaire de l'objet de synchronisation contrairement à utiliser celui en tête de file, comme dans l'approche FIFO.

Johnson[109] utilise le même concept de queue. Toutefois, son approche sert à prioriser des tâches quant à leur ordre d'accès à leur section critique, protégé par l'objet de synchronisation en question.

III.2 Modèle de programmation pour système multiprocesseur

III.2.1 Modèle de programmation : définition

Un modèle de programmation est une interface entre le programmeur et l'architecture de la machine cible. Le choix du modèle de programmation détermine comment le programmeur voit la machine sous-jacente et les services qu'elle offre. Le modèle de programmation est toutefois indépendant des mécanismes utilisés pour implanter ces services [110]. Une solution idéale consiste à appliquer le principe de synthèse logicielle afin que l'outil de synthèse insère automatiquement, comme le ferait un compilateur, l'ensemble des services requis, tant au niveau de la création de tâches, de leur synchronisation et de la gestion de mémoire, et ce, en fonction du système et de l'application. Par exemple, la synthèse logicielle générerait la synchronisation requise pour protéger une ressource si plusieurs tâches y accèdent. Dans le cas contraire, le code de synchronisation ne serait pas généré [111].

III.2.2 Besoins

Avec le gain de popularité et d'utilisation des plateformes multiprocesseurs, plus particulièrement des MPSoCs, il est nécessaire que le développement de systèmes parallèles soit facilité. Un aspect déficient se situe au niveau de la facilité de transférer une même application vers différents systèmes multiprocesseurs. Aussi, des techniques nouvelles sont nécessaires afin d'automatiser le processus de parallélisation et d'ajustement des performances du système [110]. Bref, les modèles de programmation disponibles requièrent une abstraction supérieure de la plateforme cible utilisée lors du développement initial de l'application.

La section suivante présente certains des modèles de programmation les plus connus ou répandus présentement dans le domaine des systèmes multiprocesseurs.

III.2.3 Co-Array Fortran (CAF)

CAF [112] est une extension au langage de programmation Fortran 90 conçue pour les calculs de hautes performances sur des machines parallèles à grande échelle. Il s'applique à des machines parallèles SPMD (« Single Program Multiple Data). CAF offre une interface de programmation uniforme tant pour les architectures basées sur une mémoire partagée ou des mémoires distribuées. CAF abstrait l'espace d'adressage du système à l'aide de deux niveaux. Ces deux niveaux d'abstraction servent à qualifier les données comme étant locales ou accessibles à distance (« remote »). Aussi, les tableaux nommés *co-array* servent de variables partagées qui peuvent être lues ou écrites par tous les processeurs du système, sans discrimination. Une syntaxe différente est utilisée afin de différencier les accès à des données locales contrairement à des données situées dans une autre mémoire non locale. Cela permet la gestion explicite de la localité des données, par les programmeurs, afin d'obtenir les performances désirées. Cette extension permet aussi l'utilisation de services de synchronisation afin d'assurer la cohérence d'exécution des différentes tâches, sans toutefois spécifier quelle méthode de synchronisation est utilisée [113].

III.2.4 Message Passing Interface (MPI)

MPI [114] est une spécification d'une bibliothèque de communication standardisée pour la transmission de messages. Son objectif est de permettre l'implantation de technologies transférables (« portable »), extensibles et de hautes performances qui repose sur la transmission de messages [113]. MPI se limite à la définition de l'interface de programmation, puisqu'il ne définit que le standard. Cette spécification doit être implantée. Une implantation populaire de MPI est MPICH [115].

Une courte comparaison quantitative de MPI et CAF est présentée dans [116].

III.2.5 OpenMP

OpenMP [117] est une interface de programmation en C/C++ ou Fortran qui supporte un modèle de programmation basée sur le partage de mémoire. L'approche utilisée par OpenMP afin de générer une application exécutable sur une plateforme multiprocesseur consiste à utiliser un compilateur dédié. Ce compilateur utilise les indications incluses par les développeurs afin de paralléliser des portions de l'application d'une manière automatisée. Ces indications prennent la forme de *pragma*. Par exemple, un *pragma* existe afin de paralléliser l'exécution d'une boucle logicielle sur plusieurs processeurs. Le compilateur assure automatiquement la création des multiples tâches et leur synchronisation une fois les opérations terminées. OpenMP est conçu d'abord pour des machines SMP (« Symmetric Multiprocessing ») qui permettent l'exécution de processus dynamiques sur différents processeurs homogènes.

III.2.6 Task Transaction Level Interface (TTL)

L'interface TTL [3], disponible dans les langages C et C++, offre des services de communication à des applications multitâches conçues pour le traitement parallèle ou pour des plateformes multiprocesseurs. TTL permet des communications de différents types. Ainsi, il est possible d'exécuter des lectures ou des écritures bloquantes ou non-bloquantes à une adresse absolue ou relative. Ces opérations de communication encapsulent les phases de synchronisation et le transfert des données. Il est aussi possible de spécifier explicitement ces deux séquences d'opérations dans le cas d'opérations dont la granularité est plus grosse, permettant donc de réduire le coût associé à la synchronisation en échange d'une réduction du niveau d'abstraction de l'application. Différents types de tâches sont aussi disponibles : le processus, la coroutine et l'acteur. Ces tâches diffèrent en fonction de leurs interactions avec

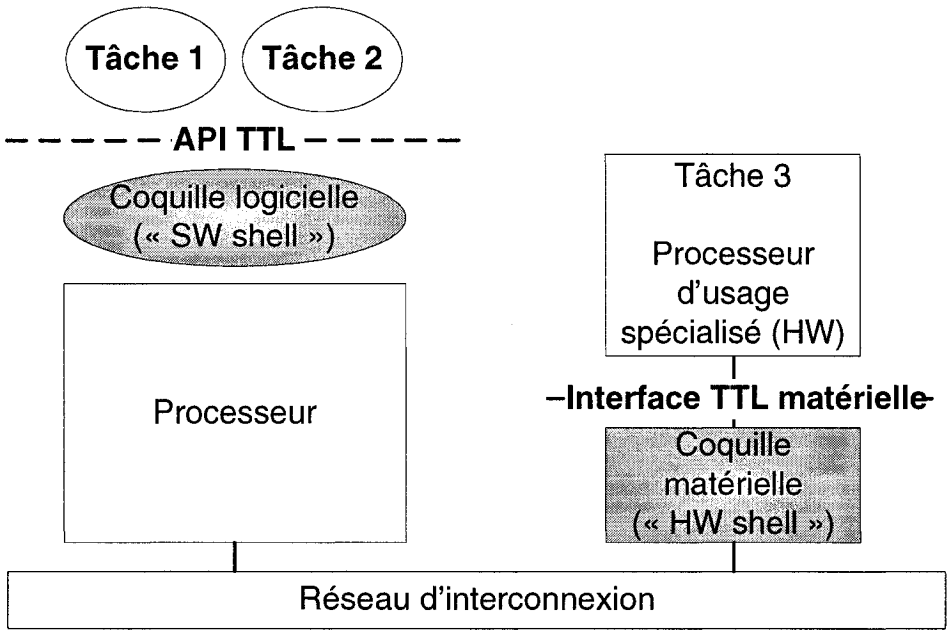


Figure III.1 L'interface TTL et ses coquilles logicielles et matérielles [3]

l'ordonnanceur du système: le processus est indépendant, la coroutine permet les changements de contexte à certains points dans son exécution et l'acteur est équivalent à une tâche sporadique. L'interface TTL doit être adaptée à chaque architecture sur laquelle elle s'exécutera en fonction des services matériels offerts. Lorsque les services TTL sont aussi implantés dans un adaptateur matériel, la communication entre des tâches logicielles et des tâches matérielles est possible. La Figure III.1 illustre ce mode d'opération. La coquille logicielle (« SW shell ») et la coquille matérielle (« HW shell ») implantent les services TTL disponibles pour leur(s) tâche(s) respective(s).

III.2.7 Unified Parallel C

UPC [118] est une extension au langage de programmation C conçue pour les calculs de haute performance sur des machines parallèles à grande échelle. UPC offre aussi une interface de programmation uniforme tant pour les architectures basées sur

une mémoire partagée ou des mémoires distribuées. L'interface de programmation présente un seul espace d'adressage partagé où tous les processeurs peuvent accéder en lecture ou en écriture à l'ensemble des variables, et ce, même si elles sont en réalité associées à un processeur en particulier. Le modèle de calcul utilisé par UPC est SPMD, tout comme CAF. Ce modèle de calcul fixe le parallélisme de l'application à son démarrage par l'utilisation d'une seule tâche par processeur. Les services de synchronisation offerts par UPC consistent en des objets de synchronisation (« lock »), des barrières de synchronisation et un contrôle de la cohérence de la mémoire. La synchronisation entre les tâches doit être assurée explicitement par les développeurs [113][119].

ANNEXE IV

BIBLIOTHÈQUE SISSMA: VERSION PERSONNALISÉE

La bibliothèque SISSMA se présente sous deux implantations. La première tire profit de la plateforme SPACE et de ses modes de communication bloquants ou non, tel que décrit plus tôt. Une seconde version, celle-ci indépendante de SPACE, est aussi disponible. Cette version est qualifiée de *personnalisée* (« custom »), puisque son implantation de la communication et de synchronisation est propre à une solution aux infrastructures de communication indépendantes[1].

La version personnalisée de la bibliothèque SISSMA se voit imposer certaines contraintes d'implantation face au manque de services antérieurement supportés par SPACE. Ainsi, les requêtes entre l'API SISSMA et le moteur de synchronisation ne doivent contenir que 32 bits, afin d'être transmises en une seule transaction sur le bus système : l'atomicité de la transaction ne pouvant être assurée pour des données plus grandes que 32 bits.

De plus, cette version utilise un mécanisme d'éveil différent lors d'une attente bloquante, puisque les communications bloquantes ne sont pas disponibles ici sans SPACE. Pour ce faire, un sémaphore local est utilisé afin de bloquer la tâche dans son exécution. L'éveil d'une tâche se fait par l'envoi d'un signal d'interruption émis par le moteur de synchronisation SISSMA. La routine d'interruption libère ainsi le sémaphore en question permettant à la tâche en attente de poursuivre son exécution.

[1] Cette version personnalisée est requise afin de démontrer l'utilisation, sur une plateforme multiprocesseur, de la bibliothèque SISSMA, puisque l'infrastructure de communication utilisée par SPACE était en cours de développement au moment des présents travaux. Une version fonctionnelle sur FPGA n'étant pas disponible à ce moment.

Dans l'ensemble, les comportements de la bibliothèque SISSMA restent les mêmes, mais l'implantation du moteur matériel de synchronisation en est affectée.

Le générateur d'interruptions dans l'implantation VHDL du module SISSMA sera obsolète une fois les adaptateurs SPACE disponibles sur une plateforme matérielle. Ils seront remplacés par un contrôleur simple responsable de l'envoi d'un message destiné à la tâche à éveiller. Cette mise à jour réduira d'autant plus la surface matérielle requise, puisque le générateur d'interruptions utilise présentement une série de comparateurs parallèles.

ANNEXE V

BIBLIOTHÈQUE SISSMA : IMPLANTATION PERSONNALISÉE ET ENCODAGE DES REQUÊTES

Une seconde implantation des composants SISSMA utilise un mode de communication personnalisé. Ce mode définit des fonctions de communication qui ne dépendent nullement de la plateforme SPACE.

La particularité de ce mode de communication tient au mécanisme utilisé pour éveiller une tâche en attente d'un objet de synchronisation. Contrairement à la plateforme SPACE, le mode de communication personnalisé ne dispose pas d'opérations de lecture et d'écriture synchrones. Pour contrer cette limitation, des signaux d'interruption sont utilisés. Chaque module utilisant une donnée partagée de type SISSMA doit avoir un signal d'entrée pour signaler une interruption (IRQ). Ainsi, lorsqu'une tâche est mise en attente due à un accès refusé à un objet de synchronisation, elle sera éveillée par la routine d'interruption qui réagit facilement à ce signal d'interruption émis par le module SISSMA. Il est de la responsabilité du module émettant une requête pour un sémaphore de transmettre l'identificateur de sa ligne d'interruption. Toutefois, notons que ce paramètre est configurable une seule fois (#define) et ne requiert pas l'intervention du développeur mais bien seulement de l'architecture système. Cette valeur pourra être générée automatiquement par SpaceStudio si désiré.

V.1 Modifications pour l'implantation sur FPGA

La plateforme SPACE n'est pas suffisamment développée afin de permettre un raffinement complet d'un système modélisé à haut niveau vers une solution exécutable

directement sur une puce FPGA. Bref, certains éléments de la plateforme n'ont pas encore d'équivalent matériel. C'est pourquoi deux implantations de l'interface SISSMA sont développées. À la base, le système de communication de la plateforme SPACE, où sont offerts des services d'envoi et de réception de messages, synchrones ou non, n'est pas implanté à bas niveau (RTL). C'est pour pallier cette déficience qu'une implantation personnalisée (« custom ») de l'interface de programmation (API) de la bibliothèque SISSMA fut créée.

Voici les modifications apportées à cette implantation :

1. L'envoi et la réception de messages entre l'interface de programmation et le module SISSMA sont remplacés par des opérations de lecture et d'écriture. Des plages d'adresses sont réservées à l'intérieur du module SISSMA. Chaque plage d'adresses est associée à une opération unique dans ce module.

2. Une transmission synchrone d'un message est remplacée par l'utilisation d'un sémaphore local. La tâche en attente d'un message de la part du module SISSMA se bloque sur ce sémaphore qui sera à son tour signalé par un signal d'interruption. Ce signal est émis par le module SISSMA lorsqu'il désire éveiller une tâche en attente. Bref, il remplace l'envoi d'un message qui aurait débloqué une tâche dans la plateforme SPACE.

3. L'API SystemC/RTOS qui permet de convertir les éléments SystemC utilisés en services du RTOS local, n'est pas utilisé, puisque sa compatibilité avec la version 3 de SPACE n'est pas complètement vérifiée en ce moment. Ainsi, les éléments de synchronisation utilisés dans SISSMA (sémaphore) qui relèvent de SystemC sont remplacés par leur équivalent dans le RTOS MicroC. Notez qu'aucun changement au code source n'est nécessaire. Seul le type de donnée utilisé doit être adapté.

La bibliothèque SISSMA offre différents niveaux de raffinement. Ainsi, à un niveau

plus élémentaire se trouve l'objet de synchronisation de base : la classe *SissmaSync*. La classe *SissmaSyncRw* présente les mêmes fonctionnalités que cette dernière en plus de permettre l'optimisation de type rédacteur-lecteur. Toutefois, puisqu'elles partagent les mêmes services, il n'est pas requis de les redéfinir. Le même processus d'héritage est utilisé afin de définir les classes *SissmaData*, *SissmaDataArray* et *SissmaDataArray2d*. Bref, l'ensemble des classes disponibles dans la bibliothèque SISSMA relève des services génériques offerts par la classe SissmaSync.

Dans cette optique, lorsque les communications changent entre l'implantation de SISSMA conçue pour fonctionner dans SPACE et celle dédiée à la carte FPGA, il ne suffit que d'adapter la classe *SissmaSync*, et ce, sans répercussion sur le reste de la structure de la bibliothèque.

En somme, les changements apportés dans la classe *SissmaSync* et dans les fonctions d'envoi et de réception de messages (*SissmaCommUtility*) couvrent l'ensemble des modifications requises afin d'exécuter une application utilisant la bibliothèque SISSMA sur une plateforme de type FPGA.

V.2 Encodage des communications avec le module SISSMA personnalisé

Les communications avec le module SISSMA demandent un encodage des adresses de lecture et d'écriture afin d'envoyer des commandes et des paramètres propres à une requête particulière.

Il est supposé que le système cible dispose d'un bus d'adressage d'au moins 32 bits.

La largeur des champs encodés pour un message est partiellement dictée par la plateforme SPACE puisqu'une compatibilité entre le projet SISSMA et SPACE fait partie de nos objectifs.

Code de contrôle

Un ensemble de 14 commandes est disponible. Dans ce cas, les quatre bits dédiés à ce champ sont utilisés. Toutefois, il est possible d'utiliser les deux combinaisons restantes afin de définir de nouvelles opérations dans le futur.

Identificateur de la tâche

Toutes les tâches doivent être identifiées par un identificateur unique afin d'être distinguées entre elles. La plateforme SPACE contient déjà ce concept et encode les tâches à l'aide d'un champ de huit bits, tout comme le module SISSMA. Dans SPACE, l'identificateur du module à la source d'une requête est utilisé afin de débloquer une tâche en attente synchrone. Dans l'implantation personnalisée, ce champ identifie le module hôte de la tâche afin de signaler le bon signal d'interruption.

Version SPACE

TaskID sert pour répondre à une requête synchrone.

Version personnalisée

IRQ Line number sert pour répondre à une requête synchrone.

Identificateur du sémaphore

Puisque la synchronisation entre tâches requiert l'utilisation de sémaphores, chaque sémaphore doit aussi être identifié. Chaque requête de synchronisation porte sur un sémaphore en particulier. Huit bits sont consacrés à cet encodage, ce qui permet 256 sémaphores distincts.

Le Tableau V.2 résume l'encodage utilisé pour chaque commande émise tandis que le Tableau V.1 liste la valeur et la signification de chaque code d'erreur possible.

Tableau V.1 Description des codes d'erreur du moteur de synchronisation SISSMA

Erreur	**Code d'erreur**
Pas d'erreur	0
Code de contrôle invalide	1
Identificateur de sémaphore invalide	2
Identificateur de Ligne IRQ invalide	3
File d'attente pleine	4
Aucune tâche en attente synchrone	5

Tableau V.2 Encodage des commandes du moteur de synchronisation SISSMA

Opération	**Options**		**Code de contrôle décimal (binaire)**
	Type accès	**Section critique**	
Acquisition d'un sémaphore	Lecteur	Courte	0 (0000)
	Lecteur	Longue	1 (0001)
	Rédacteur	Courte	2 (0010)
	Rédacteur	Longue	3 (0011)
Tentative d'acquisition d'un sémaphore	Lecteur	Courte	4 (0100)
	Lecteur	Longue	5 (0101)
	Rédacteur	Courte	6 (0110)
	Rédacteur	Longue	7 (0111)
Abandon d'un sémaphore	Lecteur	Courte	8 (1000)
	Lecteur	Longue	9 (1001)
	Rédacteur	Courte	10 (1010)
	Rédacteur	Longue	11 (1011)
Requête du statut d'un sémaphore	-	-	12 (1100)
Requête de l'identificateur de la donnée SISSMA dormante	-	-	13 (1011)
Utilisation future	-	-	14 (1110)
Utilisation future	-	-	15 (1111)

V.2.1 Description des acronymes

Tableau V.3 Description des acronymes

FP	Fanion de promotion
RI	Requête initiale

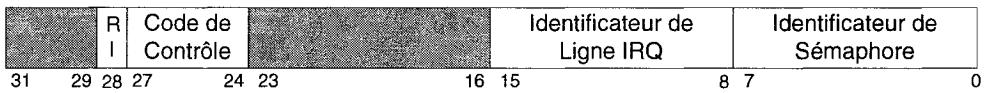
V.2.2 Opération : Acquisition d'un sémaphore

Type de requête : Lecture

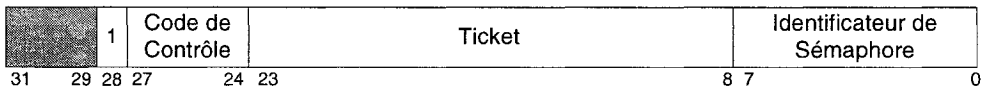
Format de la requête (paramètres):

- Code de contrôle
- Identificateur du sémaphore demandé
- Identificateur de la ligne IRQ / Ticket
- Indicateur de support pour la promotion vers une SC longue
- Identificateur de requête initiale

Dans le cas d'une requête pour une section critique longue :

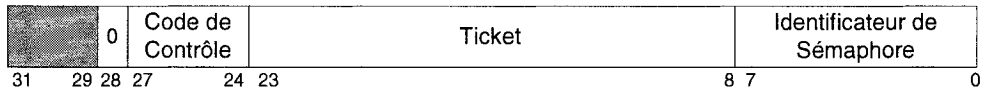


Dans le cas d'une requête pour une section critique courte (requête initiale):



Dans le cas d'une requête pour une section critique courte (requête subséquente):

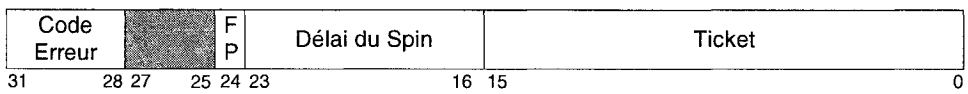
Lors de l'envoi de la requête initiale pour acquérir le sémaphore, aucun ticket n'a encore été assigné à la tâche émettrice. Dans ce cas, le ticket doit prendre la valeur de la ligne IRQ afin d'en informer le module SISSMA qui doit conserver cette valeur en mémoire si la requête est promue au type section critique longue. Dans le cas contraire, la tâche aura accès au sémaphore ou se verra assigner un ticket. Ce ticket doit être utilisé lors de la prochaine requête (mode spin).



Puisqu'il n'y a pas de moyen de distinguer une requête initiale, qui n'a pas encore de ticket valide, d'une requête ayant un ticket valide, un identificateur de requête initiale doit être utilisé. Ce bit, habituellement réservé, prend la valeur 1 lors de la requête initiale afin qu'elle se voie assigner un ticket si le sémaphore est utilisé. De la même façon, le module SISSMA doit savoir que le champ Ticket contient bel et bien la valeur du ticket de la requête et non son identificateur de ligne IRQ. Notons que ces deux variantes sont requises puisque dans le cas des SC courtes, une tâche peut émettre une requête pour le même sémaphore sans en avoir pris le contrôle entre temps. Cette situation ne se produit pas dans le cas d'une requête pour une SC longue : la tâche est automatiquement assignée un ticket et mise en attente si la ressource est en utilisation par une autre tâche.

Aussi, l'identificateur de ligne IRQ a un rôle double. En fait, s'il prend la valeur 255, c'est que la tâche à la source de la requête ne supporte pas la promotion de sa requête en une requête pour une SC longue ou ne désire pas que cette promotion ait lieu même si les conditions s'y prêtent.

Format de la réponse :



Éléments de réponse :

- Ticket (16 bits) : Un ticket de 65535 indique que la requête d'acquisition est un succès
  et que la tâche peut procéder dans sa section critique.
- Fanion de promotion vers une requête de type section critique longue (1 bit)
- Délai du prochain spin (méthode adaptative en fonction de l'historique de X

dernières requêtes et du nombre de tâches plus prioritaires (position du ticket). (8 bits)

- Code d'erreur (4 bits) : Ce code prend la valeur 0 si aucune erreur n'est survenue.

Si la requête est refusée, il est essentiel que le délai du spin ne soit pas nul, puisque les autres champs peuvent être valides tout en contenant la valeur 0.

Le fanion de promotion prend les valeurs suivantes :

0 : Il y a promotion vers une SC courte

1 : Il y a promotion vers une SC longue

Le fanion de promotion indique si la requête a été promue vers une requête de type section critique longue. Ce type de promotion se produit si une requête pour un sémaphore de type SC courte échoue et qu'une ou plusieurs tâches sont déjà en attente du sémaphore pour une SC longue.

Dans le même ordre d'idée, une requête peut être promue à une requête de type SC courte si une requête pour un sémaphore de type SC longue échoue et que la liste d'attente est pleine.

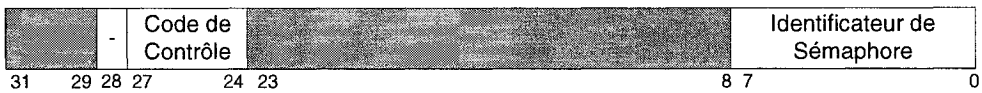
Dans le cas où une tâche est mise en attente à la suite d'une requête pour une section critique longue ou bien due à une promotion, il est possible que la file d'attente soit pleine. Dans le cas, tous les paramètres de retour sont présents, mais un message d'erreur indique la situation. La tâche opérera donc en mode actif (polling).

V.2.3 Opération : Libération d'un sémaphore

Type de requête : Lecture

Format de la requête (paramètres):

- Code de contrôle (4 bits)



- Identificateur du sémaphore abandonné (8 bits)

Format de la réponse :

Élément de réponse :

- Code d'erreur (4 bits) : Ce code prend la valeur 0 si aucune erreur n'est survenue.

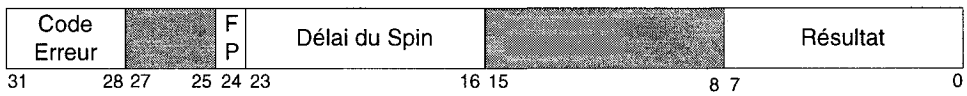
V.2.4 Opération : Tentative d'acquisition d'un sémaphore

Type de requête : Lecture

Format de la requête (paramètres):

- Code de contrôle (4 bits)
- Identificateur du sémaphore abandonné (8 bits)

Format de la réponse :



Éléments de réponse :

- Résultats de la requête : Une valeur de 255 indique que la requête d'acquisition est un succès et que la tâche peut procéder dans sa section critique. Dans le cas contraire, la valeur 0 est retournée.
- Délai du prochain spin (méthode adaptative en fonction de l'historique de X dernières requêtes et du nombre de tâches plus prioritaires (position du ticket). (16 bits)
- Fanion de section critique longue (1 bit): Indique si au moins une tâche est en attente du sémaphore afin d'accéder à une section critique longue.
- Code d'erreur (4 bits) : Ce code prend la valeur 0 si aucune erreur n'est survenue.

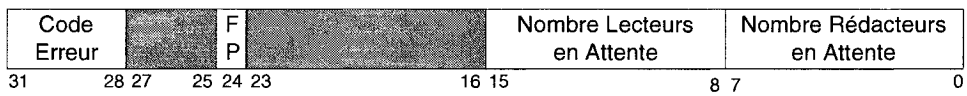
V.2.5 Opération : Requête du statut d'un sémaphore

Type de requête : Lecture

Format de la requête (paramètres):

- Code de contrôle (4 bits)
- Identificateur du sémaphore demandé (8 bits)

Format de la réponse :



Éléments de réponse :

- Nombre de lecteurs en attente (8 bits) : Si l'optimisation Rédacteur-Lecteur n'est pas utilisée ici, ce champ identifie le nombre total de tâches « lectrices » en attente ou actives sur le sémaphore (somme).

- Nombre de rédacteurs en attente (8 bits) : Si l'optimisation Rédacteur-Lecteur n'est pas utilisée ici, ce champ identifie le nombre total de tâches « rédactrices » en attente ou actives sur le sémaphore. Autrement, ce champ sera nul.
- Fanion de section critique longue (1 bit): Indique si au moins une tâche est en attente du sémaphore afin d'accéder à une section critique longue.
- Code d'erreur (4 bits) : Ce code prend la valeur 0 si aucune erreur n'est survenue.

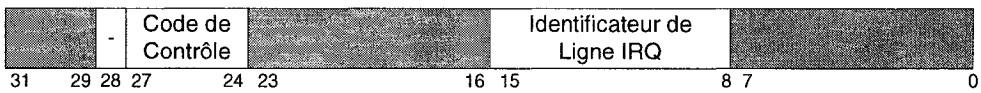
Si le sémaphore est libre, l'ensemble des 16 bits utilisés pour les compteurs (bit 15 à 0) est nul.

V.2.6 Opération : Requête de l'identificateur de la donnée SISSMA dormante

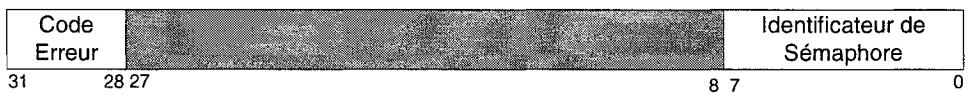
Type de requête : Lecture

Format de la requête (paramètres):

- Code de contrôle (4 bits)
- Identificateur de la ligne IRQ du module émetteur



Format de la réponse :



Éléments de réponse :

- Identificateur de la donnée SISSMA dormante (8 bits) : Cet identificateur pourra ensuite être utilisé pour éveiller une tâche en attente sur ce sémaphore.
- Code d'erreur (4 bits) : Ce code prend la valeur 0 si aucune erreur n'est survenue. Un code d'erreur est défini afin d'identifier la situation où une tâche demande un identificateur de sémaphore afin d'éveiller une tâche alors qu'aucune tâche n'est prête à être éveillée.

ANNEXE VI

INSTRUCTIONS ATOMIQUES SPÉCIALISÉES POUR LA SYNCHRONISATION MULTIPROCESSEUR

ClearThenAdd[92]

```
ClearThenAdd(p: ^word; m, i: word): void
        p^ := (p^ & ^m) + i
```

CompareAndSwap[92]

```
CompareAndSwap(p: ^word;  o, n: word): boolean
        cc : boolean = (p^ = o)
        if ( cc ) then
                p^ := n
        return cc
```

FetchAndAdd[92] / FetchAndIncrement/ ReadAndIncrement

```
FetchAndAdd(p: ^word, i: word): word
        temp: word := p^
        p^ := p^ + i
        return temp
```

FetchAndStore[80]

```
FetchAndStore( p: ^word, i: word) : word
        temp : word := p^
        p^ := i
        return temp
```

FetchClearThenAdd[92]

```
FetchClearThenAdd(p: ^word; m, i: word): word
        temp: word := p^
        p^ := (p^ & ^m) + i
```

```
        return temp
```

LoadLink / StoreConditional (LL/SC)

```
Asm(
    LL R2, (R1)        // lecture de l'état du sémaphore
    ...                // opération sur l'état de sémaphore
                       //   afin d'identifier s'il est libre
    SC R3, (R1)        // modification conditionnelle de l'état
                       //   du sémaphore
    ...                // exécution de la section critique
)
```

ANNEXE VII

MÉTRIQUES LOGICIELLES

Afin de réduire la largeur des colonnes des tableaux, les acronymes suivants sont utilisés :

Tableau VII.1 Acronymes utilisés dans l'affichage des métriques logicielles

Acronyme	Signification
CE (10M)	Nombre de cycles d'exécution (10M itérations)
ME (10M)	Marge d'erreur (10M itérations) (cycles)
CPI	Nombre de cycles par itération

Tableau VII.2 Performance de synchronisation locale sans optimisation offerte par la bibliothèque SISSMA

Opérations	# CE (10M)	ME (10M)	CPI
Lock/Unlock (Courte, Promo: Non)	1930000018	4	193
Lock/Unlock (Longue, Promo: Non)	1930000018	4	193
Lock/Unlock (Courte, Promo: Oui)	1930000018	4	193
Lock/Unlock (Longue, Promo: Oui)	1930000018	4	193
Trylock/Unlock (Courte)	1690000018	4	169
Trylock/Unlock (Longue)	1690000018	4	169
isLocked(État: libre)	1150000018	4	115
isLocked(État: occupée)	1150000018	4	115

Tableau VII.3 Performance de synchronisation à distance sans optimisation offerte par la bibliothèque SISSMA

Opérations	# CE (10M)	ME (10M)	CPI
Lock/Unlock (Courte, Promo: Non)	2610000014	4	261
Lock/Unlock (Longue, Promo: Non)	2720000013	4	272
Lock/Unlock (Courte, Promo: Oui)	2710000014	4	271
Lock/Unlock (Longue, Promo: Oui)	2760000014	4	276
Trylock/Unlock (Courte)	790000014	4	79
Trylock/Unlock (Longue)	900000013	4	90
isLocked(État: libre)	470000014	4	47
isLocked(État: occupée)	470000014	4	47

Tableau VII.4 Performance de synchronisation locale avec l'optimisation rédacteur/lecteur offerte par la bibliothèque SISSMA

Opérations	# CE (10M)	ME (10M)	CPI
Lock/Unlock (Courte, Promo: Non)	5320000015	4	532
Lock/Unlock (Longue, Promo: Non)	4770000015	4	477
Lock/Unlock (Courte, Promo: Oui)	5330000015	4	533
Lock/Unlock (Longue, Promo: Oui)	4770000015	4	477
Trylock/Unlock (Courte)	5330000015	4	533
Trylock/Unlock (Longue)	4770000015	4	477
isLocked(État: libre)	1980000017	4	198
isLocked(État: occupée)	1980000017	4	198

Tableau VII.5 Performance de synchronisation à distance avec l'optimisation rédacteur/lecteur offerte par la bibliothèque SISSMA

Opérations	# CE (10M)	ME (10M)	CPI
Lock/Unlock (Courte, Promo: Non)	2770000014	4	277
Lock/Unlock (Longue, Promo: Non)	2750000014	4	275
Lock/Unlock (Courte, Promo: Oui)	2860000014	4	286
Lock/Unlock (Longue, Promo: Oui)	2870000014	4	287
Trylock/Unlock (Courte)	920000014	4	92
Trylock/Unlock (Longue)	920000014	4	92
isLocked(État: libre)	470000014	4	47
isLocked(État: occupée)	490000013	4	49

ANNEXE VIII

MÉTRIQUES MATÉRIELLES

Afin de réduire la largeur des colonnes des tableaux, les acronymes suivants sont utilisés :

Tableau VIII.1 Acronymes utilisés dans l'affichage des métriques matérielles

Acronyme	Signification
LC	Lock Count
LCW	Lock Count Width
# S	Nombre de *slices*
# S FF	Nombre de *slices* flip-flops
# LUT 4-entrées	Nombre de LUT à 4 entrées
T (ns)	Période minimale (ns)
F (MHz)	Fréquence maximale (MHz)
# PÉ	Nombre de portes logiques équivalentes

Pour l'ensemble des métriques présentées, certains paramètres de configuration sont constants. Ces paramètres et leur valeur sont :

Lock Queue Depth	1
Max Irq Level	1
Spin Delay Clock Divider	1

Tableau VIII.2 Métriques matérielles - désactivation de l'optimisation rédacteur/lecteur et des sections critiques longues

Options		**Résultats de synthèse**					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
1	4	73	62	87	8.257	121.1	1,042
2	4	75	66	84	8.046	124.3	1,056
4	4	110	74	152	8.586	116.5	1,522
8	4	123	90	154	8.884	112.6	1,700
16	4	191	123	266	9.233	108.3	2,666
32	4	299	186	393	10.506	95.2	4,001
1	8	87	66	120	8.605	116.2	1,269
2	8	97	74	139	9.11	109.8	1,444
4	8	163	90	267	9.219	108.5	2,396
8	8	193	122	309	9.943	100.6	2,889
16	8	269	187	447	10.039	99.6	4,315
32	8	437	315	722	11.272	88.7	7,158
1	16	113	74	157	8.767	114.1	1,728
2	16	135	90	172	9.122	109.6	2,040
4	16	221	123	310	11.026	90.7	3,290
8	16	265	187	310	10.688	93.6	4,135
16	16	440	316	502	11.151	89.7	7,172
32	16	710	571	715	12.017	83.2	11,781

Tableau VIII.3 Métriques matérielles - désactivation de l'optimisation rédacteur/lecteur et activation des sections critiques longues

Options		Résultats de synthèse					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
1	4	176	105	249	9.533	104.9	2,752
2	4	321	147	460	10.284	97.2	4,698
4	4	452	224	688	12.477	80.1	7,388
8	4	848	383	1296	12.463	80.2	13,808
16	4	1599	709	2404	14.755	67.8	26,434
32	4	2964	1344	4504	15.239	65.6	49,474
1	8	210	109	322	11.046	90.5	3,203
2	8	324	159	514	11.485	87.1	5,144
4	8	528	249	854	12.617	79.3	8,635
8	8	907	431	1409	14.603	68.5	15,010
16	8	1649	800	2347	15.98	62.6	27,114
32	8	3053	1536	4850	15.851	63.1	53,230
1	16	205	121	301	11.064	90.4	3,355
2	16	338	180	516	12.72	78.6	5,554
4	16	575	301	852	13.751	72.7	9,469
8	16	927	527	1358	15.758	63.5	16,056
16	16	1792	992	2618	16.821	59.4	31,397
32	16	3539	1923	5123	16.54	60.5	60,666
2	4	286	149	428	10.117	98.8	4,527
8	4	870	408	1191	13.45	74.4	13,735
32	4	3089	1439	4519	15.998	62.5	51,778
2	8	313	165	500	11.126	89.9	5,068
8	8	908	473	1452	13.765	72.6	15,470
32	8	3173	1696	5063	16.532	60.5	55,847
2	16	344	197	519	12.522	79.9	5,692
8	16	1108	599	1674	14.979	66.8	18,621
32	16	3679	2209	5286	17.145	58.3	64,602
2	4	300	165	451	11.226	89.1	4,806

Continue sur la page suivante

Suite de la page précédante							
Options		**Résultats de synthèse**					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
8	4	901	456	1410	13.648	73.3	15,031
32	4	3220	1632	4974	16.377	61.1	54,972
2	8	324	185	513	11.219	89.1	5,330
8	8	993	552	1628	13.974	71.6	17,130
32	8	3442	2018	5515	16.695	59.9	61,198
2	16	351	233	533	12.495	80	6,096
8	16	1137	744	1736	14.836	67.4	20,088
32	16	4055	2785	6171	16.461	60.7	73,870

Tableau VIII.4 Métriques matérielles - activation de l'optimisation rédacteur/lecteur et désactivation des sections critiques longues

Options		**Résultats de synthèse**					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
1	4	NA	NA	NA	NA	NA	NA
2	4	NA	NA	NA	NA	NA	NA
4	4	NA	NA	NA	NA	NA	NA
8	4	NA	NA	NA	NA	NA	NA
16	4	NA	NA	NA	NA	NA	NA
32	4	NA	NA	NA	NA	NA	NA
1	8	83	66	104	8.14	122.9	1,170
2	8	161	74	173	8.867	112.8	1,648
4	8	151	90	217	9.766	102.4	2,059
8	8	208	123	287	9.248	108.1	2,782
16	8	341	190	484	10.298	97.1	4,590
32	8	544	314	854	11.827	84.6	7,885
1	16	157	74	264	9.189	108.8	2,194
2	16	176	90	289	9.86	101.4	2,488
4	16	181	122	307	9.924	100.8	2,836
8	16	317	186	562	10.609	94.3	4,987
16	16	483	316	835	10.857	92.1	7,809
32	16	677	570	1208	11.345	88.1	12,510

Tableau VIII.5 Métriques matérielles - activation de l'optimisation rédacteur/lecteur et des sections critiques longues

Options		Résultats de synthèse					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
1	4	NA	NA	NA	NA	NA	NA
2	4	NA	NA	NA	NA	NA	NA
4	4	NA	NA	NA	NA	NA	NA
8	4	NA	NA	NA	NA	NA	NA
16	4	NA	NA	NA	NA	NA	NA
32	4	NA	NA	NA	NA	NA	NA
1	8	166	101	244	10.672	93.7	2,726
2	8	234	142	297	9.395	106.4	3,317
4	8	402	224	521	10.424	95.9	5,724
8	8	682	389	909	11.465	87.2	10,082
16	8	979	719	1144	13.704	73	15,576
32	8	1831	1374	2080	13.259	75.4	29,395
1	16	200	109	304	12.551	79.7	3,142
2	16	267	158	392	10.866	92	4,028
4	16	443	260	637	10.515	95.1	6,700
8	16	691	453	986	12.741	78.5	11,028
16	16	1100	845	1506	13.439	74.4	18,953
32	16	2006	1629	2764	13.967	71.6	35,845
2	4	216	136	296	12.462	80.2	3,732
8	4	492	358	613	13.027	76.8	8,280
32	4	1630	1246	1921	14.664	68.2	27,543
2	8	245	146	339	12.623	79.2	4,089
8	8	523	390	669	11.937	83.8	8,619
32	8	1945	1375	2189	13.377	74.8	30,204
2	16	248	159	347	10.298	97.1	3,742
8	16	705	452	981	12.629	79.2	11,019
32	16	2075	1629	2713	14.46	69.2	35,593
2	4	215	134	294	12.636	79.1	3,704

Continue sur la page suivante

Suite de la page précédante

Options		Résultats de synthèse					
LC	**LCW**	**# S**	**# S FF**	**# LUT 4-entrées**	**T (ns)**	**F (MHz)**	**# PÉ**
8	4	487	357	607	13.545	73.8	8,237
32	4	1632	1248	1923	14.682	68.1	27,563
2	8	243	142	335	12.611	79.3	4,009
8	8	523	390	669	11.937	83.8	8,619
32	8	1946	1375	2189	13.377	74.8	30,204
2	16	247	158	349	10.895	91.8	3,746
8	16	707	452	982	12.629	79.2	11,025
32	16	2075	1629	2713	14.46	69.2	35,593