UNIVERSITE JOSEPH FOURIER DE GRENOBLE

Diplôme de Recherche Technologique

Spécialité : *« Génie Informatique »*

préparé conjointement avec

d'une part, le laboratoire VERIMAG
et d'autre part, la société SILICOMP (ORANGE BUSINESS SERVICES)

présenté et soutenu publiquement

par

Yussef BOUZOUZOU

19 décembre 2007

Titre :

Accélération des simulations de modèles de systèmes sur puce au niveau transactionnel

Directrice académique	Florence Maraninchi
Directeur industriel	Emmanuel Dufour
Co-directeur académique	Pascal Raymond

JURY

Christian Boitet	Président
Florence Maraninchi	Directrice académique
Emmanuel Dufour	Directeur industriel
Pascal Raymond	Co-directeur académique
Jean-Luc Dekeyser	Rapporteur
Frédéric Pétrot	Rapporteur
Laurent Maillet-Contoz	Examinateur

Remerciements

Je tiens à remercier tous les membres du jury ; Le président et directeur du DRT Christian Boitet pour l'attention qu'il a porté à mon travail pendant ces deux années. Je remercie aussi les rapporteurs Jean-Luc Dekeyser et Frédéric Pétrot pour avoir accepté de rapporter sur ce document. Les remarques et questions qu'ils ont soulevées m'ont permis de faire évoluer ce manuscrit pour le rendre meilleur. Je remercie aussi Laurent-Maillet Contoz pour avoir accepté d'être examinateur dans ce Jury et qui a apporté un regard avisé et pragmatique sur mon travail. Les autres membres du Jury sont mes encadrants : Florence Maraninchi et Pascal Raymond coté Verimag, et Emmanuel Dufour coté Silicomp. Bien entendu, ce travail n'aurait pas pu être réalisé sans vous qui avez toujours pu vous rendre disponibles pour faire avancer mes travaux dans la bonne direction. Pour toutes ces raisons je vous dis tout simplement, merci.

Je tiens aussi à remercier Vania Joloboff qui a été à l'origine de la collaboration, par l'intermédiaire de ce DRT, entre le laboratoire Verimag et la société Silicomp. Ensuite, je voudrais parler des personnes qui ont suivi l'évolution de mes travaux. Je pense bien entendu aux collègues de bureau, que ce soit à Verimag ou à Silicomp. Je remercie donc Dimitri, Fathi, Frédéric et Claude V. de Silicomp que j'ai ennuyé avec mes questions pendant ces deux ans, et que j'espère embêter encore longtemps. Je remercie aussi les collègues du bureau 39 à Verimag. Tout d'abord Laure qui nous a régalé de délicieux gâteaux tous les lundis matins. Je remercie Claude H. et Jérome pour toute l'aide que vous m'avez apportée pour réussir mon projet, et je sais que j'ai abusé de votre temps. Ce fut un vrai plaisir de travailler à vos cotés et vous m'avez énormément appris. Je vous souhaite de réussir tout ce que vous entreprendrez dans vos avenirs professionnels respectifs, et je n'en ai aucun doute. Je remercie aussi Matthieu, ancien pensionnaire du bureau 39, qui est maintenant devenu un « chef ». Merci pour sa disponibilité et sa réactivité quand il a fallu relire ce document et aussi pour ses conseils dans les aspects à la fois techniques et théoriques de mon travail. Enfin, pour refermer la porte du bureau 39, je remercie le tout dernier venu Giovanni.

Je voudrais aussi étendre ces remerciements à l'ensemble de l'équipe synchrone, que je ne détaille pas car la liste serait trop longue, et je risquerai d'en oublier. Ce fut un honneur d'avoir partagé ces deux années avec les membres de cette équipe, qui pour certains, ont été mes professeurs en licence et maîtrise et sans qui je ne serais pas en train d'écrire ces remerciements. Je pense notamment à Pierre-Claude, Fabienne, Anne, Karine, Paul et les professeurs des autres équipes de Verimag, Yassine, Laurent et Jean-Claude. Je remercie aussi les autres personnes de Verimag avec qui j'ai aussi travaillé avant le DRT, Romain, Liana, Laurent M., Mickaël, Marius, Erwan...

Je pense aussi que ce travail n'aurait pas été possible sans tout le personnel administratif. Je pense à Marie et Jackie de Silicomp, toute l'équipe qui entoure Christine à Verimag et qui fait un travail remarquable et aussi Martine Connan, secrétaire du DRT. Je remercie aussi Jean-Noël, l'administrateur système à Verimag et Ludovic L'Hoir qui a été son assistant durant une période. Je remercie l'ensemble des personnes que j'ai côtoyé pendant ces années, que ce soit au veri-foot ou à la vericlope-dans-le-patio ou au veri-picnic : Tayeb, Ludovic S., Mathias, Dejan, Christos, Colas, Ramzi, Yassine,

Sophie ... et bien d'autres encore.

Enfin, je veux remercier ma famille et mes proches : du coté BOUZOUZOU comme du coté HAD. Je ne citerai pas tout le monde, car pour le coup la liste serait vraiment trop longue. Je veux juste rendre hommage à mes parents qui m'ont toujours fait confiance et toujours soutenu.

Je terminerai naturellement ces remerciements par celle qui partage ma vie depuis plus de 5 ans, Hanane. Que ce fut une belle année. Je t'ai soutenue et supportée en mars pour ton diplôme et toi tu m'a renvoyé l'ascenseur pour ce DRT. Merci de ce que tu as été, de ce que tu es et de ce que tu seras à jamais.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Les modèles transactionnels pour la conception des systèmes sur puces

1.1.1 L'électronique embarquée

Les appareils électroniques sont de plus en plus présents dans notre vie quotidienne. On peut citer par exemple les décodeurs TNT, les GPS, les lecteurs DVD portables, écrans HD . . . et bien sûr les téléphones portables. Le point commun entre tous ces appareils est qu'ils sont constitués d'un circuit intégré regroupant tous les composants informatiques nécessaires à leur bon fonctionnement. C'est ce qu'on appelle un *système sur puce* (ou SoC de l'anglais System-on-Chip).

La complexité croissante de ces appareils mène naturellement à un allongement du temps de conception donc à un coût plus élevé. Cette complexité est principalement due à l'hétérogénéité des composants des SoC. D'autre part, ils sont soumis à des contraintes fortes :

- la taille des *puces* est limitée,
- la puissance de calcul doit être toujours plus grande,
- la consommation électrique doit rester maîtrisée pour garantir une certaine autonomie,
- le délai pour la mise sur le marché doit être court,
- le prix de vente doit être maîtrisé.

Le coût de développement (conception et fabrication) est donc de plus en plus élevé mais le prix de vente sur le marché reste maîtrisé grâce à la quantité vendue.

Le cœur du flot de conception classique, qui a montré ses limites de nos jours, est le niveau RTL (pour *Register Transfer Level*). Ce niveau de description est très précis de sorte qu'il est possible, à partir d'une description RTL de générer automatiquement grâce à des outils dits de *synthèse*, la

disposition des portes logiques qui permet la fabrication physique de la puce. Le problème auquel est confronté l'industrie de l'électronique embarquée est l'écart qui se creuse entre la productivité des développeurs et la quantité de code à produire pour les systèmes actuels ; ce problème est connu sous le nom de « *design gap* ». D'autre part, la simulation des descriptions RTL est trop lente pour permettre le développement et la validation du logiciel embarqué. Par exemple, pour une description d'un circuit de traitement vidéo, la simulation du décodage et l'encodage d'une seule image `MPEG4` est de l'ordre de l'heure.

1.1.2 Description de haut niveau : les modèles transactionnels

Les modèles transactionnels décrivent l'architecture et le comportement du matériel sous forme de processus concurrents. Ils peuvent servir :

- en simulation pour le développement du logiciel embarqué,
- de prototype pour l'évaluation des performances non fonctionnelles,
- d'implantation de référence pour la validation du RTL.

SystemC est une bibliothèque basée sur `C++` permettant de modéliser l'architecture et le comportement des composants matériels d'un *système sur puce* à différents niveaux d'abstraction. Elle repose sur des principes de programmation orientée objets pour la structuration statique des composants, et sur la programmation parallèle asynchrone avec des processus non préemptifs pour la description des comportements dynamiques. SystemC définit la sémantique d'exécution des modèles par l'intermédiaire d'un ordonnanceur basé sur la coopération entre les processus du modèle. Les composants communiquent via leurs ports et les ports sont reliés à des canaux de communication.

TLM (pour *Transaction Level Modeling*) est un niveau d'abstraction plus élevé que le RTL dans lequel seules les parties nécessaires au développement du logiciel sont représentées. Un modèle TLM est un ensemble de modules SystemC reliés les uns aux autres par des canaux de communications. Les communications sont réalisées par des *transactions* transportées de manière atomique depuis un module initiateur vers un module cible à travers les canaux qu'elle que soit la taille des données transportées et la taille du bus. Le concept de communication par transaction permet de modéliser rapidement des architectures de SoCs et de les simuler beaucoup plus rapidement comparativement au niveau RTL. De plus, le modèle transactionnel s'impose comme le modèle de référence pour les équipes de développement de la partie matériel synthétisable RTL et les équipes de développement du logiciel embarqué. Ainsi, cela permet le développement en parallèle du code RTL et du logiciel embarqué avec comme terrain d'entente le modèle commun.

1.1.3 Contexte du travail

Les travaux présentés dans ce document ont été réalisés dans le cadre de mon DRT, débuté le 01/01/2006 et qui s'achève le 31/12/2007. Ils ont été effectués avec Silicomp-AQL pour partenaire industriel et le laboratoire Verimag pour le suivi académique.

Verimag est un des leaders dans le domaine des systèmes embarqués. Le laboratoire accorde une attention particulière à maintenir un équilibre entre la recherche fondamentale, expérimentale et appliquée. Depuis plusieurs années, une des thématiques abordées par le laboratoire concerne la validation des SoCs au niveau transactionnel. Ceci s'est d'ailleurs concrétisé par quatre thèses en collaboration avec STMicroelectronics. Deux ont été soutenues, celles de Matthieu Moy et de Claude Helmstetter, une est en cours et sera soutenue en 2008 par Jérôme Cornet et une autre vient tout juste de démarrer. Matthieu Moy s'est intéressé aux techniques et outils nécessaires à la validation de modèle transactionnel de SoC. Claude Helmstetter a plus travaillé sur la problématique du test et a proposé

des techniques pour couvrir efficacement les différents comportements dûs à l'indéterminisme des ordonnancements et aux temps imprécis.

Silicomp est une SSTI (Société de Solutions en Technologie Informatique) qui a été récemment rachetée par le groupe Orange Business Services. Silicomp désire acquérir les compétences dans cette nouvelle approche de modélisation et ainsi pouvoir proposer du support aux industriels. Les liens entre Verimag, Silicomp et STMicroelectronics se sont renforcés ces dernières années autour de cette thématique et se concrétisent par des projets européens comme *NEVA*, ou du pôle de compétitivité *Minalogic* comme *OpenTLM* où ils sont tout trois partenaires. Ce DRT s'inscrit dans cette logique.

1.2 Problème de l'exécution parallèle des simulations sur machines multiprocesseurs

La vitesse de simulation des modèles de haut niveau a une influence importante sur les délais de conception. Pour accélérer les simulations, on a naturellement envie d'exploiter les ressources de calcul des machines multiprocesseurs, de plus en plus présentes.

La simulation des modèles SystemC est réalisée selon une politique d'ordonnancement non préemptive. Dans cet environnement, les processus doivent s'interrompre d'eux-mêmes pour permettre aux autres de s'exécuter. Les processus du modèle sont ainsi choisis un à un par un ordonnanceur, sans contrainte sur l'ordre d'exécution. Ce fonctionnement est plus connu sous le nom de multitâche coopératif. Il permet au programmeur de décrire les comportements parallèles du matériel de manière plus simple que dans un environnement préemptif. En effet, les interactions entre les processus décrivant le comportement sont beaucoup plus simples à prévoir. Réaliser un ordonnanceur SystemC est facile dans un contexte monoprocesseur puisqu'il suffit d'exécuter en séquence tous les processus.

SystemC a depuis longtemps été utilisé avec cette mécanique d'exécution. Pour ne pas changer les comportements des modèles TLM, la parallélisation éventuelle doit se faire à sémantique constante. En d'autres termes, nous voulons que les exécutions parallèles des simulations soient transparentes pour le programmeur, en dehors des gains de temps. Nous devons donc interdire les comportements qui n'auraient pas pu être observés avec un ordonnanceur séquentiel qui respecte la norme de définition de SystemC. Ceci va nous amener à définir précisément et formellement l'ensemble des comportements d'un modèle SystemC/TLM qui pourraient apparaître avec un ordonnanceur valide.

Il existe déjà des approches qui traitent de la parallélisation des simulations SystemC (voir détails dans la section 3.2), mais elles ne garantissent la préservation de la sémantique que pour des classes de modèles très simples.

1.3 Notre approche

Notre approche consiste à étudier les méthodes de parallélisation à sémantique constante pour des modèles quelconques. Paralléliser les simulations signifie que l'on va exécuter plusieurs portions de code d'un même modèle SystemC, simultanément sur plusieurs processeurs. Pour la mise en œuvre de la mécanique d'exécution parallèle, on va utiliser la couche système qui permet d'exploiter les ressources matérielles des machines multiprocesseurs. En pratique, on va se baser sur la bibliothèque de *threads POSIX (Pthread)* qui fournit un ensemble de fonctions pour la gestion de fils d'exécutions. Les primitives des *Pthread* permettent notamment de gérer leur cycle de vie, fournit des mécanismes pour leur synchronisation et la gestion des sections critiques. L'ordonnanceur du système se charge

alors de la répartition des *threads* sur les différents processeurs.

Dans notre approche, chaque *thread* créé sélectionne une à une des tâches indépendantes qu'il exécute. Pour cela, on regarde l'ensemble des ressources auxquelles les tâches accèdent, et on en choisit une qui ne partage pas de ressources. Pour savoir si des portions de code sont indépendantes, nous proposons de réaliser une analyse des dépendances sur le code du modèle avant de l'exécuter, de manière à connaître à l'avance les ressources touchées par chaque portion de code.

Notre approche combine donc une analyse statique (pour déterminer quelles portions de code sont indépendantes) et une modification de la mécanique d'exécution pour que l'ordonnanceur puisse sélectionner dynamiquement des portions de code exécutables en parallèle.

1.4 Plan du rapport

Le premier chapitre est cette introduction.

Le chapitre 2 présente le contexte technologique du travail réalisé. Il a été écrit sur la base des thèses de Matthieu Moy et de Claude Helmstetter. Nous présentons le flot de conception historique des systèmes sur puce. Nous présentons ensuite l'approche transactionnelle (TLM) pour la modélisation rapide des architectures de SoCs qui permet le développement et la validation en avance de phase du logiciel embarqué. Puis nous donnons une présentation technique de la bibliothèque SystemC et de la bibliothèque TLM (TAC) conçue par l'équipe SPG de STMicroelectronics.

Le chapitre 3 présente la problématique théorique et technique de la parallélisation des simulations SystemC au niveau transactionnel. Nous définissons la notion de *légalité* des ordonnancements. Nous poursuivons par un état de l'art des outils existants et montrons leurs limitations pour notre besoin. Nous définissons enfin le cadre théorique reposant sur la théorie des ordres partiels, permettant de résoudre le problème de préservation de la sémantique. Nous proposons notamment une granularité de parallélisation offrant un degré de parallélisme efficace et nous définissons la notion d'indépendance associée à ce grain. Nous terminons le chapitre en proposant une solution mi statique, mi dynamique au problème.

Le chapitre 4 développe la partie dynamique, ou comment mettre en œuvre la distribution des tâches SystemC sur les processeurs. Nous fournissons dans un premier temps une description détaillée du fonctionnement de l'ordonnanceur séquentiel. Nous détaillons ensuite les techniques et algorithmes implémentés. Enfin, nous donnons les tests réalisés pour vérifier le bon fonctionnement de notre implémentation.

Le chapitre 5 développe la partie statique. On y décrit les spécifications d'un outil d'analyse statique basé sur le frontal SystemC : *Pinapa*. Nous décrivons comment identifier de manière unique les points de reprise des processus. Nous donnons les idées d'analyse statique en avant en se basant sur le graphe de flot de contrôle déduit des arbres abstraits des processus. Nous terminons le chapitre en décrivant comment pourrait être réalisé un outil de vérification de la correction des simulations parallèles en se basant sur *RVS*, l'outil d'exploration des différents ordonnancements développé par Claude Helmstetter pendant sa thèse.

Le chapitre 6 décrit l'étude de cas que nous avons développée pour tester et évaluer l'accélération.

Le chapitre 7 conclut le manuscrit. On fait un résumé des contributions dans le cadre du DRT et proposons des pistes d'amélioration.

Chapitre 2

Modélisation des systèmes sur puce

Sommaire

Les systèmes sur puce sont de plus en plus répandus, et se voient confier des tâches de plus en plus complexes. Ils sont de plus soumis à des contraintes non-fonctionnelles fortes portant sur la vitesse, la consommation et le coût. Jusqu'à présent, les capacités physiques des puces progressent suffisamment vite pour faire face aux besoins. Le nombre de transistors par puce augmente ainsi d'environ 50% par an conformément à la loi de Moore, depuis des dizaines d'années ; cette augmentation devrait se poursuivre dans l'avenir, notamment avec la multiplication du nombre de processeurs par puce.

En conséquence, la conception d'un nouveau système sur puce demande un effort de plus en plus important, alors que la productivité des développeurs avec les méthodes traditionnelles n'augmente que de 30% par an. Un écart se creuse donc entre la capacité physique des puces, et la quantité de code que les développeurs peuvent écrire ; cet écart a été baptisé *design gap*. Pour lutter contre ce problème, le développement de nouvelles techniques de conception est nécessaire. Certaines de ces techniques reposent sur l'utilisation de modèles de haut-niveau, dit *transactionnels*.

Ce chapitre commence par une description du flot complet de conception des systèmes sur puce (2.1), des spécifications informelles à la description finale du matériel. Il se poursuit par une description du niveau de modélisation transactionnel (2.2), et de ses deux principales utilisations : la simulation du logiciel embarqué et l'évaluation de ses performances. Il se termine par la présentation de SystemC et du nouveau standard TLM, qui sont utilisés pour l'implémentation des modèles transactionnels (2.3).

2.1 Le flot de conception des systèmes sur puce

La réalisation d'un circuit intégré repose sur sa description *RTL* (de l'anglais : *Register Transfer Level*). Cette description est en effet le point d'entrée des outils de synthèse. Elle décrit comment sont transférées et modifiées les données entre registre à chaque top d'horloge. Un système sur puce n'est pas constitué uniquement de matériel. Il contient aussi du logiciel, et la première étape est de décider ce qui doit être conçu en matériel, et ce qui doit l'être en logiciel.

2.1.1 Partitionnement logiciel - matériel

Il y a eu de véritables progrès dans les techniques de conception de composants matériels. Cependant, le développement du matériel reste plus long et plus coûteux que celui du logiciel. L'idée est donc d'utiliser des composants matériels, suffisamment génériques pour être réutilisables, à la différence des circuits intégrés spécifiques à une application (nommés ASIC, comme *Application Specific Integrated Circuit*). Les composants matériels sont généralement appelés *IP*, comme *Intellectual Property*. Les composants logiciels viennent compléter les fonctionnalités du matériel pour obtenir la fonctionnalité globale voulue.

Les composants logiciels sont aisés à écrire, à corriger, à modifier et à adapter pour une nouvelle utilisation. La correction d'un bug peut se faire à tout moment, et même dans certains cas après que la puce soit entre les mains de l'utilisateur final. Cependant, le logiciel est beaucoup plus lent et consomme beaucoup plus qu'un composant matériel réalisant la même fonctionnalité.

Un compromis doit être trouvé entre une trop forte proportion de logiciel et une trop forte proportion de matériel. Le bon compromis dépend du contexte : un budget restreint va pousser à l'utilisation de composants logiciels alors que des contraintes temporelles sévères peuvent forcer l'utilisation de composants matériels dédiés. Généralement les fonctionnalités élémentaires, et critiques du point de vue de la vitesse, sont gérées par du matériel, alors que les autres fonctionnalités comme l'interface avec l'utilisateur sont conçues avec du logiciel. Le fait de choisir ce qui sera du matériel et ce qui sera du logiciel est appelé le *partitionnement logiciel - matériel*. Le résultat est un système intermédiaire entre un processeur généraliste et un ASIC, contenant plusieurs composants logiciels et matériels communicant et s'exécutant en parallèle. C'est cela que nous appelons *système sur puce*.

L'une des principales tâches du logiciel embarqué est de programmer les composants matériels. Par conséquent, le logiciel est très dépendant du matériel, et ne peut s'exécuter en l'état sur un ordinateur standard. Il est possible de l'exécuter soit sur la puce synthétisée elle-même, soit avec un simulateur utilisant une description du matériel.

Attendre que la première puce soit synthétisée pour exécuter le logiciel embarqué n'est pas une solution. D'une part, afin de réduire le délai avant la mise sur le marché, il est nécessaire de commencer le développement du logiciel embarqué très tôt. D'autre part, l'exécution du logiciel embarqué peut révéler des bugs dans la partie matériel. Si un bug est trouvé en simulant une description du matériel, celui-ci peut être corrigé à moindre coût. En revanche, si le bug est trouvé sur la puce synthétisée, la correction peut coûter très cher. En effet, l'une des premières étapes de la fabrication est la création du masque, et un correctif peut obliger à créer un nouveau masque, ce qui coûte autour d'un million d'euros.

Le plus naturel pour simuler le matériel est d'utiliser la description RTL. Malheureusement, celle-ci est très lente à simuler pour de gros systèmes. La cause vient du haut degré de parallélisme de la description RTL. Le matériel va vite car il se compose d'un très grand nombre de sous-systèmes s'exécutant simultanément. Pour la simulation ceux-ci doivent en revanche être exécutés en séquence, ce qui prend beaucoup de temps. Il est possible d'abstraire certains composants : certaines parties

de la description RTL sont alors remplacées par du code C censé reproduire le même comportement observable. Par exemple, une mémoire peut être simulée par un simple tableau ; un processeur par un simulateur de jeux d'instructions (ISS, *Instruction Set Simulator*). Cette technique, consistant à simuler un mélange de description RTL et de modèle C, est appelée *cosimulation* [Sch03a].

En poursuivant sur la même idée, il est possible de remplacer tous les composants RTL par des modèles en C de plus haut niveau, puis de retirer les horloges. La seule contrainte est que le logiciel embarqué puisse continuer de s'exécuter, sans modification, sur ce nouveau modèle de la puce. Ce nouveau niveau d'abstraction est dit *transactionnel*, ou TLM comme *Transaction Level Model*.

Il existe une autre alternative à la simulation sur modèle abstrait. Il s'agit d'utiliser des systèmes matériels spécifiques appelés *émulateurs matériels*. Tout comme les FPGA, il s'agit de composant matériel *reprogrammable*, c'est-à-dire que les liaisons entre cellules logiques élémentaires peuvent êtres modifiées électroniquement [Wik06]. Les modèles suffisamment grands pour simuler un système sur puce complexe sont malheureusement très coûteux, et n'offrent que peu de possibilités pour le débogage (contrairement au RTL [HTCT03]). Leur principale utilisation consiste à faire tourner de larges batteries de tests de façon automatique, dans l'espoir de trouver les derniers bugs avant la création du masque (voir par exemple [GNJ+96, HZB+03]).

La figure 2.1 montre le temps de simulation nécessaire, pour un même calcul, avec les différentes techniques présentées.

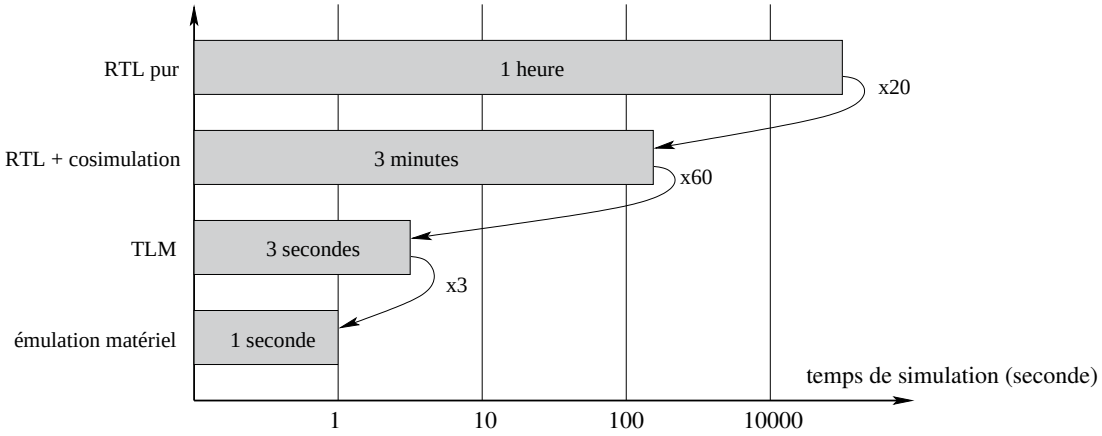

FIG. 2.1 – Temps de simulation nécessaire pour l'encodage et décodage d'une image MPEG4.

2.1.2 Les différents niveaux d'abstraction

La figure 2.2 donne un aperçu du flot de conception. Le point de départ est toujours des spécifications écrites dans un langage informel. Ensuite, une première implantation très abstraite permet de préciser et fixer les fonctionnalités voulues. Puis, une première architecture est proposée et des modèles temporisés permettent de la corriger et de la raffiner. Progressivement, on se rapproche du comportement détaillé de la puce finale. Chaque niveau d'abstraction a ses technologies et ses outils propres ; les principaux noms sont donnés à droite de la figure. Dans l'industrie, les diverses utilisations de modèles pour la conception de SoCs sont regroupées sous l'appellation *"Electronic System Level"* (ESL).

En pratique, le flot n'est pas aussi linéaire, et certains niveaux peuvent être ignorés ou remplacés par des niveaux intermédiaires. La seule description incontournable est celle au niveau RTL, d'où

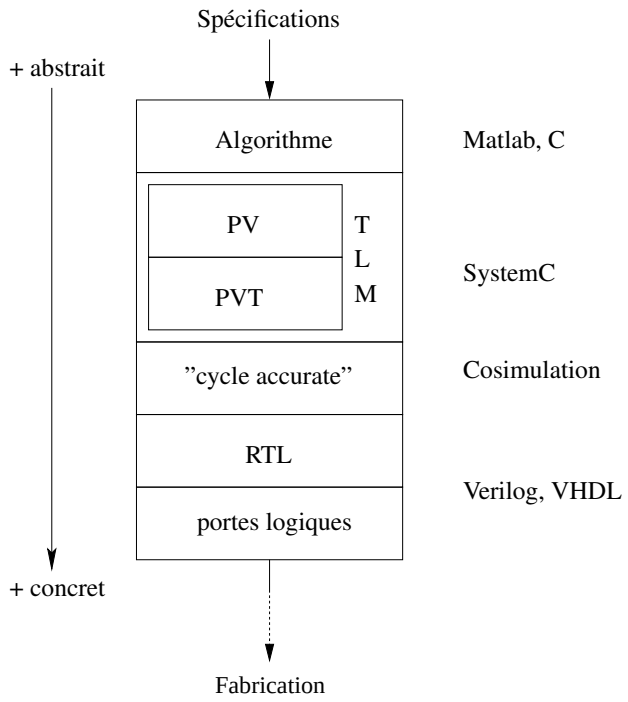

FIG. 2.2 – Les différents niveaux d'abstraction permettant d'aller des spécifications à une description synthétisable.

l'on génère automatiquement la description au niveau portes logiques. Pour les niveaux d'abstraction supérieurs, il n'existe pas d'outil automatique pour passer de l'un à l'autre. Les méthodes de raffinement manuel d'une description vers le niveau d'abstraction suivant font actuellement l'objet de recherches. En sautant un niveau d'abstraction, on peut donc espérer réduire la quantité totale de code à écrire, et en commençant la description RTL avant que les niveaux supérieurs soient finis, on peut réduire le délai avant la mise sur le marché.

2.1.2.1 Algorithme

Le niveau *algorithme* est le premier à permettre une exécution. La plupart des algorithmes sont souvent déjà disponibles, soit parce qu'ils ont déjà été écrits pour un autre contexte, soit parce qu'ils font l'objet d'une norme et qu'il existe une implantation de référence (comme pour les algorithmes de codage ou décodage vidéo). A ce niveau là, le logiciel et le matériel ne sont pas encore distingués, et le parallélisme n'est pas encore décrit. Ces descriptions sont écrites dans des langages de programmation logicielle, comme Matlab ou C++. L'une des principales utilités est de préciser et fixer les besoins grâce à des démonstrations au client.

2.1.2.2 Niveau transactionnel : TLM

Au niveau transactionnel, la description de l'architecture et du parallélisme est ajoutée. On distingue les modèles purement fonctionnels, parfois nommés PV (comme *Programmer View*), des modèles temporisés, nommés PVT comme (*Programmer View* + *Timing*). D'autre niveaux intermédiaires ont été définis [Pas02, CG03].

Les modèles fonctionnels PV permettent la simulation du logiciel embarqué, d'où leur nom. Ils doivent simuler vite afin de ne pas nuire à la productivité des programmeurs du logiciel. A ce niveau d'abstraction, la façon dont sera réalisée chaque module n'est pas encore fixée. Par exemple, tous les transferts de données peuvent se faire par un même canal de communication, considéré comme physiquement idéal (débit infini).

Pour écrire un modèle transactionnel, il est nécessaire d'avoir déjà une idée du partitionnement logiciel - matériel. Les modèles temporisés TLM-PVT permettent d'évaluer de façon précoce les choix d'architecture et de partitionnement, et donc de revoir certains choix lorsque cela apparaît nécessaire. Afin de préciser les contraintes temporelles sur les sous-systèmes, il est généralement nécessaire de préciser l'architecture et de fixer la taille des mots pouvant circuler sur les canaux de communication. Le débit des bus est alors connu, mais il n'est pas dit comment il sera physiquement obtenu.

2.1.2.3 Niveau "cycle accurate" : CA

L'étape suivante dans le raffinement vers la description RTL consiste à ajouter la notion d'horloge. Un modèle *"cycle accurate"* décrit ce qui se passe à chaque top d'horloge. Certains détails peuvent encore être décrit sous une forme qui ne permet pas une traduction automatique vers le niveau inférieur, mais globalement toute la micro-architecture est connue. Par exemple, si un processeur ou un bus utilise un pipeline, celui-ci est décrit.

2.1.2.4 Niveau transfert de registre : RTL

Le niveau *transfert de registre* RTL permet une synthèse rapide et efficace du circuit intégré. A ce niveau, la valeur de chaque bit à chaque top d'horloge est connu. Les transferts de données peuvent encore se faire par *mot*, un mot étant une donnée de la taille d'un registre. Les langages utilisés ici sont le *VHDL* et *Verilog*.

Il y a grand écart entre les niveaux TLM et RTL. Les modèles TLM ne décrivent pas les solutions matérielles ; ils sont écrits dans des langages impératifs avec une utilisation limitée du parallélisme. Une description RTL est en revanche intrinsèquement parallèle et orientée *flot de donnée*. A moins de pouvoir réutiliser des solutions existantes et génériques, un outil automatique ne peut pas inventer des solutions matérielles efficaces. Certains outils de synthèse commencent à traiter un sous-ensemble du niveau "cycle accurate" [For04, Syn05, Blu05], mais il ne faut pas espérer de synthèse efficace de modèles TLM dans un futur proche.

2.1.2.5 Niveau portes logiques, et suivants

A partir d'une description RTL, une description au niveau portes logiques est générée. Celle-ci ne contient plus qu'un réseau de portes logiques *and, or, not, ...*. Le rôle de ce niveau d'abstraction est comparable à celui de l'assembleur pour le logiciel. Ensuite, des outils de placement et routage se chargent de donner une structure en deux dimensions à ce réseaux. Cette disposition sert alors de base à la construction du masque, d'où l'on tire enfin les premiers circuits intégrés physiques.

2.1.3 Validation : vérification, simulation et test

2.1.3.1 Terminologie

S'assurer de la fiabilité d'un système est l'une des principales préoccupations dans le monde de l'informatique. Ce problème est connu sous le nom de *validation*. Plusieurs méthodologies existent,

aussi bien dans le monde du logiciel que dans celui du matériel. Cependant, leurs noms diffèrent.

Dans le monde du logiciel, le fait d'exécuter le programme pour contrôler ses sorties est appelé *test*. En général, une série de tests permet de trouver des erreurs mais ne permet pas de prouver leur absence. Le problème consistant à *prouver* qu'un programme est correct est appelé *vérification*. Ce terme utilisé seul sous-entend *vérification formelle*. On parle de *vérification semi-formelle* lorsqu'on utilise un mélange de test et de techniques formelles.

Dans le monde du matériel, le terme *vérification* est aussi utilisé mais est quasiment synonyme de *validation* puisque il ne sous-entend pas qu'il s'agit de prouver la correction de la description. Le terme *test* a en revanche un sens complètement différent : il s'agit de vérifier qu'une puce particulière n'a pas de problème physique, autrement dit que la gravure s'est bien passée. Le fait d'exécuter une description pour y trouver des erreurs est appelé *simulation*.

2.1.3.2 Aperçu des méthodologies existantes

Pour chaque puce fabriquée, il faut la *tester*, c'est-à-dire s'assurer de l'absence de défauts physiques liés à sa fabrication. C'est équivalent à vérifier que les feuilles d'un livre sont bien découpées, et qu'il ne manque pas d'encre ; mais il ne s'agit pas de vérifier le contenu, comme par exemple l'orthographe. Pour tester une puce, il faut stimuler ses parties internes et observer les sorties. Cela oblige à intégrer à la puce des systèmes matériels dont le seul rôle est de permettre son test. Cela est appelé BIST, comme *Built-In Self Test*. Cette portion de la puce ne sera plus utilisée passé la phase de test. Elle n'est pas décrite dans les modèles abstraits car elle n'y a aucune utilité.

Le logiciel embarqué est développé sur les modèles transactionnels. Normalement, il est donc prêt lorsque les premières puces sortent de la fabrication. Cependant, les vérifications finales doivent avoir lieu sur le circuit intégré final. Cela permet soit de corriger un bug du logiciel, soit dans certains cas de modifier le logiciel pour contourner un bug du matériel.

Le principal objectif est d'éviter de devoir modifier la description RTL alors qu'un masque, très coûteux, a déjà été fabriqué. Les logiciels de synthèses à partir du RTL (coûteux, eux aussi) peuvent raisonnablement être considérés comme corrects.

La vérification formelle est utilisée autant que possible. La technique la plus utilisée pour cela est la *vérification par modèle* ou *model checking*. La nature essentiellement booléenne de ces descriptions se prête bien à l'utilisation des méthodes symboliques comme les BDD ou les solveurs SAT, qui permettent de couvrir exhaustivement l'espace d'état sans les énumérer tous (outils : [HLR92, YS01, CCG+02], études de cas : [Sch03b, YG02]). Cependant, la taille des descriptions des systèmes sur puce fait que la vérification se heurte souvent au problème de l'*explosion du nombre d'états*. Certains composants, et l'assemblage des composants, doivent donc être validés par des techniques de simulations.

Un jeu de tests pour un système sur puce peut être très grand. Il peut comporter des tests générés automatiquement mais la plupart sont écrits à la main par des équipes d'ingénieurs spécialisées. Simuler tous ces tests peut demander plusieurs jours, même en distribuant le calcul sur plusieurs machines. Il n'est pas réaliste de vérifier tous les résultats à la main. Des *oracles* automatiques doivent donc être définis. Ceux-ci peuvent vérifier que les sorties respectent un ensemble de *propriétés*, généralement décrites dans une *logique temporelle* [BBP89, LMTY02]. Une autre solution est de comparer les résultats à une *implantation de référence*.

2.1.3.3 Modèle de référence

Il existe plusieurs solutions pour comparer une simulation d'une description RTL à une exécution d'un modèle plus abstrait, dit de *référence*. L'une d'elles consiste à observer les communications. Cela suppose que les canaux de communications soient semblables entre les deux niveaux d'abstraction. Une autre solution, très répandue, consiste à comparer l'état de la mémoire de la description RTL avec celle du modèle de référence, à la fin de l'exécution ou en d'autres points bien définis. Cette deuxième solution évite certains problèmes techniques, comme de devoir faire tourner ensemble du code VHDL avec du code SystemC, par exemple. Une fois les tests construits et simulés, la difficulté consiste à bien définir ce qui est pertinent dans les observations effectuées, de ce qui ne l'est pas.

Le modèle servant d'implantation de référence doit être suffisamment abstrait pour être aisé à écrire et à valider, mais aussi suffisamment concret pour que ses résultats soient comparables avec ceux du RTL. Les modèles fonctionnels TLM-PV sont souvent de bons candidats, ou les modèles temporisés TLM-PVT si l'on souhaite avoir une granularité plus proche de celle du RTL.

2.2 Les modèles transactionnels

Les modèles transactionnels décrivent l'architecture et le comportement du matériel sous forme de processus concurrents. Les trois principales utilisations sont les suivantes :

- simulateur pour le développement du logiciel embarqué ;
- prototype pour l'évaluation des performances non-fonctionnelles ;
- implantation de référence pour la validation du RTL.

Selon leurs utilisations, ils peuvent être purement fonctionnels ou temporisés.

2.2.1 Concepts communs

Nous résumons d'abord les principaux concepts des modèles TLM, tels qu'ils sont en train d'être normalisés par l'OSCI[1] [tlm06]. Une description complète de l'approche TLM peut être trouvée dans le livre [Ghe05] : *Transaction-Level Modeling with SystemC. TLM Concepts and Application for Embedded Systems*.

Les modèles TLM représentent tout d'abord une architecture ; La figure 2.3 en donne un exemple.

L'architecture est définie par un ensemble de composants, reliés entre eux par des canaux de communication. On distingue deux types de canaux de communication : les canaux transactionnels pour l'échange de données et d'informations, et les canaux de synchronisation permettant juste de notifier un événement à un autre composant. Chaque composant peut contenir zéro, un ou plusieurs processus.

Une transaction constitue un échange *atomique* de données entre un composant *initiateur* A et un composant *cible* B. L'initiateur est celui qui prend l'initiative de la transaction. Par analogie avec les applications internet, l'initiateur correspond au client et la cible au serveur. L'initiateur est parfois aussi appelé *maître*, et la cible *esclave*. Un *port initiateur*, qui permet d'initier une transaction, est toujours relié à un *port cible*, qui permet de la recevoir. Certains composants possèdent uniquement des ports initiateurs, comme les processeurs ; d'autres, comme les mémoires, possèdent uniquement des ports cibles. Enfin, certains composants possèdent les deux types de port, à l'exemple du DMA[2] que l'on programme via son port cible et qui accède à la mémoire via son port initiateur.

[1] Open SystemC Initiative

[2] abréviation de *Direct Memory Access*

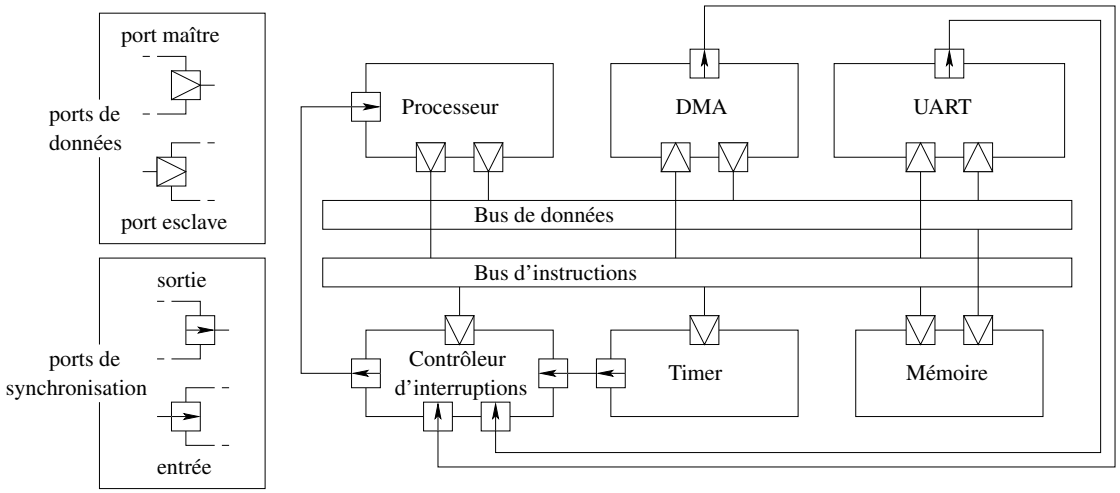

FIG. 2.3 – Exemple d'architecture d'un modèle TLM.

Les données ne circulent pas forcément de l'initiateur à la cible. Cela dépend de la *nature de la transaction* ; en général, il s'agit soit d'une *lecture*, soit d'une *écriture*. Cette *nature* est l'une des informations qui composent une transaction. La liste des informations composant une transaction est définie par un protocole. Il existe plusieurs protocoles mais la plupart définissent les informations suivantes :

- La *nature*, valant généralement *read* ou *write* ;
- L'*adresse*, généralement codée par un entier, qui détermine d'une part le composant cible, et d'autre part quel mot ou registre de ce composant est visé ;
- La *donnée* que l'on veut transmettre ou recevoir ;
- Des *méta-données* pouvant contenir : un statut de retour, des infos de durée et toutes autres informations pouvant servir au contrôle ou au debug.

Les bus sont des composants spécifiques qui se chargent de transférer les transactions qu'ils reçoivent vers les composants cibles. Pour cela, ils se basent sur l'adresse contenue dans la transaction, et sur la *carte des adresses mémoires* (ou : *memory map*), qui associe à chaque port cible une plage d'adresse (cf figure 2.4). Il est possible de coder un bus comme un autre composant, mais en pratique les modèles de bus sont fournis avec le protocole.

```
; Module name          start address       size
DMA.target_port        0x04c00             0x00080
Memory.data_port       0x80000             0x20000
...
```

FIG. 2.4 – Extrait d'une carte des adresses mémoires

La taille de la donnée est variable et dépend du protocole. Selon le niveau d'abstraction utilisé, le transfert d'une image peut par exemple se faire pixel par pixel, ligne par ligne ou en une seule fois. Inversement, certains protocoles permettent d'échanger des informations dont la taille est inférieure au mot ; cela se fait grâce au mécanisme de *byte enable* (littéralement : *octet actif*).

Le comportement du modèle est défini par des processus écrits dans un langage de haut niveau comme C++. Ils peuvent effectuer des calculs et communiquer en initiant des transactions, en réagissant à des requêtes ou en notifiant des interruptions. Un modèle est dit *transactionnel* si une tran-

saction peut être initiée par un processus en une seule instruction. Cela les distinguent des modèles de plus bas niveau dans lesquels une transaction nécessite plusieurs étapes, par exemple : écriture de l'adresse sur les signaux, puis écriture de la donnée au top d'horloge suivant, attente de la confirmation sur un autre signal, ...etc.

2.2.2 Les modèles fonctionnels (PV)

Les modèles fonctionnels, baptisés PV comme *"Programmer View"*, se destinent généralement à la simulation du logiciel embarqué. Ils doivent être suffisamment concrets pour que le logiciel puisse fonctionner comme sur la puce finale, sans retouche du code ; et ils doivent être suffisamment abstraits pour que la simulation soit rapide.

Simuler le logiciel suppose que les composants matériels soient connus, ainsi que leur interface. L'interface d'un composant est définie par une liste de ports de différentes natures. Les ports transactionnels cibles permettent généralement d'accéder à des registres, qui permettent de programmer le composant pour lui faire exécuter une requête, comme par exemple le décodage d'une image.

L'intérieur des composants est en revanche très différent de la puce finale. Les traitements sont réalisés par des algorithmes optimisés pour le logiciel. Tout ce qui concerne les propriétés non-fonctionnelles est ignoré ; les composants sont en quelque sorte physiquement idéaux : les temps de réponse, de traitement et les débits peuvent être considérés comme nuls. Cela évite les interactions implicites entre processus, comme par exemple les embouteillages pour les accès à la mémoire.

Tous les échanges d'informations ou de données entre composants ou processus doivent en revanche être explicitement synchronisés. Comme aucune durée n'est connue, il ne suffit pas d'attendre un certain temps pour être sur qu'une donnée est disponible. Les synchronisations se font grâce à des variables partagées et des événements, éventuellement véhiculés entre les composants par des fils d'interruption.

La programmation de systèmes asynchrones est reconnue comme étant plus difficile que celle des systèmes synchrones, surtout s'ils doivent être indépendants aux délais [RR02]. La cause principale vient de leur nature intrinsèquement indéterministe. Il n'est donc pas surprenant que les modèles partiellement temporisés remplacent souvent les modèles purement fonctionnels. Cela permet en effet de simplifier la conception de ces modèles, tout en respectant le cahier des charges pour la simulation du logiciel embarqué.

2.2.3 Les modèles temporisés (PVT)

Le niveau d'abstraction suivant dans le flot de conception consiste à ajouter des informations approchées sur les capacités physiques des composants. Ajouter des annotations de durée sur les instructions ou groupes d'instructions n'est pas suffisant. En effet, les propriétés temporelles dépendent aussi beaucoup de l'architecture. Les modèles temporisés ont été baptisés PVT, comme *"Programmer View plus Timing"*

Au niveau fonctionnel, un modèle de bus générique est par exemple suffisant pour tous les modèles. En revanche, pour évaluer le débit et les temps de transfert, des modifications spécifiques sont nécessaires. Tout d'abord, il faut fixer la *largeur* du bus, par exemple : 32 bits. Cela oblige à découper en plusieurs les transactions dont la donnée est plus large ; autrement dit, il faut réduire la *granularité* des communications. Ensuite, il faut tenir compte de l'architecture du bus. Il peut y avoir deux bus distincts, l'un pour les données, l'autre pour les instructions. Le bus peut aussi utiliser un *pipeline*, ou même constituer un véritable *réseau*. Pour chaque composant, la modélisation de ces

informations est indispensable pour une évaluation précise des performances temporelles du modèle complet.

Un sujet de recherche actif concerne la conception d'un modèle PVT à partir du modèle PV du même composant. Modifier le code PV pour y ajouter les informations nécessaires n'est pas une solution satisfaisante car cela permettrait de changer la fonctionnalité et tout le code devrait alors être validé à nouveau. L'idée est de décrire les informations temporelles ainsi que la nouvelle granularité des communications dans un modèle T distinct, que l'on peut voir comme un *aspect* d'un type bien particulier [CMMC07]. Les modèles PV et T sont reliés statiquement par un squelette généré automatiquement, et leurs exécutions sont ensuite *dynamiquement tissées* pour former une exécution PVT. La formalisation de l'approche devrait mener à une *preuve par construction* démontrant que la fonctionnalité est bien conservée, moyennant le respect de certaines règles.

La simulation des modèles temporisés est généralement basée sur une notion de temps globale. Toutefois, il existe aussi des techniques de modélisation et simulation basées sur des horloges locales à chaque composant. Dans [VPG06], le canal de communication se charge de resynchroniser les différentes horloges.

Il serait intéressant de pouvoir évaluer d'autres propriétés non-fonctionnelles, comme la consommation énergétique, à partir des modèles TLM. Cependant, aucun projet n'a, à ce jour, donné de résultats convaincants. Il semble que les informations nécessaires soient encore absentes à ce niveau d'abstraction ; des techniques d'analyses précises au niveau "cycle accurate" existent d'ores et déjà [BNG+06].

2.3 SystemC et la librairie TLM

La librairie TLM, permettant d'écrire des modèles du même type, est basée sur SystemC, que nous allons donc présenter en premier.

2.3.1 SystemC

2.3.1.1 Présentation des principales caractéristiques

Un langage pour la modélisation de système matériel doit respecter au minimum les trois caractéristiques ci-dessous :

- une grande vitesse de simulation ;
- une structuration en composants afin de faciliter la réutilisation ;
- une sémantique d'exécution parallèle.

De plus, une librairie implantant les structures fréquentes dans le matériel doit être disponible.

Le langage C++ répond aux deux premiers critères : la vitesse de simulation est assurée par une compilation efficace et la réutilisabilité est fournie par la couche objet. Cependant, il n'offre pas de solution simple pour la programmation parallèle. L'approche suivie par SystemC a été de compléter le langage C++ avec un mécanisme pour l'exécution parallèle de processus, ainsi qu'une librairie très fournie pour la modélisation du matériel.

SystemC est implanté sous la forme d'une librairie. Un programme SystemC est donc un programme C++. Cela a plusieurs avantages. Premièrement, cela permet un apprentissage rapide pour les nouveaux programmeurs. Deuxièmement, cela permet d'utiliser les outils d'édition déjà existant pour C++.

De plus, l'utilisation du langage SystemC est libre, et une implantation *open-source* est fournie. Cela évite d'être dépendant d'un fournisseur d'outils de conception, favorise la diffusion des modèles

décrits dans ce langage, et, ce qui nous intéresse plus particulièrement, facilite les modifications à des fins d'expérimentations. Il existe aussi des implantations propriétaires, comme NC-SystemC [Cad99] qui améliore les possibilités de cosimulation avec du VHDL.

La version 2.1 de SystemC est désormais définie par un standard IEEE [Ope05], qui a été écrit en concertation avec les principaux acteurs du domaine. Ce langage est désormais très répandu, et de nombreux outils ont été développés autour ou par-dessus, notamment pour la visualisation graphique des communications entre composants.

2.3.1.2 Alternatives à SystemC

Le langage SpecC [DGG06], lui aussi open-source, a suivi une autre approche : celle de créer un nouveau langage, inspiré de C et C++, avec les fonctionnalités requises pour la modélisation de systèmes matériels. Cette approche n'a pas eu beaucoup de succès dans l'industrie, principalement parce qu'elle ne permet pas de réutiliser des outils génériques, par exemple pour le débogage.

D'autres approches plus formelles ont été proposées pour la description de systèmes hétérogènes matériel et logiciel, comme par exemple Metropolis [BWH+03]. L'idée est là d'avoir une description formelle du langage utilisé. Cela facilite la vérification formelle, d'une part en retirant toute ambiguïté de la sémantique, d'autre part en limitant mieux ce qu'il est possible d'écrire.

Il existe aussi des langages spécifiques à la description d'architecture, et qui ignorent le comportement des composants [SPI03]. L'objectif est alors d'uniformiser la description des interfaces entre composants afin d'améliorer la réutilisabilité.

2.3.1.3 Structure d'un modèle SystemC

SystemC permet de modéliser à la fois l'architecture et le comportement du matériel, ainsi que le logiciel embarqué. Il est conçu pour être compatible avec tous les niveaux d'abstraction du flot de conception d'un système sur puce.

Les composants sont décrits grâce à une classe `sc_module` de laquelle on hérite. Cette classe peut contenir des déclarations de processus, et des ports que l'on relie ensuite aux canaux de communication. Les figures 2.6 et 2.7 donnent le code d'un programme SystemC, décrivant deux modules dont l'un est instancié deux fois. L'architecture correspondante est schématisée par la figure 2.8. Les connexions entre ports et canaux sont réalisées après l'instantiation des modules (lignes 66 à 70). Il est à noter que les modules SystemC sont hiérarchiques : ils peuvent aussi contenir d'autres modules et connexions formant un sous-système.

Les processus sont décrits par des méthodes de classes `sc_module`, sans argument et sans type de retour (lignes 8-12 et 24-37). En plus des instructions C++ normales, ils disposent d'instructions `wait` pour se mettre en attente (ligne 34), et d'évènements `sc_event` pour réveiller d'autres processus.

Deux types de connexion entre modules sont à distinguer :

1. les connexions via un canal de communication ;
2. les connexions directes entre deux modules.

Dans le premier cas, les processus de chacun des deux modules appellent des fonctions du canal du communication via des ports de la classe `sc_port` (cf figure 2.5). Un canal de communication répond à des appels de fonctions et peut notifier des événements aux processus pour les synchroniser ; il stocke des données. Les plus utilisés sont les signaux `sc_signal` et les files `sc_fifo`, mais il en existe de nombreuses variantes et l'utilisateur peut créer ses propres canaux en héritant de la classe `sc_prim_channel`.

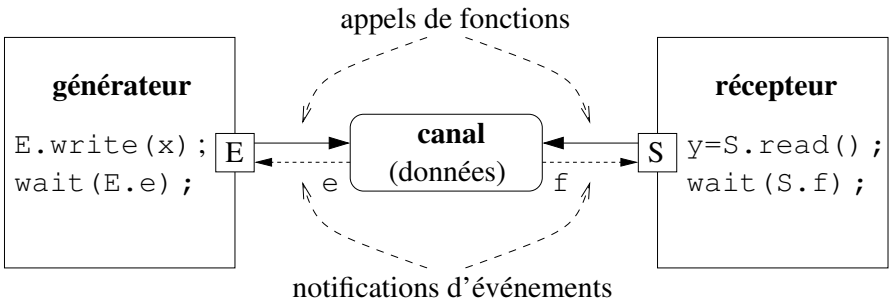

FIG. 2.5 – Connexions via un canal de communication.

Dans le second cas, le processus initiateur appelle directement une fonction d'un autre module via un port de la classe `sc_export`[3] (cf figure 2.9). Les composants bus sont aussi modélisés par des modules SystemC. Par conséquent, les liaisons de composants à bus et de bus à composants sont aussi classées dans cette catégorie. Grâce à ce mécanisme un processus peut exécuter du code d'un autre module. Cela réduit le nombre d'opérations nécessaires pour une communication particulière, mais oblige à synchroniser les processus d'une autre manière.

Lors de l'exécution du modèle, la première phase, dite d'*élaboration*, consiste à instancier les modules ainsi que leurs processus, et à connecter les différents ports. A la fin de cette phase, l'architecture du modèle est construite et la *simulation* proprement dite peut alors commencer. Cela se fait par un appel à fonction `sc_start` qui rend la main au noyau SystemC (ligne 71 de l'exemple, l'argument permet de borner la longueur de la simulation). Les synchronisations entre processus et leur ordonnancement sont détaillés plus loin.

2.3.2 La librairie TLM

Les communications dans les modèles TLM développés à STMicroelectronics n'utilisent que des liaisons directes de module à module (donc éventuellement via un module bus, mais sans canal dérivant de `sc_prim_channel`. Les premiers modèles TLM utilisaient les liaisons par canal `sc_signal` pour véhiculer les interruptions, mais depuis peu ces liaisons ont aussi été remplacées par des liaisons directes.

2.3.2.1 La fonction `transport` et les protocoles

Le cœur de la librairie TLM est désormais constitué par la déclaration ci-dessous :

```
template<REQ, RSP>
class tlm_transport_if: public sc_interface {
public:
   virtual RSP transport(const REQ&) = 0;
};
```

Comme son nom l'indique cette fonction sert à transporter une transaction, d'un port initiateur à un module cible. Bien entendu, cette simple fonction n'est pas suffisante pour écrire un modèle TLM.

[3] Historiquement cette classe devait servir pour les modules hiérarchiques, en permettant l'accès à des sous-modules ; mais son usage a ensuite été étendu pour les connexions entre modules disjoints.

```
1  #include "systemc.h"
2  #include <iostream>
3  #include <vector>
4
5  struct module1 : public sc_module {
6    sc_out<bool> port;
7    bool m_val;
8    void code1 () {
9      if (m_val) {
10       port.write(true);
11     }
12   }
13   SC_HAS_PROCESS(module1);
14   module1(sc_module_name name, bool val)
15     : sc_module(name), m_val(val) {
16     // enregistre "void code1()"
17     // comme étant un SC_THREAD (processus standard)
18     SC_THREAD(code1);
19   }
20 };
21
22 struct module2 : public sc_module {
23   sc_in<bool> ports[2];
24   void code2 () {
25     std::cout << "module2.code2"
26               << std::endl;
27     int x = ports[1].read();
28     for(int i = 0; i < 2; i++) {
29       sc_in<bool> & port = ports[i];
30       if (port.read()) {
31         std::cout << "module2.code2: exit"
32                   << std::endl;
33       }
34       wait(); // attente sans argument, utilise
35              // la liste de sensibilité statique.
36     }
37   }
38   SC_HAS_PROCESS(module2);
39   module2(sc_module_name name)
40     : sc_module(name) {
41     // enregistre "void code2()"
42     // comme étant une SC_METHOD (processus simplifié)
43     SC_METHOD(code2);
44     dont_initialize();
45     // liste de sensibilité statique pour code2
46     sensitive << ports[0];
47     sensitive << ports[1];
48   }
49 };
```

FIG. 2.6 – Exemple de programme SystemC : définition des modules.

```
50  // la vraie fonction main est dans la librairie SystemC
51  int sc_main(int argc, char ** argv) {
52    bool init1 = true;
53    bool init2 = true;
54    if (argc > 2) {
55      init1 = !strcmp(argv[1], "true");
56      init2 = !strcmp(argv[2], "true");
57    }
58    sc_signal<bool> signal1, signal2;
59    // instantiation des modules
60    module1 * instance1_1 =
61      new module1("instance1_1", init1);
62    module1 * instance1_2 =
63      new module1("instance1_2", init2);
64    module2 * instance2 =
65      new module2("instance2");
66    // connexion des ports de modules et des canaux
67    instance1_1->port.bind(signal1);
68    instance1_2->port.bind(signal2);
69    instance2->ports[0].bind(signal1);
70    instance2->ports[1].bind(signal2);
71    sc_start(-1);
72  }
```

FIG. 2.7 – Exemple de programme SystemC : fonction principale.

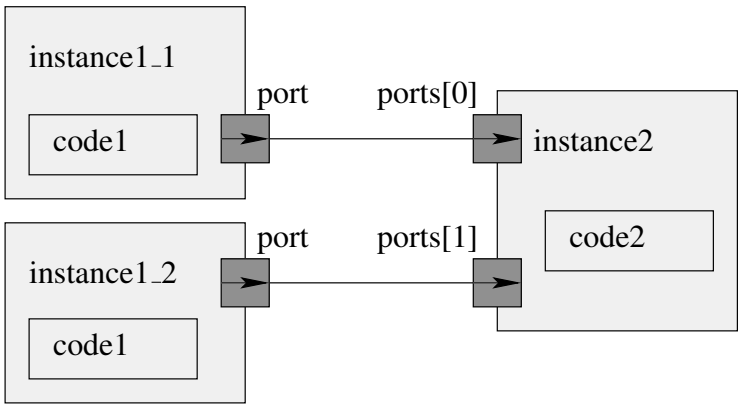

FIG. 2.8 – Aperçu graphique de l'architecture de l'exemple SystemC.

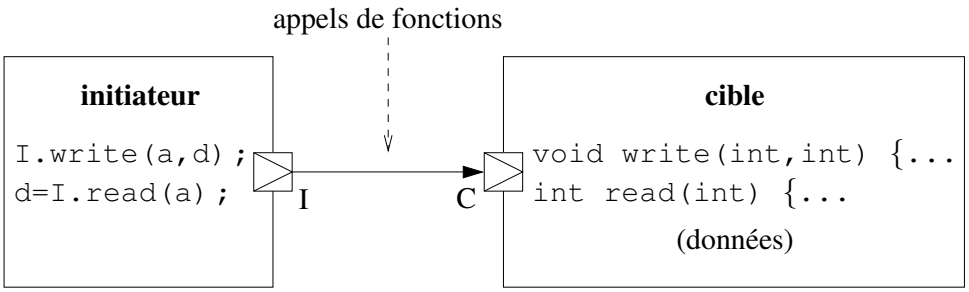

FIG. 2.9 – Connexions directes entre modules.

Pour cela, il faut disposer d'un *protocole* qui instancie cette déclaration en précisant le contenu `REQ` de la transaction et le type de réponse `RSP`.

Le protocole fournit aussi des fonctions construites par dessus la fonction `transport`. Ce sont elles qui seront utilisées par le développeur du modèle. Les plus fréquentes sont `read` et `write`. Ces fonctions sont des méthodes du port initiateur ; elles se chargent de construire un objet de type `REQ` avec la nature correspondante (`READ` ou `WRITE`), et les informations provenant de ses arguments, puis d'appeler la fonction de transport dessus. La fonction transport est implantée pour appeler une méthode du module cible qui porte le même nom. Pour le développeur, tout se passe comme s'il avait directement appelé la méthode du module cible.

Enfin, les protocoles sont souvent distribués avec des modèles de bus. Le comportement de ces modèles de bus dépend du niveau d'abstraction. Pour les protocoles dédiés aux modèles fonctionnels, le modèle de bus est souvent appelé *router* car il se contente de transmettre la transaction en fonction de son attribut adresse, sans modéliser les interactions liées à l'utilisation du bus. Dans les niveaux d'abstractions inférieurs (par exemple TLM-PVT), les modèles de bus ordonnent les différentes transactions pour pouvoir simuler son encombrement. Ils peuvent aussi proposer des fonctions spécifiques, comme par exemple `lock` et `unlock` qui permettent à un processus de se réserver le bus afin de réaliser une séquence atomique de transactions.

Pour nos travaux, nous nous intéressons essentiellement aux modèles fonctionnels. Ceux-ci utilisent deux protocoles : *TAC* et *tlm_synchro*. Le protocole TAC sert aux transactions proprement dites, c'est-à-dire aux échanges de données entre composants via un bus. Il fournit un modèle de bus : le `TAC_Router`. Le protocole tlm_synchro est destiné à la modélisation des fils d'interruptions. Ce protocole est générique (*template*) sur le type de donnée transportée, mais il est principalement utilisé avec le type booléen. Comme cela modélise des liaisons directes entre composants matériels, il n'y a ni adresse, ni bus.

2.3.2.2 Comparaison des interfaces composants et des interfaces processus

Un canal de communication, par exemple un signal SystemC, est à la fois une interface entre deux composants (car il relie deux composants ou plus), et entre processus (car les processus connectés à ses ports peuvent s'échanger des informations en appelant les fonctions du canal). Si un modèle est structuré avec un seul processus par module, et uniquement des connections via des canaux de communication, alors les interfaces composants correspondent exactement aux interfaces processus. Cela simplifie beaucoup de choses, à commencer par la compréhension globale du modèle. Ce type de structure est utilisé pour les descriptions bas niveau (RTL ou *cycle accurate*).

Les modèles transactionnels n'utilisent pas ces canaux de communication pour connecter les modules. Lors d'une transaction, un processus exécute du code de son module d'origine, puis du code

du module bus et enfin du code du module cible ; s'il le souhaite, il peut même recommencer une nouvelle transaction sans laisser les autres processus s'exécuter. Une connexion directe entre deux modules (dont l'un peut être un bus) matérialise une interface entre composants, mais n'est pas une interface entre processus. Les interfaces entre processus se situent ailleurs : à l'intérieur des modules.

Considérons par exemple un système composé de deux composants maîtres et d'une mémoire, reliés par un bus `TAC_Router`. Les composants maîtres contiennent chacun un processus. La mémoire est modélisée par un grand tableau, situé dans un module esclave sans processus. Les deux processus écrivent dans la mémoire en effectuant des transactions via le bus. Il y a alors :

- trois interfaces entre les modules, rassemblées autour du bus, et matérialisées par les ports initiateurs et cibles ;
- une interface entre les deux processus, dans le module mémoire, et matérialisée par un tableau partagé.

Il est aussi possible de considérer que le module modélisant le bus constitue une seule interface entre plusieurs composants ; c'est juste un question de terminologie. Par contre, il ne faut pas oublier les interfaces composants constitués par les fils d'interruptions. Avec le protocole tlm_synchro actuel, ces interfaces ne sont pas non plus des interfaces processus (contrairement aux signaux qu'ils ont remplacés). Les événements SystemC `sc_event` ne sont utilisés qu'à l'intérieur des modules, et servent à la communication entre processus. La figure 2.10 représente l'ensemble des interfaces pour un petit système, inspiré d'un exemple de STMicroelectronics et étendant celui décrit ci-dessus.

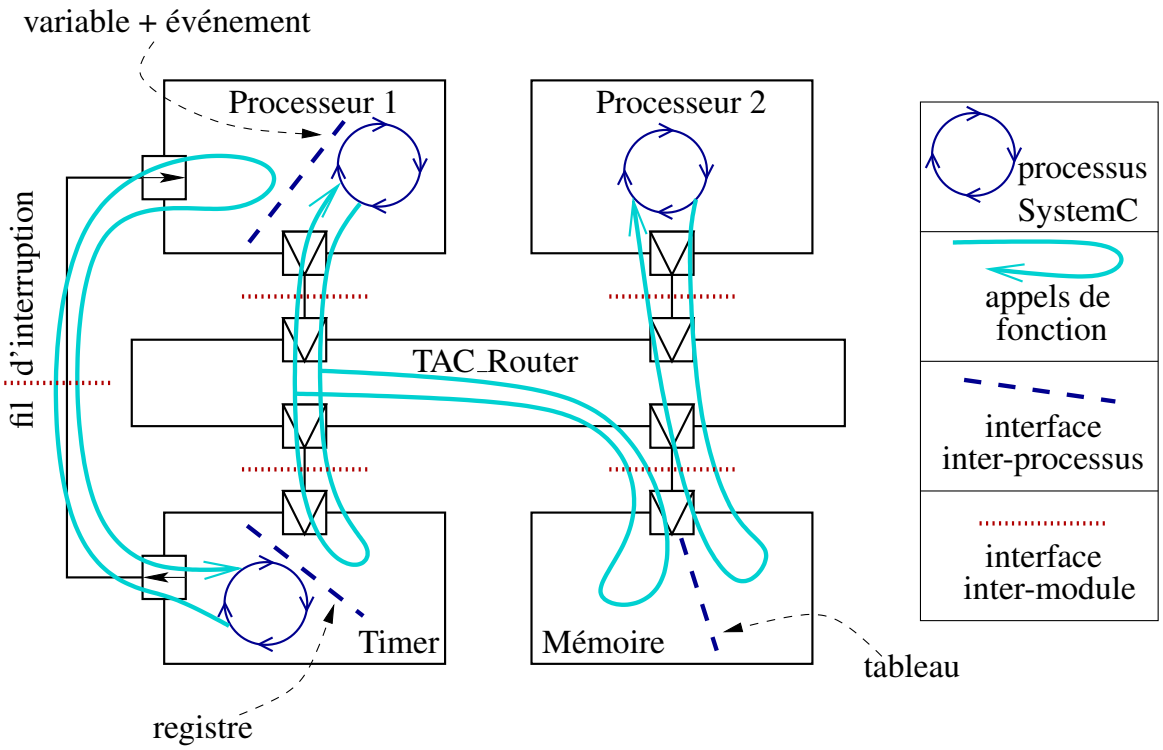

FIG. 2.10 – Exemple de modèle TLM-PV et ses interfaces.

Chapitre 3

Parallélisation SystemC à sémantique constante

Sommaire

SystemC est devenu depuis novembre 2005 un standard IEEE-1666 que l'on peut classer dans la catégorie des langages de description de matériel (HDL pour *Hardware Description Language*). En fait, SystemC n'est pas un langage à proprement parler mais plutôt une extension du langage C++ avec une bibliothèque de classes permettant de modéliser de manière abstraite le comportement des composants matériels. La bibliothèque est accompagnée d'un moteur de simulation permettant d'exécuter le comportement du matériel virtuel et de son logiciel embarqué efficacement et très tôt dans le flot de conception.

Une bibliothèque TLM (pour *Transaction Level Modeling*), extension de SystemC pour la modélisation au niveau transactionnel, à été standardisée par l'organisme OSCI (Open SystemC Initiative) sous l'impulsion de STMicroelectronics ™. Elle définit un modèle de communication entre les composants du SoC à un niveau d'abstraction plus élevé que le RTL, conservant la fonctionnalité des composants en s'abstrayant des détails de micro-architecture. Ces abstractions se font au détriment des détails architecturaux nécessaires pour les évaluations des performances, ou pour la description synthétisable. SystemC/TLM est progressivement adopté par bon nombre d'industriels pour modéliser au plus tôt dans le cycle de développement et simuler à grande vitesse les architectures de SoCs. L'objectif sous-jacent est de permettre le développement, le test et la validation du logiciel embarqué très tôt dans le flot de conception.

Le manuel de référence du langage fournit une spécification détaillée du modèle d'exécution pour la simulation des comportements concurrents de blocs matériels. Si l'on veut paralléliser les simulations dans le but de les accélérer, cela doit se faire en respectant la sémantique d'exécution définie par le standard. La réalisation d'un moteur de simulation parallèle n'a pour objectif que de simuler plus rapidement et ceci doit être transparent pour le développeur. Notre travail va se baser sur les versions SystemC-2.1 et TLM-2.0.

Ce chapitre est structuré comme suit : on commencera par définir la problématique liée aux exécutions SystemC parallélisées. Puis nous parlerons des solutions existantes les plus abouties autour de la même problématique et donnerons les limitations et restrictions de ces approches. Enfin, nous terminerons en présentant le cadre théorique de notre solution et en montrant les étapes à mettre en œuvre pour la réalisation de notre approche.

3.1 Problématique : objectif et contraintes

La parallélisation des simulations SystemC a pour objectif de réduire les temps de simulation en exploitant les ressources de calcul des machines multiprocesseurs de type SMP (Symmetric Multiprocessing), c'est-à-dire des machines comportant plus d'un processeur accédant à une mémoire physique commune.

3.1.1 Accélération des simulations

L'une des raisons majeures pour l'utilisation des modèles transactionnels dans le flot de conception des systèmes sur puce est l'accélération de la vitesse des simulations et la validation fonctionnelle du logiciel embarqué. Pourtant, le noyau de simulation SystemC officiel est défini pour une exécution monoprocessus, c'est-à-dire que les processus décrivant le comportement du matériel virtuel sont exécutés en séquence selon une politique d'ordonnancement *coopérative*, et cela limite intrinsèquement les performances des simulations. C'est dommage, puisque SystemC est utilisé pour réaliser des prototypes virtuels de systèmes finals intrinsèquement parallèles, et que ce parallélisme est explicite dans les descriptions de haut niveau. On va donc tenter de résoudre ce problème en créant un noyau d'exécution permettant l'exécution parallèle des processus et ainsi améliorer encore les performances.

Lorsque l'on utilise un prototype virtuel écrit en SystemC pour valider le logiciel embarqué, on se rend compte qu'une mauvaise programmation des modèles peut conduire à un logiciel embarqué dont le comportement dépend des ordonnancements (dans le contexte monoprocesseur). Des travaux ont d'ailleurs été réalisés par Claude Helmstetter dans le cadre de sa thèse [HMMCM06, Hel07], pour tenter de couvrir l'espace des ordonnancements possibles, et ainsi vérifier que le logiciel embarqué réagit correctement indépendamment des ordonnancements. Si l'on envisage des exécutions multiprocesseur, il faut être encore plus attentif au fait que le comportement du logiciel ne dépend ni de l'ordonnancement SystemC, ni du nombre de processeurs, ni de l'ordonnancement fourni par le système d'exploitation de la machine multiprocesseur, etc.

3.1.2 Ordonnancements « légaux »

3.1.2.1 Contraintes

La contrainte fondamentale est de respecter la spécification officielle du langage (Standard IEEE-1666 [Ope05]), de sorte que la simulation d'un modèle en parallèle soit transparente pour le développeur, en dehors des gains de temps. D'autre part, nous voulons éviter toute modification ou annotation manuelle des modèles à simuler. La figure 3.1 montre les exécutions séquentielles possibles et les observations faites par un utilisateur sur la branche du haut ; la branche du bas représente le fait que pour un même modèle en entrée, les observations en sortie d'un ordonnanceur parallèle sont cohérentes, c'est-à-dire que les mêmes sorties auraient pu être observées avec un ordonnanceur séquentiel.

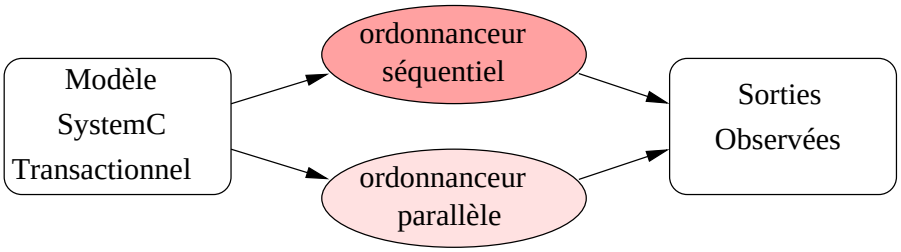

FIG. 3.1 – Des simulations parallèles.

Une autre contrainte, et pas des moindres, est le passage à l'échelle de l'implantation. En effet, il faut que le simulateur parallèle puisse traiter les modèles existant chez les industriels, notamment ceux de notre partenaire STMicroelectronics. Pour donner un ordre de grandeur, nous voulons pouvoir accélérer les simulations de modèles contenant plus de 500.000 lignes de code C++.

On va s'intéresser à la question de la « légalité » des comportements. La spécification du noyau de simulation indique que les processus SystemC (sans distinction entre `SC_THREAD`, `SC_CTHREAD` et `SC_METHOD`) implémentent une *sémantique de coroutine*, c'est-à-dire qu'une fois l'exécution d'un processus déclenchée par l'ordonnanceur, celui-ci s'exécute jusqu'à ce qu'il rende explicitement la main à l'ordonnanceur, en exécutant une instruction *wait* ou alors en retournant une valeur. Dans ce cas on parle de *multitâche coopératif*, par opposition avec le *multitâche préemptif* implémenté par la plupart des ordonnanceurs de systèmes d'exploitation.

Tout comportement qui peut être produit en respectant la sémantique de coroutine est donc légal au regard de la spécification. L'exécution d'une plateforme avec un ordonnanceur coopératif non déterministe définit un ensemble de comportements de la plateforme. Cet ensemble est un sous-ensemble des comportements « légaux », c'est-à-dire ceux autorisés par la spécification SystemC, respectant la sémantique d'exécution coopérative.

Le paragraphe ci-dessous est issu de la spécification du langage. Celle-ci autorise l'utilisation de plusieurs processeurs par une implantation SystemC, pour exécuter plusieurs processus SystemC simultanément. Par contre, elle impose que le comportement global *apparaisse identique*, c'est-à-dire que les exécutions continuent de respecter la sémantique coopérative (*coroutine* = processus non-préemptif). Enfin, elle suggère que le moyen de préserver la sémantique réside dans une *analyse des dépendances entre les processus*. On reviendra sur ce dernier point un peu plus loin.

> *An implementation running on a machine that provides hardware support for concurrent processes may permit* **two or more processes to run concurrently**, *provided that* **the behavior appears identical** *to the co-routine semantics defined in this subclause. In other words, the implementation would be obliged to* **analyze any dependencies** *between processes and constrain their execution to match the co-routine semantics.*

Dans la figure 3.2, on a représenté par un grand rond l'ensemble des comportements dits de référence `IEEE-1666`. Un sous-ensemble strict montre les exécutions possibles avec un ordonnanceur monoprocesseur particulier. La croix représente une exécution possible avec cet ordonnanceur. Enfin, on montre par l'ensemble rectangulaire les exécutions possibles en parallèle. On voit que certaines de ces exécutions sortent de l'ensemble des comportements de référence ; c'est précisément cet ensemble d'exécutions que l'on doit interdire, pour garantir le respect de la sémantique.

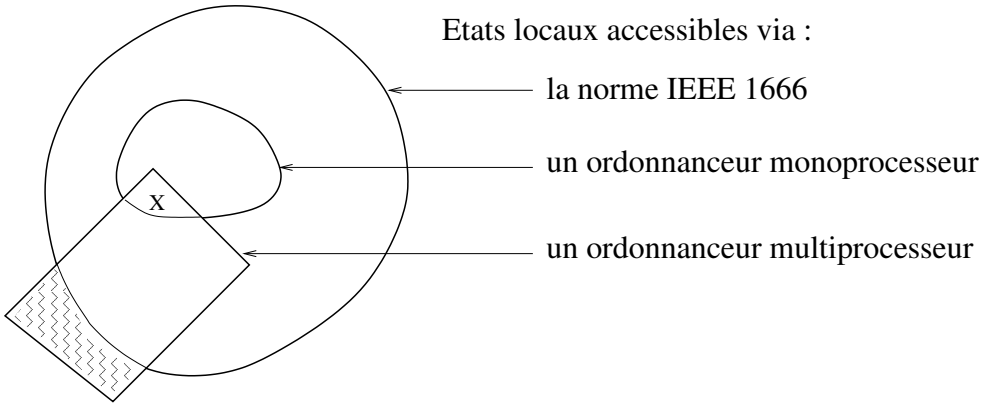

FIG. 3.2 – Ensemble des ordonnancements.

3.1.2.2 Illustration : comportement « illicite »

Pour illustrer le problème lié aux dépendances de données entre les processus, on considère les deux bouts de code a et b définis par :

$a : g++; g++;$ *// 2 instructions* α_1 *et* α_2
$b : value = g;$ *// 1 instruction* β

a effectue une instruction d'incrément d'une variable entière g. b lit la valeur de g et l'affecte à $value$. On suppose que les deux ordonnancements séquentiels $a\,;\,b$ et $b\,;\,a$ sont autorisés. Si initialement, g a pour valeur 1, alors à la fin des exécutions *valides*,

- $g = 3$ pour l'exécution séquentielle $a\,;\,b$,
- $g = 1$ pour l'exécution $b\,;\,a$.

Si maintenant, on considère les exécutions parallèles de a et b, on s'aperçoit qu'un troisième résultat est possible puisqu'il faut considérer tous les entrelacements des instructions. En effet, si l'instruction notée β s'exécute entre les deux instructions du code de a, alors $g = 2$, et ce résultat n'est pas conforme avec ce qui aurait pu être produit en séquence. Cet exemple montre un comportement que l'on veut interdire, correspondant à un comportement que l'on pourrait voir apparaître dans la zone hachurée de la figure 3.2.

3.1.2.3 Définitions

Définition 1 — Validité d'une exécution parallèle :

> *On dit qu'un comportement* C *produit en parallèle est valide, au regard de la spécification, si il existe au moins un ordonnancement séquentiel « légal » des processus qui produit* C' *équivalent à* C *(*C' ≡ C*).*

Remarque 1 *Équivalence de traces : La relation d'équivalence « ≡ » n'est définie qu'à certains points d'observation pertinents : à la fin des phases d'évaluation.*

Définition 2 — Ordonnanceur multiprocesseur valide :

> *Si pour tout comportement qui peut être produit avec notre noyau d'exécution parallélisé, un ordonnanceur monoprocesseur valide particulier exhibe un comportement équivalent, alors notre noyau d'exécution parallèle respecte le standard IEEE-1666.*

3.1.3 Parallélisation : vue globale

La figure 3.3 donne une vue générale de la parallélisation d'une exécution. Sur la partie gauche on représente les tâches à exécuter ; dans notre cadre ce sont les processus SystemC de la plateforme à simuler, identifiés par les ronds et la structure en module du système apparaît en pointillé. Dans cet environnement, les processus sont coopératifs, donc une fois qu'ils ont été choisis pour s'exécuter par le noyau de simulation, ils doivent s'interrompre explicitement pour permettre aux autres processus de la plateforme de s'exécuter. Dans ce contexte, un processus bogué, par exemple parce qu'il ne rend jamais la main à l'ordonnanceur, peut mener au blocage complet de la simulation. A droite on a représenté les ressources matérielles d'une machine, hôte de la simulation, comportant deux processeurs et la mémoire partagée (on ne fait pas de distinction entre deux processeurs et un processeur bi-cœur). Dans la partie centrale, on a représenté des processus du système d'exploitation correspondant aux fils d'exécution parallèle alloués pour cette simulation.

Pour être complet dans cette vue de la parallélisation, il nous faut encore définir les deux types d'allocation (*mapping*) :

- *mapping soft* : correspondant à la répartition des tâches SystemC sur les processus du système,
- *mapping hard* : correspondant à l'affectation des processus « système » sur les ressources matérielles.

Dans le cadre de notre travail, on va se focaliser sur le « mapping soft », c'est-à-dire comment réaliser la répartition des processus SystemC sur les processus du système d'exploitation. En effet, on peut s'abstraire du « mapping hard » que l'on délègue à l'ordonnanceur du système d'exploitation. Pour ce faire, on va utiliser la bibliothèque de processus légers à la norme POSIX, appelés *Pthreads*.

Un autre problème à résoudre est de trouver le nombre **optimal** de processus système à engager pour chaque simulation. Il faut d'une part souligner que ce nombre doit être un paramètre de simulation pour pouvoir l'adapter en fonction des ressources de la machine hôte. D'autre part, on peut considérer que les modèles que l'on veut pouvoir traiter sont composés d'un nombre de processus SystemC bien supérieur au nombre de processeurs de la machine hôte.

Il apparaît évident que ce nombre va dépendre du nombre de processeurs de la machine hôte. Si on alloue un seul « thread système », on se retrouve dans le cas d'une exécution séquentielle des processus SystemC. Si on choisit de créer autant de « threads système » que de processus SystemC, et, vu que ce nombre est bien supérieur au nombre de processeurs, on imagine aisément l'ampleur du surcoût lié aux commutations de contexte (*context switch*, en anglais). D'ailleurs, SystemC-2.1

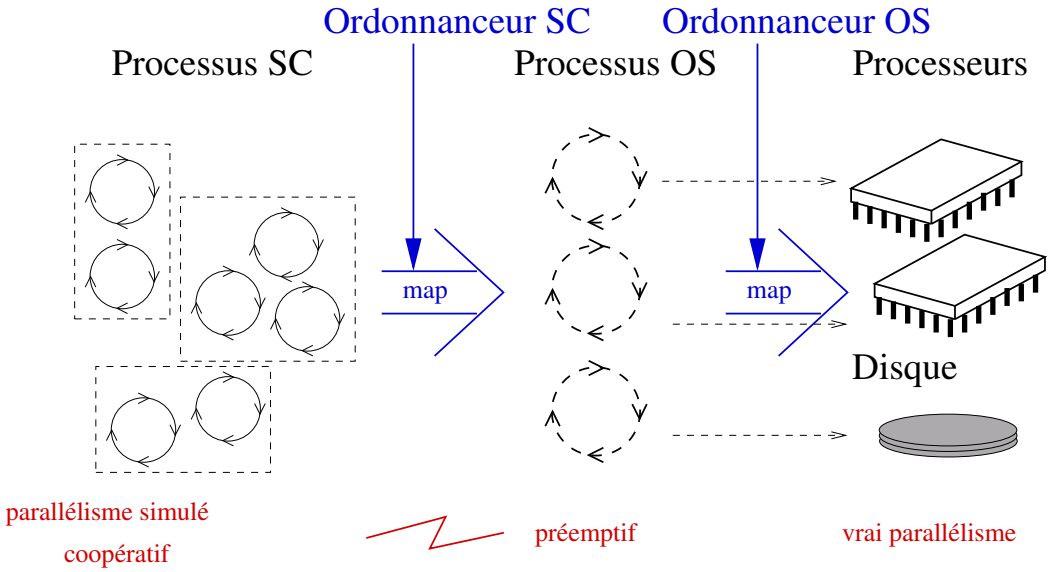

FIG. 3.3 – Vue globale de la parallélisation SystemC/TLM.

permet d'implémenter les coroutines SystemC avec l'API Pthread. La différence est que les coroutines sont ordonnancées par le noyau SystemC. Des tests ont montré que les performances se dégradent fortement avec le nombre de changements de contexte. Les résultats seraient donc catastrophiques avec l'ordonnanceur du système d'exploitation, qui est "friand" de commutations de contexte pour le partage du temps des processeurs. On ne s'intéressera pas à résoudre ce problème de manière exacte, mais de manière approchée en posant une hypothèse très intuitive qui est que *le nombre optimal de fils d'exécution concurrents à créer est proche du nombre d'unités de calcul de la machine, + 1, pour les attentes des données stockées en mémoire*.

A titre de comparaison, l'outil *make* avec l'option *-j* permet d'exécuter en parallèle les compilations (*make -j*N, crée N processus pour exécuter en parallèle les compilations). Il est communément admis que les meilleurs temps sont obtenus avec N = 3 sur une machine biprocesseurs.

3.2 Outils existants, approche structurelle

Parmi les solutions existantes traitant de la parallélisation SystemC, on va s'intéresser à deux d'entre elles, les plus abouties. L'une, réalisée dans le cadre de la thèse de Philippe Combes, a fait l'objet de publications [CCZ06] ; l'autre, développée par Éric Paire chez STMicroelectronics ™, est régulièrement utilisée par les équipes de STMicroelectronics ™. Elles sont toutes deux basées sur une approche dite « structurelle ». Plus précisément, elles se basent sur la structure en composants (ou modules SystemC) des modèles pour déterminer l'indépendance des tâches à paralléliser. Puisque les modules représentent des composants matériels qui s'exécutent en « vrai parallélisme » sur la puce finale, ce critère apparaît intuitivement judicieux. Malheureusement, on va voir que les modèles traités par ces deux approches se limitent à un sous-ensemble des programmes SystemC. Cette classe de programmes est caractérisée par les canaux de communication inter-module utilisés dans les modèles. On verra notamment que ces outils ne traitent pas les communications par transactions telles que définies dans la section 2.2.

3.2.1 Caractérisation de la classe des modèles considérés

Un programme SystemC appartient à cette classe pour les approches structurelles si et seulement si toutes les communications entre modules se font via des canaux implantant le **mécanisme de mise à jour synchrone**.

L'objectif de ce mécanisme est de protéger les lectures et écritures à travers les canaux de communication reliant les modules entre eux. En effet, les écritures ne sont répercutées qu'au δ-cycle suivant. Ce qui signifie que toutes les lectures d'un cycle d'évaluation donné font référence aux valeurs du précédent δ-cycle, et donc assurent le maintien de la cohérence des valeurs des variables partagées : c'est ce qu'on appelle « une barrière δ-cycle avec mise à jour synchrone ».

La bibliothèque SystemC fournit deux types de canaux implémentant ce mécanisme : les `sc_fifo` et les `sc_signal`. Il est possible de définir ses propres canaux implémentant ce mécanisme. Il suffit pour cela d'hériter de la classe `sc_prim_channel`, d'implémenter la fonction virtuelle `update` et de faire appel à la fonction primitive `request_update` dans l'implémentation de la fonction `write`. Le code de la table 3.1 fournit les éléments essentiels pour la définition d'un canal avec mise à jour synchrone.

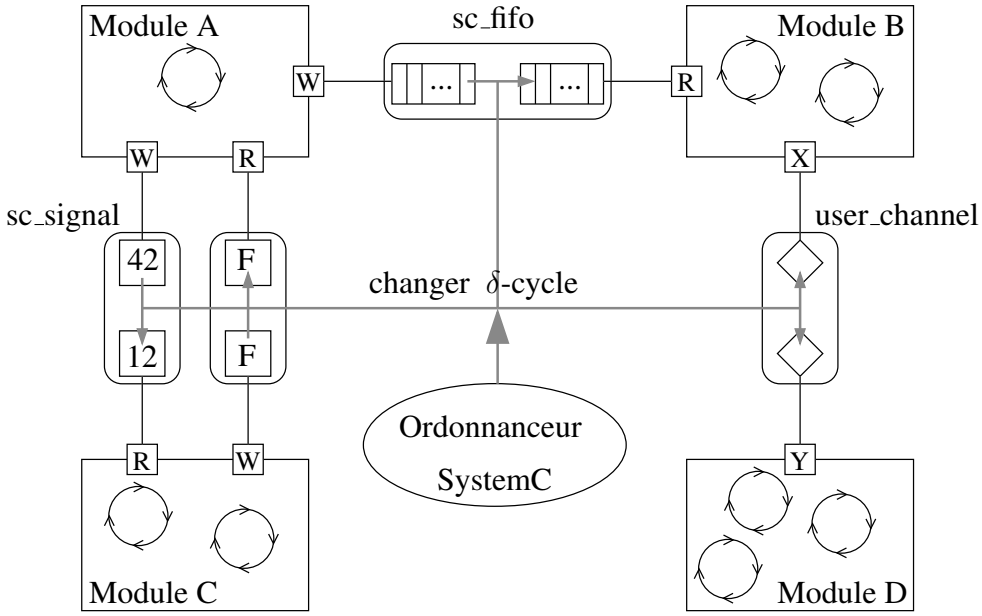

FIG. 3.4 – Modèle SystemC avec communication inter-modules globalement synchrone.

La conséquence fondamentale pour le problème qui nous intéresse ici est la suivante :

Propriété 1 *Dans tout modèle n'utilisant que des canaux avec mise à jour synchrone pour les communications inter-module : deux processus appartenant à deux modules distincts sont dits indépendants.*

La figure 3.4 illustre ce mécanisme sur trois canaux, deux prédéfinis, `sc_fifo` et `sc_signal` et un défini par l'utilisateur. Au changement de cycle, une des tâches du noyau de simulation consiste à mettre à jour les nouvelles valeurs courantes pour le cycle à venir.

```
template <class T>
class sc_my_channel
: public sc_inout_if<T>,
  public sc_prim_channel
{
  T             m_cur_val;
  T             m_new_val;
  sc_event      m_value_changed_event;
  public:

  sc_my_channel():
    sc_prim_channel( sc_gen_unique_name( "sc_my_channel" )  ),
    m_cur_val( T() ), m_new_val( T() ),
    {}
};

template <class T>
inline
void
sc_my_channel<T>::write( const T& value_ )
{
  m_new_val = value_;
  if( !( m_new_val == m_cur_val ) ) {
    request_update();
  }
}

template <class T>
inline
void
sc_my_channel<T>::update()
{
  if( !( m_new_val == m_cur_val ) ) {
    m_cur_val = m_new_val;
    //~notifier du changement de la valeur courante
    m_value_changed_event.notify_delayed();
  }
}
```

TAB. 3.1 – Création d'un type générique de canal avec mécanisme de mise à jour synchrone.

3.2.2 Application aux grappes de machines

Le problème traité par Philippe Combes est plus large que le nôtre. Il s'agit en effet de répartir la simulation d'un programme SystemC sur des machines distinctes reliées à un réseau. Dans notre cas, le fait de se limiter à des processeurs partageant la même mémoire simplifie les synchronisations entre processus et accélère les communications. L'intérêt de la simulation sur machines distinctes est évident : le coût matériel est significativement moindre (n machines monoprocesseur coûtent souvent moins cher qu'une machine avec n processeurs, avec n suffisamment grand).

Le partitionnement et la répartition des processus SystemC sur les différentes machines disponibles se fait **statiquement** grâce à des annotations `SC_NODE_MODULE` ajoutées au code source par l'utilisateur. Ces annotations permettent de rassembler des modules pour former des nœuds de modules, tels que les modules d'un même nœud communiquent beaucoup entre eux, et peu avec les modules des autres nœuds. Toutes les transitions d'un même nœud, et donc d'un même module, sont exécutées en séquence par une même machine. Comme toutes les communications entre les nœuds de modules se font avec des files (`sc_fifo`) ou des signaux (`sc_signal`), la propriété 1 s'applique, et la spécification est donc respectée.

Les synchronisations globales définies dans la spécification de SystemC ne peuvent pas être implantées sur des grappes de machines en utilisant simplement l'algorithme de référence. La solution mise en œuvre dans cet outil est basée sur le concept de *simulations parallèles à événements discrets* (*Parallel Discrete Event Simulation*, PDES), et plus précisément sur la version *conservative* (dite aussi *pessimiste*) [Bry77, CM79]. Avec cette technique, chaque machine dispose d'une horloge locale, et une machine ne fait avancer le temps localement que si elle est sûre de ne pas recevoir un message avec une date dans le passé.

Les études de cas réalisées montrent l'influence du ratio *calcul / communication*. Plus celui-ci est élevé, plus l'accélération obtenue est grande. Ce ratio dépend a priori du type d'application, mais aussi de la pertinence des annotations du programmeur.

3.2.3 Application aux machines multiprocesseurs

Éric Paire (Silicomp puis ST) a développé un prototype pour l'exécution parallèle des plateformes. Sa solution cible les machines multiprocesseurs et elle est entièrement automatisée. Son outil a été testé et validé sous Linux sur des exemples de plateformes SystemC avec canaux de communication implémentant le mécanisme de mise à jour synchrone. Une extension de l'outil permet de traiter les modèles transactionnels, mais cela est fait au détriment du respect de la spécification de l'ordonnanceur qui devient *semi-préemptif* dans le cas des transactions.

3.2.3.1 Choix d'implantation

La solution technique adoptée dans son approche est d'utiliser des **threads POSIX (Pthread)**. Les tâches à exécuter en parallèle sont regroupées par modules et l'indépendance des modules est garantie par l'utilisation de canaux à mise à jour synchrone. Dans cette perspective, on ne parle plus de processus éligibles mais de modules éligibles. Un module est dit éligible s'il contient au moins un processus éligible. Chaque fil d'exécution parallèle créé choisit un module parmi les éligibles, et se charge de l'exécution (séquentielle à l'intérieur des modules) de l'ensemble des processus éligibles qu'il contient. La répartition des tâches est automatique et dynamique contrairement à l'approche de P. Combes. Quand plus aucun module n'est éligible, on passe dans la phase de mise à jour des signaux, qui a pour effet de réveiller des processus donc des modules. Les phases de mise à jour étant

relativement courtes comparées aux phases d'évaluation, elles sont réalisées séquentiellement par le dernier fil d'exécution actif à la fin de la phase d'évaluation.

3.2.3.2 Adaptation aux modèles transactionnels

Les modèles transactionnels communiquent via des appels de fonctions d'un module initiateur vers un module destinataire. Éric Paire a complété son outil avec un mécanisme de migration de processus SystemC entre modules (on présentera les grandes lignes de ce mécanisme un peu plus loin), dans le but de le rendre utilisable avec des communications par transactions. Cette extension du domaine d'application se fait au détriment de la sémantique non-préemptive définie par la norme.

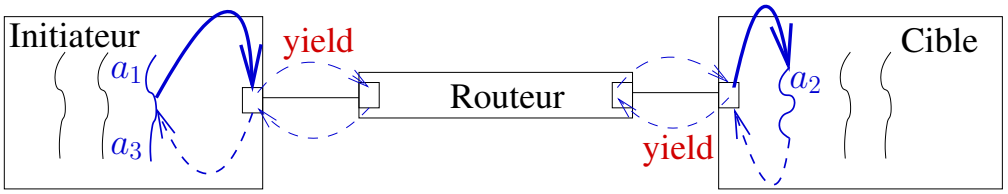

FIG. 3.5 – Réalisation d'une transaction ; insertion de points de préemption.

Considérons pour illustration la transaction schématisée par la figure 3.5. Un processus est élu pour s'exécuter dans le module Initiateur et effectue une transaction vers un autre composant, par exemple en exécutant `write(addr_target, value)`. En principe, toute la transaction devrait être atomique, mais avec le mécanisme utilisé dans cette extension, la transaction est interrompue deux fois : lors de l'appel et lors de la réponse. Nous observons donc trois portions de code a_1, a_2 et a_3 au lieu d'une seule, atomique. La portion de code a_2 est généralement exécutée par un autre processus OS que ses sœurs a_1 et a_3, puisque située dans un module différent.

Cela ajoute des comportements qui n'auraient pas été possibles avec un ordonnanceur monoprocesseur valide. Cela est problématique car il arrive que le programmeur compte sur l'atomicité d'une séquence de transactions, ce qui n'est plus le cas en considérant cette extension.

3.2.3.3 Illustration : « décalage horaire »

Dans l'exemple de la figure 3.6, on a représenté un module *TimeMng*, dont un processus A a pour rôle de modifier l'heure stockée dans un composant mémoire (*Memory*). Pour cela, il initie deux transactions de type `write` pour faire passer l'heure de 14 à 15 puis les minutes de 59 à 00.

Un processus B du module *FileSystem* initie deux transactions de types `read` pour récupérer les valeurs des heures et des minutes en mémoire, puis affiche l'heure.

Les ordonnancements possibles selon la spécification dans le cas où les deux processus sont éligibles :

– A puis B : auquel cas B affiche « 15 : 00 »
– B puis A : auquel cas B affiche « 14 : 59 »

On voit que l'exécution parallèle permettrait au processus du module (*FileSystem*) des résultats qui ne sont pas possibles avec un ordonnanceur monoprocesseur, par exemple B affiche « 15 : 59 » s'il s'exécute entre les deux actions de mise à jour de A.

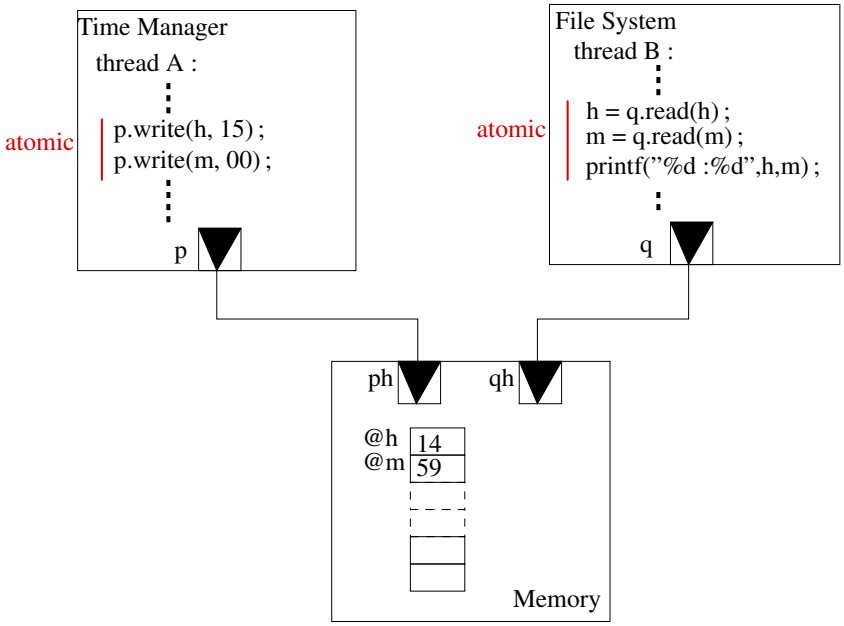

FIG. 3.6 – Décalage horaire.

3.2.4 Limitation de l'approche structurelle

On a vu que pour paralléliser les simulations SystemC il faut connaître les dépendances entre les tâches atomiques du système. Les approches structurelles se basent sur la structure en modules de la plateforme comme critère pour décider de l'indépendance. Ce choix d'implantation du prototype respecte parfaitement notre critère d'accélération. En effet, les modules étant indépendants 2 à 2 à l'intérieur d'un δ-cycle, toute transition d'un module M peut être exécutée en parallèle avec toute transition d'un module N ($M \neq N$). Ce choix est d'ailleurs intuitivement pertinent puisque les modules sont censés représenter des composants du SoC, qui s'exécutent réellement en parallèle.

Quand on considère des plateformes transactionnelles, où par définition les canaux n'implémentent pas le mécanisme de mise à jour synchrone, le critère structurel n'est plus suffisant et limite même fortement les possibilités de parallélisation. En effet, tout module initiateur d'une transaction se retrouve de fait dépendant avec le module cible servant la transaction.

Dans ce qui suit, on va tenter de définir formellement les conditions nécessaires et suffisantes pour la meilleure exploitation possible de la machine multiprocesseur en s'intéressant au *parallélisme logique*, celui caché dans les séquences de code indépendantes, et non au *parallélisme de description*, défini par la structure des modèles.

3.3 Vers une approche non structurelle

3.3.1 Les différents niveaux de granularité envisageables

La figure 3.7 donne un aperçu de trois possibilités pour le grain d'observation des dépendances.

Le premier schéma représente une vue composant d'un modèle donné, correspondant à l'approche structurelle dont nous avons parlé et dont nous avons montré les limitations. Dans la figure, on voit une communication entre les deux modules qui est réalisée par l'initiation d'une transaction depuis le

module A vers le module B, accédant (on ne distingue pas les lectures et les écritures) à une variable z du composant B. La transaction initiée par A se fait par un appel de fonction qui va exécuter du code dans le contexte du composant B, accédant à la variable z. La variable z crée une dépendance entre les modules A et B, interdisant ainsi à A et B d'être exécutés en parallèle.

Si on entre à l'intérieur des modules, on peut s'intéresser aux processus qui décrivent le comportement des modules. Cela correspond à la vue intermédiaire de la figure 3.7. On voit deux processus P et Q dans le module A et un processus R dans le module B. On voit que la transaction est initiée par le processus Q, et va accéder à une variable z, utilisée aussi par le processus R. Q et R sont dépendants et non parallélisables. Par contre, aucun objet ne lie directement P et R, donc s'ils sont tous deux éligibles, leur exécution parallèle respecterait la sémantique, comme le rappelle l'extrait du manuel de référence. Observer le modèle au niveau des processus permet de découvrir plus de tâches indépendantes.

Enfin, le troisième et dernier schéma de la figure 3.7 représente l'intérieur des processus sous forme d'automates dont les *états* sont les points d'arrêt du processus (i.e., les `wait`) et chaque *transition* correspond à la séquence de code exécutée, atomique du point de vue de l'ordonnanceur SystemC, depuis son élection jusqu'au prochain point d'arrêt. Les flèches pointillées indiquent les objets partagés auxquels chaque transition accède. Désormais, deux processus éligibles, dans des états donnés, peuvent être exécutés en parallèle si les transitions éligibles correspondantes ne sont pas reliées directement par une flèche pointillée. Il est désormais possible de paralléliser des transitions de P avec des transitions de Q, par exemple p_4 et q_2. Il s'agit d'une approximation conservant la sémantique de coroutine telle que définie par la spécification.

En résumé, plus l'observation se fait avec un grain fin, plus la simulation peut être parallélisée. La contrepartie est une analyse des dépendances beaucoup plus complexe. Nous allons étudier plus en détail la réalisation d'un ordonnanceur multiprocesseur avec une observation des dépendances à ce dernier niveau.

3.3.2 Cadre théorique de notre solution

Dans le domaine de la vérification de modèles, des techniques de *réduction d'ordre partiel* ont été développées pour tenter de minimiser la taille des systèmes, en n'explorant que les branches significatives pour les propriétés à vérifier. Dans le cadre de la simulation de modèle, les objectifs sont différents mais on va voir que les critères sont identiques.

On fournit dans ce qui suit le cadre d'application et notre vue de l'utilisation des techniques issues des réductions d'ordre partiel pour exploiter au mieux une architecture multiprocesseur. La norme impose la coopération, dont la traduction dans le code consiste en l'appel d'une fonction `wait`, pour rendre la main et permettre aux autres `sc_thread` de s'exécuter. Notre approche consiste donc à descendre au niveau le plus fin, définissant la plus petite section de code, élément atomique du point de vue de l'ordonnanceur. D'autre part, il nous faut être capable d'analyser les *dépendances* entre ces sections de code pour permettre d'accélérer les simulations en exécutant simultanément plusieurs sections *indépendantes* et ainsi garantir la validité des résultats observés.

3.3.2.1 Définitions

Une **action** est l'unité atomique d'exécution du point de vue de l'ordonnanceur du système d'exploitation. Une **transition** est une notion dynamique correspondant à l'unité atomique d'exécution du point de vue de l'ordonnanceur SystemC. Une transition peut être représentée par la séquence des actions exécutées par un processus SystemC depuis son élection par l'ordonnanceur jusqu'au retour

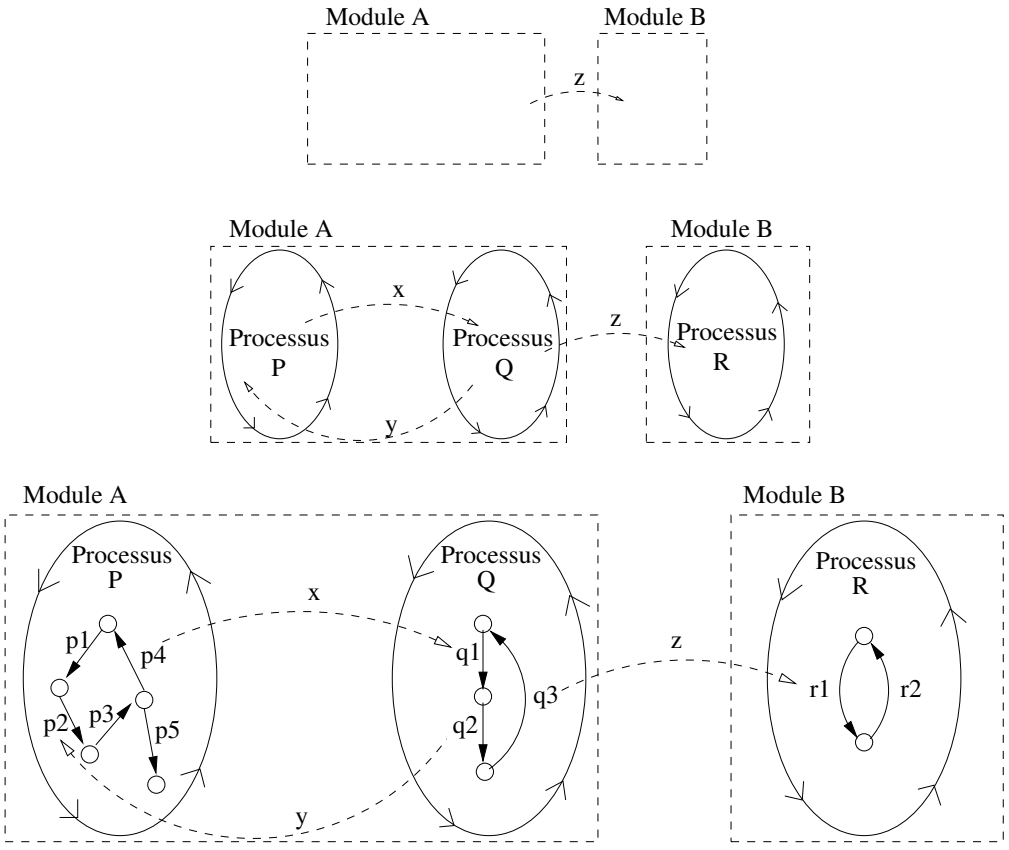

FIG. 3.7 – Représentations schématiques d'un modèle, à différents niveaux de granularité. Les rectangles pointillés représentent les modules SystemC, les grandes ellipses représentent les processus SystemC et les flèches des automates représentent les transitions SystemC. Les dépendances entre transitions sont représentées par les flèches pointillées, étiquetées par le nom des objets partagés.

du contrôle au noyau. Pour la suite, on identifiera les transitions par des lettres latines (a, b, c, ...) et les actions par des lettres grecques ($\alpha, \beta, \gamma, \ldots$).

Un **état** du système à l'exécution est défini par « une photographie » de la mémoire (valeurs des variables, mémoire partagée, pile d'exécution des co-routines, ...) à un instant donné. Tout état est défini par l'état initial du système et la séquence des transitions menant à cet état.

Pour une exécution on dispose d'une relation d'ordre ($<$) entre les transitions qui est :
- totale pour un programme séquentiel ($<_{seq}$),
- partielle pour un programme parallèle ($<_{par}$).

Soient a et b deux transitions du système. On dit que a et b **commutent** si les exécutions séquentielles $a; b$ et $b; a$ existent et mènent au même état. Autrement dit, depuis chaque état S où l'exécution séquentielle $a; b$ existe et mène à un état S', la séquence $b; a$ existe aussi et mène au même état S'. Les transitions a et b sont alors dites **indépendantes** dans le cadre des réductions d'ordre partiel.

On rappelle qu'une transition SystemC est la séquence ininterrompue des actions exécutées par un processus SystemC, ce qui est implicite dans le contexte monoprocesseur et du noyau d'exécution SystemC qui est non préemptif. Dans un contexte multiprocesseur, la granularité d'exécution est alors plus fine puisqu'il faut considérer l'entrelacement des actions des transitions qui s'exécutent en parallèle.

Pour illustrer le propos, on reprend les deux bouts de codes a et b définis dans la section 3.1.2.2, et dont on a vu que leur exécution parallèle était à interdire pour respecter la sémantique du langage :

$a : g++; g++;$ *// 2 actions* α_1 *et* α_2
$b : value = g;$ *// 1 action* β

Dans ce cas, les deux transitions ne **commutent pas** donc elles sont **dépendantes**.

Par contre si on remplace $a : g++; g++;$ par $a : g++; g--;$, alors cette fois ci les transitions commutent.

$a : g++; g--;$ *// 2 actions* α_1 *et* α_2
$b : value = g;$ *// 1 action* β

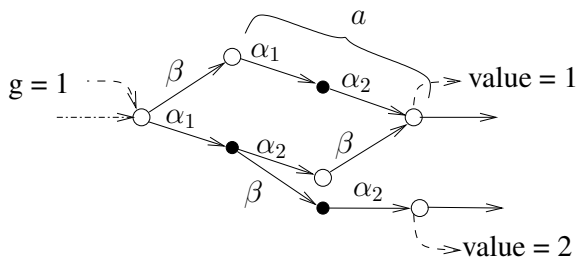

FIG. 3.8 – Entrelacement des instructions.

On voit que les exécutions séquentielles $a;\ b$ et $b;\ a$ mènent au même résultat, alors que l'exécution « réellement » parallèle $a \parallel b$ peut donner un résultat différent si l'action de b s'exécute entre les deux actions de la transition $a : \alpha_1; \beta; \alpha_2$.

Définition 3 —

Deux transitions sont dites **dépendantes** *si elles ne commutent pas* **ou alors** *si leur exécution parallèle peut mener à un* **état différent de leur exécution séquentielle**.

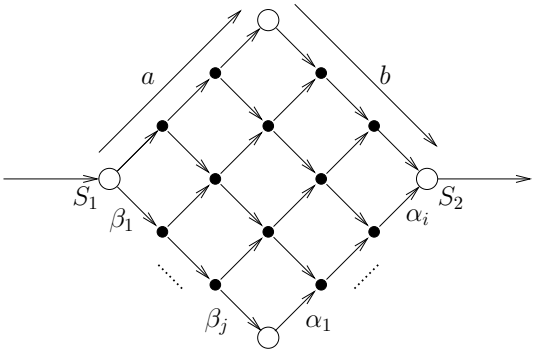

FIG. 3.9 – Entrelacement des actions de deux transitions indépendantes

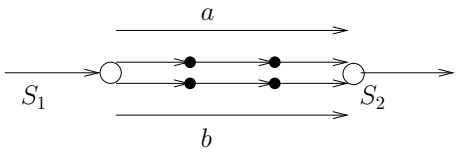

FIG. 3.10 – Execution parallèle des transitions a et b

Définition 4 —

Si deux transitions sont indépendantes, alors on peut les exécuter simultanément.

On a vu dans ce chapitre, quels étaient les ingrédients nécessaires et suffisants pour garantir une parallélisation conservant la sémantique d'exécution définie par la norme. Nous définissons notamment la notion d'indépendance pour la parallélisation de modèle SystemC au niveau transactionnel. Il nous faut être capable de déterminer si un processus SystemC prêt à s'exécuter, où éligible, ne va pas « toucher » des variables qui pourrait être modifiées par d'autres processus en cours d'exécution. Il est bien entendu possible de faire la distinction entre les lectures et les écritures sur variable. Plusieurs processus peuvent naturellement accéder à un même objet en lecture, mais dès lors qu'un processus peut modifier la valeur de l'objet, toute autre action de lecture ou d'écriture d'un autre processus est à proscrire. Nous avons aussi déterminé la granularité d'exécution parallèle la plus fine, correspondant à l'atomicité des coroutines telle que la voit l'ordonnanceur SystemC. Celle-ci fait apparaître le plus haut degré de parallélisme dans les modèles considérés et nous a permis de donner la sémantique d'exécution parallèle conforme à la norme.

La suite du document présente les techniques mises en œuvre pour la mécanique d'exécution distribuée, qui fait l'objet du chapitre 4. Ensuite, on montrera comment résoudre les dépendances de données aux différents niveaux de granularité présentés ; c'est l'objet du chapitre 5.

Chapitre 4

Mise en œuvre des mécanismes d'exécution parallèle

Sommaire

Dans ce chapitre on va s'intéresser à la partie dynamique de notre solution, donc techniques et algorithmes à mettre en œuvre pour la mécanique d'exécution répartie. La partie statique, qui traite des informations nécessaires pour l'identification des dépendances entre les transitions SystemC, fait l'objet du chapitre 5.

Pour la mise en place des mécaniques d'exécution parallèle, on fait l'hypothèse que les informations statiques sont disponibles, donc que l'on sait à tout moment décider de l'indépendance de deux transitions éligibles. Nous commencerons par une explication détaillée de la spécification du fonctionnement du noyau de simulation séquentiel. Les exemples fournis sont basés sur l'implémentation du noyau défini par l'OSCI [Ope05]. Puis, nous donnerons les techniques et algorithmes mis en œuvre pour la répartition des tâches SystemC sur les fils d'exécutions parallèles du système d'exploitation. Ceux-ci sont créés avec la bibliothèque de *thread POSIX* (Pthread), dont on donnera les principales caractéristiques. Enfin, nous clorons ce chapitre en donnant les tests réalisés pour vérifier certaines propriétés sur la mécanique mise en place, comme l'absence de *deadlock*.

4.1 Le noyau de simulation SystemC

L'exécution d'une plateforme se décompose en deux phases : une phase d'*élaboration* suivie d'une phase de *simulation*. La phase dite d'élaboration réalise la construction de la plateforme. C'est pendant cette phase que les modules, ports et canaux sont instanciés et les processus enregistrés. Les

modules sont connectés via leurs ports aux canaux de communications. La structure de la plateforme est construite une fois pour toutes pendant l'élaboration et reste valide jusqu'à la fin de la simulation.

On va maintenant détailler la spécification du fonctionnement du noyau de simulation et de son ordonnanceur.

4.1.1 Algorithme de l'ordonnanceur

4.1.1.1 Fonctionnement global

Une fois l'élaboration de la plateforme effectuée, on passe dans la phase de simulation de la plateforme qui démarre avec l'appel de la fonction *sc_start()*.

La figure 4.1 montre la mécanique d'exécution des processus, implémentée par l'ordonnanceur. Après une phase d'initialisation des structures de données internes au noyau, l'ordonnanceur, dont une spécification détaillée est donnée dans le manuel de référence du langage [Ope05], lance une à une l'exécution des processus éligibles. Les processus se suspendent d'eux-mêmes quand ils rencontrent une instruction d'attente *wait*. Il y a deux types d'instructions d'attente : un processus peut attendre sur du temps en spécifiant une durée, ou attendre la notification d'un ou plusieurs événements [1]. C'est ce qu'on appelle la phase d'évaluation des processus (EV). L'évaluation d'un processus peut provoquer la notification d'événements, c'est ce qu'on appelle *notification immédiate*, qui ont pour effet de rendre éligibles des processus en attente, sensibles à ces événements. Il peut aussi programmer des *notifications retardées* ou *temporisées*.

Quand plus aucun processus n'est éligible, on passe dans la phase de mise à jour (UP pour *update*). Cette phase se décompose en deux traitements distincts :

- d'une part, le noyau doit exécuter toutes les fonctions *update* résultant de l'appel à la fonction *request_update* des canaux primitifs et dont on a déjà discuté dans la section 3.2),
- d'autre part, le noyau traite les *notifications retardées*, résultant des appels aux fonctions `wait(SC_ZERO_TIME)` ou `notify(SC_ZERO_TIME)` de la phase d'évaluation précédente.

La combinaison d'une phase d'évaluation des processus (EV) et d'une phase de mise à jour (UP) est appelée *δ-cycle*.

Si après une phase de mise à jour plus aucun processus n'est éligible, on passe dans une phase dite de *changement de temps de simulation* (TE pour *time elapse*). Un cycle d'exécution est une séquence de δ-cycles entre deux temps de simulation. La figure 4.2 montre l'enchaînement des différentes phases lors des simulations.

4.1.1.2 États des coroutines à l'exécution

La figure 4.3 illustre les différents états des processus SystemC à l'exécution. Initialement, tous les processus sont dans l'état *eligible* exceptés ceux pour lesquels la fonction *dont_initialize()* a été appelée, qui se retrouvent dans l'état *stopped*. Un processus passe de l'état *eligible* à l'état *running* quand il est choisi par l'ordonnanceur.

À tout moment de la simulation, au plus un processus se trouve dans l'état *running* dans le cas monoprocesseur. Le processus s'exécute jusqu'à ce qu'il rende explicitement la main à l'ordonnanceur. On peut distinguer :

[1] Il existe aussi un type d'instruction d'attente qui consiste à attendre un événement pendant une durée maximale fixée, ou *timeout*. Passé ce délai, le processus est rajouté à la liste des éligibles.

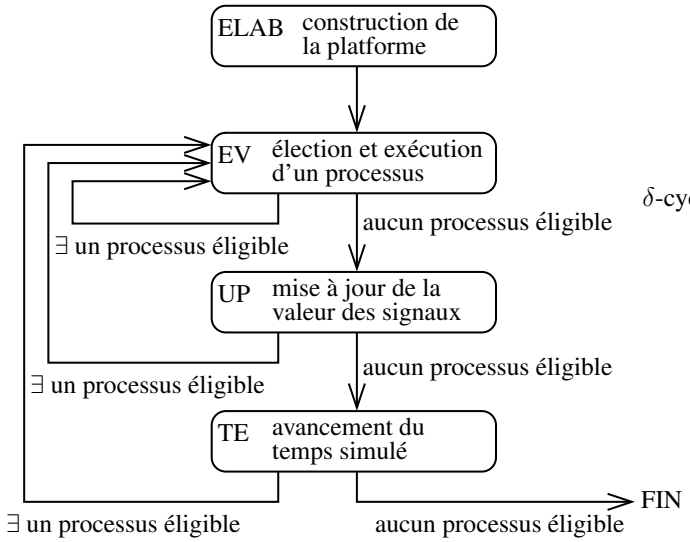

FIG. 4.1 – Ordonnanceur SystemC

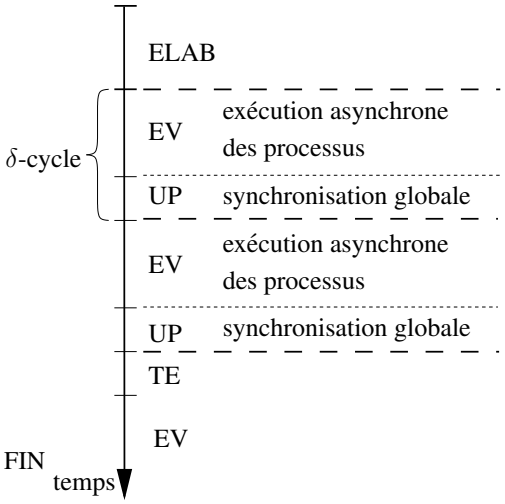

FIG. 4.2 – Diagramme d'une exécution

– les cas où un processus atteint une instruction *wait()*, auquel cas il se met en attente d'être réveillé puis réélu,
– des cas où le processus retourne, auquel cas il passe dans l'état *dead* et ne sera plus exécuté.

Les processus en attente deviennent éligibles par la notification d'événements (immédiate, délayée ou temporelle).

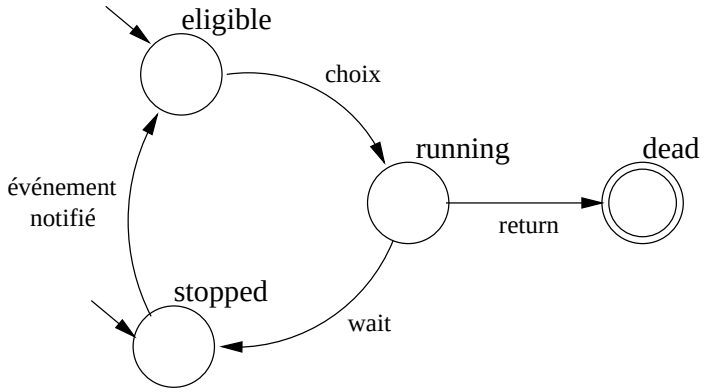

FIG. 4.3 – État des processus à l'exécution

4.1.1.3 Algorithme de l'ordonnanceur

L'algorithme 1 donne l'algorithme en pseudo code de l'ordonnanceur. On fait apparaître les trois boucles du schéma (4.1) et les ensembles nécessaires pour la gestion des différents types de notifications d'événements possibles et de leurs conséquences.

Les différents types de notifications :

1. les notifications immédiates (`notify` sans argument) : réveillent dans la phase d'évaluation courante tous les processus en attente sur l'événement notifié,
2. notifications retardées d'un δ-cycle : il existe plusieurs schémas permettant de réveiller des processus pendant la phase de mise à jour :
   - l'appel à la fonction *request_update()* lors de l'écriture dans un canal primitif : les fonctions *update* correspondante sont exécutées pendant la phase de mise à jour des canaux, qui a pour effet de réveiller les processus sensibles au changement de valeur du canal,
   - `wait(SC_ZERO_TIME)` : réveille le processus ayant exécuté l'instruction,
   - `e.notify(SC_ZERO_TIME)` : réveille les processus sensibles à l'événement `e`, qui deviennent éligibles à la phase d'évaluation suivante.
3. notifications temporisées : avancement du temps de simulation jusqu'à l'expiration d'un délai sur un événement, auquel cas les processus en attente sur cet événement sont réveillés, ou alors, les processus dont l'échéance est plus courte sont réveillés.

Description des ensembles utilisés dans l'algorithme :

- E : ensemble des processus éligible,
- S : ensemble des triplets (P, e, d) des processus P attendant l'événement e avec un délais d ,
- T : ensemble de couple (P, t) des processus P attendant l'écoulement du temps d'une durée t,
- NI : (Notification Immédiate) ensemble d'événements émis avec effet immédiat,
- NR : (Notification Retardée) ensemble d'événements émis avec effet retardé d'un δ-cycle,
- NT : (Notification Temporisée) ensemble de couples (e, t) d'événement e émis avec effet retardé d'un temps t,
- C : ensemble des canaux primitifs (*sc_prim_channel*) dont la fonction *request_update()* a été invoquée.

La phase de simulation proprement dite est précédée d'un phase d'initialisation. Elle fournit les valeurs initiales des canaux primitifs et des ensembles mentionnés plus haut et est réalisée en trois pas :

- exécuter la phase d'élaboration,
- exécuter les fonctions *update()* des canaux primitifs,
- exécuter la phase de notification *delta*.

Initialement, l'ensemble E contient tous les processus, exceptés ceux pour lesquels la fonction *dont_initialize()* a été appelée.

4.1.2 Notes sur l'implantation des coroutines

Un processus SystemC peut s'interrompre en exécutant une instruction *wait()* et doit pouvoir reprendre son exécution depuis chacun de ses points d'arrêt. Pour cela, il faut qu'il puisse sauvegarder et restaurer son contexte d'exécution, donc sa propre pile d'exécution.

SystemC fournit plusieurs implantations possibles pour les coroutines :

- L'option fiber est spécifique pour les systèmes d'exploitation `Windows` ; cette bibliothèque fournit des primitives pour la commutation de contexte, permettant ainsi à une application Windows de créer, stopper et reprendre des fils d'exécutions.
- L'option quickThread est livrée avec la bibliothèque SystemC avec différentes implémentations [Kep93] ; cette bibliothèque fonctionne exclusivement sur des systèmes de type `UNIX`.
- L'option Pthread est introduite depuis la version SystemC 2.1. Elle est basée sur la norme POSIX dont une implantation est disponible sur de nombreux système d'exploitation. Cette

Algorithme 1 : Algorithme de l'ordonnanceur SystemC

Données :
E : ensemble des processus éligible ;
S : ensemble des triplets (P, e, d) des processus P attendant l'événement e avec un délais d'expiration d (*timeout*);
T : ensemble de couple (P, t) des processus P attendant l'écoulement du temps d'une durée t;
NI : ensemble d'événements émis avec effet immédiat, ;
NR : ensemble d'événements émis avec effet retardé d'un δ-cycle, ;
NT : ensemble de couples (e, t) d'événement e émis avec effet retardé d'un temps t;
C : ensemble des canaux primitifs dont *request_update()* a été invoquée;
E, S, T, NI, NR, NT, C := Initialisation() ;
tant que $E \neq \emptyset$ **faire**
  tant que $E \neq \emptyset$ **faire**
    tant que $E \neq \emptyset$ **faire**
      `// phase d'évaluation (EV)`
      soit $p \in E$; $E := E - \{p\}$;
      $(NI, NR', NT', C') := execute(p)$;
      pour chaque $e \in NI$ **faire**
        pour chaque $(q, e, d) \in S$ **faire**
          $E := E \cup \{q\}$; $S := S - \{(q, e, d)\}$;
      $NR := NR \cup NR'$; $NT := NT \cup NT'$; $C := C \cup C'$;
      si p s'arrête sur $wait(e)$ **alors** $S := S \cup \{(p, e, +\infty)\}$;
      si p s'arrête sur $wait(e, d)$ **alors** $S := S \cup \{(p, e, d)\}$;
      si p s'arrête sur $wait(t)$ **alors** $T := T \cup \{(p, t)\}$;
    `// phase de mise à jour (UP)`
    $NR' := execute\_update(C)$;
    $NR := NR \cup NR'$;
    pour chaque $e \in NR$ **faire**
      pour chaque $(p, e, d) \in S$ **faire**
        $E := E \cup \{p\}$; $S := S - \{(p, e, d)\}$;
    pour chaque $x = (p, d) \in T / d = 0$ **faire**
      $E := E \cup \{p\}$; $T := T - \{x\}$;
  `// Phase d'avancement du temps simulé (TE)`
  $m = min(\{t/(p, t) \in T\} \cup \{t/(p, e, t) \in S\})$;
  pour chaque $x = (p, d) \in T$ **faire**
    $T := T - \{x\}$;
    si *m = d* **alors** $E := E \cup \{p\}$ **sinon** $T = T \cup (p, d - m)$
  pour chaque $x = (e, d) \in NT$ **faire**
    $NT := NT - \{x\}$;
    si *m = d* **alors**
      pour chaque $(p, e, h) \in S$ **faire**
        $E := E \cup \{p\}$; $S := S - \{(p, e, h)\}$;
    sinon $NT = NT \cup (e, d - m)$
  pour chaque *x = (p, e, d)* $\in S$ **faire**
    $S := S - \{x\}$;
    si *m = d* **alors** $E := E \cup \{p\}$ **sinon** $S = S \cup (p, e, d - m)$

option pour l'implantation des coroutines SystemC est surtout importante pour des raisons de portabilité, et de débogage.

Les implantations des processus par *fibers* et *quickThreads* sont plus efficaces en terme de changement de contexte que les *Pthreads* puisque les premiers sont plus légers du fait qu'ils ne sont pas prévus pour permettre la préemption. Par contre, les conséquences sur la vitesse des simulations ne sont pas quantifiées dans le manuel de référence du langage. Dans la mesure où notre objectif principal est d'accélérer les simulations, nous avons donc mis en place des tests pour comparer les temps de simulation avec les implantations *quickthread* et *pthread*.

4.1.2.1 Comparaison des performances avec différentes implantations des coroutines : exemple « ping-pong »

L'exemple, dont le code est donné dans la figure 4.4 définit un module SystemC possédant deux processus A et B qui réalisent un nombre élevé de changements de contexte. Leur comportement est symétrique. Initialement, seul le processus A est éligible.

- A réveille B en notifiant un événement `pong`, puis se met en attente sur l'événement `ping`
- B réveille A en notifiant un événement `ping`, puis se met en attente sur `pong`.

Le tableau 4.1.2.1 récapitule les résultats fournis par la commande `time` sous `unix`. Le temps total est le plus intéressant, puisqu'il donne le temps passé devant la machine à attendre la fin des simulations. On voit que la simulation est de l'ordre de 100 fois plus rapide avec les `quickthreads`. Bien entendu cette plateforme n'est pas du tout réaliste et représente un cas extrême d'utilisation. Compte tenu de nos objectifs, on ne s'intéressera par la suite qu'aux simulations de modèles dont l'implantation *quickthread* des coroutines.

Il faut aussi souligner que la bibliothèque *quickthread* fournit une implémentation pour différentes familles d'architectures de processeurs (80386, 88000, DEC AXP (Alpha), HP PA, KSR, SPARC V8, VAX, ...). De plus, il est possible d'étendre encore la bibliothèque pour supporter d'autres architectures. C'est précisément ce qui est fait dans [VMD] qui propose une extension de la bibliothèque pour supporter des architectures Intel 64 bits.

4.1.2.2 Exécution des coroutines

La figure 4.6 montre une optimisation implémentée dans le noyau de simulation OSCI qui permet de réduire le nombre de changements de contexte en chaînant l'exécution des coroutines. Sur le schéma du haut on représente les changements de contextes entre le noyau et les coroutines. Les flèches montantes correspondent à la sauvegarde du contexte du noyau et le chargement du contexte des coroutines ; les flèches descendantes la sauvegarde du contexte des coroutines et le retour du contrôle au noyau.

L'idée de cette optimisation est de déporter la partie du code du noyau qui choisit le prochain éligible dans la gestion de contexte des coroutines. Comme le montre le schéma du bas, quand un processus SystemC atteint une instruction *wait*, la sauvegarde de son contexte est immédiatement suivie de la restauration du contexte de la prochaine coroutine à exécuter. Cette optimisation permet de passer de $2 * N$ à $N + 1$ changements de contexte (où N est le nombre de processus SystemC à exécuter).

Pour la description de notre algorithme d'exécution distribué, et ce pour plus de clarté, on distinguera le code exécuté par les coroutines du code exécuté par le noyau d'exécution réparti. L'optimisation présentée au paragraphe précédent est bien entendu réalisable de la même manière.

```
#include "systemc.h"

class top : public sc_module
{
public:
  sc_event ping;
  sc_event pong;

  SC_HAS_PROCESS(top);
  top(sc_module_name name) :
    sc_module(name) {
    SC_THREAD(A);
    SC_THREAD(B);
    dont_initialize();
    sensitive << pong;
  }

  void A(){
    long i;
    cout << "ping pong start" << endl;
    for (i=0; i<3600000; i++) {
      pong.notify();
      wait(ping);
    }
    cout << "ping pong end" << i << endl;
  }

  void B(){
    while(true){
      ping.notify();
      wait(pong);
    }
  }
};
```

FIG. 4.4 – Comparaison des temps de simulations avec l'implantation *pthread* et *quickthread* des coroutines

temps (sec) / processus	user	system	total
Quick thread	2,464	0,004	2,47
Pthread	45,514	127,735	215,53

FIG. 4.5 – Comparaison des temps d'exécution.

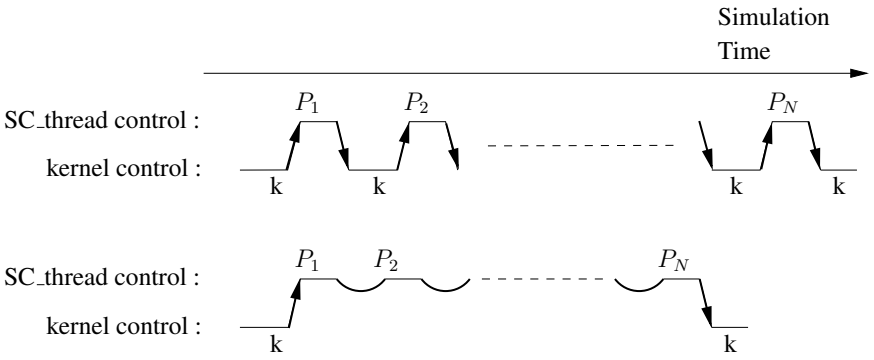

FIG. 4.6 – Optimisation des changements de contexte du noyau OSCI

4.2 Mécanique d'exécution distribuée

On a vu dans la section 3.1.3 une présentation générale de la parallélisation dans laquelle on mentionne les deux allocations à réaliser que sont :

- répartir les tâches (les processus SystemC) sur les fils d'exécutions, qu'on a appelé *mapping soft*,
- répartir sur les processeurs les processus systèmes (*mapping hard*).

On ne va s'intéresser qu'à la première allocation. L'utilisation de la bibliothèque *pthread* nous permet de nous abstraire du *mapping hard*. En effet, elle permet de créer des processus utilisateurs partageant le même espace mémoire que le processus père. Les processus fils sont alors ordonnancés par le noyau, en prenant en compte la répartition des charges sur les processeurs. On délègue donc ce problème à l'ordonnanceur du noyau système.

Avant de décrire l'algorithme de répartition des tâches SystemC, on va décrire la bibliothèque *pthread* et en donner les principales caractéristiques. On rappelle notre intuition que le nombre optimal de *pthread* à créer est proche du nombre de processeurs de la machine plus un. On fixe donc cette hypothèse pour la suite.

4.2.1 La bibliothèque *pthread*

La bibliothèque fournit un ensemble de fonction pour la gestion des processus utilisateur *pthread*. On va en décrire les principales fonctions permettant de gérer :

- le cycle de vie, principalement la création et la terminaison des processus,
- la concurrence, où comment garantir l'intégrité des données partagées,
- et les synchronisations à travers l'utilisation de *condition variable*.

4.2.1.1 Cycle de vie des processus

La fonction *pthread_create()* crée un nouveau processus qui s'exécute en parallèle avec le processus créateur. Elle prend pour paramètres :

- *thread* : identifiant unique du processus créé, fourni par le système,
- *attr* : les attributs du processus à créer,
- *routine* : fonction C qui sera exécutée une fois le processus créé,
- *arg* : structure de donnée contenant les paramètres de la fonction.

La fonction *pthread_exit()* permet de terminer un processus de manière explicite. La fin implicite est conditionné par la fin de la fonction exécutée par le processus.

La fonction *pthread_join()* permet de bloquer le processus appelant jusqu'à ce que le processus dont l'identifiant est donné en paramètre termine.

4.2.1.2 Gestion des accès concurrents et synchronisation

Pour protéger les structures de données partagées par les différents processus concurrents, la bibliothèque fournit, entre autre, un mécanisme d'exclusion mutuelle (*mutex*). La fonction *pthread_mutex_init()* permet la création d'un verrou. Les fonctions *pthread_mutex_lock(mutex)* et *pthread_mutex_unlock(mutex)* permettent respectivement d'acquérir et de libérer le verrou. Si un processus veut acquérir un verrou qui est déjà pris, son exécution est stoppée jusqu'à ce que le verrou soit libre.

La bibliothèque fournit aussi un moyen de synchroniser les processus concurrents par l'intermédiaire d'une *variable-condition* associée à un *mutex*. Ce couple définit un mécanisme de synchronisation permettant à un processus de suspendre son exécution jusqu'à ce qu'une certaine condition soit vérifiée. Les opérations fondamentales sur les conditions sont :

- signaler que la condition est satisfaite, *pthread_cond_signal(cond)*,
- et attendre la condition en suspendant l'exécution du *thread* jusqu'à ce qu'un autre *thread* signale que la condition est vérifiée, *pthread_cond_wait(cond, mutex)*.

Un processus peut se mettre en attente sur une condition alors qu'il se trouve à l'intérieur d'une section critique. Dans ce cas, le verrou est automatiquement libéré pour éviter les interblocages, et le processus ne reprendra son exécution qu'en acquérant de nouveau le verrou. La fonction *pthread_cond_broadcast(cond)* permet de réveiller tous les processus en attente sur la condition *cond*.

4.2.2 Algorithme distribué en pseudo-code

4.2.2.1 Modèle d'exécution Fork-Join

La création des *pthreads* est réalisée après la phase d'élaboration de la plateforme. La simulation proprement dite de la plateforme est ensuite distribuée sur plusieurs processus selon le modèle *Fork-Join*, comme le montre la figure 4.7. Les blocs `SIMUL` de la figure représentent les processus concurrents. Le processus père se met en attente de la fin de chacun de ses fils, puis exécute la terminaison de l'exécution, en faisant appel aux destructeurs.

Pour notre exécution parallèle, on crée plusieurs *threads* exécutant tous le même code dont l'algorithme est donné dans la figure 2. Cet algorithme est basé sur l'algorithme séquentiel et montre comment est réalisé la répartition des tâches entre les différents processus concurrents. On fait notamment apparaître de manière explicite les accès aux ressources partagées que l'on a identifiées dans l'algorithme séquentiel comme étant les ensembles nécessaires au noyau pour gérer les différentes notifications.

Description des ensembles utilisées dans l'algorithme réparti

Pour la mécanique d'exécution répartie on fait apparaître un nouvel ensemble (R : *pour Running*) qui permet de connaître à tout moment de la simulation les processus en cours d'exécution.

- E, S, T, NI, NR, NT, C : ces ensembles sont identiques à ceux définis pour la description de l'algorithme séquentiel,
- **R** : ensemble des processus en cours d'exécution,

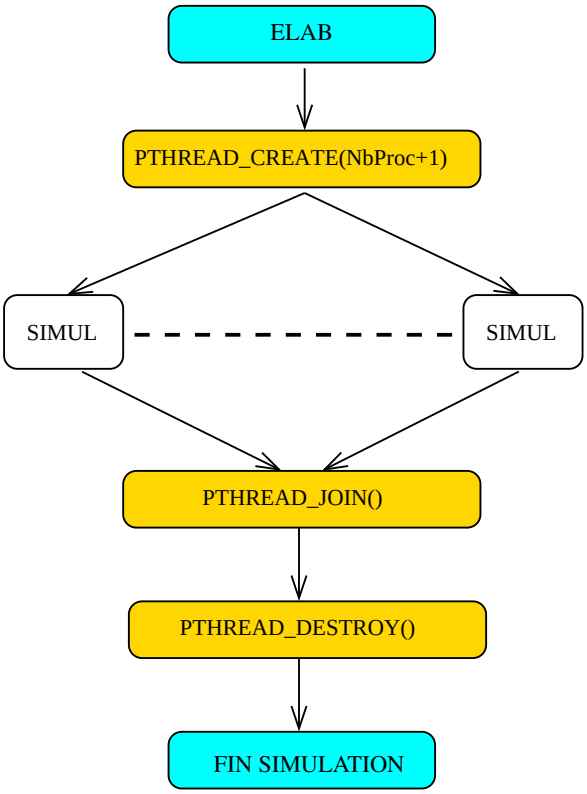

FIG. 4.7 – Modèle « Fork Join ».

– **L** : verrou partagé par les différents *thread* permettant d'accéder en exclusion mutuelle aux ressources partagées, c'est-à-dire les ensembles que l'on vient de définir,
– **V** : *condition variable* permettant de réveiller des processus en attente sur cette condition.

Il est important de noter que seules les phases d'évaluations (notées *execute(p)* dans l'algorithme) des processus sont exécutées en parallèle, et donc qu'il faut aussi protéger les accès aux ressources du noyau de simulation accédées pendant cette phase. Dans l'algorithme on voit qu'un processus manipule des variables temporaires pendant son évaluation et qu'une fois le verrou acquis il modifie les variables communes.

Les principales difficultés de la mise en œuvre de cet algorithme sont liées à la mise en place des verrous. En effet, des sections critiques inutiles ou inutilement longues dégradent les performances, voire même peuvent mener au blocage de la simulation. D'un autre coté, l'absence de verrou lors d'un accès à une des ressources critiques, que l'on a bien identifiées, peut provoquer des comportements incorrects au regard de la spécification, ou même une erreur à l'exécution.

La figure 4.8, donne une vue simplifiée de l'algorithme de la fonction exécutée par chaque fil d'exécution créé. Ce graphe fait apparaître les accès au verrou ainsi que les signalisations de changement de l'ensemble des éligibles ou de l'ensemble des processus en cours d'exécution, qui ont pour effet de réveiller les processus qui attendent de trouver une tâche à exécuter.

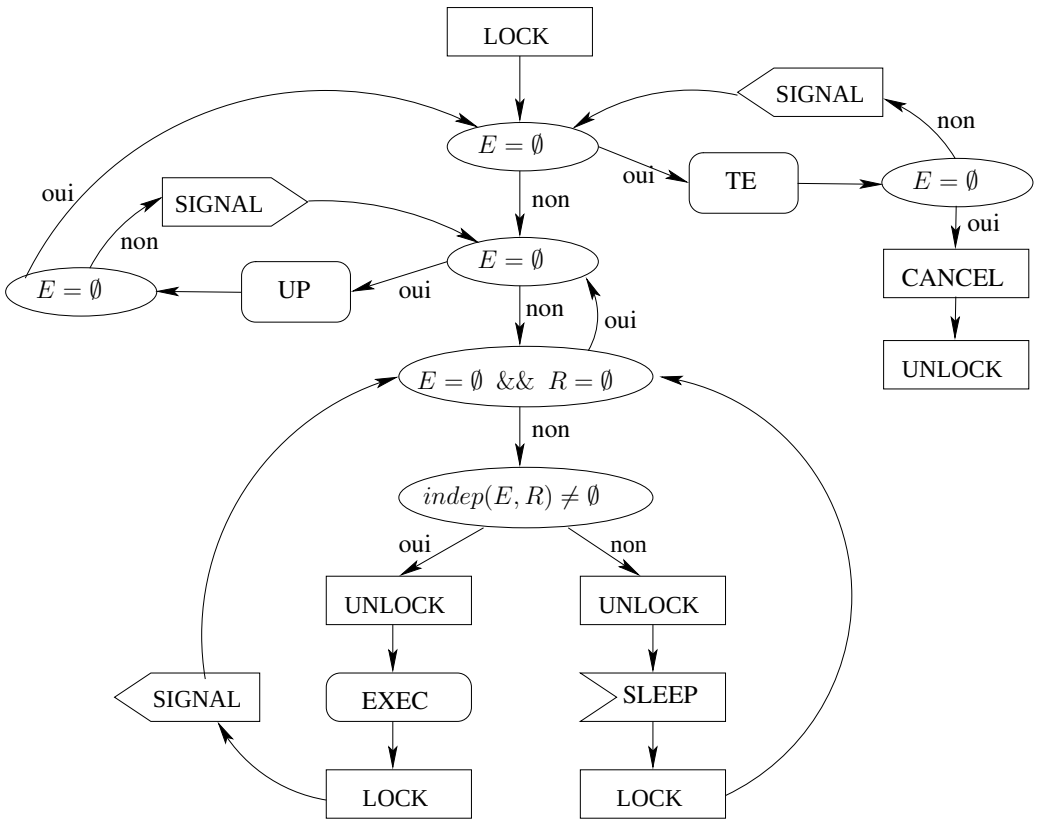

FIG. 4.8 – Fonction distribuée

Algorithme 2 : Algorithme de l'ordonnanceur SystemC

```
lock(L) ;
tant que E ≠ ∅ faire
    tant que E ≠ ∅ faire
        tant que ((E ≠ ∅)  ou  (R ≠ ∅)) faire
            si independant(E, R) = ∅ alors
                { unlock(L) ; sleep(V) ; lock(L) ; }
            sinon
                soit p ∈ independant(E, R) ; R = R ∪ {p} ; E = E − {p} ;
                unlock(L) ; (NI′, NR′, NT′, C′) := execute(p) ; lock(L) ;
                R = R − {p};
                pour chaque e ∈ NI′ faire
                    pour chaque (q, e, d) ∈ S faire
                        E := E ∪ {q} ; S := S − {(q, e, d)} ;
                NR := NR ∪ NR′ ; NT := NT ∪ NT′ ; C := C ∪ C′ ;
                si p s'arrête sur wait(e) alors S := S ∪ {(p, e, +∞)} ;
                si p s'arrête sur wait(e, d) alors S := S ∪ {(p, e, d)} ;
                si p s'arrête sur wait(t) alors T := T ∪ {(p, t)} ;
                wake_up_all(V) ;
        // post condition : un seul processus est actif
        // phase de mise à jour (UP)
        NR′ := execute_updates(C);
        NR := NR ∪ NR′;
        pour chaque e ∈ NR faire
            pour chaque (p, e, d) ∈ S faire
                E := E ∪ {p} ; S := S − {(p, e, d)} ;
        pour chaque x = (p, d) ∈ T/d = 0 faire
            E := E ∪ {p} ; T := T − {x} ;
        si E ≠ ∅ alors wake_up_all(V)
    // Phase d'avancement du temps simulé (TE)
    m = min({t/(p, t) ∈ T} ∪ {t/(p, e, t) ∈ S});
    pour chaque x = (p, d) ∈ T faire
        T := T − {x} ; si m = d alors E := E ∪ {p} sinon T = T ∪ (p, d − m)
    pour chaque x = (e, d) ∈ NT faire
        NT := NT − {x} ;
        si m = d alors
            pour chaque (p, e, h) ∈ S faire
                E := E ∪ {p} ; S := S − {(p, e, d)} ;
        sinon NT = NT ∪ (e, d − m)
    pour chaque x = (p, e, d) ∈ S faire
        S := S − {x} ; si m = d alors E := E ∪ {p} sinon S = S ∪ (p, e, d − m)
    si E ≠ ∅ alors wake_up_all(V) ;
cancel_all() ;
unlock(L) ;
```

4.3 Test de l'algorithme

L'objectif des tests réalisés est de vérifier le bon fonctionnement de l'algorithme de répartition des processus sur les différents fils d'exécutions créés. On veut notamment s'assurer que notre implémentation ne provoque pas d'interblocage entre processus ni de famine. Un ajustement de performance pourra être fait a posteriori pour connaître le nombre optimal de processus à créer. Pour ces tests, on crée $N + 1$ *threads*, avec N désignant le nombre de processeurs disponible sur la machine hôte.

Idéalement, il faudrait que notre ordonnanceur multiprocesseur utilisé sur une machine monoprocesseur soit proche de l'efficacité de l'ordonnanceur monoprocesseur de référence fourni par l'OSCI. Pour cela, il faut limiter les calculs dynamiques, mais sans oublier qu'un processus sans tâche SystemC à exécuter constitue un gaspillage des ressources matérielles.

4.3.1 Mesure

Les tests ont été réalisés sur une machine bi-processeur double cœurs (soit 4 cœurs de processeur) sur un système d'exploitation de type `Debian etch GNU/Linux 2.6`.

Le facteur d'accélération est défini par le rapport du temps d'exécution avec notre implémentation distribuée, en fonction du nombre de processeur i, sur le temps d'exécution avec l'implémentation du noyau de référence de l'OSCI avec son ordonnanceur séquentiel.

$$F_i = \frac{T_i}{T_{seq}}$$

4.3.2 Tests de la mécanique d'exécution

Pour réaliser ces tests, nous avons développé une plateforme paramétrée pour pouvoir jouer sur la longueur des transitions : le temps de calcul entre deux instructions *wait*. Dans ces tests la fonction *independant(E,R)* rend toujours vrai puisque nous n'avons pour objectif que d'éprouver la mécanique d'exécution parallèle mise en place, indépendamment du respect de la sémantique.

Par ailleurs, les résultats obtenus donneront une borne supérieur des facteurs d'accélérations dans la mesure où les surcoûts liés à l'analyse statique et à la prise de décision dynamique sont quasi nuls.

Il est à noté que l'objet du chapitre 5 est précisément d'étudier les moyens à mettre en œuvre pour implémenter cette fonction *independant(E,R)*, pour le niveau de granularité transition. D'autre part, on peut aussi noter qu'il est aisé d'implémenter cette fonction avec pour critère d'indépendance l'appartenance à des modules distincts. C'est-à-dire que deux transitions SystemC sont dites indépendantes si et seulement si elles ne sont pas dans le même module. Ceci ne répond pas à notre objectif de pouvoir traiter des modèles SystemC quelconques, mais réalise un prototype proche de l'approche d'Éric Paire. Celui-ci n'est notamment pas adapter aux modèles transactionnels mais permet la parallélisation des simulations de modèles SystemC dont les canaux de communication implémentent le mécanisme de mise à jour synchrone, comme on l'a vue dans la section 3.2.

4.3.2.1 Exemples

Le code de la figure 4.9 donne un exemple de processus de notre plateforme de test. Ce processus est constitué de plusieurs transitions, simulant des traitements plus ou moins long selon la valeur du paramètre `SIZE_TRANS`. L'objectif est d'utiliser toutes les instructions d'attentes et toutes les notifications pour observer que tous ces mécanismes sont bien protégés. On veut aussi observer qu'il

n'y pas d'interblocage dans notre implémentation et que tous les *threads* créés se répartissent les processus SystemC éligibles.

```
void process_B() {
  unsigned i = 0;
  while( i++ < 4 * SIZE_TRANS ){
    if( ( i % SIZE_TRANS ) == 0 )
      cout << "B reaches  " << i << endl;
  }
  cout << "B wait f" << endl;
  wait(f);
  cout << "B notify e" << endl;
  notify(e);
  while( i++ < 6 * SIZE_TRANS ){
    if((i%SIZE_TRANS)==0)
      cout << "B reaches  " << i << endl;
  }
  cout << "B wait 1 ns" << endl;
  wait(1, SC_NS);
  while( i++ < 8 * SIZE_TRANS ){
    if((i%SIZE_TRANS)==0)
      cout << "B reaches  " << i << endl;
  }
  notify(e);
  cout << "B ends" << endl;
}
```

FIG. 4.9 – Exemple de processus SystemC.

4.3.2.2 Résultats

Nous avons réalisé les tests sur trois plateformes P1, P2 et P3. P1 et P2 sont identiques sauf qu'on a multiplié par 10 la longueur des transitions. L'écart obtenus entre P1 et P2 montre l'influence de la longueur des sections atomiques SystemC sur l'efficacité. Dans P3 on a doublé le nombre de processus en dupliquant les processus de P2.

Le tableau 4.10 donne les temps de simulations obtenu sur les différentes plateformes avec les deux implantations du noyau.

plateform	T_{seq}	T_4	F_4
P1	12,55	04,83	2,59
P2	16,54	06,53	2,53
P3	32,55	12,75	2,55

FIG. 4.10 – Résultats des temps de simulation

Ces tests nous ont montré des accélérations de l'ordre de 2,5 par rapport aux exécutions séquentielles avec le moteur de simulation de l'OSCI. Ce résultat est tout à fait satisfaisant puisque

dans certains cycles d'exécution un seul processus était éligible. Les répartitions des tâches SystemC sur les *threads* ont pu être observées avec les affichages sur la sortie standard.

Chapitre 5

Vers une analyse statique des dépendances

Sommaire

Comme on l'a vu précédemment, la parallélisation à sémantique constante des modèles SystemC nécessite une analyse fine des dépendances entre les tâches. On propose de voir comment réaliser ces analyses sur le code pour pouvoir disposer de ces informations à l'exécution, au moment de choisir des transitions à exécuter en parallèle.

Dans ce chapitre, on va présenter les problèmes liés aux analyses des dépendances. On montrera ensuite comment pourrait être architecturé une chaîne d'outils pour la parallélisation automatique des simulations. On termine en donnant les éléments essentiels pour la réalisation de l'analyse des dépendances.

5.1 Transitions indépendantes et parallélisation

5.1.1 Rappels

Une transition est définie comme la séquence des actions exécutées depuis l'élection d'un processus par l'ordonnanceur jusqu'au retour du contrôle au noyau. Pour les processus de type `SC_THREAD` ou `SC_CTHREAD`, les débuts et fins des transitions sont identifiées par les appels de fonctions *wait()*.

Pour exécuter deux transitions simultanément en restant conforme à la spécification du langage, on doit s'assurer de leur indépendance (voir définition 3.3.2.1). Deux transitions sont parallélisables si et seulement si les ensembles de variables auxquelles elles accèdent sont disjoints : $\mathcal{V}(t) \cap \mathcal{V}(t') = \emptyset \Leftrightarrow t$ et t' sont parallélisables.

Prenons un exemple où plusieurs transitions sont en cours d'exécution sur des processeurs différents. On souhaite choisir une nouvelle transition pour s'exécuter sur un processeur libre. Pour cela, on doit être capable de déterminer les variables qui vont être accédées par la nouvelle transition pour s'assurer de son indépendance avec celles en cours d'exécution.

5.1.2 Énoncé du problème

La définition de transition que l'on a donné au chapitre 3 est une notion purement dynamique, ce qui rend difficile voire impossible de prévoir exactement les variables accédées. En effet, une transition dépend :

- du point de contrôle de départ. Statiquement, on sait repérer les débuts des transitions qui sont identifiés soit par une instruction *wait()* soit par le début d'un processus.
- des entrées du programme. Celles-ci ne sont pas prévisibles : si un branchement dépend de ces entrées, cela rend impossible l'analyse des variables accédées par la transition.
- de l'état de la mémoire interne. L'évolution de la mémoire interne est plus prévisible. Il faut connaître les valeurs initiales des variables et interpréter le code pour calculer le prochain état de la mémoire. Ceci est trop coûteux.
- et de la pile d'exécution. C'est un cas particulier de la mémoire interne, plus facile à manipuler, parce que bien structurée.

5.1.3 Illustration du problème sur un exemple

Considérons le processus SystemC dont le code est donné dans la figure 5.1. Il réalise une boucle dans laquelle il se met en attente de la notification d'un événement, effectue des opérations sur des variables et réalise un appel à une fonction $f()$ dont le code est lui aussi fourni.

```
sc_event e;
int x, y, z, t, u;
int * T;
void run(){
  do {
    wait(e);
    x++;
    y = y - T[x];
    f();
    y = T[0];
  }
  while(x<100);
}
```

```
void f(){
  if(z >= 2) {
    x = 0;
    g();
    wait();
  }
  else {
    wait();
    y = 0;
    g()
  }
  return;
}
```

```
void g(){
  t = 0;
  wait();
  u++;
  return;
}
```

FIG. 5.1 – Exemple de programme SystemC avec un processus et deux fonctions

La représentation du graphe de flot de contrôle correspondant au processus est fournie dans la figure 5.2.

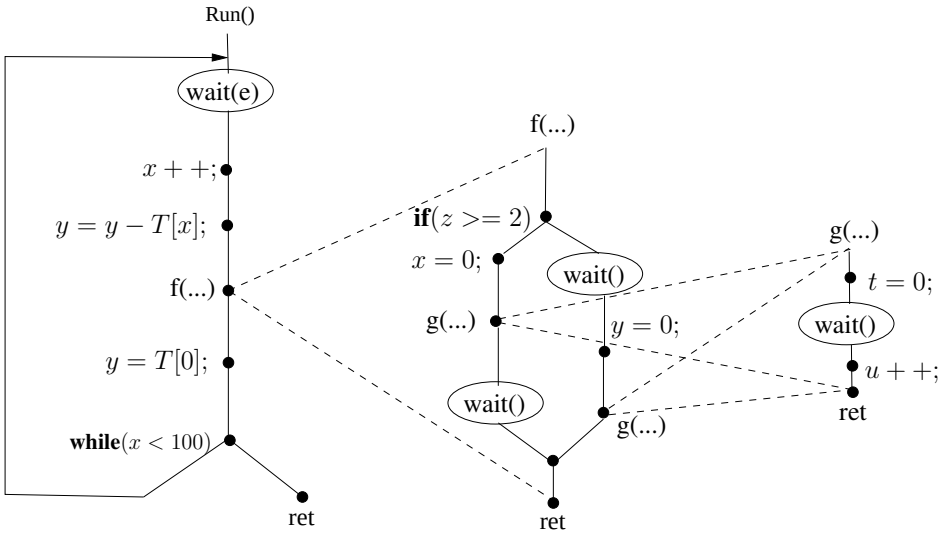

FIG. 5.2 – Graphe de flot de contrôle

5.1.3.1 Transition exacte

Si on analyse le code du processus, on voit qu'à sa première élection, le processus va exécuter directement une instruction d'attente *wait(e)*, qui marque la fin de la transition. La seule ressource qui va être touchée est `e`. Dans ce cas, la transition est parfaitement identifiée, et on peut associer au point de contrôle correspondant au début du processus, la ressource `e`.

5.1.3.2 Sur-approximation conservative

On poursuit notre analyse en avant sur le code depuis le point de contrôle identifié par l'instruction `wait(e)`. On voit que l'on peut accéder aux variables `{x, y, T}` puis on atteint un appel de fonction. On peut poursuivre l'analyse en avant dans le corps de la fonction `f()`. On arrive sur une instruction conditionnelle. Statiquement, on ne sait pas dire quelle branche sera prise à l'exécution. Une solution est de faire une analyse approchée conservative en confondant les deux transitions potentielles. Pour cela, on fait l'union des objets accessibles par les deux transitions. Sur la branche de droite on atteint une instruction `wait` qui met fin à l'analyse. La branche de gauche touche la variable `x`, puis appelle la fonction `g()`. Comme précédemment, on poursuit l'analyse dans le corps de la fonction appelée. On voit que l'on peut accéder à la variable `t` avant d'atteindre une instruction `wait`.

Pour résumer, depuis le point de contrôle identifié par l'instruction `wait(e)`, on a pu identifier toutes les variables potentiellement accessibles depuis ce point. Pour cela, on a du faire une *sur-approximation conservative* en considérant plusieurs transitions. On peut donc associer statiquement, à ce point de contrôle les ressources `{x,y,T,z,t}`.

5.1.3.3 Retour d'appel de fonction

Considérons maintenant la poursuite de notre analyse en avant depuis le point de reprise potentiel du processus, identifié par l'instruction `wait()`, dans le corps de `g()`.

On voit qu'on peut toucher la variable `u`, puis la fonction retourne. Le problème est qu'on ne sait pas statiquement d'où la fonction a été appelée.

- Une première solution à ce problème consiste à faire l'union des points de retour possibles. Cette solution semble trop grossière pour pouvoir être efficace.
- Une autre solution à ce problème consiste à expanser les appels de fonctions, c'est-à-dire remplacer chaque appel de fonction par le corps de la fonction correspondante. En pratique, cette solution n'est pas envisageable puisque elle entraîne une explosion de la taille du code.
- Enfin, une troisième solution consiste à arrêter l'analyse sur le retour de fonction. On pourra utiliser la pile d'appel, disponible à l'exécution, pour connaître le point de retour d'un appel particulier, et ainsi compléter les variables accessibles. Pour cela, il faut qu'on associe à chaque retour d'appel de fonction les objets accessibles depuis ce point. Cette solution a l'avantage d'être beaucoup plus précise que les deux premières mais elle entraîne un surcoût à l'exécution. Cependant, le surcoût reste raisonnable puisqu'il suffit de faire des unions d'ensemble. Sur le graphe de flot de contrôle de notre exemple, supposons que la fonction `g()` retourne dans la branche de gauche. Dynamiquement on fait l'union des ensembles de variables attachées au point de contrôle de la fonction `g()` et au retour d'appel. On peut donc identifier dynamiquement l'ensemble des ressources accessibles pour l'exécution de cette transition, sur cet exemple l'ensemble se restreint à `{t}`.

5.1.3.4 Synthèse de l'exemple

La figure 5.3 est le flot de contrôle précédent dans lequel on a identifié les différents points de contrôle : notés `w0`, `w1`, `w2`, `w3` et `w4`. On a aussi distingué les différents appels de fonction : notés `f1`, `g1` et `g2`.

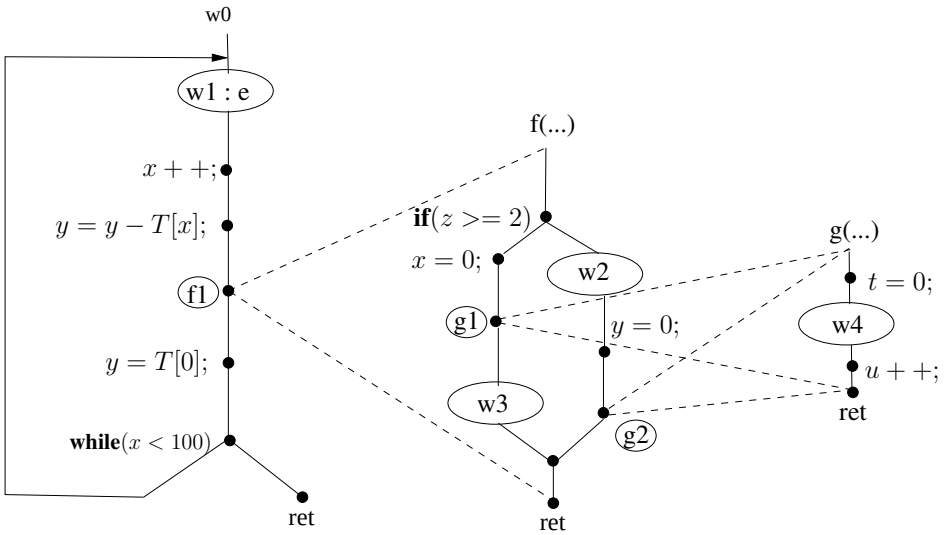

FIG. 5.3 – Graphe de flot de contrôle

Par une analyse statique en avant sur le graphe on obtient le tableau 5.4 des ressources accessibles

depuis chacun des points de reprise.

Id	Ressources accessibles	Atteint une instruction return ?
w0	{ e }	non
w1	{ x y T z t }	non
w2	{ x t }	non
w3	∅	oui
w4	{ u }	oui

FIG. 5.4 – Ensembles des ressources accessibles depuis chaque point de reprise du processus Run

Comme nous l'avons dit précédemment, nous arrêtons l'analyse lorsqu'on atteint une instruction *return*. Les transitions sont donc incomplètes dans ce cas là. Statiquement, on réalise la même analyse sur le code à partir de chaque retour d'appel. On rassemble donc toutes les ressources accessibles depuis chaque retour d'appel de fonction. Sur le graphe de flot de contrôle, nous avons identifié les retours d'appel notés `f1`, `g1` et `g2`. L'analyse en avant nous donne donc le tableau 5.5.

Id	Ressources accessibles	Atteint une instruction return ?
f1	{ y T x e }	oui
g1	∅	non
g2	∅	oui

FIG. 5.5 – Ensembles des ressources accessibles depuis chaque retour d'appel de fonction

À l'exécution, on va devoir reconstruire les transitions incomplètes. Pour cela on va exploiter les informations récoltées statiquement (les figures 5.5 et 5.4), plus celles dont on dispose dynamiquement : essentiellement le point de reprise courant du processus et la pile d'appel.

Prenons l'exemple suivant :

- le processus `Run` est éligible,
- il va reprendre son exécution au point de contrôle noté `w4` sur le graphe 5.3,
- la pile d'appel qui a mené au point de contrôle `w4` est celle représentée dans la figure 5.6 (ce qui peut être le cas si l'exécution précédente du processus est partie du point noté `w2`).

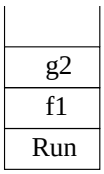

FIG. 5.6 – Représentation de la pile d'exécution

On veut savoir ce que le processus va toucher depuis ce point `w4`, pour pouvoir décider de son indépendance.

Pour cela, on commence par rechercher les informations récoltées sur le point de départ `w4`. On voit que l'ensemble des variables accédées est {`u`} et on peut atteindre un retour. Comme on peut retourner, on consulte la pile d'exécution pour trouver ce point de retour. On voit qu'on vient de l'appel noté `g2`. On consulte la table correspondante pour trouver l'ensemble des variables accédées (vide ici), et savoir si on retourne, ce qui est le cas. On consulte à nouveau la pile et on voit qu'on

vient de `f1`. On poursuit le même raisonnement en recherchant les ressources accessibles depuis `f1` (ici l'ensemble {y, T, x, e}) et si on retourne, ce qui est le cas. On consulte à nouveau la pile et on voit qu'on vient de l'appel principal, ce qui termine la procédure. Au final, on fait l'union des ressources récoltées, ce qui donne : {u, y, T, x, e}.

On va maintenant s'intéresser à spécifier comment :

– on pourrait techniquement mettre en œuvre ces analyses ;
– automatiser les simulations parallèles, c'est-à-dire, sans modification ou annotation du modèle de la part du programmeur.

5.2 Notre solution pour une parallélisation automatique

Pour savoir si une transition est indépendante de celles qui sont en cours d'exécution, on doit être capable dynamiquement de savoir quelles variables vont être touchées par cette dernière. On va pour cela utiliser des informations que l'on aura calculer statiquement. Dans cette section on montre comment on peut automatiser les simulations parallèles de manière transparente pour l'utilisateur (figure 5.7).

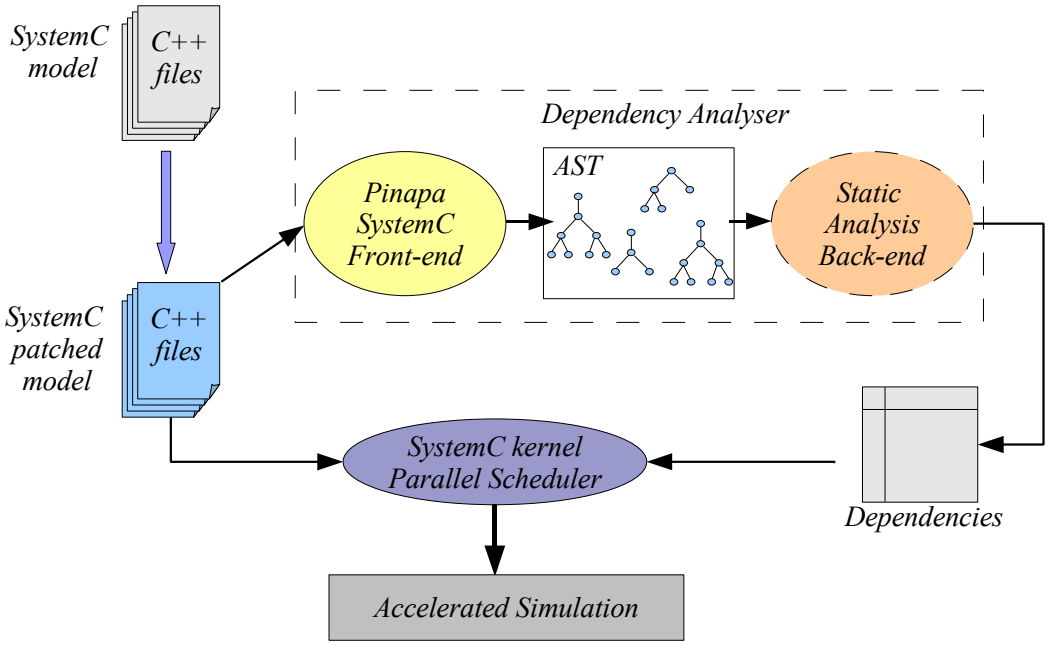

FIG. 5.7 – Chaîne d'outils pour la parallélisation automatique

Le modèle doit d'abord être instrumenté pour repérer chaque point de reprise des processus, correspondant à chaque début de transition possible. Ensuite, on doit réaliser sur le programme modifié l'analyse d'accessibilité des ressources partagées. Cette analyse correspond à la branche supérieure de la figure 5.7. Elle consiste à construire une table associant à chaque point de reprise des processus, les objets accessibles. La branche inférieure montre que l'exécutable parallélisé peut être obtenu en com-

pilant la plateforme instrumentée et ses informations de dépendances avec la bibliothèque SystemC modifiée pour la parallélisation. Dans la suite on développe ces différentes étapes.

5.2.1 Instrumentation du modèle

L'objectif de cette étape est de pouvoir distinguer les différents points de reprise de chaque processus. La bibliothèque SystemC de l'OSCI permet de connaître la liste des processus éligibles. Au grain de parallélisation qui nous intéresse, cette information n'est plus suffisante. En effet, il nous faut en plus identifier le point de contrôle dans le code du processus à partir duquel le processus va reprendre son exécution, c'est-à-dire quelles transitions sont susceptibles d'être exécutées s'il est choisi, et ce pendant l'exécution.

Sur la plateforme à simuler, ces modifications se font de manière purement syntaxique comme le montre le code 5.8. Il suffit de remplacer chacun des appels à la fonctions *wait* par un appel à notre implémentation *mpWait*, possédant un paramètre supplémentaire de type entier, identifiant de manière unique ce point de contrôle.

```
wait(1, SC_NS);
wait(50, SC_NS);
wait(SC_ZERO_TIME);
wait(e);
...
```

```
mpWait(0, 1, SC_NS);
mpWait(1, 50, SC_NS);
mpWait(2, SC_ZERO_TIME);
mpWait(3, e);
...
```

FIG. 5.8 – Transformation syntaxique des instructions *wait* en instruction spécifique unique (notées *mpWait*).

Le but de cette transformation est d'attribuer un entier unique à chacun des points de reprise du modèle. Le nombre total de points de reprise est égal à $nb\_proc + nb\_wait$:

- avec nb_proc le nombre de processus dans le modèle,
- et nb_wait le nombre d'instruction *wait* dans le code.

Naturellement, il faut adapter la librairie SystemC pour prendre en compte cette nouvelle instruction *mpWait()*.

5.2.2 Modifications de la bibliothèque SystemC

L'implémentation de référence OSCI du noyau de simulation séquentiel sur laquelle nous nous basons ne nécessite que la liste des identifiants des processus prêts à s'exécuter. Pour notre application, nous devons être capable de discriminer deux *wait()* distincts à l'intérieur d'un même processus pour pouvoir associer les variables accessibles depuis chacun de ces points de reprise.

Notre solution consiste donc à modifier l'implantation des différents *wait* par notre implémentation que l'on appelle *mpWait*. Notre fonction *mpWait()* prend un paramètre supplémentaire, de type entier, identifiant de manière unique le point de contrôle correspondant. Cet identifiant est attribué lors de la modifiation syntaxique dont on a parlé à la section 5.2.1.

La figure 5.9 montre les modifications apportées à la bibliothèque standard. On donne en exemple la modification de la signature de la fonction d'attente de changement de temps de simulation.

Le traitement associé à l'exécution d'une instruction *mpWait()* est le même que pour le *wait()* correspondant. Il suffit de mettre à jour un attribut des processus qui identifie le prochain point de

```
extern
void
wait( const sc_time&,
sc_simcontext* = sc_get_curr_simcontext() );

extern
void
mpWait( int transID,
const sc_time&,
sc_simcontext* = sc_get_curr_simcontext() );
```

FIG. 5.9 – Modification de la bibliothèque SystemC pour identifier les débuts de transitions.

reprise. Quand le processus sera de nouveau éligible, on connaîtra parfaitement le début de transition qu'il s'apprête à exécuter, et par conséquent les ressources potentiellement accessibles.

5.3 Analyse d'accessibilité : réalisation

Dans cette section, on donne les éléments techniques essentiels pour pouvoir réaliser l'analyse d'accessibilité des variables en se basant sur les arbres abstraits du programmes.

5.3.1 Le frontal SystemC : Pinapa

Pinapa est un frontal SystemC, développé par Matthieu Moy pendant sa thèse [SCI05, MMMC05]. Il repose sur `gcc` pour analyser le code `C++`, et sur la bibliothèque SystemC pour extraire l'architecture de la plateforme.

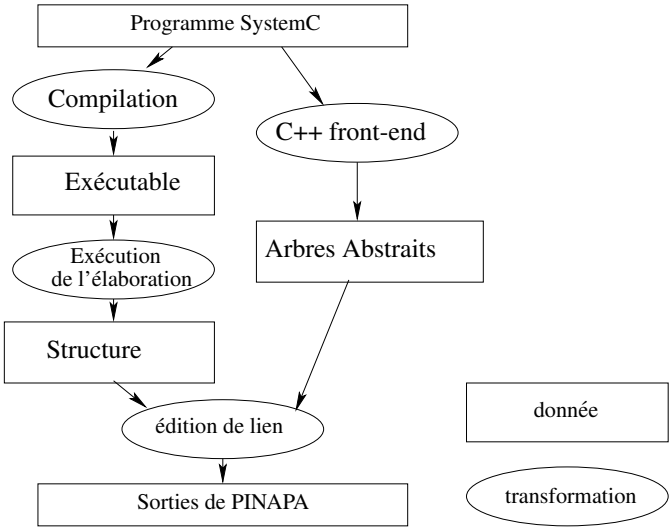

FIG. 5.10 – Principe de Pinapa

Pinapa est en réalité plus qu'un *front-end* `C++` classique puisque pour récupérer les informations d'architecture, Pinapa exécute la phase d'élaboration du modèle en utilisant la bibliothèque SystemC.

La principale tâche de Pinapa est alors de faire le lien entre les arbres abstraits fournis par `gcc` et les structures de données créées pendant la phase d'élaboration. La figure 5.10 présente le principe général.

D'autre part Pinapa définit un ensemble de primitives pour le parcours des arbres abstraits, notamment une interface *visiteur*.

5.3.2 Manipulation des arbres abstraits

L'analyse d'accessibilité des ressources partagées se fait sur le code source SystemC. Ce code est accessible grâce à Pinapa qui nous fournit les arbres des processus (fonctions `c++`). Ceux-ci sont stockés dans une table (`sc_process_table`) de la classe `pinapa::simcontext`.

Pinapa nous fournit aussi un ensemble de primitives pour parcourir l'ensemble des processus (la classe `Cursor`) et permet de récupérer les arbres abstraits associés (classe `tree` de `gcc`).

Le code suivant montre le parcours de tous les processus et la création d'un visiteur réalisant l'analyse d'accessibilité pour chaque arbre abstrait :

```
for(pinapa::simcontext::Cursor cursor =
    pinapa::simcontext::get_first(pinapa::simcontext::THREAD);
    cursor != pinapa::simcontext::Cursor::undefined;
    cursor =  pinapa::simcontext::get_next_sibling())
    {
      sc_process_b * current = cursor.m_process;
      tree t = pinapa::st_process_deco::get(current)->get_gcc_body();
      fillDepVisitor v(current);
      current.accept(DECL_SAVED_TREE(t));
    }
```

5.3.3 L'interface treeVisitor

La classe *treeVisitor* implémente une interface générique pour le parcours des arbres abstraits de `gcc`. Chaque type de nœud possède une fonction de traitement *visitNode()*, qui consiste à répercuter l'appel de la fonction de traitement sur ses fils.

On donne en exemple la fonction `visitIfStmt` qui réalise le traitement pour un nœud de type `IfThenElse` ; on appelle `visitNode` sur la condition, la branche `then` puis la branche `else`.

```
int treeVisitor::visitIfStmt(const tree node) {
  (void) visitNode(IF_COND(node));
  (void) visitNode(THEN_CLAUSE(node));
  (void) visitNode(ELSE_CLAUSE(node));
  return m_result;
}
```

Nous avons développé un visiteur qui hérite de l'interface `treeVisitor` fournie par `Pinapa`. Nous n'avons pas implémenté tous les traitements que nous avons spécifiés pour l'analyse statique.

```
class fillDepVisitor : public pinapa::treeVisitor
{
  sc_process_b * m_process;

  public:
  // constructor
  fillDepVisitor(sc_process_b * h) :
  m_process(h)  {   }

  // destructor
  virtual ~fillDepVisitor() {   }

  // ... définition des fonctions ''visitNode'' réalisant l'analyse
}
```

5.4 Conclusion

Nous avons donné dans ce chapitre, tous les ingrédients nécessaires pour la réalisation des analyses des dépendances et comment doit être architecturé un outil automatique pour la parallélisation. Après avoir spécifié tous ces besoins, on s'interroge sur l'efficacité de la parallélisation. En effet, pour être efficace il faut que nos analyses approchées soit suffisamment fines (donc potentiellement couteûses), pour pouvoir déceler des transitions indépendantes. Vu le travail encore nécessaire dans l'analyse statique, il nous paraît intéressant d'arriver d'abord à estimer l'accélération que nous pourrions obtenir sur des études de cas réelles, avant d'envisager l'implémentation d'un prototype complet. C'est précisément ce dont nous allons discuté dans la chapitre 6.

Chapitre 6

Évaluations

Sommaire

L'objectif de ce chapitre est d'évaluer les accélérations possibles sur les modèles SystemC. Pour réaliser les tests nous avons utilisé une machine bi-processeurs, double cœurs (soit 4 cœurs de processeur) avec un système d'exploitation de type `GNU/Linux 2.6 Debian etch`. Le nombre de *threads OS* à créer correspond au nombre de processeurs plus un. Ici, nous créons donc 5 *Pthread* pour exécuter les simulations en parallèle.

```
  SystemC 2.1.v1 --- Sep 25 2007 09:05:36
With 5 POSIX Threads mapped on 4 processors
Copyright (c) 1996-2005 by all Contributors
         ALL RIGHTS RESERVED
```

6.1 Premières expérimentations

6.1.1 Exemples de la librairie SystemC

La librairie SystemC est livrée avec plusieurs exemples de programmes. Ces exemples ne sont pas décrits au niveau transactionnel mais permettent d'éprouver la mécanique d'exécution. Nous avons sélectionné parmi ces exemples ceux qui utilisaient des canaux à mise à jour synchrone, comme les canaux primitifs `sc_signal` ou `sc_fifo`. Le critère d'indépendance est alors l'appartenance à des modules distincts (voir section 3.2.1).

Nous n'avons pas simulé les exemples qui utilisent des *clocked thread* (classe `sc_cthread`) car nous n'avons pas implémenté dans le noyau parallèle le traitement de ce type de processus :

– d'une part parce qu'ils ne sont pas pertinent pour le niveau TLM,

– et d'autre part, ce type de processus tend à devenir obsolète.

Ces tests n'ont pas permis d'avoir une estimation de l'accélération :

– soit parce que le modèle était trop petit,
– soit le modèle était trop séquentiel avec un seul processus éligible à chaque δ-cycle.

Par contre, ils ont permis de tester la mécanique d'exécution et dans certains cas de soulever des bogues dans l'implémentation.

6.1.2 Exemples développés « à la main »

Pour pouvoir caractériser les modèles qui se parallélisent bien, nous avons développé nos propres plateformes de tests. Dans les différentes plateformes nous avons joué sur deux paramètres :

– le nombre de processus SystemC de la plateforme,
– et la longueur des transitions, c'est-à-dire le temps de calcul entre l'élection d'un processus jusqu'au retour du contrôle au noyau.

L'objectif est de pouvoir observer l'influence de ces paramètres sur les performances.

Une des plateformes que nous avons développée est décrite dans la section 4.3. Ces tests ont montrés que plus il y a de processus éligibles et plus les transitions sont longues, plus le gain en temps est grand. Par contre, ces tests ne sont pas du tout réaliste. Pour pouvoir quantifier les accélérations potentielles sur des modèles réalistes, nous avons développé notre propre étude de cas TLM.

6.2 Étude de cas : calcul réparti d'une *fractale*

Notre choix de développer une plateforme a plusieurs motivations :

– L'évolution des versions de SystemC et TAC a fait que nous ne pouvions pas réutiliser, sans modifications profondes, les modèles développés par STMicroelectronics pour tester l'outil d'Eric Paire.
– D'autres études de cas, issues aussi de STMicroelectronics auraient pu être utilisées. Mais les tests réalisés dans les travaux de Claude Helmstetter [Hel07], ont dévoilé que les modèles analysés étaient intrinsèquement séquentiels : dans la majeure partie des phases d'évaluations, un seul processus était éligible. Sur la plateforme STi7100, modèle d'une *set-top box* (appareil de décodage pour la télévision Haute Définition), 97% des transitions sont seules dans leur δ-cycle.
– Par ailleurs, nous souhaitions aussi avoir une pleine maîtrise du code pour pouvoir le modifier et l'adapter à nos besoins.

Dans la suite, nous présentons notre étude. On commence par une description globale du modèle. Puis nous donnons une description détaillée du comportement des différents composants et des schémas de synchronisation. Enfin nous donnons certains résultats intéressants.

6.2.1 Architecture de l'étude de cas

Le comportement global de la plateforme consiste à calculer la *fractale de Mandelbrot* [Wik07] et d'afficher son résultat sous forme d'image (voir la figure 6.1). On fixe la taille de l'image à 640 x 480 pixels. Les coordonnées de chaque pixel de l'image sont des paramètres de la fonction de calcul de la *fractale*. Le résultat est ensuite converti en couleur (un entier de 0 à 255). Le but est de faire calculer une tranche d'image à plusieurs composants, que nous appelons *fractale*.

La figure 6.2 décrit l'architecture de notre étude de cas. Elle est constituée des composants suivants :

– un générateur, qui programme et active les différents composants,

– un contrôleur d'écran (LCDC), pour l'affichage de l'image,
– une mémoire (RAM) globale pour stocker les résultats,
– un bus (routeur) pour aiguiller les transactions vers les composants cibles,
– N instances du composant *fractale*.

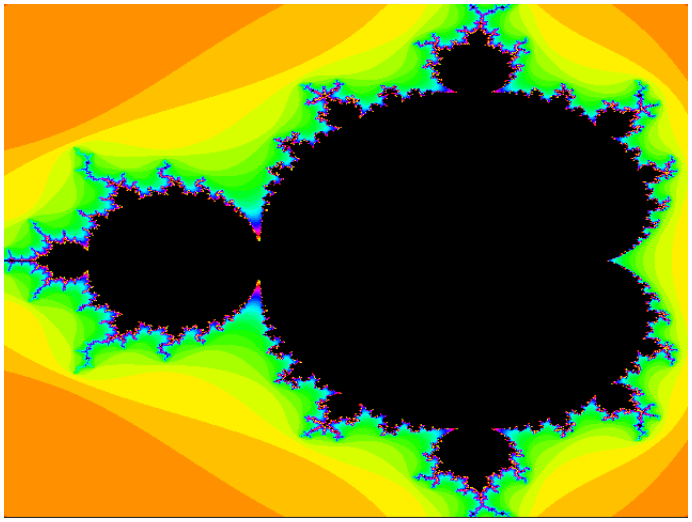

FIG. 6.1 – Image de la *fractale de Mandelbrot*

Ce modèle a plusieurs avantages pour notre objectif :
– on peut découper l'image en tranches, que l'on peut calculer en parallèle,
– le nombre de modules *fractale* est paramétrable, ce qui offre un degré de liberté pour un calcul « fortement » réparti,
– la définition de l'image est paramétrable ce qui permet de faire varier le temps de calcul de chaque pixel,
– les schémas de synchronisation utilisés et les temps de calcul sont réalistes.

Le composant mémoire est une instance de la classe `tac_memory` et le routeur une instance de la classe `tac_routeur`. Ce sont des composants standards de la bibliothèque TAC [tac05]. Un composant *fractale* effectue le calcul d'une portion de la *fractale de Mandelbrot*, puis stocke le résultat en mémoire. Le composant LCDC effectue une lecture en mémoire et effectue l'affichage de l'image de la fractale dans une fenêtre `X11`.

6.2.2 Construction de la plateforme

La construction de la plateforme est paramétrée par le nombre de module *fractale* (NB_FRACTAL_MODULE).

Le code de la figure 6.3 montre la boucle d'instanciation des modules. Les ports initiateurs sont alors connectés aux ports cibles correspondants.

Chaque composant *fractale* possède une mémoire interne (des registres). Les registres associés a un composant *fractale* sont identifiés par un décalage constant (offset) depuis une adresse de base (voir le code 6.4). On définit et utilise :
– un registre pour son identifiant.
– un registre pour stocker l'adresse en mémoire ou le composant va écrire son résultat.
– un registre `start` qui permet de lancer le calcul du morceau de fractale

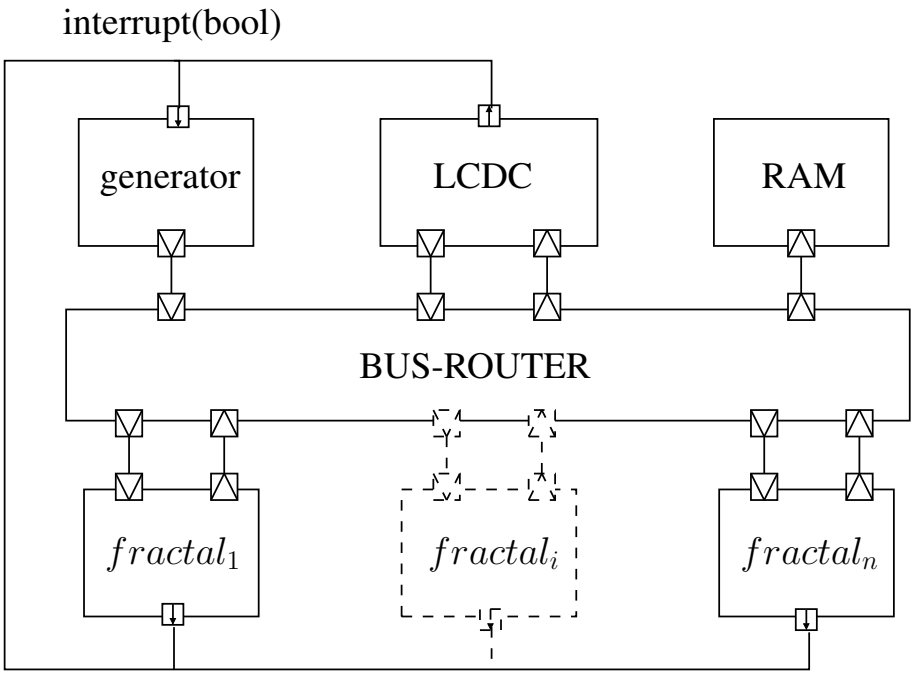

FIG. 6.2 – Architecture de l'étude de cas « Fractale ».

```
for(int i = 0;i < NB_FRACTAL_MODULE;i++) {
  char buffer[12]; // limited to 999 FRACTAL component
  sprintf(buffer,"FRACTAL%d",i);
  fractal_pv * fractal = new fractal_pv(buffer);
  fractal_list.push_back(fractal);
  // router -> fractals
  ROUTER.initiator_port((*fractal)->target_port);
  // fractals -> router
  (*fractal)->initiator_port(ROUTER.target_port);
  // fractals -> generator (interrupt port)
  (*fractal)->interrupt_out(*GENERATOR.interrupt_in_list[i]);
}
```

FIG. 6.3 – Boucle d'instanciation et de connection paramétrée.

– deux registres pour les coordonnées X et Y du pixel à calculer.

La programmation d'un composant consiste à donner des valeurs à ses registres.

```
#define FRACTAL_BASE_ADDR_REG 0x00000000
#define FRACTAL_START_REG     0x00000004
#define FRACTAL_X1_OFFSET_REG 0x00000008
#define FRACTAL_Y1_OFFSET_REG 0x0000000c
#define FRACTAL_ID_REG        0x00000010
```

FIG. 6.4 – Initialisation des décalages des registres.

D'autre part, il faut générer la carte des adresses mémoire (`memory_map`) nécessaire au routeur. On associe à chaque port cible des composants *fractale* :

– l'adresse de base du composant*fractale*,

– la taille de la mémoire interne allouée pour ses registres.

Cela est réalisé en concaténant au fichier contenant les plages correspondant aux composants LCDC et RAM, la liste des plages pour les N composants *fractale* (en conservant la même convention de nommage que dans l'appel au constructeur des *fractale*).

```
for(int i = 0; i < NB_FRACTAL_MODULE; i++)
  {
    file << "TOP.FRACTAL"<< dec << i <<".target_port          "
         << "0x" << hex << 0x20000000 + i * 0x00000100
         << "                    0x00000100 "<< endl;
  }
```

FIG. 6.5 – Génération de la carte des plages mémoires.

6.2.3 Description des schémas de synchronisation

On rappelle que les événements SystemC ne sont pas persistants. Quand un événement est notifié, seuls les processus actuellement en attente sur lui sont concernés. Un processus ne peut pas savoir directement si un événement a été notifié dans le passé ou non. Pour simuler un *événement persistant*, les programmeurs utilisent une variable partagée associée à l'événement pour mémoriser une notification éventuelle. Ils utilisent pour la notification et l'attente le schéma donné par la figure 6.6.

Notification	Attente
`e.notify();` `x = true;`	`if (!x) { wait(e); }` `x = false; //reset`

FIG. 6.6 – Modélisation d'un événement persistant avec un couple booléen / événement (`x`, `e`).

Le générateur commence par programmer les registres des N composants *fractale*. Ceci est fait par une succession de transactions d'écriture (`write`). Le générateur se met ensuite en attente d'une interruption d'un des composants *fractale*.

Une fois programmés, les processus des composants *fractale* deviennent éligibles. Dans le code 6.7, on ne montre que la fin du traitement, quand l'ensemble des pixels ont été calculés. Le processus

initie une transaction de type `write_block` (voir [tac05]) pour stocker le résultat en mémoire. Le processus déclenche ensuite une interruption, modélisée par le protocole `tlm_synchro`, fourni avec le protocole TAC [tac05]. Cette interruption est envoyée au générateur pour lui signifier la fin du traitement du composant.

```
// writing
status = initiator_port.write_block(
              MEMORY_BASE_ADDRESS + id_register*stepx*stepy,
              buffer,
              stepx*stepy/4,
              error_reason
                                   );

// write ok
interrupt_out.sync(true);
```

FIG. 6.7 – Écriture en mémoire et synchronisation de fin de traitement.

Le traitement de l'interruption a pour effet de rendre éligible le générateur. Le générateur programme le registre `start` du LCDC pour le rafraîchissement de l'image. Le générateur attend une interruption du LCDC et tourne en boucle jusqu'à ce qu'il ait reçu les interruptions de tous les composants *fractale*. On donne dans la figure 6.8, un apperçu de l'affichage pendant la simulation.

6.3 Résultats

Nous avons développé cette plateforme dans le but d'exhiber des comportements parallèles indépendants, d'où le choix du calcul de la *fractale de Mandelbrot*. Pourtant, le développement de cette étude a soulevé plusieurs problèmes pouvant dégrader les performances, voire mener à des impossibilités de parallélisation (voir la section 7.1.4.1 dans la conclusion) :

- Le premier problème que montre cette étude est le routage des transactions. Dès qu'une transition accède au bus, elle peut potentiellement accéder à tous les composants cibles implémentant le protocole.
- Le second problème concerne la mémoire centrale (RAM). Toutes les transitions qui effectuent une transaction vers ce composant se retrouvent potentiellement dépendantes. Le seul moyen de dire qu'elles sont indépendantes est d'être sûr qu'elles n'accèdent pas à ma même zone de la mémoire.
- Si l'on veut explorer différents comportements réalistes du matériel, on doit utiliser l'instruction d'attente sur du temps imprécis, notée `pv_wait(sc_time t)`. Cette instruction tire aléatoirement une valeur autour de `t`. Dans notre étude de cas, les processus des composants *fractale* sont réveillés à des instants de temps simulé différents, donc non parallélisable.

Chacun des trois problèmes identifiés conduit à une séquentialisation des simulations. On donne des pistes pour la résolution de ces problèmes dans la conclusion.

Malgré tout, on veut pouvoir quantifier l'accélération dans le cas où les problèmes sont résolus. Pour cela, on fait abstraction des deux premiers problèmes. Pour le troisième problème, on utilise l'instruction SystemC *wait(sc_time t)* qui aura pour effet de rendre éligible les processus aux mêmes cycles d'évaluation.

Le tableau 6.9 donne les facteurs d'accélération obtenus en jouant sur :

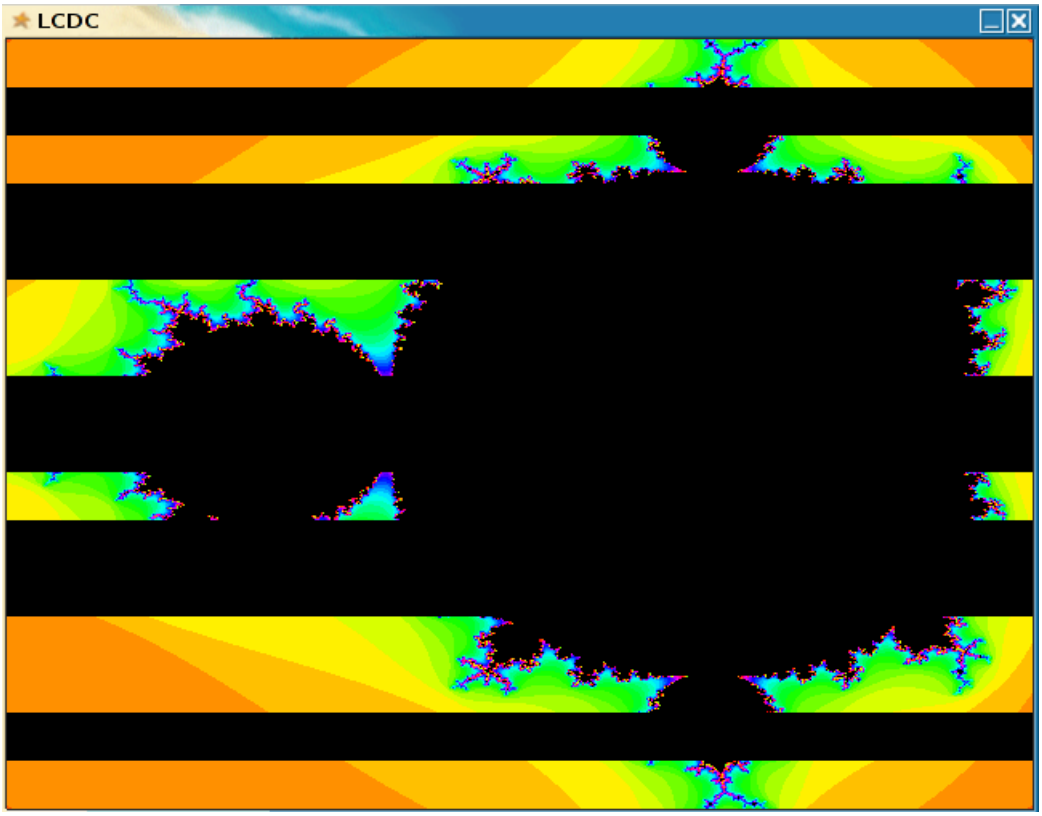

FIG. 6.8 – Affichage par tranche pendant l'exécution de la *fractale de Mandelbrot*

– le nombre de composants *fractale*
– la résolution de l'image. Pour cela, on fait varier le nombre de points réels nécessaires pour calculer la couleur d'un pixel. Par exemple une résolution de 1 signifie qu'on associe à un pixel le calcul d'un point réel. Pour une résolution de 16, la couleur du pixel est calculée en faisant une moyenne des résultats de 16 points calculés, etc...

Le facteur d'accélération F_i est donné par le rapport $T_{osci}(i)$ sur $T_{smp}(i)$ où :

– $T_{osci}(i)$ est le temps de simulation d'une plateforme P avec le noyau de simulation de référence de l'osci, sur une machine comportant i processeurs,
– et $T_{smp}(i)$, le temps de simulation de la même plateforme P avec notre noyau parallélisé, sur la même machine avec i processeur.

Dans le tableau de la figure 6.9, nous montrons les résultats observés sur une machine quadriprocesseur. Les facteurs d'accélérations sont calculés comme suit :

$$F_4 = \frac{T_{osci}(4)}{T_{smp}(4)}$$

Les résultats obtenus montrent qu'on obtient un facteur d'accélération de l'ordre de 3 en moyenne, entre les simulations séquentielles réalisées avec le noyau OSCI et notre noyau parallélisé.

On peut voir que les deux paramètres que nous faisons varier influencent l'efficacité de la parallélisation :

– plus les transitions à exécuter sont longues, plus le facteur d'accélération est élevé. Cette augmentation s'observe pour chaque plateforme (avec 4, 8, 16 ou 64 composants *fractales*).

Nombre de composants *Fractales* \ Résolution	1	4	16
4	$\frac{5,17}{2,22} = 2,33$	$\frac{19,34}{7,96} = 2,43$	$\frac{76,00}{30,26} = 2,51$
8	$\frac{5,13}{1,86} = 2,75$	$\frac{19,34}{6.84} = 2,82$	$\frac{76,25}{26,16} = 2,91$
16	$\frac{5,44}{1.75} = 3,11$	$\frac{19,35}{5,8} = 3,33$	$\frac{76,28}{21,07} = 3,62$
64	$\frac{5,08}{1,67} = 3,04$	$\frac{19,26}{5,36} = 3,59$	$\frac{75,65}{20,63} = 3,66$

FIG. 6.9 – Exemples de facteurs d'accélération obtenu en calculant le rapport du temps de simulation avec le noyau de simulation de référence de l'OSCI, sur le temps de simulation avec notre implémentation parallèle du noyau. Ces résultats ont été obtenus sur une machine avec 4 cœurs de processeur.

Évidemment, cela est vrai que s'il y a plusieurs transitions éligibles, ce qui est le cas avec les 4 plateformes de tests.

– plus il y a de processus indépendants simultanément éligibles, plus la parallélisation est efficace. Et ceci s'observe quelque soit la résolution choisie (1, 4 ou 16 appels à la fonction de *Mandelbrot* pour le calcul de la couleur d'un pixel).

Ces résultats sont intéressants car ils permettent de caractériser les modèles TLM permettant une parallélisation efficace. En effet, on voit grâce à ces tests que le meilleur facteur d'accélération de 3,66 est obtenu avec la plateforme constituée de 64 composants *fractales* avec une résolution de 16. Donc, plus il y a de longues transitions indépendantes éligibles simultanément, meilleure est la parallélisation.

Ces tests pourraient être encore plus précis en augmentant encore le nombre de composants et la finesse de résolution. Cela permettrait d'estimer la limite de l'efficacité.

Chapitre 7

Conclusion

Sommaire

Notre travail a consisté à trouver un moyen d'exploiter les machines multiprocesseurs pour accélérer les simulations SystemC. La contrainte fondamentale est de préserver la sémantique d'exécution définie par la norme du langage. Il nous a fallu pour cela un cadre formel pour garantir que les exécutions parallèles sont équivalentes aux exécutions séquentielles sur monoprocesseur. Cela nous a amené à définir la granularité des tâches à exécuter en parallèle et la notion d'indépendance associée à ce grain. Nous décrivons ensuite notre approche pour la résolution du problème qui se décompose en deux parties : la mécanique d'exécution parallèle pour exploiter les ressources de calcul, et une partie décrivant l'analyse des dépendances à réaliser avant de lancer les simulations.

7.1 Résumé des contributions

7.1.1 Étude de la parallélisation SystemC à sémantique constante

Dans nos travaux, nous avons d'abord étudié comment paralléliser les simulations SystemC de manière à ce que les comportements observés respectent la sémantique coopérative du langage. On définit pour cela la notion d'indépendance entre les tâches à exécuter en parallèle, de manière à garantir la conformité des comportements observés par rapport à la norme. Deux tâches sont dites indépendantes si elles ne partagent pas de ressources. Nous nous appuyons sur la théorie des ordres partiels pour mettre en évidence que si deux tâches sont indépendantes, alors on peut les exécuter en parallèle en respectant la sémantique coopérative du langage. Cette propriété résulte du fait qu'il existe un ordonnancement séquentiel des tâches exhibant un comportement équivalent. L'observation

des comportements se fait sur la séquence des états de la mémoire quand les processus sont en attente, par exemple aux changements de cycle (δ-cycle ou cycle temporel). On a choisi comme granularité pour la parallélisation l'unité atomique d'exécution des coroutines SystemC, ce qui permet d'avoir un haut degré de parallélisme potentiel.

Nous avons découpé le problème en deux parties : une partie dans laquelle on s'intéresse à l'implémentation d'un noyau parallélisé pour permettre d'exécuter simultanément plusieurs coroutines, et une partie dans laquelle on s'intéresse à l'analyse d'accessibilité des variables partagées depuis chaque point de reprise des coroutines.

7.1.2 Implémentation d'un noyau d'exécution parallèle

Nous avons implémenté un noyau d'exécution distribué en nous basant sur l'implémentation du noyau fourni par l'OSCI, et la bibliothèque *Pthread* pour l'exploitation des ressources matérielles. Pour cela nous créons, juste après l'élaboration de la plateforme, $N + 1$ fils d'exécution *Pthread*, où N est le nombre de processeurs de la machine hôte. Dans notre implémentation on ne s'intéresse qu'à la parallélisation des phases d'évaluation des processus. Les phases de *mise à jour* sont exécutées en séquence sur le dernier fil d'exécution actif après l'évaluation. Nous donnons l'algorithme de l'ordonnanceur séquentiel dans lequel nous identifions toutes les variables nécessaires au noyau pour la simulation. Puis, nous donnons l'algorithme distribué que nous avons implémenté et montrons que les accès concurrents aux ressources partagées se font par exclusion mutuelle. Nous avons testé la mécanique mise en place pour vérifier qu'il n'y a pas d'interblocage ni de famine.

7.1.3 Algorithmes d'analyse statique pour l'identification des dépendances

Pour décider dynamiquement de l'indépendance des coroutines prêtes à s'exécuter, nous devons connaître à l'avance les variables partagées auxquelles peuvent accéder (lecture ou écriture) les différentes coroutines aux différents points de reprise. Nous spécifions les analyses statiques à réaliser en nous basant sur les flots de contrôle des processus. Nous proposons une architecture pour automatiser la parallélisation des simulations incluant les analyses de dépendances et le noyau d'exécution parallèle. Les flots de contrôle peuvent être déduits des arbres abstraits des processus disponibles à la sortie de *Pinapa*, un frontal SystemC basé sur `gcc`. Nous avons aussi modifié la bibliothèque SystemC pour pouvoir identifier de manière unique les points de reprise des coroutines à l'exécution (les différents appels à *wait(...)*). D'autre part, nous avons développé un outil d'instrumentation pour modifier automatiquement les appels aux *wait()* par des appels à notre implémentation. Pour l'analyse statique on doit inévitablement réaliser des sur-approximations plus ou moins fines qui nécessitent des techniques d'analyse statique plus ou moins avancées. Notre approche mène alors naturellement à évaluer les gains de vitesse sur des études de cas réalistes selon la finesse de l'analyse.

7.1.4 Identification des problèmes d'efficacité de la parallélisation

Le problème de la parallélisation SystemC à sémantique constante s'avère être un problème d'analyse statique non trivial. Pour le résoudre de manière exacte, il faudrait être capable de deviner à l'avance l'ensemble des variables qui vont être touchées pendant l'exécution d'un processus, et ceci est impossible. Nous sommes contraints de résoudre le problème des dépendances de manière approchée en calculant des surapproximations d'objets accessibles. Le fait d'être pessimiste peut nuire à l'efficacité, mais en aucun cas produire des résultats non conformes, et cela reste notre objectif fondamental. Nous avons donc décidé d'évaluer l'impact de nos surapproximations au niveau transactionnel

sur l'efficacité de la parallélisation.

7.1.4.1 Étude de cas

Nous avons développé une étude de cas réalisant le calcul réparti de la *fractale de Mandelbrot*. Nous nous sommes basés sur une étude de cas réaliste que nous avons adaptée pour nos besoins. L'objectif est de pouvoir quantifier les accélérations obtenues selon la finesse de l'analyse. La plateforme est paramétrée :

- par le nombre de composants réalisant le calcul (*fractale*) pour pouvoir jouer sur le degré de parallélisme,
- et par la résolution de l'image résultat pour jouer sur la longueur des calculs.

7.1.4.2 Décodage des adresses des transactions

Nous avons pu voir grâce à cette étude de cas que l'utilisation du routeur de transaction (classe `tac_router`) nécessite des sur-approximations. En effet, le rôle d'un composant routeur est d'aiguiller les transactions depuis l'initiateur vers la cible de la transaction. L'adresse de la cible de la transaction est décodée dynamiquement grâce à la carte des adresses mémoire et la fonction cible n'est connue qu'à l'exécution. Si on est trop pessimiste dans nos sur-approximations, on doit considérer tous les composants comme cibles potentielles de la transaction. Dans notre étude de cas, et dans le cas général, cette abstraction est certes conservative mais malheureusement beaucoup trop forte pour déceler des tâches indépendantes et espérer pouvoir paralléliser.

7.1.4.3 Composant mémoire globale

Un autre cas problématique est celui de l'utilisation d'un composant mémoire. Par exemple, la classe `tac_memory` représente une mémoire centrale partagée par tous les composants. En étant pessimiste, on doit considérer tous les processus réalisant une transaction vers ce composants comme dépendants[1]. Sur notre étude de cas, les composants *fractales*, après avoir réalisé le calcul de leur portion d'image, initient une transaction d'écriture pour ranger leur résultat en mémoire. Ils se retrouvent de fait dépendants, et on ne peut pas les exécuter en parallèle avec notre approche conservative. Ce problème est évidemment généralisable à tout modèle de système sur puce avec un composant mémoire globale et l'efficacité des simulations parallèles sera fortement limitée. Même en ayant résolu le problème du décodage des adresses, ce problème demeure.

7.1.4.4 Attente sur du « temps imprécis »

Pour explorer différents comportements réalistes des plateformes, l'équipe SPG de STMicroelectronics a introduit une instruction d'attente spécifique pour simuler l'indéterminisme des temps de réponse du matériel simulé. Cette instruction `pv_wait(sc_time t)` permet de choisir aléatoirement une durée d'attente sur un intervalle de valeurs autour de `t`. En pratique, la valeur est choisie dans l'intervalle $\texttt{t} \pm 50\%$.

Sur notre étude, les composants *fractales* qui exécutent cette instruction d'attente se retrouvent éligibles à des instants de temps simulé différents. Ils ne sont donc pas parallélisables avec notre approche. Le problème se pose sur notre étude de cas parce que nous avons plusieurs instances du module *Fractale*, donc plusieurs processus décrivant le même comportement, et en particulier

[1] Bien entendu, on peut être plus fin en considérant deux lectures comme indépendantes.

le même intervalle pour l'attente. Plus généralement, si on veut pouvoir paralléliser efficacement il faut nécessairement que plusieurs processus soient éligibles simultanément. Malheureusement, l'instruction `pv_wait()` fait l'effet inverse en répartissant aléatoirement le réveil des processus à des instants différents.

7.2 Pistes pour la résolution des problèmes

On a énuméré plusieurs problèmes d'efficacité pour la parallélisation des simulations SystemC au niveau transactionnel. On propose pour chacun de ces problèmes des pistes pour leur résolution.

7.2.1 Pistes pour le problème d'adressage des transactions

Connaître statiquement l'adresse d'un composant cible d'une transaction s'avère être crucial pour la parallélisation. Mettre en œuvre des techniques avancées d'analyse statique pourrait permettre de résoudre en partie le problème. On rappelle que les adresses de base et la taille des plages d'adressage des composants sont définies dans la carte des adresses mémoire (`Memory map`). En effet, les adresses données en paramètre des transactions sont généralement de la forme `base_address + offset`, où `base_address` définit l'adresse de base du composant cible et `offset` définit le décalage depuis l'adresse de base. Si on suppose que l'adresse de base est une constante connue et que le décalage est une opération arithmétique « simple », il est alors envisageable de mettre en œuvre des techniques de pré-évaluation de constante pour connaître l'adresse des transactions et d'en déduire a priori la cible. Dans le cas général, si le paramètre adresse est une expression plus compliqué, on peut envisager d'avoir recours à des techniques d'interprétation abstraite pour tenter de borner l'adresse.

On peut aussi imaginer s'abstraire du décalage et supposer que le décalage (`offset`) ne dépasse jamais la taille allouée au composant et ainsi déterminer la cible en ne regardant que l'adresse de base. Cette deuxième idée n'est pas généralisable car en pratique, il arrive que certains modèles utilisent l'`offset` pour changer de cible pour leur transaction.

7.2.2 Plage d'adressage de la mémoire globale

Pour pouvoir trouver une solution il faut évidemment avoir résolu le problème du routage des transactions. Pour déterminer que deux transactions vers la mémoire sont indépendantes, c'est-à-dire qu'elles ciblent des espaces disjoints, on peut envisager de mettre en place les techniques avancées d'analyse statique énoncées précédemment. Une autre solution à envisager est de demander cette information au programmeur par l'intermédiaire d'un fichier de configuration, de la même manière qu'il fournit la carte des plages mémoire. L'inconvénient majeur de cette seconde solution est que le respect de la sémantique est alors délégué au programmeur.

7.2.3 Synchronisation des « temps imprécis »

Le problème posé par l'instruction `pv_wait()` est plus délicat. La solution que nous proposons consiste à synchroniser le réveil des processus pour lesquels il n'y a pas d'ordre prédéfini. Par exemple, considérons deux processus A et B. Le processus A exécute une instruction d'attente `pv_wait(100, SC_NS)` et le processus B une instruction `pv_wait(50, SC_NS)`. A sera alors réveillé aléatoirement dans un délai compris dans l'intervalle $[50, 150]$ et B dans l'intervalle $[25, 75]$. Au regard de ces valeurs, on voit que A peut être réveillé avant B, par exemple en prenant

comme valeur 50 pour A et 60 pour B. B peut aussi être réveillé avant A en prenant par exemple 100 pour A et 75 pour B. Le tirage aléatoire peut même fournir des valeurs identiques aux deux instructions d'attente, et rendre éligible les deux processus au même instant de simulation. C'est justement ce troisième cas que l'on veut exploiter pour permettre de rendre éligible des processus au même instant. Ceci peut se faire en choisissant la même valeur de réveil pour des processus dont l'intersection des intervalles d'attente est non vide.

L'inconvénient majeur de cette solution est que l'on force des synchronisations, exactement à l'endroit où l'on voulait faire apparaître de l'indéterminisme lié aux temps de réponse du matériel. La conséquence est que l'ensemble des comportements observables en parallèle sera un sous-ensemble strict des comportements observables en séquentiel.

7.3 Perspectives

L'ordonnanceur SystemC a été conçu pour des simulations séquentielles. SystemC a depuis longtemps été utilisé avec cette mécanique d'exécution. Pour ne pas changer les comportements des modèles TLM, la parallélisation éventuelle doit se faire à sémantique constante. Comme le langage ne donne aucune structure qui permettrait de garantir que deux portions de code sont parallélisables, la seule solution est de réaliser une analyse de dépendance fine. Or, on travaille sur du code SystemC quelconque, donc on ne peut que faire une analyse approchée.

Les études de cas réalisées nous ont montré qu'une parallélisation efficace dépend beaucoup de la formes des modèles TLM. Dans certains cas, nous avons montré qu'une analyse fine pouvait résoudre le problème. Cependant, le coût des analyses doit rester raisonnable pour ne pas réduire à néant le gain induit par la parallélisation. Si l'on voulait garantir un gain en parallélisation, il faudrait réaliser des analyses statiques très fines et donc très couteûses. Cela ne paraît pas conduire à une solution raisonnable.

Les autres pistes sont les suivantes :

1. on peut envisager de modifier SystemC, mais comme il est devenu un standard et que de nombreux modèles existent déjà, c'est sûrement difficile
2. on peut aussi envisager d'influencer la forme de modèles TLM, par des règles d'écriture. Dans le domaine de la modélisation TLM des SoCs, il y a une certaine habitude de travailler avec de telles règles. On a donc l'espoir de pouvoir proposer d'autres règles en relation avec la parallélisation. En particulier on peut raisonnablement définir des règles pour la modélisation des bus. En effet, un modèle de bus est très réutilisable, et le travail n'est fait qu'une fois. Cela a plus de chances d'être accepté que des règles sur le code des processus qui représentent des blocs moins réutilisable. Par ailleurs, en TLM, influencer sur les bus a beaucoup d'effet sur l'indépendance des portions de code. Par exemple, on peut étendre les modèles de bus en ajoutant des communications non atomiques. La structure en module des modèles utilisant ce type de communication devient alors un critère simple d'indépendance : deux portions de code de deux modules différents sont indépendants et donc parallélisables.

Bibliographie

[BBP89] Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors. *Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings*, volume 398 of *Lecture Notes in Computer Science*. Springer, 1989. 2.1.3.2

[Blu05] Bluespec. bluespec, 2005. http://www.bluespec.com/. 2.1.2.4

[BNG+06] Rabie Ben Atitallah, Smaïl Niar, Alain Greiner, Samy Meftali, and Jean-Luc Dekeyser. Estimating energy consumption for an mpsoc architectural exploration. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *ARCS*, volume 3894 of *Lecture Notes in Computer Science*, pages 298–310. Springer, 2006. 2.2.3

[Bry77] R. E. Bryant. Simulation of packet communication architecture computer systems. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, USA, November 1977. Technical Report TR-188. 3.2.2

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis : An integrated electronic system design environment. *Computer*, 36(4) :45–52, 2003. 2.3.1.2

[Cad99] Cadence Design Systems. NC-SystemC, 1999. http://www.cadence.com/products/functional_ver/nc-systemc/. 2.3.1.1

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2 : An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer. 2.1.3.2

[CCZ06] Bastien Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006. 3.2

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling : an overview. In *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press. 2.1.2.2

[CM79] K. Mani Chandy and Jayadev Misra. Distributed simulation : A case study in design and verification of distributed programs. *IEEE Trans. Software Eng.*, 5(5) :440–452, 1979. 3.2.2

[CMMC07] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. Séparation des aspects fonctionnels/non-fonctionnels dans les modèles transactionnels des systèmes sur puce, January 2007. FETCH (poster). 2.2.3

[DGG06] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, version 2.0*. SpecC Technology Open Consortium, 2006. http://www.ics.uci.edu/~specc/reference/SpecC_LRM_20.pdf. 2.3.1.2

[For04] Forte Design Systems. Cynthesizer, 2004. http://www.forteds.com/products/index.asp. 2.1.2.4

[Ghe05] Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005. ISBN 0-387-26232-6. 2.2.1

[GNJ+96] Gopi Ganapathy, Ram Narayan, Glenn Jorden, Denzil Fernandez, Ming Wang, and Jim Nishimura. Hardware emulation for functional verification of k5. In *DAC '96 : Proceedings of the 33rd annual conference on Design automation*, pages 315–318, New York, NY, USA, 1996. ACM Press. 2.1.1

[Hel07] Claude Helmstetter. *Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis*. PhD thesis, INPG, Grenoble, France, March 2007. 3.1.1, 6.2

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992. 2.1.3.2

[HMMCM06] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. *FMCAD*, 0 :171–178, 2006. 3.1.1

[HTCT03] Y. Hsu, B. Tabbara, Y. Chen, and F. Tsai. Advanced techniques for RTL debugging, 2003. 2.1.1

[HZB+03] Renate Henftling, Andreas Zinn, Matthias Bauer, Martin Zambaldi, and Wolfgang Ecker. Re-use-centric architecture for a fully accelerated testbench environment. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 372–375, New York, NY, USA, 2003. ACM Press. 2.1.1

[Kep93] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993. 4.1.2

[LMTY02] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with tla, 2002. 2.1.3.2

[MMMC05] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa : An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, September 2005. 5.3.1

[Ope05] Open SystemC Initiative. *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*, 2005. http://www.systemc.org/. 2.3.1.1, 3.1.2.1, 4, 4.1.1.1

[Pas02] Sudeep Pasricha. Transaction level modeling of SoC in systemc 2.0. Technical report, STMicroelectronics Ltd, 2002. 2.1.2.2

[RR02] M. Renaudin and J.-B. Rigaud. Etude de l'art sur la conception des circuits asynchrones, perspectives pour l'intégration des systèmes complexes [TIMA-RR–02/12-02–FR]. Technical report, TIMA, 2002. Document réalisé avec le support de STMicroelectronics, Crolles, janvier 2000. 2.2.2

[Sch03a] Klaus-Dieter Schubert. Improvements in functional simulation addressing challenges in large, distributed industry projects. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 11–14, New York, NY, USA, 2003. ACM Press. 2.1.1

[Sch03b] Tom Schubert. High level formal verification of next-generation microprocessors. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 1–6, New York, NY, USA, 2003. ACM Press. 2.1.3.2

[SCI05] STMicroelectronics, CNRS, and INPG. Pinapa, 2005. http://greensocs.sourceforge.net/pinapa/. 5.3.1

[SPI03] The SPIRIT Consortium. Spirit, 2003. http://www.spiritconsortium.org. 2.3.1.2

[Syn05] Synfora. PICO Express, 2005. http://www.synfora.com/products/picoexpress.html. 2.1.2.4

[tac05] TAC : Transaction accurate communication/channel, 2005. http://http://www.greensocs.com/TACPackage. 6.2.1, 6.2.3

[tlm06] OSCI SystemC TLM 2.0, draft 1 for public review, 2006. http://www.systemc.org/web/sitedocs/TLM_2_0.html. 2.2.1

[VMD] J. Vennin, Samy Meftali, and Jean-Luc Dekeyser. Understanding and extending systemc user thread package to ia-64 platform. 4.1.2.1

[VPG06] Emmanuel Viaud, François Pécheux, and Alain Greiner. An efficient tlm/t modeling and simulation environment based on conservative parallel discrete event principles. In *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, pages 94–99, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 2.2.3

[Wik06] Wikipédia. Circuit logique programmable, 2006. http://fr.wikipedia.org/w/index.php?title=Circuit_logique_programmable&oldid=12045081. 2.1.1

[Wik07] Wikipédia. Fractale de mandelbrot, 2007. http://fr.wikipedia.org/wiki/Fractale. 6.2.1

[YG02] Jin Yang and Amit Goel. Gste through a case study. In *ICCAD '02 : Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 534–541, New York, NY, USA, 2002. ACM Press. 2.1.3.2

[YS01] J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. In *ICCD '01 : Proceedings of the International Conference on Computer Design : VLSI in Computers & Processors*, page 360, Washington, DC, USA, 2001. IEEE Computer Society. 2.1.3.2

Index

Résumé

SystemC est une extension du langage C++, définissant une bibliothèque de classes C++ utilisée pour décrire des modèles de systèmes sur puces à différents niveaux d'abstraction. Le niveau de description transactionnel (TLM) permet d'exécuter le logiciel embarqué sur un prototype virtuel du matériel très tôt dans le flot de conception.

Le modèle d'exécution défini dans la spécification officielle (standard IEEE-1666) a été pensé pour des simulations sur machine monoprocesseur, c'est-à-dire que les processus décrivant le comportement du matériel sont exécutés en séquence selon une politique d'ordonnancement. SystemC étant basé sur une sémantique coopérative (les fils d'exécution rendent la main explicitement), la politique d'ordonnancement est non préemptive.

Nos travaux ont porté sur la parallélisation du moteur d'exécution SystemC pour exploiter au mieux les architectures multiprocesseurs, tout en conservant la sémantique d'exécution coopérative du langage.

Nous avons d'abord étudié différentes approches existantes, basées sur la structure en composants des modèles, et nous avons montré leurs limitations sur des modèles transactionnels. Ensuite, nous avons identifié une granularité des tâches à exécuter en parallèle permettant la parallélisation efficaces des modèles transactionnels.

La conformité des simulations vis-à-vis de la norme est assurée par une analyse préalable des dépendances de données entre les différentes tâches. Enfin, nous avons défini et implémenté les algorithmes parallèles pour la répartition dynamique des tâches indépendantes et nous donnons les éléments nécessaires pour l'analyse statique des dépendances entre les tâches à exécuter.