Simulation de haut niveau de systèmes d'exploitations distribués pour l'exploration matérielle et logicielle d'architectures multi-noeuds hétérogènes

Emmanuel Huck

N° d'ordre : 2011-XX-XXX

THÈSE

présentée à
L'UNIVERSITÉ DE CERGY-PONTOISE

pour obtenir le titre de :

Docteur ès Science de l'Université de Cergy-Pontoise

Spécialité : Sciences et Technologies de l'information et de la Communication

Par

Emmanuel HUCK

Laboratoires d'accueil :
Équipes de Traitement des Images et du Signal (ETIS/ASTRE) - UMR CNRS 8051
Thales Research & Technology (TRT/LSE)

SIMULATION DE HAUT NIVEAU DE SYSTÈMES D'EXPLOITATIONS DISTRIBUÉS POUR L'EXPLORATION MATÉRIELLE ET LOGICIELLE D'ARCHITECTURES MULTI-NŒUDS HÉTÉROGÈNES

Présentée le 25 Novembre 2011
devant le jury composé de :

Président :	M.	Frédéric	PETROT	Laboratoire TIMA-SLS
Rapporteurs :	M.	Guy	GOGNIAT	Laboratoire Lab-STICC
	Mme	Cécile	BELLEUDY	Laboratoire LEAT
Examinateurs :	M.	Dragomir	MILOJEVIC	Laboratoire BEAMS - ULB
Directeur :	M.	François	VERDIER	Laboratoire ETIS-CNRS
Co-encadrant :	M.	Benoît	MIRAMOND	Laboratoire ETIS-CNRS
Invités :	M.	Fabrice	LEMONNIER	THALES TRT-LSE
	M.	Dominique	RAGOT	THALES Communication France
	M.	Philippe	BONNOT	THALES TRT-LSE
	M.	Philippe	MILLET	THALES TRT-LSE

Résumé de la thèse

La conception classique d'un système électronique pour une nouvelle application embarquée recherche le bon compromis algorithme/architecture satisfaisant les propriétés et contraintes temps-réel attendues. Dans les systèmes réactifs distribués sur de multiples nœuds d'exécution hétérogènes, l'évaluation préalable du comportement de l'application sur la future plate-forme doit contrôler le respect permanent des attentes, malgré des charges de calcul fluctuantes.

Cette thèse soutient l'idée qu'il est nécessaire, pour concevoir un système sur puce multi-processeurs (MPSoC) dédié à une application embarquée, de valider au préalable par simulation de haut niveau l'exploration de l'espace de conception architectural, logiciel et matériel. Prédire les propriétés du système avant son assemblage est particulièrement justifié dans les circuits électroniques reconfigurables, qui modifient le support d'exécution au cours de son fonctionnement.

La méthodologie proposée dans cette thèse consiste à se focaliser sur les services dédiés de la plateforme et d'observer par simulation les comportements temps-réel attendus. Elle propose notamment d'explorer et évaluer :

- les conséquences des multiples choix de répartition des tâches de calculs entre logiciel ou matériel,
- l'influence de l'algorithmique des services du système d'exploitation gérant ces tâches,
- les implémentations des services en logiciel ou matériel,
- la distribution de ces services sur une plateforme multi-cœurs/multi-nœuds.

Ce point de vue du gestionnaire de la plate-forme consiste à focaliser la modélisation du système embarqué complet principalement sur les caractéristiques de haut niveau du système d'exploitation. Cela comprend les différentes déclinaisons possibles des services distribués, et permet de se focaliser sur la gestion globale des ressources concurrentes du système sans avoir à modéliser explicitement ni dans le détail ces ressources. Cette vision est la contribution méthodologique principale de cette thèse.

Pour y parvenir nous avons adopté une démarche de modélisation modulaire des principaux services représentatifs des OS temps réel existants à ce jour. Nous pouvons ainsi assembler les services choisis et concevoir un système d'exploitation dédié à une application particulière. Pour le concepteur, ce modèle sert alors de spécification exécutable de son système, le modèle pouvant simuler conjointement, de manière fonctionnelle et temporelle en SystemC, le matériel, l'ensemble des tâches applicatives et la couche de gestion ainsi explorée.

Pour intégrer la distribution de tâches logicielles sur les plateformes embarquées actuelles et à venir constituées de multiples nœuds de calcul hétérogènes, nous avons étendu le modèle pour permettre la communication entre plusieurs instances de gestionnaires (OS) répartis sur les processeurs et accélérateurs d'un système MPRSoC. Cette extension prend la forme de nœuds de communication et de routage génériques modélisés à haut niveau en utilisant une démarche de type transactionnelle. Ce nouveau modèle d'OS distribué permet d'explorer et d'évaluer rapidement la répartition, en logiciel ou matériel, des tâches et des services de gestion dans une simulation conjointe et fonctionnelle.

Dans le cadre d'architectures multiprocesseurs hétérogènes, nous avons contribué à la conception du simulateur du calculateur embarqué haute performance Ter@ops. Ce simulateur intègre notre modèle d'OS comme cœur de la simulation multi-threads et combine le modèle d'OS de haut niveau à un modèle raffiné des communications inter-processeurs et des échanges mémoires.

Dans le cadre du projet OveRSoC, nous avons validé le bon fonctionnement du modèle et montré son utilité pour l'exploration d'architectures reconfigurables dynamiquement dédiées à une application de vision robotique. Nous avons ainsi expérimenté l'intégration dans notre modèle de nouveaux services spécifiques dédiés à la gestion des ressources reconfigurables et évalué l'architecture temps-réel idéale de cette application de vision robotique caractérisée par une forte dynamicité, due à une sensibilité aux données visuelles d'entrée.

Mots-clés : Modélisation SystemC/TLM, RTOS, co-simulation, MPSoC, Reconfigurable

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Problématique de la conception de systèmes sur puce

Sommaire

1.1 Conception de systèmes logiciels et matériels

La conjonction des contraintes du marché des systèmes électroniques et de l'évolution des technologies de fabrication des circuits intégrés nous amène aujourd'hui à concevoir des circuits de plus en plus complexes (3,9 milliards de transistors dans la même puce FPGA : l'Altera Stratix® V [Alt11]), dans un délai toujours plus court (*time to market* de quelques mois). C'est le problème actuel du concepteur de système : repenser le processus de conception allant de l'idée au produit.

Composés initialement de quelques milliers de portes électroniques, les circuits sont maintenant des systèmes complexes structurés et intégrés sur une seule et même puce électronique. Chacune peut contenir un système complet avec un très grand nombre de processeurs, des mémoires, des périphériques évolués de technologies hétérogènes, plusieurs bus de communication jusqu'au "réseau sur puce" pour répondre à des contraintes de performance, de coût et de consommation électrique de plus en plus fortes. Les outils de conception ont évolué pour automatiser au maximum la conception des puces et de leurs masques de gravure. Le principe de la conception de tels systèmes (sur une seule puce ou non) commence généralement par la spécification de haut niveau pour aller vers leur réalisation physique.

Malgré cela, ces puces ne sont pas encore assez puissantes en regard de la complexité et de la puissance de calcul nécessaires aux applications envisagées dans de nombreux projets de recherche [Sys]. Cette complexité est telle qu'aujourd'hui on parle même de *systèmes de systèmes* [Jam05], pour décrire un ensemble de systèmes en interaction mais surtout de conceptions hétérogènes, la dernière évolution étant la possibilité de reconfigurer dynamiquement des parties matérielles.

En ouvrant des possibilités nouvelles, cette combinaison de dynamicité logicielle et de flexibilité matérielle engendre encore plus de possibilités de comportements difficilement

prédictibles voire aléatoires dans de tels systèmes. Nous allons voir qu'il est nécessaire aujourd'hui, devant de tels comportements aléatoires, de vérifier le fonctionnement dès la phase de conception, et pour cela d'élaborer de nouveaux outils/méthodes/méthodologies le permettant. Outre les enjeux humains [1], les enjeux économiques de cette garantie de fonctionnement sont à la hauteur des sommes colossales risquant d'être perdues pour un masque de circuit inutilisable ou mal conçu.

Non seulement les outils de conception actuels imposent une vérification fonctionnelle après coup, obligeant à réitérer le processus de conception, mais de plus ils permettent difficilement l'intégration de nouveaux composants matériels flexibles ainsi que la partie logicielle, pour évaluer ces aspects dynamiques. Intégrer à la fois des parties logicielles s'exécutant sur des processeurs, mais aussi d'autres parties matérielles, implémentées à l'aide de logique reconfigurable dynamiquement, oblige à repenser la validation de ces systèmes dès leur conception. Le processus de conception dans ce contexte ne peut se contenter d'une exploration au niveau matériel mais doit s'effectuer au niveau du système.

1.2 Contexte et enjeux des systèmes sur puce multi-processeurs hétérogènes

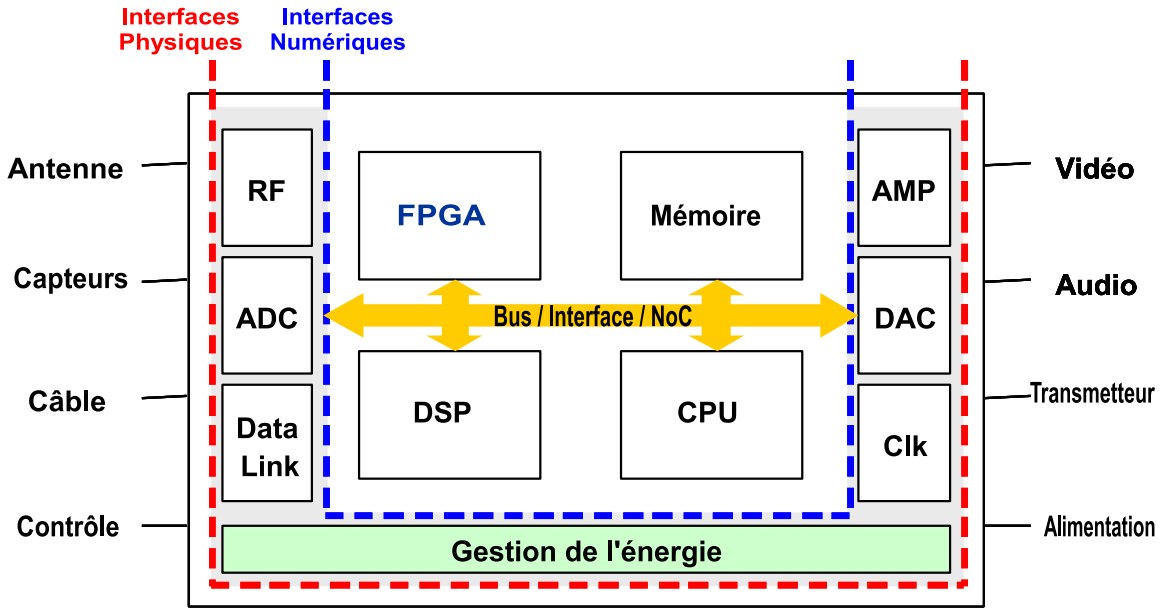


Figure 1.1: *Architecture simplifiée d'un système sur puce (SoC) embarqué*

Ces systèmes de plus en plus complexes deviennent omniprésents dans le quotidien de chacun : dans les téléphones portables, consoles de jeux ou décodeurs (TV) numériques, et bien évidemment dans de nombreux autres systèmes électroniques industriels classiques (automobile, aéronautique, etc.).

On appelle *système-sur-puce*, ou System-on-Chip multi-processeurs (MP-SoC), les ASIC [2] qui contiennent un système éléctronique complet dans la même puce.

Les MPSoC actuels intègrent des processeurs généralistes, des processeurs spécialisés, des blocs de calculs dédiés (encodage/décodage de voix d'un téléphone portable, dé-chiffrement, etc.), de la mémoire, des contrôleurs d'entrée/sortie, des moyens de communication complexes pour relier les composants entre eux (bus hiérarchiques, réseaux

1. Dans les systèmes critiques (cas non etudiés dans cette thèse)
2. Application Specific Integrated Circuit, ou circuit intégré dédié

de communication par paquets, etc.). On y trouve aussi d'autres composants hétéroclites pour réduire les coûts de production, ou permettre de miniaturiser les produits (antennes RF, convertisseurs analogiques/numériques, capteurs MEMS, etc...). On tend également à y intégrer des zones dites reconfigurables, implémentés par exemple dans les FPGA [3] comme on peut le voir sur le schéma générique de la figure 1.1.

L'ensemble de ces composants est relié par des mécanismes de communication sophistiqué (réseaux sur puce) qui supportent des échanges de données toujours croissant, et qui sont une source non négligeable de consommation énergétique, tout en fournissant un moyen pour faciliter l'exploration d'architecture.

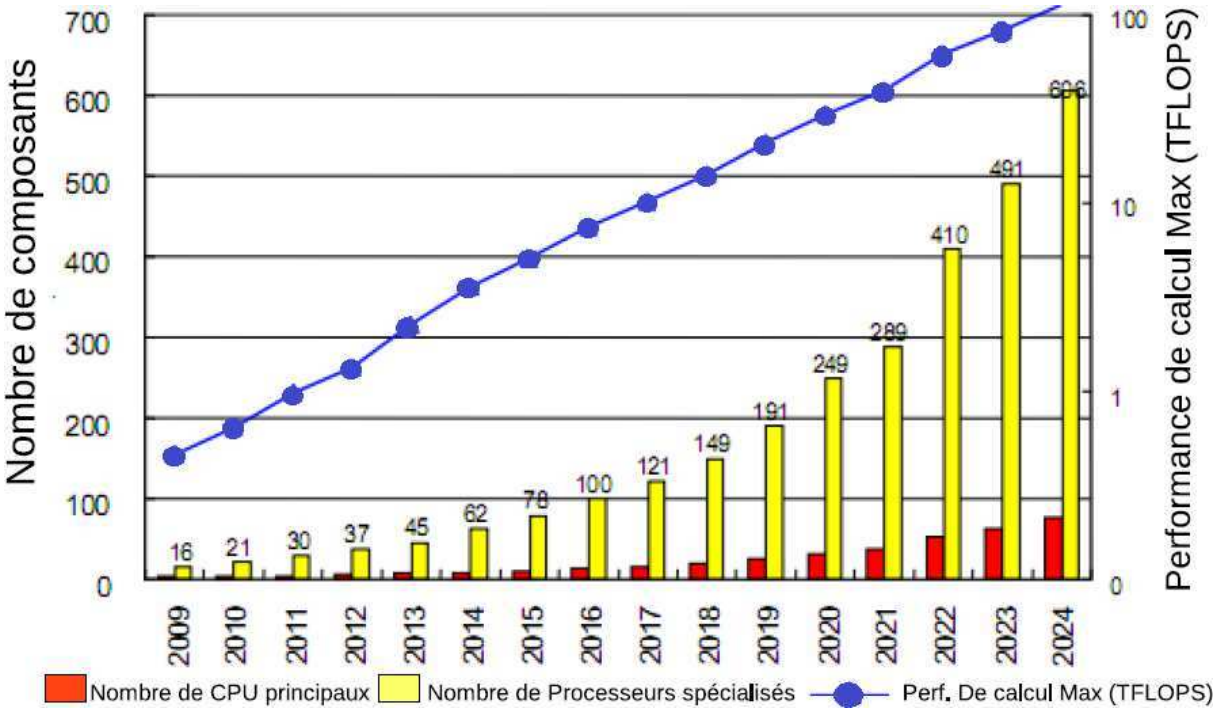


Figure 1.2: *Tendance ITRS 2009 du nombre de processeurs généralistes et spécialisés par SoC [ITR09]*

Les processeurs classiques, multi-cœurs ou non, tendent à devenir des systèmes sur puce, en y intégrant des fonctionnalités annexes. Le dernier rapport de l'ITRS [ITR09] montre que la plupart des systèmes sur puce actuels possèdent plusieurs cœurs de processeurs, et de multiples éléments de calcul périphérique (PE) comme les processeurs de données spécialisés (i.e. DSP) ou des fonctions accélératrices annexes. Cette tendance devrait s'accentuer, d'un facteur estimé à 1,4 PE [4] de plus par an, comme on peut le voir sur le graphique 1.2.

Bientôt omniprésents, les MPSoC deviennent de plus en plus complexes, sans allégement des contraintes de rapidité de mise sur le marché. Il est plus que jamais crucial de maîtriser la conception de tels systèmes et d'en contrôler la fiabilité. Cette conception a constamment évolué, et doit adapter ses modèles à cette nouvelle complexité, notamment celle de la concurrence conséquente de la parallélisation des traitements.

La conception de ces systèmes-sur-puce multi-processeurs représente un défi que les techniques actuelles ne savent pas résoudre de manière globale. Le problème alors est que, pour chaque nouvelle fonctionnalité, il devient nécessaire de concevoir un nouveau système spécifique différent de ceux préexistants, ce qui rend délicate la vérification des bonnes propriétés.

3. Field Programable Gates Array : matrice de portes logiques configurables
4. Processing Element, ou processeur élémentaire

Cet aspect de la vérification est particulièrement important pour les architectures SoC multi-cœurs, car avec la parallélisation du logiciel embarqué, le partage des ressources et les comportements dynamiques peuvent produire une quasi infinité de scénarios de fonctionnement, donc autant de cas de tests associés. C'est particulièrement vrai pour des applications fortement dynamiques dont la charge de calcul dépend de l'arrivée des données et varie selon leur valeur. Cela nécessite plus de cycles de test (ou simulation) longs et fastidieux (test à chaque niveau : unitaire, d'intégration, validation fonctionnelle...) pour vérifier le bon fonctionnement. Comme pour le logiciel, des outils de conception, de synthèse et de vérification de haut niveau sont désormais nécessaires pour concevoir un système complet.

Les systèmes comprenant de multiples nœuds d'exécution se comportent de manière encore moins prédictible, du fait de leurs interactions. Les temps de chargement des données et instructions vont dépendre de la gestion des caches mémoire, mais aussi de la vitesse du bus et surtout de sa congestion dûe à des accès concurrents par d'autres nœuds. Aussi la répartition des calculs sur les nœuds de calculs doit être pensée pour traiter la charge et les congestions éventuelles.

Seul un système d'exploitation (OS [5]) peut gérer ces besoins complexes de programmation distribuée. Le système d'exploitation va ainsi assurer le rôle de gestionnaire de plateforme [NVC10] et rendre le système plus flexible au cours de l'exécution, en répartissant les charges selon les besoins et/ou les contraintes (d'énergie par exemple). Le choix crucial va donc être celui de l'OS adéquat aux besoins de l'application et de la plateforme matérielle (elle-même choisie adéquatement). Le système de gestion de plateforme (ou système d'exploitation au sens large non restreint à sa composante logicielle) devient ainsi le composant central qui doit être pris en compte lors de la conception d'un système [NVC10]. Mais le système d'exploitation générique qui résoudrait tous les problèmes de partage des ressources, de gestion de la concurrence, et sur n'importe quelle plateforme n'existe pas. D'un OS logiciel à l'autre, les comportements temps-réel vont varier selon les algorithmes utilisés, notamment pour la gestion et l'ordonnancement des tâches. Généralement, il est possible d'acheter des OS logiciels portés sur les processeurs désirés, mais les particularités de la plateforme nécessitent d'y adjoindre ses propres pilotes et fonctions de communication entre processeurs et calculateurs hétérogènes.

La capacité de configuration des circuits configurables dits FPGA permet de penser différemment l'implémentation d'une application, en décidant de partitionner cette dernière en tâches logicielles et matérielles sous forme de circuits spécialisés (généralement des boucles de calculs facilement parallélisables). Cela était déjà vrai avec des circuits câblés mais le coût de tels accélérateurs dédiés restait prohibitif.

Les fabriquants de ce type de puce configurable proposent des *soft-cores* : des processeurs à configurer sur ces puces, ayant la particularité de pouvoir être améliorés ou adaptés aux besoins en ajoutant facilement des fonctions de coprocesseurs intégrés (un bloc de cryptographie RSA, un décodeur vidéo etc.) ainsi que les instructions spécifiques associées pour y accéder. Cela permet d'envisager de modifier aussi les processeurs pour intégrer des services spécifiques de l'OS [KGJ03], comme des tables de gestion de processus, afin d'accélérer le temps de traitement dans les services de l'OS, ou bien pour créer des ports de communication dédiés aux messages entre OS, ou encore pour *matérialiser* le traitement de l'ordonnanceur pour des algorithmes lourds et complexes. Certains vont jusqu'à concevoir la totalité des services d'un RTOS en matériel [MMA07]. On se retrouve ainsi avec le besoin d'optimiser à la fois l'algorithmique des services du gestionnaire de plateforme, et aussi leur type d'implémentation, classiquement en logiciel ou maintenant en matériel.

5. OS pour Operating System, soit système d'exploitation logiciel

Par ailleurs, avec l'apparition des technologies des circuits dits *re-configurables* par partie et dynamiquement, l'exploration de l'architecture idéale prend une autre dimension et peut même advenir après avoir conçu le SoC, lors de l'exécution. En effet, la particularité d'un circuit reconfigurable est de pouvoir changer le câblage matériel des transistors, et cela n'importe quand, en configurant/déconfigurant dynamiquement les zones reconfigurables. Le concepteur peut ainsi décider de les arranger en un réseau de processeurs ou d'accélérateurs donné, et plus tard décider de configurer une autre fonctionnalité (temporairement) câblée nécessaire au fur et à mesure des besoins de l'exécution du système. Par ce procédé, la partie matérielle devient aussi flexible que le logiciel.

Ainsi ce n'est plus seulement le logiciel qui est dynamique, mais aussi sa plateforme d'exécution. Ces zones reconfigurables peuvent accueillir dynamiquement des fonctions accélératrices, et on peut apparenter ce fonctionnement à des tâches non plus logicielles, mais matérielles. Il est donc nécessaire de gérer convenablement ce type de tâches matérielles de la même manière qu'on a créé les OS logiciels pour gérer les différents flots d'exécution des tâches logicielles sur les processeurs. La notion ici de système d'exploitation reste cohérente avec la notion de gestion de ressources.

Enfin, cette capacité de reconfiguration dynamique ouvre d'importantes perspectives sur les choix d'implémentation matérielle de services d'OS et de tout ou partie de l'application mais, surtout, demande à concevoir de nouveaux services pour le support de tâches matérielles.

Ainsi, le système d'exploitation devient le cœur de la gestion des systèmes sur puces : il doit gérer à la fois des tâches logicielles, des tâches matérielles, de manière flexible, mais en plus doit être capable d'être lui-même (ses services) distribué sur de multiples nœuds de calculs (on parle de *scalabilité*).

Il est donc nécessaire de développer des systèmes d'exploitation capables de gérer cette dynamicité du support. Pour cela, il faut développer des outils et des méthodes qui incluent les OS dans le flot de conception de systèmes embarqués.

1.3 Problématique : Comment faciliter l'exploration architecturale d'un système ?

Aujourd'hui les OS sont utilisés pour gérer le système et pour faciliter le portage d'une application vers une autre plateforme. En effet l'OS propose une interface en partie générique qui fait abstraction de la couche matérielle et garantit la bonne gestion des ressources. Dans les SoC apparaissent de fortes contraintes sur les ressources disponibles (espace mémoire et mapping [6] des tâches, taille du code réduite au maximum, gestion au plus juste des périphériques) et des contraintes temps-réel : le temps de réponse de l'OS (synchronisation et communications) doit être optimisé.

Si certains aspects fonctionnels sont facilement garantis par construction (non interblocage, synchronisme, exclusion mutuelle etc.), en revanche, le comportement temporel (temps de réponse etc.) du même OS peut diverger d'un portage à l'autre selon l'architecture sous-jacente et donc modifier sensiblement le comportement global d'une application.

Ainsi, il est nécessaire d'anticiper les points critiques à modifier et revoir l'architecture de l'OS (sa distribution sur les nœuds de calcul) et ses fonctionnalités, et même éventuellement l'architecture matérielle en conséquence. Pour cela il est nécessaire de pouvoir évaluer les comportements des différents services d'un OS, de permettre d'en explorer les algorithmes et leur mise en œuvre (logicielle, matérielle ou mixte), notamment pour la gestion des zones reconfigurables.

6. répartition des tâches sur les différents nœuds d'exécution du système

1.3.1 Évaluer le comportement de la solution par simulation

Le comportement des applications portées sur une plateforme sur puce multi-processeur varie grandement selon l'OS utilisé. En effet, les paramètres d'ordonnancement et les services diffèrent selon les OS, ce qui peut changer le comportement final d'une application. Les services fournis par un OS sont de plusieurs ordres :

- **Abstraire le matériel et ses spécificités** : fournir une interface simplifiée pour garantir une utilisation unifiée et cohérente de nœuds d'exécution matériels hétérogènes (gestion des interruptions, de la mémoire et des périphériques).
- **Gérer le matériel** : allouer intelligemment des ressources comme la mémoire, les périphériques, et surtout l'accès au processeur comme ressource de calcul en simulant de multiples flux d'exécution (*time sharing* impliquant donc l'ordonnancement de processus) lorsque le processeur ne sait exécuter qu'une instruction à la fois (ou un nombre trop limité par rapport au nombre de flux).
- **Fournir un modèle de programmation (niveau logique)** : fournir et garantir des fonctionnalités de communication et synchronisation, ainsi que la sécurité.

Les OS peuvent être évalués succinctement en regardant leurs caractéristiques :

- préemptif ou non,
- politique d'ordonnancement (liste non exhaustive : Priority Based, Earliest Deadline First, Round Robin, MEDF),
- redéclenchement de l'ordonnanceur selon les services (bloquants ou non),
- latence de l'ordonnanceur, scalabilité et complexité algorithmique de son implémentation,
- latence du traitement des interruptions,
- comportement multi-processeur : répartition de charge avec possiblité de mettre en veille un processeur et donc de de migrer des tâches, ...

Les OS eCos [ocb] et Linux [oca], pour ne citer qu'eux, divergent dans leurs fonctionnements et leurs temps de réponse pour certains services identiques. Dans l'étude de Lubbers [LP08], les simples services de *lock* et *unlock* de *mutex*[7] prennent jusqu'à dix fois plus de temps sous Linux que sous eCos sur PowerPC. De plus, les OS utilisés sont en constante évolution, et rien ne garantit que la version $n + 1$ aura le même ordonnanceur ou se comportera de la même manière sur la nouvelle architecture support.

Mais les caractéristiques de l'OS ne sont pas les seuls facteurs de performances. Selon les systèmes (les processeurs modernes ont déjà des systèmes complexes intégrant en leur sein de nombreuses fonctionnalités autrefois externes), une partie des services d'OS a été déportée au niveau matériel (MMU[8], DMA[9], contrôleur d'interruption, mode superviseur, etc.), ce qui facilite grandement le travail de l'OS et améliore les performances globales. Certains services de partage de ressources, assez contraignants lorsqu'on utilise la virtualisation (hyperviseur), commencent à être intégrés à même le processeur (cas d'instructions dédiées pour les systèmes "virtualisés" [UNR+05]).

Tous ces facteurs de variation de performances montrent qu'il devient nécessaire d'intégrer les caractéristiques du système d'exploitation au plus tôt dans le flot de développement global d'un SoC, afin de décider du choix des services, et de leur implémentation potentiellement câblée au sein du processeur ou de la puce.

7. méchanisme d'exclusion mutuelle (généralement un sémaphore unaire)
8. Memory Managment Unit, ou bloc matériel de gestion de la mémoire
9. Direct Memory Access, coprocesseur matériel transférant des données de manière autonome

1.3.2 Explorer les services d'exploitation de plateforme pour concevoir les MPSoC

Dans la mesure où les systèmes-sur-puce intègrent aujourd'hui plusieurs cœurs de processeurs, mais aussi de nombreux nœuds d'exécution/accélérateurs spécifiques, la gestion cohérente du système devra être assurée par un système d'exploitation distribué.

La distribution multi processeurs et l'hétérogénéité des systèmes périphériques augmentent les besoins de services de communication, de synchronisation et de cohérence des données. Avec la distribution des tâches sur de multiples nœuds d'exécution, on voit (ré)apparaître des besoins comme la synchronisation entre tâches réparties sur des *multi-cores*, le partage de ressources mémoire (cache mémoire de niveau 3 commun sur les Dual Core© d'Intel, ou cache avec cohérence sur les MP-core CortexA9), ou encore les problèmes de répartition de charges qui peuvent évoluer dynamiquement.

Les problèmes de communication et de synchronisation deviennent même centraux, du fait que nombre de plateformes MPSoC sont constituées de multiples processeurs et accélérateurs hétérogènes (processeurs différents et zones reconfigurables). De surcroît il n'existe pas de standard pour la gestion des interactions entre les tâches logicielles et les accélérateurs câblés de fonctions de calcul dédié.

Les puces comprenant des centaines de processeurs sont déjà à l'étude (le processeur $TILE-Gx$ de Tilera comporte 100 cœurs de processeur[Til09]), et la gestion des ces *multi-cores* n'est pas évidente. La capacité des OS actuels à être déployés sur un aussi grand nombre de cœurs n'est pas triviale, ni certaine, car généralement on utilise un OS maître qui gère les autres. Or à cette échelle la centralisation de certains services n'est plus possible, car cela crée un goulet d'étranglement.

Afin de concevoir les nouveaux services d'OS spécifiques à la gestion de la distribution des tâches et afin de gérer la répartition des services, il est nécessaire d'intégrer le ou les OS tôt dans le flot de conception. L'idéal serait de les simuler et ainsi permettre de tester leur distribution aisément sur un grand nombre d'unités de calcul, pour des architectures encore inexistantes.

1.4 Objectif : Exploration architecturale à l'aide d'un modèle modulaire pour la co-simulation logicielle/-matérielle

Pour évaluer les comportements d'une application dans son environnement réel ou simulé, c'est-à-dire vérifier l'ordre d'exécution de ses tâches et sa réponse aux stimulus extérieurs, il est nécessaire de pouvoir modéliser à la fois l'architecture sous-jacente générant des variations des temps de réponse (défaut de cache par ex.) en fonction du contexte de l'exécution, mais aussi la couche d'abstraction de ce matériel et les services du système d'exploitation qui influencent aussi grandement ce comportement. Généralement, au moment de développer le code fonctionnel d'une application, la future plateforme dédiée est développée en parallèle (on parle de co-design) et n'est donc encore ni bien définie, ni *a priori* existante, bien qu'on cherche si possible à réutiliser un maximum de composants préexistants. Aussi, la modélisation et la simulation de cette plateforme dédiée conjointement à l'exécution de l'application permet d'évaluer si son dimensionnement est au moins suffisant, à défaut d'être optimal.

Dans ces travaux, nous allons montrer que pour explorer l'architecture idéale d'une plateforme embarquée dédiée à une application, mais aussi inventer et évaluer les services de gestion de la reconfiguration dynamique par zone, il faut disposer d'outils rapides de simulation, donc de haut niveau d'abstraction, permettant d'évaluer différentes combi-

naisons susceptibles d'être adéquates et d'ainsi vérifier que le système dédié remplira ses objectifs, préalablement à son implémentation.
Nous nous intéressons particulièrement aux applications *dynamiques*, dont le comportement varie en fonction du contexte d'exécution réel, des données en entrée(s) et/ou de(s) mission(s).
D'autre part, nous choisissons délibérément d'envisager l'exploration de l'implémentation d'une application en la partitionnant en logiciel mais aussi en matériel, car l'avènement de la logique reconfigurable à faible coût permet de confier les calculs lourds (et souvent réguliers) au matériel. On peut ainsi aussi envisager d'explorer la conception d'un système d'exploitation personnalisé dont les services sont implémentés alors aussi bien en logiciel qu'en matériel. Aussi, nous proposons de simuler à haut niveau tous les aspects dynamiques, notamment ceux liés aux services fournis par la plateforme (fournis généralement par l'OS), afin de faciliter l'évaluation et la validation du système, préalables à sa concrétisation.

Au cours de cette thèse, nous nous concentrerons sur la co-simulation logicielle/matérielle dans un même environnement, dans l'optique d'explorer l'architecture idéale, mais aussi dans l'optique de concevoir et explorer des services de gestion de zones reconfigurables et de services de distribution de tâches sur des nœuds de calcul hétérogènes. Pour cela nous allons construire un modèle permettant de composer facilement un système d'exploitation à base de services répartis et interconnectés et de simuler dans son ensemble un système dynamique multi-nœuds.

Nous distinguons deux axes pour permettre l'exploration des services d'un système d'exploitation d'une plateforme :

1. l'axe des comportements algorithmiques et temporels des services de gestion de la plateforme,
2. l'axe des implémentations de ces services, en matériel ou en logiciel, répartis ou non, optimisés, voire parallélisés.

A cette fin, la solution retenue dans cette étude est de percevoir et concevoir le système d'exploitation (SE) comme un gestionnaire de plateforme composé d'un ensemble de modules de services distribués que l'on peut modifier et interchanger. Ces services sont localisés et regroupés sous le terme d'OS associés aux nœuds d'exécutions, que ce soit des nœuds processeurs ou des nœuds matériels divers. Un SE est donc une collection d'OS, que ces derniers soient logiciels ou matériels.

La recherche de cette thèse se focalisera sur la conception d'un modèle unifié et générique permetant d'obtenir une simulation fonctionnelle et temporelle d'un système sur puce dans son intégralité, à savoir la partie logicielle comme matérielle, en incluant le système d'exploitation distribué. Cette possibilité est établie en s'appuyant sur le langage SystemC utilisé comme base pour permettre la simulation de tâches logicielles et d'OS conjointement à la simulation de parties matérielles distribuées.

Ce modèle générique est utilisable comme outil de prototypage rapide de système sur puce dédié à une application. Il permet de discriminer les différents choix possibles en facilitant l'exploration et l'évaluation conjointe des architectures adéquates du logiciel, du support matériel et du SE, par étapes successives de raffinement.

Nous allons démontrer que l'avantage notable de ce modèle générique est d'améliorer le développement de systèmes dédiés aux applications en facilitant leur parallélisation en de multiples nœuds d'exécution concurrents tout en différant l'étude approndie des détails de bas niveau architectural de l'implémentation.

De plus ce modèle générique permet de concevoir et d'explorer par simulation tout type de gestionnaires de plateforme (le SE) et donc potentiellement servira a évaluer l'efficacité de différents services spécifiques dédiés à la gestion de zones reconfigurables, dont

les problèmes d'ordonnancement spatial et temporel doivent être réglés.

Après cette présentation de notre problématique de la simulation conjointe logicielle/matérielle de systèmes sur puce pour l'exploration et la recherche de la meilleure adéquation Algorithme/Architecture/SE, le corps de ce document se divise en cinq nouveaux chapitres.

Le chapitre 2 propose une revue de l'état de l'art dans le domaine de la conception des systèmes sur puces, et des OS associés. Après les nombreuses références aux langages et environnements qui tentent de concilier la conception du logiciel et du matériel d'un système sur puce, on s'intéressera aux aspects liés à la modélisation à différents niveaux des systèmes logiciel-matériel conjointement à l'OS.

Le chapitre 3 introduit le premier élément de contribution de la thèse : la définition d'un modèle générique et modulaire de services de gestion de plateforme à haut niveau ainsi que la conception de mécanismes de simulation fonctionnelle du code logiciel et des services d'OS de manière réaliste conjointement au matériel, au sein du simulateur SystemC.

Le chapitre 4 présente le deuxième élément de contribution : la distribution de services d'OS sur de nombreux nœuds d'exécution d'un système facilitant les communications pour pouvoir évaluer rapidement la distribution des tâches, mais aussi la répartition des services associés. Ce modèle est dérivé de modèles utilisés dans l'informatique distribuée.

Le chapitre 5 présente les résultats obtenus lors du déploiement de ce modèle au sein de deux projets : OveRSoC et Ter@OPS.

Le projet OveRSoC [10] propose une méthodologie de conception adaptée à l'exploration et au partitionnement d'une application et de l'OS gérant une plateforme multi-nœuds reconfigurable, que nous avons employé pour l'exploration d'une architecture reconfigurable dédiée à une application de vision robotique, et qui nous a permis de valider le bon fonctionnement de notre modèle. Les travaux au sein du projet ont amené à la conception d'un outil facilitant l'exploration itérative des services et de la conception de simulateurs de plateforme. Nous avons également intégré un simulateur détaillé de processeur combiné au modèle d'OS de haut niveau pour montrer la capacité du modèle à fonctionner avec des niveaux de simulation/abstraction hétérogènes (logiciels et matériels). Enfin, nous avons expérimenté une modélisation hétérogène de processeurs et de zones reconfigurables pour explorer l'architecture de l'application de vision.

Ensuite nous décrivons le déploiement et l'insertion de notre modèle dans le cadre du projet Ter@OPS de conception d'un simulateur SytemC de calculateur embarqué haute performance à base de nœuds d'exécution hétérogènes et destiné à supporter différents domaines applicatifs.

Enfin, le dernier chapitre concluera cette thèse et nous y dégagerons les perspectives envisagées.

10. Outil de Validation et d'Exploration d'OS pour RSoC (SoC reconfigurables)

Chapitre 2

État de l'art

Sommaire

2.1 Introduction

Les avancées technologiques[1] ont permis d'intégrer de plus en plus de fonctionnalités dans un même circuit, ce qui a mené aux Systèmes-sur-Puce, qui sont aujourd'hui les composants principaux de nombreux systèmes (téléphone portable, radar, automobile, etc.). Les concepteurs de tels systèmes sont de plus en plus confrontés aux conflits entre les exigences de performances, de flexibilité, de consommation énergétique et de coûts.

Il est largement admis que les techniques de conception traditionnelles ne suffisent plus pour appréhender la complexité et la flexibilité de ces systèmes. Cela a amené à la notion de *conception système* dans laquelle des aspects tels que l'*architecture de plateforme*, la *séparation des préoccupations* et la *modélisation et simulation de haut niveau* jouent un rôle important. Ces systèmes embarqués imposent aux concepteurs de commencer par modéliser et simuler les composants du système et leurs interactions au plus tôt lors des premières étapes de développement.

La conception orientée plateforme (platform-based design) [VG01, SVM01] permet de réutiliser intensivement des blocs matériels (Intellectual Property, IP). Cette approche considère une plateforme comme commune et partagée pour un ensemble d'applications d'un domaine donné et s'accompagne de son lot de méthodes et outils de conception. Cela permet d'accroître les volumes de production et de réduire les coûts par rapport à la conception d'un circuit spécialisé pour chaque application.

Afin d'améliorer encore le potentiel de la réutilisation d'IP, et permettre l'exploration de solutions alternatives de conception, le principe de la *séparation des préoccupations* [KNRSV00] devient un élément indispensable de la conception au niveau système. Les deux principaux types de séparation sont :

1. séparer les calculs et les communications en connectant les blocs IP des calculs par des interfaces standards et
2. séparer l'application et l'architecture (ce que fait le système, différent du comment).

Par ailleurs, les processus de conception au niveau système incitent à commencer par modéliser et simuler les composants systèmes et leurs interactions au plus tôt dans le flot de conception [EPTP07]. De tels modèles de systèmes représentent généralement le comportement de l'application, les caractéristiques de l'architecture et leurs relations (mapping, répartition matérielle/logicielle). Ces modèles sont des abstractions de haut niveau, ce qui minimise les efforts de modélisation et optimise la vitesse de simulation pour permettre une évaluation rapide lors des premières phases de conception. Ces modèles abstraits permettent en général de vérifier une réalisation en fournissant des estimations sur les performances, la consommation et le coût d'un design.

Ces aspects de la conception système doivent être accompagnés de méthodologies de conception efficaces et intelligentes permettant d'explorer l'espace de conception [MD05]. Pour appréhender la complexité de l'exploration, il est nécessaire d'avoir des outils d'évaluation performants, surtout au début d'un projet quand l'univers de choix est le plus grand.

Nous allons dans ce chapitre observer l'état actuel des systèmes temps-réel embarqués, de leurs architectures et des systèmes de gestion de ces plateformes généralement conçues avec un système d'exploitation. Ensuite nous nous attarderons sur les méthodes et flots de conception de ces systèmes, puis sur la définition de la modélisation de niveau système et des techniques employées. Enfin, nous passerons en revue les modèles et simulateurs de la littérature permettant une modélisation conjointe des aspects matériels et logiciels des systèmes sur puce. Nous étudierons particulièrement les techniques de simulation

1. confirmées par la loi empirique de Moore

permettant de valider un système modélisé dans son intégralité, c'est-à-dire qui permettent de modéliser le système d'exploitation en plus du logiciel et de l'architecture matérielle.

2.2 Étude des systèmes temps-réel embarqués

Les systèmes électroniques embarqués sont caractérisés par leurs interactions avec leur environnement et comportent différents capteurs et interfaces spécifiques à leur fonction. Ils comportent parfois une interface avec l'utilisateur. La structure de base d'un tel système se décompose selon le schéma simplifié de la figure 2.1 : divers capteurs appréhendent une partie de l'environnement. Ces informations sont numérisées par échantillonnage puis traitées par le(s) calculateur(s) au cœur du système. Ce dernier convertit les résultats de traitement en signaux analogiques générant des actions sur l'environnement. L'activation du frein ABS d'une voiture en est un exemple.

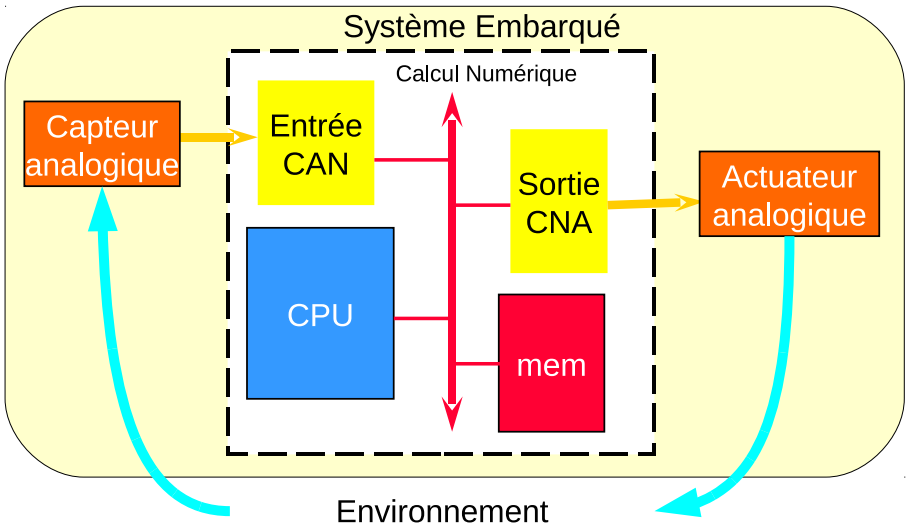


Figure 2.1: *Schéma de principe d'un système embarqué à base de calculateur numérique*

Ces systèmes selon leur type d'application doivent répondre à des contraintes particulières qui les caractérisent par rapport à d'autres systèmes électroniques :

- l'encombrement : ils doivent être de taille réduite pour être transportés et intégrés au plus proche d'un mécanisme ou d'un capteur. C'est la définition même du terme *embarqué*.
- la consommation : la conséquence de l'embarquement signifie que l'accès aux ressources énergétiques risque d'être fort contraint (batterie lourde et chère), par conséquent il est nécessaire qu'ils consomment le moins possible, pour prolonger leur autonomie.
- les délais : le facteur temps est une composante importante, car les actions/réactions à mettre en œuvre sont souvent fortement contraintes. Un système ABS de voiture doit réagir en continu (boucle d'asservissement) dans un délai infèrieur à la microseconde. On parle alors de contraintes temps-réel, lorsque les traitements doivent être réalisés dans un délai maximum (et parfois minimum), afin de de garantir la validité/pertinence des sorties vis-à vis de l'évolution des entrées.
- la sécurité : les applications employant ce genre de systèmes ne tolèrent généralement ni panne ni erreur de calcul, sous peine de catastrophe. On ne peut pas tolérer qu'un système ABS de voiture se bloque en plein freinage. Dans l'aéronautique, des petites erreurs de calculs (suite à une conversion de 32 vers 16 bits d'un grand nombre : troncature destructrice des bits de poids fort significatifs [ari]) ont mené à l'explosion de la fusée Ariane V.

- le coût : les systèmes embarqués ne sont pas toujours destinés à la production de masse (excepté dans l'automobile et les télécommunications) mais produits au cas-par-cas en fonction de chaque application, avec des coûts unitaires très élevés, ce qui nécessite d'arbitrer le choix de l'architecture optimale. En effet, les coûts des technologies actuelles de gravure de plaques de silicium rendent quasiment rédhibitoire la conception de circuit dédiés pour des quantités inférieures à la dizaine de milliers. Ce phénomène est de plus en plus pallié par l'utilisation de circuit généraliste configurable de type FPGA.
- la longévité des produits et donc de leur partie électronique : en logique économique et industrielle il est moins cher de racheter un nouveau système électronique que de le réparer, du fait de l'obsolescence rapide des composants électroniques. En effet, l'évolution des technologies rend très coûteux de recréer la même carte/circuit 3 ou 4 ans plus tard, et dans ce cas il est souvent nécessaire de reconcevoir tout le système.
- la résistance à l'environnement : selon les applications il faut garantir leur fonctionnement au pôle nord ou au milieu du désert coté température, leur résistance à l'humidité, au choc (si mobiles), à l'accélération (dans les avions supersoniques), aux rayonnements cosmiques (cause de nombreuses erreurs dans l'aéronautique, particulièrement les satellites), et simplement aux rayonnements électromagnétiques.

Nous allons maintenant suivre l'évolution des systèmes embarqués amenant à la complexité actuelle. L'accroissement progressif des contraintes prises en compte par ces systèmes embarqués amène à une complexité nouvelle, pour laquelle il manque encore des outils adaptés.

2.2.1 Historique de l'évolution des systèmes électroniques

Le développement architectural des systèmes électroniques suit selon Makimoto [Mak00] des cycles, comme indiqué sur cette représentation en figure 2.2. La dernière vague technologique annoncée par Makimoto est celle des systèmes reconfigurables. On peut noter

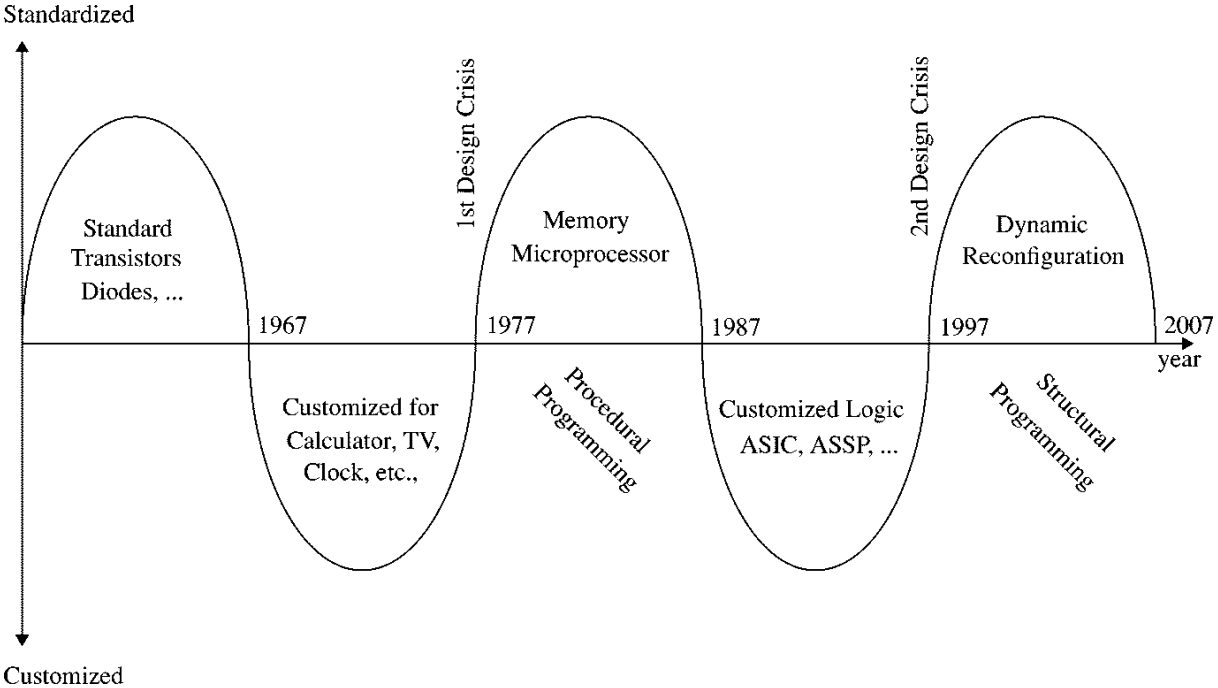


Figure 2.2: *Vagues architecturales selon T. Makimoto prévue en 2000 [Mak00]*

que la dernière prédiction présentée avec les composants reconfigurables ne s'est toujours pas refermée en 2011 sur une standardisation.

Tredennick [Tre95] décrit l'évolution de la micro-électronique au premier stade comme étant à ressource et algorithme fixes, puis l'apparition des processeurs avec la partie algorithmique variable (paradigme *Von Neumann*) et enfin l'actuelle évolution des ressources maintenant variables en plus de l'algorithmique grâce à la reconfiguration matérielle. Tredennick rejoint Makimoto dans cette remise en cause du paradigme de programmation classique *Von Neumann*, en ce sens que les systèmes configurables sont basés sur les flux de données et n'ont plus besoin d'instructions. Ce changement de paradigme est même approfondi dans les travaux de Becker et Hartenstein [Har01, BH03] qui introduisent les termes de *Configware* et *Morphware* pour le décrire.

Cela nous amène à présenter une certaine classification des systèmes embarqués, qui correspond plus ou moins aussi aux générations successives de systèmes matériels embarqués, du fait de l'évolution conjointe des technologies et des besoins applicatifs, qui vont du simple système aux *many-cores* reconfigurables.

2.2.1.1 Les systèmes embarqués simples

Partie matérielle de ces systèmes simples Les systèmes embarqués de première génération étaient assez simples : un processeur unique contrôle un nombre restreint de périphériques, généralement d'entrée/sortie, vers de simples gestionnaires analogiques de capteurs ou d'actionneurs. Ces périphériques pouvant commencer à se complexifier avec le développement des premiers *circuits intégrés à application spécifique* (ASIC).

Le processeur est un simple calculateur, qui par ailleurs contrôle l'ensemble du système. La simplicité d'une telle architecture limite les communications au bus de processeur, généralement de type maître/esclave où le processeur joue le rôle du maître et les périphériques se contentent de recevoir et exécuter les données/ordres reçus. Ce genre de fonctionalités simples sont aujourd'hui entièrement intégrées dans les puces appellées microcontrôleurs.

Partie logicielle de ces systèmes simples La puissance de calcul et le nombre de ressources étant encore peut élevé, le programme logiciel est généralement monolithique. La réaction aux évènements est quant à elle gérée généralement par des routines de traitement d'interruptions. pour garantir son efficacité et sa faible empreinte mémoire, Pour l'efficacité et la faible empreinte mémoire, on codait le plus souvent directement en langage assembleur, ce qui est de plus en plus rare de nos jours, grâce aux compilateurs de langages de plus haut niveau de plus en plus efficaces.

2.2.1.2 Les systèmes embarqués distribués

Partie matérielle des systèmes embarqués numériques Les besoins applicatifs ne cessant d'évoluer ont nécessité la création de systèmes offrant de plus en plus de puissance de calcul, comme le réclament notamment les applications de traitement non plus analogique mais numérique du signal.

L'architecture des systèmes plus complexes se compose, en plus d'une multitude de périphériques divers, d'un processeur central contrôlant des processeurs annexes. Le système reste souvent supervisé par le processeur principal. Les processeurs périphériques servent aux tâches de calcul intensif et sont souvent spécialisés, comme les processeurs de traitement du signal (DSP).

Cette architecture (cf. figure 2.3) nécessite de disposer d'un bus propre par processeur, chacun étant relié aux autres par des ponts pour leur permettre de communiquer via la mémoire centrale. Les communications entre les différentes puces assemblées sur le même circuit imprimé multicouche empruntent des pistes de grande longueur (comme la carte mère des PC), altérant l'intégrité des signaux à haute fréquence, ce qui est peu compatibles

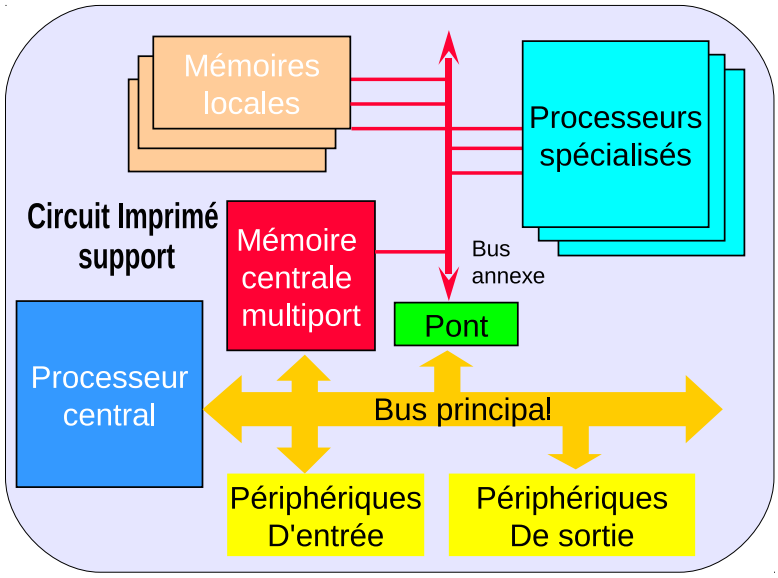


Figure 2.3: *Circuit imprimé connectant de nombreuses puces*

avec les contraintes de consommation et de vitesse. Il est donc devenu nécessaire de rapprocher les composants au mieux, voire si possible dans la même puce.

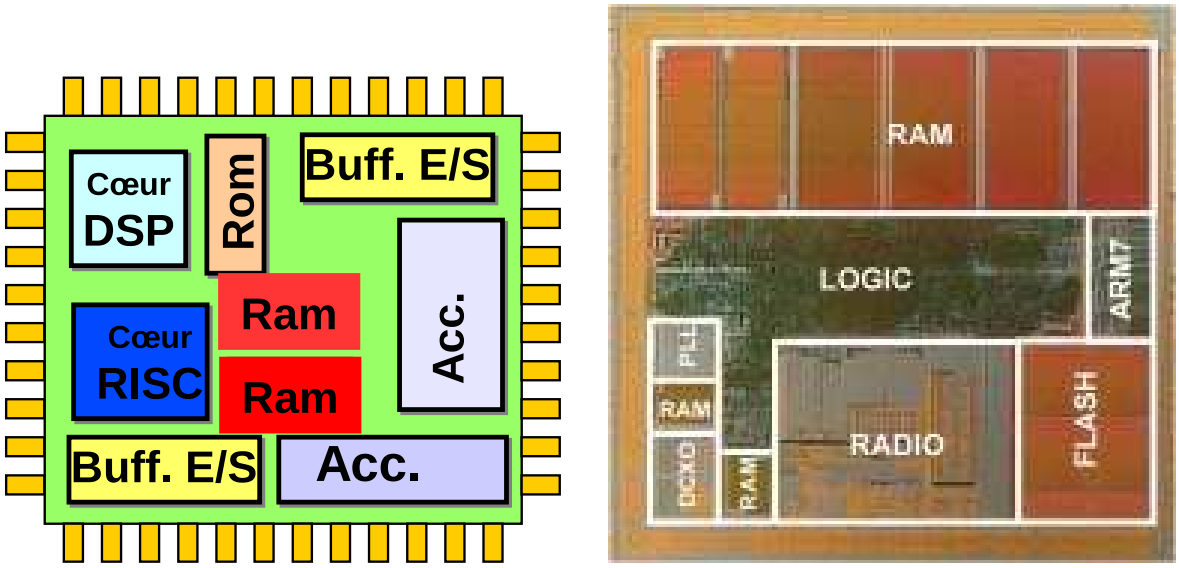
Partie logicielle des systèmes embarqué multi-calculateurs La complexité de ces architectures ne peut être gérée par un seul programme sur le processeur central. Le logiciel embarqué est donc réparti sur plusieurs processeurs (le principal et les annexes) ou dans des accélérateurs cablés. Il devient nécessaire de gérer plusieurs tâches, réparties sur ces différents processeurs hétérogènes. Les systèmes d'exploitation sont donc adjoints aux logiciels applicatifs pour gérer cette complexité liée à la distribution.

Les OS ont pu être développés conjointement à l'apparition de langages de plus haut niveau comme le langage C, encore aujourd'hui largement utilisé dans ce domaine. En revanche, les processeurs périphériques sont parfois encore programmés en assembleur.

2.2.1.3 L'intégration dans une même puce : les systèmes-sur-puce

La loi de Moore stipulant le doublement du nombre des transistors par puce tous les 18 mois s'étant révélée valable empiriquement, on peut aujourd'hui fondre sur la même puce l'ensemble des composants d'une carte (cf. Figure 2.4(b)). Ainsi il est techniquement possible d'intégrer toutes les fonctions (ou presque) d'un système embarqué, hormis quelques composants analogiques comme les cristaux de quartz générant les horloges. Les technologies ont même tellement évolué que des composants périphériques analogiques (des antennes, des capteurs de position, de températures etc.) voire mécaniques (MEMs) ou opto-électronique peuvent être intégrés dans le silicium. On parle alors de systèmes hétérogènes au sens de l'intégration dans la même puce des composants numériques et non numériques (analogiques, mécaniques, ...). Les technologies de gravures ou le type de substrat silicium étant assez différentes, certains fabricants conçoivent maintenant les puces par empilement de puces de différentes technologies interconnectées dans le même boîtier, technique nommée SiP (System in Package). Nous ne nous intéresserons pour notre part qu'à la partie numérique.

Partie matérielle des SoC La demande en puissance de calcul et en flexibilité a amené à concevoir des systèmes comportant de plus en plus de processeurs. On peut qualifier ces architectures de multi maîtres/multi esclaves, car la centralisation de la gestion ne peut plus être supportée par un seul processeur principal.



(a) Architecture Typique de SoC (b) Photos d'un SoC bluetooth de Alcatel

Figure 2.4: *Schéma et réalisation de SoC*

La complexité de tels systèmes fait que l'on ne recommence pas à zéro le développement de chaque nouveau système sur puce, mais que l'on procède plutôt par assemblage de composants déjà créés et testés. On regroupe généralement sous le terme de *propriété intellectuelle* ou IP [2] ces blocs fonctionnels ainsi assemblés. Cela peut être un simple processeur comme une fonction câblée d'accélération d'un algorithme particulier, comme un codeur/décodeur de fichier MP3, un bloc de cryptage/décryptage de données transmises, ou une transformée de Fourrier inverse.

La puissance de calcul étant répartie sur de nombreux processeurs, elle suffit généralement aux besoins des applications. En revanche, le goulet d'étranglement se situe maintenant au niveau des communications mémoire. Ces communications sont les facteurs qui décident souvent de l'architecture envisagée. De nombreuses formes d'architecture de communication sur puce (multi-bus, barres connectées, réseau de communication) sont développées pour améliorer les capacités des circuits. Le principal défi est d'interfacer les IP en fonction des protocoles d'échanges désirés, selon le ou les bus ou réseaux sur puce (NoC [3]) utilisés et ceux des IP. En général on doit rajouter des transacteurs (wrapper) entre les différents protocoles des bus et ceux particuliers des IP, comme on le fait avec des ponts entre bus distincts.

L'architecture des calculateurs et processeurs doit être définie en fonction des besoins spécifiques de chaque partie de l'application et l'évaluation de la configuration optimale peut être fastidieuse. Par ailleurs, l'architecture mémoire joue un rôle déterminant dans les charges de transfert de données entre les processeurs, les mémoires, et les périphériques ; il faut décider leur localisation en fonction de leur partage ou non par un ensemble de processeurs. On utilise par exemple de plus en plus souvent des mémoires double port, capables d'être lues ou écrites par deux processeurs simultanément, ainsi que des blocs DMA [4] permettant de décharger les processeurs des transferts de données, en leur indiquant uniquement la fin de ces derniers par interruption.

Partie logicielle des SoC La part du logiciel dans ces architectures est de plus en plus importante, car il apporte de la flexibilité aux plateformes, et bien évidemment

2. Intelectual Property ou fonction matérielle câblée qui généralement s'achète
3. Network-on-Chip
4. Direct Memory Access

participe à la gestion des ressources distribuées. De plus il devient nécessaire d'utiliser des systèmes d'exploitation afin de masquer la complexité du système au programmeur. Cette complexité est ainsi reportée au développement des couches support que sont l'OS et ses pilotes de périphériques.

Les OS sont alors déclinés selon les besoins en un ou plusieurs OS coopérant entre eux. Cette dernière solution est la plus répandue du fait de l'hétérogénéité des processeurs présents dans le même design mais elle est plus complexe à mettre en œuvre, si on veut utiliser leurs pleines capacités spécifiques.

La répartition des tâches logicielles entre les différents processeurs et son adaptation continue pendant l'exécution devient aussi un problème complexe, que les OS commencent à prendre en charge. Pour des raisons d'efficacité ou de consommation électrique, on peut vouloir *migrer* une tâche d'un processeur à un autre, afin de changer la configuration de la charge par processeur et éventuellement mettre les cœurs inutilisés en veille.

2.2.1.4 La génération actuelle : la flexibilité

De nouveaux systèmes sont apparus : les composants configurables et plus récemment reconfigurables. Ce sont des matrices de transistors/portes logiques préarchitecturées pour que les interconnexions puissent être modifiées après la fonderie (une seule fois ou à souhait) pour spécialiser la fonction désirée, tout en abaissant les coûts unitaires grâce aux grandes séries de production.

Partie matérielle flexible On regroupe sous le terme de PLD (pour *Programmable Logic Device*) ou FPGA (pour *Field Programable Gates Array*), ces circuits particuliers composés d'une matrice de portes logiques interconnectées par des connexions programmables. Les FPGA récents permettent même de modifier pendant l'exécution la configuration d'une sous-partie du système, ce qui augmente encore le degré de flexibilité matérielle envisageable.

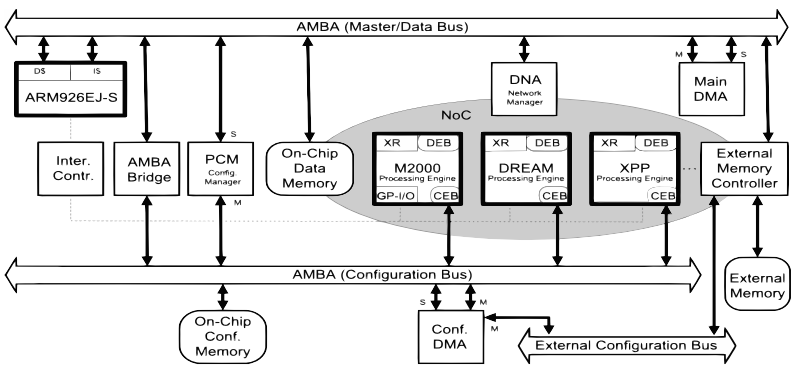
Il est ainsi devenu possible de concevoir n'importe quelle fonction numérique désirée, et de l'implémenter dans ce type de circuit reprogrammable standard, évitant ainsi une coûteuse fonderie microélectronique, au prix d'un surcroît de consommation du fait de la surface de la logique aditionelle nécessaire. L'avantage est de pouvoir modifier facilement le design pour suivre l'évolution du produit, ou simplement pour le corriger après production. Il suffit pour cela de charger le code particulier indiquant les connexions désirées, appelé *bitstream*, aussi simplement que l'on charge un programme logiciel dans la mémoire d'un processeur.

Ainsi, non seulement il est aisé de concevoir son propre accélérateur de calcul pour un coût modique, mais en plus il est possible de définir soit-même l'architecture des processeurs et IP de son design. Cela permet notamment de les spécialiser en leur rajoutant des fonctions dites co-processeurs, avec les instructions spécifiques d'appel. Par exemple, il peut être utile d'ajouter une opération *racine carrée* calculant en un ou deux cycles d'horloge le résultat, au prix d'une surface matériellle plus importante, plutôt que de dérouler séquentiellement une longue suite d'instructions d'un algorithme de racine carrée basé uniquement sur les opérations élémentaires d'additions/soustractions et multiplications/divisions entières. Il est ainsi possible d'optimiser un compromis entre performance et utilisation de ressources. Un inconvénient des systèmes reconfigurables est la lenteur des fonctions ainsi implémentées comparés à une gravure dans un ASIC standard, du fait du rallongement de fils d'interconnexion, liés à la présence des éléments de reconfiguration, impliquant par ailleurs un surcoût silicium (on compte souvent un facteur 10 en fréquence et en nombre de transistors par rapport à un ASIC).

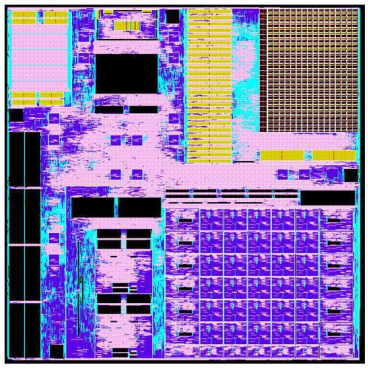
Les outils pour FPGA étant suffisamment évolués, il devient presque aussi facile de

concevoir un bloc matériel (IP) et le système dans son entier que lors de la conception d'un SoC normal, car la vérification du design est moins critique.

De plus en plus de SoC récents comportent un espace reconfigurable en leur sein afin que l'utilisateur final puisse y ajouter des fonctionnalités non prévues de manière câblée en dur. La figure 2.5 montre une réalisation d'un SoC Reconfigurable (RSoC) offrant trois types de zones reconfigurables différentes (différentes granularités), développé par les partenaires du projet de recherche européen MORPHEUS [UE] et dirigé par l'équipe du laboratoire Thales TRT/LSE.



(a) Architecture du R-SoC Morpheus, contennant trois types de granularités de zones reconfigurables : M2000, DREAM et XPP



(b) Photo du SoC Reconfigurable multi-grain Morpheus

Figure 2.5: *Schéma et réalisation du SoC reconfigurable Morpheus [UE]*

Gestion de cette flexibilité La problématique de gestion des parties flexibles que sont les logiciels embarqués se trouve additionnée de la gestion des zones reconfigurables. En effet, les différentes IP doivent être gérées pour prendre en compte le temps de chargement de leur *bitstream* et leur localisation dans l'espace configurable. La capacité de certains composants FPGA d'être reconfigurables par partie permet aussi d'envisager de charger une IP à un instant donné, puis de réutiliser cet espace ultérieurement pour y loger une autre IP. Cela ressemble a s'y méprendre à la gestion du partage du temps processeur pour des tâches logicielles, et la gestion des chargements des données et instructions.

Afin de démocratiser leur utilisation, des OS dédiés doivent être développés pour gérer ce type de ressources, notamment des zones reconfigurables par partie.

Par ailleurs, étant donné que ces composants offrent une liberté sur l'architecture souhaitée, il devient aussi simplement possible de personnaliser les processeurs pour leur ajouter des fonctionnalités câblées spécifiques (permettant d'accélérer des bloc de calcul, ou certaines fonctions des RTOS [IM00, KGJ03, KSM03, WP03]) ou des couches de communication supérieures des middlewares. Le partitionnement d'une application entre différents nœuds de calcul se voit donc ainsi complexifié par la potentialité de créer des accélérateurs de calcul matériels dédiés (IP) pour les parties les plus régulières et/ou gourmandes en temps de calcul plutôt que de les laisser s'exécuter séquentiellement sur un processeur.

2.2.2 Modèle conceptuel simple actuel

Le modèle le plus simple utilisé actuellement pour organiser un système numérique (pas nécessairement sur puce) est une vue en couches, chacune correspondant à un niveau d'abstraction (figure 2.6) et comprenant plusieurs parties. Ce modèle permet de schématiser la complexité d'une plateforme en traçant une frontière entre "matériel" et "logiciel".

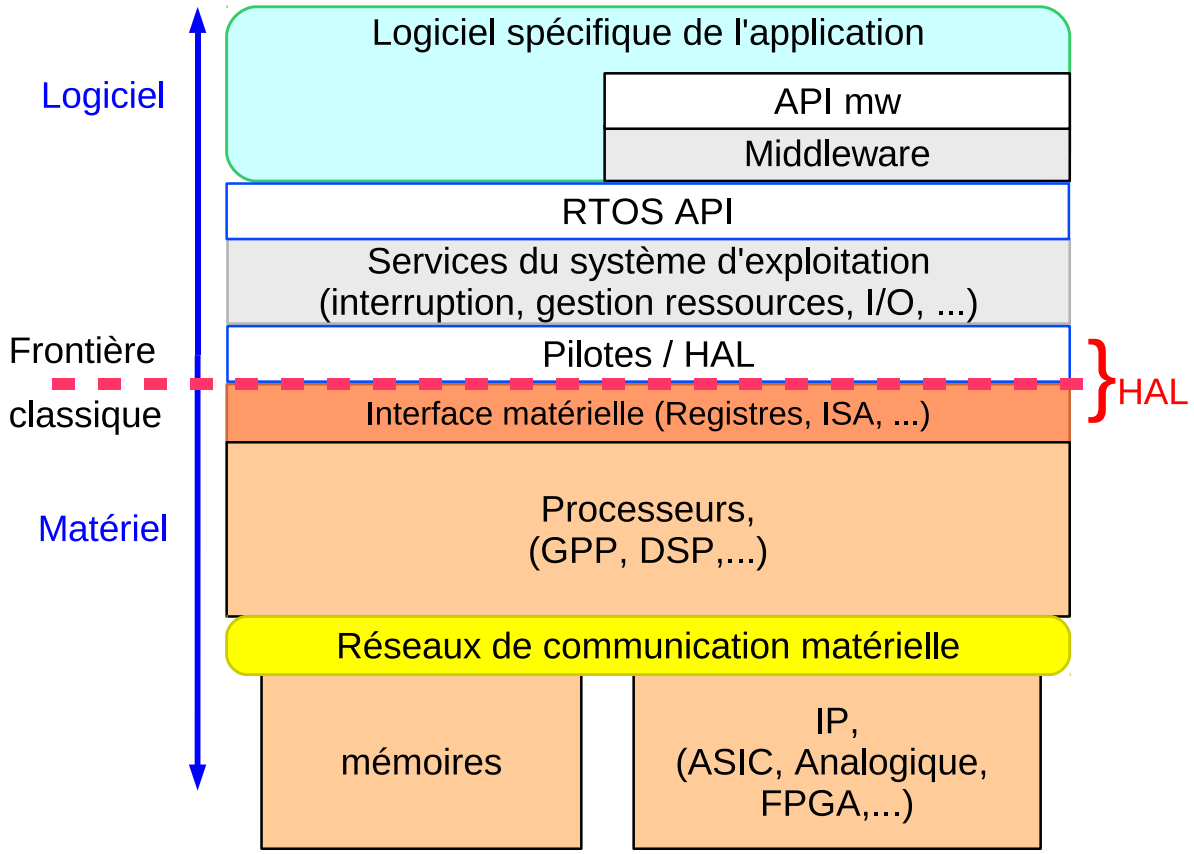


Figure 2.6: *Vue en couches d'un système numérique*

La partie matérielle se décompose en trois parties : une couche basse, une couche de communication et l'ISA [5].

- La couche basse contient les principaux composants matériels du système. Ce sont généralement des composants standards (processeurs, DSP [6], blocs accélérateurs et mémoire), mais aussi depuis plus récemment des zones reconfigurables.

- La couche de communication matérielle embarquée sur la puce. Elle peut comprendre un réseau de communication complexe allant du simple pont entre deux bus de processeurs jusqu'au véritable réseau sur puce à commutation de paquets. Il est parfois nécessaire d'ajouter des couches d'adaptation entre les composants et les bus de communication, malgré leur standardisation croissante.

- L'accès au matériel passe par l'intermédiaire des instructions machine correspondant au processeur (l'ISA). L'ISA décrit aussi la manière de se servir des registres d'un processeur, de sa pile (pour les processeurs ARM, les registres R0-R12 sont à usage général, R13 est le pointeur de pile SP, etc.) des registres spécifiques de "périphériques" intra-processeur (R16 registres de Status, contrôleur IRQ, ...) et de ports de communication accessibles localement ainsi que tout ce qui permet de contrôler des périphériques. Les compilateurs sont conçu pour traduire automatiquement les langages de haut niveau du programmeur en langage machine correspondant l'ISA du processeur ciblé.

Toute les applications logicielles reposent *in fine* sur le code des instructions machine correspondant au processeur : l'ISA. Elles sont écrites dans des langages de programmation

5. Instruction Set Architecture, l'inventaire et le codage des instructions
6. Digital Signal Processors

de plus haut niveau qui, associés à un compilateur, permettent à l'application de rester portable (l'ISA pour les opérations de calcul) et de respecter l'ABI [7] (conventions de bas niveau pour gérer les appels de fonctions comme la pile et les types des paramètres, et les appels systèmes [PH97]). Mais une application s'appuie aussi sur un ensemble d'interfaces de programmation ou API [8] : la HAL, l'API du système d'exploitation (OS [9]) mais aussi l'API des librairies spécifiques ou *middleware* de son choix. Généralement, cela facilite la réutilisabilité des fonctions déjà développées, ainsi que la portabilité sur d'autres supports matériels, en ne se préoccupant que du code métier si l'API fournie remplit bien son rôle quelle que soit la plateforme.

La partie logicielle supportant une application se découpe donc en plusieurs couches [TG98] : HAL, couche OS, sur-couche des librairies et middleware, toutes reposant *in fine* sur le code des instructions machine.

- L'HAL (Hardware Abstraction Layer) : le lien avec le matériel passe par des contrôleurs (logiciels) de périphériques matériels et d'entrées/sorties spécifiques que l'on appelle *drivers* (ou *pilotes*) qui permettent à tout logiciel d'accéder aux ressources matérielles. La HAL est donc la couche logicielle la plus basse qui, en gérant les spécificités du matériel, libère le reste du logiciel (services de l'OS et application) des dépendances à l'architecture. Ces pilotes forment la couche personnalisée d'un OS adapté à (on dit aussi *porté sur*) un processeur donné. On associe aussi le terme de HAL aux fonctions très bas niveau des OS comme les primitives de sauvegarde des registres ou de gestion des interruptions.

- La couche qui gère les ressources de l'architecture et des besoins de plus haut niveau : cette couche est généralement appelée système d'exploitation ou OS (pour *Operating System*). L'OS repose sur l'interface d'abstraction du processeur (l'HAL), bien que l'on puisse considérer que la HAL fait partie de l'OS. On regroupe par le terme de *BSP* (Board Support Package) l'ensemble des logiciels et intefaces de base accessibles au programmeur, à savoir l'ISA, la HAL et l'API de l'OS et ses drivers, mais aussi le bootloader (ou BIOS), le logiciel de démarage d'un système.
  En séparant l'application de l'architecture, l'OS permet d'adapter les particularités du matériel à n'importe quelle application. Il lui fournit les fonctions adaptées à l'architecture matérielle spécifique sous-jacente et permet d'utiliser le potentiel du système à son maximum et de manière transparente, en répartissant les ressources dans un souci d'efficacité.
  Cette même recherche d'efficacité a cependant modifié cette séparation entre architecture et application, en répartissant autrement les rôles entre matériel et logiciel. Pour soulager l'OS des fonctions de gestion de la mémoire et des registres intermédiaires qui représentent une charge de calcul non négligeable, des contrôleurs matériels spécifiques (MMU,DMA,registres pointeurs de pile, fenêtres de registres dans les processeurs SPARC ...) ont été adjoints aux processeurs.
  Il en va de même du contrôle matériel du niveau de droits d'exécution de certaines instructions réservées à l'OS et aux pilotes (mode superviseur), comme la gestion des interruptions.
  Enfin, la possibilité d'implémenter de nombreuses fonctionnalités de l'OS (comme l'ordonnanceur) sous forme matérielle, grâce notamment aux FPGA, permet d'obtenir des accélérations importantes (jusqu'à 50 % selon [LMD+03]).

7. Application Binary Interface
8. Application Programming Interface
9. OS pour Operating System, soit système d'exploitation logiciel, à distinguer ici du SE qui englobe tous les OS logiciels et les blocs matériels de gestion d'un MPSoC

Ainsi le matériel originellement cantonné au rôle de simple automate [Tur50] d'exécution et d'accélération de calcul est maintenant chargé aussi de fonctions de contrôle et d'exploitation originellement implémentées par le système d'exploitation ou l'application. Cette nouvelle répartition des rôles indique que la frontière entre le logiciel et le matériel n'est pas tout à fait aussi rectiligne qu'on pourrait le croire. C'est cette brèche dans la frontière logiciel/matériel de gestion de plateforme que nous nous proposons d'explorer plus loin.

– La sur-couche des librairies et middleware. Appuyée sur les services de l'OS, elle propose des fonctionnalités très spécifiques au domaine de l'application (protocoles de communication (stack internet IP...), indépendance vis-à-vis du déploiement (CORBA), etc.). Elle n'est pas indispensable mais souvent utile pour faciliter le développement d'applications, surtout dans un souci de réutiliser les fonctions déjà testées et éprouvées (*software design reuse*), et aussi pour fournir une abstraction de la plateforme de plus haut niveau que celle de l'OS.

2.2.2.1 Influence de la représentation sur la création

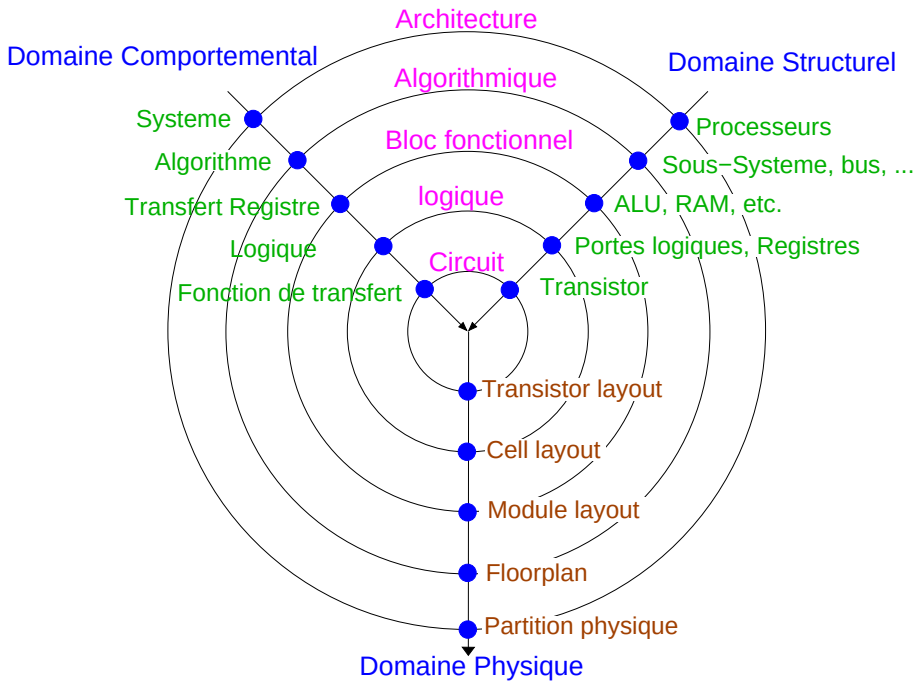


Figure 2.7: *Schéma en Y de conception conjointe de Gajski-Kuhn [GK83]*

Le processus de conception tend à suivre cette représentation en couches, en séparant les acteurs de la conception du logiciel et du matériel. Le matériel est ainsi conçu sur mesure, tant pour les composants de calcul que pour le réseau d'interconnexion. Le concepteur de circuit ne doit fournir aux développeurs logiciels qu'une mince couche de logiciel suffisante pour l'accés aux ressources matérielles. Cette démarche permet de concevoir indépendamment les parties logicielles et matérielles, en parallèle (pour accélérer la conception), jusqu'à leur l'intégration (selon le modèle de co-développement en Y, cf. figure 2.7).

Cette conception en Y implique trois étapes de vérification : chaque couche doit d'abord être vérifiée indépendamment puis conjointement lors de l'assemblage final. Seule l'intégration finale de toutes les couches, matérielles et logicielles, permet de contrôler le

bon fonctionnement du système dans son ensemble. Ce n'est donc généralement qu'à ce dernier stade que se révèlent certains dysfonctionnements majeurs (taille de la mémoire insuffisante, contraintes temps-réel non satisfaites, etc.) ou une sous utilisation du système souvent sur-dimensionné (faute de compromis judicieux), ce qui signifie souvent une surconsommation énergétique, un surcoût silicium ou une nouvelle itération de conception (coûteuse) pour y remédier.

2.2.2.2 Modularité pour répondre aux contraintes de flexibilité

Afin de faciliter la conception de systèmes complexes, chaque couche est conçue comme un assemblage de composants (sur étagère ou nouvellement conçus). Une telle séparation est nécessaire pour introduire de la flexibilité et de la modularité dans l'architecture matérielle et logicielle. Ainsi la conception d'un système consiste surtout à assembler/architecturer des composants pour respecter des contraintes de performances et de délai/coût du système global et non plus seulement au niveau du composant. Au vu du nombre de possibilités offertes par l'ensemble quasiment infini de l'espace d'exploration architectural, la recherche de l'*Adéquation Algorithme Architecture* (A^3 [Sor94]) se voit ainsi facilitée si les composants logiciels mais surtout matériels peuvent être remplacés au gré des besoins à partir du moment où ils offrent une interface équivalente.

Cette modularité facilite l'exploration et la réutilisabilité des composants mais nécessite une abstraction nouvelle : celle de leurs interfaces de communication. Cette abstraction des interfaces de communication devient aujourd'hui le pivot de la conception des SoC. En effet il devient crucial de bien coordonner la conception des couches de communication intra-logicielles et intra-matérielles. Car si cette évolution permet de séparer totalement la conception des couches de communication et celle des composants logiciels et matériels, l'abstraction des interfaces de communication (et leur standardisation) devient alors le pivot de la conception des systèmes-sur-puces.

Par ailleurs, la décomposition d'un système en composants et l'abstraction des communications sous forme d'interfaces entre les différentes couches et composants permettent aussi de décrire les éléments à des niveaux différents les uns des autres et surtout autorisent l'usage d'un niveau plus élevé d'abstraction. On peut ainsi décrire au niveau comportemental (une partie des) composants et interfaces, ce qui est généralement le point de départ d'une spécification d'application. Cela permet potentiellement de simuler rapidement le système dans son ensemble et de vérifier sa validité en mixant composants existants (de bas niveau donc) et nouveaux (spécification de haut niveau seulement). Ces modèles de haut niveau devraient à terme aussi permettre de générer automatiquement les codes du matériel et du logiciel, bien que cette automatisation ne soit pas encore réaliste.

Ce schéma en Y (figure 2.7) séquentiel et modulaire de conception séparée du matériel et du logiciel n'est cependant plus applicable à partir d'un certain degré de complexité. Lorsqu'un système est utilisé pour une tâche très précise, il est souvent plus efficace et économe s'il est spécifique à cette fonctionnalité que s'il a été conçu à partir de composants génériques. Les systèmes embarqués sont très souvent utilisés dans ces conditions, et il est donc intéressant qu'ils soient conçus spécifiquement pour les fonctions qu'ils doivent remplir. Généralement, les contraintes temps-réel de tels systèmes ne peuvent être respectées que si le système est conçu dès le départ pour les respecter.

2.2.3 Conception et exploration conjointe de MPSoC

La grande difficulté de la conception d'un système sur puce découle de la nécessité de synthétiser les connaissances du domaine de la conception du matériel avec celles de

la partie logicielle qui est par essence flexible. Il est d'autant plus difficile d'englober toutes les particularités de ces domaines ayant chacun leur complexité de conception et leurs spécificités que les communautés des différents secteurs de conceptions (traitements analogiques, numériques et logicels) ne se comprennent souvent pas.

Nous allons donc présenter quelques concepts de la conception conjointe de système (pas nécessairement concentré sur une seule puce). Tout d'abord, les méthodologies de conception, avec les langages et modèles associés. Puis nous décrirons la problématique de l'exploration de l'espace de conception.

2.2.3.1 Cycle classique de conception

Dans un flot classique de conception d'un SoC, l'application cible est dans un premier temps décrite fonctionnellement. Dans cette description, l'application est décomposée en sous-systèmes ou blocs fonctionnels (figure 2.8) qui correspondent à des parties de l'application ayant localement du sens. Ce processus de décision du partitionnement en blocs logiques et de répartition des blocs entre logiciel et matériel est alors une étape clé pour l'identification de l'architecture adéquate.

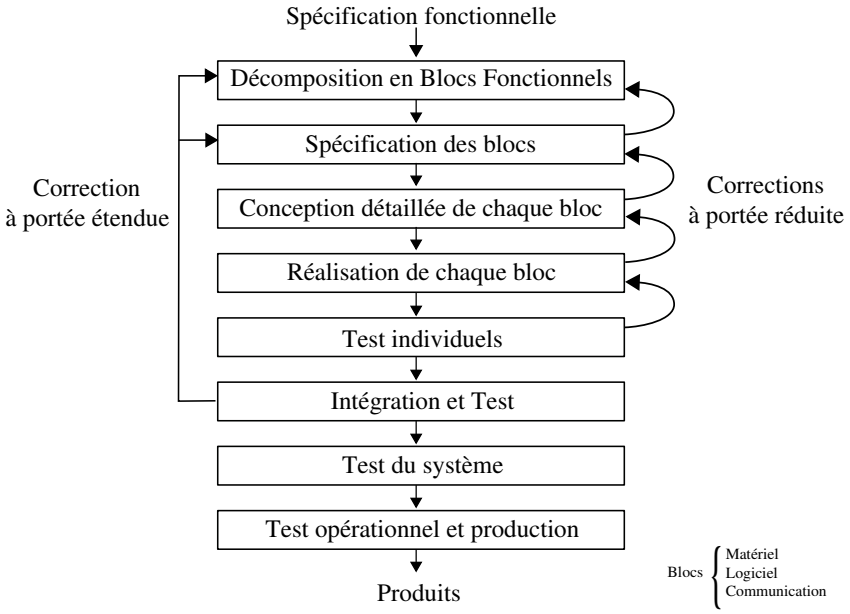


Figure 2.8: *Flot de conception classique*

Un processus de conception est alors appliqué à chaque bloc identifié par le concepteur, ce qui conduit à un modèle de réalisation du bloc. Chaque modèle de réalisation est validé individuellement en y apportant des corrections à *portée réduite*, c'est à dire limitées à ce bloc. Une fois chaque bloc validé, l'ensemble des blocs est assemblé ou intégré pour former le système complet. Les erreurs identifiées à cette étape d'intégration ont en général des portées étendues parce qu'elles peuvent remettre en cause les spécifications de plusieurs blocs : par exemple, des incompatibilités temporelles entre les signaux utilisés dans les échanges de données entre blocs. Il est alors nécessaire de reprendre le processus de conception des blocs concernés et ce tant que leur intégration globale n'est pas jugée correcte. Il peut être nécessaire d'itérer plusieurs fois avant d'arriver à une solution fonctionnelle ce qui allonge d'autant le temps de conception.

Par ailleurs, les objectifs de performances, de surface de silicium et de consommation dépendent directement de la décomposition en blocs de la spécification et des choix de réalisation de ces blocs. Si ces objectifs ne sont pas atteints par des optimisations locales dans les blocs il peut être nécessaire de remettre en cause la décomposition et/ou les choix

de réalisation. Ces changements ont en général des conséquences importantes sur le temps de conception qui peuvent même conduire à l'abandon du projet.

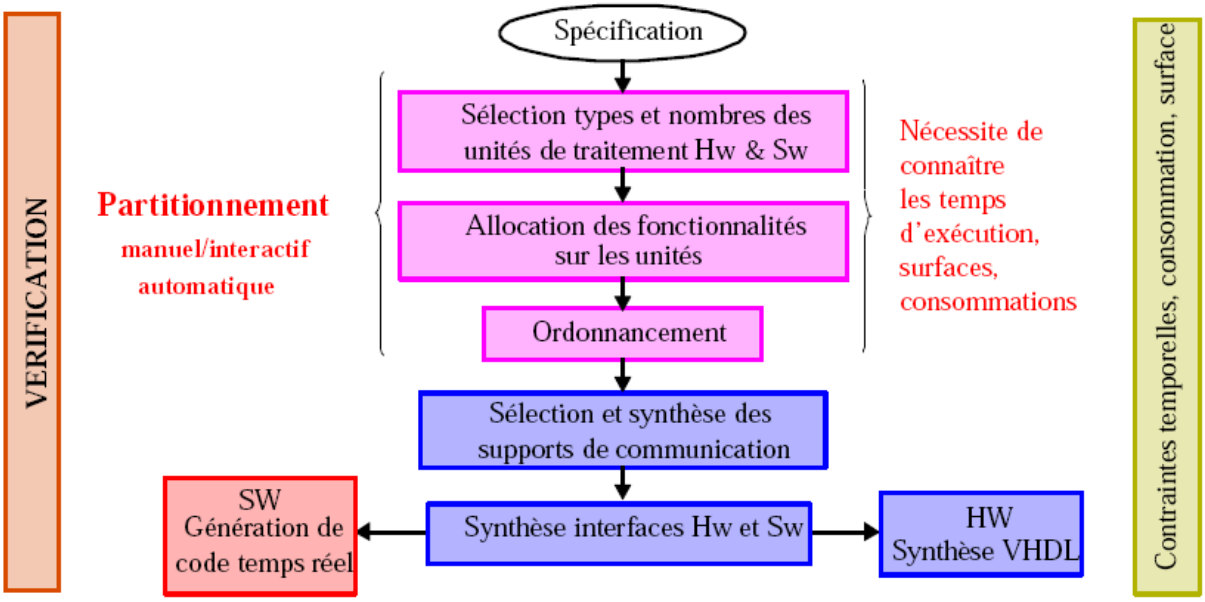


Figure 2.9: *Flot de conception système*

Ces difficultés viennent principalement de la vérification tardive du système complet dans le processus de conception et de la difficile vérification d'un système complexe composé d'unités hétérogènes. Dans la pratique, la vérification au plus tôt est abordée principalement par simulation ou co-simulation (car hétérogène).

L'important est de concentrer dans les premières étapes du flot de conception l'essentiel des choix stratégiques (figure 2.9), sans avoir à effectuer de développements importants qui pourraient révéler des erreurs dans ces choix. Ceci conduit en particulier à éviter une décomposition initiale arbitraire de la spécification en sous-systèmes et milite donc pour considérer une spécification globale de l'application dans un environnement *ad hoc* commun.

À partir de cette description de l'application et de l'expression des contraintes, il s'agit d'explorer un espace de conception système dont la dimension dépend des unités logicielles et matérielles disponibles (les IP), des choix possibles d'allocation et d'ordonnancement des fonctionnalités de l'application sur ces unités. L'espace est borné par les contraintes de temps, de surface et de consommation du système. Trouver une solution optimum dans cet espace est un problème NP-complexe [Gri04]. Les outils commerciaux (e.g. CoWare) essaient d'aborder ce problème avec peu de limitations sur le type d'architecture cible à partir de spécifications exprimées en C/C++, VHDL, Verilog ou SystemC. Ces outils s'appuient sur l'expérience du concepteur pour élaborer une solution. Ils offrent principalement des outils d'évaluation en co-simulation de la solution exprimée. Ce problème a été également largement abordé par la communauté scientifique, sous le terme de partitionnement logiciel/matériel, en considérant souvent des hypothèses restrictives pour en limiter la complexité. Le partitionnement de la spécification nécessite de connaître *a priori* des évaluations des caractéristiques des implémentations des fonctionnalités de l'application sur les unités (logiciel) ou en des unités (matériel) de l'architecture. Ces évaluations peuvent résulter d'études antérieures ou être définies par "profiling". Pour des applications temps réel strict, il est nécessaire de recourir à des techniques d'analyse du temps d'exécution dans le pire cas (WCET, *Worst Case Execution Time*). Ces analyses, principalement étudiées pour des processeurs ayant un comportement plutôt déterministe (absence de cache de données par exemple), restent peu développées pour des cibles matérielles.

2.2.3.2 Autres méthodes/flots de conception

La conception conjointe logicielle/matérielle, ou *Codesign*, tente d'apporter une solution efficace à ces problèmes. En 1983, Gajski et Kuhn [GK83] ont présenté le modèle en Y (Y chart) pour décrire les étapes de conception pour les circuits VLSI (Very Large Scale Integration). La figure 2.7 p. 22 détaille cette organisation. Dans ce modèle, le système est décrit selon trois vues : structurelle (la description électronique), comportementale (la description fonctionnelle) et géométrique (le résultat physique). Aujourd'hui encore, plusieurs travaux s'inspirent de ce modèle Y pour organiser les phases de codesign dans un SoC. Dans les premières phases de conception, les deux parties matérielles et logicielles sont développées parallèlement et séparément. De ce fait, la conception de la partie logicielle peut commencer avant que la conception de l'architecture ne soit complètement terminée ce qui réduit le temps de conception. Pour une plus grande réduction du temps de simulation, les développeurs réutilisent des composants (développés en interne ou achetés) et les intègrent au flot de conception.

Dès les premières étapes de spécification franchies pour les deux parties, on peut réaliser une phase d'association en plaçant l'application sur les composants de l'architecture. Les tâches de l'application seront placées sur les différents types de processeurs et leur codes (données et instructions) seront placés en conséquence dans les différentes mémoires disponibles. Cette phase d'association est à la fois topologique (placement) et temporelle (ordonnancement statique). Une fois l'association terminée, la description du système est complète.

Il est alors possible de vérifier les choix du concepteur sur les performances du système. En d'autres termes, nous vérifions l'adéquation de la capacité de calcul des processeurs avec les demandes des tâches ainsi que la localité des données par rapport aux processeurs qui les manipulent. Plusieurs critères doivent être recherchés. Le premier est le bon fonctionnement du système, c'est-à-dire s'assurer que le SoC répond toujours correctement aux entrées. Le deuxième critère important est le temps d'exécution qui représente la durée nécessaire pour obtenir les sorties. En outre, des critères physiques peuvent être observés afin de s'assurer que le comportement du SoC est compatible avec son usage, tels que la consommation électrique ou la température de fonctionnement.

Quelle que soit la complexité d'un système, la méthode SADT [Ros85] d'analyse structurelle permet de comprendre pourquoi il existe, ou doit être conçu, quelles fonctions il doit remplir et, enfin, comment elles sont réalisées. La méthode, appuyée par un modèle graphique (figure 2.10), procède par approche descendante (du plus général au plus détaillé) en s'intéressant aux activités du système. Ses deux principes de base sont :

- Procéder par analyse descendante : le premier niveau du modèle est en général très abstrait, et progressivement les activités et les moyens nécessaires à leur réalisation sont détaillés.
- Délimiter le cadre de l'analyse : afin d'aborder l'analyse et la description du système, il est fondamental de préciser le contexte (limite du système), le point de vue et l'objectif de l'analyse.

La méthodologie pour la Conception de Systèmes Électroniques (MCSE [Cal90]) est une démarche globale descendante de conception conjointe développée à l'École polytechnique de Nantes. Elle repose sur un ensemble de modèles et de méthodes permettant de simplifier le passage d'une spécification d'un système à une réalisation matérielle. Cette démarche s'articule autour de quatre étapes successives, définissant un flot en 'Y' classique :

- l'élaboration des spécifications, qui consiste à définir le plus précisément possible les contraintes fonctionnelles du système et ses interactions avec son environnement.

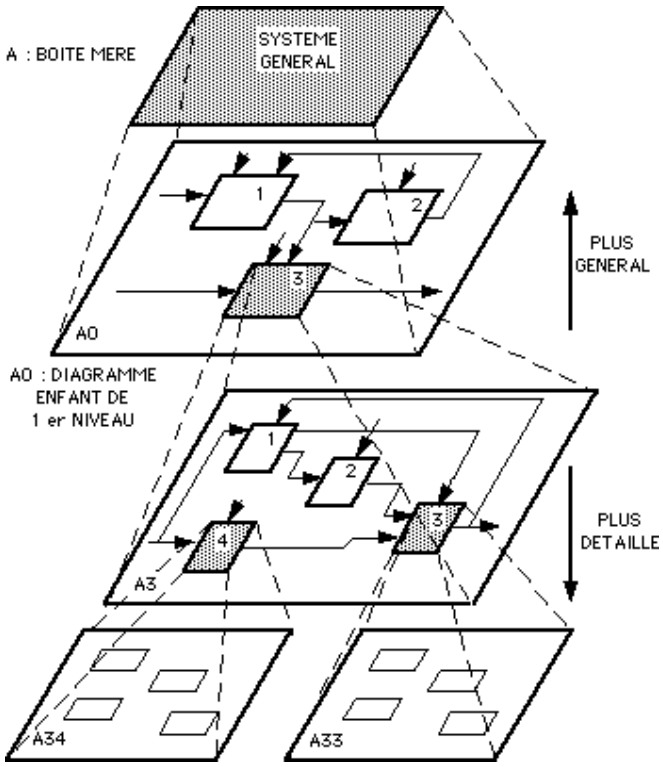


Figure 2.10: *Conception structurelle hiérarchique selon la méthode SADT*

- La conception fonctionnelle qui vise à définir et à valider une solution interne du système et ce indépendamment de toute contrainte matérielle.
- La conception architecturale qui permet de dimensionner et d'analyser la mise en oeuvre de la solution fonctionnelle.
- L'étude de la réalisation qui consiste à intègrer les différentes solutions matérielles et logicielles élaborées.

Cette approche est actuellement supportée par l'outil CoFluent Studio [Cof, MPC04]. Cet outil permet la saisie graphique des modèles associés à chaque étape, il facilite et automatise donc les différentes étapes de conception.

L'apparition d'UML [10] [OMGb] dans les années 90 et son utilisation de plus en plus importante dans l'industrie nous montrent que les approches d'ingénierie guidées par les modèles permettent de mieux appréhender la complexité croissante des systèmes informatiques. L'apport principal de ce type d'approche est de prendre en compte cette complexité dès le premier niveau de conception. De plus, l'utilisation de méthodes semi-formelles permet d'introduire progressivement des outils d'analyse et de vérification. Enfin, elles permettent de créer un support de communication entre le monde des développeurs et celui des méthodes formelles.

De nombreuses méthodologies basées sur la description formelle de haut niveau sont regroupées par le terme de MDE (Model Driven Engineering). Ces méthodes appliquées au codesign utilisent les concepts et formalismes du langage UML formalisé par l'approche OMG MDA (Model Driven Architecture [OMGa]) et utilisent la transformation de modèle (avec le langage QVT) pour raffiner le modèle de haut niveau vers des modèles exécutables, ce qui permet aussi de valider et vérifier les propiétés automatiquement. On peut citer la méthodologie conçue dans le projet MARTES, qui regroupe les travaux sur les profils UML MARTE [RSG+05], SysML [OMG06a] et UML4SoC [OMG06b].

GASPARD [WL99] est un outil de développement de systèmes embarqués développé

10. Unified Modeling Language

par l'équipe WEST de l'INRIA Lille. Le but de Gaspard est de fournir un unique environnement de développement pour tout le processus de conception d'un système embarqué. Il permet [ABN+06], à partir d'une description UML du système, la modélisation du système, la simulation, le test et la génération de code pour l'application et l'architecture matérielle. Gaspard part d'une description générale du système en UML, puis par raffinement et transformation de modèles successifs génère le code de l'application et de l'architecture matérielle.

Les nombreux flots de conception de systèmes embarqués ont en commun d'aider à concevoir un système, et d'essayer de faciliter la vérification des choix préalables nécessaires pour trouver un compromis satisfaisant lors de l'exploration de l'espace de design. Ces flots de conception reposent sur des modèles et langages de description d'architecture qui permettent de raffiner facilement le modèle vers la réalisation matérielle ou d'effectuer des simulations rapides. Il est donc nécessaire d'analyser les formalismes sous-jacents utilisés, ainsi que les langages capables de supporter une telle méthodologie de bout en bout.

2.3 Modélisation des solutions à base de calculateurs

Du point de vue du concepteur de SoC, la conception d'un système peut être considéré comme une séquence d'étapes de modélisation. Le processus commence avec les spécifications de haut niveau, puis affine et transforme les modèles jusqu'à ce qu'ils soient exécutables sur la plateforme cible, conduisant à l'implémentation. Ces modèles décrivent les propriétés et les comportements possibles du système conçu. Ils peuvent être utilisés pour l'analyse, la simulation, l'exécution ou la validation du système.

Nous considérons que la conception de systèmes complexes fait appel à plusieurs domaines techniques. Chaque domaine a ses propres techniques de conception et de langages de modélisation, qui conviennent pour ses besoins spécifiques.

Par exemple, un algorithme de *streaming* vidéo ne se modélise pas comme une algorithme de compresion/décompression d'image. Toutefois, un boitier de télévision numérique va intégrer ces algorithmes de traitement d'image, et devoir gérer la diffusion de flux vers d'autres postes, donc gérer aussi des problématiques réseau qui feront partie du modèle global du système. Par conséquent, plusieurs modèles du système, qui couvrent différents aspects techniques et utilisent des langages de modélisation différents, doivent être utilisés à différentes étapes du processus de conception. Dans le cas plus simple et générique (limité au SoC) qui nous intéresse, les modèles pour le logiciel et pour le matériel sont othogonaux et ont du mal à s'interfacer.

Lorsque des modifications sont apportées à ces modèles sans synchronisation automatique, différents problèmes se posent. Tout d'abord, il est difficile de garder ces modèles cohérents, surtout parce qu'ils utilisent des formalismes différents et sont gérés indépendamment. En outre, si ces modèles ne sont pas liés au niveau d'abstraction auquel ils appartiennent, leur intégration dans le modèle global du système peut se produire seulement à un niveau inférieur qui ne rend pas compte des choix de conception qui ont été faits dans ces modèles. Cette intégration se fait généralement en utilisant un langage de programmation au lieu d'un langage de modélisation. Le problème est que les langages de programmation disent comment les choses se font, pas ce qu'ils sont censés faire ni pourquoi on choisit de les faire d'une façon particulière. L'intégration des modèles de haut niveau dans un langage de bas niveau nous prive de solutions de rechange utiles qui auraient dû être conservées jusqu'à la mise en œuvre.

En outre, lorsque des problèmes sont détectés dans le modèle de système plus raffiné, l'identification de ce qui doit être changé dans les modèles des sous-systèmes pour résoudre ces problèmes peut être une tâche très complexe. Des techniques de modélisation conjointe

permettent de décrire l'ensemble du système comme une composition de sous-systèmes hétérogènes décrits avec des languages différents, afin d'éviter ce genre de questions.

Les langages de modélisation orientés composants, qui sont de plus en plus utilisés pour la réutilisation et des raisons de testabilité, permettent plus ou moins naturellement une modélisation hétérogène (analogique et numérique, à différents niveaux de précision). Dans ce contexte, Lee a étudié [LSV98] l'association de techniques basées sur les composants et l'hétérogénéité hiérarchique. Lee définit un concept qui est fondamental dans notre approche : le concept de modèle de calcul (MoC).

Nous allons tenter de faire un inventaire des modèles de haut niveau, techniques, langages et méthodologies employés pour concevoir un SoC, sans prétendre être exhaustif. Les modèles de haut niveau sont généralement le point de départ d'une spécification d'un système et vont être raffinés jusqu'au niveau de description fin des transistors et instructions logicielles, selon différentes techniques, dont le flot de raffinement classique en Y (figure 2.7 p. 22). La manière de décrire le système peut selon les besoins être représentée selon différents points de vue (vues en couche, vues temporelles, vues fonctionnelles, vues structurelles...) qui ne sont pas forcément liés.

2.3.1 Les modèles de calcul parallèle

2.3.1.1 Modèles de calcul

Les architectures de calcul parallèle rassemblent les systèmes qui contiennent un ensemble de ressources de calcul, de mémoire et de systèmes de connexion pour communiquer et permettent d'exécuter une ou plusieurs applications. La variété des architectures distribuées (de la machine séquentielle super-scalaire à la grappe composée de nœuds multiprocesseurs symétriques) motive la définition de modèles génériques de programmation parallèle. Il s'agit ici d'offrir une sémantique indépendante de l'architecture pour autoriser la ré-utilisabilité et faciliter le couplage de codes (on parle parfois de langages de coordination, ou de programmation par squelettes) ; mais il s'agit aussi de permettre des exécutions performantes, en spécialisant l'ordonnancement à l'architecture. Ceci est parfois réalisé par annotation de code, comme pour OpenMP.

Les applications logicielles concurrentes ont besoin d'être développées selon un ou plusieurs modèles de programmation parallèles pour être exécutées sur de telles architectures. La définition d'une interface de programmation pour machine parallèle peut répondre à plusieurs objectifs : la portabilité des codes existants, la parallélisation automatique ou encore la répartition de charge. Au niveau d'un langage de programmation, l'extraction du parallélisme est un problème complexe, que nous n'abordons pas dans ce travail.

Skillicorn et Talia [ST98] recensent six concepts clefs (implicites ou explicites) que l'on cherche à masquer au programmeur logiciel : la concurrence, la décomposition, le mapping, les communications, la synchronisation et la flexibilité (dynamicité/ordonnancement). Dans notre cas de recherche de la meilleure adéquation algorithme architecture, nous considérerons que l'utilisateur exprime le parallélisme de son algorithme de manière explicite de façon à nous éviter toute analyse complexe du code.

Il y a différentes manières de conceptualiser les calculs d'un système de calcul. Un paradigme ou un modèle de traitement est un modèle, ou une *ontologie*, qui spécifie quelles entités, relations et évènements existent (ou sont à modéliser, décrire, spécifier) dans un ordinateur (existant ou à développer). Autrement dit, un modèle de calcul (ou MoC pour Model of Computation) définit comment les parties d'une application communiquent entre elles et coordonnent leurs actions. Un MoC est donc une abstraction formelle de l'exécution d'un programme sur un ordinateur. Il permet de décrire et définir les raisonnements en formalisant les aspects de concurrence et de distribution d'un système. Il définit la

sémantique des comportements et interactions qui gouvernent les éléments constituants le système, à différents niveaux d'abstraction.

Choisir un MoC sous-entend de souscrire à la manière particulière de conceptualiser ce modèle, et donc prédétermine ce qu'un système peut faire et la manière de le fragmenter en sous parties.

Il faut distinguer le MoC (que l'on traduit par *modèle de calcul, de traitement*) et le modèle de *programmation* d'une part, et le distinguer du modèle d'*exécution* d'autre part.

Un modèle d'exécution est spécifique à une architecture matérielle. Il décrit comment s'exécutent les fils d'exécution/threads sur cette architecture : il détaille l'ensemble des mécanismes d'exécution des calculs, des communications, partages des données, synchronisations ou encore gestion des entrées-sorties ou la façon dont les threads sont répartis sur les éléments de calcul ou les processeurs. Un modèle d'exécution est ainsi très proche du fonctionnement du matériel.

Un modèle de programmation est un ensemble de technologies logicielles et d'abstractions fournissant au concepteur une manière d'exprimer le programme ou l'algorithme de façon à correspondre à l'architecture cible. Ces technologies logicielles existent à différents niveaux d'abstraction et englobent les langages de programmation, les bibliothèques, les compilateurs, les RTOS, etc. Selon le domaine, les contraintes de l'application et l'habitude du programmeur, divers modèles de programmation sont possibles.

Alors qu'un MoC est un modèle beaucoup plus abstrait qui décrit un système à un plus haut niveau de manière synthétique en masquant certains détails comme nous allons le voir.

2.3.1.2 Modèles de calcul classiques

De nombreux modèles de calcul (MoC) ont été définis dans la littérature, chacun apportant ses spécificités. On parle parfois de modèles de traitement, ou de modèle de spécification pour les MoC utilisés à haut niveau. La principale difficulté réside dans la modélisation du temps et de la concurrence [FMMR10].

Le choix du modèle de spécification dépend (faute d'existence de MoC universel) du genre d'application du système envisagé, selon qu'il est dominé majoritairement par le contrôle ou par les données. On utilise classiquement des MoC de type *Système à événements discrets, Système réactif synchrone, Machine d'états finis, Réseaux de Petri, Processus concurrents et communicants (Graphe de tâches)* si le concepteur considère qu'un système est dominé par le contrôle :

- Réactions à des événements discrets
- Pas d'hypothèse sur l'occurrence des événements
- Tous les événements doivent être pris en compte
- Contrôle/commande de processus

Sinon on s'orientera plutôt vers les MoC *Kahn Process Networks, Dataflow Process Networks* lorsque le système est plutôt dominé par les données :

- Les sorties sont fonctionnellement dépendantes des entrées
- Les occurrences des valeurs sur les entrées sont périodiques
- Traitements intensifs
- Traitement du signal et des images

Les travaux de [ELLSV97] établissent ces concepts et énumèrent différents MoC couramment utilisés, que l'on retrouve aussi dans [CSG98] et une synthèse de Jantsch et Sander [JS05, Jan05] et que nous résumons succintement ainsi :

- Synchronous Data Flow (SDF)
  Les flots de données synchrones [LM87] sont des modèles statiques dans lesquels le nombre d'éléments de données produits ou consommés est connu lors de la concep-

tion. Les nœuds de traitement (ou tâches) communiquent à travers des files d'attente et peuvent être ordonnancés statiquement selon le flux d'arrivée des données (synchronisation implicite par les données). Généralement, les applications de traitement du signal se prêtent bien à ce genre de modélisation. On peut les spécifier à l'aide de DFG (Data Flow graph), où chaque nœud représente un élément de calcul et les arcs des chemins de données. On retrouve ce modèle pour décrire aussi bien des applications logicielles que pour décrire des systèmes d'accélérateurs matériels.
- Continuous Time (CT) : les éléments de calcul communiquent avec des signaux en temps continu. Ces systèmes sont typiquement utilisés pour représenter des équations différentielles comme dans les systèmes physiques et l'électronique analogique. Les outils et langages associés sont typiquement ceux de Matlab/Simulink.
- Processus Séquentiels Communicants(CSP) : qui communiquent via des messages synchrones comme le *message passing* ou les rendez-vous [Hoa78]. Les langages CSP, Occam, Lotos se prêtent bien à ce genre de formalisme.
- Réseaux de processus (PN) comme les Kahn PN : les réseaux de processus de Kahn [Kah74] communiquent via des messages asynchrones transitant par des queues (FIFO [11]) de taille infinie. C'est un modèle courant pour les applications de traitement du signal.
- Finite State Machine (FSM) ou machine à états finis : ces FSM sont représentées par des graphes de flots de contrôle (Control-Flow Graph (CFG)) et d'états (*statecharts* de Harel [HN96]), dans lesquels les nœuds représentent des états et les arcs sont les traitements déclenchés par des éléments booléens que l'on nomme gardes. On les utilise souvent pour décrire les automates (logiciels ou matériels) mais aussi pour décrire les états des tâches logicielles dans un RTOS. Certains [SLSV00] définissent même un modèle abstrait basé sur les FSM pour le codesign, nommé ACFSM (Abstract codesign finite-state machine).
- Discrete-Events (DE) : les nœuds de calcul interagissent via des files d'évènements. On les utilise souvent pour décrire des circuits numériques asynchrones. On y associe aisément les langages de design matériel (HDL) comme Verilog et VHDL.
- Petri Net : les nœuds d'un réseau de Petri effectuent des calculs asynchrones avec des points de synchronisation explicites.
- Synchronous Reactive (SR) les éléments effectuent leur calculs et réitèrent jusqu'à ce que le système se stabilise. Ce MoC est utilisé par les langages synchrones tel que *Esterel* [BG92], *Lustre*[HLR92], *Signal* ou *SyncCharts* qui servent à décrire le logiciel et une partie du matériel.

De nombreux frameworks de modélisation, comme Ptolemy [BHLM94], Metropolis [DDM+07], ou BIP [Sif05], cherchent à unifier ces différentes vues, en proposant de simuler un système hétérogène à travers une composition hiérarchique de différents MoC.

2.4 Méthodes d'exploration et validation

L'exploration de l'espace de conception consiste à trouver une solution qui permet de se rapprocher des objectifs et contraintes du système spécifié. Les choix primaires vont être dirigés par des contraintes de coût et taille physique, de consommation, de rapidité, et de flexibilité. Ensuite, on va pouvoir affiner les évaluations. On va regarder l'utilisation des ressources, mesurer les débits des communications ou de l'utilisation mémoire, ou d'autres métriques. On va aussi considérer la fiabilité, évaluer le déterminisme de l'exécution, et considérer des aspects de compatibilité avec des composants existants, ou la facilité d'utilisation du système voir sa testabilité.

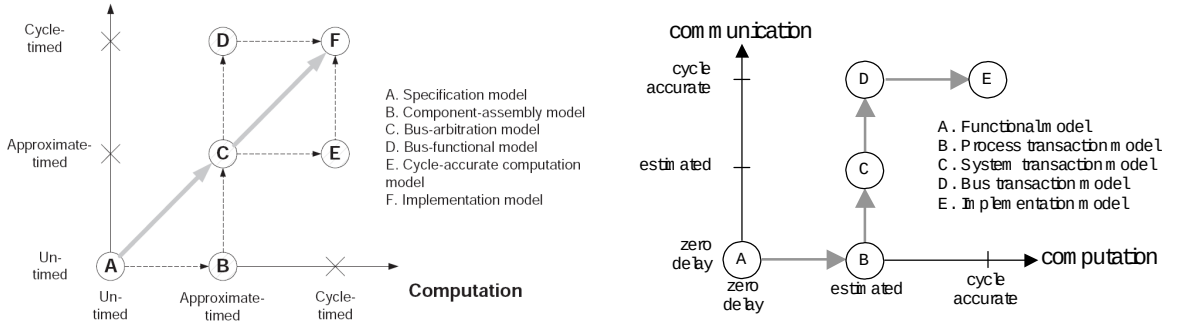
11. First In First Out, premier arrivé premier sorti

L'exploration peut s'effectuer à différents niveaux d'abstraction, plus le niveau est bas, plus les mesures et métriques permettant de faire des choix sont précis, mais en contrepartie cela demande beaucoup plus d'effort et de temps pour les obtenir, ce qui rend l'exploration de différentes configurations plus laborieuse.

L'approche la plus utilisée pour s'assurer de ces critères est la simulation conjointe des deux parties logicielles et matérielles appelée aussi la co-simulation. Elle consiste à simuler l'ensemble du système et à fournir au développeur des informations sur le déroulement de l'application sur la plateforme matérielle (le temps d'exécution, la consommation, etc.). Ces informations permettent de localiser les points à optimiser/modifier dans le logiciel, dans le matériel ou dans l'association qui a été réalisée entre les deux.

2.4.1 Modélisation hétérogène et niveau de précision

Une clef de l'exploration est sa complexité, qui croît avec le niveau de raffinement des architectures et applications. L'objectif est d'éliminer de l'espace de recherche, le plus tôt possible dans le flot de conception, les solutions architecturales les moins performantes. L'utilisation de plusieurs niveaux de modélisation offre aux développeurs l'avantage de disposer d'un outil de simulation et d'estimation des performances dès les premières phases de conception. A haut niveau d'abstraction, la précision des estimations (vitesse d'exécution, consommation, etc.) obtenues est relativement faible. Au prix d'une augmentation des temps de conception et simulation, le niveau de précision peut être augmenté par un raffinement dans la description du système.



(a) Gajski définit des niveaux de raffinement TLM selon que l'on s'intéresse à la partie algorithmique ou aux communications

(b) Graphe de modélisation TLM selon Jantsch

Figure 2.11: *Niveaux de modélisation des transactions (TLM) et raffinements*

Plusieurs travaux de recherches proposent une classification des niveaux d'abstraction, en allant du niveau le plus bas vers le niveau le plus haut, appellé généralement le niveau des transactions où les détails des échanges sont omis. Une grande diversité de formalismes est utilisée dans la littérature pour désigner les différents niveaux de raffinement. Cependant une sorte de consensus semble émerger du domaine de la co-simulation où l'on parle de niveaux de modélisation transactionels, ou TLM (*Transaction Level Modeling*). La terminologie employée peut cependant varier, on le constate par exemple en comparant l'approche de Gajski [CG03a](figure 2.11(a)) avec celle de Jantsch [DJO04](figure 2.11(b)). Généralement, les points de différence concernent soit la terminologie soit le niveau de précision dans l'estimation des performances dans les parties calcul et communication.

On distingue cependant deux axes communs, ceux de la communication et du traitement pour lesquels trois niveaux sont globalement définis de la manière suivante : le niveau fonctionnel, le niveau *approximate time* et le niveau *cycle accurate*. Le niveau fonctionnel ou *untimed* correspond à une spécification (Matlab, C, ...) pour la simulation

fonctionnelle sans notion de temps. Le niveau *approximate time* confère à une suite d'opérations de granularité généralement élevé (thread, fonction) une latence estimée. Enfin le niveau *cycle accurate* tient compte des délais à un niveau fin, celui des fronts d'horloge (ex. *Clock'event* en RTL).

La communauté du codesign a cherché à augmenter toujours plus le niveau d'abstraction afin de permettre de concevoir mais surtout faciliter l'exploration. Un consensus est apparu autour de l'idée de *modélisation transactionnelle*. Elle consiste à découper la modélisation du système en niveaux d'abstractions, de manière à structurer le raffinement : UTF, TF, BCA, CA, parfois trouvés dans la littérature comme le modèle CP/CPT/PV/PVT/CC, comme on peut le voir en figure 2.12.

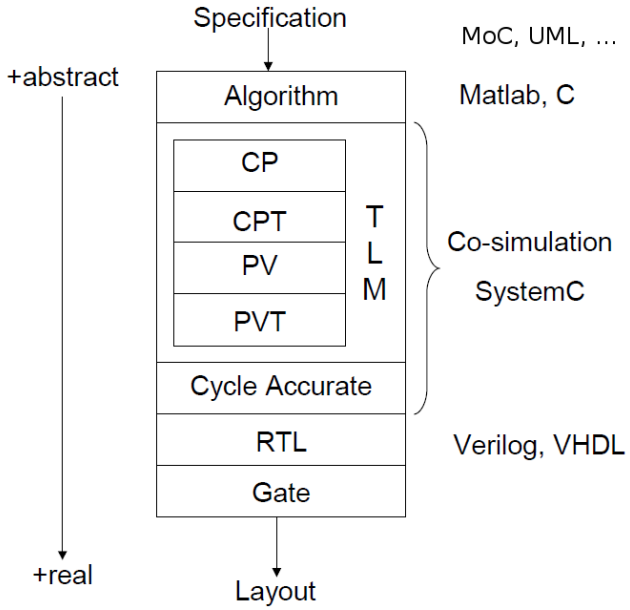


Figure 2.12: *Niveaux de modélisation TLM de la communauté du codesign*

Algorithme : A ce niveau de la description, nous disposons seulement de la description de l'application sous forme de spécification algorithmique. Pour un certain nombre d'applications, le ou les algorithmes sont connus et documentés. Ceci est le cas lorsque soit ils ont déjà été conçus dans une précédente réalisation soit ils font l'objet d'une norme comme les algorithmes de codage ou décodage vidéo, soit sont des composants sur étagère ouverts (librairies ou IP). A ce niveau de la description, l'architecture matérielle du système n'est pas définie et les algorithmes sont décrits dans des langages de haut niveau, comme Matlab ou C++ ou tout autre modèle de calcul (MoC, cf. sous-section 2.3.1). L'utilité de ce niveau est justifiée par la possibilité de réaliser une vérification fonctionnelle précoce de l'application via une exécution de celle-ci sur la machine hôte ou l'utilisation d'outils dédiés.

Niveau transactionnel : Depuis les premières publications sur TLM (Transaction Level Modeling) en 2000 [GZD+00, GLMS02], plusieurs définitions et classifications des différents niveaux de TLM ont été présentées dans la littérature [Don04, CG03a, CoW, RSPF, CCGM03, Ghe06, ACG+07] ou par besoin dans un standard pour l'interconnexion d'IP (OCP-IP [OCP]).

Toutes ces propositions ont en commun les points suivants :

- TLM est présenté comme une taxonomie de plusieurs sous-niveaux.
- Les communications et les calculs sont séparés en niveaux (composants ayant des comportements et des interfaces)

– Les transactions entre les modules sont simplifiées en utilisant des méthodes de communication par canaux (*channels*)[GZD+00].

Le **sous-niveau Processus Communicant (CP)** : Dans le sous-niveau CP, l'application est découpée en tâches encapsulées dans des processus communicants. Dans CP il est possible donc de mesurer la quantité d'opérations de communication entre les tâches. Ce sous-niveau permet une première expression du parallélisme dans l'application sans se référer à l'architecture du système. Des annotations temporelles peuvent aussi être spécifiées au sous-niveau CP pour estimer le temps d'exécution. Dans ce cas, nous obtenons le sous-niveau **CPT** ce qui correspond à une modélisation sous forme de processus communicants temporisés.

Le **sous-niveau Vue du Programmeur (PV)** : C'est uniquement à ce niveau que l'architecture du système est prise en compte et où la co-simulation logicielle/matérielle devient possible. La partie matérielle est représentée sous forme de composants de différents types (processeurs, mémoires, réseau d'interconnexion, etc.). Des modules de communication sont utilisés comme mécanismes de transport des données et des politiques d'arbitrage entre les requêtes sont appliquées au niveau des ressources partagées. La description des composants doit être suffisamment précise pour que le concepteur puisse exécuter l'application de façon similaire au système final. Ainsi, le sous-niveau PV permet une première vérification du système entier (logiciel et matériel).

Néanmoins, les propriétés non fonctionnelles telles que le temps d'exécution ou la consommation d'énergie, sont soit omises soit approximées (par estimation rapide du nombre d'instructions ou tout autre moyen). Dans ce dernier cas, des annotations temporelles sont ajoutées à la description pour obtenir le sous-niveau **PVT**, qui correspond à un niveau *vue du programmeur temporisé*. L'objectif dans PVT est d'atteindre des estimations de performance proches de celles obtenues par les bas niveaux de description.

C'est ce niveau PVT que nous choisirons pour notre modèle, car il représente un bon compromis entre vitesse de simulation et précision lors de la phase initiale d'exploration conjointe d'architecture logicielle/matérielle. Notre modélisation prennant en compte le logiciel et surtout l'OS en plus des aspect materiels, nous appellerons ce niveau SAT, comme décrit au chapitre suivant, en section 3.2.4.

Niveau Cycle Précis Bit Précis (CABA [12]) : Au niveau CABA, le temps d'exécution des éléments fins du système (les transitions entre blocs) est décrit de façon très précise. Il permet de simuler le comportement des composants au cycle près. Au niveau calcul, une description de la micro-architecture interne du processeur (pipeline, prédiction de branchement, cache, etc.) est réalisée. Au niveau communication, un protocole de communication précis au bit près est adopté. Cette description détaillée et fine du système permet d'améliorer la précision de l'estimation des performances.

Niveau RTL : La modélisation au niveau RTL correspond à la description de l'implémentation physique du système sous forme de blocs élémentaires (bascules, multiplexeurs, UALs, registres, etc.) et à l'utilisation de la logique combinatoire pour relier les entrées/-sorties de ces blocs. En général, nous utilisons les outils de synthèse standard pour obtenir le masque (layout) du système à partir du niveau RTL.

Notre approche consiste à considérer que l'exploration prend son sens à un haut niveau d'abstraction (partie TLM de la figure 2.12) afin d'évaluer rapidement un large spectre de solutions. Comme indiqué précédemment les niveaux précis relèvent de la vérification et/ou de la simulation. Aussi, l'exploration de type fonctionnelle et temporelle est effectuée au niveau algorithmique à partir d'un code C sans *a priori* sur l'implantation. En effet, en SystemC une simulation TLM est au moins 25 fois plus rapide qu'une simulation au niveau Cycle Accurate [VP06]. Le parallèle entre la rapidité de simulation et la précision

12. Cycle Accurate Bit Accurate

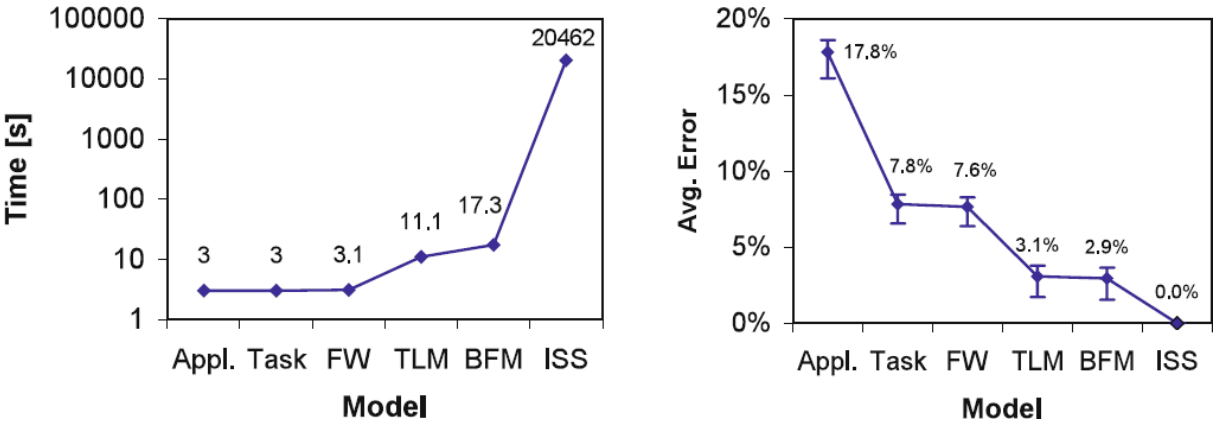


FIGURE 2.13: *Vitesse et précision de simulation à différents niveaux [SGD07]*

selon les niveaux d'abstraction a été très bien démontré par Gerstlauer [SGD07] comme on peut le voir dans la figure 2.13 (le niveau FW représente un niveau *firmware*, où les fonctionnalités de l'OS et les couches basses dont la gestion des interruptions sont modélisées).

L'objectif est de fournir rapidement au concepteur les clefs lui permettant de prévoir les gains potentiels resultants de l'exploration du parallélisme d'une architecture. Cependant pour évaluer la projection physique, qui consiste à préciser le modèle d'architecture, on peut utiliser un mélange de niveaux de modélisation qui permettent par exemple d'estimer plus précisément les temps de traitements et d'accès mémoire ou la surface de l'architecture, sans nécessiter de détailler l'intégralité du système.

2.4.2 Langages de design matériel et système

Comme nous venons de le voir, il est important de pouvoir modéliser un système complet, comprenant des parties logicielles et matérielles, à plusieurs niveaux d'abstraction. Pour faciliter cette approche et permettre un passage rapide entre les niveaux, il serait intéressant d'utiliser un même langage de description permettant de couvrir tout le flot de conception tout en restant exécutable à tous les niveaux d'abstraction. Malheureusement, ce langage n'existe pas encore. Etant donné que le logiciel est typiquement développé en C/C++ et le matériel en langage HDL (Hardware Description Language), plusieurs initiatives font aujourd'hui des efforts pour unifier le langage de description. Certaines initiatives tentent d'adapter le C/C++ à la description matériellei (i.e. SystemC), d'autres cherchent à étendre les HDL à la description logicielle et système (i.e. SystemVerilog). Une troisième voie appelle à la spécification de nouveaux langages (i.e CAL).

Dans le cadre de cette thèse, nous nous intéressons à la co-simulation logicielle/matérielle dans la conception des MPSoC. Notre objectif est de permettre d'évaluer les performances du système avant d'atteindre les phases finales de conception (au-delà de RTL). Le succès dans cet objectif nécessite tout d'abord le choix d'un langage bien adapté à la description de haut niveau, permettant ainsi l'exécution de notre système tout entier dans un temps raisonnable. Il sera intéressant aussi que le langage de description choisi permette de visualiser le comportement dynamique du système. Au cours de notre développement, le standard SystemC a été adopté comme langage de description des systèmes MPSoC. En effet, les spécificités de ce langage répondent au mieux à notre problématique de co-simulation comme nous le détaillerons dans ce qui suit.

2.4.2.1 SystemC et la librairie TLM

Nous allons nous intéresser en particulier au langage SystemC, car il confère l'avantage de pouvoir spécifier différents MoC [PSMN06, DHG+08] et offre le potentiel pour une co-simulation logicielle/matérielle à haut niveau.

SystemC est une plate-forme de modélisation composée de librairies de classes C++ et d'un noyau de simulation. Il introduit de nouveaux concepts (par rapport au C++ classique) afin de supporter la description et la simulation du matériel et ses caractéristiques inhérentes comme la concurrence et l'aspect temporel, en se basant sur un moteur de simulation évènementiel. Ces nouveaux concepts sont implémentés par des classes C++ tels que les modules, les ports, les signaux, les FIFO, processus (threads et methods), etc., permettant ainsi la conception de systèmes matériels à différents niveaux d'abstraction.

L'initiative SystemC se développe dans le cadre de l'OSCI (Open Source systemC Initiative) qui est chargée de diffuser, promouvoir et rédiger les spécifications de SystemC. Depuis décembre 2005, SystemC est standardisé auprès de l'IEEE sous le nom de IEEE 1666-2005. Au cours de ces dernières années, SystemC a suscité un intérêt de plus en plus grand auprès des chercheurs et des industriels. La première raison de cet intérêt est que SystemC est basé sur le standard C++ largement utilisé pour le développement logiciel, et que par ailleurs, il ne nécessite pas de licence d'utilisation. Il devient donc possible d'utiliser les outils conventionnels de déverminage[13] et de vérification fonctionnelle [GHPS09] qui sont généralement maîtrisés par les développeurs.

Par ailleurs, l'emploi de SystemC pour modéliser les différents composants d'un système embarqué autorise l'utilisation d'un seul moteur de simulation. Ceci se traduit par un gain en temps de simulation par rapport aux simulateurs HDL et surtout évite l'utilisation des techniques de synchronisation entre différents moteurs de simulation.

La librairie TLM est un des principaux attraits de SystemC. Elle définit de nouveaux concepts permettant de supporter la description du matériel au niveau transactionnel. Ce protocole TLM décrit un ensemble de méthodes standardisées de communication jouant le rôle d'API pour l'utilisateur, facilitant ainsi la modélisation TLM et l'interconnexion de modules. Ces transactions se présentent sous la forme de requêtes de commande ou de requêtes de réponse qui servent à convoyer les transactions entre les initiateurs et les cibles et vice versa. Afin de préciser le format de chaque type de requête ainsi que la liste des arguments nécessaires, un protocole de communication est défini. A titre d'exemple, pour initier une opération de lecture ou d'écriture, des appels de fonctions de type `read()` ou `write()` sont utilisés. Ces fonctions de requêtes sont transmises à travers les ports des composants connectés à des canaux de communication. Les ports ainsi connéctés ont la charge de convertir les les requêtes entre interface métier et interface TLM.

En résumé, SystemC possède des caractéristiques intéressantes pour la description de l'architecture à haut niveau. Cependant, ce langage n'est, pour le moment, pas synthétisable dans son intégralité. Il n'est donc pas possible de l'utiliser au-delà du niveau RTL. Pour cette raison SystemC est utilisé en général pour les tâches de modélisation et de vérification rapide du système.

2.4.2.2 Autres langages de design matériel et système

De nombreux langages sont apparus ces dernières décennies pour pouvoir modéliser, simuler et implémenter des systèmes temps-réel, notamment pour prendre en compte le comportement dynamique et le parallélisme et faciliter la conception de système en tenant compte de ses particularités et contraintes :

13. vrai terme pour l'anglicisme *débogage*

- les langages de description de matériel : Verilog, VHDL, Abel, AHDL (Altera), et aussi SystemC dans sa sous partie synthétisable.
- Les langages de conception d'architecture de haut niveau comme HandelC, HardwareC et (SAE) AADL (qui semble s'imposer dans le domaine aéronautique et militaire).
- Les langages de niveau système ou System Level Design Langage(SLDL) se prêtent bien à la conception conjointe de systèmes à haut niveau d'abstraction. Il existe en effet d'autres langages que SystemC qui peuvent être aussi utilisés pour une conception et la co-simulation logicielle/matérielle des systèmes sur puce monoprocesseur ou multiprocesseur. On peut citer SpecC, SystemVerilog, YML, ImpulseC, EmbeddedC (avec MARTE), CAL [EJ03].

Nous décrirons ici à titre d'exemple, SpecC [Spe], VHDL [vhd04] et SystemVerilog [Sys05].

SpecC est un langage de description et de spécification système basé sur le standard C, avec des extensions syntaxiques. Ces extensions comprennent, par exemple, des types de données adéquats, des commandes pour supporter la concurrence et la description de machines à états. Il a été développé à l'Université de Californie à Irvine. Au-delà du langage lui-même, les travaux menés par l'équipe de D. Gajski introduisent des concepts intéressants pour la modélisation système ainsi que des techniques de raffinement. Contrairement à SystemC, SpecC n'est pas un langage *open source*, limitant ainsi les travaux de recherche et le développement d'outils autour de ce langage, nécessaires pour une adoption plus large.

Malgré la préférence américaine pour le langage Verilog, VHDL est le langage de description matériel le plus répandu dans le monde. Il est conçu pour décrire le comportement et/ou l'architecture d'un système électronique numérique. Son utilisation dans le monde industriel ou académique est fortement liée à une phase avancée de la conception d'un système VHDL, car il permet une description du système au niveau RTL. L'intérêt d'une telle description réside dans son caractère exécutable pour vérifier le comportement réel du circuit.
En outre, des outils de CAO existants permettent de passer presque automatiquement d'une description en VHDL à un schéma en porte logique et par la suite au masque de gravure final. En comparaison avec SystemC, le langage VHDL ne permet pas la vérification de la conception à un haut niveau d'abstraction (notamment TLM). Ainsi, dans un même environnement de co-simulation, ces deux langages peuvent être utilisés dans des phases de conception différentes suivant le niveau d'abstraction. Ils peuvent donc être considérés comme deux langages complémentaires.

Le langage SystemVerilog [14] présente une autre approche que SystemC ou VHDL : il résulte de l'extension du langage standard de description matérielle Verilog. A ce dernier, sont associés de nouveaux concepts afin que SystemVerilog puisse supporter la description de haut niveau et la vérification des modèles développés. C'est un avantage par rapport à SystemC, bien que SystemC en possède d'autres, notamment pour les aspects logiciels.

2.5 Les simulateurs et modèles conjoints pour l'évaluation de systèmes

Une des principales difficultés de la modélisation d'un (futur) système numérique réside dans la nécessaire incorporation des aspects matériels de la reconfiguration (communication flexible et distribuée, simulation de la préemption matérielle). Une autre est la

14. standardisé en 2005 : IEEE 1800-2005

difficulté de simuler le logiciel et les services des OS conjointement à l'architecture multiprocesseurs. La question traitée ici est donc de déterminer comment on peut modéliser, explorer, raffiner et valider de telles plateformes afin de permettre l'évaluation des choix très tôt dans le flot de conception.

La présence d'un OS permet de constituer une couche chargée d'assurer l'exécution correcte de l'application, mais également constitue une couche d'abstraction de la plateforme (une interface de programmation) facilitant le déploiement des tâches de manière mixte, en logiciel ou en matériel. Il est évident qu'on ne peut se contenter d'un OS classique car ceux gérant la reconfiguration dynamique n'existent pas encore. Il est donc nécessaire de pouvoir explorer aussi les services de l'OS.

Nous allons donc passer en revue l'état de l'art des travaux cherchant à modéliser des OS de MPSoC et permettant aussi de co-simuler conjointement cet ensemble hétérogène logiciel et matériel. Ce genre de travaux promeut le prototypage rapide pour répondre aux questions de design (puce hétérogène ou pas, multicœur reconfigurable ou pas...) et aide à la conception conjointe matérielle/logicielle avant d'avoir un design figé.

Dans la littérature, le terme co-simulation possède un sens plus large que simuler le logiciel et le matériel ensemble. En effet, il regroupe différents aspects de simulation dont la problématique est l'intégration de multi-modèles de calcul, l'interopérabilité multi-langage et multi-niveaux [CHLM+99, Nic02], à temps continu et discret etc. Au cours de cette thèse, nous nous focaliserons sur la co-simulation logicielle/matérielle dans un même langage, SystemC, dans l'optique de concevoir des services de gestion de zones reconfigurables et de services de distribution de tâches sur des nœuds de calcul hétérogènes.

2.5.1 Les frameworks de co-simulation intégrés

De nombreux environnements d'exploration globale d'architecture tels que MetroPolis (I [BWH+03] et II [DDM+07]), Mescal [MKS+02], MESH [CPT03], Milan [MP02], et des environnements basés sur SystemC, comme celui des travaux de Kogel [KDK+04], facilitent l'évaluation des performances au niveau système en permettant d'associer des spécifications comportementales d'application à des spécifications d'architecture. Par exemple, dans le framework MESH [CPT03, PBN+03], la technique de simulation de haut niveau est centrée sur l'ordonnanceur représenté comme une machine à étatfinis. Les ressources (blocs matériels, logiciel et ordonnanceur) sont vues comme trois niveaux d'abstraction modélisés par des threads logiques gérés par des ordonnanceurs associés chacun au fil d'exécution physique unique de son processeur. Une technique de mesure sur ISS est utilisée pour associer des annotations d'évènements logiques aux évènements physiques (liés aux ressources matérielles, en particulier les accès mémoires) et associer des budgets temps permettant aux threads matériels de se synchroniser. La limite de cette approche de haut niveau est son incapacité à gérer une préemption lors d'interruptions aléatoires.

Le framework Plateform Designer (PD) des brésiliens Araujo *et al.* [AGB+05] est un outil permettant de concevoir rapidement une plateforme MPSoC sous la forme d'un modèle SystemC à l'aide du langage ArchC ADL et de l'outil *acsys*. De nombreux éditeurs proposent aussi des outils de conception conjointe : Coware/Synopsis [CS06], SPACE codesign [CBR+03, CBR+04a, CBR+04b], Scicos/SciLab [CCN06], AAA/SynDEx [GLS99] ou PragmaDev [Pra08]. Certains outils académiques comme Ptolemy [BHLM94] ou SpecSyn [GVNG98] addressent la problématique de la conception selon le paradigme *spécifier-explorer-raffiner*. La *survey* [Gri04] présente en détail de nombreuses méthodes, outils et environnements pour l'exploration rapide de l'espace de conception.

Bien que tous ces frameworks cités soient intéressants pour leur capacité à embrasser un ensemble de modèles de traitement et facilitent grandement l'élaboration de l'architecture d'un SoC, ils ne permettent pas de simuler conjointement l'application logicielle du futur

système à l'aide d'un OS ni d'explorer ce dernier. On se propose donc de permettre la simulation d'OS dédié à la conception de systèmes embarqués, permettant l'exploration de plateforme, ce qui nécessite de concevoir un modèle générique de conception d'OS incluant nécessairement simulation et exploration.

2.5.2 Simulation de système d'exploitation

Dans les systèmes temps réel actuels les algorithmes que l'on demande d'implémenter sont de plus en plus gourmands en calculs et les contraintes de coût et d'énergie se retrouvent parfois reléguées au second plan par la seule difficulté de trouver une architecture satisfaisant les contraintes du budget temps-réel. C'est pour cela que la phase initiale d'exploration est importante voire indispensable et si possible à haut niveau quand on ne dispose pas des blocs matériels déjà adéquats, ou que ces derniers existent et sont parfaitement caractérisés, mais qu'il est tout de même difficile d'évaluer les interactions et le déroulement dynamique entre les différentes parties.

Afin de fournir une certaine efficacité et flexibilité à ces systèmes complexes, on utilise de plus en plus de logiciel embarqué, et pour gérer ce logiciel, l'utilisation d'un ou plusieurs OS devient obligatoire, notamment pour gérer la concurrence et les communications entre tâches distribuées sur différents nœuds (processeurs et IP).

Généralement, la partie logicielle s'appuie sur un OS contrôleur, mais beaucoup de glue architecturale participe aussi au rôle de contrôleur de plateforme (MMU, DMA, contrôleur d'IRQ, etc.), surtout dans les systèmes constitués de nombreux nœuds d'exécution (processeurs et accélérateurs). Ainsi, une validation technique d'un SoC ne peut omettre le système d'exploitation de la plateforme, matérialisé sous forme d'OS logiciels et de blocs matériels aidant au contrôle, mais les techniques classiques de validation du logiciel et du matériel sont souvent disjointes et incluent rarement les OS dans la simulation.

Il est donc nécessaire de concevoir un simulateur de système d'exploitation adjoint au simulateur du système. Ce genre de modèle requière :

- de la cohérence : le modèle d'OS doit correspondre et respecter le schéma de fonctionnement de l'implémentation finale.
- de la précision : le modèle d'OS doit permettre une estimation fiable des futures performances.
- la garantie de l'interopérabilité : les interactions avec le matériel sont essentielles dans les systèmes embarqués et temps-réel, donc on doit pouvoir valider autant le logiciel que le matériel. Par conséquent il faut pouvoir modéliser les communications et les interactions entre ces deux univers.

2.5.2.1 Techniques de validation de logiciel embarqué

Il existe plusieurs techniques pour simuler du logiciel et son (ou ses) OS, selon que l'on se place du point de vue purement logiciel ou que le couplage avec le matériel et notamment la gestion des interruptions, de la mémoire et des communications représente un point clef du fonctionnement du système. Dans un cas, il est parfois possible l'exécuter sur une architecture semblable ou sur un simulateur fin de l'architecture (ISS et matériel modélisé au niveau cycle). Dans d'autres cas, la modélisation de plus haut niveau est une excellente alternative.

2.5.2.1.1 Niveau de simulation d'OS L'utilisation d'un modèle de système d'exploitation devient courante dans les travaux de recherche destinés à la validation conjointe matériel/logiciel. Selon leur niveau d'abstraction, on distingue différents types de modèle de simulation. Il existe au moins trois principales techniques pour valider par simulation

le logiciel et son OS : la simulation abstraite (haut niveau), la simulation native, et les simulations basés sur des simulateurs de jeu d'instructions (ISS).

Modèle d'OS abstrait : L'OS est fourni par le simulateur et lui permet de jouer un ensemble de tâches parallèles décrites dans un langage de haut niveau. L'OS permet aux tâches de communiquer à l'aide de quelques primitives d'échanges et de synchronisation. L'ordonnancement des tâches est effectué par le noyau de simulation. On peut ici se contenter de tâches abstraites sans code fonctionnel, en utilisant les caractéristiques de tâches (priorité, deadline, WCET etc.) estimées ou évaluées au préalable.

Généralement, ces simulations se basent sur des modèles *ad hoc* (MoC) permettant de simuler des graphes de tâches. Rapides et efficaces pour valider et pour étudier l'ordre de précédence des tâches, elles ne permettent pas d'obtenir des mesures précises de l'exécution (car basées sur des estimations du WCET et n'exécutent pas de tâches fonctionnelles), ni de réellement prendre en compte les différents services d'OS, ni l'environnement du support matériel : les interruptions, la mémoire et les transferts de données.

Modèle d'OS virtuel : Dans une simulation native, les comportements des services de l'OS sont similaires à celui représenté. Si les caractéristiques de temps et d'algorithmique des services sont complètement modélisées, le code exact (en assembleur) de l'implémentation finale de l'OS n'est pas nécessairement utilisé. On peut se servir des facilités de bibliothèques/langages objets comme les environnements de simulation comme SystemC, SpecC. Cela permet une exécution native, mais nécessite de modéliser convenablement et avec suffisamment de détails les caractéristiques de l'OS et de l'application. Il faut en particulier fournir un moyen de spécifier l'écoulement du temps afin de pouvoir le simuler, mais aussi des moyens d'interagir avec l'environnement des autres processeurs et périphériques du système, de la co-simulation en somme, ce qui n'est pas évident. C'est ce type de modèle que nous nous proposons de construire dans cette thèse.

Modèle d'implémentation finale de l'OS : Une simulation complète et précise nécessite de simuler le processeur cible ainsi que le véritable OS, avec les codes assembleurs de ses services, de manière à modéliser précisément le déroulement des interruptions et autres points critiques. Cela permet de valider les fonctionnalités de l'OS final mais aussi de pouvoir déverminer [15] tout le code, y compris l'application. Pour cela il faut disposer de la machine cible ou d'un ISS adéquat, et bien évidement du code final de l'OS porté sur la cible avec le compilateur adéquat, ce qui nécessite beaucoup de travail, que ce soit pour la conception de l'ISS, du compilateur associé, ou la génaration du code de l'OS On peut noter la popularité de l'émulateur QEMU qui permet d'émuler de nombreux processeurs.

2.5.2.1.2 Simulation avec un ISS Les simulateurs de jeu d'instruction de processeur s'exécutent sur une machine de développement standard et permettent de reproduire le comportement exact d'un logiciel exécuté sur un processeur cible. Cela permet de valider une application même lorsque le processeur visé n'existe pas encore. Généralement, cela permet d'étudier pas à pas l'état du processeur (registres etc.) au fur et à mesure de l'exécution des instructions. Ce genre d'outil est fort utile pour les concepteurs de processeurs et des compilateurs associés, lors de la phase de conception, avant de passer en fonderie. La plupart des ISS sont basés sur le principe d'interprétation, comme par exemple dans SimpleScalar [LLC]. Le simulateur construit une représentation logicielle des structures de données en mémoire et de l'état interne du processeur. Et ensuite il le tient à jour en exécutant une boucle qui procède à l'exécution des fonctions principales d'un processeur :

15. francisation de l'anglissisme *déboguer*

- Fetch : Lit un mot d'instruction depuis la mémoire simulée,
- Decode : Analyse l'instruction pour en extraire et décoder les signaux de contrôle pour l'unité de traitement,
- Lecture des données,
- Execute : Répartit les données vers l'unité virtuelle de traitement désirée. Dans ce cas l'ISS se branche vers le code de l'instruction simulée et met à jour l'état du processeur selon l'exécution simulée de l'instruction,
- Écriture des résultats dans la mémoire simulée.

Ce procédé de simulation de jeu d'instruction par interprétation est flexible et efficace, mais la simulation est assez lente du fait du temps de décodage et d'interprétation de chaque instruction.

L'avantage de l'utilisation d'un ISS est que le code utilisé pour la simulation est identique à celui de l'implémentation finale et les informations sur les durées des opérations permet de profiler facilement une application. On peut par ailleurs instrumenter l'ISS en y ajoutant, en plus des durées des opérations, la caractérisation de la consommation par instruction [FAMH08], ce qui permet d'obtenir une estimation de la puissance consommée, bien que d'autres paramètres autres que les seules instructions soient à prendre en compte (les valeurs des bits des données qui conditionnent les transitions), notamment dans le cas avec communications multiprocesseur, ou leur modèle montre une erreur de 17%. Ces travaux ont débouché sur l'outil d'évaluation Hellfire [AFM+10] qui permet d'explorer le mapping des tâches sur de multiples processeurs, et d'observer la consommation et les taux d'utilisation (processeur et bus) et de *deadline miss*, en fonction de la fréquence du tick OS et des processeurs, au prix d'une lente simulation sur leur ISS amélioré.

Un autre procédé consiste à compiler la simulation, ce qui est connu sous le terme de *static translation*. Le programme est alors décodé et compilé en un binaire optimisé pour la simulation. Cela peut grandement améliorer les performances (12 MISPS [16] pour [RMD03] sur des Pentium3 en 2003), mais ce n'est pas très flexible car tout le code doit être connu à l'avance. Ce n'est donc pas très adapté à des applications qui modifient leur code ou chargent dynamiquement des librairies.

Une troisième technique consiste à *translater dynamiquement* ([RMD03]) des suites d'instructions en une représentation intermédiaire. Pour cela il faut préalablement décoder les instructions et les transformer en une représentation intermédiare permettant une interprétation à la volée. On peut espérer une accélération si les blocs d'instructions ne sont décodés qu'une fois et si lors des exécutions suivantes on récupère directement du cache mémoire leur interprétation. Cela à l'avantage d'être flexible et d'accélérer le décodage répétitif (cas des boucles). On peut mentionner que les langages interprétés (Java, Perl etc.) fonctionnent eux aussi comme des ISS dans leur principe, bien que la cible matérielle n'existe pas, ce qui explique leur lenteur inhérente.

Ces ISS basés sur l'interprétation et la compilation donnent des performances acceptables, avec une bonne précision au niveau cycle. On peut trouver aujourd'hui des ISS qui simulent jusqu'à 1600 MIPS pour des processeurs ARM [Ini]. Mais ces interpréteurs sont peu portables et demandent une forte capacité mémoire. De plus, ils ne sont plus du tout adaptés à des simulations multiprocesseur, où le temps de simulation va devenir critique, mais encore moins lorsque l'on veut pouvoir y intégrer des périphériques, notamment pour prendre en compte l'environnement et les interruptions qu'il génère. Dans ce cas, il est de plus en plus évident que le choix d'augmenter le niveau d'abstraction de la simulation s'impose.

Une technique pour accélérer la simulation est de se passer d'ISS et d'exécuter simplement une simulation *native*, c'est à dire de ne pas modéliser l'architecture en utilisant directement le processeur de la station de travail. Cela présuppose de programmer dans un

16. Millions d'Instructions Simulées Par Secondes

langage de haut niveau (autre que l'assembleur et donc portable) que l'on peut compiler pour la station hôte, ce qui permet d'accélérer énormément la simulation. Mais alors on ne pourra utiliser la partie codée en assembleur du code de l'OS spécifique au processeur cible. Or cette partie en assembleur est nécessaire pour utiliser l'OS, notamment le code assembleur spécifique à l'architecture cible relatif aux appels systèmes (à ne pas confondre avec ceux de l'OS de la machine hôte de la simulation). On a donc besoin d'utiliser un moyen de rediriger les appels systèmes du logiciel simulé vers ceux du modèle de l'OS. La principale difficulté réside dans la conception d'un tel modèle qui peut avoir un grand impact sur la fiabilité de la simulation. C'est ce genre de modélisation que nous allons étudier dans cette thèse.

2.5.2.2 Modélisation conjointe d'architecture et OS à haut niveau

De nombreux travaux ont été menés cette dernière décennie pour intégrer les systèmes d'exploitation au processus de conception rapide, et plus particulièrement au niveau de la modélisation et de la simulation.

Certains travaux se sont focalisés sur l'automatisation de la génération de RTOS et de leur code pour le logiciel embarqué. Cela permet de paramétrer certains services, particulièrement l'ordonnanceur, et de générer le plus petit OS correspondant aux services minimum nécessaires pour une application donnée. Les travaux sur Colif [GYJ01] appliquent ce genre d'approche, en particulier pour générer les *wrappeurs* logiciels et matériels nécessaires à l'implémentation des communications distribuées entre processeurs hétérogènes.

Un des inconvénients de ces méthodes de génération de RTOS : la focalisation sur le logiciel embarqué ne fournit pas d'information sur les aspects temporels et ne permet pas l'exploration du système dans son ensemble, en particulier la modélisation des IP périphériques ou accélératrices, comme c'est le cas dans les MPRSoC. Il est donc nécessaire d'utiliser un modèle permettant d'intégrer dans une même simulation l'exploration de l'OS et des services spécifiques adaptés à la fois aux tâches logicielles, mais aussi à l'architecture. Cela devrait permettre de gérer les contraintes de l'embarqué et des comportements dynamiques attendus dans un environnement unique.

La première tendance dans la modélisation au niveau système a été de créer ou étendre des langages existants. Le but des langages de description de niveau système (SLDL) tel que SpecC et SystemC est d'augmenter l'utilisation du C/C++ pour les spécifications d'extensions (bibliothèques/langages) fournissant les concepts systèmes nécessaires, comme modéliser le temps, la concurrence et les liens de communications. Ce genre de concept est particulier à la modélisation de systèmes numériques matériels, et nous verrons qu'il est intéressant de s'en servir comme support à la modélisation conjointe pour intégrer facilement le logiciel aux modèles de blocs matériels décrits avec ces langages.

Dans le domaine des approches fondées sur la simulation pour la conception MPSoC, SystemC est de facto un langage pour la modélisation des systèmes. SystemC permet de résoudre les problèmes liés à la conception MPSoC à plusieurs niveaux d'abstraction dans un même simulateur. D'autres langages basé sur le C/C++ ont été proposés tels que SpecC [GYG03] et SoCOS [MVD00].

Dans [HWTT04], des bibliothèques C/C++ sont mises en œuvre pour fournir un modèle de RTOS et une API d'accès aux fonctionnalités d'OS aux applications. Des mécanismes supplémentaires pour s'interfacer avec un simulateur HDL y sont également discutés.

Dans [GYG03], SpecC est à la base d'un modèle de MPSoC centré sur l'OS.

Une méthodologie, basée sur la modélisation abstraite, a été présentée dans [TCM01], la méthode est basée sur SpecC et se limite au modèle de préemption de l'ordonnanceur pour les systèmes mono-processeur.

SoCOS est une autre librairie C++ équivalente à SystemC conçue par Desmet *et al.* [MVD00] destinée à la modélisation d'OS, en fournissant des facilités d'intégration des nouveaux services d'OS, notamment en permettant d'élaborer plusieurs horloges et des processus dynamiques. SoCOS propose trois modèles de calcul : asynchrone, réactif (les interruptions) et synchrone (RTL). L'inconvénient est que cette approche est basée sur un noyau de simulation propriétaire et que la gestion des interruptions semble problématique. En revanche, SoCOS permet un raffinement automatique entre l'API simulée et celle de l'OS raffiné.

Les travaux initiés par Gajski et son équipe [GYG03, YGG03] pour modéliser un RTOS à haut niveau se sont donc basés sur le langage SpecC, un des premiers du genre. Ils ont utilisé les primitives de ce langage pour décrire le comportement dynamique d'un système multi-tâche à haut niveau. Gerstlauer, Yu et Gajski proposaient ensuite une méthode de raffinement permettant l'implémentation aisée et automatique, sans pour autant résoudre le problème de gestion des interruptions dépendant de la granularité temporelle explicite des tâches. Leurs travaux ont aussi initié l'abstraction des communications et la modélisation TLM [CG03a].

La modélisation au niveau transactionnel (TLM) a été largement utilisée pour la conception de modèle de MPSoC, par exemple [CG03b, FPG+03, GLMS02, OBdNC+03, PDBR04]. On peut à ce sujet mentionner l'apparition récente de la librairie UNISIM [ACG+07] reposant sur SystemC focalisée sur la modélisation de communications de type transactionnelle, et facilitant l'exploration d'architectures. L'utilisation de SystemC dans la modélisation TLM, discutée dans [OBdNC+03, PDBR04], est principalement motivée par la facilité de décrire des modèles de bus ou NoC [GSDG05] de MPSoC à différents niveaux d'abstraction.

Dans [BBB+03] Bennini adresse la question de la co-simulation et de l'émulation d'un modèle de MPSoC décrit en SystemC pour une simulation cycle-accurate. Des modèles de réseaux cycle-accurate sur la base de SystemC ont été proposés dans [LAB+04] (MPARM) et [FMP+04]. Le modèle cycle-accurate MPARM intègre un ISS dans un simulateur SystemC. Pour le système d'exploitation, le portage de RTEMS [Cor] est disponible. Cela permet une analyse d'ordonnancement détaillé du RTOS, y compris de préemption. La représentation cycle-accurate, tout en étant très précise pour l'évaluation des performances de RTOS, a cependant des effets sur la vitesse de simulation et de l'évolutivité du système. En outre, le temps requis pour évaluer l'impact sur les performances des changements relativement mineurs dans les systèmes modélisés de la sorte est, souvent, rédhibitoire.

Les nombreux autres travaux de modélisation d'OS sont dans l'ensemble basés sur SystemC. SystemC a adopté et amélioré de nombreuses caractéristiques de SpecC, comme la communication à base de canal abstrait et le support de la modélisation de comportements matériels, communément décrit en langage spécifique (HDL). SystemC, reposant sur des mécanismes de communication et de synchronisations événementiels, a diminué la nécessité de construire de lourdes bibliothèques personnalisées pour modéliser les interactions inter-processeurs. Toutefois, l'ajout de support aux exigences de la modélisation de flux logiciels et de RTOS dans SystemC, nommé SystemC 3.0 comme annoncé en 2002[Gro02], tel que la préemption et l'ordonnancement, est resté en suspens [BRS07], bien que la création dynamique de thread ait été introduite dans la dernière version 2.2 de SystemC.

Des modèles conjoints de MPSoC et RTOS de niveau système pour l'exploration

de l'espace de conception des applications temps-réel, ont été proposées dans [GYG03, HRR+04, HWTT04, MPC04, PAS+06].

Les travaux de Gajski et Gerstlauer se sont poursuivi pour proposer un framework complet (CSE) [DGP+08] pour concevoir et explorer un système, de la méthodologie [SSGD07], en passant par l'annotation automatique des blocs de code du simulateur [HAG08], à la génération automatique [YDG04] du code final de l'implémentation du RTOS. Par la suite, Schirner ayant travaillé avec eux sur le modèle d'OS [SGD07] et la génaration automatique des communications logiciel/matériel [SGD08] s'est inspiré de leur modèle, mais cette fois-ci en utilisant SystemC pour proposer un modèle plus précis de de la préemption [SD08] avec la particularité qu'il modèlise les tâches logicielle par des pthreads synchronisé au SC_thread SystemC du RTOS par des variables conditionelles [HSA09]. Gesrtlauer a aussi fini par translater son modèle de RTOS abstrait de SpecC vers SystemC [ZMG09].

R. Le Moigne ([MPC04],[MCP05]) applique la méthode MCSE de l'école polytechnique de l'université de Nantes qui est utilisable à l'aide de l'outil CoFluent Studio, afin d'inclure l'OS dans la simulation. Elle part du modèle fonctionnel TLM vers SystemC, pour une évaluation en trois étapes, avec processeurs seulement, avec des liens de communication simples, puis avec des liens et interfaces physiques précisés, selon plusieurs types (bus, routeur, point à point). Pour Le Moigne [MPC04], la fonctionnalité de RTOS a été intégré dans l'exécution des tâches. La préemption est modélisée par des appels de méthode dans la classe SystemC de la tâche. La motivation de cette approche est de réduire les surcoût de simulation de communication avec une couche OS indépendante, qui ne serait pas extensible en cas d'un très grand nombre de tâches. Toutefois, l'inconvénient de cette approche est de réduire la flexibilité dans la réalisation de différents schémas d'ordonnancement car ils doivent être soigneusement intégrés dans l'exécution des tâches.

Posadas dans [PVB05, PAS+06, PAV+05] se focalise sur la modélisation de RTOS (basée sur des communications par canaux) aussi précise que possible quant aux contraintes temporelles du matériel, dans un contexte MPSOC. Il propose une bibliothèque SystemC reprenant l'API POSIX qui modélise la préemption. Pour les interruptions matérielles, au lieu d'une suspension spontanée de la tâche, un délai égal au temps entre la préemption et l'événement de reprise est suivi, et ajouté à la fin de l'exécution de la tâche, maintenant ainsi la cohérence de synchronisation dans le modèle. Dans [PAS+06] il améliore son travail sur *PERFidy* en se focalisant sur l'accès à une variable partagée, puis conçoit un outil, SCope [PQVM07], qui permet d'analyser un système simulé dans son ensemble, avec le notable avantage par rapport aux autres approches : les estimations des durées d'exécution du code sont obtenues automatiquement grâce à un procédé de surcharge des opérateurs C++, implémenté dans PERFidiX. .

L'approche suivie par Sifakis [Sif01] décrit une méthodologie basée sur le principe de composition pour modéliser des systèmes temps-réel bien que l'approche n'est pas étendue à la modélisation de systèmes temps-réel mis en œuvre sur des plateformes multiprocesseurs hétérogènes.

Dans [CPT03], Cassidy présente un modèle de performance de haut niveau pour systèmes multiprocesseurs multi-threadés. Cette approche est basée sur la modélisation d'une couche abstraite d'ordonnanceurs qui ressemble à l'objectif de notre approche.

Le framework ARTS [MVM07] combine de nombreuses caractéristiques des modèles décrits ci-dessus. Il fournit une couche d'API de RTOS reposant sur SystemC, semblable au modèle basé sur SpecC [GYG03], et utilise les principes de la décomposition décrits par Sifakis [Sif01] pour réaliser les caractéristiques d'OS. Cependant, le cadre de modélisation ARTS est significativement différent des modèles d'encapsulation purement comportementaux du code de l'application et des simulateurs détaillés au niveau cycle-accurate. L'accent est mis sur l'exploitation de la décomposition du modèle et la mise en œuvre de

la capacité de préemption en SystemC. Cette capacité de préemption est cruciale pour capturer le comportement complexe entre le RTOS et l'application. Leur modèle est centré sur l'OS qu'ils décomposent en éléments indépendants (tâches et gestionnaires) qui communiquent par messages événementiels. Dans ce modèle l'OS est représenté par trois éléments [MVG03] : l'ordonnanceur sur nœud de calcul abstrait, le gestionnaire/allocateur de ressources, le gestionnaire de synchronisation. De plus la gestion des communications et entrées/sorties est modélisée par une tâche et un bloc de communication dédié. Cela leur permet d'étudier plus en détails la modélisation du réseau de communication, notamment les problématiques de conception de wrappeur entre niveaux de modélisation des communications (OCP). Les tâches gèrent leur état et sont synchronisées selon une horloge globale, ce qui permet plusieurs changements d'état entre deux synchronisations. Par ailleurs, le fait d'utiliser des métriques et un mapping explicite facilite une exploration rapide en générant plusieurs jeux de plateforme et mapping associés automatiquement. Néanmoins, la gestion des interruptions n'est pas suffisamment précise pour modéliser convenablement la préemption.

He, Mok et Peng ([HMP05]), ont produit un outil en SystemC fournissant une API modélisant les OS Linux et DSP/BiOS. Cela inclue une modélisation des entrées/sorties très étoffée avec un modèle générique de driver et de données échangées qui peut s'adapter à n'importe quel driver, avec des fonctions *callback* pour les ISR et une interface standard (open/close, R/W et IO_CTRL). La modélisation du temps repose sur des annotations succinctes dans le code, et l'exactitude de la prise en compte des interruptions repose sur la possibilité de prédire ces dernières (ce qui n'est pas toujours possible).

Hastono [HKH04a, HKH04b] crée un module SystemC qui comprend les services minimaux d'un OS, à savoir la gestion de tâche, l'ordonnancement et la gestion du temps et des synchronisations. Il met l'accent sur la possibilité de changer d'ordonnanceur (RM, EDF, déclenché sur évènement ou temps). La particularité de son modèle est de partir d'un graphe de tâches caractérisées par leur période, deadline et leur durée (WCET), et d'avoir une extension qui permet de rendre les durées variables en utilisant une densité de probabilité de Gumbel pour les tâches de durée variable. Son modèle de préemption reste cependant sommaire, car il peut se produire uniquement en fin de période insécable d'appel à la fonction d'avancement du temps de l'OS (*wait*).

Hessel *et al.* [HRR+04, HRR+06, HMS07] propose un flot de conception et un modèle de RTOS, en étendant SystemC. Le flot demande de décider dès le départ quelles parties de l'application seront implémentées en dur. Ensuite il redécoupe les tâches en sections unitaires, puis partitionne ces (mini-)tâches en clusters répartis sur les processeurs. Les tâches doivent ensuite inclure les primitives du RTOS au niveau TLM (communications interprocessus de type évènementiel, mémoire partagée, FIFO, et bus) et le choix entre plusieurs types d'ordonnanceurs (FCFS,RR,RM, EDF) permet d'évaluer le comportement de l'application. Néanmoins les tâches simulées ne sont qu'une représentation non fonctionnelle (limité aux paramétres priorité, périodicité, WCET, BCET après profiling) et la gestion des interruptions dépend de la granularité temporelle explicite des tâches, ce qui peut mener à une inversion de la suite temporelle de la simulation. Ils proposent aussi dans [VOM+06] une possibilité de raffinement de l'ordonnanceur déporté sur une unité de traitement distincte, un autre processeur (SoRTS) ou une IP matérielle (HaRTS), afin de réduire la durée de context switch de l'OS.

Une approche différente est proposée dans [CBR+03, CBR+04a, CBR+04b] par Chevalier *et al.*, où les tâches logicielles sont toujours encapsulées dans des modules SystemC mais plutôt que d'intégrer un modèle de système d'exploitation dans SystemC, c'est un système d'exploitation à part entière qui ordonnance les modules représentant des tâches logicielles. Cela confère à ce type de solution plusieurs avantages, notamment une certaine simplicité dans le mécanisme de raffinement permettant de passer entre les différents ni-

veaux d'abstraction existants. Elle permet également de faciliter le processus d'exploration d'architecture en modifiant directement le type (logiciel ou matériel) des modules SystemC et offre la possibilité d'exécuter et donc de valider une partie importante du logiciel final.

Hassan [HSTI05b, HSTI05a] a été un des premiers à proposer un moyen de simuler un RTOS en SystemC. Son modèle repose sur une librairie de simulation qui comprends des appels systèmes basés sur l'API standard RTK-spec I, II et TRON et le service de simulation temporelle `SIM_wait()`. Ce service permet d'attendre des *tokens* qui sont des évènements de synchronisation des tâches (toutes périodiques, modélisées par un réseau de Petri) sur l'horloge système ou sur les évènements de l'OS ou des périphériques matériels d'un microcontôleur 8051 modélisés au niveau BFM (TLM, CA ou RTL au choix). Les appels systèmes sont tous atomiques et les interruptions peuvent être déclanchées en cascades. Pour utiliser son simulateur, il faut annoter le code avec les primitives de simulations (presque le double de code [Has06]), afin de pouvoir remonter les informations de consommation de temps et d'énergies (basés sur des estimations), ou évaluer la meilleur configuration d'assignement de graphes de tâches et communications sur une architecture multi-processeurs[HOI07].

2.5.2.3 Gestion de la reconfiguration

Avec la possibilité de concevoir des tâches matérielles comme des tâches logicielles (ou presque), apparaît la même problématique qu'avec les premiers processeurs, où la gestion complexe des ressources finit par nécessiter un OS, comme le faisait remarquer une équipe de chercheurs australiens en 1999 [DW99].

Par exemple, Zhou [ZCQ+05] ne propose pas de modélisation, mais une extension du RTOS μC-OS-II (appelé SHUM-COS), qui permet de lancer des tâches matérielles sur le FPGA comme des tâches logicielles. Il se base sur une table de configuration préétablie (par analyse du graphe de tâche), afin de palier à toutes les demandes potentielles de l'application. Les IP doivent être encapsulées de manière à, entre autre, inclure la gestion de ces tâches matérielles.

C'est aussi une extension d'un OS existant, RTAI, que proposent Nollet *et al.* avec leur OS OS4RS [NCV+03] pour gérer des tâches matérielles capables de communiquer à travers une infrastructure générique associée aux zones reconfigurables qui implémentent matériellement le service de communication.

Le projet EPICURE([ACD+03, DGP+06]) s'intéresse au partitionnement logiciel/matériel d'une application sur SoC. Il se base sur le formalisme mathématique de Esterel (SSM) et permet la génération automatique de code C++/VHDL et l'optimisation et l'exploration d'architecture à l'aide de métriques. Ces dernières aidant un algorithme générique à maximiser les contraintes. Une interface ICURE est utilisée pour gérer les communications entre IP et le logiciel : les appels aux IP se font comme pour de simples fonctions, le tout géré par un gestionnaire de reconfiguration (gros grain) hardware.

Dans [VGBP03], est présenté le paradigme MOLEN qui consiste à ajouter des instructions spécifiques (des `#pragma` utilisé à la compilation) pour gérer la ressource reconfigurable, avec quelques éléments matériels spécifiques, un compilateur amélioré et des directives d'écritures de codes C assez légères. Ce principe, mis en oeuvre notamment dans le projet MORPHEUS [UE], permet de charger les fonctions accélaratrices matérielles à la volée et de prendre en charge le transfert des paramètres de manière presque transparente pour l'utilisateur.

2.6 Synthèse

La conception d'un système sur puce dédié à une application est un processus complexe, pour lequel il est recommandé de recourir au prototypage rapide afin de trouver et valider la solution mixte logicielle et matérielle garantissant un fonctionnement convenable, avant de déclancher le travail de conception réel.

Nous venons de voir un panorama de solutions existantes pour modéliser conjointement une plateforme SoC reconfigurable, avec le logiciel qui s'y rattache. Néanmoins, nous n'avons pas trouvé de solution qui permet dans un environnement unique de concilier les aspects reconfigurables et autorise l'exploration des services de gestion du système de manière fonctionnelle et temporelle, et à haut niveau.

Aussi on se propose de permettre de simuler des tâches logicielles et des systèmes d'exploitation dans l'environnement SystemC. Cela permetra d'explorer et valider par simulation, rapidement et au plus tôt dans le cycle de développement, les choix architecturaux de répartition logiciel/matériel des tâches et des services d'OS. Pour cela nous devons concevoir un modèle exécutable générique de système multiprocesseurs hétérogènes reconfigurable, qui se doit d'être fonctionnel et réaliste (temporel et dynamique), tout en autorisant d'explorer les services d'un système d'exploitation gérant ce type de plateforme.

Chapitre 3

Modélisation de systèmes embarqués basée sur l'OS

Sommaire

3.1 Introduction

Ce chapitre présente notre proposition de modélisation de système complet (logiciel et matériel) centré sur le chef d'orchestre qu'est le système d'exploitation. Nous verrons

tout d'abord le sens pris par le terme OS dans notre modèle qui s'étend à la notion plus générale de système d'exploitation. Cette modélisation s'inscrit au cœur du framework OveRSoC proposé pour évaluer rapidement des applications implantées sur systèmes embarqués comportant plusieurs nœuds d'exécution ainsi qu'une partie reconfigurable. Nous expliquons ensuite comment implémenter un modèle d'OS en SystemC, puis comment le rendre modulaire afin de faciliter l'exploration de ses services.

3.2 Définition du modèle de système basé sur les services d'OS

Ce chapitre va détailler notre méthodologie de modélisation de système basée sur le point de vue du gestionnaire de plateforme (le SE) et des services fournis par ce dernier. Nous choisissons la modélisation car un modèle permet d'abstraire/capter une partie de la réalité intéressante et de masquer certains détails parfois inutiles. Par ailleurs la simulation a de nombreux avantages : cela permet d'explorer des idées sans coût de réalisation de prototypage, et de pouvoir instrumenter intégralement et de manière non intrusive tout en contrôlant l'environnement du système observé.

3.2.1 Choix de conception pour l'exploration *architecturale* du système

Les applications logicielles peuvent être vues comme un ensemble de portions de code (calcul), nommées processus ou tâches. On ne considérera pas ici la particularité des threads POSIX ou les données sont implicitement partagées, ce qui a une incidence non négligeable lors du déploiement sur plusieurs nœuds d'exécution.

Dans les systèmes temps-réel, chaque tâche peut être annotée avec des contraintes temporelles ou de priorité, selon le comportement de l'application désirée, lors de la phase de spécification. Par exemple la périodicité ou le délai maximum d'exécution sont des critères souvent utilisés pour ordonnancer une tâche.

Le système d'exploitation est une couche logicielle fournissant une interface générique d'abstraction du matériel (processeurs et périphériques) offerte aux tâches d'une application. Un OS peut aussi être vu comme un fournisseur de services d'accès aux ressources (matérielles) avec contrôle de leur répartition correcte (gestion mémoire, ordonnancement et préemption, gestion des interruptions...), ainsi que des services d'interaction entre tâches (synchronisation et communication...).

Dans nos travaux, nous nous intéressons essentiellement aux services suivants :

- Ordonnancement : L'ordonnanceur est responsable de l'ordre d'exécution des tâches sur les unités de traitement (processeurs et zones reconfigurables). La politique d'ordonnancement choisie peut avoir un fort impact sur les performances. Il s'agit du point essentiel de tout RTOS.
- Traitement des interruptions : Comme il ne peut y avoir qu'un seul processus s'exécutant à un instant donné dans un processeur classique, l'OS doit être capable d'interrompre ce dernier pour traiter une interruption externe et déclencher la routine (logicielle) associée, puis redonner la main au processus interrompu. Cela s'appelle la préemption et consiste en une sauvegarde des éléments matériels supports des calculs en cours (le contexte d'exécution : registres etc.) afin de les libérer pour que le nouveau flux d'exécution puisse utiliser pleinement les ressources CPU, puis les restaurer après la fin de ce traitement.
- Gestion du temps : Dans un système temps-réel, il est important de pouvoir maîtriser cet aspect, et fournir des fonctions de mesures temporelles et de délai.

- Synchronisation de tâches : De manière à permettre aux tâches d'interagir entre elles, on doit considérer un service de synchronisation (par exemple par sémaphore), permettant de protéger des ressources partagées, et de se synchroniser pour respecter un certain ordre logique.
- Communications interprocesseur : Il va sans dire que pour gérer convenablement plusieurs nœuds d'exécutions concurrents, ces derniers doivent être à même de pouvoir communiquer entre eux ; entre processeurs, mais aussi avec les périphériques concurrents.

La politique d'ordonnancement est un des facteurs de performance majeur d'une application embarquée, mais le surcoût du changement de contexte, la vitesse du bus ou des accès mémoire, ainsi que le partitionnement en tâches de l'application peuvent aussi lourdement impacter le comportement final de l'application. De ce fait, nous plaidons la nécessité d'une modélisation qui permette d'évaluer par simulation les futures performances d'une application et d'un OS dédié selon ces paramètres. Cette modélisation permettra de mieux définir quelle politique d'ordonnancement ou de partitionnement satisfera au mieux les contraintes temporelles fixées.

Le système d'eploitation d'un système MPSoC peut prendre en compte bien d'autres services, comme ceux décrits dans la machine virtuelle Ter@ops (voir chapitre 5.3), mais notre étude se limite ici à l'élaboration d'un modèle permettant de faire fonctionner et d'assembler entre eux les services de base d'un noyau d'OS temps-réel en SystemC, et ainsi permettre à un utilisateur d'explorer facilement de nouveaux services, et composer un système d'OS formant le système d'exploitation de sa plateforme.

Contraintes du modèle à construire

De manière à permettre l'exploration des choix de conception, on se propose de définir un modèle exécutable de RTOS avec les contraintes suivantes :

- Généricité : dans le sens où l'on veut favoriser l'exploration de différentes solutions, il doit facilement être étendu à de nouvelles implémentations. Cela peut être obtenu par une architecture modulaire des constituants du modèle.
- Gestion de l'hétérogénéité : on doit permettre l'interopérabilité des applications sur plusieurs modèles d'OS. Cela peut être rendu possible en utilisant une interface standard d'accès aus services.
- Gestion de zones reconfigurables : ces zones viennent apporter une certaine flexibilité au matériel, et la gestion de telles zones requière sans aucun doute un OS dédié, comme pour toutes ressources partagées et utilisées différemment dans le temps.
- Raffinement du modèle : on doit permettre différentes implémentations d'un même service, en conservant une même interface, qu'il soit en logiciel ou en matériel.
- Exploration du modèle : cette propriété est une conséquence des trois propriétés précédentes.

3.2.2 Caractéristiques nécessaires pour l'exploration

Afin de rendre notre modélisation conjointe réaliste et utile pour l'exploration, quelques caractéristiques doivent être prises en compte : la possibilité d'explorer les services, de les distribuer, de les raffiner vers l'implémentation finale et bien entendu pouvoir gérer les aspects reconfigurables.

3.2.2.1 Support de l'exploration des services : la modularité

En choisissant SystemC, on bénéficie de tous les aspects de la technologie de conception orienté objet et sa possibilité de polymorphisme. En effet, pour pouvoir tester différents algorithmes de services de gestion d'une plateforme MPSoC (que nous considèrerons comme

hétérogène et reconfigurable), nous choisissons de modéliser les services sous la forme de modules distincts, existants en librairies (ou à concevoir celon un patron générique) et interopérables. Ainsi en découpant les fonctionnalités de l'OS en services distincts on peut explorer leurs algorithmes de manière indépendante et par ailleurs composer un nouvel OS avec les services spécifiques aux besoins de la plateforme et de l'application temps-réel.

De nombreux OS existants sont déjà implémentés selon cette philosophie modulaire, notamment ceux basés sur des microkernels, mais ne considèrent qu'une implémentation logicielle des services.

3.2.2.2 Support au MPSoC et la distribution des services

Afin de pouvoir représenter un système distribué sur de multiples nœuds d'exécution, on doit pouvoir représenter des communications et échanges de données entre les tâches mais aussi entre les services répartis du système d'exploitation. Il nous faut donc permettre de modéliser les communications avec des temps de transmission afin d'obtenir un système réellement fonctionnel.

Par ailleurs notre modèle devrait aussi permettre d'explorer le partitionement matériel/logiciel de certains services d'OS, ce que SystemC permet basiquement de faire, mais il faudra être capable de communiquer entre le bloc raffiné et la couche logicielle HAL de l'OS qui interagit avec ces services matériels. On pourait ainsi implémenter une IP d'ordonnanceur qui normalement est un algorithme long à exécuter de manière logicielle, et obtenir de l'accélérateur matériel le numéro de la prochaine tâche élue en seulement quelques cycles.

Il faut donc pour résumer permettre de modliser les communications logiciel/logiciel, mais aussi logiciel/matériel.

3.2.2.3 Support de la reconfiguration matérielle

Historiquement, les OS sont apparus avec la nécessitée d'utiliser au mieux les ressources de calcul. Cela a tout d'abord commencé par la gestion de processus par lot, pour en arriver aux systèmes complexes d'aujourd'hui avec partage du temps, gestion des priorités et des communications, et surtout la gestion de la mémoire. Cela est devenu tellement crucial que l'on a déporté au niveau matériel les tâches complexes et récurrentes de l'OS comme la gestion du cache mémoire de la mémoire virtuelle sous la forme de MMU et la gestion des transferts avec des blocs DMA. Il en va de même avec la gestion des bibliothèques que les programmes utilisent : l'OS charge dynamiquement ces bibliothèques partagées qui contiennent des algorithmes généralement génériques. Aujourd'hui, de nombreuses bibliothèques sont implémentées sous forme de blocs IP pour une utilisation accélérée (car parallélisées) et sont intégrées soit sous forme de coprocesseurs de calcul (instructions spéciales MMX, SSE etc.) soit comme composants annexes, comme les modules de cartes graphiques.

Avec l'avènement de l'électronique reconfigurable, il est désormais possible de changer la fonctionnalité de ces zones matérielles et de les réutiliser pour différentes IP au cours du temps.

La gestion du chargement et déchargement de ces zones avec l'IP nécessaire pour une application à l'instant t revient fonctionnellement à la gestion du chargement de bibliothèques logicielles. On peut donc espérer que sous peu, l'OS sera modifié convenablement pour gérer cette fonctionnalité afin que l'utilisateur ne voit pas de différence entre utiliser une bibliothèque logicielle ou matérielle. La différence entre les deux réside dans le chargement d'une IP qui se fait en transférant non pas du code exécutable du disque dur vers la RAM (puis dans le cache selon les sous fonctions appelées) mais un *bitstream* vers le port de configuration du composant. Une autre différence notable est la forme des appels

de fonction d'une IP. En effet, l'IP étant (généralement) séparée du processeur, le passage des paramètres ne peut se faire comme un appel de fonction logicielle par empilage des paramètres dans la pile. Parfois ce ne sont plus des paramètres mais des flux d'entrée (un peu comme un pointeur sur un tableau n'ayant qu'une adresse de début pour la suite de donnée). Il en va de même pour la récupération du résultat. Cela passe généralement par l'utilisation de DMA pour le transfert des données en entrée et sortie, ainsi qu'une gestion de début et fin de procédure via des registres de contrôle et lignes d'interruptions.

Autrement dit, il devient intéressant de rajouter des services spécifiques de gestion de zones reconfigurables comme ceux apparus pour remplir des fonctions de gestion de mémoire et de tâches se partageant les ressources, et cela de manière automatique et transparente pour l'utilisateur.

3.2.2.4 Raffinement de l'OS

Généralement on entend par raffinement d'un OS le code compilé d'un ou plusieurs OS qui tournent sur un ou plusieurs processeurs. Or on oublie que le raffinement ultime d'un OS consiste à raffiner certaines de ses fonctionnalités sous forme matérielle. Les processeurs modernes ont été modifiés en vue d'utiliser un OS afin que leur fonctionnement s'exécute selon plusieurs modes (généralement modes utilisateur/superviseur). Cela permet de garantir la cohérence et la sécurité du système réalisé mieux que s'il était supporté uniquement par le logiciel et l'OS.

Raffinement logiciel, déploiement

Le raffinement classique d'un OS logiciel consiste à fournir la couche d'abstraction du processeur généralement appelée HAL dans laquelle on trouve souvent la gestion des interruptions, la gestion des registres des communications et de la mémoire. Dans notre cas, l'utilisation de SystemC implique d'écrire les services en langage C++. Il est donc aussi nécessaire de transformer les algorithmes des services en un langage compilable sur la cible visée (du simple C/assembleur généralement).

Raffinement matériel, déploiement

Des parties de la couche bas niveau d'accès et d'abstraction du matériel (HAL) sont de plus en plus implémentées en matériel, comme les MMU ou autre DMA. Avec un processeur soft-core, on peut maintenant envisager d'ajouter des instructions spéciales pour appeler des fonctions implémentées comme co-processeur en matériel ou pour de nouveaux services comme ceux liés à la gestion du reconfigurable. Notre modèle doit pouvoir envisager ce genre d'exploration.

Notre approche initiale consiste à modéliser le comportement de la partie logicielle (OS et application), en faisant abstraction dans un premier temps de la plateforme matérielle support de l'exécution. C'est donc une approche de haut en bas du point de vue du système d'exploitation généralement dévolu à un OS logicel, qui suit le principe de séparation des préoccupations.

3.2.3 Le principe de séparation des préoccupations

L'un des principaux défis de la méthode proposée est de maintenir le modèle d'OS aussi abstrait que possible pour des raisons d'exploration tout en assurant la précision de l'estimation de performance. L'OS est maintenu à un haut niveau de description afin de facilement ajouter, supprimer, et de déployer des services sans impact sur l'application cross-compilée en code binaire : l'application est compilée une seule fois et le concepteur peut, non seulement modifier et affiner la mise en œuvre des services, mais aussi évaluer le nombre de processeurs et d'accélérateurs reconfigurables dynamiquement (ARD) de la

plate-forme. Ainsi on peut déployer à la fois les tâches de l'application et les services d'OS sur un MPSOC. En conséquence, le modèle sépare les systèmes en trois couches indépendantes comme décrits dans la Figure 3.1 en suivant aussi le principe de la séparation des préoccupations de [KNRSV00].

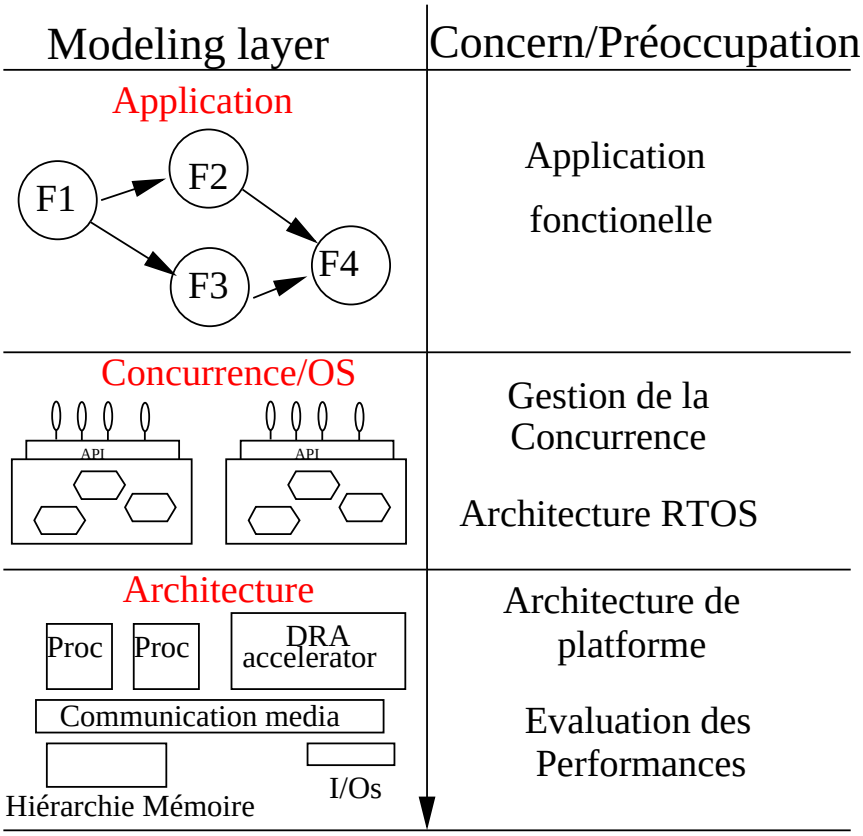


Figure 3.1: *Notre approche suit le principe de la séparation des préoccupations*

La couche supérieure se concentre sur la validation des spécifications fonctionnelles de l'application. Cela peut être décrit comme du pur code C fonctionnel partitionné en sous fonctions indépendantes.

Certaines de ces fonctions applicatives sont associées à la notion de tâche dans la couche suivante. Le code fonctionnel appelle les services d'un OS par le biais d'une API standard comme expliqué dans la section 3.3. Les communications et la synchronisation entre les tâches dépendent des services fournis par l'OS, comme les mutex, sémaphores, FIFOs, boîtes aux lettres, mémoire partagée etc.

Au cours de l'étape de raffinement suivante, la couche OS traite de la concurrence entre les processus logiciels explicitement définis. C'est dans ce but que nous avons développé le modèle d'OS décrit dans la section 3.3. Les tâches concurrentes sont également créées grâce à des services spécifiques de l'API de l'OS. Plusieurs algorithmes d'ordonnancement peuvent être testés à ce niveau, selon les contraintes de l'application et le mapping possible des tâches sur l'architecture sous-jacente sans modification de la couche fonctionnelle. Lors de cette étape, le concepteur peut aussi explorer l'architecture de services distribués sur les différents nœuds (ayant chacun un OS).

Enfin, à la couche *Architecture*, l'architecture exécutive du système embarqué est spécifiée comme une composition d'éléments hétérogènes de calcul (Processing Element, PE) et d'éléments de communication (CE). Chaque PE et CE peut être modélisé à différents niveaux d'abstraction et le raffinement peut être réalisé sans impact sur les autres couches de la modélisation. Précisément, l'ISS d'un processeur à usage général exécutant

une séquence d'instructions est un modèle de raffinement pour un bloc de calcul abstrait. L'indépendance de la couche matérielle est assurée par une API de bas niveau, la couche d'abstraction matérielle (HAL) qui rends toujours les mêmes services de bas niveau mais avec plus ou moins de précision. Cette couche est également responsable des paramètres d'évaluation : délai d'exécution, latence de communication, etc.

L'adoption d'une telle approche de modélisation permet d'atteindre l'objectif présenté, c'est-à-dire d'explorer la mise en œuvre de l'OS à un haut niveau, donc rapide, tout en offrant une évaluation suffisamment précise des performances de l'ensemble du système.

Nous avons observé sur une application, que le temps d'exécution des services de l'OS temps-réel représente environ 3 pour cent du total de l'application au moment de l'exécution. Cette observation est cohérente avec les résultats présenté par Kohout et. al dans [KGJ03] qui caractérisent l'overhead d'OS en fonction de la charge des applications. Les mesures d'overhead sont comprises entre 2% et 9% pour des RTOS préemptifs et entre 0.6% et 1.25% pour des stratégies non préemptives. Or ces observations sont valables pour un système mono-processeur. Dans notre cas, nous allons déployer des applications sur des SoC reconfigurables et surtout multi-processeurs, où les stratégies d'ordonnancement et les communications induites vont complètement changer le comportement du système et les temps d'attente des ressources à partager.

3.2.4 Niveau d'abstraction et modélisation *Service Accurate + Time* (SAT)

Afin de pouvoir explorer une nouvelle façon de développer un système nous pensons que nous devons modéliser un système-sur-puce du point de vue de son gestionnaire, le SE qui contrôle l'ensemble. Cela inclut le fait que ce gestionnaire peut être réparti sur plusieurs nœuds de calculs. Chaque nœud exécute un OS qui coopère avec les autres nœuds pour gérer de manière globale la plateforme. Un périphérique matériel aura virtuellement un OS minimal (et matériel) capable au moins d'être passif et fournir les service de traitement des requêtes venant des autres OS / nœuds, ne serais-ce que pour se synchroniser ou recevoir des données à "consommer". Les communications par interruption (périphérique vers processeur) puis lecture (sens inverse) de registres dédiés est une forme de communication souvent employé entre logiciel et matériel. Autrement dit on définit ici l'OS comme le gestionnaire d'un nœud de calcul et le système d'exploitation comme constitué de l'ensemble des OS.

La méthodologie standard de développement consiste soit à utiliser une conception top-down (depuis le point de vue général vers le cas particulier, du plus abstrait au plus détaillé) soit bottum-up, c'est-à-dire par construction à partir de briques préexistantes. Nous décidons de partir du niveau système pour autoriser à concevoir en fonction des besoins sans à priori sur les implémentations finales, et donc faire abstraction du support matériel. Cela permet aussi de simuler le comportement d'une application très rapidement puisque de nombreux détails y sont omis. A haut niveau, on peut ainsi se focaliser sur le comportement du système, généralement géré par un ou plusieurs OS et se concentrer sur les services fournis par la plateforme.

Notre modèle a besoin d'être fonctionnel (effectuer les calculs sur les données des tâches logicielles ou matérielles) et réaliste : il doit expliciter le temps afin de prendre en compte les comportements temporels imprédictibles comme les interruptions, qui généralement sont le point critique d'un système temps-réel.

Nous avons défini notre modèle en nous basant sur les services fournis par le système, qu'ils soient logiciels (généralement l'OS) ou matériels (contrôleur d'IRQ, MMU etc.). Nous appelons le niveau de description de notre modèle SAT pour *Service Accurate plus*

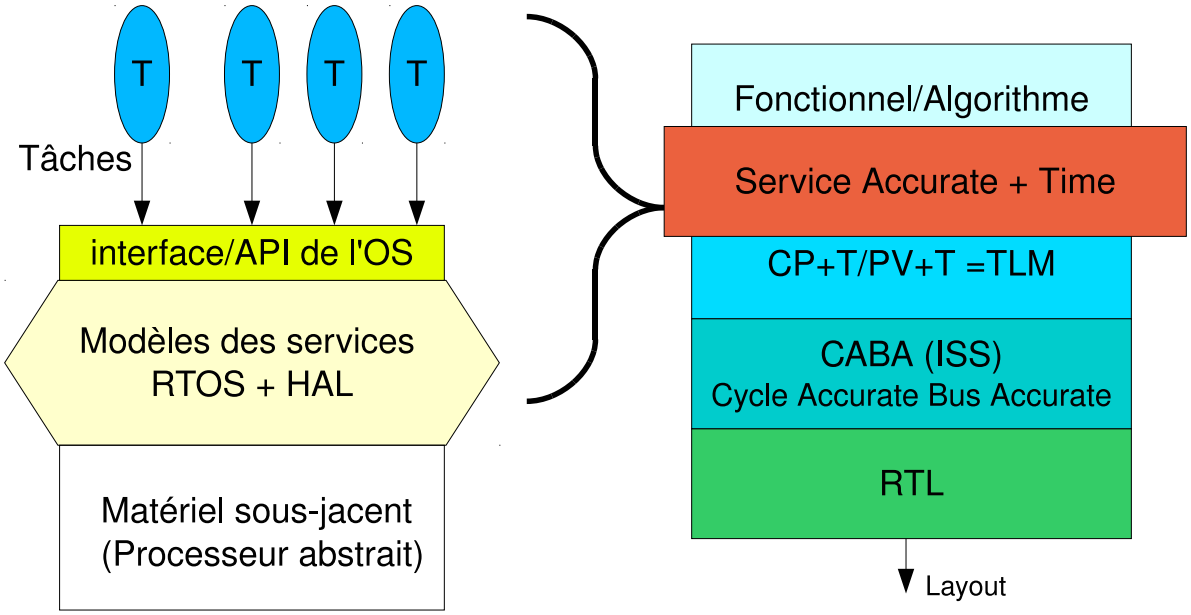


Figure 3.2: *Niveau de modélisation SAT*

Time, soit un niveau où les services sont décrits à haut niveau et temporisés, le reste (tâches et accélérateurs matériels) pouvant être définis de manière plus ou moins précise selon le besoin et le niveau de modélisation utilisé en SystemC pour les décrire.

Ce niveau de modélisation, tel que montré sur la figure 3.2, implique que l'architecture ne soit pas modélisée explicitement. Toutes les tâches sont fonctionnelles, annotées temporellement par des mesures approximées ou mesurées, voire simulées, et tous les services d'OS sont explicites et annotés temporellement.

On peut aussi comparer notre niveau d'abstraction aux niveaux définis par Donlin [Don04], qui introduit cinq niveaux d'abstraction, décris dans la sous-section 2.4.1 du chapitre précédent et la figure 2.12 page 33.

Nous pouvons considérer que notre niveau d'abstraction SAT est équivalent au niveau CPT, mais cette terminologie est surtout centrée sur le point de vue de la description matérielle. Or nous souhaitons nous focaliser sur les services d'OS. Dans cette couche on pourra explorer tout les services avec ce que cela implique en terme de dynamicité, et quelque soit la future décision prise quant à la forme de leur implémentation (logicielle / matérielle, localité et durée etc.).

3.2.5 Support de la simulation

La complexité de la modélisation d'un système sur puce dépend du choix du support de la modélisation et du niveau de détails désiré. Nous justifions ici des choix que nous allons utiliser.

Notre problématique centrée sur la modélisation d'un système MPSoC se focalise sur ses aspects comportementaux. Il est ainsi nécessaire de modéliser conjointement le support matériel comme la partie logicielle responsable des interactions. Comme décrit dans l'état de l'art, il existe plusieurs solutions pour modéliser un SoC dans son intégralité tant au niveau matériel que logiciel, de manière hétérogène (simulateur VHDL/Verilog couplé à des simulateurs de logiciel) ou homogène avec des langages supports comme SystemC, SpecC.

Le niveau de précision que l'on souhaite obtenir lors d'une simulation aura pour contrepartie la lenteur du simulateur. C'est pourquoi il faut pouvoir faire un compromis entre précision et rapidité. Avec SystemC, il est possible de faire un mélange hybride de plusieurs niveaux de précision dans une même simulation de système numérique. La méthodologie

Transactional Level Modeling (TLM) de SystemC [Don04] instaure à ce sujet une manière de concevoir les modules d'une simulation de manière à pouvoir interfacer ces différents modules de niveaux de modélisation/précision différents : niveaux TLM Algo, CP(T), PV(T) ou Cycle Accurate (CA) et RTL (voir figure 2.12 page 33).

Il faut noter que plus le modèle est fin plus il est lent en temps de simulation. Par ailleurs, le temps de simulation est aussi fonction du nombre d'éléments (et donc d'évènements SystemC) simulés. Dans notre cas, nous cherchons non seulement à simuler de nombreux nœuds de calcul, mais aussi la partie logicielle et l'OS, ce qui augmente la complexité du simulateur. Nous presentons donc une modélisation hybride matérielle/logicielle qui est un compromis entre précision et rapidité de simulation présenté dans la section 3.2.4 (SAT).

On peut donc raisonnablement utiliser SystemC pour sa capacité de modélisation en niveaux de précision hétérogènes mais aussi pour sa capacité à modéliser des communications temporisées. Par ailleurs notre modèle devrait aussi permettre d'explorer le raffinement de certains services d'OS sous forme d'implémentation en matériel, ce que SystemC permet de faire intrinsèquement.

Nous choisissons SystemC car bien que ne supportant pas nativement la modélisation du logiciel embarqué et des mécanismes liés aux OS, il est par essence conçu pour modéliser facilement les aspects concurrents du matériel. On a donc fait l'hypothèse de pouvoir l'étendre à notre besoin de modélisation conjointe au niveau logiciel avec l'OS, et ce sera là notre première contribution.

3.3 Construction du modèle d'OS

Nous décrivons dans cette section les mécanismes de modélisation adaptés pour définir une forme abstraite d'OS générique capable de s'adapter par sa construction modulaire aux besoins spécifiques d'applications embarquées sur puce.

3.3.1 Modélisation en SystemC de logiciel multi-tâches

Notre modèle est basé sur le langage SystemC, qui est un moteur de simulation et d'exécution en parallèle de blocs matériels, implémentés en langage objet C++. Il permet de modéliser l'exécution concurrente de processus de simulation au sein de conteneurs appelés modules (`sc_modules`) qui exécutent du code C++/SystemC (les processus d'exécution). La modélisation du temps d'exécution du noyau du simulateur SystemC est adaptée pour modéliser la concurrence contrairement au principe classique du temps partagé dans les microprocesseurs qui supportent l'exécution logicielle de manière séquentielle. Pour cela, le noyau SystemC traite les blocs de code fonctionnel de chaque thread/processus des modules en considérant qu'ils s'exécutent en un temps nul, jusqu'à ce qu'il rencontre la directive SystemC `sc_core:wait()` explicitant ainsi le temps consommé par la portion de code précédente, ou l'attente d'un évènement. A la rencontre de cette directive, le kernel SystemC s'arrête et procède à la simulation des autres threads simulables à cet instant donné. Il prend soin d'enregistrer la durée d'exécution (`wait()`) afin de pouvoir réveiller ultérieurement au moment convenu le thread/processus ainsi arrêté. Ensuite, le moteur avance la référence unique du temps uniquement lorsqu'il n'y a plus de threads de modules à exécuter à l'instant donné et avance le temps au plus proche instant suivant donné par l'une des directives `wait()`. Ce mécanisme implique de connaître à l'avance ou d'être capable d'estimer les temps d'exécution de chaque portion de code afin d'annoter convenablement ce code pour que SystemC puisse modéliser l'évolution du temps. Sinon, toute la simulation se déroule en un temps nul (niveau PV ou PVT si temporisé).

SystemC fournit aussi des mécanismes de synchronisation par évènements (`sc_event`), que les modules peuvent attendre indéfiniment. Chaque module peut aussi communiquer à l'extérieur avec les autres modules par l'intermédiaire de ports (`sc_port`) connectés à des signaux, qui peuvent être simples (`sc_signal`) ou plus complexes comme des bus abstraits (les `sc_channel`). Ces derniers sont en fait des modules SystemC particuliers fournissant une interface. Les modules peuvent communiquer à travers eux uniquement en appelant les méthodes de cette interface via leurs ports, comme montré dans la figure 3.3.

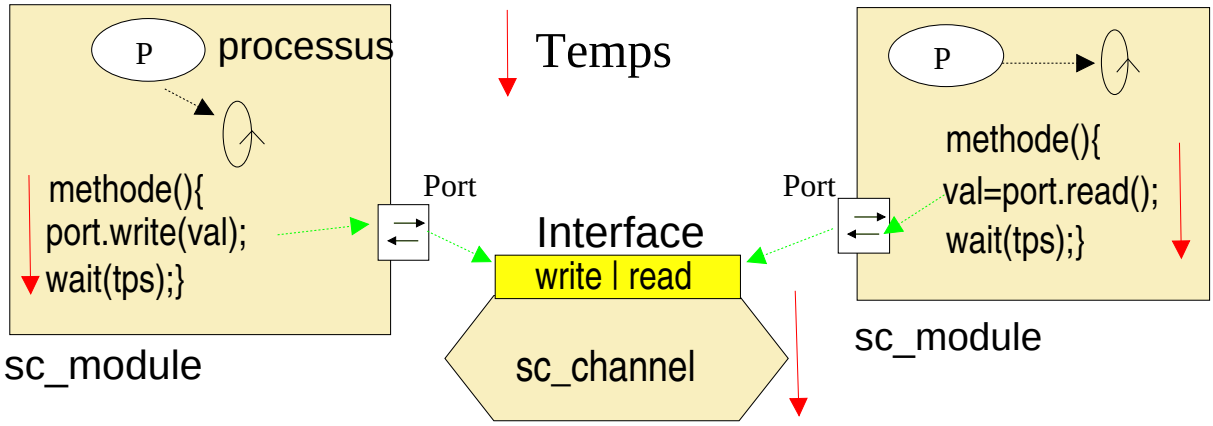


Figure 3.3: *SystemC simulant l'exécution des processus des modules en parallèle*

Comme pour d'autres langages de conception matérielle, il est possible de concevoir des modules composés de sous modules, ce qui permet d'avoir différents niveaux de hiérarchie. Pour notre modèle de système d'exploitation, nous utilisons largement les mécanismes de hiérarchisation et d'interface que fournit SystemC.

SystemC pour le logiciel et l'OS

Puisque SystemC exécute du code C++, il est possible de simuler des modules représentant la partie logicielle d'un système, tout en continuant de bénéficier du modèle de simulation concurrentiel et événementiel pour simuler conjointement le matériel.

Pour cela il faut quelque peu adapter l'application logicielle à SystemC. On peut le faire en voyant l'application comme une collection de fonctions C/C++ (les tâches logicielles) indépendantes.

Les tâches sont les unités de concurrence (potentielle) élémentaires qui s'exécutent sur le système et correspondent dans SystemC au modèle des threads POSIX, car dans SystemC, tout s'exécute dans le même environnement unique d'un seul processus UNIX qu'est le simulateur SystemC. Cette propriété est très pratique pour faire un prototypage rapide, mais peu précise quant aux transferts de données entre nœuds. Si l'on veut modéliser des processus type UNIX indépendants pour chaque tâche, ce qui est préférable pour évaluer la distribution multi-nœuds, alors il est préférable que les données globales partagées soient déclarées explicitement comme tel, comme pour des processus UNIX, afin de faciliter par la suite leur répartition et gestion sur des unités de calcul séparées et donc ainsi expliciter les échanges de données par ds appels système.

Nos tâches sont donc des fonctions normales écrites en langage C, contenant des appels système qui seront *linkés* à l'édition de lien par nos soins vers notre modèle d'OS.

Afin de les intégrer dans notre modèle SystemC, la méthode proposée est d'exécuter chacune des tâches comme le corps de processus/`SC_THREAD` distinct dans un (ou plusieurs) `sc_module` SystemC comme montré dans la figure 3.4.

L'OS est implémenté comme un canal hiérarchique (un `sc_channel`) qui est capable de fournir les services système à travers une API correspondant aux appels système standards

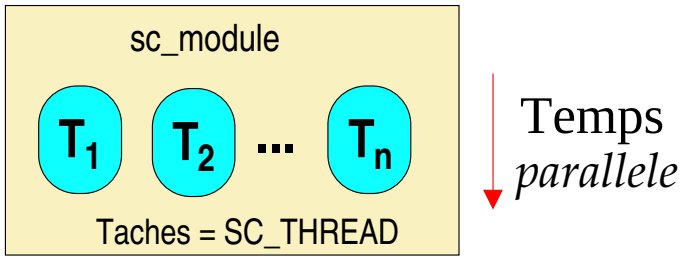


FIGURE 3.4: *Tâches concurrentes implémentées naïvement en SystemC*

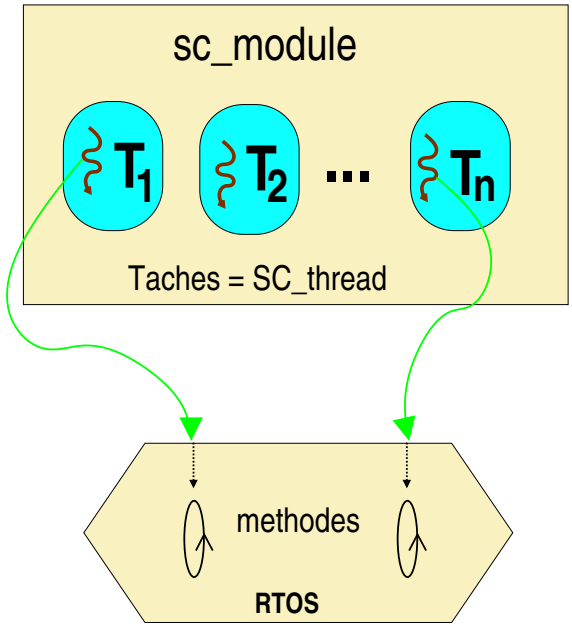


FIGURE 3.5: *Tâches accédant à un OS simple implémenté en SystemC*

(`syscalls`). De cette manière, les appels système appelés par chaque tâche ne sont plus des appels de fonction d'un OS standard, mais redirigés vers les méthodes de notre canal OS (Figure 3.5). Ainsi porter une application sur notre modèle d'OS demande de modifier la syntaxe de chaque appel système, mais cela peut être fait automatiquement en utilisant des macro-instructions masquant la transformation comme montré dans le listing 3.1.

Listing 3.1: *Gestion simple de redirection transparente pour l'utilisateur des appels système (API inspirée de µC-OS II)*

```
#define OSTaskCreate    os->create_task
#define OSMutexCreate   os->OS_create_mutex
#define OSMutexPend     os->OS_mutex_lock
#define OSMutexPost     os->OS_mutex_unlock
...
```

Comme nous souhaitons garder le code de l'application indépendant de l'OS (qu'il soit réel ou simulé), la redirection des syscalls se fait directement par un appel aux méthodes de l'objet "OS" global (ce qui est possible puisque SystemC est écrit en C++). Voici un exemple simple d'une application tournant sur un OS :

Listing 3.2: *Une application simple utilisant notre modèle*

```
#include "my_os.h"//l'objet OS maison qui fournit les system calls
#include "OS_interface.h" // son interface (API)
extern my_os* os; // La reference a l'objet global representant
    //le fournisseur de services c.a.d l'OS
OS_sem *s;//mutex global
OS_sem *rdv;//semaphore global avec 0 ressource = Rendez-Vous
```

```
task *t2, *t3; // pointeur global des taches
//L'application elle-meme :
void main_task(void) {
  int my_PID;
  my_PID  = OS_PRIO_SELF ;
    // Appel systeme : creation d'un Mutex
  s = OSMutexCreate(my_PID, NULL);//s est une var globale
  // Creation de la premiere tache applicative
  t2 = OSTaskCreate ( &f2,  9 );
  t3 = OSTaskCreate ( &f3,  8 );
    // Le temps de calcul (fictif) pour d'autres initialisations
  OS_WAIT(my_PID, sc_time(50,SC_US)); // BB sans code fonctionnel
  OSTaskDel(my_PID);// informe l'OS de sa fin
}//main_task()

void f2(void) {
  int my_PID;
  my_PID  = OS_PRIO_SELF ;
  // temps de calcul fictif
  OS_WAIT(my_PID, sc_time(80, SC_US));
  while (1) { // infinite periodic loop
    OSMutexPend(s, 0, my_PID);
    // Le temps de calcul de travail de la tache
    OS_WAIT(my_PID, sc_time (120, SC_US));
    // On relache le semaphore qui va relancer la tache f3
    OSMutexPost(s);
    OS_WAIT(my_PID, sc_time (80, SC_US));
    OSTimeDlyHMSM(0,0,0,1); // 1 ms
  }// End loop forever
  OSTaskDel(my_PID);
}//f2

void f3(void) {
  int my_PID;
  my_PID  = OS_PRIO_SELF ;
  OS_WAIT(my_PID, sc_time (80, SC_US));
  while(1) {
    OSMutexPend(rdv, 0, my_PID);
    OS_WAIT(my_PID, sc_time (80, SC_US));
    OSMutexPend(s, 0, my_PID);
    // Le temps de calcul
    OS_WAIT(my_PID, sc_time (150, SC_US));
    // On relache le semaphore
    OSMutexPost(s);
  }
  OSTaskDel(my_PID);
} // f3
```

On est ainsi capable de simuler d'une première manière naïve des tâches logicielles en SystemC et d'appeler des appels système simulés.

Néanmoins, pour obtenir une simulation temporelle, il est tout de même nécessaire d'ajouter des directives `wait()` aux blocs de code entre chaque appel système. Le temps d'exécution simulé dépend de l'architecture cible visée, mais on peut se contenter d'approximations à notre niveau de simulation SAT.

Par ce procédé, des tâches logicielles peuvent être intégrées et simulées de manière temporelle et fonctionnelle dans une simulation SystemC. Néanmoins ces tâches logicielles sont exécutées en parallèle, ce qui n'est pas conforme à une simulation réaliste où les tâches se partagent un nœud d'exécution de type processeur, où le fil d'exécution est unique. Il reste donc à traiter le problème de non concurrence d'exécution des processus de chaque

tâche sous SystemC qui n'est pas nativement supporté par le langage.

3.3.2 Gestion du temps garantissant un fil d'exécution unique

SystemC permet de modéliser une infinité de fils d'exécution s'exécutant de manière concurrente comme c'est le cas des nombreux blocs d'un système électronique. En revanche, si l'on veut modéliser des processus logiciels, il nous faut être capable de contraindre cette concurrence infinie à un seul fil d'exécution par processeur pendant une même unité de temps et contrôler laquelle des tâches est exécutée à chaque instant.

3.3.2.1 Modélisation des tâches et du temps avec les Blocs de Base (BB)

Afin de pouvoir faire apparaître les parties de calcul des tâches, nous considérons le modèle suivant : les tâches sont les unités de concurrence élémentaire qui sont composées d'une séquence de blocs de calculs et d'appels système.

Nous appellerons Blocs de Base ou Basic Blocs (BB) les suites de calcul entre deux appels système (le début et la fin d'une tâche étant considérés comme des appels système).

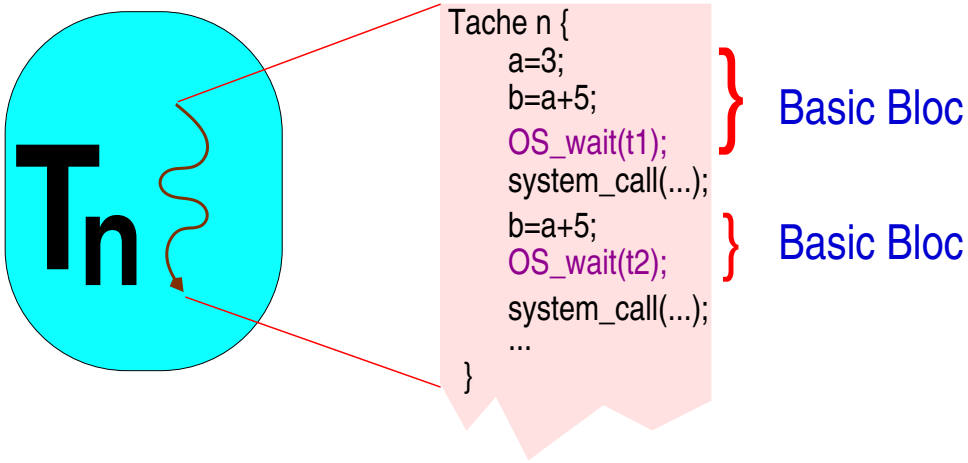


Figure 3.6: *Tâches découpées en succession de blocs de base et appels système*

Avec SystemC, ce code fonctionnel de calcul (des BBs) s'effectue en un temps nul du point de vue de la simulation du temps. Il faut donc être capable d'associer à chaque bloc de base un temps d'exécution afin de renseigner sa consommation de temps de calcul processeur (ou par la suite de temps d'exécution dans une zone reconfigurable si c'est une tâche matérielle). Ainsi à chaque BB est associé un appel spécifique pour la modélisation temporelle de type nouveau appelé `OS_wait()`, comme nous le présentons dans la figure 3.6. Dans un premier temps, cet `OS_wait()` peut être considéré comme un simple avancement du temps comme le simple wait() de SystemC. Mais cet `OS_wait()` va surtout nous permettre de contrôler qu'il n'y a qu'un fil d'exécution à la fois et permettre la modélisation de la préemption.

3.3.2.2 Gestion de la non concurrence, maintient d'un fil d'exécution unique

Comme explicité plus haut, nous devons introduire des mécanismes permettant de contourner artificiellement l'exécution normale en parallèle du moteur d'exécution SystemC de manière à ce qu'une seule tâche puisse utiliser l'unité de traitement (processeur) à un instant donné. Ainsi conçu, notre système lance tous les fils d'exécution (les `SC_THREAD`s associés) de nos tâches en même temps. Or le comportement réel désiré est que le modèle de processeur, support de l'exécution des tâches, soit capable de garantir l'exécution d'un seul et unique fil à la fois.

Afin de maintenir cet unique fil d'exécution, nous allons associer aux `SC_THREAD` relatifs aux tâches un évènement (`sc_event` appelé `event_run`) ajouté à sa liste de sensibilité et déclaré comme non initialisé au démarrage. Nous ajoutons cet évènement dans le bloc de description de tâche interne à notre OS (TCB, Task Control Block). Pour chaque tâche, cet élément de simulation permet de mettre en attente indéfiniment une tâche jusqu'à ce que l'OS la réveille en notifiant cet évènement de réveil au moment souhaité. Ainsi, chaque tâche est le corps d'un processus (un `SC_THREAD`) du module OS ayant son évènement de déclenchement dans sa liste de sensibilité.

Ainsi, l'ordonnanceur pourra individuellement notifier l'évènement `event_run` d'une seule tâche au moment désiré pour démarrer cette tâche. Il sera par la suite possible d'inclure un `wait(event_run)` dans le code des appels système désirant bloquer la tâche appelante et ainsi la bloquer jusqu'à ce que l'OS le désire. Par ce mécanisme, on est ainsi capable de contrôler la non concurrence initiale du système de tâches en faisant en sorte que l'ordonnanceur ne notifie qu'un seul évènement à la fois, celui de la tâche élue. Ce procédé de modélisation évite à haut niveau (SAT) de modéliser les changements de contexte et donc le matériel. Ce principe est représentatif de la philosophie de modélisation SAT. Il reste encore à modéliser la préemption.

3.3.2.3 Modélisation des interruptions et préemption

La préemption est la capacité d'un OS (si supporté par le processeur) à dérouter le flux d'exécution normal des instructions d'une tâches et de lancer un traitement spécifique momentané. Généralement, cela arrive lorsqu'une interruption est déclenchée. Dans ce cas, la routine de traitement de l'interruption correspondante est exécutée puis rend la main au code interrompu. Cela nécessite que l'on prenne garde, dans le cas réel et non modélisé, de bien sauvegarder le contexte d'exécution de la tâche en cours (pointeur de code, empilage de registres, ...) et de le restaurer par la suite.

Pour modéliser cela, nous encapsulons les appels `wait()` dans une méthode de notre OS nommée `OS_wait()` qui va permettre de gérer le comportement préemptif souhaité.

Algorithme 1: Algorithme de l' OS_wait(duration)

```
time_remaining = duration ;
repeat
    finished = 1;
    t0 = sc_time_stamp();
    wait(time_remaining, reschedule_ou_interrupt_event);
    if Not timed_out then
        t1 = sc_time_stamp();
        e_delay = t1 - t0;
        time_remaining -= e_delay;
        finished = 0;
        wait(task.run_event);
        // waiting for task elected again
until Not finished;
```

De manière à pouvoir modéliser la préemption, l'OS est sensible à un tic périodique (une interruption modélisée comme un évènement), qui relance l'ordonnanceur lors de sa notification. Cela implique aussi d'interrompre la tâche courante. Pour cela, notre mécanisme `OS_wait` de surcharge de l'appel classique `wait()` décrit dans l'algorithme 1 est en fait une boucle d'attente sur le temps à consommer souhaité ou sur l'évènement d'interruption de l'OS. Si on émet cet évènement, on est capable de détecter que l'attente à été interrompue avant son terme et dans ce cas on place indéfiniment la tâche en attente

de son évènement de réveil propre. On prend soin de noter le nouveau temps restant qui est le temps initial réduit du temps déjà écoulé avant l'interruption. Ainsi, lorsque l'ordonnanceur élit à nouveau cette tâche, il peut alors la réveiller en lui envoyant à nouveau son évènement de déclenchement et cette dernière reprendra son exécution pour le temps restant, comme montré dans la figure 3.7.

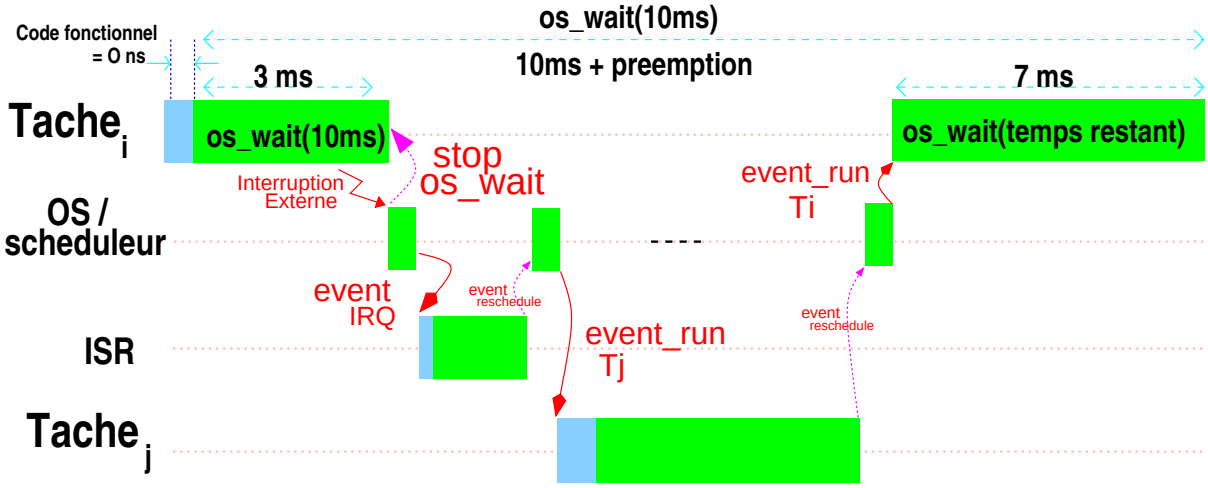


FIGURE 3.7: *Diagramme de Gantt de la préemption de tâches*

Grâce à cela l'ordonnanceur est capable d'interrompre la tâche en cours, et il peut la relancer ultérieurement pour une durée déduite de son exécution déjà effectuée avant l'interruption.

Une limitation à ce procédé est que l'on place les `OS_wait()` après les blocs de code (BB), qui sont de par le fonctionnement de SystemC déjà exécutés entièrement en un temps nul, avant l'`OS_wait` et donc en l'occurrence d'une éventuelle préemption.
Si nous ne souhaitons faire qu'une validation fonctionnelle, alors cela n'a pas trop de conséquences sur les résultats de la simulation : le comportement global peut ainsi être vérifié et les deadlocks potentiels identifiés.

Sinon une solution serait d'exécuter le code fonctionnel d'un BB après l'`OS_wait`, mais dans ce cas, le code fonctionnel n'est exécuté qu'à la fin du temps d'exécution intégralement écoulé.

Pour obtenir une simulation plus précise, la meilleure solution consiste à diminuer la granularité temporelle des blocs des tâches, jusqu'à placer un `OS_wait()` après chaque instruction (il faudrait même faire cela au niveau du langage assembleur). Ainsi la granularité de l'exécution serait beaucoup plus fine, mais au prix d'un gros travail d'annotation manuel, ce qui n'est pas vraiment envisageable, ni très performant en terme de rapidité d'exécution du simulateur.

Une solution plus fine sans besoin de modifier le code consiste alors à utiliser un simulateur fin de processeur, et donc de descendre d'un niveau d'abstraction, à savoir utiliser un Instruction Set Simulator (ISS), modifié pour continuer de jouer les appels système au niveau SAT de notre modèle. Cette possibilité de raffinement sera plus largement décrite en section 5.2.4 où l'utilisation d'un ISS permettra de d'annoter et insérer automatiquement les codeOS_wait() des BB lors des appels systèmes.

3.3.3 Gestion des fils d'exécution

Une fois la capacité de maîtriser l'exécution non parallèle de tâches logicielles dans notre modèle, il reste encore à rendre le modèle d'OS complètement fonctionnel.

3.3.3.1 Création dynamique de tâches

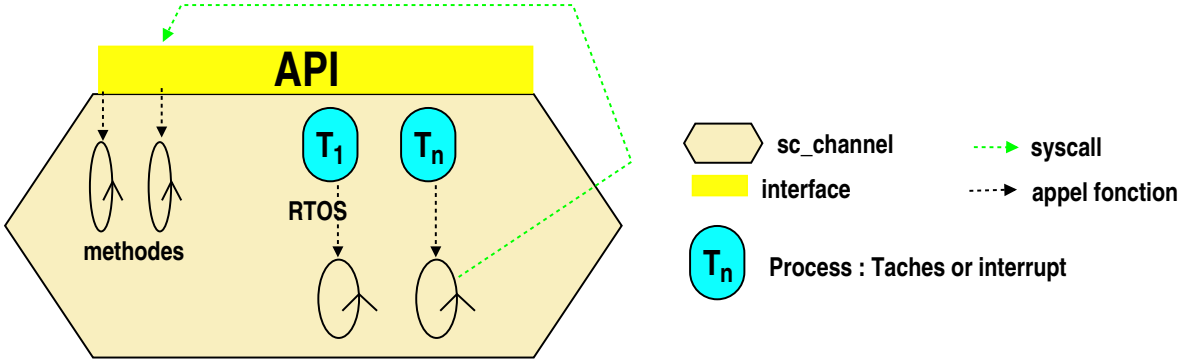


Figure 3.8: *modèle d'OS capable de créer dynamiquement des tâches*

La principale caractéristique d'un système logiciel est sa capacité à évoluer dynamiquement en fonction du besoin et du contexte. Cela passe par la possibilité de pouvoir lancer de nouvelles tâches à n'importe quel moment et d'être capable de les interrompre au profit d'autres tâches plus prioritaires (cette propriété est également souhaitable pour la modélisation des systèmes reconfigurables).

Depuis SystemC 2.0, nous avons la possibilité de créer dynamiquement des threads SystemC après le début de la simulation grâce à la commande `sc_spawn()` qui prend en entrée une fonction C++ avec éventuellement des paramètres (par exemple de type argc/argv comme un Main classique). Cette fonction crée en fait un `SC_THREAD` SystemC qui sera un processus de plus appartenant à l'objet SystemC contenant le processus qui a fait l'appel (cf. Figure 3.8).

Comme notre modèle d'OS doit être capable de créer dynamiquement des tâches, on utilise donc cette fonction `sc_spawn()` pour permettre d'exécuter le code d'une tâche. Nous initialisons le nouveau processus de manière à ce qu'il ne soit pas démarré et qu'il soit sensible à un évènement propre à chaque tâche permettant de lui notifier qu'il peut s'exécuter (jusqu'à ce que le système reprenne la main soit par un appel système, soit par une interruption). Lors de la création nous créons donc l'évènement qui est associé à la nouvelle tâche et le stockons dans la table des processus (TCB pour *Task Control Bloc*) de l'OS.

3.4 Modularité et interactions entre services

Maintenant que nous sommes capable de simuler le comportement d'un OS multi-tâches préemptible et interruptible en SystemC, il nous reste à concevoir un modèle permettant d'explorer facilement les services d'un OS pour les RSoC. On doit permettre d'explorer des OS comportant plus ou moins de services, et plus particulièrement les nouveaux services qui doivent être conçus et validés, comme ceux de gestion de tâches matérielles. Ces système d'exploitations sont des réalisations *ad-hoc* pour lesquels il est nécessaire de tester des services aux comportements différents et potentiellement distants.

3.4.1 Découpage en services

De manier à construire un RTOS modulaire, nous avons regroupé les principaux services en sous catégories. La localité des données internes partagées a été le principal critère de partitionnement. Les catégories actuelles de services permettant de faire un OS minimal, selon les différents standards d'OS et ceux observés, sont les suivantes :

1. Gestion des Processus (tâches) et partage du temps processeur : création, délétion, suspension, ...
2. Ordonnancement : politique d'ordonnancement, le cœur des RTOS
3. Communication synchrones et asynchrones : par port, mémoire partagée ou FIFO
4. Synchronisation : sémaphore/mutex, variables conditionnelles et synchronisation entre processus, drapeaux
5. Gestion mémoire : allouer, libérer, transférer
6. Gestion de stockage : lecture / écriture de fichiers
7. Gestion des interruptions
8. Gestion du temps : timer, timeout
9. Gestion de la distribution de l'OS
10. Service de la plateforme de simulation : simulation explicite du partage du temps avec des appels à la directive `OS_wait()`

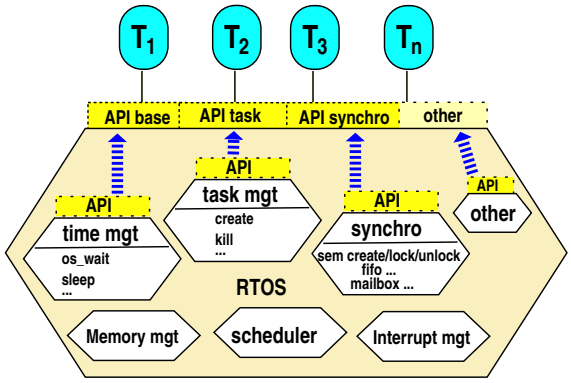


FIGURE 3.9: *Modèle d'OS modulaire : l'OS est modélisé par un canal hiérarchique* `sc_channel` *contenant plusieurs sous-canaux. Chaque sous-canal représente et modélise un groupe de services. Le canal principal regroupe l'ensemble des services fournis à l'application sous forme d'une unique API.*

Le dernier groupe est ajouté pour les besoins de la simulation temporelle mais ne constitue pas en soi un des services du RTOS. D'autres services peuvent être rajoutés ou se greffer au dessus, comme ce sera le cas pour la *Machine Virtuelle* de Ter@ops (cf. section résultats 5.3).

De manière à offrir ces services à l'application tout en maintenant l'application indépendante de l'implémentation de l'OS, nous choisissons d'implémenter chaque service sous forme d'un canal SystemC (cf. FIGURE 3.9). Chaque canal implémente son interface (hérite d'une `sc_interface`) spécifique, et l'OS exporte (en utilisant le mécanisme de `sc_export`) ses interfaces aux applications de manière à ce que celles-ci ne voient qu'une seule et unique API. La séparation entre les interfaces et leur implémentation est importante car elle permet de faciliter l'exploration des différentes manières d'instancier un service en remplaçant un module par un autre de même interface.

Voici une description de la manière de décrire une interface en SystemC :

Listing 3.3: *exemple de l'interface du service gestionnaire des tâches*

```
1 class basic_gestiontache_if : virtual public sc_interface {
  public:
    virtual task* create_task(void (*f)(), int priorite) = 0;
4   virtual void  kill_task(unsigned int pid) = 0;
    virtual int   get_pid(task *t) = 0;
```

```
  virtual task* create_ISR(void (*f)(), int priorite) = 0;
};
```

3.4.1.1 Service core ou de simulation

Ce service est nécessaire à la simulation, mais n'est pas un service d'OS au sens traditionnel. Il n'est là qu'à des fins de simulation. Il consiste simplement à indiquer que l'on consomme du temps de calcul du processeur, à travers le pseudo appel système OS_wait().

3.4.1.2 Service d'ordonnancement

Le service d'ordonnancement (ou scheduler) est le principal service d'un OS car il est là pour permettre de répartir la ressource processeur partagée entre les différentes tâches. C'est généralement l'élément central de gestion demandé à un RTOS. L'algorithme consiste à chercher la nouvelle tâche élue selon les critères de choix (algorithme Round-Robin, Earliest Deadline First, etc.), et de (re)charger le le bon code correspondant, en le plaçant sur la prochaine instruction à exécuter. Dans notre cas de simulation, l'ordonnanceur contient la liste des tâches et une fois que la nouvelle tâche est élue (selon l'algorithme choisi/à explorer), il suffit de lui envoyer son évènement `task.run_event` pour la libérer et donc la laissser poursuivre son exécution. La seule difficulté est de garantir qu'une seule tâche ne s'exécute à la fois, donc que toutes soit arrétées au moment de relacher l'élue. Par construction, toutes les tâches sont crées dans l'état bloqué Ensuite au cours de l'exécution, le gestionnaire d'interruption s'appuie sur l'OS_wait() pour garantir l'arrêt impromptu, et les appels système bloquants gèrent eux même l'arrêt selon le besoin, ainsi que le relancement de l'ordonnanceur.

Le service d'ordonnancement contient une particularité spécifique au besoin de modélisation en SystemC. En effet, ce service ne peut s'exécuter dans le contexte d'une tâche, et donc aurait besoin d'être modélisé comme une tâche ou un traitant d'interruption indépendant. Mais comme son rôle est d'élire la nouvelle tâche, on ne peut lui demander de s'élire lui-même auparavant. La solution est donc de créer un `sc_thread` unique qui exécutera l'algorithme d'ordonnancement, à chaque fois que son évènement propre `reschedule` sera lancé par la fonction `reschedule()` accéssible par les autres modules de service nécessitant de relancer l'ordonnanceur.

3.4.1.3 Service de gestion des tâches

Le service de gestion des tâches sert à permettre le suivi des tâches (états courants, deadline, priorité) et de maintenir le système cohérent dans une base (TCB). Par ailleurs c'est aussi lui qui permet d'ajouter dynamiquement de nouvelles tâches dans le système. Il est généralement indissociable du service d'ordonnancement, mais de manière à pouvoir explorer différents algorithmes d'ordonnancement, on le sépare de ce dernier. Du point de vue de la simulation en SystemC, ce service gère la création dynamique des threads SystemC, comme on peut le voir dans le listing 3.4. Cela comprend la gestion des évènements associés à chacunes des tâches pour le contrôle de leur exécution convenable.

Listing 3.4: *Création dynamique de SC_ THREAD avec paramètres pour simuler les tâches*

```
/** @brief Appel systeme de creation de tache avec parametres
 * @param void(*f)() un pointeur sur une fonction(void*)
 * @param void* un pointeur sur les data (*struct tache_parametres)
 * @param int : la priorite de la tache
 * @return task* : un pointeur sur la TCB de la tache creee*/
```

```
task* basic_gestiontache::create_task
(void (*ff)(void*),void* pointeurdata, int priorite) {
  task *t;///<  la nouvelle tache
  char taskname[10] ; //= malloc(15*siszeof(char)) ;
  int real_PID;
  real_PID = ((osID-1)*NB_MAX_PROCESS_PER_OS) + ++pid_cpt;
  sprintf(taskname,TACHE_BASE_NAME,real_PID);
  // Creation de la tache, on attribue un nom a la tache
  t = new task(priorite,taskname, CREATED , real_PID , osID);
  t->codewithparam = ff;
  /// creation dynamique du process,
  sc_spawn_options o;
  o.dont_initialize();
  o.set_sensitivity(&(t->synchro));
  // creation de process selon que la tache a ou non des parametres
  //funcparam * lesparams=(funcparam*)pointeurdata;
  sc_process_handle h;
  h= sc_spawn(sc_bind(ff,pointeurdata),t->name,&o);

  task_list.push_back(t);// Ajout de la tache dans la liste
  M_os_execute(M_get_currenttask(), sc_time(TEMPS_CREATE_TASK) );
  M_reschedule();
  /// L'appel a cette primitive bloque la tache appelante
  wait(M_get_currenttask()->synchro); // waiting for task.run_event
  return(t);
}//END create_task avec param
```

3.4.1.4 Service IRQ

Comme il est intéressant dans un système embarqué de pouvoir prendre en compte l'environnement généralement sous la forme d'interruptions, on a besoin d'être capable de les gérer. Le gestionnaire d'interruption est généralement un petit bloc matériel capable de mémoriser plusieurs lignes d'interruptions et d'interrompre le processeur jusqu'à sa désactivation par l'OS. On se doit de modéliser ce bloc matériel associé à tout PE, ainsi que son interface coté logicielle au niveau de l'OS permettant de valider/dé-valider et traiter ou installer des routines d'interruptions adéquates. Ces dernières seront modélisées sous forme de threads d'IRQ comme les tâches, mais de plus forte priorité, initialisée en état *attente* (d'interruptions). Le modèle du simulateur devra aussi comprendre un module SystemC externe qui se chargera de simuler les évènements issus de l'environnement comme dans un testbench VHDL classique.

3.4.1.5 Service de gestion du temps

Le timer est un module lui aussi matériellement distinct du processeur (bien que souvent intégré dans la même puce) permettant de programmer une/des interruptions à une date ultérieure ou de manière cyclique. Cela sert basiquement à l'OS pour déclencher régulièrement l'ordonnanceur capable de simuler le temps partagé, ainsi que pour gérer des timeouts et délais pour chaque tâche. Il est généralement indispensable dans les systèmes temps–réels.

3.4.1.6 Service de synchronisation

Le service de synchronisation permet à différentes tâches de coopérer/se synchroniser ou de se donner des rendez-vous. Il sert aussi à l'OS ou à d'autres modules pour des problèmes de communication et de gestion de périphérique ou de ressources paratagées. Dans un système multi-nœuds , ce service est indispensable afin de pouvoir communiquer d'un nœud à l'autre et on le retrouve sous diverses déclinaisons (semaphore, spinlock, file de messages etc.).

3.4.2 Gestion de concurrence de certains services

Lorsque l'on cherche à modéliser à haut niveau un nœud d'exécution avec son OS, on se rend compte que certains services fournis par ce nœud d'exécution s'exécutent en réalité en parallèle. En l'occurrence, la gestion des IRQ est effectuée par un bloc matériel dédié, extérieur au CPU [1] connecté à ce dernier. Autrement dit, à haut niveau, le service de gestion des IRQ a un flot d'exécution concurrent à l'exécution du code logiciel (OS et application), hormis la couche HAL logicielle qui sert d'interface à ce service. Il en va de même pour un timer ou d'autres "périphériques externes" intégrés dans les CPU modernes. Cela implique que les services qui ont leur propre exécution indépendante du CPU vont posséder en leur sein un (ou plusieurs) thread(s) SystemC caractérisant ce comportement concurrent de la partie logicielle ou matérielle.

3.4.3 Gestion de multiples flux, modélisation du multithreading et de tâches matérielles

Un modèle d'OS sur un nœud d'exécution peut représenter un processeur évolué capable d'exécuter physiquement plusieurs threads en parallèle. Dans ce cas, nous sommes en mesure de simuler facilement plusieurs threads concurrents. En revanche, pour être capable de les interrompre indépendamment les uns des autres, il nous faut associer à chaque tâche un évènement propre pour son interruption et modifier le code de l'`OS_wait` afin qu'il attende cet évènement d'interruption associé à la tâche en cours, et garantir ainsi l'arrêt d'un seul des threads. Néanmoins, la modélisation du comportement multithread risque de ne pas être conforme (trop simpliste) au fonctionnement interne des pipelines des processeurs multithread.

3.4.4 Interactions entre services

La principale conséquence de la décomposition en blocs des fonctionnalités de l'OS est qu'il faut que chaque module puisse inter-opérer avec les autres modules, car bien que séparés, ils ne sont pas complètement indépendants. Par exemple, le gestionnaire de sémaphores a besoin de communiquer avec les modules d'ordonnancement et de gestion des tâches pour informer qu'une tâche se retrouve bloquée ou libérée suite à un accès à un sémaphore et relancer l'ordonnanceur. Pour cela nous avons défini une interface distincte de l'API de programmation pour la communication interne entre modules (la première interface externe étant destinée aux appels de services systèmes depuis l'application). Elle se présente sous la même forme que l'interface de base comme l'indique le listing 3.5.

Listing 3.5: *Exemple de l'interface interne du gestionnaire de tâche*

```
class basic_gestiontache_iif : virtual public sc_interface
{ public:
```

1. Core Processing Unit, l'unité de calcul du processeur

```
/// modifier l'etat des taches
 virtual void change_task_state(int pid_task,task_state newetat)=0;
/// Donne la liste des taches (au scheduleur par exemple)
 virtual std::list<task*> get_task_list(void) = 0;
/// retourne le pointeur sur la tache de PID, pour le OS_WAIT
 virtual task* get_ptask(int pid) = 0;
};
```

Chaque module doit donc posséder des ports pour accéder aux services des autres modules, la communication se faisant par des liaisons point-à-point (cf. Figure 3.10).

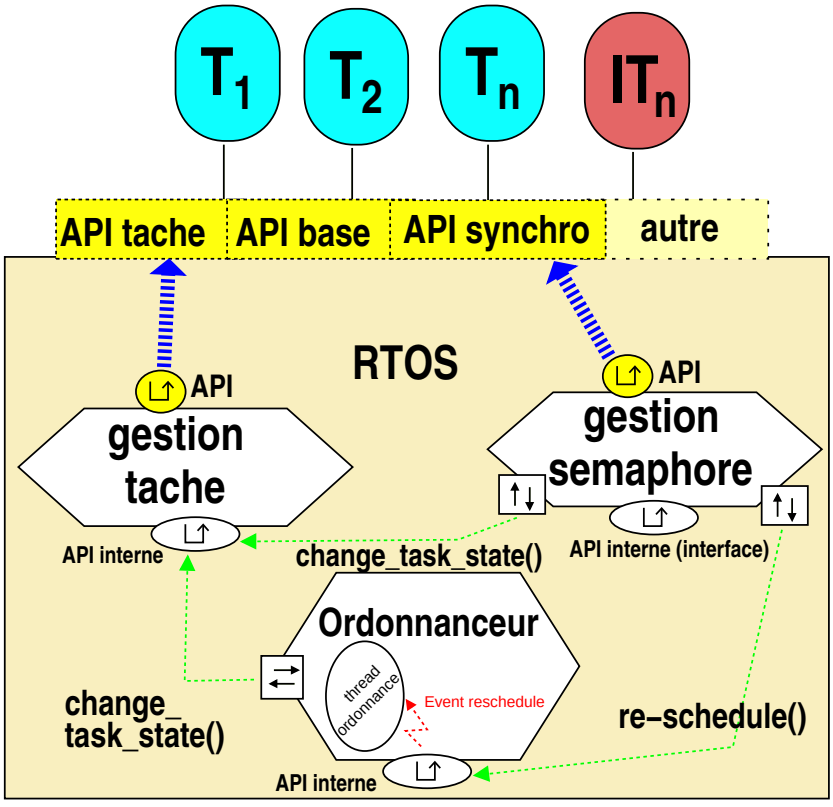


Figure 3.10: *Communication entre modules via leur interface interne*

Les interactions ainsi modélisées permettent de faire communiquer n'importe quel service avec un autre dont il dépend. Néanmoins, cela implique que les services possèdent autant de ports que de services distincts requis et de devoir lier les bons ports aux bonnes interfaces dans la constitution d'une instance du modèle d'OS, ce qui peut s'avérer fastidieux et source d'erreur.

Aussi afin de rendre les communications entre modules aussi génériques que possible, et afin de pouvoir remplacer un service par un autre équivalent, nous allons proposer un protocole permettant de rendre ces communications génériques dans le chapitre suivant.

Nous pouvons néanmoins déjà tester les capacités du modèle, sans se préocciper des facilités d'interconnexion des modules et valider le fonctionnement corect de notre modèle d'OS en SystemC.

3.5 Instrumentation et exploration simple du modèle pour son évaluation

3.5.1 Instrumentation du modèle

Pour que la simulation permette d'évaluer le déroulement de l'exécution du système, il est important de pouvoir remonter des informations utiles pour évaluer si les choix architecturaux ou de gestion du système sont valides ou non. Cela permet de dérouler le flot de conception jusqu'à ce que le système remplisse les contraintes. Pour cela nous avons instrumenté le modèle pour qu'il nous remonte des informations pertinentes lors de la simulation. Nous avons fait en sorte de pouvoir visualiser sur un diagramme (semblable à un diagramme de Gantt, cf. figure 3.12) les informations utiles comme le numéro de la tâche en cours d'exécution, ainsi que l'apparition d'interruptions, le nombre de changements de contextes ou d'appels système. On peut aussi remonter la mesure du temps inoccupé du CPU (*idle* time). D'autres mesures peuvent être envisagées et être ajoutées facilement, sans que cela n'influence l'exécution du modèle, si ce n'est le ralentir un peu. Nous verrons plus tard que l'application elle même peut être instrumentée de manière non intrusive pour ajouter des informations de debug, sans influence sur le temps d'exécution simulé.

Une liste des métriques intéressantes à été dressée dans le cadre du projet OveRSoC, mais nous n'avons pu toutes les mettre en œuvre, faute de temps.

3.5.2 Simulation comportementale

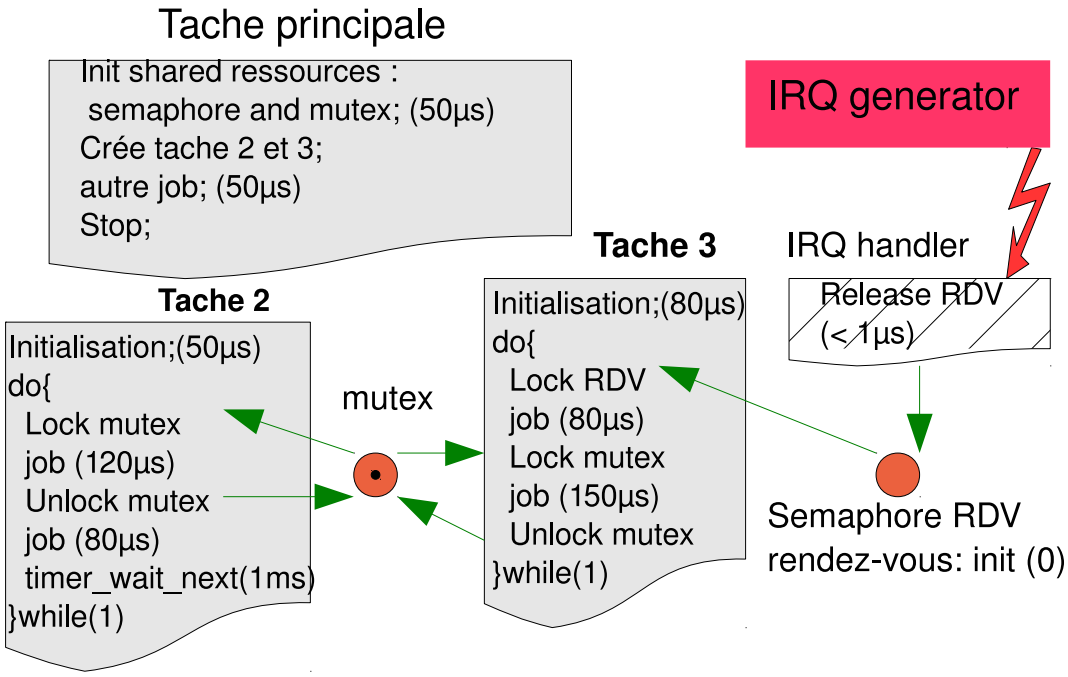


Figure 3.11: *Application simple de test fonctionnel : une tâche principale, une cyclique, et une qui traite les eventuelles interruptions, en partageant des données avec la tâche cyclique*

Nous avons implémenté un modèle de RTOS avec une signature des appels système et un comportement semblable au RTOS open source μC/OS-II[muc02]. Dans μC/OS-II, l'ordonnanceur est basé uniquement sur la priorité unique des tâches définies par l'utilisateur (pas de politique *round-robin*). Le réglage du *tick* OS qui réveille périodiquement

l'ordonnanceur afin de simuler des quantums de temps (et vérifier l'interruptibilité du modèle) est réglé sur une période de 500 μs (Figure 3.12).

On a défini une petite application synthétique (cf. figure 3.11) pour démontrer le bon comportement du modèle simulé. Cette application est composée de trois tâches et une ISR (tâche de traitement d'interruption). La tâche principale T1 s'exécute avec la plus forte priorité afin de lancer les autres tâches, et initialise les ressources logiques (création d'un sémaphore d'exclusion mutuelle *mutex* et un de *rendez-vous*). La tâche T2 va s'exécuter continuellement de manière périodique (toutes les millisecondes) et la tâche T3 est un traitement sporadique déclenché lors d'interruptions externes. Le sémaphore mutex (initialisé à 0) protège une ressource partagée (données d'image par exemple) entre les tâches T2 et T3. Le sémaphore de rendez-vous est libéré lorsque qu'une interruption apparaît et que l'ISR a fini de la traiter, ce qui fournit un moyen de déclencher la tâche T3. Les Basic Blocs des tâches sont marqués avec des temps de traitements fictifs à titre de premier exemple.

La figure 3.12 illustre les activités du RTOS simulé, sous la forme d'un diagramme de Gantt autogénéré. Ce diagramme montre quatre mécanismes fondamentaux : la préemption de tâche, le déclenchement périodique de l'ordonnanceur via un tick, le fonctionnement d'interruptions et la synchronisation de tâches.

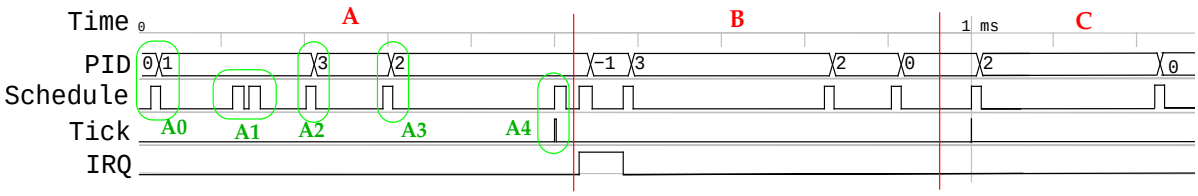


Figure 3.12: *Trace d'exécution montrant la tâche courante et l'activité de l'ordonnanceur, du TICK et des interruptions*

Le PID [2] 0 représente la tâche de fond de l'OS qui fait l'initialisation de l'OS puis de la tâche principale, et reprend la main quand le système ne fait rien (c-à-d. est *idle*). On peut voir en zone A le fonctionnement lors de l'initialisation. En A0, l'OS relance le scheduleur une fois son travail (tâche T0) d'initialisation fait, ce qui mène à l'élection de la tâche principale T1 seule instanciée à ce moment. Cette dernière procède aux initialisations de l'utilisateur, et crée les deux tâches T2 et T3, ce qui redéclenche consécutivement l'ordonnanceur (en A1), mais la tâche T1 conserve le processeur car plus prioritaire. Une fois la tâche principale finie définitivement, l'ordonnanceur se redéclenche et élit la tâche T3 (en A2) car plus prioritaire que la tâche T2. T3 se bloque en attendant le sémaphore de rendez-vous en A3 se qui laisse la possibilité à T2 de s'exécuter.

En A4, le tick OS relance automatiquement l'ordonnanceur, ce qui n'a pas de conséquence, car à ce moment seule T2 est dans l'état *prête*. Ce tick permettra par la suite de modifier la politique d'ordonnancement et implémenter des politiques plus complexes de partage du temps de calcul.

La partie B représente le comportement relatif à l'apparition d'une interruption. Une interruption est déclenchée, ce qui préempte la tâche T2 et lance l'ordonnanceur qui déclenche le traitant d'interruption (PID -1 de la tâche ISR). L'ISR acquitte l'interruption et relance l'ordonnanceur une fois son traitement rapide fini. Dans son traitement, l'ISR a libéré le sémaphore de rendez-vous qui donc place la tâche T3 dans l'état *prête* et relance l'ordonnanceur. L'ordonnanceur choisit cette fois-ci la tâche la plus prioritaire parmi celles qui sont dans la liste des tâches prêtes. Une fois le traitement de T3 parvenu à son terme,

2. Processus IDentifier

T3 attend à nouveau le sémaphore rendez-vous jusqu'à la prochaine interruption. Ainsi T2 reprend son travail, et attend sa prochaine période (toutes les 1 ms) comme on peut le voir en zone C.

3.5.3 Exploration comportementale d'un service de synchronisation

Grâce à la nature modulaire du modèle d'OS il est facile de remplacer un service par un autre de même signature d'appel système, mais implémenté avec une algorithmique différente. La figure 3.13 montre les résultats d'exécution de deux implémentations différentes du service de mutex : lorsque le relâchement de ce sémaphore libère une tâche, on peut attendre le prochain relancement naturel de l'ordonnanceur lors de la fin d'une tâche (figure 3.13.A) ou décider de relancer immédiatement l'ordonnanceur (figure 3.13.B), ce qui permet ici de lancer la tâche 3 plus prioritaire. Ce simple changement peut donner lieu à des schémas d'exécution complètement différents pour une même application testée dans les mêmes conditions (interruption au même moment).

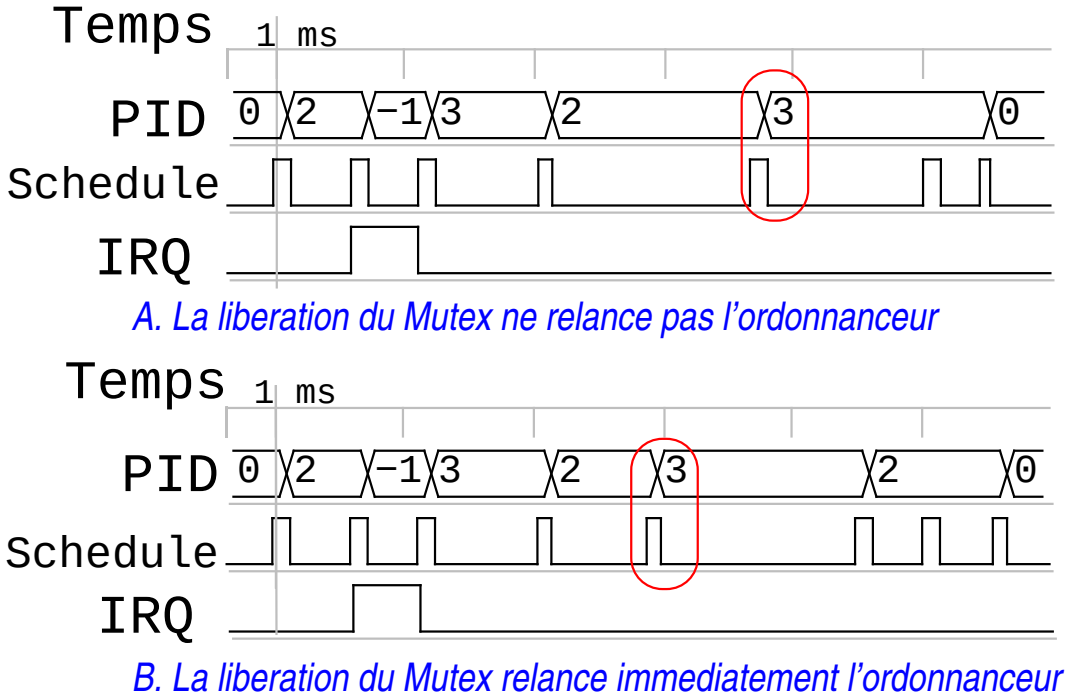


Figure 3.13: *Exploration de l'implémentation du service de sémaphore : choix de relancer immédiatement l'ordonnanceur ou pas.*

La précédente application a été reprise identiquement et on a observé le comportement lors du déclenchement d'une interruption. On peut voir dans les deux cas que la tâche T3 lorsqu'elle est libérée et que le processeur lui est alloué, peut s'exécuter normalement jusqu'à la requête du sémaphore mutex. La tâche T2 l'ayant déjà pris, T3 est donc obligé d'attendre, ce qui relance T2. Lorsque T2 relâche le mutex (zone entourée de la figure 3.13), cela replace T3 dans l'état *prêt*. En revanche, dans le cas A, le service de sémaphore ne relance pas immédiatement l'ordonnanceur, et T2 s'exécute jusqu'à son terme, ce qui retarde d'autant plus le traitement de la tâche T3, alors que dans les cas B, la tâche T3 est immédiatement relancé dès que T2 relâche le sémaphore, ce qui peut être plus intéressant puisque T3 est le traitement secondaire d'une interruption.

On voit bien ici que l'exploration des services d'un OS facilite l'évaluation comportementale du système et permet de trouver l'algorithmique adéquate pour une application particulière.

La vision générique du modèle d'OS tel que décrit dans le diagramme de classes UML 3.14 est la suivante : un OS est un `sc_module` hiérarchique composé de un ou plusieurs bloc de services. Chacun de ces services est un `sc_channel` qui implémente deux interfaces. Une pour la partie de l'API de l'OS dont le bloc de sevice est responsable, une pour les services rendus aux autres blocs. Pour cela, un service faisant appel aux services d'autres blocs possède un port pour se connecter à ces derniers. L'OS exporte les interfaces externes des blocs de services de manière à présenter l'API de l'OS à l'application.

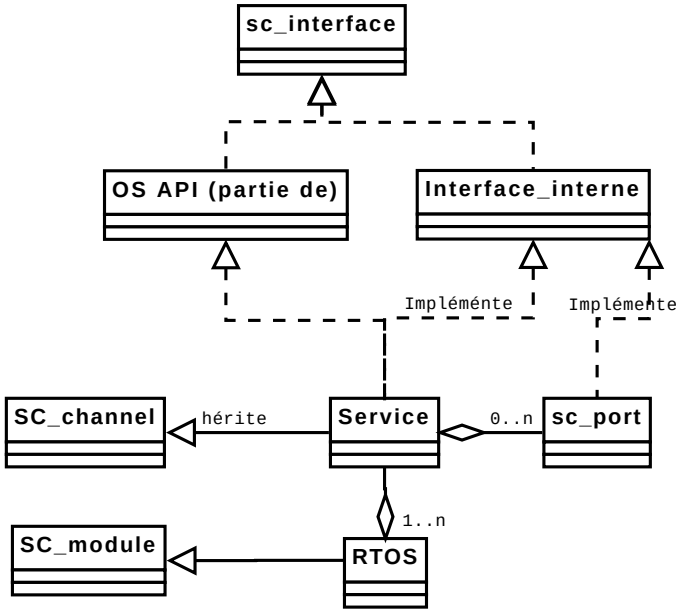


Figure 3.14: *Composition d'un modèle d'OS modulaire en SystemC*
Diagramme de classes UML de la composition d'un modèle d'OS en SystemC à base de blocs de services

3.6 Validation du modèle sur un cas concret de vision robotique

3.6.1 Une application robotique de perception et de navigation visuelle

Le traitement en temps-réel de scènes visuelles représente un problème majeur dans le domaine de la robotique autonome. De nombreux comportements sont issus de ces traitements : navigation, reconnaissance et manipulation d'objets, suivi de cible, voire interactions complexes avec les humains. Pour le moment, de telles applications de robotique demandent tellement de puissance de calcul qu'il est difficile de les embarquer dans le robot, et on les déporte sur des serveurs de calcul externes à défaut de mieux.

Or depuis quelques années, de nouvelles approches du traitement visuel sont apparues. Un dispositif visuel n'est plus considéré comme isolé, mais comme une partie d'une architecture intégrée à l'environnement d'exécution (on parle de vision active[Bal91]). Il tient compte de plus en plus des paramètres liés aux propriétés dynamiques du système dans lequel il s'insère, et utilise des algorithmes d'automates dynamiques qui scrutent les interactions du système avec son environnement dans une boucle d'ajustement continu.

L'application étudiée ici est un sous ensemble d'un système cognitif permettant à un robot équipé d'une caméra numérique CCD [3] de percevoir son environnement pour y

3. Charge-Coupled Device

naviguer. L'architecture globale de ce système de vision est inspirée de mécanismes de nature biologique et est basé sur les interactions entre le traitement du flot visuel et les mouvements du robot (architecture Per-Ac [GZ95]) qu'étudient les membres de l'équipe NeuroCyber d'ETIS. L'apprentissage des associations sensorimotrices permet au système de réguler sa dynamique [MGGH05] et ainsi naviguer, reconnaître des objets et créer un compas visuel [GJB+00].

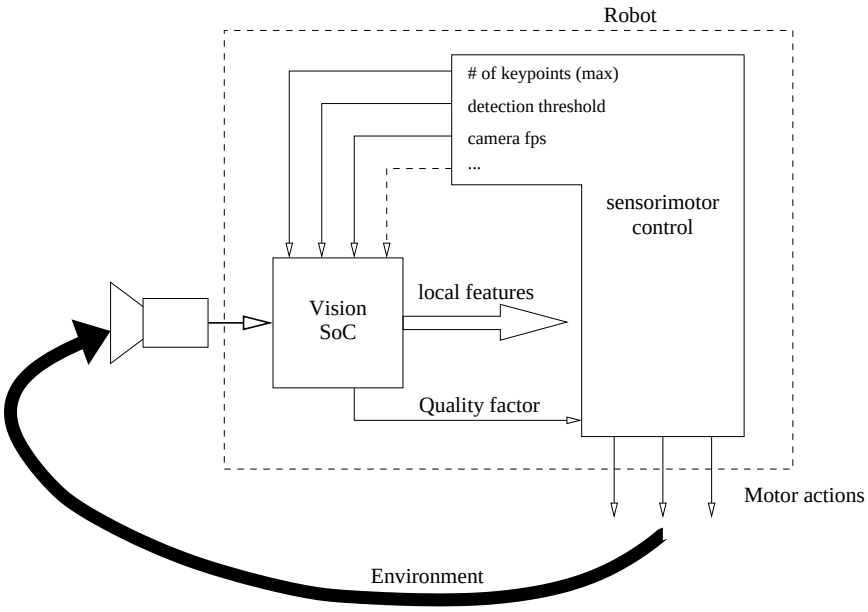


Figure 3.15: *Boucle sensorimotrice du robot dans laquelle s'inscrit l'application de vision*

Cette application nécessite d'être implémentée sur une puce dédiée afin de pouvoir l'embarquer sur le robot. Ce SoC doit être suffisamment flexible pour permettre de supporter différentes missions de navigation et s'adapter à l'évolution des contraintes dues à la dynamicité intrinsèque du système (voire fig. 3.15). Par ailleurs, l'architecture doit fournir un support de calcul intensif pour exécuter le traitement bas niveau des images. Une solution envisagée pour cela est d'implémenter l'application en partie en matériel et en logiciel. Les traitements intensifs réguliers peuvent être pipelinés dans des blocs matériels, la partie de contrôle irrégulière restant attachée à des processeurs. Ces choix amènent à concevoir un système hétérogène composé de traitements parallèles et distribués. Une telle application présentant des comportements dynamiques et adaptatifs requiert d'être gérée par un RTOS, ce qui nous incite à utiliser notre modèle d'OS pour prototyper le système et explorer son architecture.

3.6.1.1 Description de l'application de détection d'amer d'une vidéo en temps-réel

Nous allons décrire succinctement l'application dans cette partie. Elle est principalement constituée d'opérateurs classiques de filtrage, de sous-échantillonnage et d'extraction d'imagettes et pourrait être considéré comme un traitement de type flot de données. Néanmoins, cette application s'insère dans une boucle dynamique et adaptative de régulation. Un tel contexte interdit d'utiliser un flot de conception classique qui ne gère pas les aspects dynamiques.

L'application que nous utilisons est proche de celle issue de travaux de Gaussier *et al.* dans [LGC00] et intègre une approche multi-échelle pour extraire les primitives visuelles. Le système visuel fournit des caractéristiques locales de points d'intérêt (ou amers) détectés dans un flot continu d'images de caméra CCD en niveaux de gris 8 bits. Ces

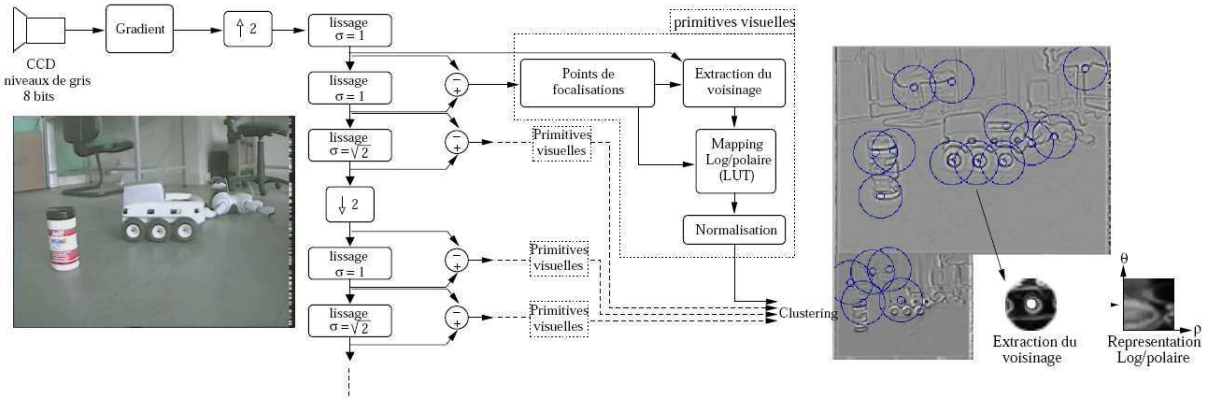


Figure 3.16: *Architecture globale de l'algorithme. Les caractéristiques locales sont extraites du voisinage des amers détectés sur chaque différence de gaussiennes.*

caractéristiques locales alimentent un réseau de neurones qui associe les actions motrices aux informations visuelles : ce réseau de neurones peut apprendre la direction des déplacements du robot en fonction de la reconnaissance de la scène.

Le principe de base de cette application est d'effectuer la détection des points d'intérêt sur une *pyramide multi-résolution*. Le point d'entrée de l'algorithme est une image de gradients (rehaussement des contours) calculés sur un flot vidéo en niveaux de gris dont l'objectif est d'être exécuté en temps-réel (à la cadence vidéo maximale possible). En effet, plus l'algorithme sera rapide, plus la vitesse de déplacement du robot pourra être élevée. Le système visuel que nous étudions peut être découpé en deux modules :

- un mécanisme multi-échelle (plusieurs niveaux de filtrage d'image) pour permettre l'extraction des points d'intérêt caractéristiques : les 3 niveaux de filtrage ne sont analysés que si les contraintes temporelles le permettent, et lors des déplacements rapides, seule l'image la plus grossière suffit pour détecter/éviter les obstacles. C'est la partie de calcul intensif qui de par sa régularité se prête bien à une parallélisation. Mais les modes de fonctionnement du robot (*rapide*, *intermédiaire* ou *lent*), rendent dynamique la demande de calcul de ce traitement.
- un mécanisme de détection et d'extraction des points d'intérêt. Les détecteurs utilisés permettent d'extraire le voisinage des points qui sont des coins ou aspérités de l'environnement visuel. Plus précisément, ces points d'intérêt correspondent aux maximums d'amplitude locaux de gradients d'images filtrées par différence de Gaussiennes (fig. 3.16). Ces points extraits correspondent à des zones importantes et de forte courbure dans l'image de la caméra. Un tri est effectué pour limiter la liste aux N premiers maximas ordonnés en fonction de leur intensité et pour extraire ensuite des imagettes du voisinage de ces amers sous forme de région circulaire avec un rayon de 20 pixels. Cette partie d'analyse et d'extraction est sujette à une forte variation du nombre de tâches à effectuer car dépendante de l'environnement visuel, chargé ou plat. Cette partie fortement dynamique nécessite donc un OS pour s'adapter aux besoins variables.

Une description plus complète de cette application peut être trouvé dans [VMM+08].

3.6.1.2 Description de l'application logicielle

Nous allons d'abord décrire la conception logicielle de l'application, utilisant des services d'OS standard dans un premier temps, en particulier la création dynamique de tâches.

Dans le but d'illustrer l'application, la figure 3.17 montre un sous ensemble de l'appli-

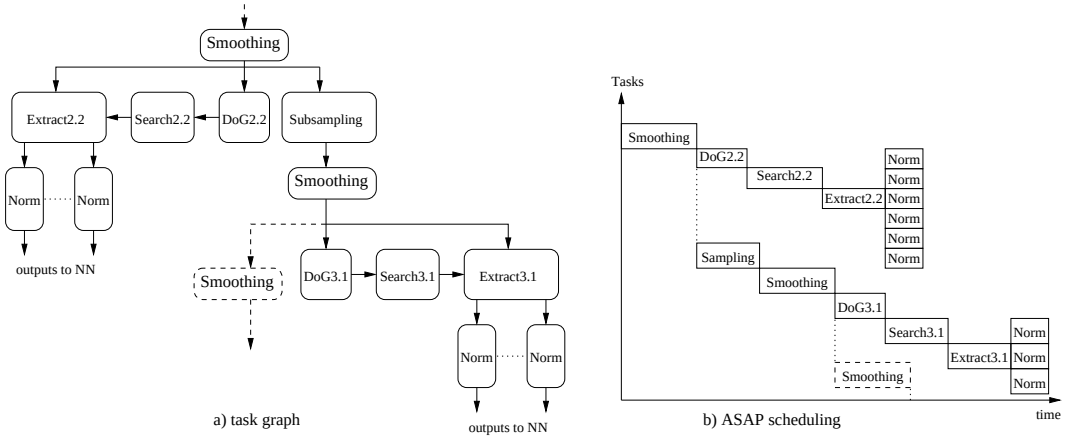


Figure 3.17: *Exemple de partitionnement de l'application en un graphe de tâches en a). Seulement deux demi-échelles d'un niveau sont illustrés ici. Dans l'hypothèse de ressources de calcul infinies et d'un ordonnancement ASAP (au plus tôt), cela produirait l'exécution du graphe b) où les deux demi-échelles comportent respectivement six et trois points d'intérêts.*

cation de vision, partitionnée en de nombreuses tâches. Les tâches `Sampling`, `Smoothing` et `DoG` correspondent aux opérations de sous-échantillonage (calcul de l'image des gradients), au filtrage de gaussiennes et à la différence de gaussiennes. La tâche `Search` recherche, trie et extrait les coordonnées des points d'intérêt au dessus du seuil de détection. Elle prend en entrée les résultats de `DoG`, avec comme paramètres N et γ. La tâche `Extract` construit les imagettes de voisinage des points d'intérêt et `Norm` transforme et nomalise les caractéristiques locales. Elle prend en entrée les coordonnées de points d'intérêt fournies par `Search` ainsi que l'image fournie par `Smoothing`. Le découpage réel est un peu plus complexe, mais cette figure permet d'illustrer la modélisation et la simulation de l'application avec un OS temps-réel. Le support d'un OS est nécessaire pour permettre d'obtenir le comportement dynamique souhaité, à savoir créer dynamiquement autant de tâches de traitements (`Extract, Norm`) que nécessaire. Cette charge de calcul ne peut être estimée ni lors du développement ni à la compilation, ce qui justifie l'utilisation d'un OS temps-réel.

3.6.2 Un RTOS dédié comme solution d'implémentation spécifique au domaine

Lorsque cette application est intégrée à la navigation des robots mobiles ou pour la reconnaissance d'objets, elle doit évidemment satisfaire des contraintes temps-réel. Par exemple, les caractéristiques de l'image locale extraites dans le voisinage de points d'intérêt peuvent être utilisées pour la perception des obstacles et, par conséquent, pour l'orientation de la trajectoire. Dans le cas général, ce sous-système visuel doit vérifier les contraintes temps-réel car il est utilisé dans une boucle sensori-motrice globale.

Toutefois, en raison de la complexité du comportement dynamique de notre application de vision, une caractérisation précise de son comportement temporel n'est pas triviale *a priori*. Exprimer un délai limite (deadline) ou une période globale correspondant à toutes les conditions du contexte (motivation du robot et état interne, nature des scènes visuelles, etc.) est impossible à la compilation et dépend principalement de la dynamique du système complet (voir fig. 3.15). Dans notre cas, utiliser la contrainte de la période (le nombre de trames par seconde - fps) est un paramètre, parmi d'autres signaux de régulation (nombre maximum de points d'intérêt extraits N, seuil de détection γ), donnés par le système de contrôle externe. Tous ces paramètres varient de façon dynamique durant le fonctionnement du système.

Nous avons effectué un profil temporel de l'application de vision et mesuré le temps d'exécution des tâches de l'application. Cette expérimentation a été faite avec une version purement logicielle de l'application exécutée sur un seul processeur embarqué 32 bits. Le processeur utilisé est un Nios2 d'Altera avec une fréquence d'horloge de 100 MHz. Les tâches de l'application peuvent être divisées en trois groupes selon le délai d'exécution :

- les tâches de calcul intensif de type flot de données à temps de traitement constant,
- les tâches à temps d'exécution corrélé avec le nombre de points d'intérêt (figure 3.18)
- les tâches à temps d'exécution imprévisibles (figure 3.19).

Ces résultats préliminaires montrent évidemment qu'il serait intéressant de mettre en œuvre les tâches de calcul intensif sous forme de modules matériels purs (éventuellement en pipeline). D'autre part, les résultats de l'exécution des tâches de recherche, tri et d'extraction confirment la nécessité de leur implémentation en logiciel, et l'inadéquation de l'application à un ordonnancement statique.

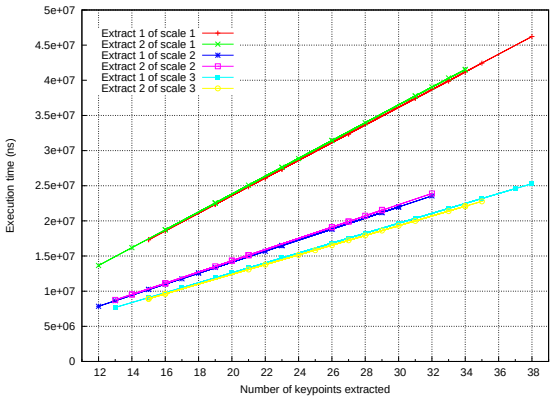


Figure 3.18: *Tâches avec un temps d'exécution dépendant linéairement du nombre de points d'intérêts traités*

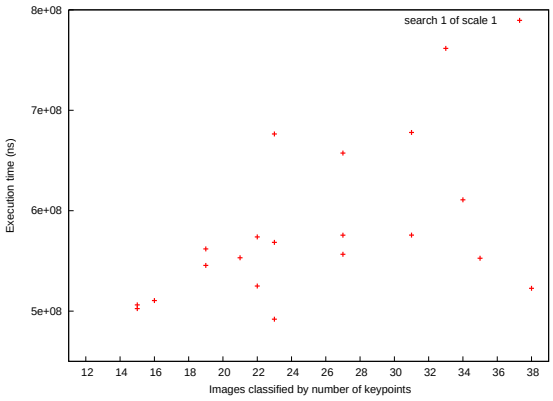


Figure 3.19: *Tâches* search *de recherches des points d'intérêts à la première échelle, caractérisée par un temps d'exécution imprévisible, car ne dépend du nombre de points d'intérêts des images*

Il est reconnu que les systèmes multi-échelle profitent des distinctions que procure un traitement visuel fait à différentes échelles. Dans l'approche classique mais limitée du *grossier au plus fin*, les basses résolutions sont considérées comme prioritaires. Les données visuelles de ces résolutions sont rapidement intégrées dans le système pour obtenir une première description grossière de l'environnement. Cette description est ensuite affinée par d'autres échelles. Plus raisonnablement, le système visuel se comporte d'une manière plus souple, mais reste fondé sur ces différentes priorités. Il peut favoriser l'utilisation des fréquences différentes en fonction des objectifs. Dans une tâche de navigation, cette approche peut être étendue : dans les environnements non-encombrés, une priorité à l'échelle basse peut être donnée pour naviguer grossièrement. La vitesse du robot et/ou d'acquisition visuelle (images par seconde) peut être augmentée. Au contraire, pour la reconnaissance précise d'objets, d'autres points d'intérêt issus d'autres échelles doivent également être pris en compte (d'autres échelles auraient une plus grande priorité) et les vitesses du robot et de la caméra (fps) doivent être réduites en conséquence.

Pourtant, la conception d'un système temps-réel dur impose la connaissance statique sur des bornes supérieures des paramètres de deadline de l'application. Ainsi, notre application, avec ses propriétés particulières, ne semble pas apte à s'exécuter en temps-réel dur avec des temps d'exécution déterministes et prédictibles. Pour cette raison, nous avons défini trois modes (jeux de paramètres et de contraintes) sur la base d'une approche sé-

Table 3.1: *Modes de fonctionnement de l'application*

Mode	N	FPS	deadline	mission
Fast	10	20	50ms	Détection d'obstacle environnement connu
Intermédiaire	30	5	200ms	Exploration ou contournement d'obstacles
Détaillé	120	1	1s	Analyse statique détaillée de l'environnement

lective du grossier au plus fin, qui correspondent aux trois grands types de comportement du robot :

- Mode Fast : le robot se déplace dans un environnement appris, en découvrant peu d'obstacles. Il suffit d'une description grossière du paysage, mais à une cadence élevée. Ainsi, seuls les échelles inférieures sont traitées mais les points d'intérêt de l'image courante doivent être fournis au réseau de neurones (contrôle sensori-moteur dans la figure 3.15) à une cadence suffisante pour la vitesse du robot. C'est la raison pour laquelle dans ce premier mode on fixe le nombre maximum de points d'intérêt extraits à $N = 10$, et le nombre d'images traitées par seconde (fps) à 20.
- Mode intermédiaire : le robot se déplace lentement, par exemple lorsque le passage est bloqué (évitement d'obstacles, passage de porte) et a besoin de plus de précision sur son environnement. Dans ce mode, les échelles moyennes et basses sont traitées. Ainsi les informations sur les points d'intérêt d'une bande de fréquence plus étendue qu'en mode Fast sont envoyés au réseau de neurones. Dans le mode intermédiaire on fixe $N = 30$ et le système fonctionne à 5 images par seconde.
- Mode haut niveau de détails : le robot est arrêté dans une phase de reconnaissance (suivi d'objets, exploration de nouveaux lieux). Toutes les échelles sont entièrement traitées et la recherche des informations exhaustives sur l'environnement visuel est assuré au détriment du temps de traitement. Pour ce dernier mode nous fixons $N = 120$ et la cadence d'images à 1 fps.

Pour l'ensemble de ces modes, résumés dans le tableau 3.1, nous considérons une valeur constante pour le seuil de détection : $\gamma = 200$. Les paramètres du système, tels que le nombre d'images par seconde traitées par le système (et par conséquent la période), ont été définis à partir des mesures sur un échantillon représentatif d'images (acquises par le robot) sur le processeur softcore embarqué utilisé pour déterminer des échéances temporelles réalistes (le Nios II introduit précédemment). En outre, la partie supérieure, en moyenne, et inférieure des bornes de temps d'exécution pour ces trois modes ont été analysées afin d'extraire de la dynamique globale des comportements prédictibles. Cette approche limite la dynamique du système afin que dans chaque mode, le système soit maintenu dans des contraintes temps-réel. Elle permet également d'explorer et de définir le dimensionnement de l'architecture du système sur puce.

Dans chaque mode, et d'un point de vue de la planification des tâches, ces comportements peuvent être modélisés en attribuant des priorités différentes aux tâches d'extraction des caractéristiques locales et de normalisation. En outre, dans certains cas, les tâches les moins prioritaires peuvent ne pas être exécutées, sans endommager de manière significative le comportement du robot. Selon les conditions extérieures, l'exécution ou non des tâches les moins prioritaires doit être intégrés par une information de Qualité de Service (QoS). Cette information sera nécessaire pour assurer une boucle sensori-motrice globale efficace. Un tel *facteur de qualité* pourrait facilement être calculé au niveau local avec le ratio du nombre des caractéristiques extraites par rapport au nombre total de points d'intérêt. Ce scénario d'exécution ne peut être obtenu qu'à l'aide d'un système d'exploitation temps-réel dynamique.

Nous sommes ici confrontés à un double problème. Tout d'abord, intégrer le développement d'un RTOS embarqué dans un flot de conception mixte matériel/logiciel. En second lieu, proposer une méthodologie de conception pour explorer les stratégies d'ordonnancement dynamiques spécifiques à l'application. Une telle approche nécessite un processus de conception méthodique fondé sur le concept de la modélisation de haut niveau qui permet aux concepteurs d'explorer et de valider les solutions de déploiement d'applications sur une architecture dédiée. Elle permet également d'affiner progressivement ce modèle vers l'architecture matérielle finale et la mise en œuvre du logiciel. Aujourd'hui, le développement d'un modèle de haut niveau de RTOS constitue un défi et l'exploration de politiques d'ordonnancement dynamique dédié n'est pas traitée par les approches existantes, telles que [GYJ01], [DLJ97] ou [MPC04]. C'est pourquoi nous avons développé le modèle de système d'exploitation présenté dans cette thèse. C'est ce modèle que nous allons utiliser pour explorer l'architecture de l'application de vision.

3.6.3 Intégration de l'application dans le modèle simple non temporel

Dans un premier temps l'application peut être testée sur le modèle sans aucune modification. Il suffit de modifier le nom du *main* en tout autre nom pour ne pas interférer avec le *main* de SystemC et donner le pointeur de cette fonction au constructeur de l'OS.

Afin de bénéficier des fonctionnalités de l'OS, il est préférable de découper l'application en tâches, et d'utiliser les appels système de ce dernier, pour la création de tâches et les services proposés. Dans le cas de l'application de vision les appels système utilisent la même API (celle de μC/OS-II) que celle que notre modèle d'OS redéfinie. Il n'y a donc pas eu de modification à effectuer et l'application peut ainsi se dérouler sans aucun problème. Le seul besoin est d'inclure la librairie de notre modèle d'OS plutôt que celle du vrai OS μC/OS-II (par un simple changement de `#include`) et indiquer quelle est la tâche pricipale (`ain`).

3.6.4 Rétroannotation de l'application et du modèle

Avec le portage immédiat de l'application, le modèle se comporte comme désiré, mais les temps d'exécution du code des tâches applicatives ne se trouve pas reporté sur le temps de simulation.

Une simulation temporelle est possible avec ce modèle. En effet, des données de synchronisation temporelles peuvent être insérées dans le code SystemC. Des estimations du temps d'exécution peuvent être ajoutés dans le modèle en utilisant une fonction dédiée `OS_wait()` à l'intérieur des modules RTOS de chaque service pour modéliser les durées des appels système. Ce faisant, il devient possible de simuler le surcoût produit par les appels RTOS et les opérations d'ordonnancement. En outre, une évaluation de la performance globale est également possible en donnant le temps d'exécution pire cas (WCET) des estimations de chaque portion de code en tant que paramètres au pseudo appel de service `OS_wait()` dans le code de l'application.

Afin de produire une simulation réaliste, il est donc nécessaire d'annoter les blocs de base des tâches (BB) et des services de l'OS avec des `OS_wait()` pour rendre la simulation temporelle réaliste.

Nous avons donc instrumenté l'application pour mesurer les temps des BBs et lancé un banc de mesure sur une plateforme réelle, à savoir un processeur softcore NIOS-II adjoint de l'OS μC/OS-II implémenté sur une carte FPGA, comme montré sur la figure 3.20. Afin de rendre le modèle complètement réaliste, nous avons aussi instrumenté les services de

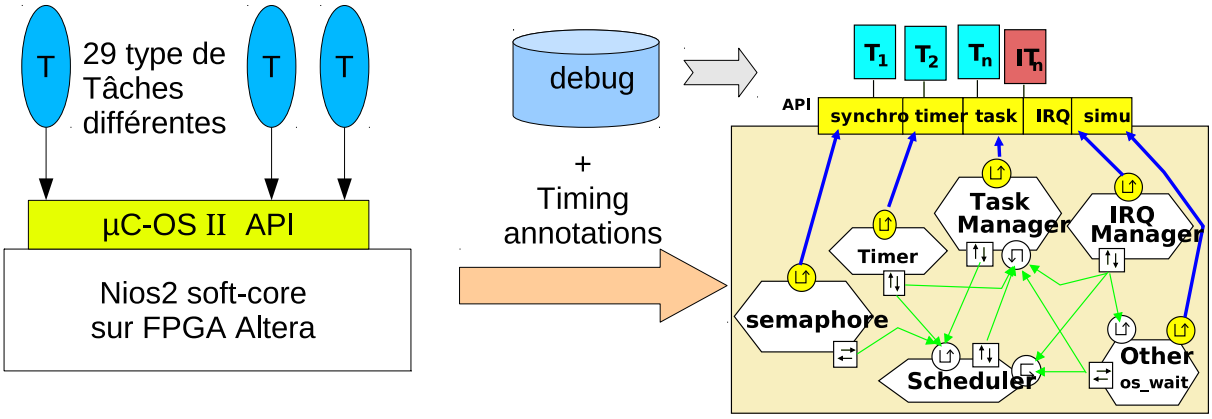


Figure 3.20: *Mesure sur plateforme réelle pour rétroannotation du modèle et de l'application*

l'OS μC/OS-II afin de reporter les délais mesurés correspondant au temps de simulation des services de l'OS modélisé.

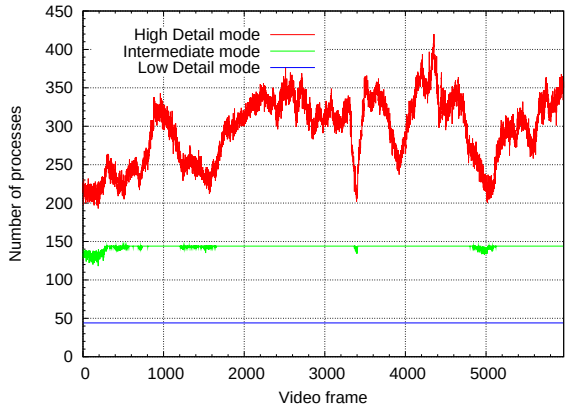


Figure 3.21: *Nombre de tâches créées et gérées par le modèle d'OS pendant la simulation de l'application, selon les trois modes(limitant le nombre de points d'intérêts ou pas du tout).*

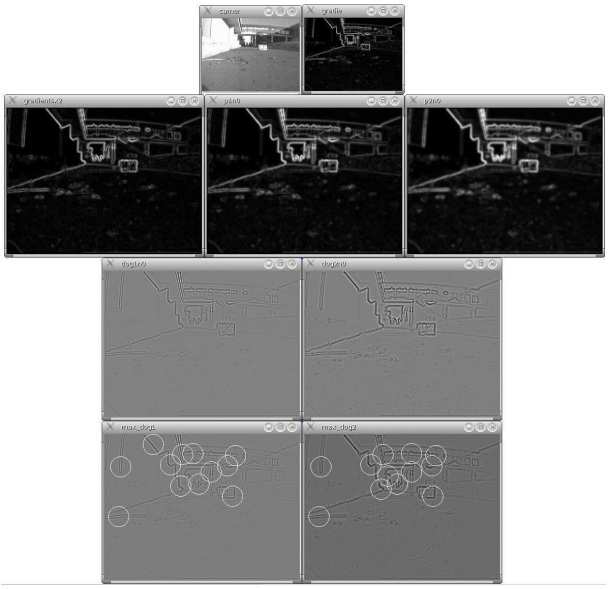


Figure 3.22: *Résultats graphiques de la simulation fonctionnelle du modèle SystemC obtenu en utilisant la bibliothèque C++* `gtk+2.0`.

La simulation de notre application avec le modèle de RTOS produit des résultats intéressants : une interface graphique C++ (bibliothèque `gtk+2.0`) a été incluse dans le code SystemC et nous a permis une vérification fonctionnelle rapide (fig. 3.22).

En outre, le modèle produit des traces d'exécution qui permettent un examen approfondi du comportement de l'application. Par exemple les résultats de la simulation présentés sur la figure 3.21 montrent l'évolution du nombre de processus actifs dans les différents modes en fonction de la complexité de l'environnement visuel, ce qui illustre bien le comportement dynamique du modèle de RTOS. En mode haut niveau de détail, toutes les échelles sont traitées et le nombre de processus dépend du nombre de points d'intérêts dans les images (pas de seuil). Dans cet exemple, le système d'exploitation crée dynamiquement, ordonnance et supprime jusqu'à 420 processus. Dans les modes *intermédiaire* et *rapide* le nombre de points d'intérêt est respectivement limité à 30 et à 10 (soit autant de processus d'extraction et de normalisation) par demi-échelle.

3.6.5 Validation du modèle rétroannoté par rapport à la mesure réelle

Nous avons évalué la précision de notre approche de modélisation en réalisant plusieurs séries de mesures. Ensuite on a comparé le temps d'exécution simulé comparativement aux mesures effectives sur la carte prototype pour un ensemble d'images représentatives. Les durées moyennes de l'application selon la configuration sont indiquées dans le tableau 3.2.

Table 3.2: *Comparaison entre temps d'exécution réel mesuré et temps d'exécution simulé sur le modèle d'OS (moyenne sur 1000 images)*

Mesure sur carte (moyenne)	Estimation par Simulation	Pourcentage d'erreur	Temps de simulation par image
29268.57 ms	28369.59 ms	3.07%	192 ms

Selon ces résultats, la précision de notre modèle de simulation haut niveau est dans les 3-4 % d'erreur par rapport aux mesures réelles. Cette précision est acceptable à ce niveau de description où les éléments de traitement sont analysés. En outre, la modélisation de l'ordonnancement et de la préemption de tâches réalisé par μC/OS-II sur la carte est précisément modélisé par notre modèle de haut niveau. Ces résultats sont a relativiser car la caractérisation (temporelle) d'un OS pour en extraire ses caractéristiques n'est pas triviale, car l'instrumentation est souvent intrusive et perturbe les mesures. On remarque qu'ici, la simulation est plus rapide que l'exécution réelle (un facteur x150), du fait que le processeur hôte de la simulation est beaucoup plus rapide que le processeur Nios-II de la carte Altera cadencé à seulement 100 MHz.

Nous avons également comparé la durée de simulation de notre application, avec le modèle de RTOS ou sans (application brute). La durée de simulation est d'environ 2mn 53s pour l'application décrite sous forme de code C fonctionnel pur (à l'aide de l'API d'accueil Linux) et 3mn 12s lorsque l'application s'exécute sur un modèle SystemC de RTOS. Ces résultats ont été obtenus avec une machine hôte de simulation équipée d'un processeur Intel Dual-core à 1,66 GHz et sur une série de 20 images représentatives.

Ces résultats montrent qu'une méthodologie d'exploration basée sur un OS annoté répond au compromis entre temps de simulation et précision de l'estimation lors des étapes préliminaires de conception afin de modéliser et explorer la concurrence sans tenir compte des spécificités des éléments de l'architecture de traitement. Ces propriétés nous aideront à explorer le dimensionnement de l'architecture dans le chapitre suivant.

3.7 Conclusion du chapitre

Après avoir défini les caractéristiques attendues d'un modèle de simulation temporel et fonctionnel de système d'exploitation, nous avons implémenté les mécanismes nécessaires pour les réaliser de manière cohérente dans l'environnement SystemC, en particulier la capacité du modèle à simuler la préemption du flot d'exécution. Avec un tel modèle, on a montré qu'on est capable de modéliser des tâches logicielles fonctionnelles, temporisées, préemptibles, et capable d'appeler les services de l'OS, le tout dans un simulateur SystemC. Le plus dur étant de s'assurer que l'ordonnancement et la modélisation des évènements sont bien simulés conformément à l'ordre temporel attendu correspondant au comportement d'un véritable OS, ce que nous avons validé en confrontant l'exécution réelle d'une application de vision aux résultats temporels de sa simulation.

L'objectif étant aussi l'exploration des services d'OS conjointement à l'exploration de l'architecture, nous avons donc modifié le modèle d'OS de manière à le rendre modulaire pour faciliter les changements de services et leur agrégation selon les besoins. Plusieurs services peuvent être explorés et validés en remplaçant un module de service par un autre offrant les deux mêmes interfaces (celle de l'API de l'OS et celle interne pour les communications entre modules de services). Nous avons montré dans ce chapitre comment la structure modulaire de notre modèle facilite l'exploration (algorithmique) par l'ajout et/ou l'amélioration de plusieurs implémentations de services de base.

Une des caractéristiques que nous pouvons alors envisager d'explorer est le partitionnement entre logiciel et matériel de certains services, comme le service de sémaphore globalement partagé, que l'on peut envisager de réaliser sous forme de périphériques matériels partagés entre plusieurs nœuds. La distribution de l'application sur plusieurs processeurs ou accélérateurs matériels est aussi un cas d'exploration architecturale que l'on souhaite autoriser. Pour cela, il nous faut développer le modèle afin que plusieurs nœuds d'exécutions/OS puissent être simulés en parallèle ce qui nécessite qu'ils soient capables de communiquer ensemble. Cette distribution de l'application et des services de l'OS sur plusieurs nœuds d'exécution fera donc l'objet de notre étude dans le chapitre 4.

Chapitre 4

Modélisation de distribution de services pour MPSoC

Sommaire

4.1 OS distribués et services distribués, définition

Les OS fournissent des services en se comportant comme une couche d'abstraction du matériel. Lorsque la plateforme possède plusieurs nœuds d'exécution, il est bon de pouvoir représenter les échanges entre ces différents nœuds, et ce de manière transparente pour l'utilisateur comme le concepteur du modèle d'OS.

Généralement, les difficultés se situent autour des problématiques similaires à celles rencontrées dans les réseaux classiques pour la communication et la synchronisation entre les nœuds, avec la différence majeure que les temps de latence sont beaucoup plus courts au sein d'une seule puce. Les communications vont dépendre des supports de la plateforme (bus, NoC) et du modèle de mémoire utilisé, comme décrit plus loin (section 4.3.1).

Nous allons tout d'abord étendre le modèle pour le cas multiprocesseurs, et ensuite développer un module de service partagé entre plusieurs nœuds : un service de sémpahore nécessaire pour les synchronisations et le partage de ressources distribuées.

Dans les sections suivantes nous proposerons un modèle modifié afin de fournir une interface générique de communication entre les modules de services d'un OS, ce qui doit faciliter le remplacement ou l'ajout de nouveaux modules connectés par un port unique, sans avoir à gérer un nombre de connexions allant croissant avec le nombre de modules coopérants. Puis nous proposerons une extension de ces connexions génériques afin de fournir un modèle de distribution de services sur de multiples nœuds OS.

Enfin, nous experimenterons l'utilisation du modèle distribué pour explorer l'architecture idéale de l'application de vision déjà utilisée, en la distribuant sur de multiples nœuds d'exécutions, que ce soit des processeurs et OS logiciels ou des nœuds de types zone reconfigurable pour jouer des tâches matérielles.

4.2 Multi-OS hétérogènes indépendants

Une fois que l'on a défini et implémenté les modules d'un OS, on peut facilement en simuler plusieurs en parallèle. Pour cela, il faut préciser quelles tâches sont associées à quel OS, dès la compilation, à l'aide de macro instructions. L'inconvénient de ce système actuel est que les appels système du code des tâches sont liés statiquement lors de la compilation à un OS donné. Si on souhaite réutiliser le même code pour qu'il soit simulé en parallèle sur plusieurs OS différents, il faut alors gérer un procédé de gestion de la localisation des tâches plus flexible.

4.2.1 Gestion du mapping

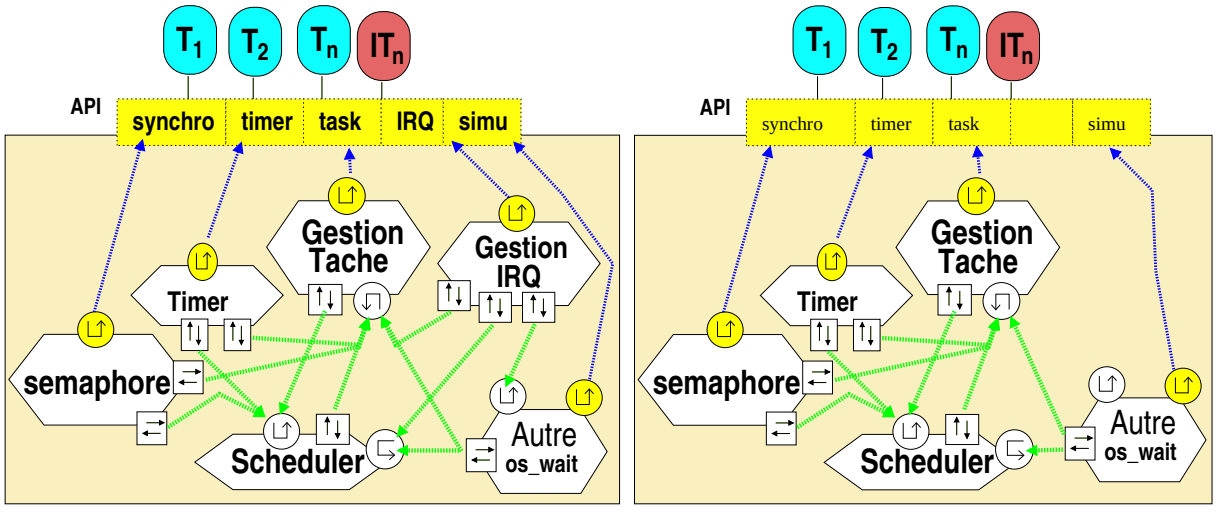


Figure 4.1: *Plusieurs modèles d'OS modulaires différents peuvent coexister et s'exécuter en parallèle*

Afin d'assigner l'OS sur lequel une tâche va s'exécuter, on utilise un tableau global des OS, qui permet de lier les appels système d'une tâche à son OS, lors de la compilation. Le mapping est donc statique, mais cela permet néanmoins de localiser les tâches et d'évaluer une distribution statique sans communication entre les OS, ni possibilité de réutiliser le code d'une tâche pour la jouer sur plusieurs OS.

```
// OS_LOCAL est le numero d'OS, a definir pour chaque tache
#define OSMutexPend os[OS_LOCAL-1]->BASIC_MUTEX_IF->OS_mutex_lock
#define OSMutexPost os[OS_LOCAL-1]->BASIC_MUTEX_IF->OS_mutex_unlock
```

La figure 4.1 montre l'exécution de deux OS avec leurs tâches respectives s'exécutant en parallèle, mais sans aucune interactions possibles entre les nœuds. Si l'on veut étendre le modèle pour faciliter la réutilisation du code, nous avons besoin d'un modèle plus flexible.

4.2.1.1 Contexte d'exécution de tâches

SystemC est un système hiérarchisé de modules contenant les threads exécutés. En utilisant la commande `sc_spawn()`, le thread créé sera un descendant du thread l'ayant créé et donc appartiendra indirectement au module du créateur. Autrement dit, tous les threads créés seront exécutés dans le contexte d'exécution du module (de service) contenant le thread original d'initialisation, quelque soit l'emplacement du code C correspondant. Dans ce cas, les nouveaux threads créés auront une hiérarchie de dépendance similaire à la parenté des processus UNIX, comme on peut le voir dans la figure 4.2.

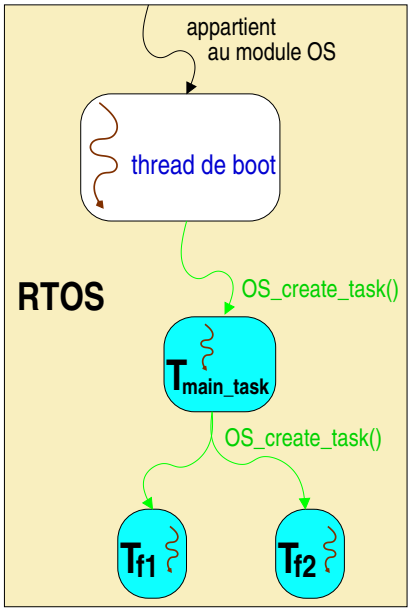


Figure 4.2: *Hiérarchie des threads et modules SystemC*

Afin de pouvoir accéder aux services du système donc faire des appels système, nous allons nous servir de cette hiérarchie afin de remonter jusqu'au niveau du module de l'OS afin d'appeler dynamiquement l'OS local puis le sous composant de cet OS en charge du service demandé. Cela permettra d'éviter de mettre l'objet OS en variable globale, et de compiler statiquement le code d'une tâche instanciée N foix tout en permettant la découverte dynamique à l'exécution de l'appartenance d'une tâche à un OS.

4.2.1.2 Localisation dynamique des appels système : get_OS

Une manière simple de localiser les appels système de tâches compilées indépendamment de l'OS est de définir une fonction utilisant les capacités de réflexivité de C++ et de la hiérarchie des modules et threads SystemC. Cela est possible grâce à la capacité de SystemC de fournir le contexte d'exécution n'importe où dans le code, à travers la fonction `sc_get_curr_simcontext()` et de remonter la hiérarchie avec `get_parent_object()`. La capacité reflexive du C++ permettant de connaître l'état courant (l'*introspection*) et de s'en servir pour modifier un algorithme (l'*intercession*), dans notre cas la capacité de transtyper dynamiquement un object avec `dynamic_cast<>` lorsque cela est possible.

Voici le code de la fonction `get_os()` qui permet de remonter la hiérarchie jusqu'au module OS parent du thread de la tâche en cours :

```
#include "my_os.h"
my_os* get_os(void)
{
  sc_object* obj;
  my_os* api = NULL;
  obj=sc_get_curr_simcontext()->get_curr_proc_info()->process_handle;
  while ( obj != NULL && api == NULL)
  {
    obj = obj->get_parent_object(); //on remonte dans la hierarchie
    if(obj==NULL){ cout<< endl<<"error get_os"<<endl;}
    api = dynamic_cast<my_os*>(obj); // NULL si inconvertible
  }
  if (api == NULL){cout << " get_os() NO API found" <<endl;}
  return api;
}
```

Ainsi, à l'aide de cette fonction `get_os()`, on retrouve l'OS qui est le module parent des modules de service, dont celui de création des tâches qui est lui même le parent des `SC_THREAD`s SystemC abritant les tâches simulées.

Ainsi l'interface sera définie par une macro qui inclue une fonction de recherche dynamique de l'OS dans laquelle une tâche se situe, ce qui permet de ne plus lier statiquement le code d'une tâche à un OS :

```
#define OSMutexPend get_os()->BASIC_MUTEX_IF->OS_mutex_lock
#define OSMutexPost get_os()->BASIC_MUTEX_IF->OS_mutex_unlock
etc.
```

Il est ainsi possible de lancer plusieurs fois le même code fonctionnel sous forme de multiples tâches localisées sur de nombreux processeurs.

4.2.1.3 Localité des SC_THREADs créés

Il y a néanmoins un cas où ceci n'est plus valide. En effet les appels SystemC s'exécutent dans le contexte du thread SystemC appelant et lors de la création de tâche par une autre, le nouveau thread crée est hiérarchisé sous ce thread. Il est donc le fils de son géniteur comme le veux la logique lors de la création de processus sous UNIX. Le problème est que si une tâche parente a fini de s'exécuter, alors le kernel SystemC peut décider de le retirer de la mémoire (pas nécessairement immédiatement) et la chaîne hiérarchique se retrouve rompue, car SystemC ne gère pas le rattachement de l'orphelin à l'objet *grand-père*.

Dans ce cas la fonction `get_os()` ne fonctionne plus et provoque un crash du simulateur dont la source est difficile à retrouver.

Nous avons trouvé une modélisation permettant d'éviter ce cas de figure et par ailleurs de garantir la localité d'exécution du thread. La solution est la suivante : que la création de tous les SC_THREADs des tâches soit assujettie à un thread unique qui ne sera jamais détruit par construction. Le module responsable du service de création de tâches va remplir cette fonction.

Ce module contient un thread sensible à un évènement SystemC qui sera présent indéfiniment. Lors de l'appel d'une fonction de création de tâches, les paramètres sont stockés dans le module et l'évènement déclencheur de ce thread particulier est relâché. Alors le thread exécute sa tâche de création à l'aide de la fonction SystemC `sc_spawn()` en récupérant les paramètres stockés, puis se remet en attente d'une nouvelle demande. Ainsi chaque nouveau thread créé aura pour parent direct ce thread spécifique au service de création de tâche, qui par construction ne sera jamais détruit ni enlevé de la mémoire de la simulation.

Par ce moyen, la localité des threads est maintenue tout au long de la vie du système modélisé. De plus l'avantage est que la fonction `get_os()` a ainsi une profondeur de remonté fixe : thread courant ⇒ thread créateur ⇒ module de création des tâches ⇒ OS, comme indiqué sur la figure 4.3. Grâce à ce mécanisme, le code d'une tâche n'est pas lié explicitement à l'OS sur lequel il va s'exécuter et nous garantissons le bon fonctionnement tout au long de la simulation.

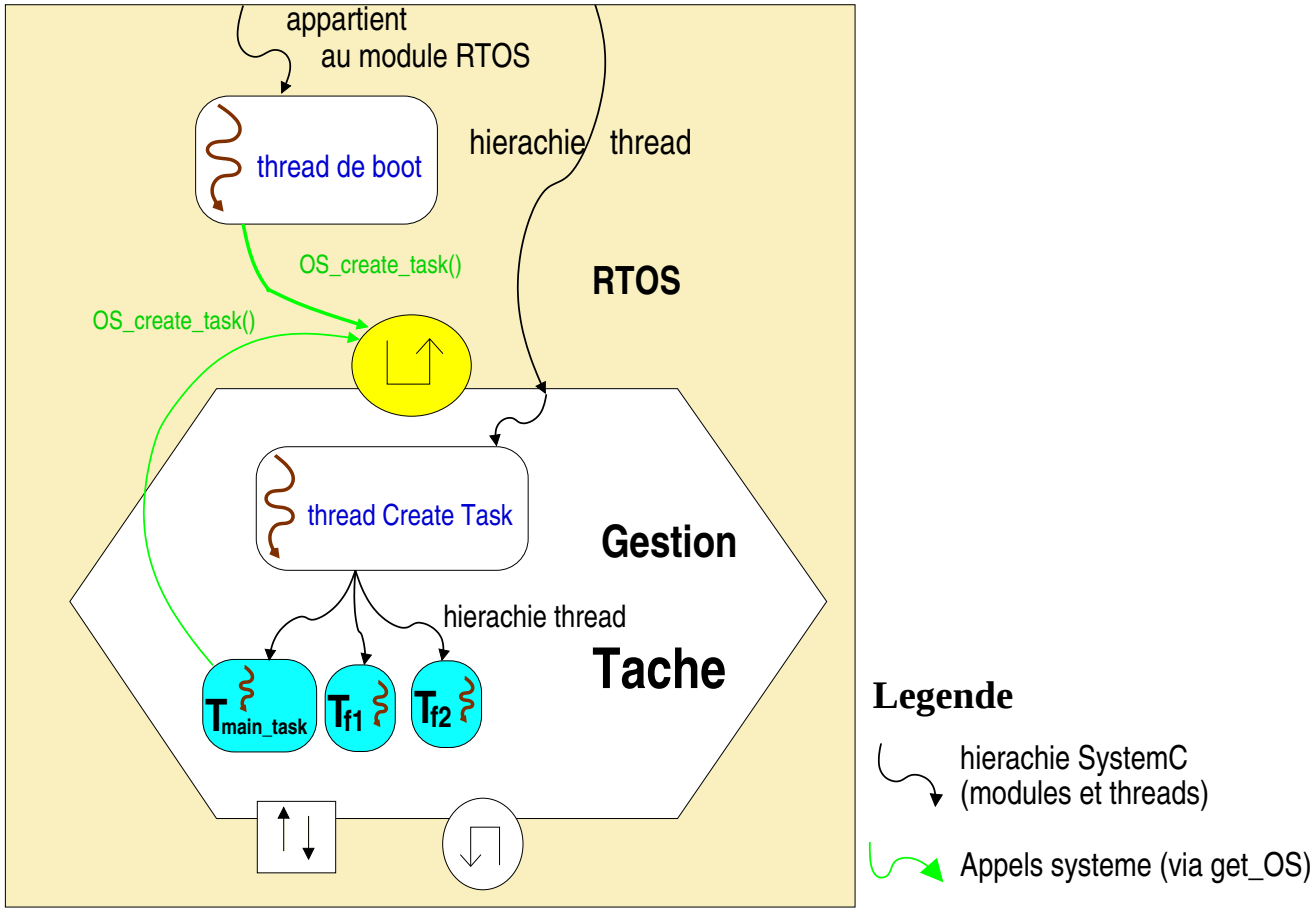


Figure 4.3: *Hiérarchie des threads et modules SystemC régulière et localisée*

On peut donc dynamiquement lancer une création de tâche sur n'importe lequel des OS du système, les appels système qu'une tâche utilisera seront dynamiquement redirigés vers l'OS de la tâche appelante puis vers le module responsable du service demandé. Ce procédé permettra également par la suite d'être capable de créer des tâches depuis un nœud d'exécution dans un autre nœud distant.

4.2.2 Multi OS hétérogènes indépendants

A partir de ce système, on peut faire tourner dans la même simulation de nombreux OS avec leur application respective basée sur le même code source, en se contentant d'indiquer sur quel OS va s'exécuter chacune des tâches *main*. Les sous tâches ensuite créées seront logiquement crées dans leur OS local. Une même tâche n'a alors besoin d'être compilé qu'une seule et unique fois pour être exécutée sur le même nœud de calcul ou dans plusieurs OS distincts sans modification.

Nous avons testé la scalabilité du modèle en jouant la même application (l'application simple du chapitre précédent, listing 3.2) sur un grand nombre de nœuds en parallèle, de 1 à 500, en mesurant le temps système et utilisateur (cumulé) utilisé par la simulation. Les résultats sont donnés dans la figure 4.4. On peut dire que le modèle est scalable, car

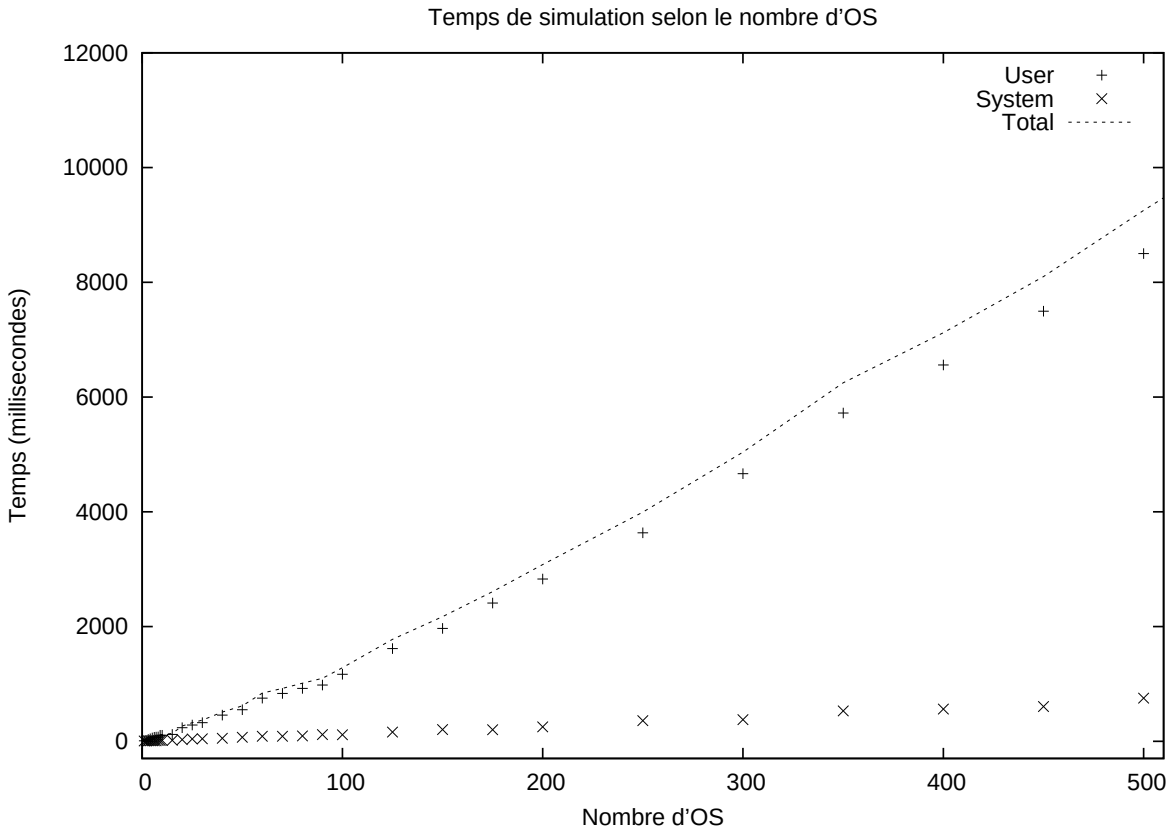


Figure 4.4: *Études simple de la scalabilité temporelle du modèle en fonction du nombre de nœuds simulés en dupliquant l'application monoprocesseur simple (listing 3.2 page 59)*

l'augmentation du temps de simulation progresse quasiment linéairement en fonction du nombre de nœuds OS simulés, bien qu'à partir du seuil de 100 OS, le temps de simulation progresse alors de manière un peu plus importante. Cela est sans doutes dû à des soucis de défaut de pages (car plus de données à gérer), et au noyau SystemC qui doit parcourir la liste des SC_THREADs alors de plus en plus grande, et la capacité du système hôte a gérer de nombreux threads, avec la mémoire associée.

4.3 Service partagé et connexion de services distants

4.3.1 Approche de modélisation des communications dans les systèmes distribués

Modèles de programmation distribuée

L'exécution d'une application multi-tâche sur une architecture multiprocesseur est généralement plus longue, en nombre d'opérations, que celle implémentée sur une unique ressource de calcul. Ceci est principalement dû aux ajouts nécessaires de communications et de synchronisations entre les tâches distribuées sur les nœuds, habituellement gérées/masquée par le système d'exploitation local et l'utilisation d'une mémoire commune. La gestion de variables globales partagées impose également l'utilisation de mécanismes garantissant l'intégrité des données qui induisent des latences supplémentaires et réduisent les performances globales du système.

En pratique, le programmeur définit des sections critiques plus ou moins grandes afin de garantir l'exécution des sections de code concurents qui risquent des problèmes d'intégrité et de cohérence, selon le modèle de programmation choisi pour la plateforme.

Ces modèles de programmation doivent permettre de remplir deux fonctions contradictoires : réduire le temps et les coûts de développement par le biais de modèles de haut niveau, et par ailleurs améliorer les performances du système en utilisant les modèles (et donc les primitives d'une API) de programmation bas niveau de l'architecture.

Un modèle de programmation parallèle permet au développeur de s'abstraire de l'hétérogénéité des différentes plateformes matérielles et logicielles ainsi que des communications, en utilisant une interface de programmation unifiée (API) d'une librairie.

Deux principaux modèles de programmation parallèles sont traditionnellement reconnus dans les architectures parallèles : le modèle à mémoire partagée et le modèle par passage de messages. Aucune des deux solutions ne présente un avantage majeur mais l'utilisation d'une mémoire partagée est souvent considérée par les développeurs comme étant plus simple.

D'autre modèles peuvent être rencontrés tels que le traitement parallèle des données (Single Program Multiple Data ou SPMD), le modèle data-flow, ou le traitement systolique[CSG98].

Une API définit concrètement les opérations de communication et de synchronisation que l'application peut utiliser, ces dernières étant implémentées sur la plateforme grâce aux compilateurs et aux librairies liées à l'architecture, au langage, à l'OS et éventuellement au middleware. Les deux librairies les plus utilisées sont OpenMP [DM98] et MPI [For08, Pac96]. OpenMP est basé sur l'utilisation d'une mémoire partagée pour autoriser ou non l'exécution de sections critiques alors que MPI utilise le passage de messages.

Afin de définir ces interfaces convenablement, il faut pouvoir simuler le logiciel conjointement au matériel, et définir un moyen de joindre les couches logicielles et matérielles. Les auteurs de [JBP06] regroupent le terme d'interface pour les MPSoC sous le terme *Hardware dependant Software, HdS* et proposent un MoC à quatre niveaux, incluant le matériel et son interface, cherchant à expliciter : les ressources matérielles, les stratégies de gestion et de contrôle de ces ressources, l'architecture du/des processeurs et son implémentation.

Le modèle SAT que nous allons définir dans cette thèse s'en rapproche, bien que focalisé uniquement sur les aspects d'exploration des modélisations temporelles des stratégies de gestion et de contrôle des ressources, en utilisant SystemC comme support des autres niveaux de modélisation.

Programmation par mémoire partagée

Le modèle dit *à mémoire partagé* se définit par le fait qu'une ou plusieurs portions d'une application peuvent communiquer au moyen de lectures et écritures dans des variables partagées. Ces espaces de mémoire sont localisés dans un même espace d'adressage (à une adresse prédéfinie, généralement par le linker), nécessairement partagée par les applications. De ce fait, les applications résident dans le même espace d'adressage (au moins en partie).

Ce modèle implique un fort couplage entre les entités communicantes pour se synchroniser. Puisque les adresses mémoire sont partagées, il faut autoriser et contrôler qu'une seule écriture puisse s'effectuer à la fois, afin de garantir la consistance mémoire. En revanche plusieurs lectures peuvent s'effectuer en parallèle. Pour se faire, un *écrivain* doit acquérir un *verrou* avant d'écrire. Cela requiert un mécanisme matériel spécifique pour réaliser cette synchronisation de manière indivisible et atomique (Test & Set des processeurs, voire mémoires transactionnelles [HM93]). De plus, plusieurs écritures successives doivent être sérialisées afin de garantir l'ordre des accès et éviter les interblocages (*race conditions*).

De nos jours les API standards comme POSIX proposent des primitives de synchroni-

sation (comme les sémaphores [Dij65]) et d'accès à des mémoires partagées. Cela permet à différents processus ou threads s'exécutant sur un ou plusieurs processeurs de communiquer simplement au travers de ces variables partagées, ou qu'elles soient, et cela de manière transparente.

Les architectures classiques de type SMP (Symmetric Multiprocessor) avec une architecture mémoire de type UMA (Uniform Memory Architecture) garantissent un temps d'accès identique depuis les différents processeurs, ce qui facilite la gestion cohérente des lectures/écritures.

L'inconvénient est que ce modèle d'architecture ne supporte pas la mise à l'echelle (*n'est pas scalable*) pour le cas de très nombreux processeurs, ou dès que seulement deux processeurs sont hétérogènes, ce qui est fréquent dans le monde de l'embarqué. Dans ce cas là, la cohérence et l'ordonnancement des lectures/écritures comme la garantie de l'exclusion mutuelle [AKH03] devient un casse-tête à résoudre au niveau matériel, comme au niveau logiciel.

De plus, le fait de rendre ces communications implicites pour le programmeur ne favorise pas l'écriture de code facilement parallélisable, car les outils manquent alors d'indications utiles pour le réaliser efficacement. Le choix du mapping des différents processus/threads sur les différents nœuds de la plateforme va alors jouer un rôle prépondérant sur l'efficacité effective et les temps des communications impliqués par les synchronisations distantes implicites.

Programmation par passage de message

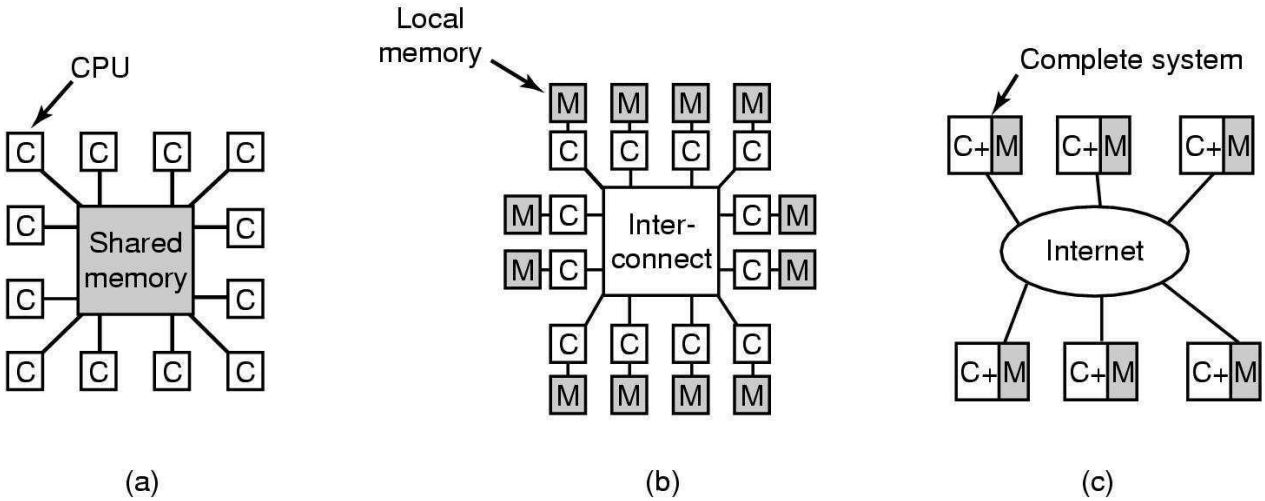


Figure 4.5: *Architecture de communication par (a) Mémoire partagée, (b) Message Passing, (c) Distribuée et hybride*

Avec le modèle par passage de messages [Bal95], les différentes parties d'une application communiquent en s'envoyant et en recevant des messages. De ce fait, les différents processus peuvent avoir des espaces mémoire totalement privés. De plus, cela permet de relâcher le couplage entre l'émetteur et le récepteur, en utilisant un mécanisme de bufferisation autorisant à produire plus vite que les données ne sont lues/utilisées. Une API qui fournit ce genre de primitives de *send* et *receive* est par exemple le standard MPI [For08].

L'avantage de cette méthode est de permettre de distribuer facilement une application sur de multiples nœuds d'exécution, ce qui est de plus en plus courant au vu de la charge de calcul des applications modernes.

En revanche, les inconvénients sont multiples : le programmeur doit expliciter ces échanges ; par ailleurs le fait de devoir formater les messages ajoute un surcoût à de simples

écritures en mémoire, et dans le cas où les deux entités en communications seraient sur le même nœud d'exécution, cela revient à ajouter une couche protocolaire non nécessaire.

Combinaison des deux modèles

Les deux modèles peuvent être utilisés selon les besoins, ou alors conjointement de manière combinée.

Chaque nœud de calcul peut avoir sa propre mémoire locale, avec une partie privée et une partie partagée qui correspoond à une mémoire physiquement distribuée, mais logiquement partagée. Cette architecture se nomme NUMA pour (Non-Uniform Memory Architecture), ou les temps d'accès aux mémoires sont dépendants et varient en fonction de leur localisation. Un accès à la mémoire locale sera toujours plus rapide que dans une mémoire distante.

Dans ce cas le contrôleur mémoire détermine si l'accès est local ou s'il donne lieu à un message vers le contrôleur distant (à l'aide d'un DMA). Cette solution de communication consiste à utiliser uniquement des ports de communications spécifiques (comme les cartes réseau ou les DMA), ce qui risque de demander d'interrompre le récepteur ou l'émetteur par interruption à chaque communication pour l'en informer.

A l'opposé des API classiques pour les systèmes standards (non embarqués) tels que POSIX (Portable Operating System Interface) ou MPI (Message Passing Interface), les API pour les MPSoC ne peuvent être complètement standardisées car elles doivent être dédiées à la plateforme sous-jacente elle même personalisée et souvent hétérogène pour permettre de remplir les contraintes de performance d'une application ou d'un domaine spécifique.

Avec l'apparition des systèmes malléables au niveau portes logiques, de nombreuses solutions intermédiaires ou spécifiques peuvent répondre aux besoins de certaines contraintes des applications pour leur échange de données, comme l'utilisation de buffers dédies en matériel (cas des processeurs systoliques comme le transputer), ou de canaux de communication évolués comme les réseaux-sur-puce (NoC [BM06]).

Tout cela signifie que le raffinement des communications (bus, mémoires et protocoles) peut avoir un impact majeur sur l'efficacité d'une architecture, qu'il est nécessaire d'explorer pour correspondre au mieux aux besoins et contraintes d'une application.

Par ailleurs, ces modèles sont fortement liés aux aspects de programmation logicielle, et ne prennent pas en compte des entités matérielles comme les IP reconfigurables, qui sont de plus en plus à intégrer comme éléments importants du flot de calcul.

Le service de communication du programmeur

Aujourd'hui, les points d'entrée pour accéder à une ressource distante (un service) sont fournis par des API dédiées, facilitant la tâche du programmeur. On retrouve souvent les RPC (Remote Procedure Call) dans le monde UNIX, et de plus en plus souvent, l'utilisation de middleware de type CORBA s'ajoute au système d'exploitation afin d'étendre les fonctionnalités de communications transparentes.

Les communications physiques entre nœuds de calcul distants sont généralement décrites dans le monde des réseaux au travers du prisme du modèle OSI, qui distingue les communication sous forme de couches. Les RPC ou CORBA s'appuient sur ces protocoles lorsque cela est nécessaire, et de manière transparente pour l'utilisateur.

Les RPC Les RPC sont aujourd'hui une norme ([Sri95]) issue des travaux de Birrell et Nelson [BN84] utilisées pour les appels de fonctions entre processeurs distants. Cela se base sur l'idée qu'un appel de fonction, et particulièrement un appel système, est un mécanisme de transfert de donnée et du contrôle de l'exécution vers l'environnement

d'exécution de la fonction appelée. Ce mécanisme peut être étendu à un transfert de données et de contrôle à travers une communication réseau. Quand une procédure distante est invoquée, le processus appelant est suspendu, et les paramètres sont passés à travers le réseau vers l'environnement d'exécution distant. Lorsque l'appel de fonction distante est réceptionné, la procédure souhaitée est exécutée. Lorsque la procédure se termine et produit ses résultats, ils sont transmis vers l'environnement d'appel, où l'exécution reprend comme un simple retour d'appel sur une machine unique. Pour cela il faut disposer de processus de type serveur capable de traiter les messages de demande d'un coté et le retour des résultats de l'autre. Ce fonctionnement impose de rendre l'appel bloquant, et ainsi permettre à un autre processus de s'exécuter en attendant la réponse, mais aussi d'expliciter l'adresse de la requête. Le modèle RPC est plutôt restreint pour décrire des systèmes distribués complexes, car il repose sur un modèle maître/esclave point-à-point.

Le modèle Proxy Skeleton de CORBA CORBA [OMG98] est développé par l'OMG, tout comme le langage de description UML. L'objectif de ce groupe est de faire émerger des standards pour l'intégration d'applications distribuées hétérogènes à partir des technologies orientées objet. Ainsi les concepts-clés mis en avant sont la réutilisabilité, l'interopérabilité et la portabilité de composants logiciels. L'élément clé de la vision de l'OMG est CORBA (Common Object Request Broker Architecture) : un *middleware* orienté objet. Ce bus d'objets répartis offre un support d'exécution masquant les couches techniques d'un système réparti (système d'exploitation, processeur et réseau) et il prend en charge les communications entre les composants logiciels formant les applications réparties hétérogènes.

Le bus CORBA propose un modèle orienté objet client/serveur d'abstraction et de coopération entre les applications réparties. Chaque application peut exporter certaines de ses fonctionnalités (services) sous la forme d'objets CORBA : c'est la composante d'abstraction (structuration) de ce modèle. Les interactions entre les applications sont alors matérialisées par des invocations à distance des méthodes des objets : c'est la partie coopération. La notion client/serveur intervient uniquement lors de l'utilisation d'un objet : l'application implantant l'objet est le serveur, l'application utilisant l'objet est le client. Bien entendu, une application peut tout à fait être à la fois cliente et serveur.

Le langage OMG-IDL (Interface Definition Language) permet d'exprimer, sous la forme de contrats IDL, la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci. Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, c'est-à-dire les types d'objets (ou interfaces IDL) et les types de données échangés entre les objets. Le contrat IDL isole ainsi les clients et fournisseurs de l'infrastructure logicielle et matérielle les mettant en relation à travers le bus CORBA. Les contrats IDL sont projetés en souches IDL (ou interface d'invocations statiques, en anglais Proxy) dans l'environnement de programmation du client et en squelettes IDL (ou interface de squelettes statiques, en anglais Skeleton) dans l'environnement de programmation du fournisseur. Le client invoque localement les souches pour accéder aux objets. Les souches IDL construisent des requêtes, qui vont être transportées par le bus, puis délivrées par celui-ci aux squelettes IDL qui les délégueront aux objets. Ainsi le langage OMG-IDL est la clé de voûte du bus d'objets répartis CORBA.

4.3.2 Modélisation de services distants partagés

Notre approche pour modéliser la distribution des services d'OS s'inspire de la philosophie de middleware comme le standard CORBA. Cela consiste à utiliser des couples

de modules de service appelés *proxy* et *skeleton*. Le proxy offre la même interface que le service distant et sert de point d'entrée local au service. Le proxy appelle le service du skeleton (service distant) via un port spécifique au travers d'une infrastructure de communication qui arrivera sur l'interface exportée par l'OS distant du module skeleton.

Pour produire des modules de services partagés, il est nécessaire que le module skeleton comporte une troisième interface, l'interface externe (cf. figure 4.6).

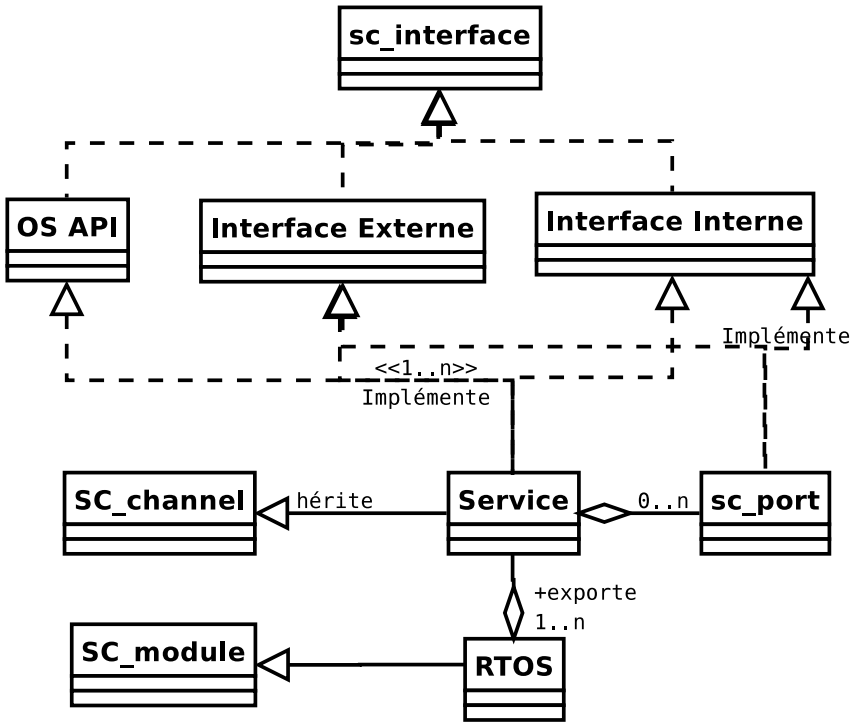


Figure 4.6: *Diagramme de classes UML du modèle de système d'exploitation avec connexion de services distants en SystemC*

L'interface du service du skeleton est exportée par l'OS le contenant (comme les services normaux d'un OS), mais celle-ci n'est pas destinée à être vue par l'application. Selon le service skeleton, le module comporte ou non une interface d'OS (API exportée pour l'application) en plus de l'interface interne proposée à des modules externes à l'OS local.

Afin de faire fonctionner le tout, il faut connecter le port du proxy à un port de l'OS, pour ensuite se connecter sur l'interface externe proposée par l'OS distant, elle même redirigée vers le module de service skeleton, comme on peut le voir sur la figure 4.8. Cela complexifie encore le nombre de connexions à gérer lors de la conception du système, mais nous verrons dans la section suivante les moyens employés pour limiter le nombre de connexions.

Par ce procédé, on peut ainsi modéliser des OS qui sont des enveloppes vides qui redirigent les appels système vers un OS central qui aura tous les modules skeletons et centralisera la gestion de la plateforme, comme dans un modèle SMP.

Le fonctionnement actuel ne représente pas de module de communication, et il serait sans doute nécessaire de l'ajouter afin de représenter plus fidèlement le comportement d'un bloc distinct de communication surtout si plusieurs services sont distribués. Si l'on veut basiquement expliciter les temps de communications, on pourrait se contenter d'ajouter un temps arbitraire dans le code du service skeleton. Étant donné que le contexte d'exécution d'une méthode d'un module distant s'exécute dans le contexte d'exécution SystemC du thread appelant, alors on modéliserait bien le temps d'attente coté proxy relatif à l'envoi, au traitement par le skeleton et au retour de la requête. Cela constitue un

appel bloquant jusqu'à ce que le traitement distant renvoie le résultat. Cela peut suffire pour une simulation rapide, mais ne correspond pas au comportement réel que l'on veut simuler.

En effet, le traitement par le service skeleton doit consommer du temps CPU sur l'OS distant du skeleton. Pour cela, on va appliquer le même principe de localité que pour la création de tâches. Le module skeleton va déclarer un traitant d'interruption qui fera le travail localement et consommera du temps du nœud local. La fonction proposée par le skeleton (par exemple `get_sem()`) ne fera plus le traitement, mais se contentera de stoker les paramètres, et déclenchera une pseudo interruption déclenchant le traitant d'interruption dédié, qui sera joué lorsque l'ordonnanceur local lui donnera le droit de consommer du temps de calcul. On notera que ce système de communication où l'opérateur distant écrit les données puis déclenche une interruption pour notifier qu'un message est arrivé correspond bien à la réalité des cartes réseau et autres modules matériels de communication.

Par ce biais, on fait bien consommer du temps à l'OS du skeleton.

Mais la réponse de la requête ne peut être transmise car l'appel de la fonction `get_sem()` par le proxy est terminée instantanément après le procédé déport/déclenchement d'interruption. On pourrait imaginer utiliser les évènements SystemC pour synchroniser la méthode d'interface `get_sem()` avec la fin du traitement de l'interruption, mais cela figerait le modèle dans une configuration où le proxy ne peut faire que des appels bloquants jusqu'à la réponse. Une solution plus viable est de séparer les communications aller et retour.

Le retour va fonctionner sur le même principe, mais coté proxy. On doit donc implémenter un traitant d'interruption dans le proxy coordonné avec une fonction corollaire (`get_sem_retour()`) qui déclenchera ce traitant. Cela signifie que le proxy doit proposer une interface accessible depuis le skeleton. Autrement dit, les modules proxy/skeleton doivent tous les deux avoir un port pour envoyer la requête ou la réponse, et présenter une interface externe pour recevoir la requête ou la réponse, soit la nécessité de posséder un couple port/interface de retour pour les deux modules, comme indiqué sur la figure 4.7.

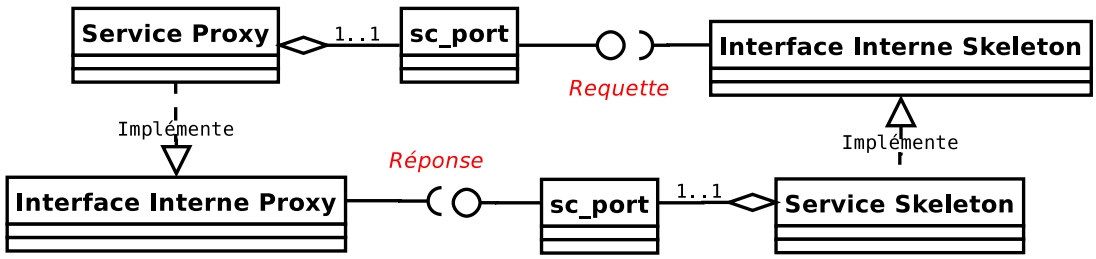


Figure 4.7: *Couple Proxy/Skeleton présentant chacun un couple port/interface pour les communications*

L'avantage de ce système est que le temps d'attente et de calcul est bien *consommé* dans le bon nœud d'exécution. Par ailleurs, il est aussi possible de raffiner les connexions afin de détailler les communications avec un ou plusieurs modules qui vont consommer du temps au fur et à mesure du transfert sur le système de communication, comme nous le verrons dans le cadre du projet Ter@ops (cf. section 5.3).

4.3.3 Exemple de services partagés : la synchronisation

Une façon simple de déporter un service dans un autre OS est la suivante : le service local continue de fournir la fonctionnalité aux tâches locales de la même manière. En revanche, le module de service proxy va faire appel au module de service skeleton d'un nœud de calcul distant pour effectuer l'exécution de ce dernier.

Nous avons envisagé de modéliser un service de synchronisation partagé, que nous envisageons d'implémenter en matériel. Dans notre cas, c'est un service de sémaphore qui est représenté seul dans un OS / nœud ne présentant en fait que ce service skeleton (considérons le comme une implémentation matérielle) comme montré dans la figure 4.8. De ce fait, seule l'interface externe est nécessaire.

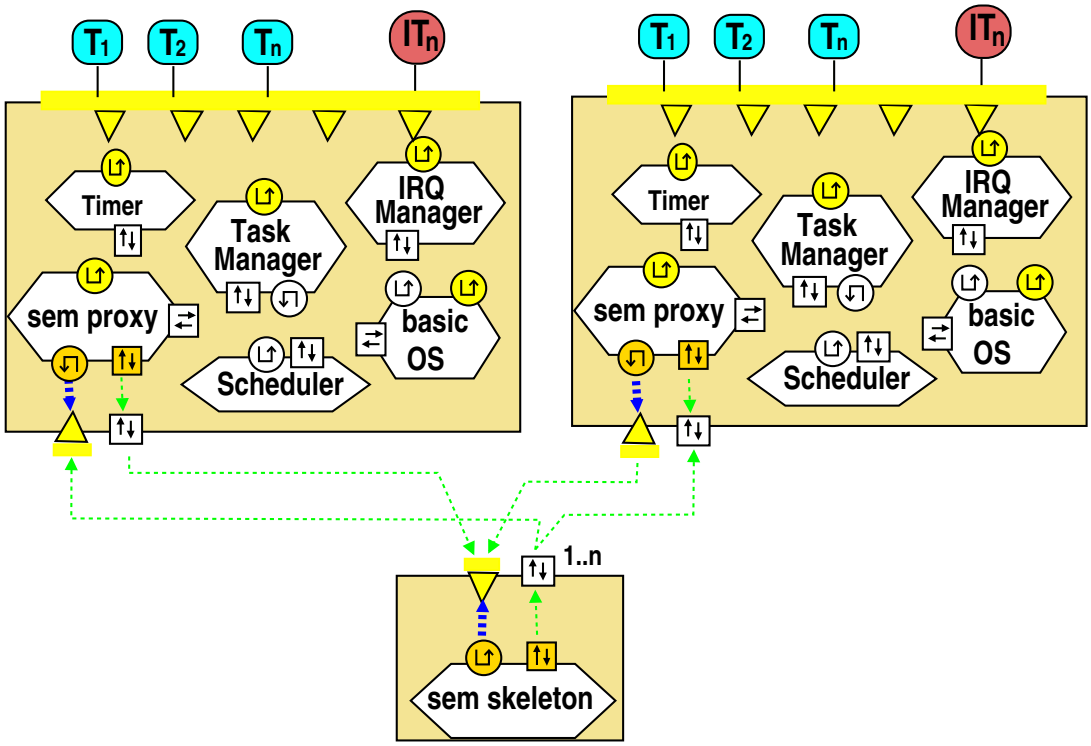


Figure 4.8: *Multi-OS avec service partagé distant*

Nous avons testé le modèle en répartissant les deux tâches de l'application simple qui se partage un sémaphore pour protéger des données critiques. La figure 4.9 représente le diagramme de séquence de l'exécution du principe de l'appel à une fonction déportée.

Dans le cas du service de sémaphore, on a constaté que le proxy doit nécessairement posséder un pseudo appel système interne de plus, dans le cas ou le sémaphore distant qui est libéré libère une tâche locale. Dans ce cas le service distant doit l'indiquer au proxy de l'OS de la tâche à libérer. Il suffit pour cela d'ajouter un service à l'interface externe, comme pour la fonction `get_sem_retour()`.

On se rend compte, que le fait de déporter l'exécution d'un service dans un autre nœud, non seulement ajoute des temps de communication variables, mais demande au concepteur de repenser la sémantique des appels système. Cela signifie que la distribution d'un service peut nécessiter de repenser le fonctionnement impliqué par la concurrence d'exécution (section critiques) et la qualité de bloquant ou non bloquant des appels système originaux.

On voit ici que notre modélisation d'OS s'étend à n'importe quel nœud de calcul, quel que soit sa nature (logiciel ou matériel) et que, à haut niveau de modélisation, l'implémentation distante de l'algorithmique d'un service reste correcte du point de vue comportemental et temporel.

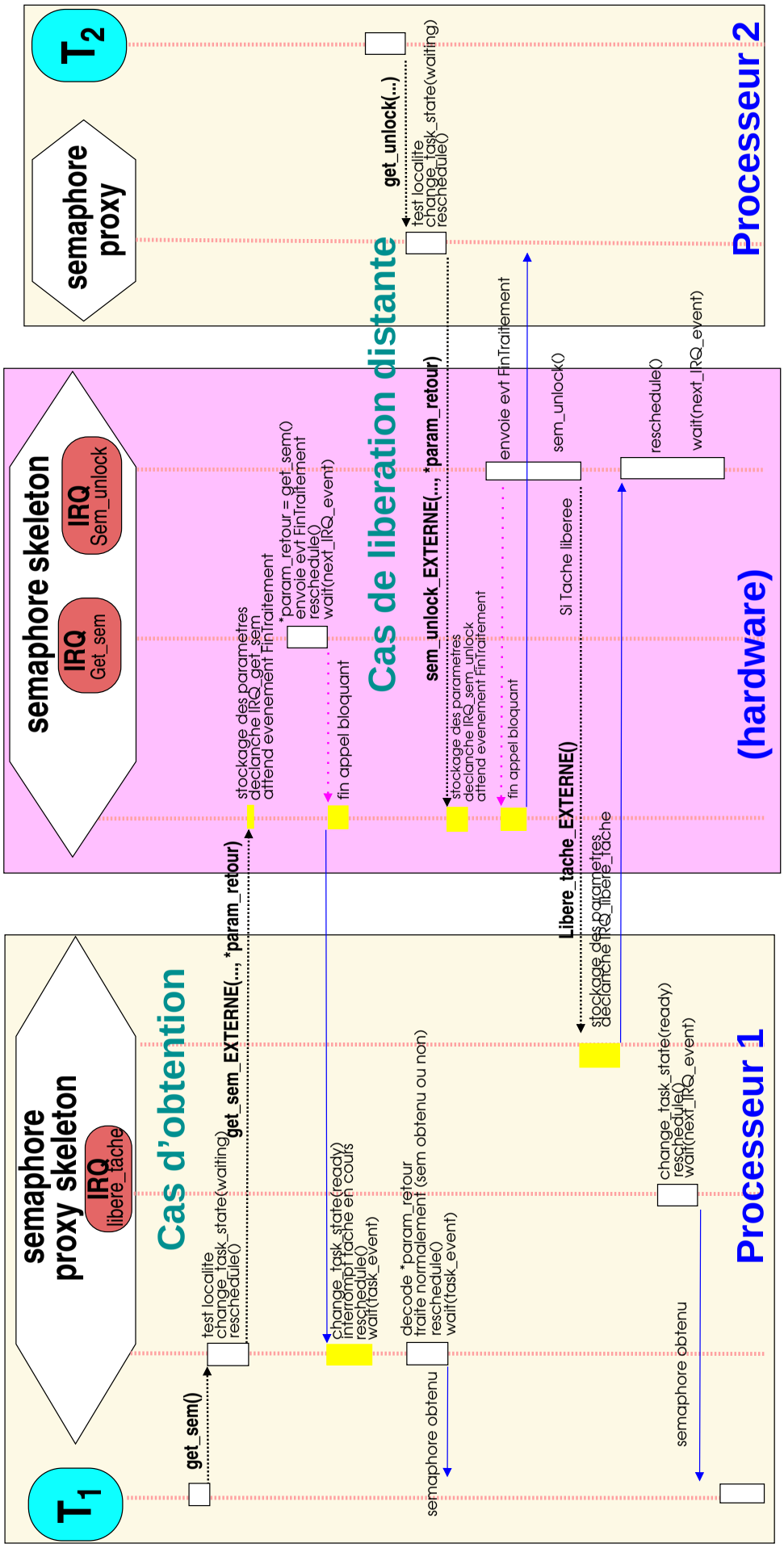


Figure 4.9: *Appels à un service partagé distant. Le service distant traite sous forme d'interruptions les requettes afin que le temps de simulation consommé soit bien localisé dans le bon nœud. Les appels étant bloquant, on libère le processeur en attendant la réponse d'un service. Ici sont représenté la demande de sémaphore de la tâche T1 qui est obtenu, ainsi que le cas de libération de ce sémaphore par une autre tâche par un autre nœud qui libèrere cette tâche T1*

4.4 Communications Génériques

Notre modèle à base d'agrégation de modules de services inter-connectés fonctionne et permet de simuler un OS et les tâches qui s'exécutent dessus. L'inconvénient de notre système est la difficulté d'ajouter / supprimer des modules de services simplement, à cause des nombreuses connexions à connecter manuellement entre les services et vers l'OS. Nous allons donc chercher à rendre ce système de communication inter-services plus générique.

4.4.1 Utilisation de TLM

De manière à simuler des systèmes complexes comme les SoC, il est nécessaire d'élever le niveau d'abstraction pour faciliter la tâche du concepteur mais surtout accélérer le simulateur de ce système. L'attitude actuelle consiste à modéliser le système au niveau transactionnel. On se réfère généralement au terme générique **TLM** qui signifie Transactional Level Model(ing) ou modélisation au niveau transactionnel (cf. sous-section 2.4.2.1). Dans le cas de SystemC, c'est le nom d'une norme qui consiste à formaliser une interface unique de communication entre modules SystemC (des blocs matériels), de manière à permettre l'interconnexion de modules hétérogènes de niveaux de précision différents. Il existe une première version (TLM 1.0) dont nous nous servons ici. La nouvelle version (TLM 2.0) est un peu plus contraignante car elle définie certains champs des données échangées (dans l'optique de modéliser des blocs matériels), et est donc plus complexe à mettre en place avec des paramètres ne nous servant aucunement pour notre modélisation de niveau *service* (SAT).

La norme TLM définie plusieurs méthodes permettant de faire des transactions. Nous avons choisi dans un premier temps d'utiliser la méthode `transport()`. La méthode `transport()` est fournie sous forme de **template**. Nous devons donc choisir le type unique de données échangées (un paquet pour la requête et un pour la réponse), de manière générique pour tous les services nécessaires (et même ceux encore non définis).

À partir de cette définition des paquets, il est maintenant nécessaire pour chaque module de créer les convertisseurs entre leurs *conveniente interface* (l'interface métier) et celle TLM. En fait cela consiste à faire implémenter par le module l'interface `transport()` (en plus ou à la place de son interface propre), et de créer un port particulier, charger de traduire automatiquement les appels aux méthodes en appels TLM (donc à la méthode transport). Le fait de créer un port spécifique permet aux modules qui veulent faire appel à cette interface de ne pas avoir à changer leurs appels. La traduction se fera automatiquement dès qu'un module utilisera ce port spécifique, sans avoir à se préoccuper de re-coder cette translation pour chaque module appelant.

Il faut ainsi produire ces ports spécifiques et interfaces pour chaque module. Cela implique de modifier en interne des modules les appels aux ports. Tout d'abord les ports sont maintenant des ports spécifiques. On remplace un port classique `sc_port <basic_gestiontache_iif> p_bgt_iif` par un `basic_gestiontache_initiator_port p_bgt_iif` qui est un de conversion entre protole TLM et métier conçu par nos soins. Et de ce fait, tous les appels doivent avoir la légère modification de changer le pointeur de port en objet port directement :

```
#ifndef USE_TLM
  t = p_sched_iif->get_currenttask();
#else
  t = p_sched_iif.get_currenttask();
#endif
```

Ces modifications permettent de n'avoir à présenter qu'une seule et unique interface aux autres modules et non plus une par couple d'entraide proxy/skeleton. Cela permet de

modifier rapidement l'interface d'un module de service sans avoir à changer les connexions externes s'y connectant. Mais surtout cela facilite les connexions depuis les modules demandeurs, n'ayant plus besoin (ou presque) de se préoccuper du type de l'interface du module avec lequel il se branche, étant donné qu'il n'y en a plus qu'une.

4.4.2 Utilisation du *Calling Abstraction Service* (CAS) interne

Malgré l'utilisation de TLM, le nombre de ports connectés depuis un module vers les modules fournisseurs de services internes reste inchangé. Si un module doit demander des services à trois autres, il doit avoir trois ports.

L'idée consiste à créer un objet unique dans l'OS de distribution des comunications auquel tous les modules de services se connecterons par un port unique. Il se chargera de rediriger une requête d'un service à un autre sans que ce dernier n'ai à se préoccuper de se connecter à ce dernier. Ce mécanisme s'inspire du principe des ORB de Corba. Nous nommerons CAS cet objet de gestion des communications/connexions internes pour *Calling Abstraction Service*.

Pour cela, comme inspiré de TLM, nous devons spécifier une interface unique permettant ces échanges de requêtes entre services.

Chaque composant doit pouvoir récupérer un paquet de requête par la méthode transport et être capable de répondre si ce module de service est capable de traiter la requête. Ce dernier permettra au CAS de router automatiquement les requêtes au bon module.

```
class oversoc_if : public virtual sc_interface {
public:
  virtual oversoc_rsp transport(oversoc_req *REQ)=0;
  // remplace a la fois les fonctions read et write
  virtual bool is_compatible(oversoc_req *REQ)=0;
  // est utilise du cote reception pour determiner la destination
};
```

Il faut maintenant définir un paquet générique capable de transporter n'importe quelle requête, avec un identifiant du service demandé et d'autres informations utiles au transport (virtuel). Idem pour le paquet de la réponse.

```
struct oversoc_req {
 int address_src;// num du noeud de l'emetteur
 int address_dest;// num du noeud destination
 int type_req;  // type de commande/requete
 void *mydata;  // donnee specifique au couple emetteur/recepteur
    // "type_req", deja definit en utilisant TLM
};
```

- Pour l'instant les adresses source et destination ne sont d'aucune utilité car les paquets restent dans l'OS. Le type de requête identifie de manière unique le service demandé.
- Pour identifier le type de requette, lors de la compilation on doit faire un inventaire exhaustif des services de tous les modules pour leur assigner un numéro unique de la même manière que cela est défini avec la méthode TLM.
- Le champ `mydata` sert de pointeur sur une structure propre à chaque service (paramètres de l'appel de la fonction du service).

La fonction `transport` renverra un paquet de type :

```
struct oversoc_rsp{
  int address_dest; // adresse du demandeur de service = emetteur
 int type_req;     // adresse du module faisant la requete
 void *mydata;     // donnee specifique
 oversoc_status status; // statut du retour
};
```

La fonction `transport()` implémentée dans un service esclave récupérera le paquet en paramètre, décodera son type en fonction de son genre (ie. `OSCREATE_SEM, RESCHEDULE` ...), fera l'action appropriée et retournera après un certain temps un paquet réponse contenant au moins le statut :

```
enum oversoc_status {
  OVERSOC_OK = 0, // paquet traite sans probleme
  OVERSOC_BUSY,   // destination indisponible
  OVERSOC_ERROR   // erreur d'envoi
};
```

La fonction de requête de service est considérée comme bloquante. C'est-à-dire que l'exécution du service appelé jusqu'à réception de la réponse n'est pas nécessairement instantanée même si l'acheminement par le canal CAS interne est lui instantané à un premier niveau.

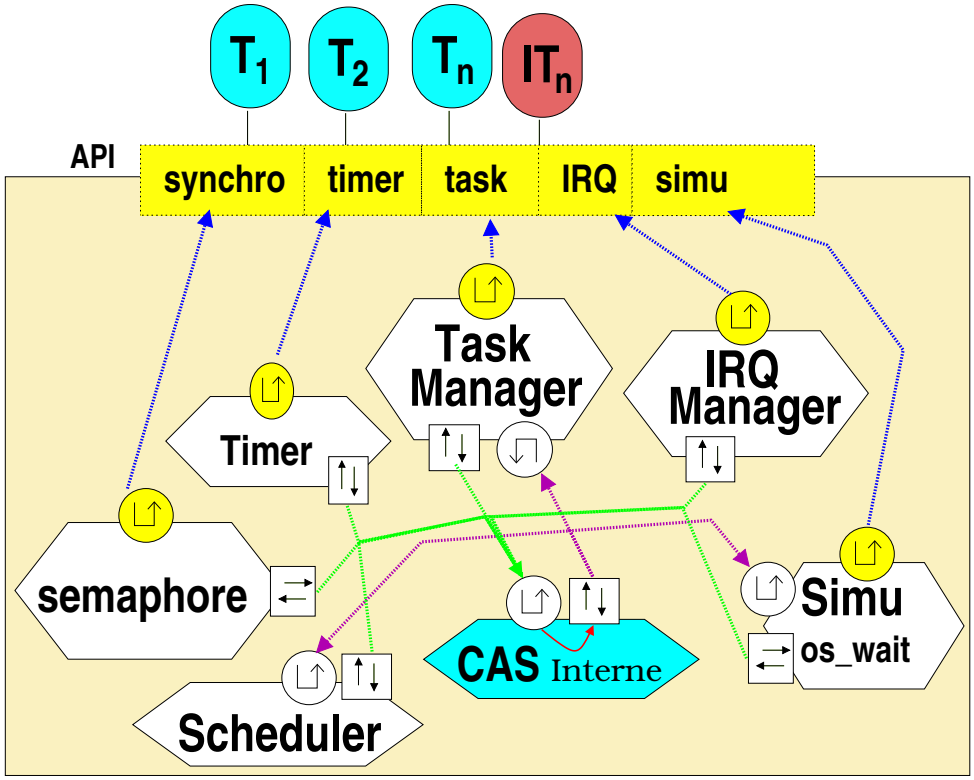


Figure 4.10: *Modèle d'OS avec le module interne de communication CAS permettant d'interconnecter simplement tous les modules d'un même OS*

La fonction `is_compatible` implémentée par chaque composant esclave (les modules de service proposant des services aux autres modules) extrait le champ `type_req` du paquet REQ et vérifie si sa valeur correspond à un service dont il est responsable et renvoie vrai le cas échéant. Cette fonction sert en interne au CAS de l'OS pour le routage vers le bon port connecté au service qui sait répondre à la requête.

Grâce à ce module CAS, les modules ont besoin d'une seule connexion pour s'interfacer avec tous les autres modules de services auquels ils font appel. De la même manière, les modules qui rendent des services aux autres doivent présenter leur interface interne uniquement au CAS, comme on peut le voir sur la figure 4.10. Cela simplifie de manière importante la première modélisation décrite dans la figure 3.10 page 69, en unifiant ports et interfaces.

4.4.2.1 Schéma simplifié du modèle d'OS distribué et du CAS

La vision générique du modèle d'OS tel que décrit dans le diagramme de classes UML 4.11 est la suivante : un OS est un `sc_module` composé de un ou plusieurs modules de services. Chacun de ces modules est aussi un `sc_channel` qui implémente deux interfaces. Une pour la partie de l'API de l'OS dont le module est responsable, une pour les services rendus aux autres modules. Pour cela, un module faisant appel aux services d'autres modules possède un unique port pour se connecter à ces derniers à travers le CAS. L'OS exporte toujours les interfaces externes de manière à présenter l'API de l'OS à l'application.

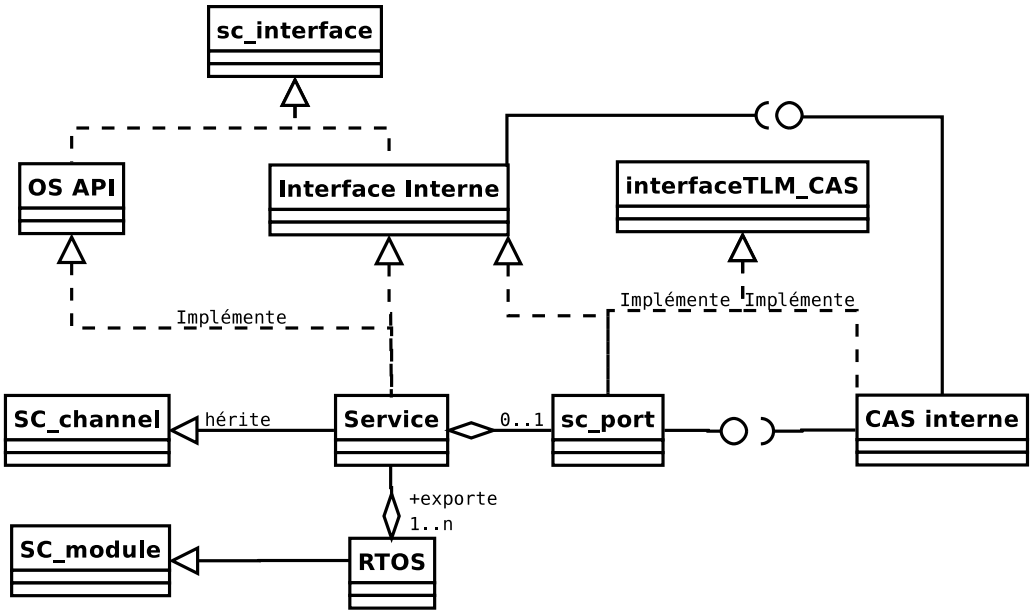


Figure 4.11: *Modèle d'OS avec services distribués en SystemC, diagramme de classes UML*

Il reste maintennant à s'intérresser à la généralisation des communications entre services pour permetre des communications distantes entre nœuds.

4.4.3 Connexion générique : le CAS de haut niveau

Nous avons défini un canal simple nommé Calling Abstraction Service (CAS) ; ce service est présent dans chaque OS en un seul exemplaire pour router les demandes entres les modules de services. Nous allons le modifier pour le généraliser au routage externe vers des modules distants pour être capable de communiquer entre les OS.

Ce CAS externe va maintenant servir à la recherche et au routage des demandes de services vers le bon module de service demandé que celui-ci soit local ou distant. On pourrait dire qu'il s'apparente à un ORB, sans découverte de service dynamique (la connexion de port ne peut se faire qu'en début de simulation en SystemC).

Pour cela, nous allons simplement modifier les paquets transmis par le CAS de base afin d'y adjoindre les informations de localité, à savoir les adresses de la source et du destinataire des requêtes, ce qui dans notre cas équivaut au numéro d'OS ou du nœud.

Nous allons créer un CAS particulier, le CAS externe, qui sera le pont de connexion entre les OS, plus précisément, entre les CAS interne de chaque OS. Ce dernier regardera donc les adresses et demandera au CAS interne de l'OS désigné si la fonction demandée est bien implémentée dans cet OS et dans ce cas lui transmettra la requête, sinon renverra un message d'erreur en retour. Autrement dit, la seule différence notable entre le CAS interne et le CAS externe, c'est que le CAS externe route selon les adresses des OS de

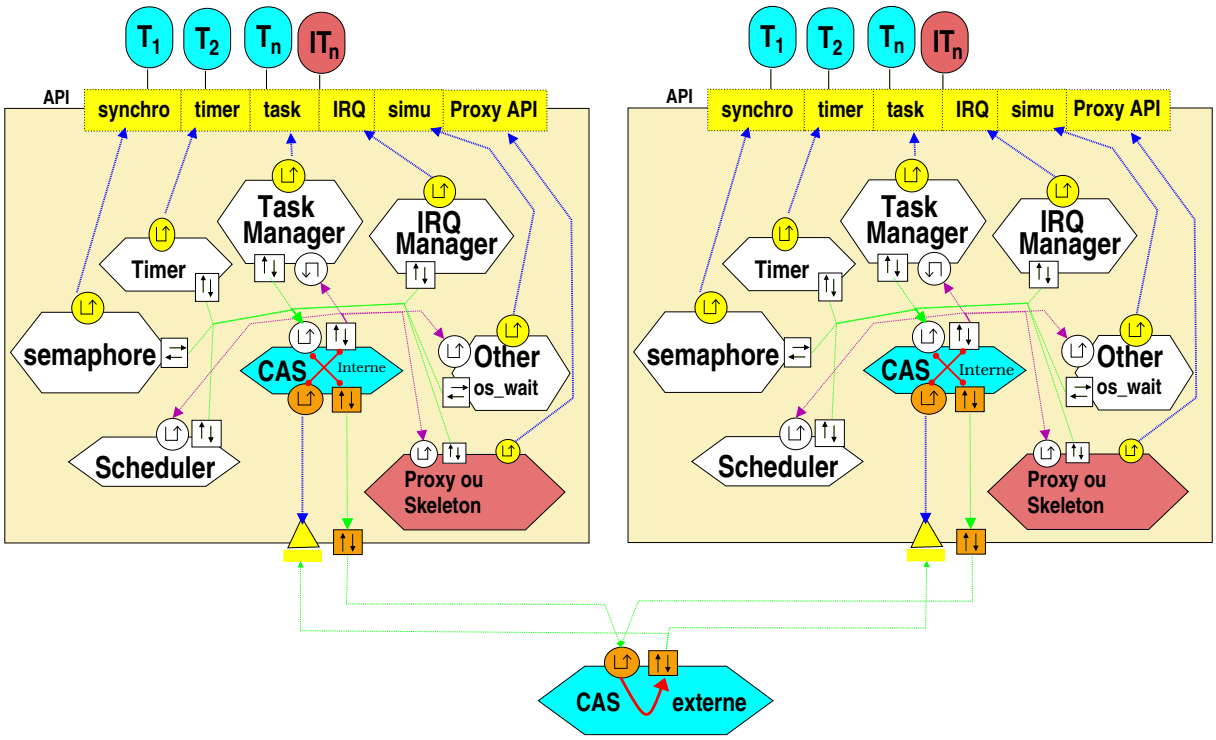


Figure 4.12: *Multi-OS avec service partagé distant connecté par le CAS externe faisant office de crossbar intégral*

destination de la requête, alors que le CAS interne route vers le bon module interne selon le type de la requête.

Le CAS externe a besoin de connaître la localisation des OS pour router les demandes vers la bonne connexion de sortie. Aussi, l'interface présentée par les CAS internes connectés au CAS externe répond aussi à la méthode `is_compatible` par vrai ou faux selon qu'il a le bon numéro d'OS et en plus possède un module qui répond au type de la requête. Ce qui correspond d'une certaine manière à un décodage d'adresse partie haute pour le numéro d'OS et basse pour le type de la requête.

La figure 4.12 représente la connexion d'un module proxy à un module skeleton distant au travers d'une connexion relayée par le CAS interne vers le CAS externe qui route la requête vers le bon nœud / OS dans lequel se trouve le module skeleton.

On peut ausi envisager des améliorations possibles au système des CAS. Pour éviter d'utiliser mainte fois la fonction `is_compatible` dans les CAS pour constamment rechercher dynamiquement l'OS où le service est implémenté, on pourrait créer dynamiquement une table de routage dans les CAS externes et internes afin d'accélérer la simulation, car la création dyanamique de nouveau services après la phase d'initiation de la simulation n'est pas possible.

4.4.4 Raffinement du *CAS* externe

Éventuellement, il pourrait être intéressant d'ajouter dans la couche architecture (cf. séparation des préoccupations de la figure 3.1) des fonctions de temps représentant le temps de traversée des bus ainsi modélisés plus précisément. Notamment pour gérer les congestions ou améliorer la qualité de service du système, ou tout simplement affiner les temps de lectures et d'écritures des données. Ce genre de travaux de modélisation fine de la partie communication est déjà développé dans des projets de recherche sur les Bus et NoC, et nous verrons dans la partie de déploiement de notre modèle dans le simulateur de Ter@ops, que nous pouvons remplacer le CAS par un modèle de NoC précis en SystemC

OCP.

4.5 Exploration d'architecture MPSoC

4.5.1 Distribution et exploration matérielle d'un service de synchronisation partagé

La parallélisation d'une application implique généralement de découper en sous tâches l'algorithme initial et d'ajouter des primitives de synchronisation entre les sous-tâches, afin de conserver le comportement initial souhaité. Le point critique de la parallélisation est de garantir l'exclusion mutuelle d'accès aux données qui peuvent être utilisées en même temps du fait du découpage temporel potentiellement concurrent. Nous avons donc envisagé de construire un service partagé de sémaphore qui permettra de remplir des fonctions de synchronisation et d'exclusion mutuelle.

Nous avons fait l'hypothèse que ce service sera raffiné sous forme d'un bloc matériel dédié que tous les nœuds d'exécution pourront atteindre et qui garanti un fonctionnement cohérent avec les caractéristiques attendues d'un service de sémaphore.

Nous avons donc développé un module VHDL qui permet de mémoriser la valeur d'un sémaphore, renvoyer l'état courant prédécrémenté et qui peut ensuite enregistrer le numéro de l'OS demandeur (ainsi que le PID de la tâche), afin que lors de la libération du sémaphore, le bloc matériel puisse indiquer au bon nœud OS qu'un sémaphore est libéré. Les détails de ces travaux sont décrit en annexe A page 143.

Nous avons utilisé les résultats de la synthèse et les mesures d'échanges de données sur le bus AMBA afin de reporter cela dans un modèle de ce bloc de service matériel pour pouvoir l'utiliser avec notre simulateur. Ainsi, on dispose d'un service partagé localisé sur un nœud dédié permettant d'envisager l'exploration de systèmes multi-processeurs capables de se synchroniser entre eux, autrement que par services logiciels. Ce genre de service peut servir à établir un système évènementiel de communication plus complexe par mémoire partagée ou autoriser un moyen de synchronisation entre tâches logicielles et matérielles (IP sur zone reconfigurables par exemple).

4.5.2 Modélisation et distribution d'une application de vision robotique

Cette partie présente les derniers résultats obtenus en utilisant le modèle de RTOS présenté dans les sections précédentes pour la modélisation d'une architecture distribuée multi-processeurs/nœuds pour l'application de vision.

En utilisant l'avantage de la modularité et de la généricité du modèle de RTOS SystemC, nous avons développé un modèle multi-RTOS. A ce niveau d'abstraction, chaque élément de calcul est représentée par un modèle de RTOS unique. Chaque RTOS est responsable de l'exécution de sa partie de l'application (nous supposons que le partitionnement de l'application est fait au préalable). L'application est donc présentée sous la forme d'une tâche principale par nœud d'exécution. Ces tâches *main* sont responsables de la création de toutes les tâches de l'application et des traitants d'interruption locaux. Chaque RTOS est également complété par un port de communication par lequel il peut communiquer avec le monde extérieur.

Un service commun partagé à travers le CAS externe est nécessaire pour assurer la communication et la synchronisation entre les tâches exécutées sur les différents nœuds. Un module de sémaphore partagé a été modélisé par l'ajout d'un modèle de RTOS d'appoint comportant un seul module de service, celui de sémaphore, s'exécutant sur un nœud de

calcul dédié. Ce module sera raffiné soit comme un seul processeur qui exécute un service de sémaphore dédié soit par un bloc matériel de synchronisation inter-processeurs. La figure 4.13 montre comment de multiples modèles de RTOS peuvent interagir en utilisant ce module partagé. Chaque module de service local de sémaphore a été remplacé par un module proxy qui relaie à travers le réseau de CAS internes et externe la requette vers le bon module de service skeleton.

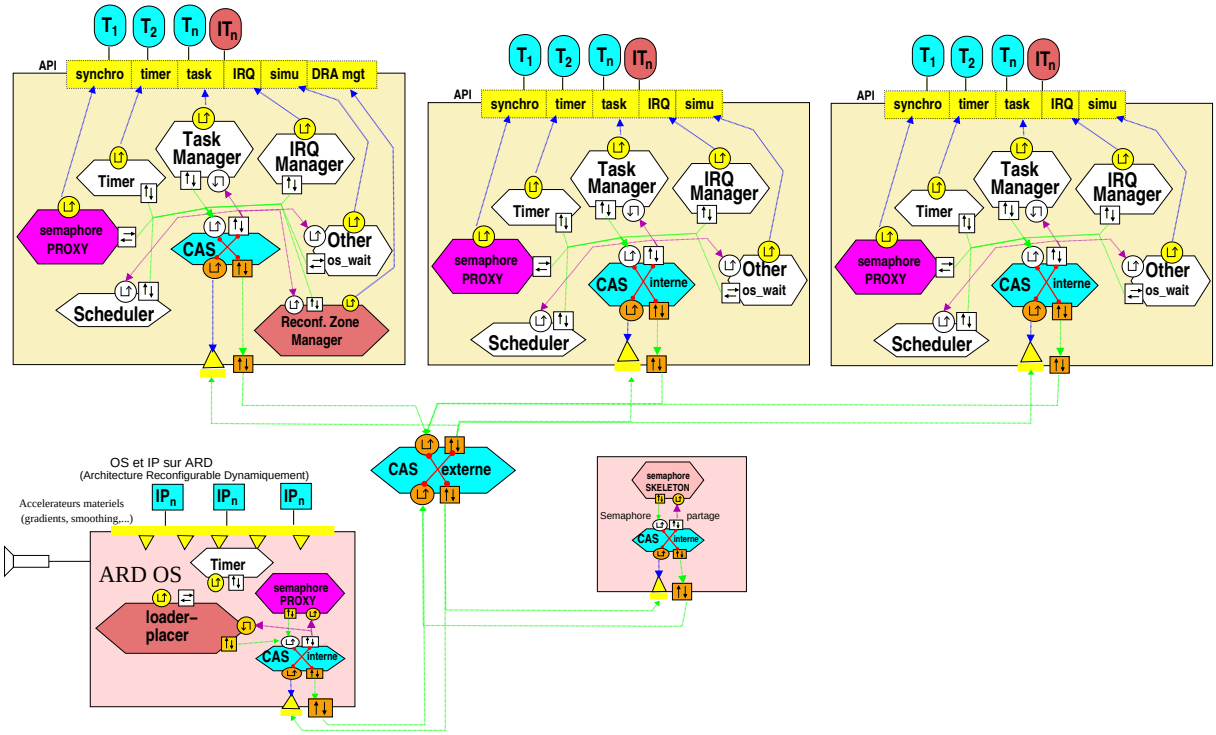


Figure 4.13: *Plateforme MPSoC modélisée comme une collection de modèles d'OS interagissants*

4.5.3 Exploration de son architecture logicielle

Basé sur le flot de conception présenté, nous utilisons notre framework de modélisation pour explorer l'architecture de l'application de vision. Tel que décrit dans la section 3.6.2 nous avons établi le profil de l'ensemble de l'application sur une plate-forme embarquée. Nous avons également construit le profil des services temps-réel de μC/OS-II (déterministe). Les données de synchronisation ont été mesurées et rétro-annotées dans le modèle de haut niveau afin d'explorer et d'évaluer le dimensionnement d'architecture et les stratégies de mise en œuvre : répartition des tâches, des services de distribution, les algorithmes d'ordonnancement ...

Comme illustré dans le tableau 4.1 les mesures sur carte permettent de déterminer les parties critiques de l'application. En mode haut niveau de détail, le gradient et les filtres HF gaussiens représentent plus de 80 % du temps d'exécution total. En outre, pour l'implémentation logicielle, tous les traitements réalisés sur les hautes fréquences, sauf l'extraction, sont très critiques et dépassent même les contraintes temps-réel du mode de détail élevé de 1000 ms. Plus généralement, l'implémentation logicielle de la tâche gradient commune au trois modes est incompatible avec l'un des trois comportements temps-réel identifiés.

Pour ces raisons, nous avons utilisé la possibilité de distribution du modèle d'OS pour évaluer le gain de la parallélisation sur les temps d'exécution dans les trois modes identifiés.

Table 4.1: *Profil logiciel des tâches significatives de l'application : moyenne des temps d'exécution pour 20 images représentatives*

Tâche	temps d'exécution moyen (ms)	Pourcentage
Gradients	13915.4	49%
filtre HF Gaussien	9795.8	35%
filtre LF Gaussien	426.0	1.5%
HF DoGs	443.0	1.6%
LF DoGs	15.3	0.05%
HF recherche	1286.1	4.6%
HF extraction (+ norm.)	58.8	0.21%
LF recherche	45.1	0.16%
LF extraction (+ norm.)	33.1	0.12%
autre (tâche MF, échantillonnage)	2238.2	7.9%
Total	28257.2	100 %

La figure 4.14 montre le gain potentiel en utilisant plusieurs processeurs (de 2 à 5) pour le mode de détail élevé.

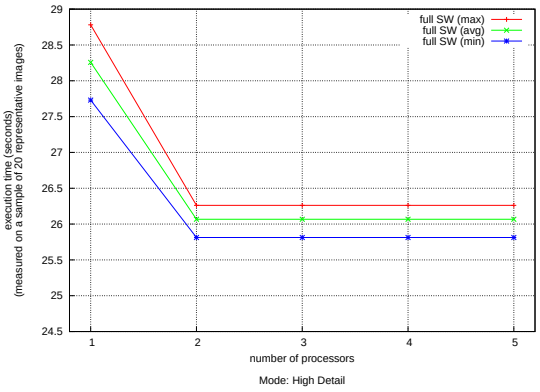


Figure 4.14: *Temps d'exécution selon le nombre de processeurs, dans une implémentation logicielle pure.*

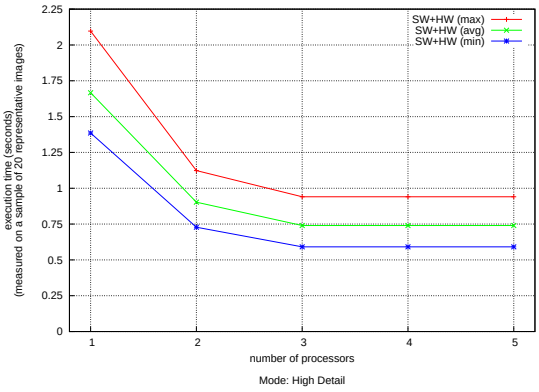


Figure 4.15: *Temps d'exécution selon le nombre de processeurs, dans une implémentation mixte matériel/logiciel.*

Mais la parallélisation n'a pas d'effet significatif au-delà de 2 processeurs. En effet, la concurrence dans l'application apparaît seulement entre la pyramide gaussienne séquentielle et les différents échelles (tâches recherches et extractions), et entre les tâches de normalisation à l'intérieur de chaque échelle, ces derniers ne représentant qu'un faible pourcentage du temps total de l'application logicielle. En outre, comme illustré dans la table 4.1 la différence de complexité entre les échelles explique le gain non significatif pour la parallélisation des tâches de normalisation sur un troisième processeur.
Aussi seule une mise en œuvre en matériel et logiciel pourrait alors respecter nos contraintes variables en accélérant la pyramide de gaussiennes.

4.5.4 Exploration avec un mapping matériel

Selon les résultats de la première phase d'exploration, les traitements critiques et réguliers identifiés (gradient, HF filtres gaussien, et les DoG) sont des candidats à implémentation matérielle fixe sous forme d'accélérateurs dédiés. Ces résultats amènent à un premier processus de raffinement. En effet, afin d'évaluer l'accélération, Thomas Lefebvre, en thèse au laboratoire ETIS, a développé une description VHDL des tâches les plus gourmandes en calculs (la pyramide de filtres). Les caractéristiques temporelles rapportées par l'outil de synthèse matérielle (Altera Quartus II pour notre exemple) ont été intégrées dans le modèle. Les tâches matérielles correspondantes ont été modélisées comme des tâches SystemC indépendantes et simultanées avec des temps d'exécution retro-annotés avec ces nouvelles mesures. En outre, chaque bloc matériel fournit une ligne d'interruption pour la synchronisation avec la partie logicielle lorsque les données sont produites. Du partitionnement logiciel/matériel identifié nous avons mené une deuxième série d'expérimentation qui a ammené les évaluations de performance décrites dans la figure 4.15 (la figure ne représente que les résultats pour le mode de détail élevé). En comparant les figures 4.14 et 4.15 nous avons trouvé une augmentation de la vitesse jusqu'à un facteur x17 grâce à l'accélération matérielle. La mise en œuvre de la pyramide en matériel induit aussi des possibilités de parallélisation plus importantes sur le troisième processeur (mais pas au delà).

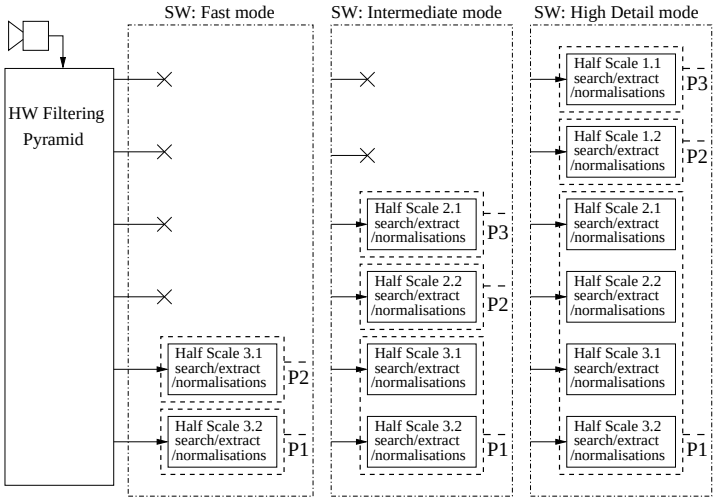


Figure 4.16: *Architectures logicielle pour les trois modes, avec accélération matétiels de certaines tâches.*

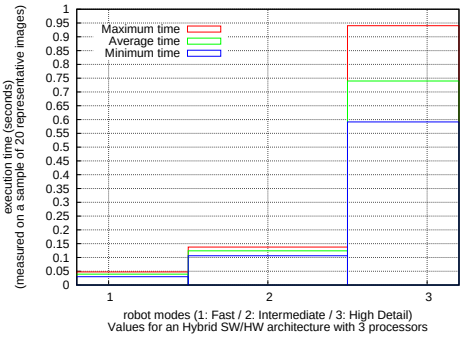


Figure 4.17: *Comparaison des temps d'exécution pour les différents modes sur l'architecture hybride (accélérateurs matériels + multiprocesseurs logiciels).*

Mais le second défi de l'exploration a été de trouver une architecture respectant les contraintes correspondant aux trois modes d'application. Plus précisément chaque mode exécute différents traitements en vertu des taux de trames différents. La variation des contraintes conduit finalement à trois architectures embarquées qui sont présentés dans la figure 4.16. En mode rapide, les échelles inférieures sont mises en œuvre sur deux processeurs. Si deux ou plusieurs processeurs sont utilisés, le pire temps d'exécution d'application est d'environ 48 ms, correspondant ainsi au robot rapide. En mode intermédiaire, 3 transformateurs sont nécessaires menant à un temps d'exécution total de moins de 150 ms. Dans ce mode, puisque la contrainte de période est assouplie les deux demi-échelles inférieures peuvent être mises en œuvre sur le même processeur. Enfin, en mode haut niveau de détail, les tâches de recherche et d'extraction des deux demi-échelles de haute fréquence

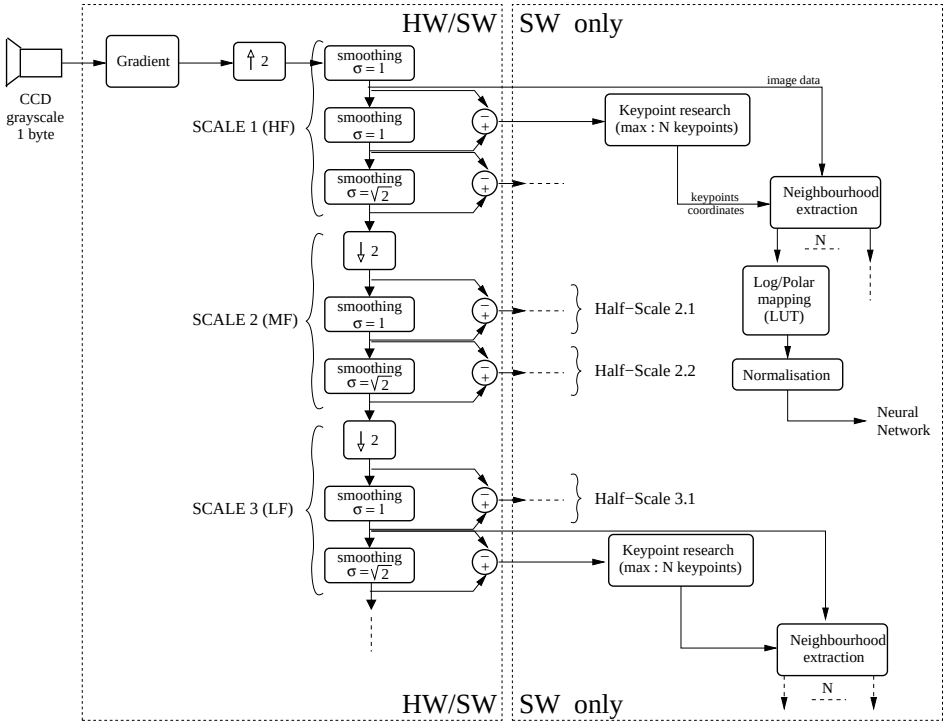


Figure 4.18: *Architecture retenue pour partitionner en matériel une partie des tâches*

sont traitées sur des processeurs distincts alors que les fréquences basses et moyennes sont exécutées sur un seul processeur. Le temps d'exécution total maximum dans ce mode est de 950 ms sur nos trois processeurs modélisés. Dans les trois cas, l'architecture contient au moins deux éléments communs : l'accélérateur matériel et un processeur exécutant au moins les tâches de basse fréquence. Les résultats de performance obtenus avec la solution mixte logicielle/matérielle sont résumés dans la figure 4.17. Le meilleur partitionnement logiciel parmi ceux explorés est représenté sur les figures 4.16 et 4.18.

Le système final répond à toutes les exigences de l'application dans chaque mode.

Conclusion de l'exploration de l'application de vision

En conclusion, nous avons modélisé avec succès une plate-forme réaliste multi-processeurs avec notre modèle de système d'exploitation distribué. Grâce aux propriétés du modèle, nous avons exploré et défini une architecture adaptée aux exigences dynamiques de l'application. La plate-forme cible considérée est un SoPC (System on Programmable Chip) de Altera [Alt07].

Avec ces conclusions un FPGA est configuré avec 3 microprocesseurs RISC (Nios II) et les blocs matériels sélectionnés (accélérateurs et sémaphore). Chaque processeur exécutera une instance de RTOS personnalisée raffinée à partir du modèle de système d'exploitation. La synchronisation inter-processeurs sera réalisée par le service sémaphore matériel partagé. L'application entière peut donc être mise en œuvre sur une seule puce (SoC).

Afin d'optimiser les changement de mode on pourrait s'intéresser au raffinement du système d'exploitation afin de supporter de nouveaux services spécifiques tels que la migration des logiciels en ligne et l'ordonnancement DVS [1] puisque seuls deux des trois processeurs sont utiles dans le mode rapide. Ces mécanismes seraient gérés par le système d'exploitation dédié et distribué. Une autre perspective intéressante serai la gestion du

1. Dynamic Voltage Scaling

matériel reconfigurable dynamiquement afin de proposer une architecture matérielle qui s'adapte au mode de fonctionnement, configurant le nombre de processeur au fur et à mesure des besoins.

4.5.5 Évaluation du surcoût de simulation du modèle d'OS

Nous avons évalué le temps de simulation de l'application sur notre modèle de RTOS en comparaison avec une description purement fonctionnelle. Nous avons fait varier le nombre de nœuds d'exécution de l'architecture de 1 à 6 OS (et processeurs) lors de l'exploration de la distribution de l'application de vision.

Les tâches s'exécutent et communiquent de la même manière sur la carte de prototypage et dans la simulation à travers un espace de mémoire partagée unique protégée par les sémaphores partagés. Le tableau 4.2 montre la scalabilité de notre modèle. Il indique le temps de simulation t_n d'une plate-forme composée de n RTOS et le surcoût moyen de temps de simulation $s_n = \frac{t_n - t_1}{t_1}$ pour des tailles de plate-forme différente. t_0 représente la durée d'exécution de la spécification de l'application fonctionnelle en langage C pur. Les simulations ont été réalisées sur une station de travail Intel Dual-Core à 1,66 GHz avec 2 Go de RAM.

n	0 (appli fonctionnelle)	1	2	3	4	5	6
Temps de simulation t_n (secondes)	5.5	6	7.4	8.6	9.8	11.1	12.8
Overhead s_n (%)	-8.9	0	23.3	43.3	63.3	85	113.3

Table 4.2: *Surcoût de simulation du/des modèle(s) d'OS(s) selon leur nombre pour l'exécution d'une même application*

Pour les plateformes mono-processeur, le modèle de RTOS n'impacte pas beaucoup le temps de simulation : la surcharge n'est que de 8,9 % de plus que l'application purement fonctionnelle en C. Les résultats indiquent que le surcoût de durée de simulation est d'environ 23 % de plus par RTOS de plus simulé. Cette surcharge est due essentiellement au noyau de simulation SystemC qui parcourt la liste complète des `SC_THREAD` SystemC du système simulé, qui augmente automatiquement avec le nombre de RTOS.

4.6 Conclusion du chapitre

Afin de permettre une exploration de système d'exploitation et d'architecture distribué, nous avons modifié le modèle pour autoriser les communications entre services d'OS répartis sur de multiples nœuds d'éxécution matériels ou logiciels ou sont localisés les OS du système.

Avec ce modèle simple et générique de routage de requêtes entre les modules et entre les nœuds d'exécution, on peut rapidement définir un système multiprocesseurs reconfigurable. Ce modèle est suffisant pour permettre d'évaluer la distribution et l'exécution de tâches sur différents nœuds, y compris la simulation à haut niveau de tâches matérielles localisées sur un nœud/OS matériel comme on l'a exposé en explorant l'architecture de l'application de vision.

Maintenant que nous avons un système permettant le prototypage rapide par simulation conjointe logicielle/matérielle de MPSoC, nous allons nous intéresser à la simulation à différents niveaux de précision de certains éléments, que ce soit le code des tâches logicielles (avec un ISS), des tâches matérielles (projet OveRSoC modélisant plus finement les

zones reconfigurables), ou les communications et les échanges mémoire (projet Ter@ops). Il nous faut par exemple pouvoir raffiner le CAS externe de manière à obtenir des informations de temps de communications entre nœuds plus précises, essentielles à l'évaluation des flux de données dans les systèmes multi-coeurs. Ce type de résultat plus fin sera obtenu dans les expérimentations et les intégrations de notre modèle effectuées dans le cadre des projets Ter@ops et OveRSoC que nous allons décrire au chapitre suivant.

Chapitre 5

Intégration et validation du modèle dans deux projets de collaboration

Sommaire

5.1 Introduction

Dans ce chapitre nous allons décrire d'autres expérimentations faites pour valider notre modèle de gestionnaire de plateforme. Le but de notre modèle est de permettre l'exploration conjointe logiciel/matériel à haut niveau de plateformes flexibles (reconfigurables) pour des applications dynamiques, et nous l'avons déployé au sein de deux projets qui nécessitaient aussi la modélisation conjointe du logiciel et du système d'exploitation pour fonctionner.

Une première section présente l'intégration de ce modèle et de ses extensions dans le cadre du projet ANR OveRSoC, ayant pour objectif l'exploration et la validation de systèmes reconfigurables. Pour cela, une méthodoligie d'exploration a été élaborée, et a donné lieu à la conception d'un outil pour faciliter cette exploration. Une approche de raffinement sous forme d'ISS, ainsi que l'ajout d'un modèle de zone reconfigurable on été intégrés pour valider le projet.

La deuxième section présente l'intégration du modèle au sein du projet Ter@OPS cherchant à modéliser et simuler en SystemC un SoC hétérogène expérimental devant fournir

une puissance de calcul d'un Tera opérations par seconde pour une large gamme d'applications issues de domaines différents. Le modèle d'OS y joue un rôle central pour simuler et gérer la partie applicative qui s'exécute sur les multiples nœuds/tuiles du système, et a du être adapté pour pourvoir s'intégrer dans le modèle raffiné des communications distantes du NoC, ainsi que pour modéliser finement les échanges mémoires.

5.2 Travaux dans le projet OveRSoC

5.2.1 Objectifs du projet OveRSoC

Pour obtenir la puissance de calcul nécessaire aux algorithmes complexes, les concepteurs de circuits actuels s'orientent vers des solutions hétérogènes (logicielles et matérielles) basées sur des unités de calcul travaillant en parallèle et garantissant le maximum de performances.

Des lors que la technologie le permet, certains blocs matériels peuvent également être reconfigurés, et ce dynamiquement, au sein du circuit. Ces Architectures Reconfigurables Dynamiquement (ARD) autoriseront l'exécution de plusieurs fonctionnalités matérielles au cours du temps. Cette flexibilité permet entre autres de pouvoir s'adapter efficacement à un contexte applicatif de plus en plus dynamique.

En contrepartie, l'utilisation de ces ARD pose un problème en termes de contrôle et de gestion. Aussi, il devient de plus en plus commun d'envisager l'utilisation de systèmes d'exploitation temps-réel dont le but est de fournir des services permettant de gérer les communications, la mémoire, l'exécution des tâches, leur séquencement, etc. Dans ce contexte, les blocs chargés dynamiquement dans les ARD peuvent être vus comme des tâches matérielles pouvant communiquer, entre elles ou avec des tâches logicielles.

Pour gérer cette mixité de tâches logicielles et matérielles, il est nécessaire d'utiliser des systèmes d'exploitation temps-réel Dans la littérature on retrouve deux approches. La première consiste à utiliser des OS déjà existants (RTAI, RTLinux, VxWorks, etc.) et de modifier leur fonctionnalité afin qu'ils puissent s'adapter à la gestion d'architectures reconfigurables [NCV+03]. La seconde est de construire un OS spécifique en y ajoutant des services dédiés à la gestion du reconfigurable [SWP04], [UHGB04].

Quelle que soit l'approche suivie, il est relativement facile de comprendre que la tâche consistant à développer des services spécifiques n'est pas aisée et qu'elle requiert beaucoup de temps. Quelques travaux recents ont proposé de s'intéresser au sujet de manière à modéliser à haut niveau l'ensemble de la plate-forme. Dans [ASA+08] ou [LP07], les auteurs proposent un modèle de programmation permettant la gestion du reconfigurable à partir d'un OS. Néanmoins, ces modèles sont très spécifiques et ne permettent pas de modéliser différents types d'OS.

Le projet OveRSoC s'inscrit en filigrane dans les travaux de cette thèse. OveRSoC signifie Outil pour la vérification et l'exploration de Systèmes sur puce Reconfigurables (RSoC). Le but du projet OveRSoC était donc de développer un outil d'exploration générique permettant d'aider le concepteur dans la description de sa plateforme, de manière à pouvoir explorer des choix critiques de conception portant notamment sur le système d'exploitation et particulièrement sur ceux de la gestion de zones reconfigurables (distribution des services, services dédiés à la gestion du reconfigurable, etc.). Cette description est réalisée à haut niveau d'abstraction de manière à pouvoir valider ou remettre en cause des choix le plus tôt possible dans le flot de conception.

Nous allons tout d'abord présenter la méthodologie ainsi que l'outil réalisé. Puis nous developperons des aspects de raffinement du modèle (ISS et ARD fin) avant de les mettre en pratique sur le cas de notre application de vision robotique.

5.2.2 Méthodologie OveRSoC de conception et d'exploration globale

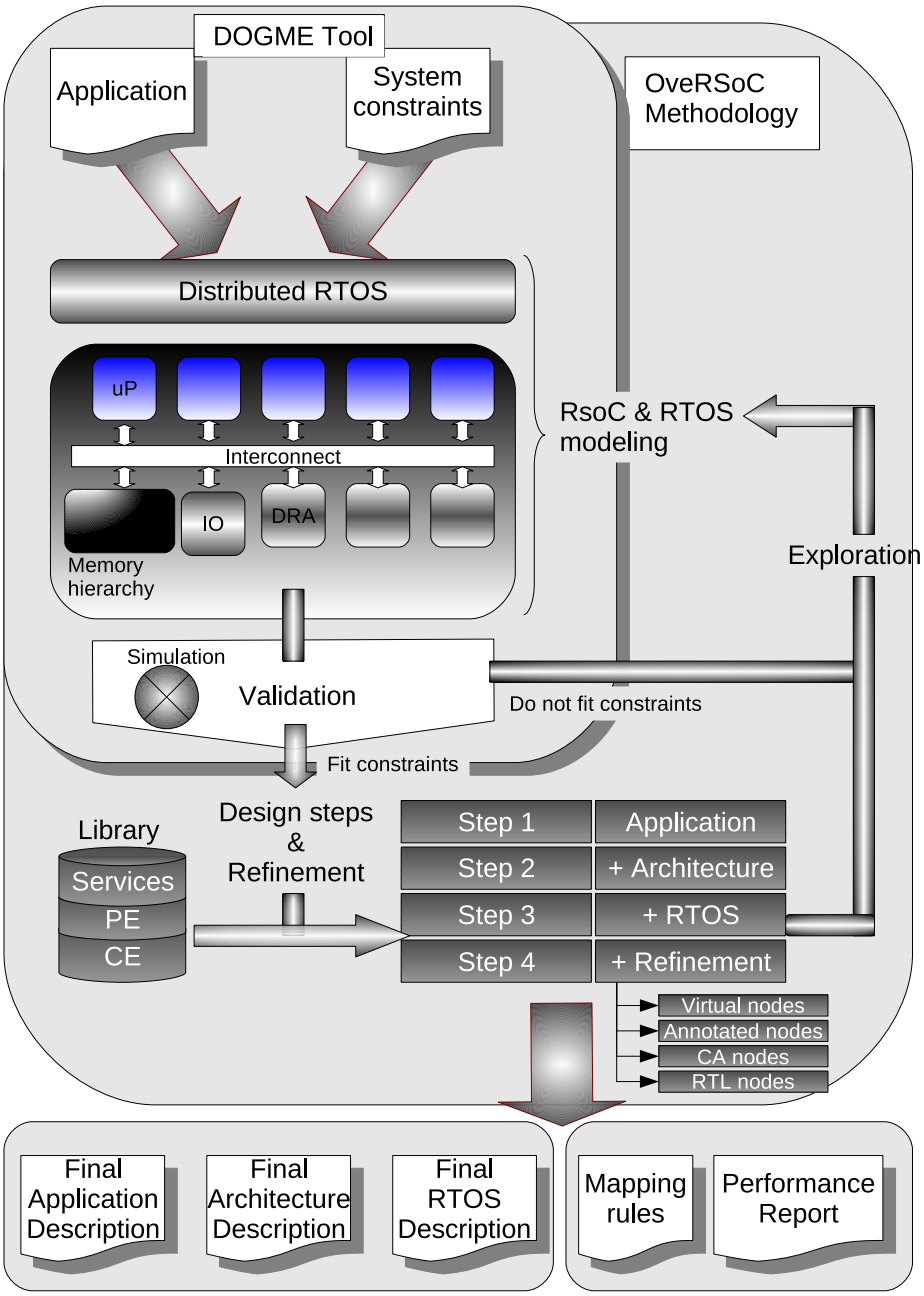


Figure 5.1: *Le flot d'exploration et de raffinement OveRSoC. L'exploration est définie comme un processus itératif : la modélisation, la simulation et la validation et l'exploration. Les entrées de la méthode sont les spécifications de l'application en code C fonctionnel pur, et les contraintes du système. Une fois le système validé, le processus de conception recommence à nouveau à un niveau inférieur d'abstraction jusqu'à la description du système final. A chaque niveau d'abstraction, l'objectif de l'exploration dépend du paradigme de la séparation des préoccupations (section 3.2.3). Ce paradigme est défini comme un processus en 4 étapes où les préoccupations suivantes sont successivement abordés : spécification, définition du RTOS, choix de l'architecture, et raffinement de la plateforme.*

La méthodologie OveRSoC a pour objectif de définir les services appropriés d'un RTOS pour la gestion des ARD. Les entrées du flot d'exploration consistent à spécifier non seule-

ment l'application mais également la plateforme sur laquelle cette application va être exécutée. L'application est décrite comme un ensemble de tâches logicielles ou matérielles, ces dernières pouvant être mises en œuvre dynamiquement sur une ou plusieurs architectures reconfigurables. La spécification comportementale des tâches restera du langage C/C++ pur fonctionnel, quel que soit leur implémentation finale (logiciel ou matériel). La communication entre ces tâches est également définie sous la forme d'un graphe de connexions.

La plateforme RSoC est, quant à elle, composée de trois composants principaux : l'OS dont le but est de gérer l'ensemble de la plateforme, les unités reconfigurables, et le processeur servant de support d'exécution au système d'exploitation. Ces trois composants sont modélisés à haut niveau d'abstraction à l'aide du langage SystemC. Le choix d'un tel niveau d'abstraction permet de disposer d'un modèle pouvant être simulé facilement et surtout rapidement de manière à pouvoir tester différentes configurations et disposer de résultats très tôt dans le flot de conception. Les composants modélisant la plateforme sont décrits à l'aide d'une liste d'attributs permettant de spécifier leur comportement.

A titre d'exemple, différents types d'ordonnanceurs peuvent être considérés en changeant simplement les attributs correspondants dans un fichier de description. Parallèlement, le concepteur a la possibilité de spécifier un certain nombre de contraintes relatives à son application (latence max. d'une tâche, capacité mémoire restreinte, etc.). Après avoir défini les attributs et les contraintes, l'ensemble de la plate-forme est simulé en SystemC conduisant à la génération de métriques qui seront ensuite affichées par l'outil et mises à disposition du concepteur. Le concepteur pourra alors analyser manuellement ces métriques afin de remettre en cause certains choix de conception ou de s'assurer de la portabilité de l'application sur la plate-forme considérée. Ce processus est itératif (figure 5.1) dans la mesure où le concepteur pourra modifier les attributs qui semblent poser problème puis simuler une nouvelle fois l'ensemble de la plate-forme. Le modèle de simulation est conçu de manière hiérarchique pour permettre d'affiner progressivement la description de celui-ci, permettant au concepteur de disposer de métriques de plus en plus précises.

Méthodologie de conception et d'exploration globale

Afin de concevoir une plate-forme RSoC, nous devons suivre une méthodologie dédiée qui comprend l'exploration et la validation du RTOS. Nous proposons donc une exploration globale avec un flot de conception basé sur des étapes successives de raffinement à la fois de l'architecture et de la mise en œuvre des stratégies RTOS. Ce flot est en partie inspiré par les travaux de Gajski *et. al.* [GZGH00] et Jerraya *et. al.* [JW04] et est illustré dans la figure 5.2. Une approche de modélisation en trois étapes (niveaux spécification, architecture et implémentation) a été proposé dans [GYG03]. Le principal inconvénient de cette proposition est qu'il existe toujours un écart entre la conception du modèle architectural (celui présentant le modèle d'OS) et le modèle final d'exécution (avec des processeurs modélisés comme des simulateurs de jeu d'instructions). Pour combler cette lacune, nous proposons un modèle d'architecture distribuée intermédiaire où le parallélisme peut être exploré. Le modèle de RTOS de haut niveau présenté dans cette thèse est utilisé à la fois dans la deuxième et troisième étapes de ce flot.

La méthodologie proposée consiste en quatre niveaux successifs de raffinement de la modélisation (modèles de spécification, d'exécution, de distribution et d'implémentation) décrites ci-dessous :

5.2.2.1 Modèle de spécification

Ce modèle est écrit comme un pur modèle SystemC de l'application où chaque tâche applicative est un thread ou un process SystemC et où les communications inter-tâches

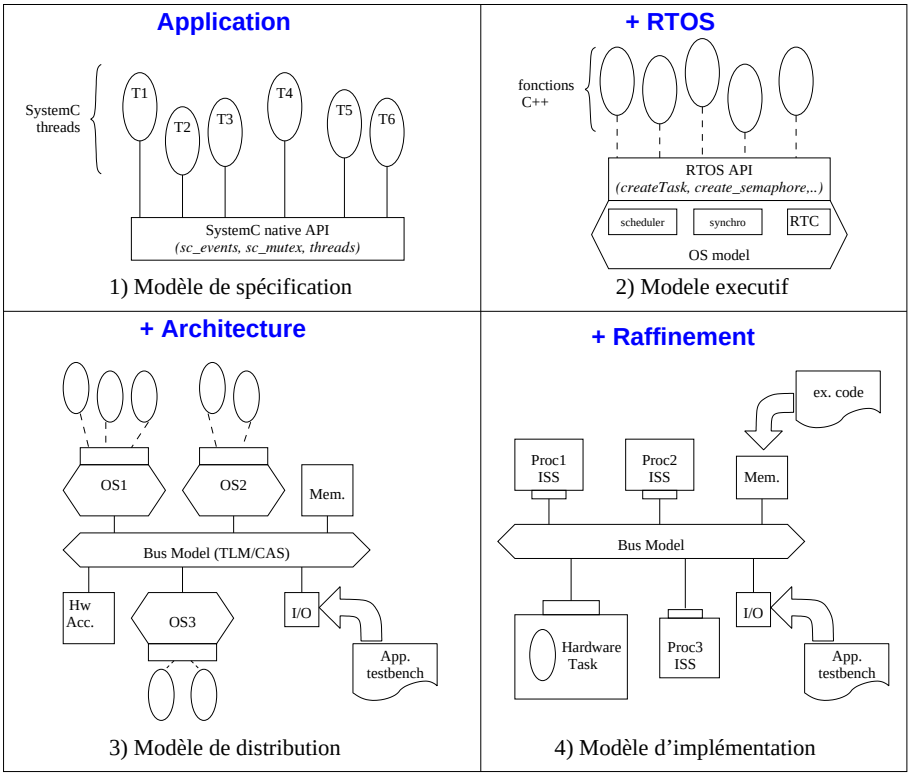


Figure 5.2: *Le flot de raffinement de plate-forme proposée commence avec le modèle spécification et prévoit au moins quatre mesures de raffinement jusqu'au modèle d'implémention.*

ont lieu à travers des canaux primitifs SystemC : `sc_mutex`, `sc_fifo`, etc. Les tâches sont synchronisées par `sc_events`. L'ensemble de l'application est spécifiée comme un seul `sc_module` avec toutes les tâches comme fonctions membres. Une vérification fonctionnelle complète de l'application est possible à ce niveau. Le noyau SystemC et son API modélise l'architecture virtuelle pure qui exécute l'application. En raison du principe de concurrence inhérent à SystemC, un nombre infini de ressources est simulé à ce niveau. En outre, des bibliothèques C++ peuvent être utilisées pour déboguer le code et peut donner une trace fonctionnelles de l'application.

On pourrait se contenter de prendre une application codée en langage C / C++ qui repose sur l'API POSIX pour faire un test fonctionnel utilisant des fonctionnalités de multithreading et des synchronisations, mais utiliser directement SystemC de bout en bout de la méthode est préférable.

5.2.2.2 Modèle exécutif

Le deuxième niveau affine le précédent en ajoutant une couche explicite au modèle de l'application, celle du modèle de RTOS de cette thèse. Le code source de l'application a accès aux services du RTOS à travers l'ensemble de l'API du modèle de RTOS. Tous les accès aux canaux primitifs SystemC sont donc remplacés par les appels système. Cela peut être fait automatiquement par pré-traitement (preprocessing) avant la compilation.

Afin de rendre la modélisation plus réaliste, il est préférable d'indiquer les temps de traitement des blocs de base (BB) des tâches (les `OS_wait()`), mais cela n'est pas obligatoire.

A ce niveau, un certain nombre de stratégies peuvent être explorées pour personnaliser les services de l'OS (politiques d'ordonnancement, le modèle de préemption et stratégies de certains services en option). L'exploration du partitionnement logiciel ou matériel des tâches applicatives n'est pas encore abordé à ce niveau. L'objectif principal est d'explorer le séquencement global des opérations.

5.2.2.3 Modèle de Distribution

Le modèle de distribution est le niveau où l'on va répartir les tâches sur différents nœuds d'exécution. On va devoir ajouter des primitives de communication entre les tâches pour se partager les données et on pourra ainsi évaluer le déroulement découlant des choix du mapping. A ce niveau, les ressources de calcul ne sont pas modélisées en détail, mais les modèles de RTOS incluent les ressources nécessaires à la communication entre nœuds (CAS externe). Néanmoins, le choix des ressources de calcul peut être ajusté (avec la fréquence du processeur), et on peut alors évaluer rapidement le facteur d'accélération de tel ou tel mapping choisi.

5.2.2.4 Modèle d'implémentation

Cette étape de raffinement et la dernière avant la synthèse physique de la plateforme. La partie matérielle (mémoires, ressources d'entrée/sortie, interrupt handlers, accélérateurs, etc.) est décrite au niveau RTL (Register Transfer Level). Les processeurs sont représentés par leurs ISS et le code de l'application est (cross-)compilé en code binaire compatible avec les processeurs cibles. Les résultats sont précis au niveau cycle (CA) ou même au niveau des signaux électriques (Bit Accurate). A ce niveau l'exploration et la validation des nombreuses stratégies a déjà été effectuée et on peut utiliser les outils classiques de conception matérielle pour réaliser le RSoC.

5.2.3 L'outil DOGME

Le rôle de l'outil DOGME[1] est de générer automatiquement un modèle (SystemC) complet exécutable d'une plate-forme de type RSoC à partir d'une description schématique. Cette description fait appel à des composants prédéfinis dans une bibliothèque de composants et permet également au concepteur de définir les propres éléments d'une bibliothèque. L'outil suit quatre étapes de conception :

La description de la plate-forme : Cette phase consiste en l'instanciation graphique (cf. figure 5.3) de composants de l'OS (ses services) à partir d'une ou plusieurs bibliothèques prédéfinies. L'utilisateur choisit également sur quelles unités d'exécution ces services seront implantés. Il décide aussi de la localisation de tâches applicatives sur les différents nœuds/RTOS.

La génération du code SystemC : Après vérification de toutes les instanciations ainsi que des connexions entre composants, le code des différents services ainsi que leur interconnexion est généré.

Compilation et simulation de la plate-forme : Afin de compléter la création de la plate-forme, le code structurel SystemC est combiné avec le code comportemental des composants issus des bibliothèques. Le code global est ensuite compilé puis simulé.

Analyse des résultats de simulation (métriques) : Des diagrammes sont affichés par l'outil pour tenir compte de l'évolution de la plate-forme au cours du temps. D'autre part, des informations pertinentes portant sur le déroulement de l'application sont également fournies de manière à aiguiller le concepteur dans ses choix.

L'outil a été réalisé par M. Aïchouch, ingénieur au laboratoire, sous la plateforme de développement Eclipse 3.3[Ecl] et constitue un plugin indépendant. Il permet d'utiliser les librairies de service d'OS déjà implémenté au sein du projet et permet de sauvegarder des designs dans un format XML afin de pouvoir les réutiliser ou les transmettre à d'autres

1. Distributed Operating system Graphical Modeling Environment

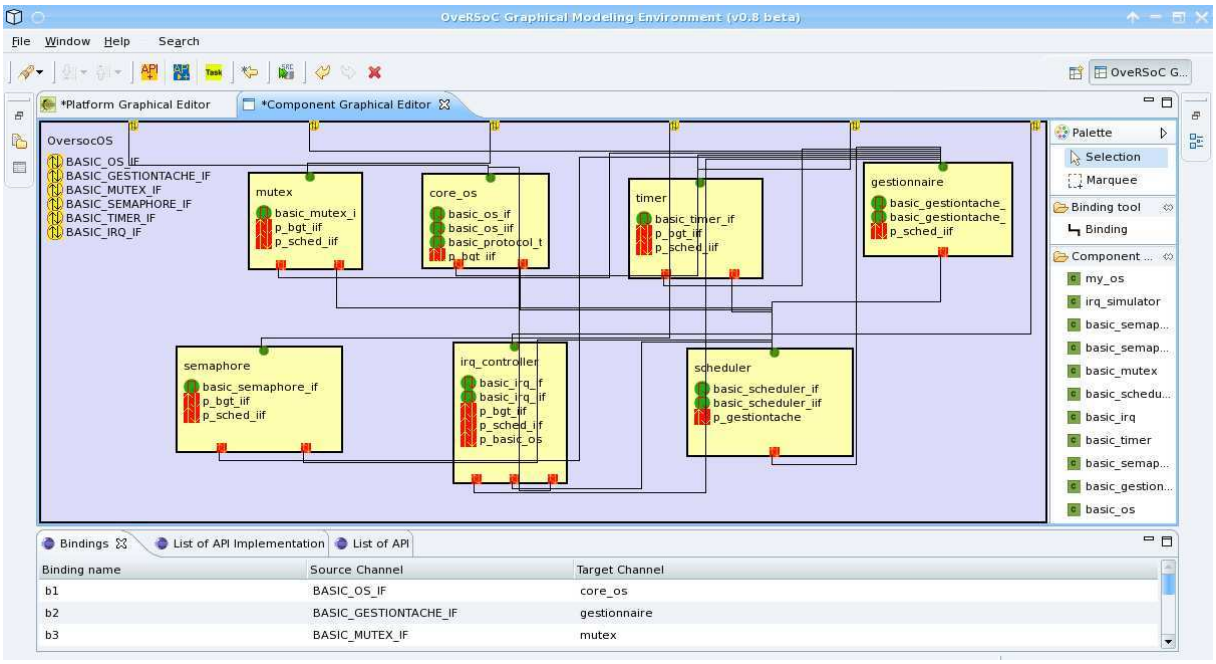


Figure 5.3: *L'outil DOGME représente un RTOS distribué à travers des vues hiérarchiques : la vue graphique* Éditeur de composants*, dans lequel les services sont organisés à l'intérieur de chaque RTOS, et la vue graphique* Éditeur Plateforme*, où les groupes de services (les OS) sont interconnectés selon le nombre et le type de nœud d'exécution de la plateforme RSOC. Ici, la vue* Éditeur de composants *est montrée.*

utilisateurs. Ces services suivent les modèles de construction décrits aux chapitres 3 et 4. L'outil a été testé sur une application ce qui nous a permis de vérifier ses capacités pour faciliter le mapping des threads applicatifs, leur partitionnement logiciel/matériel, l'exploration et l'instanciation des services des RTOS dédiés et la distribution des services entre les OS.

5.2.4 Consolidation du modèle de temps, utilisation d'un ISS

Dans ce travail, nous avons utilisé un ISS (simulateur de jeu d'instruction) pour la simulation du logiciel. Comme une preuve de concept de notre approche de modélisation logicielle, nous [2] avons développé un ISS en SystemC correspondant à un processeur ATMEL AVR [AVR]. Le ciblage d'un processeur réel câblé ou d'un simulateur sous forme d'ISS suit le même flux de compilation. Nous pouvons ainsi réutiliser des outils de compilation standard, dans notre cas, les codes logiciels sont compilés avec *avr-gcc* qui génère l'exécutable adéquat. Le code binaire doit ensuite être chargé (bootstrap) dans un module externe représentant la mémoire dans le modèle SystemC. L'ISS communique avec la mémoire à travers les canaux hiérarchiques standards. A ce niveau de modélisation, les communications peuvent être affinés au niveau Register Transfer Level (RTL). L'ISS va chercher des instructions et simule l'exécution des *opcode*. En fonctionnement normal, l'ISS extrait, exécute les opcodes 16-bits et incrémente le compteur de programme. L'ISS peut également être interrompu et préempter une tâche à un niveau très fin de modélisation.

Comme les composants au sein du modèle peuvent être décrits à différents niveaux d'abstraction, le défi est donc de synchroniser la simulation fonctionnelle et temporelle à travers les niveaux. Cela est particulièrement difficile pour le logiciel modélisé qui existe sur trois niveaux différents en même temps : le code de l'application est modélisée comme

2. équipe OveRsoC : S. Viateur, B. Bennani, B. Miramond et moi-même

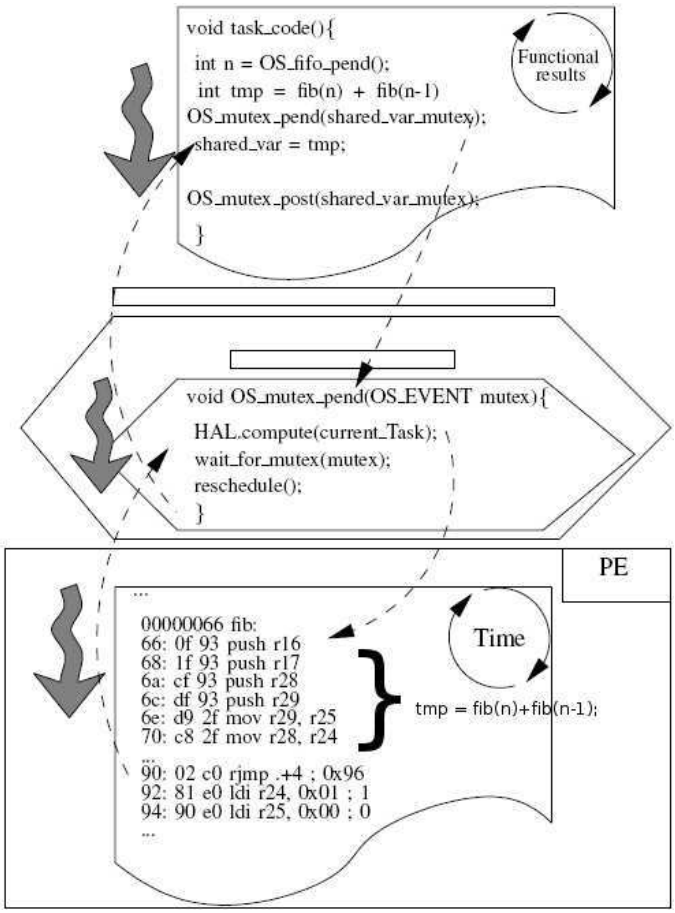


Figure 5.4: *Exemple d'un* Simulation Couple. *Le logiciel a deux représentations : une fonctionnelle de haut niveau non* timé, *et une autre sous forme de code exécutable compilé interprétable par un processeur ou un ISS, qui alors saura déterminer le temps d'exécution au fur et a mesure de son interprétation.*

des fonctions C, les services du RTOS comme des opérations SystemC de haut niveau, et la version du logiciel compilée pour une simulation au niveau des instructions par l'ISS. Grâce à l'approche de séparation des préoccupations adoptée, les aspects fonctionnels (couche application) et temporelle (couche de l'architecture) peuvent être séparés. Les aspects fonctionnels et temporels sont donc limités aux couches correspondantes. Par conséquent, une description du logiciel précis au cycle près (CA) a son équivalent fonctionnel de haut niveau à l'intérieur de la couche supérieure. Ici, la duplication de la description de l'application suit et renforce la séparation des préoccupations. Ce principe facilite la conception de logiciels embarqués en permettant la réutilisation d'IP logicielles, la simulation des portions de code avec des niveaux de développement hétérogènes, et l'exploration des services de RTOS. En outre, la méthode peut être appliquée également à l'implémentation de tâches applicatives matérielles puisque le modèle de haut niveau ne fait aucune hypothèse sur le partitionnement matériel / logiciel. Cette coexistence de la description de la tâche et de sa version raffinée forme un couple que l'on nomme Simulation Couple (SC).

Ainsi, l'exécution cohérente du SC ne dépend que d'une définition commune des points de synchronisation. Ces correspondances sont faites par les appels système du RTOS présents à la fois dans le code de haut niveau et dans le code binaire. Ainsi, la granularité des blocs de base (BB) dans l'ISS est toujours définie comme les séquences d'instructions entre deux appels système. À chaque tâche est associée un processus de SystemC et un événement de synchronisation géré par le modèle de RTOS et partagée par tous les BB

de la tâche.

Comme le montre la figure 5.4 l'ordonnanceur lance la tâche prête et élue en lui notifiant son événement de synchronisation. Le processus correspondant est activé et le code fonctionnel est exécuté dans la couche supérieure en un temps de simulation nul jusqu'à rencontrer un appel à l'API du RTOS. Les appels système de haut niveau à ce moment vont déléguer à l'ISS l'évaluation du temps d'exécution du BB qui précède. Sans interruption, l'ISS estime la durée du BB et avance le temps de simulation. Si une interruption se produit au milieu d'un BB, l'ISS s'arrête au moment précis et sauve le contexte de travail. L'interruption est alors relayée au modèle de RTOS qui la traite (joue l'ISR). Lorsque l'ordonnanceur est réactivé, il peut décider (en fonction de la politique d'ordonnancement choisie) quelle tâche doit repartir ou en élire une nouvelle. Le même scénario se répète à nouveau jusqu'à la fin de la simulation.

Avec un couplage entre le modèle de haut niveau et un ISS, on est capable de simuler n'importe quelle plateforme particulière à un niveau de précision très précis, excepté pour les durées des appels système qui continuent d'utiliser des `OS_wait()` basés sur des mesures ou estimations. Cette approche s'apparente fortement à l'approche utilisée dans les travaux de [KEBR08] où l'OS est modélisé à haut niveau et découplé des tâches exécutées sur un ISS.

L'inconvénient est que la simulation avec un ISS est énormément plus lente, dans notre cas de l'ordre de 500 fois plus longue qu'avec le modèle de haut niveau avec le code exécuté en natif et annoté temporellement. Ce mode de fonctionnement doit être réservé pour la validation finale lorsque l'architecture est déjà suffisamment stabilisée, uniquement pour vérifier des points précis. Par ailleurs, cela n'est possible qu'à condition de disposer d'un ISS en SystemC correspondant au modèle de processeur embarqué ciblé, et modifié pour être couplé à notre modèle pour s'interrompre et rendre la main au modèle de plus haut niveau lorsqu'il détecte un appel système. Dans notre cas, la conception de l'ISS d'AVR a demandé une grosse charge de travail, pour peu de bénéfices en rapport aux besoins de précision pour explorer et évaluer l'architecture à haut niveau. L'utilisation de ArchC [Cen] comme ADL [3] SystemC paraît alors évident.

5.2.5 Multi OS hétérogènes, utilisation du modèle d'ARD

Dans la section précédente nous avions effectué une exploration de l'architecture pour l'application de vision robotique. Les résultats nous ont conduit à la partitionner en un ensemble de 12 tâches logicielles (rejouées N fois selon le nombre de points d'intérêt) et un ensemble de 18 tâches implémentées comme accélérateurs matériels. En se basant sur la synthèse de blocs matériel effectuée par T. Lefebvre, nous avons annoté leurs caractéristiques dans le simulateur (nombre de slice, durée d'exécution, latence des communications et temps de configuration).

Le simulateur comprend maintenant un modèle d'ARD décrit à un niveau plus fin de détail. Ce modèle est un OS particulier aux services spécifiques à la gestion de zones reconfigurable (création de tâches : chargement des bitstreams, synchronisation, etc.). Les services d'un ARD seront par hypothèse implémentés en matériel, mais ce qui compte à notre niveau d'exploration, c'est surtout l'aspect algorithmique et temporel de la gestion du placement des tâches matérielles que que le projet OveRSoC cherche à explorer pour bien dimensionner la zone reconfigurable.

Identification des services spécifiques à la gestion du reconfigurable

Afin de gérer les ressources reconfigurables présentes au sein du RSoC, les services suivants ont été définis puis modélisés :

3. Architecture Description Language

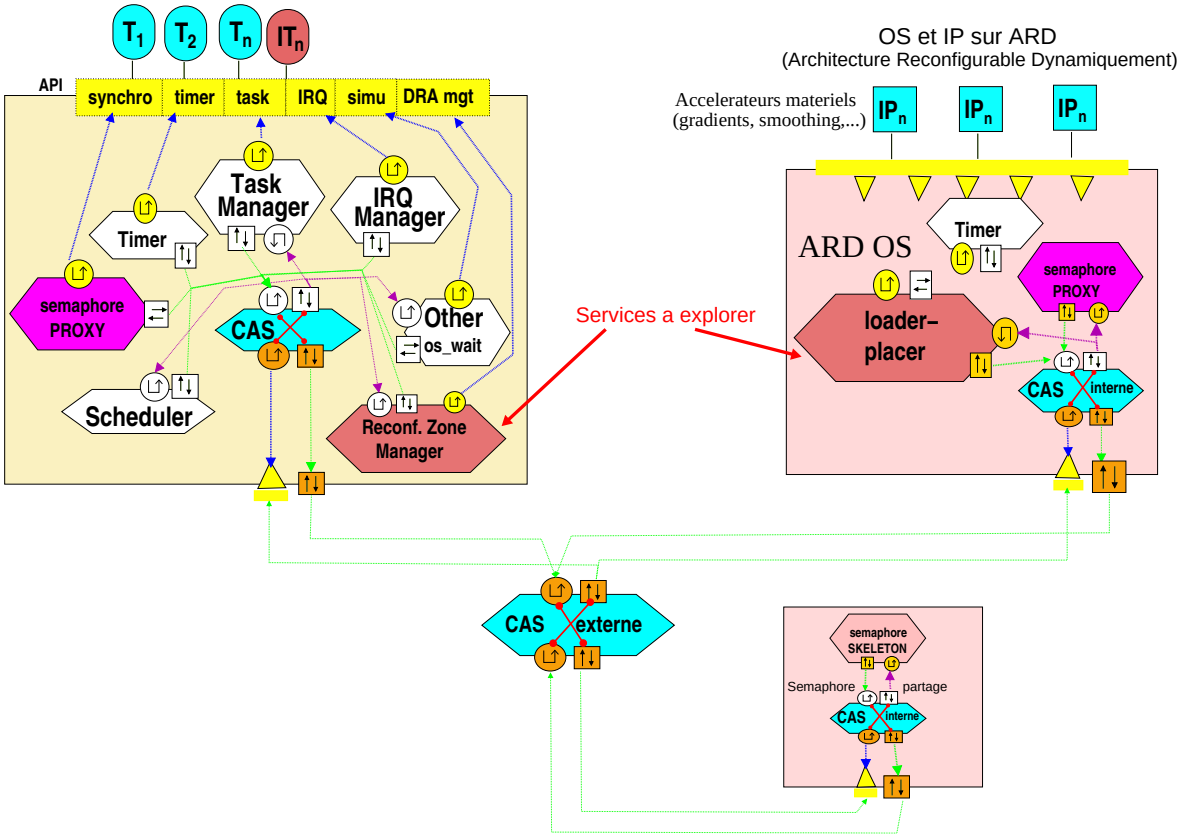


Figure 5.5: *Modèles d'OS spécifiques au RSOC : un nœud processeur avec un OS standard personnalisé avec un gestionnaire d'ARD, un nœud/OS spécifique dans l'ARD, et un autre OS spécifique avec un seul service raffiné de sémaphore partagé représentant un périphérique externe. Les OS communiquent par l'intermédiaire du CAS externe qui sera raffiné en bus ou un autre média éfficace.*

- **L'ordonnancement spatio-temporel** est sans aucun doute le service le plus important dans un OS multi-tâches temps-réel. L'ordonnancement des tâches matérielles dans un ARD ajoute une dimension spatiale au problème classique d'ordonnancement de tâches logicielles. Dans notre modèle, plusieurs types de services ont été élaborés de manière à prendre en compte le placement 1D ou 2D des tâches dans l'ARD.
- **Le service de gestion des ressources et de la reconfiguration** : ce dernier est très proche du service de placement et a besoin de connaître l'état du système global afin de gérer l'utilisation optimale des ressources de calcul sur le circuit. Un nouveau modèle de tâches met également en exergue les propriétés des tâches matérielles.
- **La préemption des tâches et la migration** : la migration des tâches matérielles au sein d'ARD voire en son équivalent logiciel est une propriété intéressante nécessitant de mettre en place la préemption coté matériel. Ces propriétés sont supportées par des services tels que le placement dynamique et la communication.
- **Les communications flexibles** : ce service permet de gérer les échanges de données entre les tâches de même nature ou de différente nature.

Grâce à la capacité de notre modèle à créer des tâches localisées dans un nœud particulier, on est capable de créer, préempter et détruire des tâches matérielles d'un nœud ARD à partir d'un service proxy sur un OS standard. Comme illustré dans la figure 5.5 on propose un ensemble de modules de services qui remplissent les fonctions de gestion de tâches matérielles. On peut définir différents niveaux de raffinement des détails de l'ARD, en considérant un premier niveau où l'ordonnanceur de l'ARD considère uniquement la surface disponible pour élire les tâches. D'autres niveaux de détail plus fin peuvent

être définis, prenant en compte la forme 2D des tâches, leur deadline et leur temps de configuration pour ordonnancer l'ensemble.

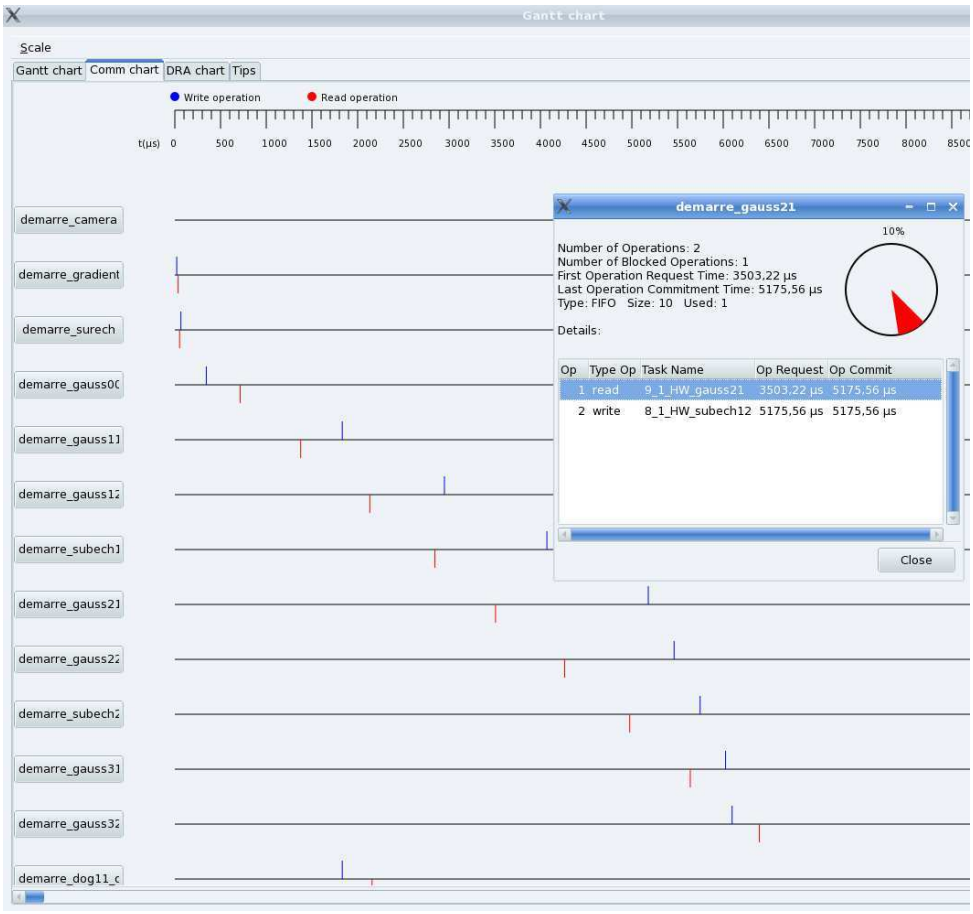
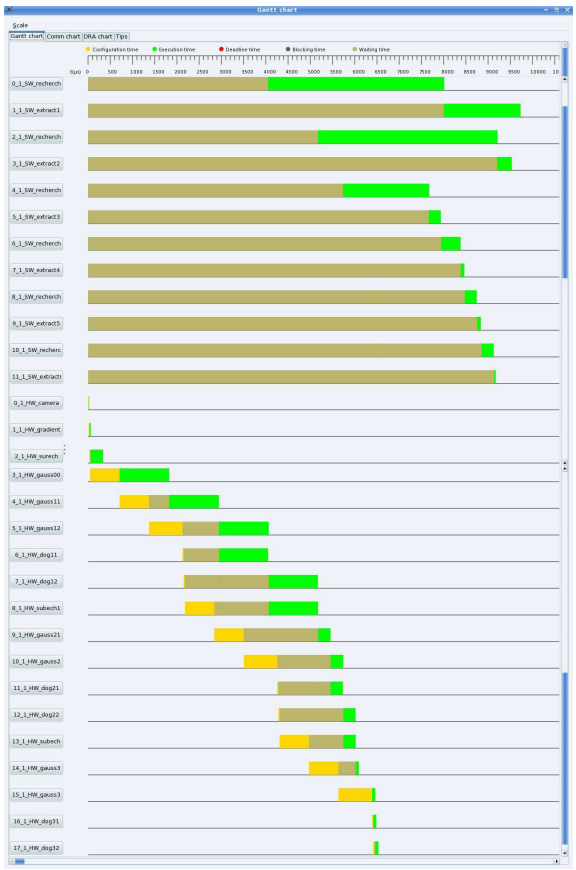


Figure 5.6: *L'outil DOGME fournit plusieurs mesures aidant le concepteur à évaluer les solutions simulées. Ce graphique affiche les communications dans le temps entre les tâches. L'outil calcule aussi le taux de remplissage des FIFO (buffers matériels ou logiciels) support des communications.*

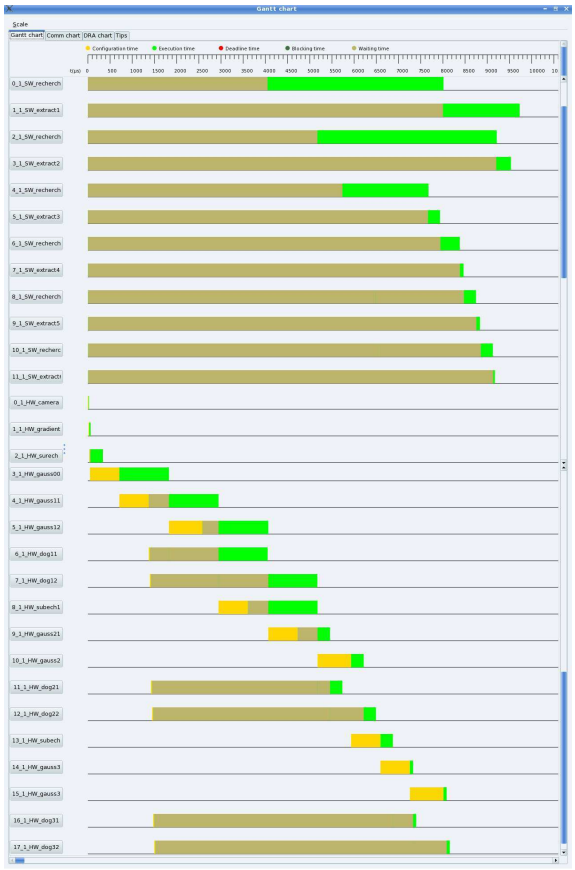
La phase précédente d'exploration nous avait montré une configuration avec 3 nœuds processeur et un nœud reconfigurable (ARD) adjoints d'un nœud de synchronisation (le service de sémaphore externe partagé et indépendant).

Quelques exemples de paramètres mesurés et utilisés pour le dimensionnement du système sont le diagramme de Gantt 5.7(a) de toutes les tâches, le graphique de l'utilisation de l'ARD (Fig. 5.7(c)) ainsi que les communications par FIFO décrites dans la figure 5.6. Le diagramme de Gantt représente l'état de chaque tâche (logicielle ou matérielle) au cours du temps : prêt, en cours d'exécution, en attente et un état particulier pour la configuration réservée aux tâches matérielles. Les diagrammes de Gantt de la figure 5.7(a) décrivent la configuration de l'architecture du système de la phase précédente (3 OS normaux, un OS/ARD et un nœud de synchronisation). Les 12 lignes supérieures représentent l'ordre des tâches logicielles sur les 3 processeurs et les lignes restantes représentent les 18 tâches matérielles fonctionnant en parallèle dans l'ARD. Cette architecture correspond à la meilleure performance possible, car la taille de l'ARD a été calculée comme la somme de l'occupation des tâches matérielles. Plus précisément, la partition matérielle utilise près de 1200 slices [4], 14 blocs BRAM et 12 blocs DSP48. Par conséquent, l'utilisation des ressources prévues pour l'architecture globale (ARD + trois processeurs) est d'environ

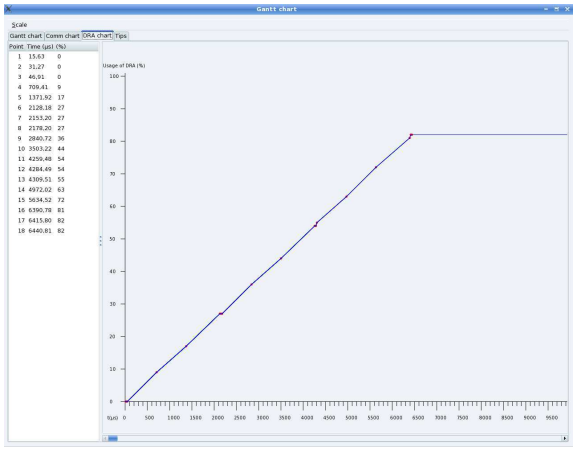
4. les slices des FPGA Virtex-5 sont organisés différemment des générations précédentes. Chaque slice de FPGA Virtex-5 contient quatre LUT et quatre flip-flops



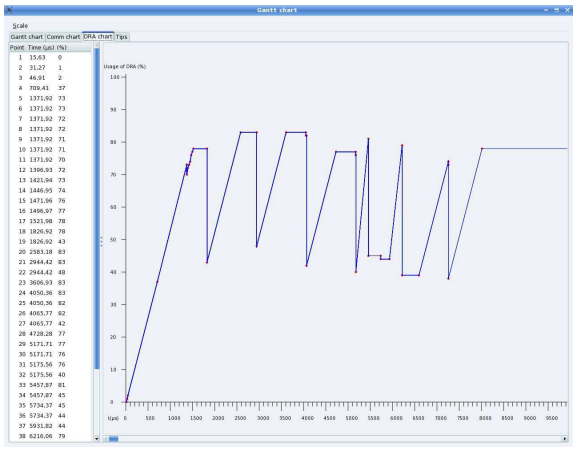
(a) Diagramme de Gantt d'un gros ARD



(b) Diagramme de Gantt d'un petit ARD



(c) Taux d'occupation d'un gros ARD



(d) Taux d'occupation d'un petit ARD

Figure 5.7: *Les cadres du haut représentent les diagrammes de Gantt pour toutes les tâches réparties sur 3 processeurs et un ARD de 4500 slices à gauche et seulement 3000 à droite. Les diagrammes du bas représentent le taux d'occupation des ARD au cours du temps et montrent le surcoût de la reconfiguration. Pour des diagrammes plus lisibles, ces quatres figures sont reproduites en pleine page en annexe B page 153.*

4375 slices, 21 blocs BRAM et 16 blocs DSP48. Cette estimation correspond par exemple à la taille d'un circuit Virtex 5 LX30T [Xil].

Nous avions dans cette exploration considéré l'espace reconfigurable comme suffisant pour accueillir tous les blocs matériels en même temps. Or il n'est pas nécessaire de faire fonctionner toutes les tâches matérielles en parallèle. On peut donc se poser la question de la taille minimale de la zone reconfigurable permettant de jouer l'application dans les délais impartis sans gaspiller de la surface silicium.

De manière à réduire la taille de la partie matérielle, nous avons fait varier le nombre de slices de l'ARD et évalué la capacité du système à adapter l'ordonnancement (algorithme simple du premier arrivé et qui tient dans une zone libre) sur une zone plus restreinte. Dans la figure 5.7 nous avons représenté les résultats de l'exploration de l'architecture. On peut remarquer sur le diagramme de Gantt les différences d'ordonnancement des IP matérielles selon le taux d'occupation de l'ARD. La comparaison entre l'ARD aux ressources largement suffisantes (Fig. 5.7(c)) et celui de taille réduite (Fig. 5.7(d)) montre une claire différence de l'utilisation de l'ARD au cours du temps.

Dans le premier cas (Fig. 5.7(a) and 5.7(c)), l'ARD n'est jamais complètement occupé et les tâches sont configurées dès qu'elle sont crées et que la configuration peut être faite (une seule à la fois). Elles s'exécutent alors normalement selon la disponibilité des données. Dans le second cas (Fig. 5.7(b) et 5.7(d)), on considère un ARD avec seulement 3000 slices. Il est impossible de configurer toutes les IP en même temps sur cet ARD. Dans ce cas l'ordre de configuration dépend de la disponibilité des ressources reconfigurables, en plus des dépendances de données.

A ce niveau grossier de modélisation, l'ordonnanceur matériel de l'ARD prend en compte uniquement les ressources en terme de slices disponibles. Il cherche les tâches en état dormant et les supprime de la zone reconfigurable en libérant des ressources utiles pour permettre de configurer de nouvelles tâches demandant à être exécutées. Pour l'estimation du temps de configuration, nous avons utilisé une métrique dépendante de la taille des *bitstreams* des tâches, selon la technologie ciblée (environ 50 μs par bloc de 16 slices sur un Virtex 5).

5.2.6 Conclusion du projet OveRSoC

Les travaux au sein du projet OveRSoC ont permi de valider notre modèle d'OS sur un cas plus complexe. En particulier, on a ainsi pu modéliser une architecture multi-nœuds hétérogènes, sur laquelle on a pu distribuer les services, en particuliers les services liés à la gestion de zone reconfigurables.

5.3 Exemple de déploiement du modèle : Travaux dans le projet Ter@Ops

Nous allons tout d'abord aborder la description de l'architecture Ter@OPS, puis comment le modèle d'OS développé dans cette thèse s'intègre dans cette structure.

5.3.1 Objectifs du projet Ter@OPS

Le projet Ter@ops [Sys] est un projet du pôle de compétitivité System@TIC de la région Île de France [Fra] dirigé par l'équipe de Thales TRT/LSE. Il s'est déroulé sur trois ans et a suscité les efforts de nombreux partenaires, composé à 56 % d'industriels, 19% de PME et 25% d'académiques, dont ETIS.

Problématique du projet Ter@ops Les systèmes complexes « embarqués » requièrent de plus en plus de traitements intensifs et des performances temps-réel garanties. Les processeurs de traitement réalisés pour de tels systèmes (multimédia, imagerie, etc.) sont traditionnellement spécialisés par types d'applications pour produire la puissance de calcul la plus grande pour la plus petite consommation. Ces solutions dédiées doivent cependant faire face à une augmentation rapide des coûts d'accès au silicium, à des algorithmes toujours plus complexes et rapidement évolutifs, à des besoins de faible consommation et de robustesse. Or ces architectures sont peu flexibles et difficilement programmables. Les autres systèmes complexes de traitement intégrés, employés dans l'informatique traditionnelle (PC) sont très souples dans leur emploi mais au détriment de leur efficacité (forte puissance de calcul mais consommation élevée). Ces limitations, l'évolution rapide des capacités d'intégration (2,5 milliard de transistors sur une puce Altera Stratix IV en 2008 [EET]) et les contraintes du marché (coût de fabrication, délais, risques technologiques) conduisent à une rénovation des concepts de base des machines de traitement pour les applications embarquées. Nous sommes actuellement à un point d'inflexion sur le marché du traitement haute performance pour l'embarqué. Ainsi, ce projet propose une solution de rupture apportant à la fois efficacité de traitement et souplesse d'emploi par une facilité de programmation pour de nombreuses applications de traitement intensif (imagerie, multimédia, etc.). Celle-ci repose sur :

- Une architecture de traitement programmable massivement parallèle,
- Un environnement de développement complet pour la programmation d'applications sur cette *machine multiprocesseurs sur puce*.

Objectifs L'étude proposée est une première phase de validation par maquettage des concepts novateurs proposés en vue du développement en vraie grandeur du produit. Ter@ops vise à concilier :

- la souplesse d'une solution réutilisable multi-applications pour l'efficacité économique,
- la densité de performance (puissance de calcul et faible consommation),
- la robustesse et la sûreté de fonctionnement,
- l'efficacité en programmation malgré la complexité de l'architecture,
- la pérennité des applications par la virtualisation des services de la machine.

Il s'agit avant tout d'adresser deux grandes classes d'algorithmes (image /signal) très utilisés dans les applications des partenaires industriels et qui sont déterminants pour le dimensionnement de la machine. Cela passe par la conception d'une machine parallèle sur silicium, un modèle de programmation versatile supportant plusieurs types de parallélisme et des outils de programmation assurant une forte productivité dans son exploitation. L'approche Ter@ops est rendue possible par la mise en coopération des compétences systèmes associées aux compétences en plateformes SoC, en compilation et architectures de traitement parallèles.

Enjeux L'enjeu pour les industriels de ces domaines où les quantités de produits sont souvent moyennes (quelques milliers à quelques centaines de milliers de pièces) est l'accès à des technologies de systèmes complexes intégrés efficaces, déterminantes pour la performance globale de leurs produits. Cela concerne leur capacité à développer les systèmes embarqués de demain.

Le résultat attendu en fin de ce projet est un simulateur avec l'environnement de développement et d'exécution nécessaire. Un second projet permettra ensuite de passer à la conception de la puce de manière concrète.

5.3.2 Architecture retenue

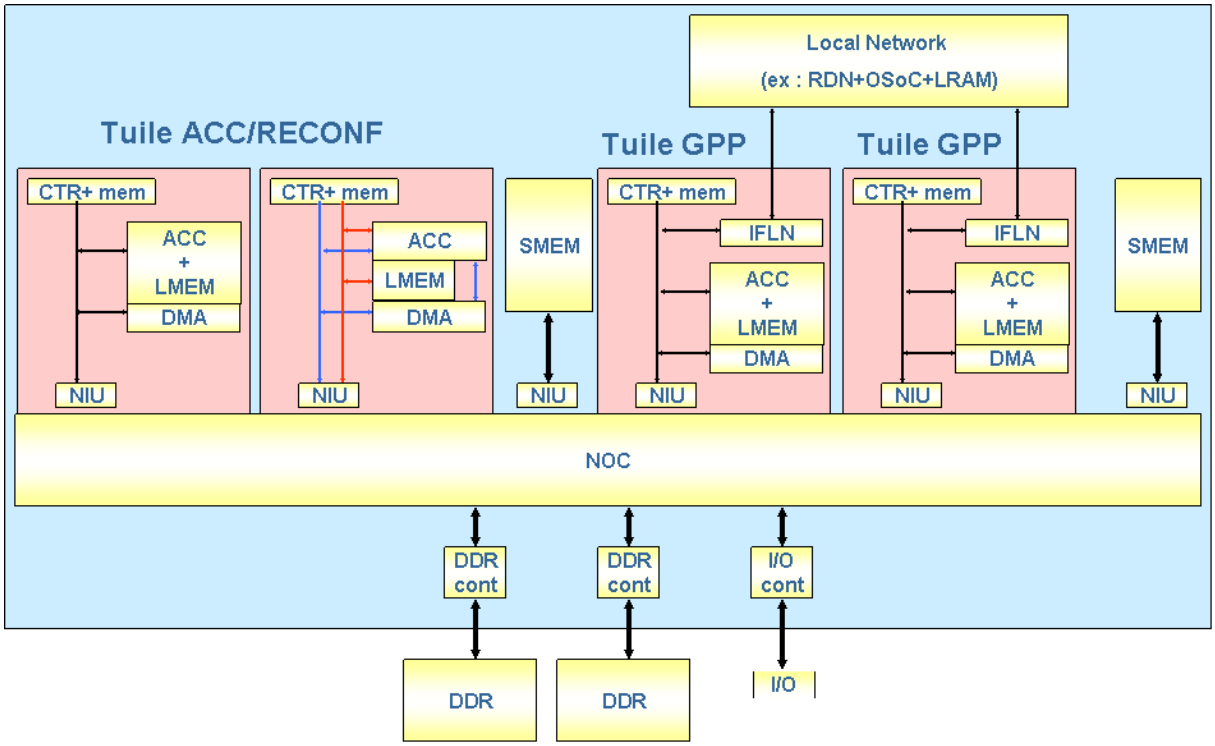


Figure 5.8: *Architecture Ter@ops Générique : Les tuiles quelques soient leur fonction accélératrice (GPP/SIMD/SCMP/reconfigurable) possèdent toutes un NoC interne reliant un contrôleur, une mémoire locale, un module DMA ainsi qu'une interface vers les NoC externes*

Afin de fournir une plateforme générique permettant d'exécuter diverses applications hétérogènes de calcul intensifs, l'architecture retenue se base sur le principe d'un système *globalement homogène, localement hétérogène*. Cela se traduit par une architecture à base de tuiles regroupant une interface générique et masquant les disparités des accélérateurs qu'elles contiennent, le tout relié par plusieurs réseaux sur puce spécialisés comme schématisé sur la figure 5.8. Cette approche permet de répondre à la problématique de *scalabilité* nécessaire pour obtenir la puissance de calcul demandée, tout en ayant la flexibilité de choisir la répartition des types d'accélérateurs des multiples tuiles.

Architecture générique des tuiles

Bien que les tuiles aient chacune une fonctionnalité particulière, grâce à l'accélérateur qu'elles embarquent, elles sont néanmoins toutes architecturées de la même façon. Chaque tuile est constituée, comme on peut le voir dans la figure 5.9, des sous-modules suivants :

- Un petit processeur (CTR) à usage génaral (GPP) fait office de contrôleur de la tuile. Le CTR dispose d'un cache mémoire.
  - L'utilisation d'un cache instructions est nécessaire pour donner la possibilité d'utiliser du code en mémoire partagée (onchip ou offchip), dans le cas où tout le code ne tient pas dans la mémoire locale (LMEM).
  - L'utilisation d'une mémoire cache de données n'est pas vraiment nécessaire car elle ne peut mettre en cache que la LMEM. Les caches de données ne sont donc pas utilisés, afin d'éviter l'utilisation de modules de gestion de cohérence de cache et de laisser au logiciel applicatif le contrôle total de la répartition et de l'utilisation mémoire.
- Un module de transfert de données (DMA, Direct Memory Access) géré uniquement par le CTR
- Une mémoire locale de la tuile (LMEM)

- Un réseau interne (Internal Network) à chaque tuile connectant les différents organes entre eux. Tous les éléments de la tuile ne sont pas forcément interconnectés entre eux.
- Un accélérateur (ACC) composé des éléments suivants :
  - ACC_IF : une partie interface dialoguant avec le CTR et gérant l'infrastructure de l'accélérateur ACC (i.e. chargement du micro-code ou de la matrice FPGA, reset, ...). Contrairement aux autres éléments de l'accélérateur, cet élément n'est pas optionnel. Autrement dit, tout accélérateur contient toujours au moins l'ACC_IF.
  - ACC_MCU : une partie contrôle dialoguant avec le CTR et gérant le ACC_PROC (par exemple, un séquenceur de micro-instructions)
  - ACC_PROC : le cœur de l'accélérateur (e.g. un SIMD)
  - ACC_MEM : la mémoire interne de l'accélérateur, accessible uniquement par l'ACC_DMU ou l'ACC_CTRL. L'ACC_PROC effectue ses traitements sur l'ACC_MEM.
  - ACC_DMU : DMA très spécifique lié aux besoins d'organisation des données dans l'accélérateur.
- Deux modules d'interface réseau entre tuile (NIM, Network Interface Module) : en fonction du type de message à véhiculer :
  - NIM_ctrl_sync : utilisé uniquement pour faire passer des messages de contrôle et synchronisation. La latence exigée sur ce lien est typiquement < 10ns lorsque le réseau est "vide". Les NIM_ctrl_sync sont reliés entre eux formant ainsi un NoC spécifique garantissant la latence.
  - NIM_highBW : utilisé pour tous les messages à forte bande passante.
- PI : Les NIM sont capables de s'isoler du réseau grâce à un module de Power Isolation qui permet de désactiver n'importe quelle tuile dans l'architecture. Cette interface est pilotée par la tuile debug/supervision. Ce module s'assure qu'il est possible de désactiver la tuile avant de le faire (i.e. plus de message en cours de transit dans le NoC), et est donc implicitement réparti dans les NIM. Pour alléger les schémas suivants, le module PI ne sera pas dessiné, mais il est implicitement présent dans chaque NIM. Le PI isole tous les NIM d'une tuile en même temps.

Architecture synthétique Les détails du projet Ter@ops peuvent se résumer sur l'architecture ainsi schématisée sur une page dans la figure 5.10. Cela ne montre pas tous les aspects du projet qui a dû traiter de l'analyse des besoins des diverses applications, des problèmes de dimensionnement et de conception du SoC Ter@ops en intégrant les accélérateurs hétérogènes, de sûreté de fonctionnement, du flot de conception et des outils associés en plus du simulateur de l'architecture, synthèse des résultats du projet. Pour de plus amples détails, veuillez vous référer aux nombreux documents présents sur le site internet du projet [Sys].

5.3.3 Simulateur de l'Architecture

La validation des concepts de cette architecture matérielle a été effectuée au moyen d'un simulateur transactionnel avec temps, réalisé en langage SystemC.

Dans un projet incluant plusieurs partenaires, dire qu'on utilise SystemC n'est pas suffisant. En effet, l'utilisation de SystemC n'impose rien sur les interfaces des modules qui composent l'architecture. Ainsi il est nécessaire de fixer des règles de codage, ainsi que des règles de protocole de communication entre les modules. Les règles de codage des modules doivent préciser que les modules doivent être codés comme des composants. Ainsi l'accès direct aux variables des modules est interdit. On utilise des méthodes get() et set() qui permettent d'isoler le module de comportement non voulus, mais surtout qui rendent possible la composition et l'interchangeabilité de modules. Au niveau des

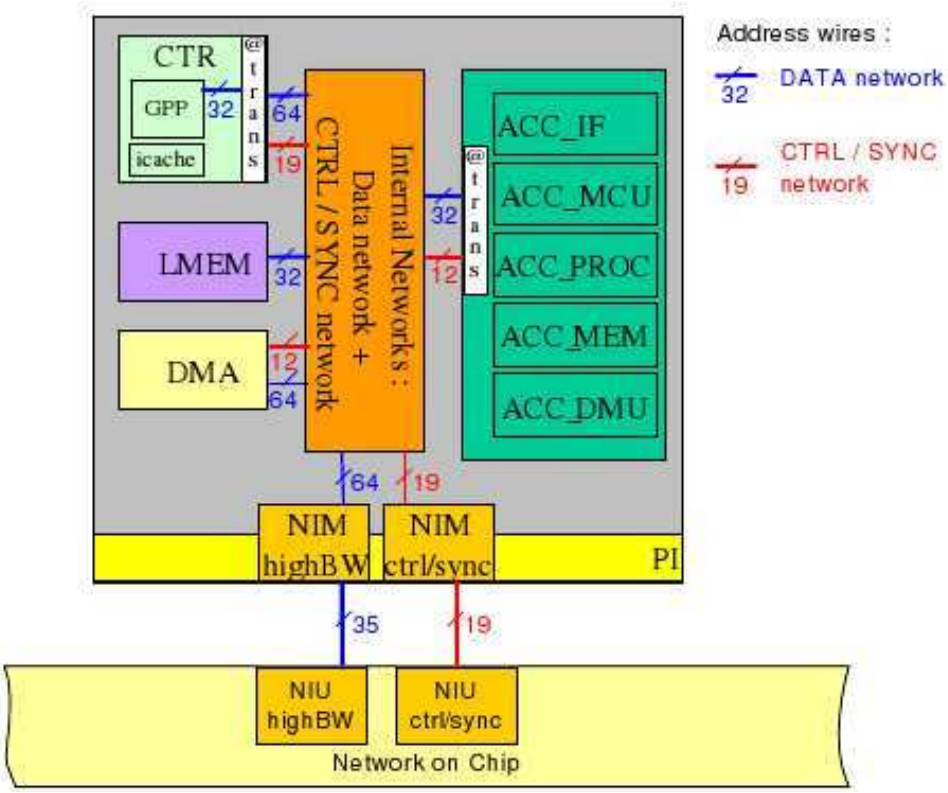


Figure 5.9: *Architecture d'une tuile générique*

interfaces plusieurs standards existent comme Virtual Component Interface (VCI [VSI]), Open Core Protocol (OCP-IP [OCP]) et Transaction Level Modeling (TLM 2.0).

Pour le projet Ter@ops, au moment du choix, TLM 2.0 n'existait pas, et OCP se présentant comme un successeur de VCI, nous avons choisi OCP. OCP est un protocole permettant de connecter deux modules entre eux comportant différents niveaux de précision (TL 1 à 4). Le niveau TL1 étant le plus précis (*Cycle Accurate*), TL2 et TL3 incluent des approximations des timings intra-burst et extra-burst, alors que TL4 n'inclue pas les timing mais est bloquant. OCP est un protocole point à point, unidirectionnel, entre un maître et un esclave. Un module A utilise son port OCP maître pour envoyer un message de requête dans le port esclave du module B. Le module B traite le message et envoie une réponse dans le port maître du module A. La communication est unidirectionnelle car seul le module A peut créer des requêtes, le module B ne peut que retourner des réponses. Pour créer une communication bidirectionnelle, il faut un port maître et un port esclave sur chaque module. OCP est un moyen d'interface et d'échange de message, il n'est pas un protocole de bus ou de réseau. OCP fournit le moyen de connexion et les méthodes de transfert de messages entre modules, mais n'impose pas le format de l'information qui passe dans le canal. Afin de standardiser l'information échangée entre les modules, des structures (ou classes C++) ont été créés, permettant à un module de ne pas avoir à sérialiser/dé-sérialiser l'information dans le canal. Le développement d'un module est alors simplifié mais surtout une modification dans la structure de l'information échangée (niveau de détail) n'implique pas de recoder le module, il suffit de modifier la structure du message à l'aide de transacteurs entre niveux de précision TL 1-4. Cette classe de messages permet de pister toutes les informations qui transitent dans le modèle et ainsi de réaliser une trace des temps caractéristiques de passage du message. Avec les éléments présentés nous avons mis en œuvre un ensemble de composants interchangeables permettant de tester et d'estimer les performances de différents modèles de bus, de mémoire, de processeurs, ou d'accélérateurs, ainsi que des stratégies DMA.

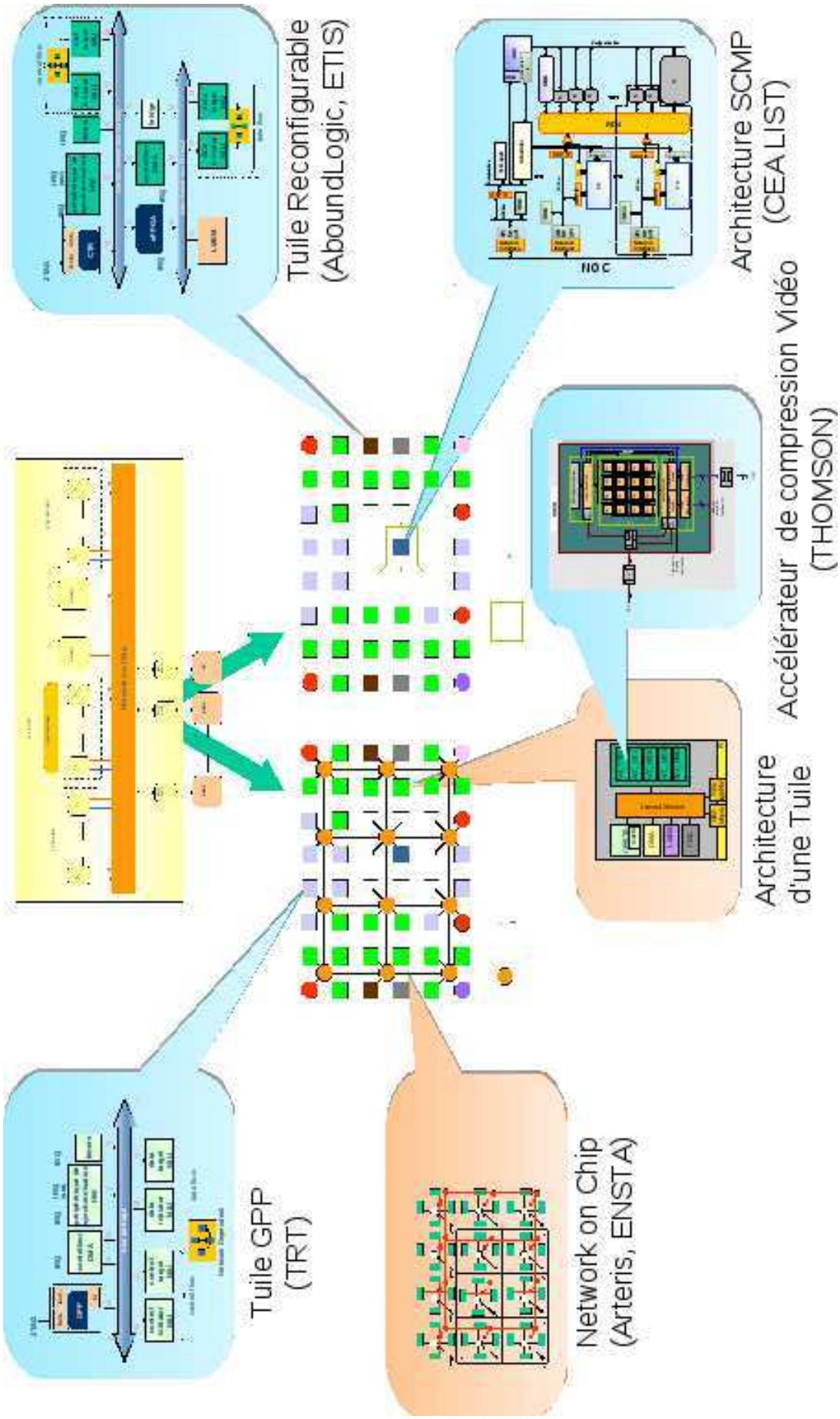


Figure 5.10: *Vue d'ensemble de l'architecture Ter@ops*

5.3.3.1 Architecture globale du simulateur

Pour réaliser une architecture en SystemC on instancie des modules puis on raccorde les ports des modules avec des canaux. Nous avons choisi de laisser à l'utilisateur du simulateur la possibilité de composer les éléments de son architecture en remplissant une trame pré-établie. Ainsi le nombre de tuiles est fixé, mais pas le type de chaque tuile. Ceci permet à l'utilisateur réalisant une application de placer son application sur les éléments qu'il souhaite et donc d'effectuer une exploration d'architecture. L'architecture se compose d'un Network On Chip (NoC) reliant des tuiles et des mémoires globales communes aux tuiles.

Le NoC et la mémoire : Le NoC est l'élément central, il permet les échanges entre les tuiles ainsi que l'accès à toutes les mémoires de l'architecture. Le NoC Arteris utilisé permet une vision totale de la mémoire et présente donc une architecture logique en mémoire partagée.

La tuile : La tuile est composée en deux parties, un cœur et une coquille. La coquille est un cadre dans lequel le développeur de la tuile accroche un cœur qu'il réalise en branchant des composants les uns avec les autres. La coquille permet d'homogénéiser les interfaces. L'intérieur du cœur dispose ainsi de deux ports OCP, l'un vers le NIM ctrl/sync, l'autre vers le NIM highBW. Les deux NIMs réalisent la conversion de protocole entre le cœur OCP et le protocole utilisé par le réseau.

Le contrôleur (CTR) : Pour réaliser le contrôleur, nous avons privilégié l'approche de la modélisation de niveau système pour éviter d'utiliser un ISS (instruction set simulateur) qui permettrait de simuler finement les instructions du processeur réel qu'on embarquerait. L'utilisation d'un ISS aurait augmenté le temps de simulation et n'aurait rien apporté aux résultats de simulation de niveau système. Ceci est d'autant plus vrai que la fonction du contrôleur est de stimuler les organes de la tuile et pas de réaliser des traitements lourds. Ainsi toutes les fonctions réalisées par le CTR sont des fonctions de bibliothèques appartenant à l'équivalent d'un système d'exploitation de la puce. C'est donc le point d'entrée que nous avons choisi pour la simulation de l'OS. Le CTR de chaque tuile exécute sa propre instance d'OS : celui développé dans cette thèse.

La mémoire locale (LMEM) : Cette mémoire est modélisée par un tableau. Les requêtes OCP sur le réseau permettent de lire et écrire dans cette mémoire. Plusieurs modèles de LMEM sont possibles. Nous avons réalisé une LMEM capable de réaliser des échanges de différentes granularités. Ainsi elle peut échanger sur le réseau au minimum 1 octet et au maximum 2^{32} octets. Cette limitation est donnée par la taille du message OCP.

Le réseau interne (iNoC) : Pour relier les organes dans la tuile nous avons choisi d'utiliser un module d'interconnexion générique, une sorte de réseau interne. Ainsi nous sommes capable de modéliser un cross-bar complet des ports, un bus unique ou toute combinaison de bus. Ceci permet l'exploration rapide au niveau de la structure de communication, sans avoir à modifier les autres modules de la tuile.

Le NIM : Le NIM a pour fonction de traduire les messages OCP (TL2) en messages NTTP (TL3 pour le NoC Arteris) et vice-versa. Il traduit les protocoles et est indépendant du contenu du message. Il redirige les messages venant du NoC vers les bons organes dans la tuile.

Le DMA : Le DMA permet de réaliser des échanges entre les mémoires sans intervention du CTR. Le CTR programme un échange à réaliser par le DMA en envoyant un message au DMA. Puis le DMA réalise cet échange.

L'accélérateur : L'accélérateur est implémenté et fourni par le concepteur de la tuile. Dans le cas de la tuile reconfigurable, il a été conçu par l'équipe du laboratoire ETIS, sur la base des travaux issus du projet OveRSoC évoqué plus haut.

5.3.4 Architecture logicielle et simulation

Dans le cadre d'un système aussi complexe et aussi programmable que l'architecture Ter@ops, le logiciel devient un élément clé du développement du produit. Il est donc primordial de concevoir les outils et briques logicielles qui vont permettre les développements applicatifs en même temps que la conception de l'architecture matérielle. Ce principe conduit à concevoir l'infrastructure logicielle en couches, dont les couches les plus basses sont directement intégrées à l'infrastructure générale.

Structure logicielle En termes d'infrastructure générale, une couche d'abstraction est primordiale pour permettre l'intégration et le développement rapide des applications. Cette couche, appelée communément *middleware* a pour but d'offrir des services unifiés et des méthodes de programmation homogènes pour l'ensemble des tuiles du système. Comme résumé sur la figure 5.11, le middleware permet d'abstraire les différentes interfaces matérielles des tuiles, au travers d'une API unifiée, celle de la Machine Virtuelle Ter@ops. La Machine Virtuelle (middleware) est composée de l'OS et de services middleware au-dessus.

5.3.4.1 Buts de la Machine Virtuelle (MV)

La Machine Virtuelle (MV) offre les services suivants :
- Service d'activation d'un co-processeur au sein d'une tuile,
- Opérateurs de traitement sur le co-processeur (lié à un domaine d'application),
- Dialogue entre tuiles,
- Servitudes (chargement, démarrage,..),
- Et est un support à l'exécution

Six principes, à la base de la définition des spécifications et du développement ont été identifiés, comme fondateurs de la MV :
- Garantir une indépendance entre l'application et les spécificités du matériel par une approche en couches. Les couches permettent de garantir la conservation du code d'une application en cas de changement du hardware et de conserver une partie de la machine virtuelle indépendante du matériel.
- Indépendance fonctionnelle de l'application par rapport à son placement physique sur l'architecture. Ce principe est l'essence même d'une machine virtuelle : faire abstraction de la cible sur laquelle est exécutée l'application.
- Minimiser l'empreinte mémoire de la MV en s'appuyant sur une approche composant. Sur chaque tuile, seuls les composants nécessaires au logiciel applicatif exécuté sur une tuile sont linkés.
- Garantir un certain niveau de prédictibilité en optant pour une implémentation de la machine virtuelle où tous les services sont exécutés localement à la tuile. Les opérateurs de traitement, par exemple, seront toujours exécutés localement à la tuile.
- Les deux modèles de programmation suivants devront pouvoir être utilisés :

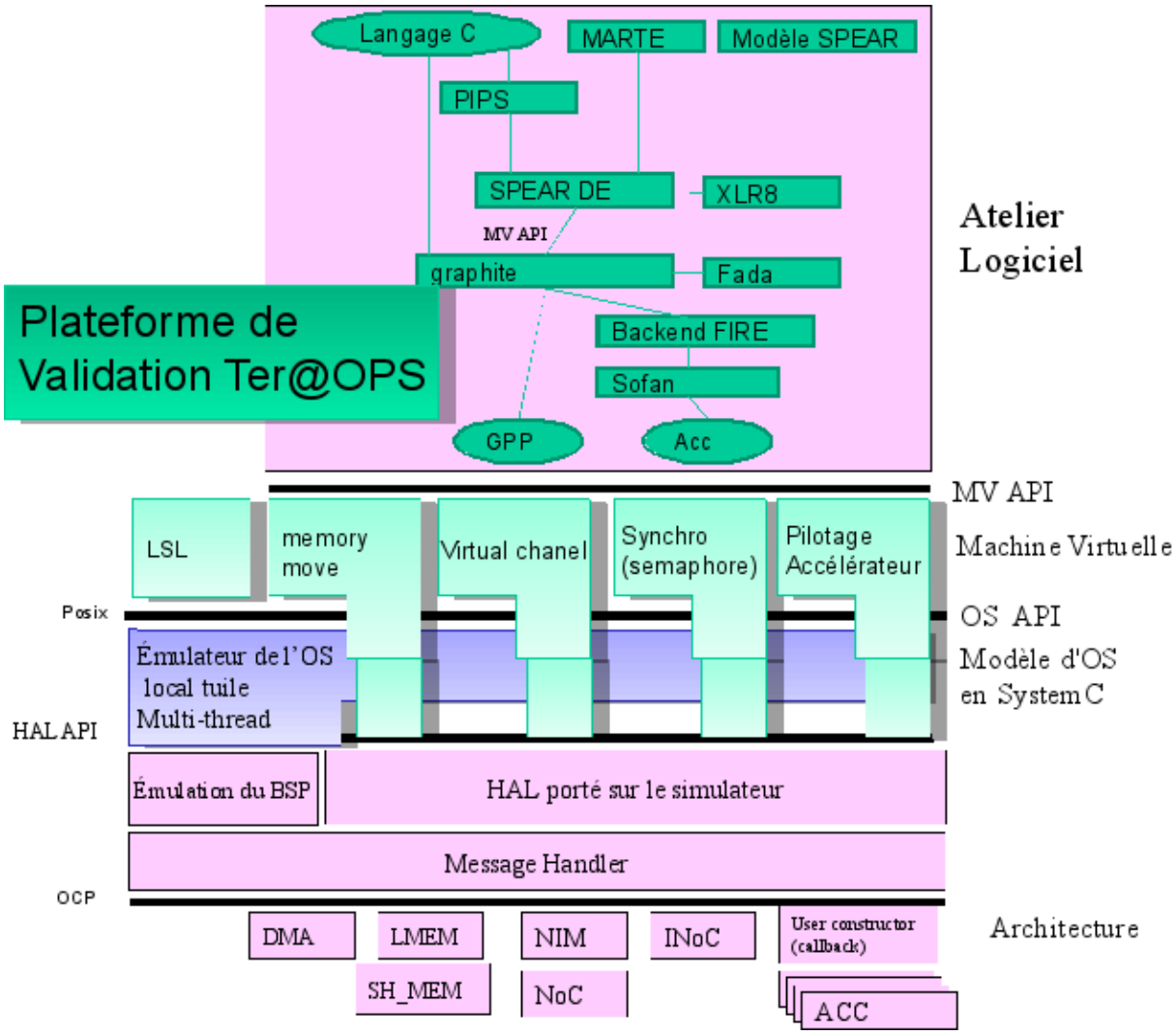


Figure 5.11: *Structure de l'architecture logicielle de Ter@ops en couches*

- Message passing : basé sur des échanges de message synchrones ou asynchrone entre les processus ; utilisé principalement dans un contexte dataflow statique.
- Mémoire partagée : basé sur des transferts de données d'une zone mémoire à une autre zone mémoire et utilisation de sémaphore/évènement pour synchroniser les processus ; utilisé principalement dans des applications dynamiques avec de fortes contraintes temps-réel.

– La machine virtuelle Ter@ops est indépendante du domaine d'application. En conséquence, les librairies d'opérateurs ne font pas partie du middleware mais sont activées de façon générique par la machine virtuelle.

5.3.4.2 Modèles de programmation de la MV

Les deux modèles de programmation cités ci-dessus sont décrits séparément dans les deux sections suivantes mais ne sont pas exclusifs.

Message Passing Il s'agit d'un modèle de programmation de type "processus communicants". Ce modèle est adapté aux architectures distribuées. Les objets manipulés par le programmeur sont des threads et des Virtual CHannel (VCH). Les VCH sont des queues de messages asynchrone gérées par la MV. Les synchronisations sont effectuées au moyen de messages uni-directionnels.

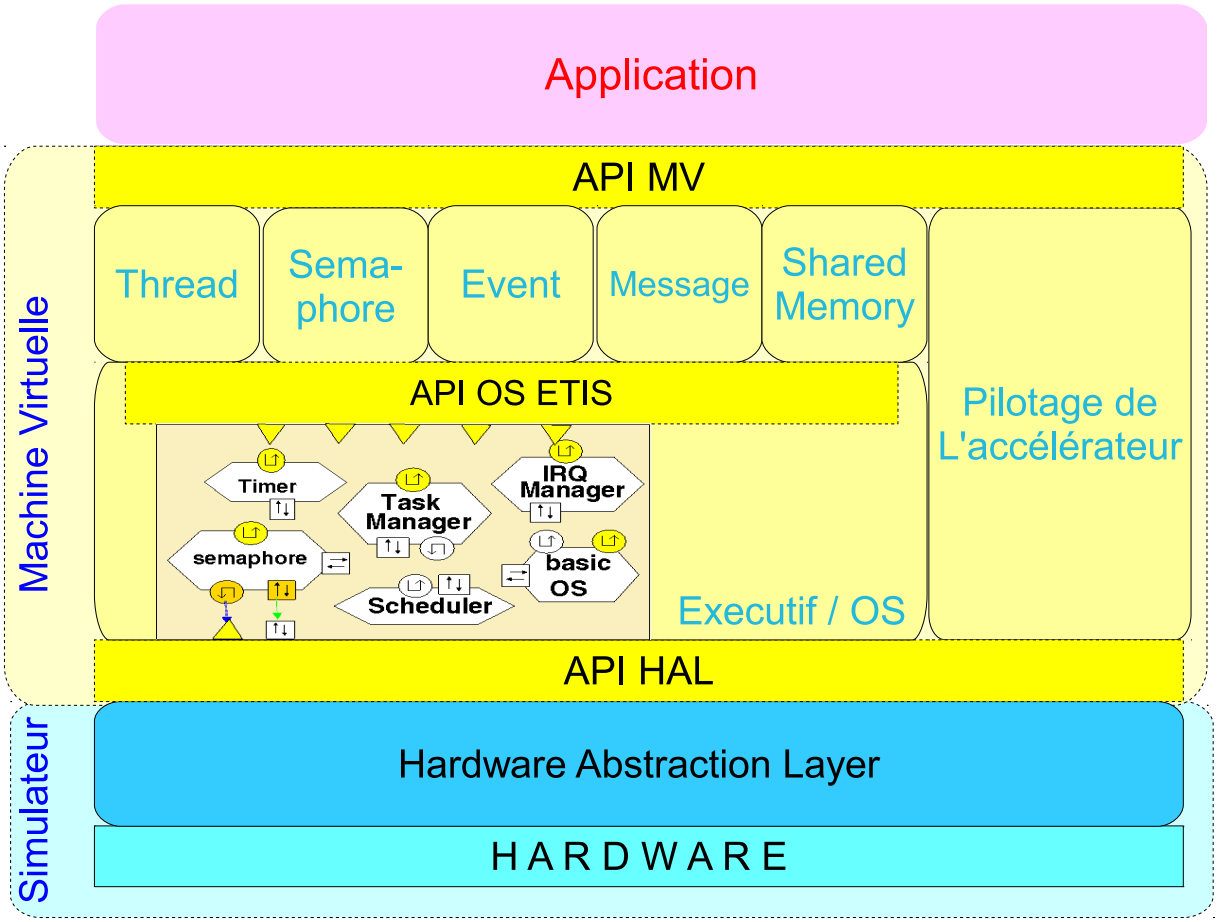


Figure 5.12: *La Machine Virtuelle, dont les services s'appuient sur le support d'exécution fournit par notre modèle d'OS*

Mémoire partagée Ce type de modèle nécessite des mécanismes de type evènementiels et de partage (gestion de conflit). Les objets manipulés sont : des threads, des objets de synchronisation et des variables partagées. La politique d'ordonnancement n'est pas imposée, cependant un ordonnancement préemptif est recommandé pour ce type de modèle de programmation. Les objets de synchronisation peuvent être des sémaphores ou des événements. La création dynamique de threads est supportée grâce à notre modèle d'OS sous-jacent.

5.3.4.3 Les services de la MV

Vue du programmeur, l'API de la MV est organisée en deux sous-ensembles : configuration (partie verticale de la figure 5.12) et runtime (partie horizontale de la figure 5.12).

Le placement est statique, et il est défini au moment de la compilation. Les objets manipulés (threads, canaux de communication et mécanismes de synchronisation) sont définis statiquement et instanciés au démarrage par le Configuration Manager (CM) :

- service d'initialisation,
- service de gestion des threads,
- service de gestion mémoire,
- service de gestion des synchronisations.

Seuls les services du sous-ensemble runtime sont utilisés par le programmeur :

- services de synchronisation,
- service Message Passing,
- service Timer,

– service de transfert mémoire,
– service de pilotage de l'accélérateur.

5.3.4.4 Construction de la MV sur le modèle d'OS en SystemC

La machine virtuelle avait besoin de services déjà présents dans le modèle d'OS, tels que le service de synchronisation par sémaphore, mais surtout la capacité à lancer une tâche périodique prioritaire de scrutation des files de messages. Elle s'appuie donc sur les services de simulation de notre OS et devient un simple relais vers les service d'OS lorsque les services sont déjà fournis par l'OS. En effet, étant dans un modèle multi-nœuds d'exécution, les services de synchronisation doivent absolument êtres pris en compte. L'arrivée d'une synchronisation s'apparente alors à une interruption externe. Cela n'est possible que dans un modèle où les tâches simulées sont préemptibles, ce que permet notre OS.

Un autre point a été l'intégration d'un sous-ensemble de l'API POSIX.1c (celle des threads), puisque les applications tests sont généralement déjà écrites pour cette API standard. Étant donné que le moteur de simulation SystemC (le kernel systemC) s'appuie déjà sur l'interface POSIX de l'OS (Linux dans notre cas) de la machine hôte, il n'est pas possible de spécifier une interface identique pour l'OS, car il y aurait un conflit lors de l'édition de liens. Aussi afin de clarifier et contourner le problème, les interfaces des services de l'OS sont identiques à celles de POSIX, mais précédés du préfixe `tera_` .

5.3.4.5 Intégration de l'OS dans l'architecture de communication OCP, raffinement du CAS

Le principal service nécessitant d'être intégré au simulateur pour fonctionner a été le service de sémaphore partagé, qui permet de synchroniser les threads applicatifs quelle que soit leur localisation sur les tuiles. Le principe reste le même, c'est-à-dire le modèle de distribution de nos services, avec une interface proxy qui gère la redirection de l'appel vers le nœud/tuile exécutant effectivement le service (le skeleton). Le modèle développé dans la thèse a été adapté au cas spécifique de Ter@OPS où les communications doivent passer par l'infrastructure du NoC de contrôle de l'architecture, et donc utiliser pour cela l'API HAL fournie.

Il a donc fallu faire des transacteurs entre les paquets de messages originaux (ceux du CAS) en paquets de types OCP. Pour cela nous avons dû ajouter un service d'identification de la tuile locale et distante afin de générer des adresses source et destination en conformité avec le plan d'adressage de la puce Ter@ops.

Il a aussi été nécessaire d'implémenter un *routeur d'évènements* : un thread spécifique de réception des paquets du réseau de contrôle afin qu'il lance une interruption déclenchant le traitant d'interruption qui décode le paquet et le redirige vers le service adéquat, en l'occurrence le gestionnaire de sémaphore de l'OS, ou bien le gestionnaire d'évènements de la MV. Dans le cas du sémaphore, on peut voir dans la figure 5.13 les redirection d'appels entre le nœud local proxy, vers le skeleton, qui lui même redirige ver le bon nœud du thread à libérer (si nécessaire). À chaque communication inter-nœuds, on crée un messages OCP contennant l'identifiant de la requette, du destinataire, ainsi que les données/paramètres de la fonction. Et si le service doit renvoyer une réponse, l'identifiant de la source doit aussi être transmis pour la réponse ultérieure. À la reception, l'interface du NIM déclanche l'interruption de traitement associé au service appellés. Le services sera ensuite traité selon les disponibilités d'ordonnacement du nœud de reception.

Une extension aurait pu être faite pour utiliser la largeur de l'espace d'adressage alouée au contrôleur de tuile afin que le NIM décode automatiquement la redirection du paquet

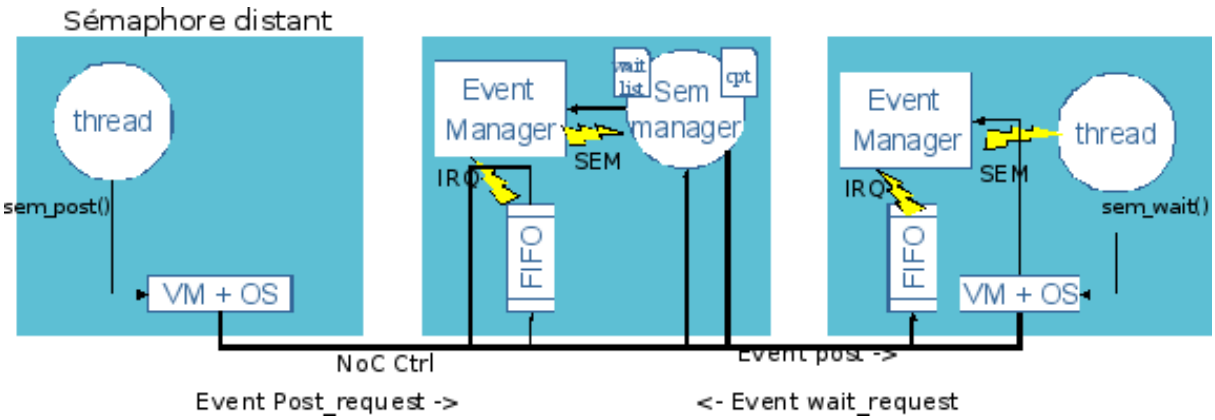


Figure 5.13: *Schéma d'appel distant pour relacher un sémaphore, qui est redirigé vers le nœud le gérant, et qui libère ensuite un autre thread distant bloqué en attente*

vers le bon service, comme le fait déjà le CAS interne, mais il aurait falu intégrer les services qui sont uniquement dans la machine virtuelle.

5.3.5 Gestion de la mémoire et implications pour la modélisation

Un point faible de notre modèle de gestionnaire de plateforme de haut niveau est la gestion mémoire. Ter@ops a permis de renforcer ces aspects de simulation et gestion de la mémoire dans un simulateur SystemC.

Modèle de la mémoire dans Ter@ops

La plateforme Ter@ops supporte deux modèles de programmation : un modèle de type “passage de messages” et un modèle de type “mémoire partagée”. Ces deux modèles sont généralement considérés comme faisant partie des modèles les plus courants dans les systèmes embarqués. Afin d'être suffisamment flexible pour supporter ces deux types de modèles de programmation avec des performances satisfaisantes, l'architecture Ter@ops regroupe :

- des mémoires partagées (NUMA) au niveau global pour un modèle de programmation par mémoire partagée ou pour servir de mémoires tampons de forte capacité pour les communications.
- des mémoires distribuées dans chaque tuile pour la "scalabilité" de l'architecture et une implémentation efficace du passage de message.

Ces mémoires font partie du même espace d'adressage global accessible par toutes les tuiles. Une des applications envisagées, l'encodage H264 HD, nécessitant 24 Go de données, il a été décidé de fournir un espace d'adressage sur 35 bits, soit jusqu'à potentiellement 32 Go de mémoire adressable.

L'adressage interne de Ter@ops doit intégrer les caractéristiques suivantes :

- l'espace d'adressage est unifié, grâce aux mécanismes proposés par le NoC : toutes les LMEM sont visibles par toutes les tuiles.
- les mémoires partagées internes (SHM) doivent avoir des plages d'adresses contigües (modèle NUMA).
- chaque tuile accède à une image de son espace d'adresses interne (LMEM, registres des périphériques internes à la tuile ...) à une adresse de base constante : 0x0000_0000.

Ces contraintes ont amené à élaborer un plan d'adressage global assez complexe permettant d'adresser chacun des sous-éléments des tuiles (mémoire interne, contrôleur,

DMA, accélérateur,...), mais aussi chacune des tuiles et mémoires partagées ou externes à la tuile.

Modèle simulé de la mémoire dans le simulateur

Il est évident que la plateforme de simulation sur PC ne peux pas gérer une mémoire aussi grande. On a fait le compromis de modéliser précisément uniquement les zones utilisées des mémoires partagées, ce qui a été rendu possible grâce à l'outil SPEAR DE [LE03] capable de donner la taille exacte des données utilisées ainsi que leur mapping en mémoire.

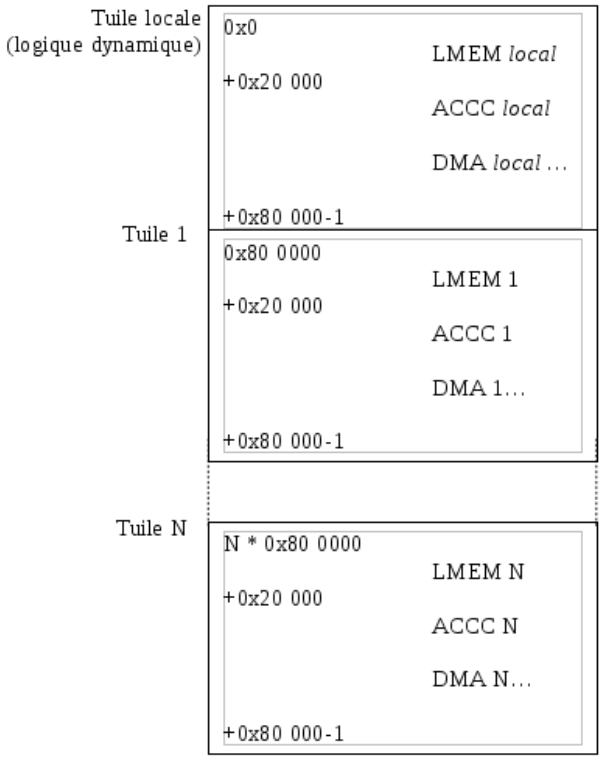


Figure 5.14: *Mapping mémoire des tuiles*

Par ailleurs, l'espace d'adressage local de chaque tuile Ter@ops (et donc les mémoires locales, cf. figure 5.14) manipule des adresses basses (entre 0 et 0x80000-1) et cela ne correspond à aucune adresse réelle valide dans la mémoire de l'OS support de la simulation (les adresses basses sont généralement réservées au BIOS ou à l'OS).

Le problème vient du fait que la HAL du CTRL et de l'architecture des tuiles et des interconnexions du simulateur fonctionnent tous avec des adresses logiques Ter@ops selon le plan d'adressage de Ter@ops ; alors que du coté applicatif, les tâches logicielles fonctionnelles s'exécutent avec des adresses physiques de l'hôte où tourne la simulation SystemC. La machine virtuelle prend donc en entrée de ses appels des adresses physiques et non pas des adresses logiques Ter@ops.

Il a donc fallu trouver une solution pour que les transferts de données simulés entrant et sortant des tuiles puissent être effectués dans l'espace d'adressage du simulateur avec une correspondance dans l'espace d'adressage Ter@ops pour bien les prendre en compte.

Solution développée pour prendre en compte les transferts mémoires au niveau Cycle Accurate, de manière optimisée

En effet, il s'agit de la question de la modélisation de la mémoire de Ter@ops en SystemC, et de la manière d'allouer la mémoire des threads applicatifs en adéquation

avec la mémoire hôte et la représentation coté simulation Ter@ops.

Par défaut, l'allocation (statique) d'une variable dans un thread applicatif (en C) se fait directement dans la mémoire de l'hôte (adresse virtuelle Linux). Cela ne pose pas de problème pour les variables "privées" d'un thread, par contre les variables/tableaux accédés par un thread d'un autre contrôleur de tuile et/ou par un DMA doivent se trouver dans une mémoire *partageable*, c'est-à-dire une mémoire locale (LMEM) ou une mémoire partagée (SHMEM), et par conséquent mappés dans l'espace d'adressage simulé de Ter@ops. Sans quoi, on ne saurait détecter et donc simuler les transferts mémoires.

La LMEM de chaque tuile est allouée en SystemC par un `new()` dans son constructeur. L'adresse "hôte" de cette réservation reste dans l'objet "LMEM", seule son adresse logique dans l'espace mémoire simulé est connue/fixée globalement puisque simulée selon l'adressage Ter@ops. Le DMA manipule des adresses dans l'espace mémoire logique Ter@ops (simulé), et le simulateur doit donc disposer d'un mécanisme interne pour les convertir en adresses physiques de l'hôte de la simulation.

Pour faire le lien entre les deux espaces d'adressage, il manquait deux choses :

1. un moyen de convertir les adresses physiques sur l'hôte en adresses logiques Ter@ops et inversement, pour que l'applicatif fonctionne en adresse physique de l'hôte et que le code reste fonctionnel, bien que la simulation de l'architecture fonctionne en adresse logique Ter@ops.
2. un mécanisme pour allouer/déclarer de la mémoire physique locale aux threads lié dans l'espace d'adressage logique Ter@ops.

La solution consiste à faire les conversions de système d'adressage physique/logique (PC/Teraops) dans les fonctions de la Machine Virtuelle, afin de garder toutes les fonctionnalités à partir de l'HAL sous forme logique.

Ainsi, les fonctions de l'OS et de la MV vont continuer de prendre en entrée des adresses physiques de l'hôte mais vont utiliser une fonction de conversion avant d'utiliser toute fonction de la HAL, puis convertir inversement les adresses de retour, de manière à garder les codes fonctionnels des threads simulées intacts.

Cela implique de pouvoir convertir les adresses physiques en adresses logiques. Or il est impossible en l'état de déterminer si une adresse PC (locale à un thread ou non) appartient à la tuile locale ou à une autre, puisque les adresses physiques des données de thread et des mémoires physiques n'ont aucune corrélation.

La solution mise en œuvre est la suivante. Les zones de données de thread susceptibles d'être manipulées vers ou depuis l'extérieur (via un DMA par exemple) sont allouées virtuellement dans la zone de mémoire physique PC allouée préalablement pour la LMEM locale. Ainsi, pour toute adresse PC manipulée, en supposant que les adresses PC début et fin de chaque LMEM sont globalement connues par le simulateur (tableau global ou fonction globale), il est facile de retrouver auquel des intervalles de plage mémoire LMEM physique il appartient, et donc à quelle mémoire/tuile logique Ter@ops cela correspond. On peut ensuite calculer la conversion facilement, en récupérant l'offset de l'adresse PC de la donnée par rapport à l'adresse PC début de la mémoire qui la contient, et s'en servir pour calculer l'adresse logique lui correspondant, selon le mapping schématisé en figure 5.15. De cette manière, les transferts de données sont faits dans l'espace physique, tout en demandant la simulation des temps de transfert au NoC avec des adresses logiques adéquates.

Pour cela les données utilisateur qui doivent être transférées depuis ou vers l'extérieur de la tuile sont déclarées non pas de la manière suivante :

```
char data[256];
```

mais de la sorte :

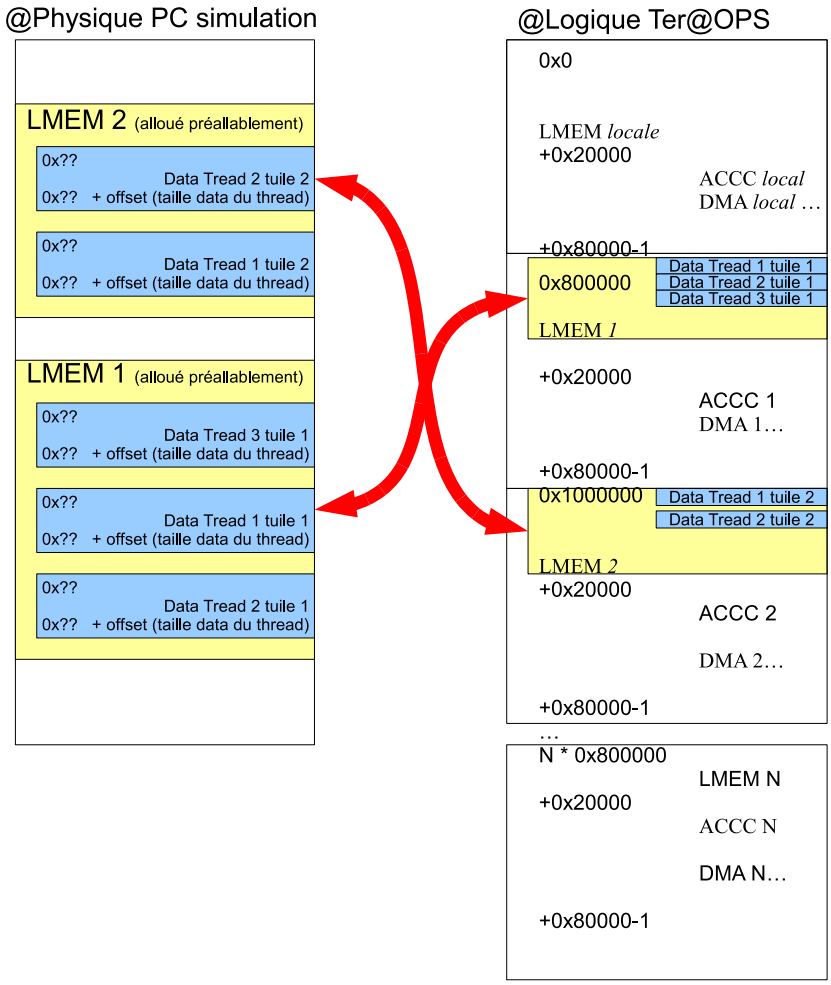


Figure 5.15: *Mapping entre adresses PC et adresses logiques aux tuiles Ter@OPS*

```
char *data = tera_malloc(256);
```

où ce pseudo `malloc` va demander et réserver le nombre d'octets dans la mémoire physique et logique de la LMEM de la tuile et lui renvoyer un pointeur physique qui est connu par construction comme faisant partie de l'espace de la mémoire de la tuile. Ainsi le code reste fonctionnel, et les fonctions de conversion seront utilisées pour faire fonctionner Ter@ops et ces composants avec des adresses logiques résolues selon que l'adresse physique est dans l'une ou l'autre des mémoires physiques allouées.

Cela force à modifier quelques peu le code de l'utilisateur, mais permet d'expliciter les variables partagées du code, ce qui peut servir à mieux concevoir le code et faciliter les travaux de vérification.

Bien évidement, il a fallu mettre en place des mécanismes pour gérer ces pseudo allocations de données et gérer des espaces de mémoire pour que chaque bloc de données soit alloué de manière contigüe et sans entrelacement. Et de la même manière, il a fallu aussi modifier les fonction de la MV ou de l'OS afin de permettre les conversions automatiques entre les addresses applicatives vers celles logiques du simulateurs et inverssement, comme on le représente dans la figure 5.16.

5.3.6 Conclusion

La programmation du simulateur Ter@ops se fait à travers un modèle de programmation homogène fournit par l'API de la Machine Virtuelle qui assure l'indépendance matérielle de l'application.

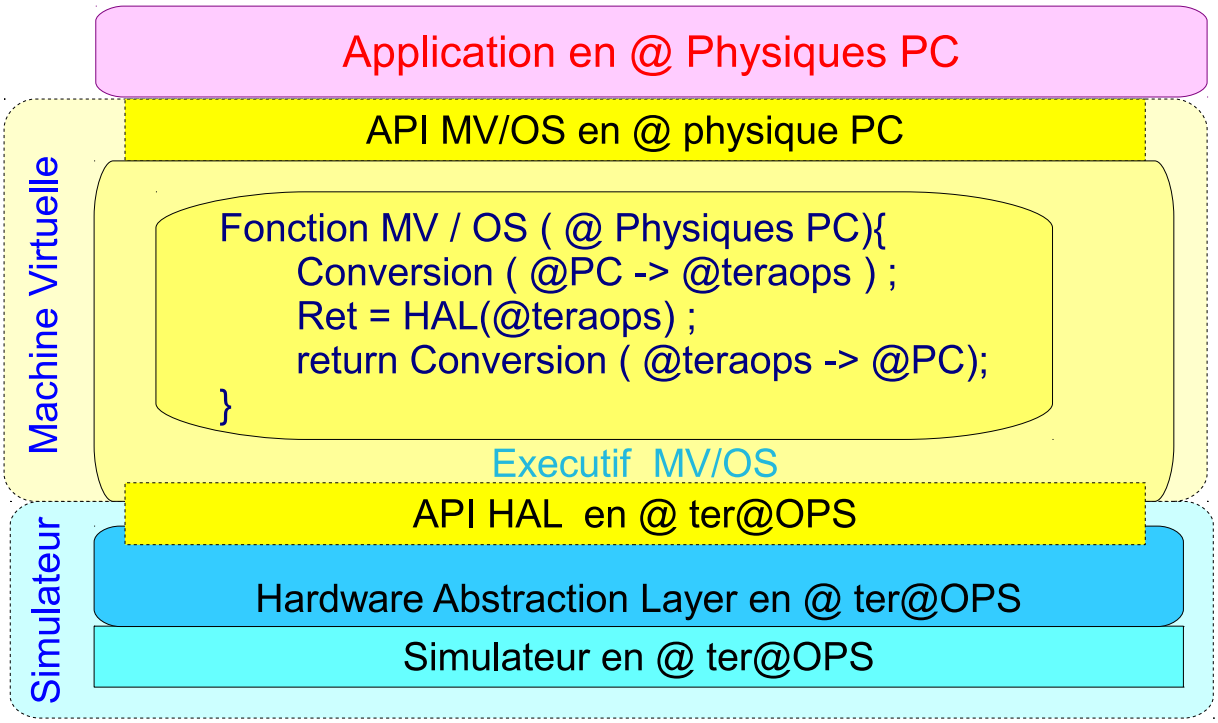


Figure 5.16: *Translation cohérente entre adresses fonctionnelles et adresses logiques du mapping Ter@OPS*

Le modèle d'OS est essentiel au simulateur Ter@ops pour simuler une application multi processus purement logicielle sans avoir besoin de raffiner les modèles des accélérateurs des tuiles. Ce modèle permet de simuler plusieurs threads sur un même nœud d'exécution. Cette propriété a permi de développer la Machine Virtuelle qui en elle-même nécessitait plusieurs threads pour s'exécuter convenalement, en plus de permettre tout simplment de faire des essais de validation de portage des différentes applications test du projet.

Le modèle de programmation par mémoire partagée nécessitant des mécanismes d'évènements et de synchronisations distribués, les services d'interruption et de sémaphore de type proxy/skeleton ont servi de base pour concevoir la Machine Virtuelle.

Par ailleurs, la réflexion menée pour simuler la mémoire de manière cohérente entre l'application et le simulateur SystemC nous ont permis de vérifier la capacité de notre modèle à s'étendre à une modélisation plus détaillée des échanges mémoires, au dépend d'une explicitation de ses derniers par notification d'appels système au DMA.

Enfin, l'intégration de notre modèle d'OS avec une infrastructure de communication complexe et détaillée des NoC de Ter@ops a permis de constater la robustesse de notre système CAS de gestion de redirection des appels système.

Pour conclure, notre modèle d'OS a démontré sa capacité à s'intégrer dans une simulation de matériel de niveaux hétérogènes, et a montré sa capacité à déployer et simuler facilement une application parallèle sur un système multi-cœurs, d'une manière rapide en évitant d'utiliser des ISS lents.

5.4 Conclusion du chapitre

Les expérimentations menées ont permis de valider le bien fondé de notre démarche en montrant qu'elle facilite l'exploration de la plateforme conjointement à l'exploration du gestionnaire de cette dernière. On a pu ainsi tester des niveaux de raffinement hétérogènes entre les éléments simulés, ainsi que la possibilité de connecter des éléments de simulation de zone reconfigurable. Cela a facilité l'exploration du découpage et la répartition des

tâches entre logiciel et matériel afin d'obtenir une architecture qui peut s'exécuter en temps-réel. L'outil Dogme devrait permettre l'ajout simple de nouveaux services de gestion ainsi que l'exploration simplifiée de nouvelles applications à répartir entre partie logicielle et zones reconfigurables. Les travaux au sein du projet Ter@OPs on par ailleurs permis de tester le raffinement des communications, en particulier celle liés aux transferts mémoires, ce qui permet d'obtenir une simulation plus précise en s'appuyant sur le simulateur de réseau sur puce. Cela correspond bien à notre modèle SAT mixant différents niveaux de précision, ceux de l'OS restant à haut niveau, mais d'autres pouvant être plus fin, que ce soit la partie applicative (avec un ISS ou un ARD) ou les communications.

Chapitre 6

Conclusion et perspectives

6.1 Bilan

Pour répondre à l'inadaptation des anciens modes de conception aux exigences des systèmes embarqués reconfigurables, nous avons développé, dans ces travaux un modèle de simulation conjointe centré sur l'OS permettant d'explorer l'architecture idéale d'une plateforme embarquée dédiée à une application dynamique, dans la perspective d'inventer et d'évaluer des services de gestion de la reconfiguration partielle et dynamique des architectures de type FPGA.

Pour cela, il était nécessaire de disposer d'un outil de co-simulation complet et rapide, donc de haut niveau d'abstraction, permettant d'évaluer différentes combinaisons logicielles et matérielles susceptibles d'être adéquates et de vérifier que le système remplira ses objectifs.

Nous avons donc cherché à concevoir un modèle exécutable réaliste capable d'embrasser l'ensemble des aspects d'un système-sur-puce, comprenant les parties logicielles et matérielles, y compris son système d'exploitation.

Pour cela, la solution retenue a été de se focaliser sur le système d'exploitation et de le percevoir comme un gestionnaire de plateforme composé d'un ensemble de tâches et de modules de services distribués, modifiables et interchangeables, et pouvant se raffiner indifféremment sous forme logicielle ou sous forme d'accélérateurs câblés.

Tout d'abord, nous avons focalisé notre travail pour être capable de simuler conjointement le logiciel, le matériel, et le système d'exploitation, qui gère dynamiquement le flot d'exécution. Cette possibilité a été établie en s'appuyant sur le langage SystemC, auquel nous avons rajouté une surcouche autorisant la simulation à haut niveau de tâches logicielles conjointement aux parties matérielles distribuées. Ce modèle est capable de gérer la simulation du logiciel de manière fonctionnelle et temporelle, en tenant compte des interruptions grâce à un mécanisme de simulation de la préemption.

Ensuite, nous avons étendu ce modèle à la simulation de nœuds d'exécution distribués. Pour cela nous avons élaboré un module SystemC nous permettant d'organiser les transferts de données et services, de sorte que l'on puisse interconnecter un très grand nombre de nœuds d'exécution.

Puis, pour faciliter l'exploration des services de gestion de plateforme distribués, nous avons défini un modèle générique permettant de créer, combiner et répartir facilement les différents services personnalisés de gestion d'un système sur puce, permettant d'explorer différentes combinaisons par cycle de simulation court.

Nous avons pu valider le bon fonctionnement du modèle sur un cas concret de conception de plateforme dédiée à une application de vision robotique. Les aspects de co-

simulation avec des blocs hétérogènes (en terme de type de nœud d'exécution et de communication ou de niveau de précision modélisé) ont pu être expérimentés durant deux collaborations dans des projets de recherches, Ter@ops et OveRSoC.
Cela nous a permis de constater que notre modèle autorise à explorer non seulement les comportements fonctionnels mais aussi les potentielles implémentations et répartitions des tâches (matérielles ou logicielles), mais encore et surtout les services d'OS à personnaliser et à composer selon les besoins d'une application.

En conclusion, nous pouvons dire que notre modèle est utilisable comme un moyen de concevoir et élaborer un système sur puce dédié à une application, ce qui est facilité à l'aide de l'outil graphique Dogme. Le modèle permet de discriminer les différents choix possibles en facilitant l'exploration et l'évaluation de l'architecture logicielle et matérielle adéquate, conjointement au système d'exploitation adapté, par étapes successives de raffinement.

Notre objectif d'autoriser à la fois l'exploration algorithmique mais aussi l'implémentation mixte des tâches et des services de SE tout en autorisant l'exploration de leur distribution sur de multiple nœuds hétérogènes a été rempli.

L'avantage notable de ce modèle est qu'il facilite grandement le prototypage rapide d'applications et facilite sa parallélisation en de multiples nœuds d'exécution concurrents sans obliger à entrer immédiatement dans les détails de bas niveau architectural de l'implémentation.

De plus ce modèle générique permet de concevoir et d'explorer des services spécifiques dédiés à la gestion d'accélérateurs reconfigurables, pour lesquels des problèmes d'ordonnancement spatial et temporel doivent être réglés.

Nous avions choisi une approche consistant à prendre le point de vue du gestionnaire de plateforme pour concevoir un système distribué. Ce choix s'avère judicieux pour la phase initiale d'exploration conjointe logicielle/matérielle, mais s'appuie sur la condition préalable de posséder des estimations des temps d'exécution des différentes tâches.
Par ailleurs, le choix de faire une abstraction de haut niveau des nœuds de calcul risque de donner des évaluations trop fortement tronquées à propos de mesures des échanges de données, qui peuvent alors avoir un rôle prépondérant sur l'exécution réelle dans un système distribué pour des applications fortement communicantes. Il faudrait alors peut-être étendre le concept de SE ou de gestionnaire de plateforme pour mieux prendre en compte dans le processus de conception la gestion des transferts de données entres mémoires et nœuds de calcul.

Des améliorations peuvent être proposées en perspective de ces travaux de thèse, notamment pour palier le besoin de mesures ou d'approximations comme préalable à l'utilisation de notre modèle.

6.2 Perspectives

À partir de notre modèle générique, on pourra développer une bibliothèque de services plus variés, pour permettre un plus grand choix d'exploration algorithmique, en particulier ceux des ordonnanceurs qui jouent un rôle important pour répartir la charge entre les nœuds de calcul, notamment pour des raisons de consommation d'énergie en décidant de mettre en veille certains accélérateurs lorsque cela est possible.

Plus particulièrement, il serait intéressant d'étendre les fonctionnalités liées à la gestion de zones reconfigurable, voire éventuellement des ordonnanceurs mixtes, capables de jouer à loisir plusieurs versions d'une même tâche, en logiciel (processeurs hétérogènes) ou en matériel (formes et latences variables).

Afin de permettre au modèle de faciliter la conception, la phase de raffinement permettant de passer du modèle au code réel de l'OS devrait être étudié pour en permettre une certaine automatisation, comme certains le font entre les services SystemC modélisés et les services d'OS logiciels déjà existants.

Dans cette perspective, on pourrait aussi aborder le raffinement des services d'OS en matériel, comme certains services de gestion mémoire. L'intégration de services câblés ne devrait pas poser de problèmes, car les soft-cores sont prévus pour être configurés pour ajouter de telles fonctionnalités. Cela pourrait même amener à concevoir des processeurs incluant tous les services d'OS de manière câblée, afin de faciliter le respect des contraintes temps-réelles, même si les programmeurs de systèmes embarqués aiment bien pouvoir changer d'ordonnanceurs au fur et à mesures de l'évolution de leurs besoins

Il serait intéressant aussi de permettre de mieux évaluer les aspects de transferts mémoires, qui n'ont été abordés que sous la forme de services explicites dans le projet Ter@ops, ce qui nécessite une annotation manuelle du code, et donc sa modification. La difficulté réside dans l'identification des opérations qui vont générer un transfert mémoire, ce qui n'est pas du tout évident avec des langages de haut niveau tel que le langage C, car le mapping mémoire n'est pas explicite.

Dans un souci de facilitation de l'intégration d'application avec le modèle, il est nécessaire de permettre une annotation automatique des estimations des durées d'exécution des blocs de code, travail long et fastidieux si l'on veut éviter d'utiliser un ISS lent à cette fin. On pourrait pour cela s'inspirer des travaux de Posadas [PQVM07] à ce sujet.

Enfin, une automatisation de l'exploration pourrait aussi être envisagée, ne serait-ce que pour trouver le nombre minimal ou idéal de nœuds de calcul nécessaires pour exécuter en temps-réel une application, bien que ce problème d'optimisation multi-critères soit complexe. Notre modèle pourrait être ainsi la brique fondamentale à un outil de design, d'analyse et d'optimisation pour l'exploration architecturale des futurs *many-cores* hétérogènes.

Annexe A

Service de sémaphore partagé autonome

A.0.1 Contexte et problématique

Le Co-Design d'un OS en logiciel **et** en matériel est une problématique grandissante surtout pour des applications temps réel. En effet la gestion de services d'OS en matériel, et donc en parallèle permet de rendre un OS temps réel (RTOS) plus déterministe, d'améliorer la gigue (variation du temps de réponse) et d'augmenter la granularité du scheduler [APA+05]. De plus, l'implémentation matérielle des mécanismes de gestion des ressources partagées permet de répondre aux problématiques liées au parallélisme de tâches distribuées sur différents nœuds d'exécution hétérogènes.

La synchronisation est un des principaux points à optimiser dans une application distribuée pour pouvoir jouir au maximum du parallélisme. Il existe plusieurs outils de synchronisation adaptés à différents contextes, précisément décrite dans [LG07] [SPSI08]. Ces outils peuvent être implémenté en logiciel ou en matériel[RG02]. Le service de sémaphore distribué est une des possibilités offertes pour résoudre les problèmes de ressources partagées ou de synchronisation dans le contexte d'un système multi-nœuds d'éxécution hétérogènes.

Le terme de sémaphore et un nom générique pour le concept d'un système de gestion de jetons (qui symbolisent une/des ressource), avec gestion d'une file d'attente des tâches devant attendre qu'une autre tâche libère au moin une ressource. Le sémaphore est donc un compteur à N jetons (N peut être égal à 1 dans le cas d'un sémaphore binaire). Un service de sémaphore réalise les opérations de lecture, de test et de mise à jours de la valeur d'une variable (le nombre de jeton) en une opération atomique. Cette fonctionalité de *test & set* permet ainsi d'être sûr que le processus n'est pas interrompu au milieu de ces opérations dans le cas de système dit multi-processus, simulé (temps partagé) ou réel (multiprocesseur).

Selon le nombre de ressources modélisée dans un sémaphore, on distingue plusieurs cas d'utilisation. Le Mutex est l'élément de base pour protéger l'accés à une ressource unique (1 jeton). Il est possible de l'utiliser un sémaphore comme verrou (lock) ou comme indicateur (flag) selon le nombre de ressources initialisées. La barrière de synchronisation fonctionne avec un sémaphore initialisé $-N$ ressources et permet d'attendre N processus au pointde rendez-vous temporel.

Les sémaphores sont des outils assez bas niveau qui peuvent être source de problèmes d'inter-blocages (deadlock) qui peuvent survenir à cause d'une erreur de programmation. Cela se produit lorsque deux processus A et B attendent respectivement deux sémaphores Sem1 et Sem2 pour libérer ces mêmes sémaphores dans l'ordre inversé Sem2 et Sem1. Un autre problème bien connu à propos des sémaphores concerne l'inversion de priorité, qui peut se produire lorsqu'il y a accès à une même ressource par des tâches de haute et

de basse priorité. Si une tâche de basse priorité utilise la ressource, une tâche de haute priorité ne pourra pas prendre la main. Notre solution matérielle ne prévient pas de ces genres de problèmes qui peuvent être détéctés au préalable.

Dans notre thématique qui consiste à déporter des services d'un OS en matériel pour bénéficier ainsi du gain de performance apporté par le parallélisme d'un tel choix d'implémentation, le sémaphore est un service particulièrement intéressant à déporter en matériel puisqu'il présente une alternative intéressante pour résoudre les problèmes de ressources partagées ou de synchronisation entre nœuds d'éxécution hétérogènes.

En coopération avec un étudiant de l'ENSEA, Vincent Alvo, nous avons donc développé une Intellectual Property (IP, bloc matériel) réalisant la fonction de sémaphore, c'est-à-dire un sémaphore utilisé par plusieurs processus/tâches distribuées sur plusieurs processeurs/nœuds . Alors qu'une solution logiciel utiliserait la couche de communication inter-processeur, notre solution matérielle utilise une ressource indépendante et accessible par tous les processeurs/nœuds et peut communiquer directement avec chacun d'eux par interruption. Nous espérons ainsi raccourcir les temps de requêtes de sémaphore et surtout rendre ce temps beaucoup plus déterministe pour des applications temps-réel, c'est-à-dire avec des contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Ainsi la fonctionnalité classique du sémaphore de protection de ressource partagée pourrait être étendue à une fonction de synchronisation pratique et temps-réel entre tâches distribuées, que ces dernières soient logicielles ou matérielles.

Problématique liée au parallélisme spatial et temporel

C'est en général l'OS qui gère les objets sémaphores. Dans le cas mono-processeur, l'OS aillant une connaissance omnisciente de l'objet, il peut facilement re-activer les processus en attente. Le problème se pose dans le cas multi-processeur, ou chaque processeur est utilisateur d'un objet global. Il est donc possible de 1) tester en continu les objets (pooling) 2) utiliser la couche de communication inter-processeur de l'OS (MPCI pour RTEMS), pour une solution purement logiciel 3) utiliser un module matériel qui averti de sa propre initiative les processeurs.
Une autre méthode dite de synchronisation spéculative [MT02] propose un autre concept pour accéder à des ressources partagées. L'exécution du code n'est plus bloqué, il se poursuit et les conflits ou erreurs (si ils existent) sont détectés a posteriori.

Pour implémenter une solution de synchronisation par sémaphore sur multi-processeurs/nœuds, on à le choix entre une solution logicielle ou matérielle.

La solution logicielle doit informer en permanence tout les OS de chaque agissement sur le sémaphore Chaque micro-action entraîne une nécessaire lourde mise-à-jour sur toutes les unités, ce qui surcharge la communication inter-processeur.

La solution matérielle consiste en un (ou plusieurs) objet physiquement indépendant (IP). Chaque processeur/nœud se contente d'une connaissance **locale** et l'IP garde la connaissance **globale**. Cela réduit et optimise les communications inter-nœuds, tout en étant facilement extensible à N unités. La solution envisagée serait donc une IP accéssible par tous les OS via le bus système, l'appel et le retour de traitement se faisant par la lecture/écriture d'un registre de l'IP, et pour donnner l'information de la libération d'un sémaphore, suite à une mise en attente, l'IP reveille par intérruption le processeur de l'OS concerné (et uniquement ce dernier), comme on peut le voir dans la figure A.1.

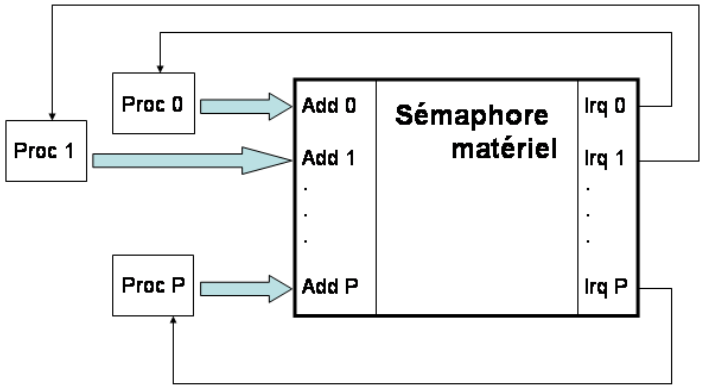


Figure A.1: *Principe fonctionnel du sémaphore matériel*

A.1 L'IP Sémaphore matériel

Dans un contexte d'application multiprocesseurs, le service de sémaphore inclut un protocole pour protéger l'accès à des ressources partagées. Il utilise une opération atomique de *test & set* pour donner ou non le sémaphore à une tâche. Si la ressource n'est pas disponible, le processeur concerné peut endormir la tâche concernée.

Des compromis ont été pris lors de la réalisation effective du module. La principale problématique concerne la répartition des fonctions entre le logiciel et le matériel. Nous avons choisi de ne porter en logiciel que le nécessaire pour éviter les communications inter-processeur[WTS96].

Le scheduling est assuré par chaque processeur, qui peut ainsi suivant les besoins facilement adapter ses règles. Il peut donc au choix rendre la main à la tâche le plus prioritaire d'abord, ou alors suivant un politique de premier arrivé premier servi (FIFO). La question de la politique de priorité c'est aussi posée au niveau global. Faut-il servir d'abord une tâche très prioritaire sur le processeur 1 ou une tâche peu prioritaire en attente depuis longtemps sur le processeur 2. [LS95] nous démontre qu'une simple gestion FIFO donne des meilleurs résultats pour une scheduling temps-réel sur un sémaphore global. En effet, l'ordre des priorités d'un sémaphore global doit être indépendant de la priorité des d'exécution des tâches.

Par ailleurs, pour éviter à la partie logicielle de faire du poling (lecture répétitive) pour détecter la libération d'un sémaphore, l'OS n'a pas à tester la disponibilité de la ressource, une interruption lui est envoyé dès la libération d'un sémaphore concernant une tâche résidant su ce processeur.

A.1.1 Description fonctionnelle

Le bloc de sémaphore matériel dispose d'une interface au bus AMBA (APB) ou Avalon esclave qui donne accès à des registres de contrôle. Le tableau A.1 présente les registres d'*un* sémaphore parmi **N**.

Un sémaphore est accessible par chaque processeur par l'intermédiaire d'un adresse unique à chaque processeur. Chacune des adresses pointe sur le même objet sémaphore. C'est ce qui permettra par la suite de déterminer quelle est la ligne d'interruption à activer pour réveiller le bon processeur ayant une tâche en file d'attente à libérer.

– Pour initialiser le sémaphore à une valeur **X** quelquonque, n'importe quel processeur peut écrire dans le registre Init Value la valeur **X**. Celle-ci peut être un valeur

Offset	Register Name	R/W	Bit Description (32 bits)
0	Read Sem value	R	VALUE
1	Init Value	W	VALUE
2	Sem Post	W	DON'T CARE
3	Sem Pend Processor 1	R	VALUE
4	Sem Pend Processor 2	R	VALUE
...			
P + 2	Sem Pend Processor P	R	VALUE

TABLE A.1: *Register map d'*un *sémaphore*

Offset	Semaphore Name
0	Registres de contrôle des Irq
K	Semaphore 0
2 * K	Semaphore 1
3 * K	Semaphore 2
...	
N * K	Semaphore N

TABLE A.2: *Adresse de base des N sémaphores du module*

signé sur 32 bits. Toutes les interruptions sont alors remises à zéro et la file d'attente est vidée.

– N'importe quel processeur peut alors faire une requête de sémaphore en effectuant une lecture du registre SEM PEND PROCESSOR P lui correspondant. Le bloc matériel s'occupe d'effectuer l'opération de *read - modify - write*. La valeur retournée indique la disponibilité de la ressource.

  La valeur est supérieure ou égale à zéro : le sémaphore est pris.

  La valeur est négative : la ressource n'est pas disponible, le numéro du processeur ayant passé la requête est enregistré dans la file d'attente. Une interruption sera générée sur ce processeur dès qu'un sémaphore sera libéré. Il devra alors lire dans le REGISTRE DE CONTRÔLE DES IRQ quel sémaphore est libéré, A.3. La lecture du registre re-arme l'interruption du processeur correspondant.

– Tous les processeurs rendent le sémaphore à la même adresse. Il suffit d'écrire n'importe quelle valeur dans le registre SEM POST.

– A titre informatif, il est possible de lire la valeur du sémaphore dans le registre READ SEM VALUE.

Le module contient **32** sémaphores distincts. Les valeurs et les files d'attentes sont séparées. Les différents sémaphores sont accessible aux adresses de base disponibles dans la table A.2. Le nombre **K** correspond à l'occupation d'un sémaphore dans l'espace d'adressage. Il dépend directement du nombre de processeur utilisé.

$$K = NbProcesseur + 3$$

A.1.2 Description VHDL

Nous avons séparés le bloc de sémaphore partagé en plusieurs fichiers de conception VHDL pour faciliter sont développement :

Offset	Register Name	R/W	Bit Description (32 bits)
0	Sem Flag register Processor 0	R	SF31 SF30 SF29 ... SF1 SF0
P - 1	Sem Flag register Processor P	R	SF31 SF30 SF29 ... SF1 SF0

Table A.3: *Registres de contrôle des Irq*

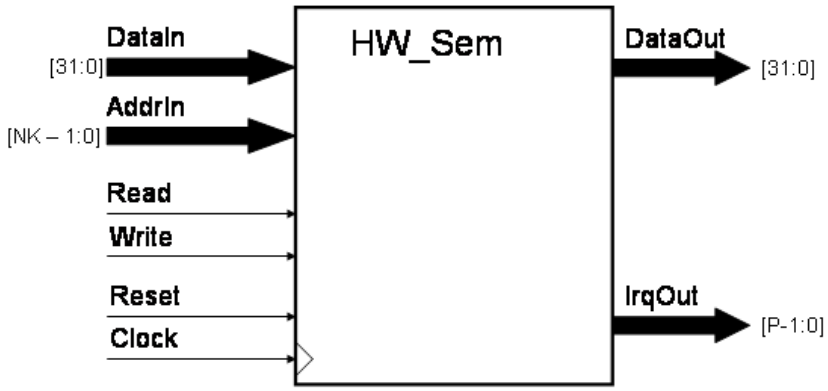


Figure A.2: *Entity du bloc HW_Sem*

- Hw_sem.vhd : description fonctionnelle du bloc sémaphore. C'est le fichier principal du module de sémaphore. L'*entity* est représenté dans la figure A.2.
- Hw_sem_testbench.vhd : Test bench de test du fonctionnement.
- Hw_sem_amba.vhd : Bridge pour adapter au bus AMBA (ARM, LEON). Voir section A.1.3.
- Hw_sem_Avalon.vhd : Bridge pour adapter au bus Avalon (NiOS).

A.1.3 Intégration au LEON 3

Le Leon3 est un soft-processeur open source développé par la compagnie Gaisler. Il a ainsi l'avantage d'être multiplateforme. Il est basé sur un bus AMBA (Advanced Microcontroller Bus Architecture), initialement développé pour les processeurs ARM et qui est devenu un standard pour les architectures 32bits. L'architecture proposée par gaisler permet d'inclure jusqu'à 6 processeurs Leon3. Nous avons choisi d'intégrer le module de sémaphore sur le bus périphérique APB.

Bus AMBA

L'architecture est donc basé sur autour du bus AMBA. Celui-ci se compose de deux bus, l'ASB/AHB et l'APB (voir figure A.4).

- Advanced System Bus (ASB) ou Advanced High-performance Bus (AHB)

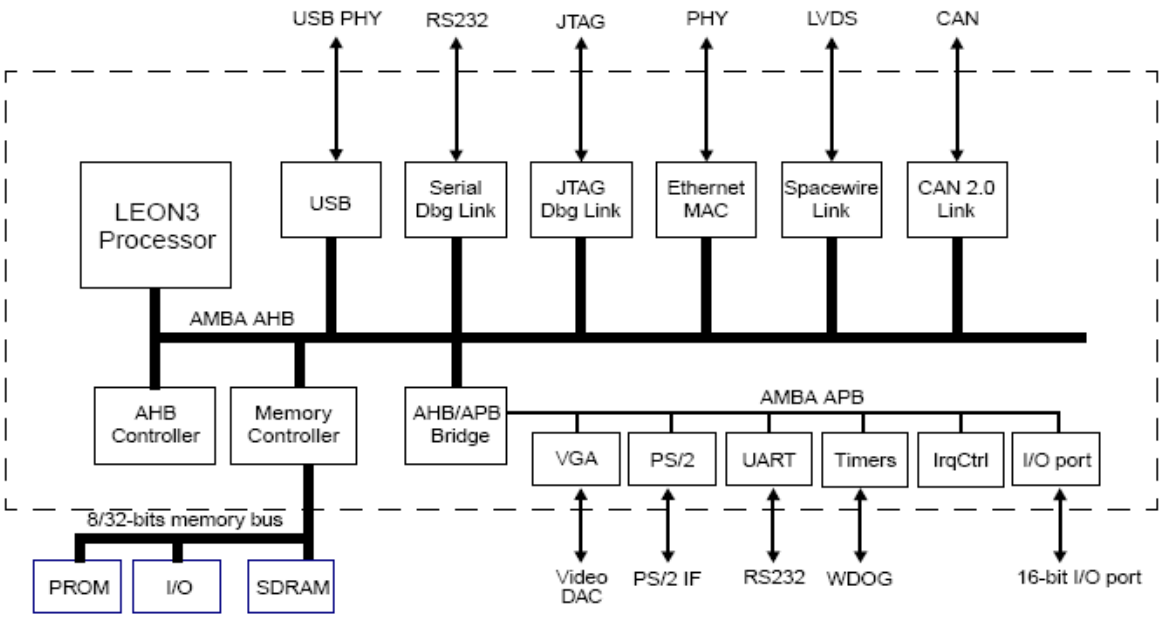


Figure A.3: *Architecture LEON typique*

– Advanced Peripheral Bus (APB)

L'ASB ou AHB est généralement utilisé pour des liaison rapide, hautes performances, multi-maîtres. L'APB est utilisé pour des liaisons basse consommation, destinée au contrôle avec une interface simple [Cor99]. Nous avons donc décidé d'interfacer le module de sémaphore au bus APB, puisqu'il ne s'agit que de simple lecture/écriture de mot de 32bits.

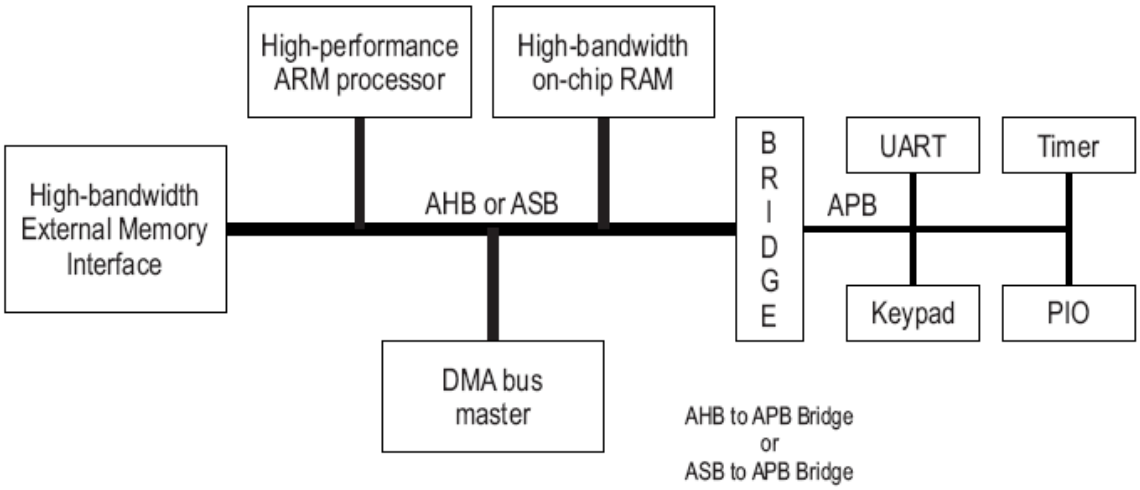


Figure A.4: *Système AMBA typique*

L'interfaçage de l'IP sur l'APB nécessite une légère mise en forme des signaux pour les adapter au format de la figure A.5.

– Les constantes NAHBIRQ et NAPBCFG sont défini dans le fichier *amba.vhd*.
– Le bus de sortie `pRData` est connecté en direct et seulement au maître APB. Il n'a donc pas besoin d'être 3-états.
– Les différents signaux de Plug & Play servent uniquement au maître à détecter les périphériques dans la phase d'initialisation.

L'APB gère automatiquement l'adressage des périphériques sur le bus, en générant les Chip Select (pSel) adéquat grâce à un scanne automatique du bus lors d'une phase d'initialisation. Cette fonction de Plug & Play facilite grandement la gestion haut niveau des

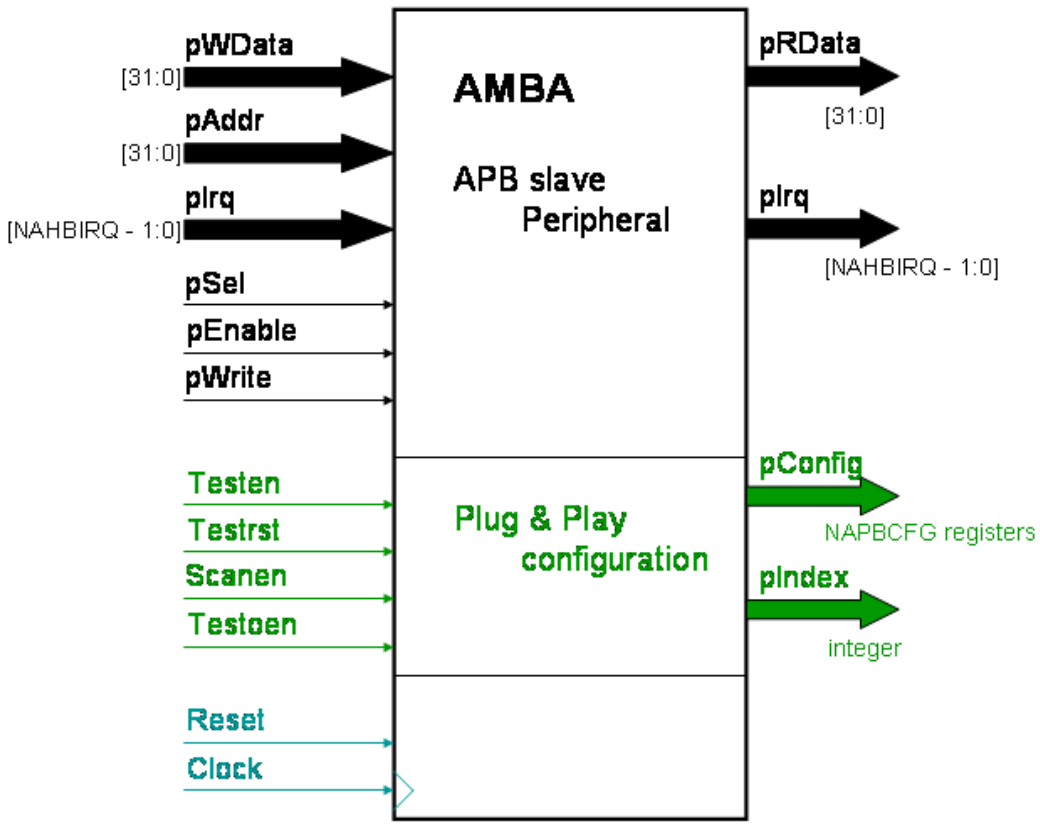


Figure A.5: *Format standard d'un périphérique APB*

périphériques. Du point de vue de l'IP, il s'agit de signaux complètement indépendantes qui n'interfèrent pas avec la fonction, même dans l'adressage. A noter que cette allocation n'est pas dynamique, la table des périphérique est crée à la compilation et est sauvegardé en mémoire.
Il est donc nécessaire de définir deux registres contenants les informations utiles pour mapper le périphérique. Ces deux registres sont présentés en figure A.6.

- L'identification register : contient le nom du vendeur de l'IP, la description de l'IP, la version ainsi que le numéro du (ou du premier) fil d'interruption utilisé.
- Le registre d'adresse : cotient l'adresse d'accès, la taille de la zone mémoire et un champ permettant d'utiliser le bus de donnée en 8, 16 ou 32 bits.

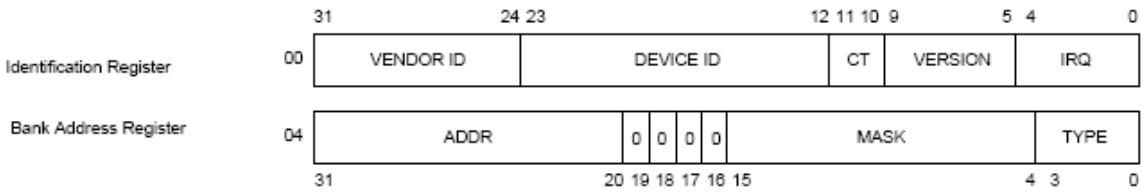


Figure A.6: *Registres de configuration du système de Plug & Play de l'APB*

Ajouter une IP au LEON3

Voici un résumé des opérations à effectuer pour ajouter une IP à l'architecture du LEON3.

1. Écrire, tester et valider le module selon le cahier des charges. Voir section A.1.
2. Adapter l'IP au format du bus AMBA. Voir section A.1.3
3. Linker l'IP au LEON3, au fichier *leon3mp.vhd*.
   Pour cela il faut ajouter le *component* à l'*entity* du LEON3, puis relier les fils du bus et définir les constantes.

```
----------------------------------------------------------------------------------------------------------
-- APB Bridge and various peripherals
semcore1 : apbsemcore
generic map (pindex =>14, paddr =>14, pmask =>16#FFF#, pirq =>19, sizeIRQ =>2, FIFOsize =>16, NbSem =>32)
port map (rstn, clkm, apbi, apbo(14));
----------------------------------------------------------------------------------------------------------
```

   - `pindex` [0 :NAPBSLV - 1] : Trouver une place libre sur le bus. NAPBSLV correspond au nombre de périphérique maximal.
   - `paddr` : adresse de base pour accéder au périphérique. L'adresse complète sur 32 bits correspondrait à 0x—XXX00, ou — correspond à l'adresse du bridge APB sur le bus AHB, XXX correspond au 12bits de paddr et 00 correspond à zéro.
   - `pmask` : indique le nombre d'adresse disponible pour le périphérique. On dispose au maximum de 0xFFF adresses, soit 4096 mots.
   - `pirq` : indique le numéro du ou du premier fil d'interruption branché au périphérique. Un bridge APB dispose de 32 fils d'interruptions pour l'ensemble de ses périphérique.

4. Inclure les fichiers .vhd de l'IP dans le makefile. Ajouter à cette ligne les fichiers .vhd de l'IP.

   ```
   VHDLSYNFILES=config.vhd ahbrom.vhd svga2ch7301c.vhd leon3mp.vhd
   ```

A.2 Résultats

Des résultats significatifs ont été apporté au projet. Nous avons 1) réalisé le module de sémaphores materiels en VHDL, testé et validé fonctionnellement 2) développé l'interface aux bus AMBA et Avalon 3) monté une architecture à base de LEON3 incluant le module, qui a bien été détecté par les outils gaisler.
Pour obtenir des résultats intéressant et quantifié sur l'intérêt du module de sémaphores, il faudrait encore 1) développer les drivers pour RTEMS 2) réécrire les appels systèmes aux fonctions de sémaphore de RTEMS 3) mesurer et comparer les temps d'accès et de synchronisation avec la solution logiciel utilisant la couche de communication inter-processeur MPCI.

A.2.1 Module de sémaphores matériel

Un test bench nous a permis de valider le fonctionnement des sémaphores. Le test bench s'applique au module logique et n'inclut pas les bridges d'adaptation aux différents bus.

A.2.2 Interface aux bus AMBA et Avalon

L'interface au bus Avalon a été presque immédiate, dans la mesure où le modèle d'interface du sémaphore logique a été très largement inspiré par le bus Avalon. Il ne

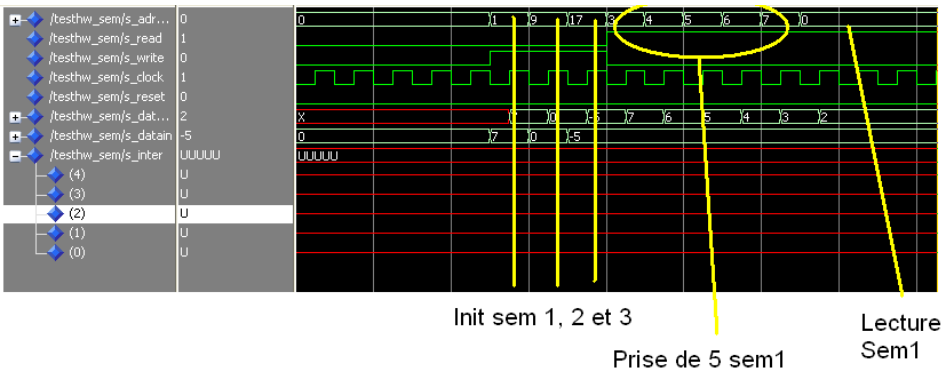


Figure A.7: *Initialisation et prise de sémaphores*

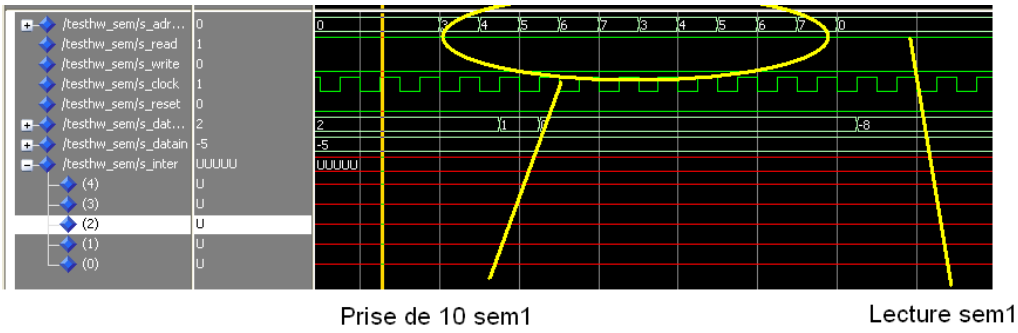


Figure A.8: *Demande jusqu'à vider le sémaphore*

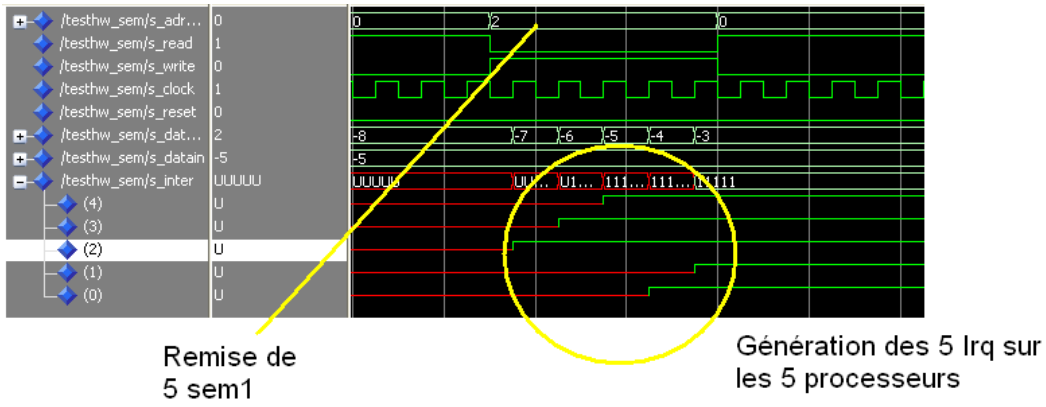


Figure A.9: *Post de sémaphore et génération d'irq*

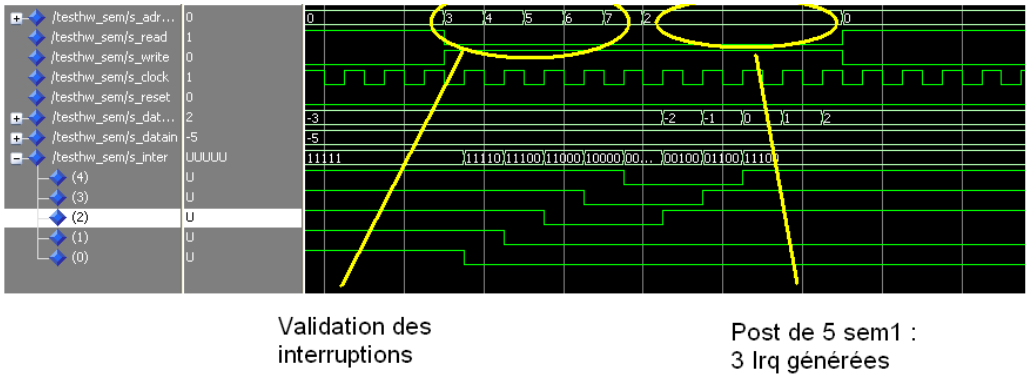


Figure A.10: *Validation des interruptions*

s'agit donc que de mapper les fils sur la structure correcte du bus.
L'interface au bus AMBA s'est avéré plus complexe. La simulation n'était pas envisageable pour valider le bridge. La validation s'est faite en générant une architecture complète à base de LEON3. La fonction de Plug & Play du bus AMBA nous a permis de vérifier que notre périphérique était reconnu par le contrôleur de bus.

A.3 Conclusion

Nous avons démontré la faisabilité du projet, en implémentant un sémaphore intégrant la fonction de *test & set* en un cycle d'horloge. L'implémentation complète (jusqu'au niveau du driver RTEMS) de la solution de sémaphores logiciels n'a pas été terminée. Le coût du bloc matériel en terme de consommation et de nombre de bascules est également à prendre en compte, même si le module est relativement léger et a été optimisé pour n'occuper que le minimum de place en fonction des paramètres comme le nombre de sémaphores, le nombre de processeurs et les tailles des FIFOs nécessaires.

Les mesures de simulation des accès au sémaphore matériel nous ont permis de concevoir un module SystemC modélisant ce service d'OS déporté dans notre modèle et ainsi pemettre une simulation réaliste de synchronisation entre tâches distribuées sur de multiples nœuds d'éxécution hétérogènes, que les tâches soient logicielles ou matérielles.

Annexe B

Résultats d'explorations de l'application de vision pour différentes tailles d'ARD

Cette annexe présente les résultats présentés page 120, d'une manière plus lisible.

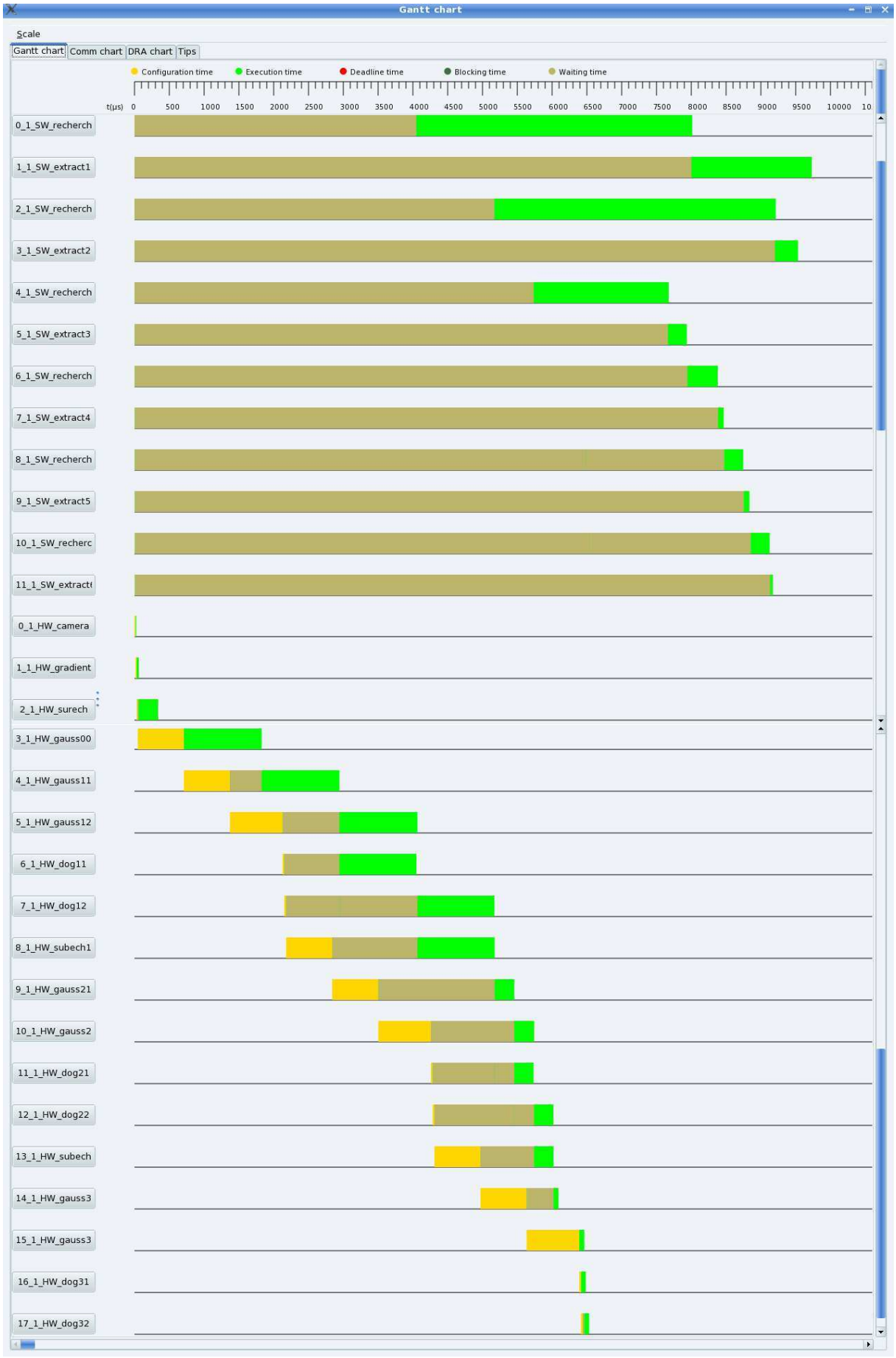


Figure B.1: *Diagramme de Gantt pour toutes les tâches réparties sur 3 processeurs et un ARD de 4500 slices*

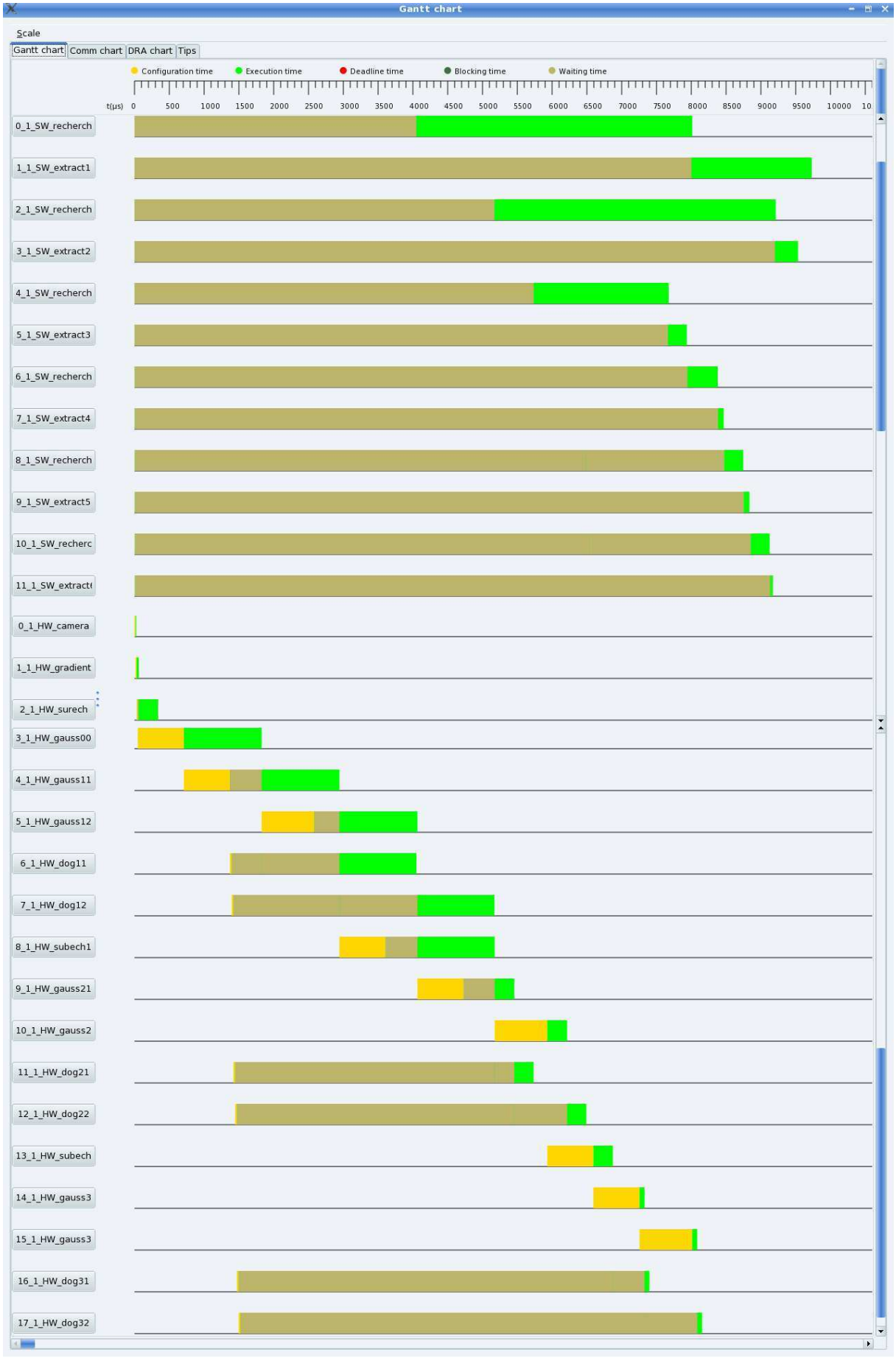


FIGURE B.2: *Diagramme de Gantt pour toutes les tâches réparties sur 3 processeurs et un ARD de 3000 slices*

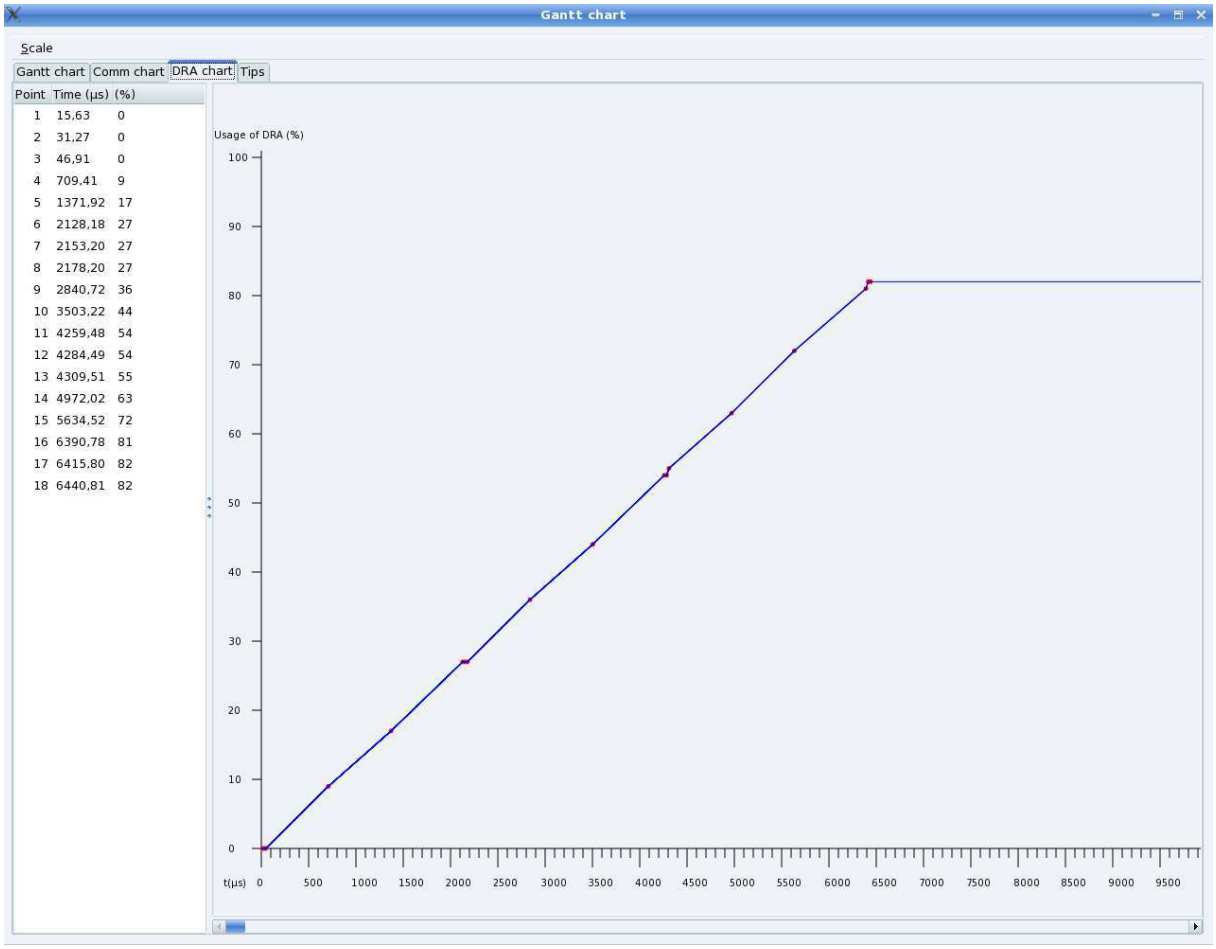


Figure B.3: *Taux d'occupation d'un gros ARD (4500 slices + 3 processeurs) au cours du temps qui montre le surcoût de la reconfiguration*

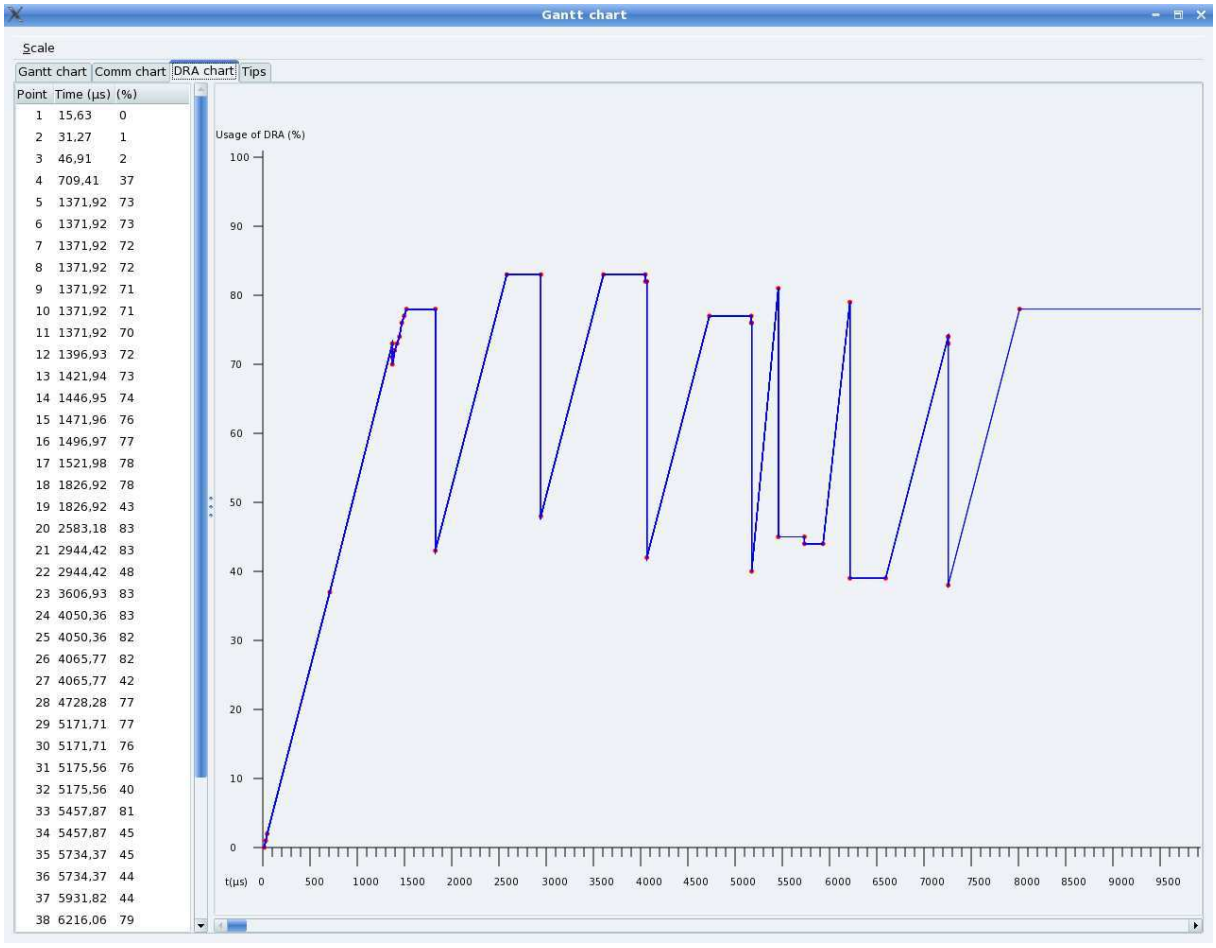


Figure B.4: *Taux d'occupation d'un petit ARD (3000 slices + 3 processeurs) au cours du temps qui montre le surcoût de la reconfiguration*

Bibliographie

[ABN+06] Rabie Ben Atitallah, L. Bonde, S. Niar, S. Meftali, and J.-L. Dekeyser. Multilevel MPSoC Performance Evaluation Using MDE Approach. In *System-on-Chip, 2006. International Symposium on*, pages 1 –4, 13-16 2006.

[ACD+03] Michel Auguin, Karim Ben Chehida, Jean-Philippe Diguet, Xavier Fornari, Anne-Marie Fouilliart, Christian Gamrat, Guy Gogniat, Philippe Kajfasz, and Yannick Le Moullec. Partitioning and CoDesign tools & methodology for Reconfigurable Computing : The EPICURE philosophy. In *International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, page 6, July 2003.

[ACG+07] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM : An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Comput. Archit. Lett.*, 6(2) :45–48, 2007.

[AFM+10] A. Aguiar, S.J. Filho, F.G. Magalhaes, T.D. Casagrande, and F. Hessel. Hellfire : A design framework for critical embedded systems' applications. In *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, pages 730 –737, 2010.

[AGB+05] Cristiano Araujo, Millena Gomes, Edna Barros, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo. Platform designer : An approach for modeling multiprocessor platforms based on SystemC. *Design Automation for Embedded Systems Journal*, 10(4) :253–283, dec 2005.

[AKH03] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion : major research trends since 1986. *Distrib. Comput.*, 16(2-3) :75–110, 2003.

[Alt07] Altera Corp. Creating Multiprocessor Nios II Systems, ver. 1.3. http://www.altera.com/literature/lit-nio2.jsp, December 2007.

[Alt11] Corporation Altera. Altera Breaks Semiconductor Industry Record for Most Transistors on an Integrated Circuit. available at : http://www.altera.com/corporate/news_room/releases/2011/products/nr-sv_mi
april 2011.

[APA+05] David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, and Ron Sass. HThreads : A Hardware/Software Co-Designed Multithreaded RTOS Kernel. *Emerging Technologies and Factory Automation, 2005. ETFA 2005.*, pages 338–346, 2005.

[ari] ARIANE 5 Flight 501 Failure. available at : http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[ASA+08] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1) :34–44, Jan. 2008.

[AVR] processeur Alf and Vegard RISC. http://www.atmel.com/products/AVR/.

[Bal91] Dana H. Ballard. Animate Vision. *Artificial Intelligence*, 48 :57–86, 1991.

[Bal95] Henri E. Bal. Interprocess Communication and Synchronization based on Message Passing. In Amsterdam Vrije Universiteit, editor, *Reader Parallel Programmeren*, 1995.

[BBB+03] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC Cosimulation and Emulation of Multiprocessor SoC Designs. *IEEE Computer*, 36(4) :53–59, April 2003.

[BG92] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BH03] Jürgen Becker and Reiner Hartenstein. Configware and Morphware Going Mainstream. *J. Syst. Archit.*, 49(4-6) :127–142, 2003.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy : A Framework for Simulating and Prototyping Heterogenous Systems. *Int. Journal in Computer Simulation*, 4(2) :0–, 1994.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.*, 38(1) :1, 2006.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[BRS07] Bishnupriya Bhattacharya, John Rose, and Stuart Swan. Language extensions to SystemC : process control constructs. In *DAC '07 : Proceedings of the 44th annual Design Automation Conference*, pages 35–38, New York, NY, USA, 2007. ACM.

[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.

[Cal90] J. P. Calvez. *Spécification et conception des systèmes : une méthodologie.* Editions Masson, 1990.

[CBR+03] Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, and Francois-Raymond Boyer. SPACE : a hardware/-software SystemC modeling platform including an RTOS. In *Forum on Design Languages(FDL'03)*, 2003.

[CBR+04a] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, M. Aboulhamid, and F-R. Boyer. *Languages for System Specification*, chapter SPACE : a hardware/-software SystemC modeling platform including an RTOS, pages 91–104. Kluwer Academic Publishers, 2004.

[CBR+04b] Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, and Francois-Raymond Boyer. SPACE : a hardware/-software SystemC modeling platform including an RTOS. *Languages for system specification : Selected contributions on UML, SystemC, System Verilog, mixed-signal systems, and property specification from FDL'03*, pages 91–104, 2004.

[CCGM03] Marcello Coppola, Stephane Curaba, Miltos Grammatikakis, and Giuseppe Maruccia. IPSIM : SystemC 3.0 Enhancements for Communication Refinement. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 20106, Washington, DC, USA, 2003. IEEE Computer Society.

[CCN06] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab, ScicOS*. Springer, 2006.

[Cen] The ArchC Resource Center. ArchC ADL. http ://www.archc.org.

[CG03a] 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.

[CG03b] Lukai Cai and Daniel Gajski. Transaction-level modeling in system level design. CECS technical report (03-10). Technical report, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, march 2003.

[CHLM+99] P. Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. A. Jarraya. Multilanguage Design of Heterogeneous Systems. In *CODES '99 : Proceedings of the seventh international workshop on Hardware/software codesign*, pages 54–58, New York, NY, USA, 1999. ACM.

[Cof] Cofluent. Cofluent Studio^{TM}. available at www.cofluentdesign.com.

[Cor] OAR Corporation. RTEMS open-source Operating System, the Real-Time Executive for Multiprocessor Systems. available at : http://www.rtems.com.

[Cor99] ARM Corporate. AMBA Specification Revision 2.0. 1999.

[CoW] CoWare. CoWare SystemC IP TLM Model Library and Virtual Platform Designer tool . available at http://www.coware.com/solutions/tlm.php.

[CPT03] Andrew S. Cassidy, JoAnn M. Paul, and Donald E. Thomas. Layered, Multi-Threaded, High-Level Performance Design. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10954, Washington, DC, USA, 2003. IEEE Computer Society.

[CS06] Coware and Synopsys. Platform Architect tool, 2006.

[CSG98] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, August 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A Next-Generation Design Framework for Platform-based Design. In *DVCon 2007*, February 2007.

[DGP+06] J. P. Diguet, G. G., J. L. Philippe, Y. Le Moullec, S. Bilavarn, C. Gamrat, K. Ben Chehida, M. Auguin, X. Fornari, and P. Kajfasz. EPICURE : A Partitioning and Co-design Framework for Reconfigurable Computing. *Microprocessors and microsystems*, 30(6) :367–387, September 2006.

[DGP+08] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. System-on-chip environment : a SpecC-based framework for heterogeneous MPSoC design. *EURASIP J. Embedded Syst.*, 2008 :5 :1–5 :13, January 2008.

[DHG+08] Markus Damm, Jan Haase, Christoph Grimm, Fernando Herrera, and Eugenio Villar. Bridging MoCs in SystemC Specifications of Heterogeneous Systems. *EURASIP Journal on Embedded Systems*, 2008 :1–16, 2008.

[Dij65] Edsger Wybe Dijkstra. Cooperating Sequential Processes, Technical Report EWD-123. Technical report, Technological University, Eindhoven, The Netherlands., sept 1965. Reprinted in *Programming Languages*, F. genuys, Ed., (Academic Press, New york, 1968).

[DJO04] Abhijit K. Deb, Axel Jantsch, and Johnny Öberg. System Design for DSP Applications in Transaction Level Modeling Paradigm. In *DAC '04 : Proceedings of the 41st annual Design Automation Conference*, pages 466–471, New York, NY, USA, 2004. ACM.

[DLJ97] B.P. Dave, G. Lakshminarayana, and N.K. Jha. COSYN : Hardware-software Co-synthesis Of Embedded Systems. *Proceedings of the 34th Design Automation Conference*, pages 703–708, Jun 1997.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP : An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5 :46–55, 1998.

[Don04] Adam Donlin. Transaction Level Modeling : Flows and Use Models. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'04)*, pages 75–80, 2004.

[DW99] O. Diessel and G. Wigley. Opportunities for Operating Systems Research in Reconfigurable Computing, 1999. Technical report ACRC-99-018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Mawson Lakes, SA.

[Ecl] Eclipse Rich Client Platform. available at : http://eclipsercp.org/.

[EET] EETimes/Altera. Give me strength ! New FPGA has 2.5 billion transistors ! available at : http://www.eetimes.com/electronics-products/fpga-pld-products/4104287/Alt

[EJ03] J. Eker and J. W. Janneck. CAL Language Report Specification of the CAL Actor Language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[ELLSV97] Stephen Edwards, L. Lavagno, E. A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems : Formal Models, Validation, and Synthesis. In *Proceedings of the IEEE 85(3)*, pages 366–390, March 1997.

[EPTP07] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Jnl. of Embedded System*, 2007(1) :2–2, 2007.

[FAMH08] S.J. Filho, A. Aguiar, C.A. Marcon, and F.P. Hessel. High-Level Estimation of Execution Time and Energy Consumption for Fast Homogeneous MPSoCs Prototyping. In *Rapid System Prototyping (RSP'08). The 19th IEEE/IFIP International Symposium on*, pages 27 –33, 2008.

[FMMR10] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling time in computing : A taxonomy and a comparative survey. *ACM Comput. Surv.*, 42(2) :1–59, 2010.

[FMP+04] Franco Fummi, Stefano Martini, Giovanni Perbellini, Massimo Poncino, Fabio Ricciato, and Maura Turolla. Heterogeneous Co-Simulation of Networked Embedded Systems. *Design, Automation and Test in Europe Conference and Exhibition*, 3 :30168, 2004.

[For08] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard, Version 2.1. available at http://www.mpi-forum.org, Sept 2008.

[FPG+03] F. Fummi, G. Perbellini, P. Gallo, M. Poncino, S. Martini, and F. Ricciato. A Timing-Accurate Modeling and Simulation Environment for Networked Embedded Systems. pages 42 – 47, jun. 2003.

[Fra] Institutions Française. Pôle de compétitivité System@TIC Paris-Région. available at : http://www.systematic-paris-region.org.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with Systemc : TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GHPS09] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an industrial SystemC/TLM model using LOTOS and CADP. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*, pages 46–55, July 2009.

[GJB+00] P. Gaussier, C. Joulain, J.P. Banquet, S. Leprêtre, and A. Revel. The visual homing problem : an example of robotics/biology cross fertilization. *Robotics and Autonomous Systems*, 30 :155–180, 2000.

[GK83] Daniel D. Gajski and Robert H. Kuhn. Guest Editor's Introduction : New VLSI Tools. *IEEE Computer*, 16(12) :11–14, December 1983.

[GLMS02] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized Rapid Prototyping For Real-Time Embedded Heterogeneous Multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[Gri04] Matthias Gries. Methods for Evaluating and Covering the Design Space During Early Design Development. *Integr. VLSI J.*, 38(2) :131–183, 2004.

[Gro02] T Grotker. Modeling Software with SystemC 3.0. 6th European SystemC Users Group Presentations, 2002.

[GSDG05] Andreas Gerstlauer, Dongwan Shin, Rainer Dömer, and Daniel D. Gajski. System-Level Communication Modeling for Network-on-Chip Synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 45–48, New York, NY, USA, 2005. ACM.

[GVNG98] Daniel Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. SpecSyn : An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. *IEEE Transactions on VLSI Systems*, 6 :84–100, 1998.

[GYG03] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS Modeling for System Level Design. In *Conference on Design, Automation and Test in Europe (DATE'03)*, pages 10130–10136, March 2003.

[GYJ01] L. Gauthier, S. Yoo, and A. Jerraya. Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 679–685. IEEE Press, 2001.

[GZ95] P. Gaussier and S. Zrehen. PerAc : A Neural Architecture to Control Artificial Animals. *Robotics and Autonomous System*, 16(2-4) :291–320, December 1995.

[GZD+00] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC : Specification Language and Methodology. Kluwer Academic Publishers.* Kluwer Academic Publishers, march 2000.

[GZGH00] A. Gerstlauer, S. Zhao, D. Gajski, and A. Horak. SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder, 2000.

[HAG08] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. *Design Automation and Test in Europe, 2008. DATE '08*, pages 3–8, March 2008.

[Har01] R. Hartenstein. A Decade of Reconfigurable Computing : a Visionary Retrospective. In *DATE '01 : Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[Has06] M. AbdElSalam Hassan. *A system Level Modeling Methodology for RTOS Centric Embedded Systems.* PhD thesis, Osaka University, 2006.

[HKH04a] P. Hastono, S. Klaus, and S.A. Huss. Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems. In *Forum on specification and Design Languages(FDL'04)*, September 2004.

[HKH04b] P. Hastono, Stephan Klaus, and Sorin A. Huss. An Integrated SystemC Framework for Real-Time Scheduling Assessments In System Level. In *The 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, Lisbon, Portugal, December 2004.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Trans. Softw. Eng.*, 18(9) :785–793, 1992.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory : architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2) :289–300, 1993.

[HMP05] Zhengting He, Aloysius Mok, and Cheng Peng. Timed RTOS Modeling for Embedded System Design. In *IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05)*, pages 448–457, 2005.

[HMS07] F. Hessel, C. Marcon, and T. Santos. High Level RTOS Scheduler Modeling for a Fast Design Validation. In *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pages 461 –466, 2007.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4) :293–333, 1996.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8) :666–677, 1978.

[HOI07] M. AbdElSalam Hassan, E. Okushi, and M. Imai. A SystemC simulation modeling approach for allocating task precedence graphs to multiprocessors. In *ASIC, 2007. ASICON'07. 7th International Conference on*, pages 1205 –1208, 22-25 2007.

[HRR+04] Fabiano Hessel, Vitor M. Da Rosa, Igor M. Reis, Ricardo Planner, Cesar A. M. Marcon, and Altamiro A. Susin. Abstract RTOS Modeling for Embedded Systems. In *IEEE International Workshop on Rapid System Prototyping (RSP'04)*, pages 210–216, Washington, DC, USA, 2004.

[HRR+06] Fabiano Hessel, Vitor M. Da Rosa, Carlos Eduardo Reif, César Marcon, and Tatiana Gadelha Serra Dos Santos. Scheduling Refinement in Abstract RTOS Models. *ACM Transactions on Embedded Computing Systems*, 5(2) :342–354, 2006.

[HSA09] Yonghyun Hwang, Gunar Schirner, and Samar Abdi. Automatic Generation of Cycle-Approximate TLMs with Timed RTOS Model Support. In Achim Rettberg, Mauro Zanella, Michael Amann, Michael Keckeisen, and Franz Rammig, editors, *Analysis, Architectures and Modelling of Embedded Systems*, volume 310 of *IFIP Advances in Information and Communication Technology*, pages 66–76. Springer Boston, 2009.

[HSTI05a] M. AbdElSalam Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai. RTK-Spec TRON : a simulation model of an ITRON based RTOS kernel in SystemC. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 554 – 559 Vol. 1, 7-11 2005.

[HSTI05b] M. AbdElSalam Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai. Virtual Prototyping of T-Engine Systems Using RTOS Centric Co-Simulation in Systemc. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society : ITI 3rd International Conference on*, pages 171–191, Dec. 2005.

[HWTT04] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. RTOS-centric hardware/software cosimulator for embedded system design. In *CODES+ISSS '04 : Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 158–163, New York, NY, USA, 2004. ACM Press.

[IM00] Vincent John Mooney III and Giovanni De Micheli. Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems. *Design Automation for Embedded Systems*, 6 :89–144, 09 2000.

[Ini] Open Virtual Platform Initiative. OVPsim instruction set simulators. available at : http://www.ovpworld.org/.

[ITR09] ITRS. Rapport 2009 - System Drivers. Technical report, International Technology Roadmap for Semiconductors, 2009.

[Jam05] Mo Jamshidi. System-of-Systems Engineering - a Definition. *IEEE International Conference on system, Mana and Cybernetics (IEEE SMC'05)*, October 2005.

[Jan05] Axel Jantsch. Models of Embedded Computation. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. Invited contribution.

[JBP06] Ahmed Amine Jerraya, Aimen Bouchhima, and Frédéric Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In Ellen Sentovich, editor, *DAC*, pages 280–285. ACM, 2006.

[JS05] Axel Jantsch and Ingo Sander. Models of Computation and Languages for Embedded System Design. *IEE Proceedings on Computers and Digital Techniques*, 152(2) :114–129, March 2005. Special issue on Embedded Microelectronic Systems ; Invited paper.

[JW04] Ahmed Amine Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*, chapter The What, Why, and How of MPSoCs, pages 1–18. Morgan Kaufmann, 2004.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74 : Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[KDK+04] Tim Kogel, Malte Doerper, Torsten Kempf, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Virtual Architecture Mapping : A

SystemC Based Methodology for Architectural Exploration of System-on-Chip Designs. In *Third International Workshop on Computer Systems : Architectures, Modeling, and Simulation, (SAMOS'03)*, pages 138–148, 2004.

[KEBR08] Matthias Krause, Dominik Englert, Oliver Bringmann, and Wolfgang Rosenstiel. Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation. In *CODES+ISSS '08 : Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2008. ACM.

[KGJ03] Paul Kohout, Brinda Ganesh, and Bruce Jacob. Hardware Support for Real-time Operating Systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*, pages 45–51, New York, NY, USA, 2003. ACM.

[KNRSV00] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level Design : Orthogonalization of Concerns and Platform-based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12) :1523–1543, December 2000.

[KSM03] P. Kuacharoen, M. Shalan, and V. Mooney. A Configurable Hardware Scheduler for Real-time Systems. In *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 96–101, 2003.

[LAB+04] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing On-Chip Communication in a MPSoC Environment. *Design, Automation and Test in Europe Conference and Exhibition*, 2 :20752, 2004.

[LE03] Eric Lenormand and Gilbert Edelin. An industrial perspective : A pragmatic high end signal processing design environment at Thales. In *SAMOS-III, Computer Systems : Architectures, Modeling, and Simulation, LNCS 3133*, page 145. Springer, September 2003.

[LG07] Shaoshan Liu and Jean-Luc Gaudiot. Synchronization Mechanisms on Modern Multi-core Architectures. 4697 :290–303, 2007.

[LGC00] S. Leprêtre, P. Gaussier, and J.P. Cocquerez. From Navigation to Active Object Recognition. In *Proceedings of the Sixth International Conference on Simulation for Adaptive Behavior (SAB)*, pages 266–275, Paris, France, 2000.

[LLC] SimpleScalar LLC. SimpleScalar instruction set simulators. available at : http://www.simplescalar.com.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flown. *Proceedings of the IEEE*, 75(9) :1235–1245, September 1987.

[LMD+03] Jaehwan Lee, Vincent John Mooney, III, Anders Daleby, Karl Ingström, Tommy Klevin, and Lennart Lindh. A comparison of the RTU hardware RTOS with a hardware/software RTOS. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, pages 683–688, New York, NY, USA, 2003. ACM.

[LP07] E. Lubbers and M. Planner. ReconOS : An RTOS Supporting Hard-and Software Threads. pages 441–446, Aug. 2007.

[LP08] E. Lubbers and M. Platzner. A Portable Abstraction Layer for Hardware Threads. In *International Conference on Field Programmable Logic and Applications, 2008 (FPL'08)*, pages 17–22, Sept. 2008.

[LS95] Victor B. Lortz and Kang G. Shin. Semaphore Queue Priority Assignment for Real-Time Multiprocessor Synchronization. *Software engineering*, October 1995.

[LSV98] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12) :1217–1229, 1998.

[Mak00] Tsugio Makimoto. The Rising Wave of Field Programmability(keynote). In *Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing, 10th International Workshop (FPL'OO)*, pages 1–6. Springer Verlag, aug 2000.

[MCP05] Rocco Le Moigne, Jean-Paul Calvez, and Olivier Pasquier. *Modélisation et simulation basée sur SystemC des systèmes monopuces au niveau transactionnel pour l'évaluation de performances*. PhD thesis, September 2005.

[MD05] Benoit Miramond and Jean-Marc Delosme. Design Space Exploration for Dynamically Reconfigurable Architectures. In *DATE'05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 366–371. IEEE Computer Society, 2005.

[MGGH05] M. Maillard, O. Gapenne, Ph. Gaussier, and L. Hafemeister. Perception as a dynamical sensori-motor attraction basin. In M. Capcarrere et al., editor, *Advances in Artificial Life (8th European Conference, ECAL)*, volume LNAI 3630 of *Lecture Note in Artificial Intelligence*, pages 37–46. Springer, sep 2005.

[MKS+02] A. Mihal, C. Kulkarni, C. Sauer, K. Vissers, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, and S. Malik. Developing Architectural Platforms : A Disciplined Approach. *IEEE Design and Test of Computers*, 19(6) :6–16, dec 2002.

[MMA07] F. Muller, F. Muhammad, and Michel Auguin. Design of a Hardware Multiprocessor Real-Time Operating System. In *DATE University Booth (DATE'03)*, april 2007.

[MP02] Sumit Mohanty and Viktor K. Prasanna. Rapid System-Level Performance Evaluation and Optimization for Application Mapping onto SoC Architectures. In *in Proc. of the IEEE International ASIC/SOC Conference*, 2002.

[MPC04] R. Le Moigne, O. Pasquier, and J-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *Conference on Design, automation and test in Europe (DATE'04)*, volume 3, pages 30082–30087, Paris, France, Feb. 2004.

[MT02] José F. Martínez and Josep Torrellas. Speculative synchronization : applying thread-level speculation to explicitly parallel applications. *SIGOPS Oper. Syst. Rev.*, 36 :18–29, October 2002.

[muc02] *MicroC/OS-II : the real-time kernel*. CMP Media, Inc., USA, 2002. available at : http://www.micrium.com/page/support/bookstore.

[MVD00] Hugo De Man, D. Verkest, and Dirk Desmet. Operating System based software generation for Systems-on-Chip. In *Proceedings of the 37th Conference on Design Automation (DAC'00)*, pages 396–401, 2000.

[MVG03] Jan Madsen, Kashif Virk, and Mercury Gonzalez. Abstract RTOS Modeling for Multiprocessor System-on-Chip. In *Symposium on System-on-Chip (SOC'03)*, pages 147–150, November 2003.

[MVM07] Shankar Mahadevan, Kashif Virk, and Jan Madsen. ARTS : A SystemC-based framework for multiprocessor Systems-on-Chip modelling. *Design Automation for Embedded Systems*, 11(4) :285–311, 2007. 10.1007/s10617-007-9007-6.

[NCV+03] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 174, 2003.

[Nic02] G. Nicolescu. *Spécification et validation des systèmes hétérogènes embarqués*. PhD thesis, TIMA, 2002.

[NVC10] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Safari Through the MPSoC Run-Time Management Jungle. *Journal of Signal Processing Systems*, 60 :251–268, August 2010. 10.1007/s11265-008-0305-4.

[OBdNC+03] O. Ogawa, S. Bayon de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai. A Practical Approach for Bus Architecture Optimization at transaction Level. pages 176–181, 2003.

[oca] opensource community. Linux Operating System, open source, royalty-free. available at : http://www.kernel.org.

[ocb] opensource community. The embedded Cygnus operating system (eCos), a Real-Time OS, open source, royalty-free. available at : http://ecos.sourceware.org.

[OCP] OCPIP. Open Core Protocol International Partnership (OCP-IP). available at : http://www.ocpip.org.

[OMGa] OMG. OMG Model Driven Architecture. available at : http://www.omg.org/mda/.

[OMGb] OMG. Unified Modeling Language. available at :www.uml.org/.

[OMG98] OMG. The Common Object Request Broker : Architecture and Specification, revision 2.2. OMG Document 98-07-01, Object Management Group, February 1998. Also called CORBA/IIOP 2.2 specification.

[OMG06a] OMG. Systems Modeling Language (SysML) Specification. OMG document : ad/2006-03-08-01, version 1. Draft, April 2006.

[OMG06b] OMG. UML Profile for System on a Chip (SOC). OMG Available Specification, version 1.0.1 formal /06-08-01, August 2006.

[Pac96] Peter S. Pacheco. *Parallel programming with MPI.* 1996.

[PAS+06] H. Posadas, J. Adamez, P. Sanchez, E. Villar, and F. Blasco. POSIX modeling in SystemC. In *Asia and South Pacific Design Automation Conference (ASP-DAC'06)*, pages 485–490, Microelectron. Eng. Group, Cantabria Univ., Santander, Spain, January 2006. IEEE Computer Society.

[PAV+05] H. Posadas, J.A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation : A POSIX model. *Design Automation for Embedded Systems*, 10(4) :209–227, December 2005.

[PBN+03] JoAnn M. Paul, Alex Bobrek, Jeffrey E. Nelson, Joshua J. Pieper, and Donald E. Thomas. Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors. In *DAC '03 : Proceedings of the 40th annual Design Automation Conference*, pages 408–411, New York, NY, USA, 2003. ACM.

[PDBR04] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Extending the Transaction Level Modeling approach for Fast Communication Architecture Exploration. In *DAC '04 : Proceedings of the 41st annual Design Automation Conference*, pages 113–118, 2004.

[PH97] David A. Patterson and John L. Hennessy. *Computer Organization and Design : The Hardware/software Interface, Second Edition.* Morgan Kaufmann Publishers, 1997.

[PQVM07] Héctor Posadas, David Quijano, Eugenio Villar, and Marcos Martínez. SCope : SoC Co-simulation and Performance Estimation in SystemC. In IEEE/ACM Design, Automation and Test in Europe 2007 University Booth, 2007.

[Pra08] PragmaDev. Real Time Developer Studio, 2008.

[PSMN06] H.D. Patel, S.K. Shukla, E. Mednick, and R.S. Nikhil. A rule-based model of computation for SystemC : integrating SystemC and Bluespec for co-design. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 39–48, July 2006.

[PVB05] Héctor Posadas, Eugenio Villar, and Francisco Blasco. Real-Time Operating System modeling in SystemC for HW/SW co-simulation. In *DCIS'05 : XX Conference on Design of Circuits and Integrated Systems, , IST Lisboa.*, nov. 2005.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. pages 5–17, 2002.

[RMD03] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation : a technique for fast and flexible instruction set simulation. In *DAC'03 : Proceedings of the 40th annual Design Automation Conference*, pages 758–763, New York, NY, USA, 2003. ACM.

[Ros85] Douglas T. Ross. Applications and Extensions of SADT. *IEEE Computer*, 18(4) :25–34, 1985.

[RSG+05] L. Rioux, T. Saunier, S. Gerard, A. Radermacher, R. De Simone, T. Gauthier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, and C. Andrè. MARTE : A New OMG Profile RFP for the Modeling and Analysis of Real-Time Embedded Systems. In *DAC 2005 Workshop UML for SoC Design, UML-SoC'05*, June 2005.

[RSPF] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. OSCI White paper : Transaction Level Modeling in SystemC . available at : http://www.systemc.org.

[SD08] Gunar Schirner and Rainer Dömer. Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. In *Design, Automation and Test in Europe, DATE 2008*, pages 122–127. IEEE, 2008.

[SGD07] G. Schirner, A. Gerstlauer, and R. Domer. Abstract, Multifaceted Modeling of Embedded Processors for System Level Design. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 384 –389, 2007.

[SGD08] Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer. Automatic Generation of Hardware dependent Software for MPSoCs from Abstract System Specifications. In *ASP-DAC*, pages 271–276. IEEE, 2008.

[Sif01] Joseph Sifakis. Modeling Real-Time Systems-Challenges and Work Directions. In *EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software*, pages 373–389, London, UK, 2001. Springer-Verlag.

[Sif05] Joseph Sifakis. A Framework for Component-based Construction Extended Abstract. In *SEFM '05 : Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 293–300, Washington, DC, USA, 2005. IEEE Computer Society.

[SLSV00] Marco Sgroi, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal Models for Embedded System Design. *IEEE Des. Test*, 17(2) :14–27, 2000.

[Sor94] Y. Sorel. Massively Parallel Systems with Real Time Constraints, the Algorithm Architecture Adequation Methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, May 1994.

[Spe] SpecC standard. available at : http://www.specc.org.

[SPSI08] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. Phasers : a unified deadlock-free construct for collective and point-to-point synchronization. pages 277–288, 2008.

[Sri95] R. Srinivasan. RPC : Remote Procedure Call Protocol Specification Version 2, Request for Comments : 1831. available at : http://tools.ietf.org/html/rfc1831, 1995.

[SSGD07] Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, and Rainer Dömer. Embedded Software Development in a System-Level Design Flow. In Achim Rettberg, Mauro Zanella, Rainer Dömer, Andreas Gerstlauer, and Franz Rammig, editors, *Embedded System Design : Topics, Techniques*

and Trends, volume 231 of *IFIP Advances in Information and Communication Technology*, pages 289–298. Springer Boston, 2007. 10.1007/978-0-387-72258-0_25.

[ST98] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30 :123–169, 1998.

[SVM01] Alberto L. Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, 18(6) :23–33, 2001.

[SWP04] C. Steiger, H. Walder, and M. Platzner. Operating Systems for Reconfigurable Embedded Platforms : Online Scheduling of Real-Time Tasks. *Computers, IEEE Transactions on*, 53(11) :1393–1407, Nov. 2004.

[Sys] System@TIC. Projet Ter@OPS du Pôle de compétitivité System@TIC. available at : http://teraops-emb.ief.u-psud.fr/index.php.

[Sys05] SystemVerilog.org. *1800-2009 IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*. IEEE standard association, 2005. This standard represents a merger of two previous standards : IEEE Std 1364™-2005Verilog® hardware description language (HDL) and IEEE Std 1800-2005 SystemVerilog unified hardware design, specification and verification language.

[TCM01] H. Tomiyama, Y. Cao, and K. Murakami. Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 93–100, 2001.

[TG98] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[Til09] TileraCorporation. TILE-Gx Processors with up to 100 cores. available at :http://www.tilera.com/products/TILE-Gx.php, october 2009.

[Tre95] N. Tredennick. Technology and business : forces driving microprocessor evolution. *Proceedings of the IEEE*, 83(12) :1641 –1652, dec 1995.

[Tur50] A. M. Turing. Computing Machinery and Intelligence. *Mind*, 59 :433–460, 1950.

[UE] UE. Projet Européen MORPHEUS (Multi-purpOse dynamically Reconfigurable Platform for intensive HEterogeneoUS processing), contrat EU IST-4-027342, Leader : Thales TRT/LSE. available at : http://www.morpheus-ist.org/.

[UHGB04] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. On-demand FPGA Run-time System for Dynamical Reconfiguration with Adaptive Priorities. In *Field Programmable Logic and its Applications (FPL)*, number 3203 in Lecture Notes in Computer Science, pages 454–463, 2004.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer Journal*, 38(5) :48–56, 2005.

[VG01] Frank Vahid and Tony Givargis. Platform Tuning for Embedded Systems Design. *IEEE Computer*, 34(3) :112–114, mar 2001.

[VGBP03] Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, and Elena Moscu Panainte. The Molen Programming Paradigm. In *International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, page 6, July 2003.

[vhd04] vhdl.org. *IEC 61691-1-1 Ed.1 (2004-10) (IEEE Std 1076-2002) : Behavioural Languages – Part 1-1 : VHDL Language Reference Manual*. IEEE standard association and VHDL Analysis and Standardization Group (VASG), 2004.

[VMM+08] François Verdier, Benoît Miramond, Mickaël Maillard, Emmanuel Huck, and Thomas Lefebvre. Using High-Level RTOS Models for HW/SW Embedded Architecture Exploration : Case Study on Mobile Robotic Vision. *EURASIP Journal on Embedded Systems*, Volume 2008(349465) :17, 2008. http ://www.hindawi.com/getarticle.aspx ?doi=10.1155/2008/349465.

[VOM+06] M. Vetromille, L. Ost, C.A.M. Marcon, C. Reif, and F. Hessel. RTOS Scheduler Implementation in Hardware and Software for Real Time Applications. In *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, pages 163–168, 2006.

[VP06] E. Viaud and F. Pecheux. A New Paradigm and Associated Tools for TLM/T Modeling of MPSoCs. In *Research in Microelectronics and Electronics 2006, Ph. D.*, pages 217 –220, 0-0 2006.

[VSI] VSI Alliance™, On-Chip Bus Development Working Group . Virtual Component Interface (VCI) Standard Version 2 (OCP 2 2.0). available at : http://www.vsi.org/, april.

[WL99] Equipe West-LIFL. GASPAR tool - Graphical Array Specification for Parallel and Distributed Computing. available at :http://www2.lifl.fr/west/gaspard/, 1999.

[WP03] H. Walder and M. Platzner. Reconfigurable Hardware Operating Systems : From Design Concepts to Realizations. In *Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*, pages 284–287, 2003.

[WTS96] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Prioritized Interprocessor Synchronization in an ITRON-MP Implementation. pages 48–, 1996.

[Xil] Xilinx. Virtex 5 familly overview. http://www.xilinx.com/.

[YDG04] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded Software Generation from System Level Design Languages. In Masaharu Imai, editor, *Design Automation Conference, 2004 Asia and South Pacific (ASP-DAC'04)*, pages 463–468. IEEE, 2004.

[YGG03] Haobo Yu, Andreas Gerstlauer, and Daniel Gajski. RTOS Scheduling in Transaction Level Models. In *IEEE/ACM/IFIP int. conf. on Hardware/-software codesign and system synthesis (CODES+ISSS'03)*, pages 31–36, 2003.

[ZCQ+05] Bo Zhou, Yonghui Chen, Weidong Qiu, Yan Chen, and Chenglian Peng. Reduce SW/HW Migration Efforts by a RTOS in Multi-FPGA Systems. In *CSCWD*, pages 636–645, 2005.

[ZMG09] Henning Zabel, Wolfgang Müller, and Andreas Gerstlauer. Accurate RTOS Modeling and Analysis with SystemC. In Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer, editors, *Hardware-dependent Software*, pages 233–260. Springer Netherlands, 2009.

Résumé

Simulation de haut niveau de systèmes d'exploitations distribués pour l'exploration matérielle et logicielle d'architectures multi-nœuds hétérogènes

Concevoir un système embarqué implique de trouver un compromis algorithme/architecture en fonction des contraintes temps-réel.

Thèse : pour concevoir un MPSoC et plus particulièrement avec les circuits reconfigurables modifiant le support d'exécution en cours de fonctionnement, la nécessaire validation des comportements fluctuants d'un système réactif impose une évaluation préalable que l'on peut réaliser par simulation (de haut niveau) tout en permettant l'exploration de l'espace de conception architectural, matériel mais aussi logiciel, au plus tôt dans le flot de conception.

Le point de vue du gestionnaire de la plateforme est adopté pour explorer à haut niveau les réactions du système aux choix de partitionnement impactés par l'algorithmique des services du système d'exploitation et leurs implémentations possibles.

Pour cela un modèle modulaire de services d'OS simule fonctionnellement et conjointement en SystemC le matériel, les tâches logicielles et le système d'exploitation, répartis sur plusieurs nœuds d'exécution hétérogènes communicants.

Ce modèle a permis d'évaluer l'architecture temps-réel idéale d'une application dynamique de vision robotique conjointement à l'exploration des services de gestion d'une zone reconfigurable modélisée. Ce modèle d'OS a aussi été intégré dans un simulateur de MPSoC hétérogène d'une puissance estimé à un Tera opérations par seconde.

Mots-clés : Modélisation haut niveau TLM, RTOS, co-simulation, MPSoC, Reconfigurable, SystemC

Abstract

High level simulation of distributed operating system for hardware and software exploration of heterogeneous multi-nodes architectures

Designing an embedded system implies to look for the right algorithm/architecture compromise depending on the real-time constraints.

For MPSoC an especially with reconfigurable devices which enable to modify the running executing support, a validation by a preliminar evaluation of the variable behaviors of a reactive system becomes necessary. This could be done by a high level simulation allowing to explore, early in the design flow, the architectural design space, hardware and software, especially the RTOS.

The platform manager point of view is used to explore the systems reactions to the partitioning choices and also the influence of the various algorithms and the impact of implementations of the operating system's services refined in hardware or software.

For that, a SystemC model composed of modular OS services allow to jointly and functionally simulate hardware, software tasks and the operating system, distributed on heterogeneous communicating execution nodes.

To evaluate the perfect real-time reconfigurable architecture of a dynamical robot vision application, we explored its partitioning and the useful OS services accordingly. This model has been integrated in a big simulator of an heterogeneous chip designed to provide a Tera operations per second power.

Keywords : High level modeling, RTOS, co-simulation, MPSoC, Reconfigurable, SystemC, TLM