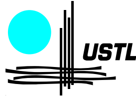

# Modélisation d'IP pour la simulation SystemC d'OS de type Linux embarqué

Mémoire de Master Recherche Informatique
filière conception de systèmes embarqués

Julien Taillard
Julien.Taillard@lifl.fr

Responsables :
Philippe Marquet
Samy Meftali

Equipe WEST
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
France

1er Juillet 2005

**Résumé**

Alors que l'utilisation des systèmes embarqués augmente, les méthodes de conception des systèmes embarqués n'arrivent pas à suivre l'évolution. Ainsi, il y a un écart entre le nombre de transistors que l'on est capable de mettre sur une puce et le nombre que l'on est capable de gérer dans un temps raisonnable. C'est pourquoi, il est nécessaire de développer les outils de développement et notamment les méthodes de simulations permettant de vérifier rapidement ces systèmes. Une solution qui est suggérée pour réduire cet écart est d'augmenter le niveau d'abstraction dans la conception des systèmes.

Ce mémoire présente une méthode de simulation de systèmes embarqués à un haut niveau d'abstraction pour essayer de réduire cet écart dans la productivité. La simulation intègre des systèmes d'exploitation de type Linux embarqué pour permettre l'ordonnancement dynamique. La simulation est effectuée de manière séparée, c'est à dire la simulation du matériel (en SystemC) d'un côté et la simulation du logiciel (application + système d'exploitation) de l'autre. Cela permet d'exécuter les choses voulues sur le matériel tout en ayant une simulation assez rapide. Des mesures sont faites durant la simulation (nombre de changement de contexte, temps d'exécution) pour vérifier la validité du système.

**Mots clés :** Systèmes embarqués, simulation, Linux, ordonnancement dynamique.

# Remerciements

L'ensemble de ces 5 mois de stage de recherche n'aurait pu se dérouler dans d'aussi bonnes conditions sans l'attention et les conseils de nombreuses personnes envers qui je suis redevable.

Je tiens à remercier mes responsables de stage Philippe Marquet et Samy Meftali pour l'intérêt constant qu'il ont montrer pour mon travail. Je n'oublierai pas Monsieur Jean-Luc Dekeyser pour son accueil au sein de son équipe.

Je remercie par ailleurs tous les autres membres de l'équipe WEST, Luc, Christophe, Philippe, Pierre, Lossan, Sebastien, Ali, Rabie, Arnaud, Eric, Ashish, Mickaël, Ouassila, Eric, Yosri, Ahmed pour leurs remarques avisées et la bonne ambiance quotidienne.

# Table des matières


**1 Introduction** **9**

**2 État de l'art** **11**
2.1 Le flot de conception d'un système embarqué . . . 11
2.1.1 Flot général . . . 11
2.1.2 Le flot de conception de l'équipe WEST : Gaspard . . . 13
2.2 Les langages de description d'architecture (HDL) . . . 13
2.2.1 Définition . . . 13
2.2.2 SystemC . . . 13
2.3 Les niveaux d'abstraction dans la conception de systèmes embarqués . . . 15
2.3.1 Le niveau porte logique . . . 15
2.3.2 Le Modèle RTL . . . 15
2.3.3 Le Modèle TLM . . . 16
2.4 Les types de simulations/validations de logiciels . . . 18
2.4.1 Simulation fonctionnelle . . . 18
2.4.2 Simulation à l'aide d'un Instruction Set Simulator . . . 18
2.4.3 Les modèles de simulations de systèmes d'exploitation . . . 19
2.5 Les cosimulations matérielles/logicielles . . . 21
2.6 Le temps dans les cosimulations . . . 22

**3 Cosimulation avec un système d'exploitation de type Linux embarqué** **25**
3.1 Principes généraux de la simulation . . . 25
3.1.1 La simulation matérielle . . . 26
3.1.2 Système d'exploitation virtuel . . . 26
3.1.3 Applications . . . 28
3.1.4 Coordination entre les entités de simulations . . . 28
3.1.5 Le temps dans la cosimulation . . . 29
3.2 Protocole de Communication . . . 30
3.2.1 Les besoins . . . 31
3.2.2 Définition du protocole . . . 31
3.2.3 Un exemple . . . 35
3.3 Implantation . . . 35
3.3.1 Organisation de la simulation matérielle . . . 35

3.3.2 L'ordonnancement des applications . . . . . . . . . . . . . . . . . . 37
3.3.3 Librairie ptrace . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.3.4 Connaître les accès en RAM . . . . . . . . . . . . . . . . . . . . . 38

**4 Test et validation 41**
4.1 Architecture de tests . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.1.1 Au niveau TLM . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.1.2 Au niveau RTL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2 Validation de l'approche . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2.1 Calcul des taux d'erreurs . . . . . . . . . . . . . . . . . . . . . . 43
4.2.2 Validation du système virtuel . . . . . . . . . . . . . . . . . . . . 43
4.2.3 Temps d'exécution de la cosimulation . . . . . . . . . . . . . . . . 44

**5 Conclusion et perspectives 45**

# Chapitre 1

# Introduction

L'évolution de la technologie d'intégration sur puce a permis l'augmentation du nombre de transistors pouvant être placés sur une puce. Cela a permis l'apparition des System On Chip (SOC) qui sont l'intégration sur une même puce de différents composants tels que des processeurs, des bancs mémoires, des circuits spécialisés (ASIC) et des circuits reconfigurables (FPGA). Mais les outils permettant la conception de ces systèmes n'arrivent pas à suivre l'évolution technologique. Ainsi, la technologie permet de mettre sur une puce plus de transistors que ce que les concepteurs sont capables de gérer dans un temps raisonnable, d'où une perte de productivité. Pour essayer de résoudre ce problème, il faut développer les outils et méthodes de conceptions des systèmes embarqués.
Cela peut passer par une modification dans le flot de conception des systèmes, et notamment au niveau de la simulation des systèmes embarqués. La simulation permettant la validation du système se fait tardivement dans le flot de conception ce qui peut entraîner des retours en arrière dans ce flot en cas de problèmes.
C'est pourquoi une validation effectuée tôt semble judicieuse. De même, les systèmes d'exploitation n'apparaissent pas explicitement au niveau du logiciel, ils sont enfouis dedans ce qui fait que l'ordonnancement est statique. Or, on souhaiterait avoir un ordonnancement dynamique pour avoir un système plus flexible.
Nous allons présenter dans ce mémoire une méthode de simulation de systèmes embarqués avec un système d'exploitation de type Linux embarqué. Cette méthode permettra de faire des simulations très tôt dans le flot de conception d'un système embarqué tout en ayant un niveau système d'exploitation qui apparaît souvent très tard dans la conception de système embarqués.

Dans un premier temps, nous verrons l'état de l'art, ensuite l'approche proposée, puis la méthode de validation envisagée. Nous terminerons par une conclusion et dégagerons les perspectives de ce travail.

# Chapitre 2

# État de l'art

La validation simultanée du logiciel et du matériel, tôt dans le flot de conception d'un système embarqué, peut permettre d'éviter de remarquer des erreurs trop tard dans le flot de conception et ainsi de faire des retours en arrière dans le flot de conception. La difficulté de mise en oeuvre d'une telle modification vient du fait qu'il y a beaucoup de paramètres à prendre en compte pour permettre une bonne simulation. Les différents paramètres peuvent être la rapidité de la simulation, sa précision et sa difficulté d'exécution.

Nous allons tout d'abord étudier le flot de conception d'un système embarqué, puis les langages de description d'architecture servant à décrire l'architecture matérielle. Ensuite nous allons voir les niveaux d'abstraction utilisés dans la conception de matériel puis les types de simulations logiciels, les types de cosimulations et enfin le temps dans ces simulations.

## 2.1 Le flot de conception d'un système embarqué

### 2.1.1 Flot général

La conception d'un système embarqué se fait grâce au codéveloppement ou codesign. Le codesign permet de développer conjointement les diverses parties d'un système hétérogène (logiciel, électronique, etc).
Il n'existe pas un unique flot de conception de systèmes embarqués mais plusieurs qui sont basés sur des outils différents avec leurs avantages et leurs inconvénients qui peuvent être la spécification de départ et la manière dont les raffinements sont effectués. Mais on peut extraire un flot de conception théorique qui est commun aux différentes méthodes, c'est ce dont nous allons parler ici en faisant abstraction d'un flot spécifique. Le flot de conception du codesign est défini ici (Figure 2.1)

Tout d'abord, on part d'une description fonctionnelle du système que l'on veut obtenir. C'est le cahier des charges du système.

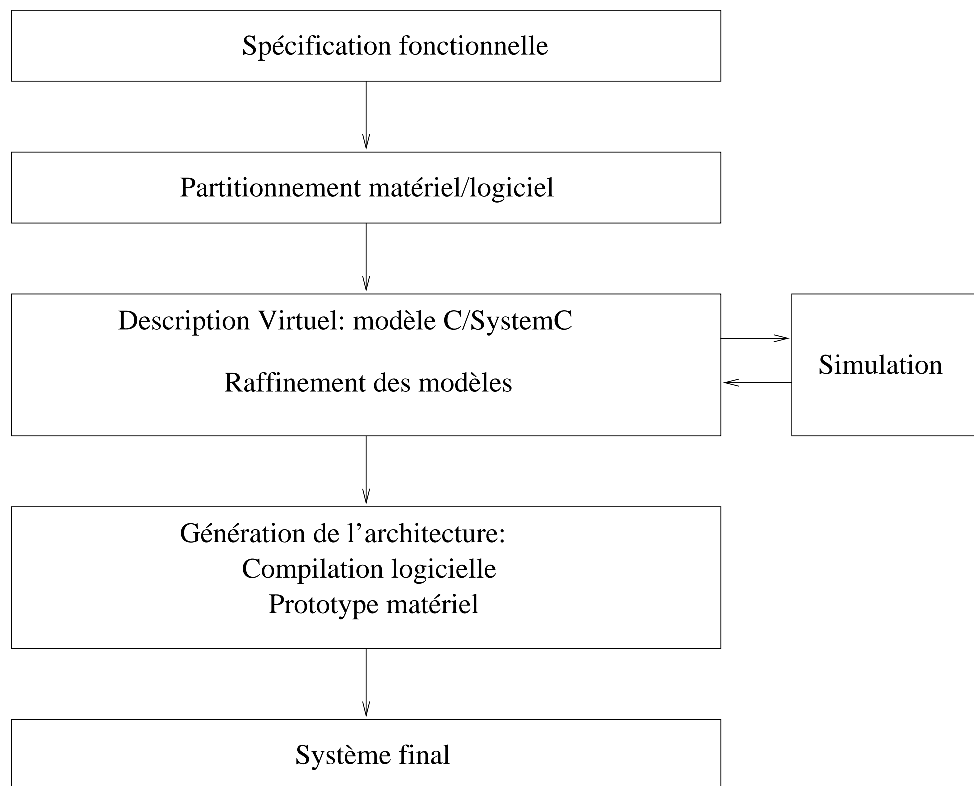


FIG. 2.1 – Flot générique de codesign

Ensuite, il faut partitionner les tâches entre tâches matérielles et tâches logicielles. Il faut aussi choisir le matériel que l'on va utiliser : choisir le(s) processeur(s), le type de mémoire, la taille de la mémoire, etc.
Puis, on produit un modèle de l'architecture ainsi que l' (les) application(s) que l'on va raffiner et valider par différentes simulations jusqu'à obtenir ce que l'on veut, en terme de temps d'exécution du système , de surface de la puce et de consommation d'énergie. C'est l'exploration de l'espace de conception.
Une fois le système validé par la simulation, on produit un prototype pour vérifier qu'il n'y a pas d'erreur puis le système est fabriqué. Les différents flots de conception [**?**] existants se basent sur des outils mais on peut les comparer selon différents critères :

- temps de développement et difficulté,
- coût de production final,
- réutilisabilité,
- qualité du contrôle pendant la phase de conception,
- pas d'échec de réalisation finale.

### 2.1.2 Le flot de conception de l'équipe WEST : Gaspard

Gaspard [?] est un outil de développement de systèmes embarqués développé par l'équipe WEST. Le but de Gaspard est de fournir un unique environnement de développement pour tout le processus de conception d'un système embarqué.
Gaspard doit permettre, à partir d'une description UML[1] du système, la modélisation du système, la simulation, le test et la génération de code pour l'application embarquée 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 flots de conception de systèmes embarqués sont basés sur des langages de description d'architecture. Ceci permet de raffiner le modèle du matériel et de faire les simulations.

## 2.2 Les langages de description d'architecture (HDL)

### 2.2.1 Définition

Les langages de description d'architecture (HDL : Hardware Description Language) sont des langages spécifiquement définis pour permettre la description d'architecture matérielle. Ils introduisent tous les concepts nécessaires à cela, les structures de communication, horloges, etc.. Ces langages permettent la simulation de l'architecture afin de la vérifier avant de faire un circuit prototype. Souvent un sous-ensemble de ces langages permet de synthétiser automatiquement, grâce à des outils, le code vers un plus bas niveau, surtout une synthèse entre le niveau RTL et le niveau porte logique [?, ?].

Les langages de description sont nombreux : SystemC, VHDL, Verilog, SpeC.
Nous allons nous focaliser sur SystemC qui est le langage que l'on va utiliser par la suite.

### 2.2.2 SystemC

SystemC [?] est un langage basé sur le C++. Il est soutenu par le consortium OSCI (Open SystemC Initiative) qui est formé de différents constructeurs de matériel et fournisseurs d'outils tel que ARM, Cadence et Synopsys.

[1] Unified Modeling Language

#### Concept d’architecture dans SystemC

Les éléments de base de SystemC sont les modules et les canaux de communications. Les modules comportent des processus (les fonctionnalités), des ports de communications, des variables partagées entre les processus, mais aussi d’autres modules pour avoir une représentation hiérarchique de l’architecture. Les différents modules sont connectés entre eux par des canaux de communications liés au port. Ces canaux de communications peuvent véhiculer des types primitifs (bit par exemple) mais aussi des structures définies par l’utilisateur pour s’abstraire des protocoles de communications.

#### Les structures SystemC

Les modules sont des SC_MODULE qui sont des classes. Les processus sont implémentés sous forme de co-routines. Il existe différents types de co-routines en SystemC : les SC_METHOD et les SC_THREAD (les SC_CTHREAD étant un cas particulier de SC_THREAD). Les SC_METHOD s’exécutent entièrement quand on les appelle alors que les SC_THREAD s’exécutent jusqu’au prochain appel à la primitive de synchronisation “wait()”. Les co-routines peuvent être réveillées grâce à un mécanisme de sensibilité aux signaux. Ainsi, les co-routines peuvent être mises en attente du changement de valeur d’un de leur signal.

#### Le noyau SystemC

L’exécution de la simulation en SystemC est gérée par le noyau SystemC. L’ordonnancement du noyau ne gère ni la préemption, ni la notion de priorité entre les co-routines. Il exécute les co-routines prêtes à être exécutées car elles ont été réveillées au temps précédent par un évènement (changement de valeur d’un signal auquel la co-routine est sensible, fin d’une attente temporelle, etc..).

Les possibilités de modélisation du logiciel avec SystemC 2.0.1 ou 2.1 sont encore limitées **[?]**. Le contrôle des processus (suspension, pause,arrêt) est, par exemple, impossible. SystemC 3.0 **[?]** dont les innovations devraient se porter principalement sur le logiciel pourrait apporter les choses suivantes :

- création dynamique de processus,
- modélisation de l’ordonnancement des processus,
- gestion de la préemption des processus,
- introduction d’un système d’exploitation temps réel générique,
- annotation en délai des processus.

## 2.3 Les niveaux d’abstraction dans la conception de systèmes embarqués

Il existe plusieurs niveaux d’abstraction dans les modèles de conception de systèmes embarqués. Mais deux sont plus utilisés que les autres, il s’agit des niveaux TLM (Transaction Level Modeling) et RTL (Register Transfert Level). Ces deux niveaux ont des fonctionnalités et interviennent à des moments différents dans la conception d’un SoC (Figure 2.2). Plus le niveau d’abstraction est haut, plus la simulation est rapide.

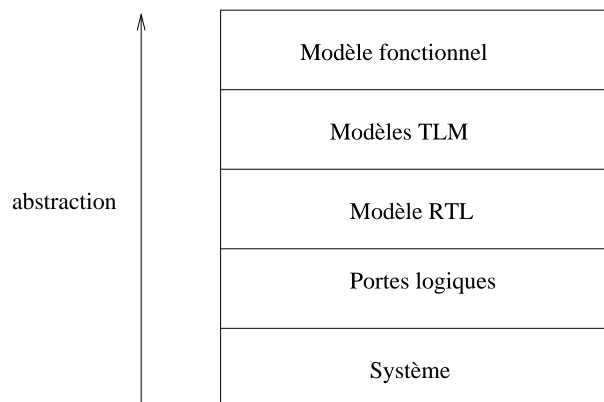


FIG. 2.2 – Les différents niveaux d’abstraction

### 2.3.1 Le niveau porte logique

Il s’agit du niveau le plus bas manipulé par des informaticiens. Il décrit le système au niveau des portes logiques. C’est donc un très bas niveau qui est très long à implémenter si on fait tout à la main, c’est pourquoi le niveau RTL a été crée pour faciliter la conception des systèmes.

### 2.3.2 Le Modèle RTL

Habituellement, les flots de conceptions de systèmes embarqués utilisent principalement le modèle RTL (Register Transfert Level) car ce modèle est plus ancien que le niveau TLM.
Le modèle RTL permet de modéliser le système au niveau des transferts de registres. Ainsi, à ce niveau d’abstraction, on doit modéliser tous les signaux passant entre les différentes entités de simulations. Ce niveau est temporisé au cycle d’horloge près. C’est pourquoi les simulations matérielles effectuées au niveau RTL sont relativement longues, ce qui peut allonger le temps de conception du système.

Mais le modèle RTL ne permet pas de supporter l’augmentation de la complexité

des systèmes, à cause de son bas niveau. Une solution suggérée pour ce problème est l'augmentation du niveau d'abstraction des modèles pour améliorer la productivité, c'est pourquoi le modèle TLM a été introduit [?].

### 2.3.3 Le Modèle TLM

Le modèle TLM (Transaction Level Modeling) n'est pas un modèle à proprement parler, mais un ensemble de modèles qui partent de très hauts niveaux jusqu'aux limites du niveau RTL. Ce modèle étant récent, les avis sont partagés quant à son utilité [?].
Différents concepteurs donnent leur avis sur ce niveau [?]. Ainsi Ghenassia, Scuito, Kunkel, Gajski et Mielenz pensent que le TLM peut apporter beaucoup, notamment pour l'accélération de la simulation fonctionnelle du matériel ; mais cela demande encore beaucoup de recherche pour apprendre à s'en servir, avoir des outils et des flots permettant le raffinement du modèle et aussi pour faire la synthèse d'un plus bas niveau à partir d'un modèle TLM (prinpalement vers le niveau RTL).
Lennard pense que c'est un niveau intéressant mais que cela ne changera pas grand chose. Il pense que les concepteurs continueront à optimiser leurs IPs pour des protocoles de communications spécifiques et resteront donc au niveau RTL.
Ce modèle n'étant pas défini précisément chacun le voit comme il veut mais certains tentent d'en faire des définitions. Ainsi, différents niveaux d'abstraction TLM ont été définis [?, ?], ces niveaux changent de noms selon les modèles mais globalement, ils recouvrent la même idée.
Les différents niveaux d'abstraction définis [?, ?] sont dans un degré décroissant d'abstraction (figure 2.3) :
- modèle d'assemblage des composants,
- modèle d'arbitration du bus,
- modèle précis du bus,
- modèle de calcul précis.

#### Modèle d'assemblage des composants

A ce niveau d'abstraction, les entités représentent des éléments de calculs (Processing Element :PE) et des mémoires qui communiquent à travers des canaux.
Un PE peut être une architecture complexe, un processeur, un DSP (Digital Signal Processor) ou un IP (Intellectual Property).
Les canaux permettent le passage de messages entre les entités. Ces messages représentent uniquement des transferts de données ou des éléments de synchronisation. Il n'y a pas de protocole de communication implémenté.
Les communications par canaux ne sont pas temporisées, alors que les éléments de calcul le sont approximativement. Le temps estimé est annoté grâce à des appels à une primitive de synchronisation indiquant qu'il faut attendre un certain temps avant de

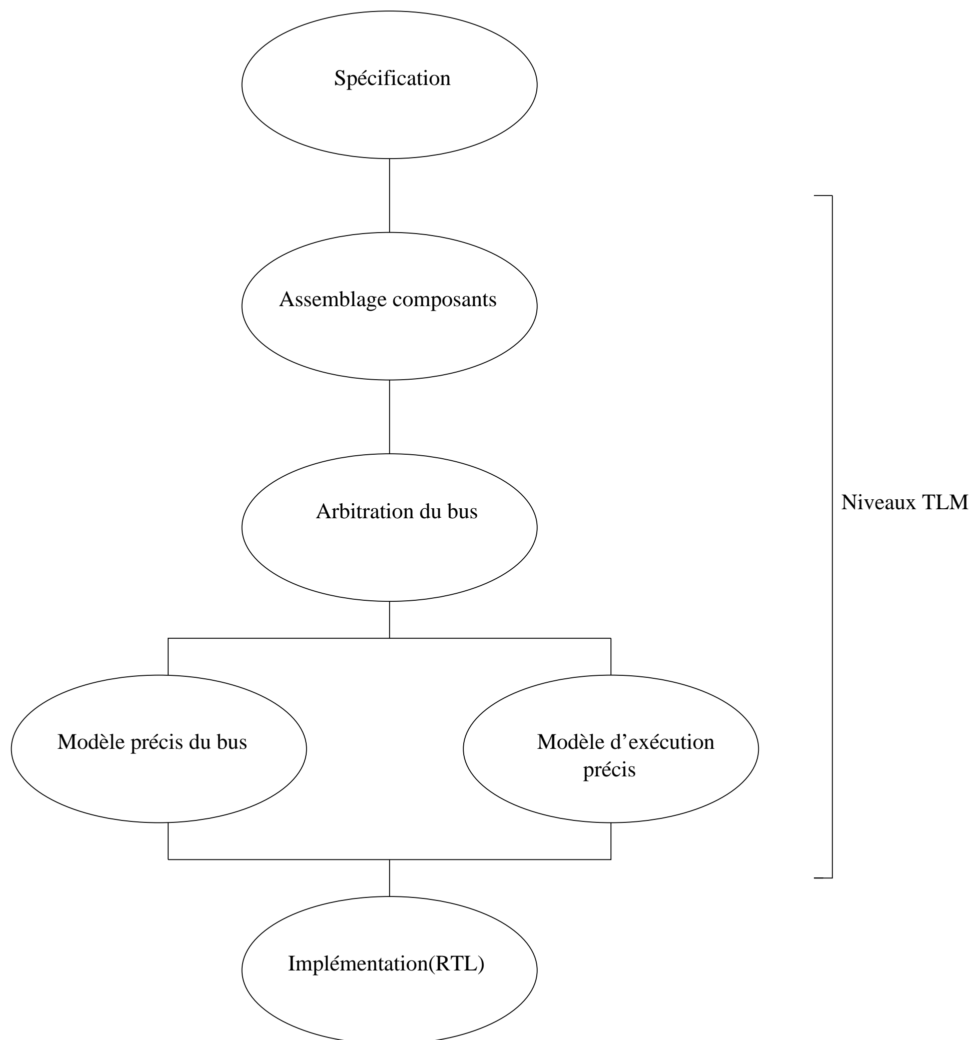


FIG. 2.3 – Les niveaux du modèle TLM

continuer (wait() en SystemC).

**Modèle d'arbitration du bus**

A ce niveau, les bus entre les entités font toujours des transferts de données par messages mais sont représentés comme des entrées/sorties bloquantes ou non bloquantes, sans que le protocole de bus ne soit détaillé. Un arbitre de bus est inséré comme une nouvelle entité. Les temps de communications par bus sont estimés et annotés avec une primitive de synchronisation par transaction. Le modèle d'exécution est le même que celui du modèle d'assemblage des composants.

#### Modèle précis du bus

Ce niveau est nommé "modèle fonctionnel du bus" dans [?, ?].
A ce niveau, le temps de communication est décrit de manière précise alors que le temps des éléments de calcul est approximatif.
Le protocole du bus est décrit précisément. Ce niveau permet de tester la validité du protocole de bus et de l'architecture générale.

#### Modèle de calcul précis

A ce niveau, le temps d'exécution est précis alors que le temps de transfert par bus est estimé. Ce niveau peut être généré à partir du niveau d'arbitration du bus.
Le modèle de bus est le même que dans le niveau d'arbitration du bus.

Ces niveaux sont utilisés aussi bien pour la modélisation que pour la simulation des architectures et des systèmes embarqués.

## 2.4 Les types de simulations/validations de logiciels

La simulation des logiciels (Système d'exploitation + application ou application seule) permet de vérifier que le logiciel ne comporte pas de bogues mais aussi qu'il produit le résultat attendu pour une entrée donnée.
Il existe trois types de simulations de logiciels :
- Simulation fonctionnelle,
- Simulation à l'aide d'un Instruction Set Simulator,
- L'utilisation de modèle de simulation de systèmes d'exploitation.

### 2.4.1 Simulation fonctionnelle

Le but de la simulation fonctionnelle est de vérifier que le logiciel fonctionne bien sans se soucier du système sur lequel le logiciel sera embarqué.
On peut ajouter une notion de temps dans ce type de simulation en y insérant des annotations sur le temps, mais généralement les simulations fonctionnelles ne considèrent que les fonctionnalités de l'application.

### 2.4.2 Simulation à l'aide d'un Instruction Set Simulator

Un Instruction Set Simulator (ISS) est un simulateur de jeu d'instructions qui permet de simuler le logiciel à un très bas niveau. C'est une simulation matérielle au niveau RTL qui prend le code assembleur du logiciel et l'exécute.

Il exécute le logiciel compilé en assembleur du processeur simulé à la vitesse de ce processeur (figure 2.4). Cela permet donc une cosimulation avec le matériel cible au niveau RTL qui "supportera" ses instructions.

Ce type de simulation est donc temporisé, mais ces simulations logicielles sont lentes du fait de la simulation matérielle à un niveau RTL. De plus, pour effectuer ce type de simulation, il faut être déjà avancé dans le flot de conception car il fait usage de détails précis (comme le jeu d'instructions) sur le matériel simulé et d'un compilateur permettant de compiler le code pour le processeur cible.

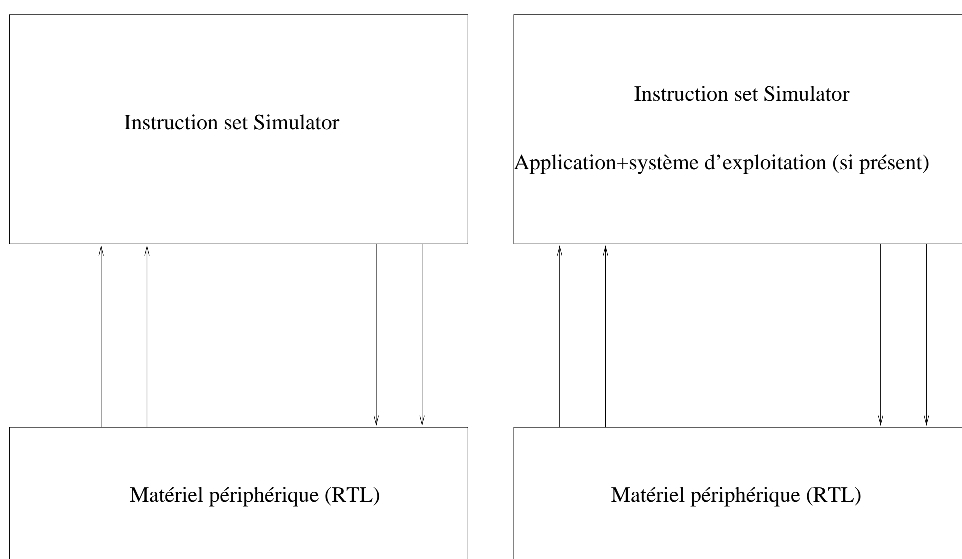


FIG. 2.4 – Fonctionnement d'un ISS

### 2.4.3 Les modèles de simulations de systèmes d'exploitation

Les modèles de simulations de systèmes d'exploitation permettent de simuler l'application avec un niveau système d'exploitation supplémentaire. Cela permet de vérifier que l'application est bien conçue pour le système d'exploitation mais aussi de tester différents paramètres du système tel que la politique d'ordonnancement.

Il existe trois types de simulation de systèmes d'exploitation :
- Simulation native,
- Simulation d'un système virtuel,
- Utilisation de modèle global de synchronisation.

#### Simulation native

Dans une simulation native, le logiciel et le système d'exploitation sont compilés pour la machine hôte de simulation.

Dans ce type de simulation, il s'agit donc de vrai systèmes qui sont compilés vers la machine de simulation pour les tests. Cela est notamment le cas dans VxSim développé par la société Wind River qui permet la simulation du système VxWork [?] sur une machine hôte sous Unix ou Windows NT.
Faire ce type de simulation implique que le futur système d'exploitation soit choisi et qu'une version de ce système soit spécifiquement développée pour faire ce type de simulation.

**Simulation à l'aide d'un système virtuel**

Un système virtuel fournit les services d'un système d'exploitation classique, mais n'est pas un "vrai" système d'exploitation, dans le sens où un tel système ne permet que la simulation. Ce n'est pas un système d'exploitation pouvant être utilisé dans le système final.

Ces systèmes permettent donc d'observer ce qui se passe avec différentes configurations du système. Il est possible d'observer le nombre de changements de contexte, le temps d'exécution des différentes applications et le respect des contraintes temps réel dans le cas de système temps réel.
Ces systèmes fournissent une API[2], il faut donc que les applications que l'on veut simuler utilisent cette API pour que cela fonctionne. Les API étant différentes d'un système d'exploitation final à l'autre, il faut que les applications soient écrites pour le système virtuel puis ensuite modifiées pour mettre l'API du système définitif.

A partir de tel système, il peut être possible de générer le code du système d'exploitation final. Ainsi, dans le système SocOS [?], les appels à l'API SocOS peuvent être remplacés par des appels à l'API d'un vrai système et permettre la génération du logiciel pour le système d'exploitation du système final.

Dans le même genre, Yoo [?], propose de générer automatiquement un modèle de système d'exploitation vers un processeur virtuel dans le but de réduire l'écart entre le système de simulation et le système final, mais cela ne permet pas de simuler des systèmes d'exploitation commerciaux ou autres que le système qu'il propose. Il utilise le même principe que dans la thèse de Lovic Gauthier [?] qui génère le système d'exploitation vers des processeurs cibles, c'est à dire qu'il utilise des bibliothèques de code spécifique à chaque processeur. Préalablement, il faut avoir écrit le code à mettre dans la bibliothèque ce qui demande un travail important pour obtenir une grande diversité de processeurs.

Une autre approche peut être de respecter une API standard de systèmes d'exploitations. Ceci a, par exemple, été réalisé [?] au moyen d'un système d'exploitation virtuel respectant le standard $\mu$ICRON 4.0 [?] . Le standard $\mu$ICRON 4.0 est un ensemble d'API pour systèmes d'exploitations standardisé au japon notamment. Ce standard a été conçu spécialement pour les petits et moyens systèmes embarqués et inclut des pro-

[2]Application Program Interface : ensemble de protocoles, routines permettant le développement d'applications

blèmes de temps réel. Environ 40 % des systèmes d'exploitation temps réel utilisés au Japon sont basés sur ce standard. Cela permet donc d'effectuer une simulation avec l'application conçue pour un de ces systèmes sans avoir à la modifier. Mais il n'existe pas vraiment de standard sur cela en Europe, excepté le standard POSIX respecté par les systèmes Linux.

#### Utilisation de modèle global de synchronisation

Ce modèle permet de simuler le temps d'exécution du système d'exploitation grâce à une annotation du modèle.
Par exemple, le temps d'ordonnancement d'une tâche peut être calculé par rapport au nombre de tâches prêtes à être exécutées ou alors par rapport à la taille des contextes des tâches. Puis ce temps est ajouté au temps de simulation quand il y a un changement de contexte.
Le modèle global de synchronisation est surtout utilisé dans le domaine des systèmes temps réel, spécialement pour analyser l'effet de l'ordonnanceur dans l'analyse de l'ordonnancement des tâches.

Simuler le logiciel permet de vérifier les fonctionnalités mais ne permet pas de vérifier que le système fonctionnera bien sur l'architecture cible. C'est pourquoi on fait des cosimulations qui sont des simulations du logiciel sur le matériel cible.

## 2.5 Les cosimulations matérielles/logicielles

Les cosimulations permettent la simulation des différentes parties d'un système (matériel et logiciel).
Elles permettent d'effectuer la validation d'un système avant la disponibilité d'un prototype mais aussi la simulation à différents niveaux d'abstraction.
Il existe deux façons de faire une cosimulation (figures 2.5 et 2.6) :
- une seule simulation pour tout le système,
- plusieurs simulations pour chaque élément du système.

Dans les 2 types de cosimulations, les modèles de simulations / validations de logiciels décrits ci-dessus peuvent être utilisés.

#### Cosimulation avec description unique

Dans ce type de cosimulation, on transforme les descriptions des différentes entités à simuler qui sont dans des langages différents (VHDL, SystemC, C, ..) vers une unique description dans un seul langage. Souvent le langage pour la description unique est un langage de programmation classique tel que le C pour que la simulation soit la plus rapide possible.

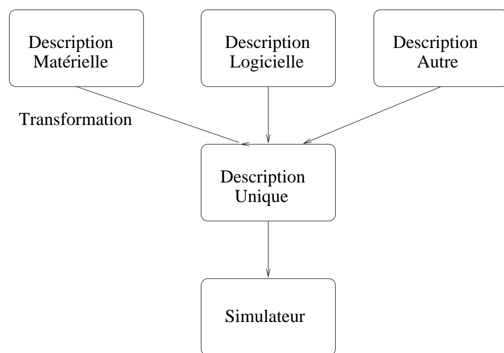


FIG. 2.5 – cosimulation avec description unique

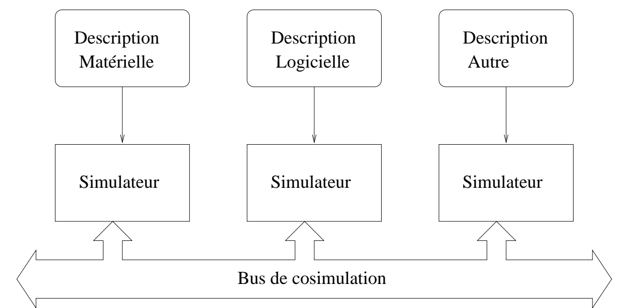


FIG. 2.6 – cosimulation multilangage

Les cosimulations du logiciel avec ISS sont des descriptions uniques du système ; on exécute les instructions sur le matériel cible.

#### Cosimulation multilangage

Ici, les descriptions restent dans le langage dans lequel elles ont été développées. On simule chacune de ces descriptions avec le simulateur adapté et on fait communiquer les différentes simulations avec un bus de cosimulation pour que les simulations s'envoient les informations permettant une bonne simulation. Ces informations peuvent servir à la synchronisation des simulations, mais aussi être des choses à exécuter sur une autre simulation (accès mémoire par exemple).
Ce type de cosimulation permet de développer les simulations dans le langage le plus adapté à chaque situation.

## 2.6 Le temps dans les cosimulations

La notion de temps dans les cosimulations est importante. Sans notion de temps dans la simulation, la simulation est grossière et ne permet pas d'observer grand chose. Sans temps, il est impossible de vérifier le respect de contraintes temps réel par exemple. De même, les effets d'une politique d'ordonnancement ne peuvent être observés, c'est pourquoi il y a dans la plupart des (co)simulations une notion de temps.

Le temps est implicite dans la plupart des simulations matérielles, il est géré dans les langages de description d'architecture tels que SystemC.

Le temps dans la simulation peut être vu de 2 façons différentes dans la simulation matérielle :
- Nombre de cycles d'horloge,
- Temps réel (secondes, millisecondes).

Les simulations bas niveau, RTL typiquement, sont temporisées au cycle d'horloge près. Le temps d'exécution de telles simulations est donc donné au cycle d'horloge.
A un niveau plus élevé (TLM timé), le temps peut être soit exprimé en cycle d'horloge, soit en temps "réel". De plus, au vu du niveau d'abstraction, une simulation au niveau TLM timé temporisé au cycle d'horloge n'est pas très significative. Dire qu'un accès mémoire prend 5 millisecondes peut être suffisant à ce niveau.

Dans les simulations logicielles, le temps d'exécution est plus difficile à définir, car celui-ci dépend du système sur lequel il va fonctionner.
Une des solutions les plus répandues pour introduire une notion de temps d'exécution dans le logiciel est d'insérer des appels à une fonction gérant le temps d'exécution (fonction delay() par exemple).
Pour approximer le temps d'exécution du logiciel, il faut :
- compiler le logiciel vers l'assembleur du processeur cible,
- identifier les différentes fonctions dans le code assembleur,
- évaluer le temps d'exécution de chaque fonction en regardant le code assembleur et les spécifications du processeur,
- insérer les annotations dans le code source de l'application.

# Chapitre 3

# Cosimulation avec un système d'exploitation de type Linux embarqué

Pour simuler des systèmes d'exploitation de type Linux embarqué avec du matériel à un haut niveau d'abstraction (TLM timé), nous avons développé une cosimulation qui répond à nos besoins. Nous allons tout d'abord voir les principes de cette cosimulation, puis le protocole de communication défini et enfin quelques éléments sur l'implantation du système virtuel.

## 3.1 Principes généraux de la simulation

Notre but étant de simuler des systèmes d'exploitation de type Linux embarqué, nous pensons que tracer les appels systèmes Linux des applications peut permettre de faire la simulation sans devoir développer une API spéciale.
De même, ne voulant pas modifier l'application, il nous semble que mettre des annotations de temps dans l'application et le système d'exploitation est quelque chose de contraignant. En conséquence, nous n'allons pas insérer d'annotations dans le logiciel mais plutôt estimer le temps d'exécution du logiciel sur la machine cible par rapport au temps d'exécution sur la machine de simulation.

Pour effectuer cette simulation très tôt dans le flot de conception avec un système d'exploitation, nous allons faire une simulation séparée :
- une simulation matérielle au niveau TLM timé (en SystemC),
- une simulation native du logiciel avec un système d'exploitation virtuel,
- et réaliser la communication entre les 2 entités par un protocole de communication par socket.

Nous allons voir tout d'abord les différentes entités de la simulation, puis comment se déroulent les interactions entre ces entités.

### 3.1.1 La simulation matérielle

La simulation matérielle permet de simuler le matériel et d'exécuter les fonctionnalités matérielles que l'on souhaite. Nous allons notamment exécuter les accès mémoire car c'est probablement ce que font le plus les applications, et c'est une chose qui prend beaucoup de temps, il est donc intéressant de les simuler.

#### Le niveau d'abstraction de la simulation

Le but de la simulation étant de faire une simulation rapide très tôt dans le flot de conception d'un système embarqué, le niveau d'abstraction supporté par celle-ci doit être haut. C'est pourquoi la simulation est effectuée à un niveau TLM timé (niveau abstraction du bus).

**Un haut niveau TLM** Le niveau d'abstraction de base pour faire cette simulation est le niveau TLM timé (niveau d'arbitration du bus) qui se trouve à un haut niveau d'abstraction. Ce niveau va permettre de simuler les principales fonctionnalités que l'on veut faire, principalement lecture/écriture en mémoire, sans que l'architecture ne soit raffinée.

**Adaptation possible à d'autres niveaux d'abstraction** Il est possible d'adapter cette simulation à des niveaux d'abstraction de plus bas niveau, ainsi la simulation peut être effectuée à un niveau RTL. L'intérêt de faire la simulation à un niveau RTL peut être discutable, mais comme pour le moment il n'existe pas beaucoup d'IPs matériel au niveau TLM, on peut prendre des IPs au niveau RTL pour faire la simulation. Dans ce cas, on peut remplacer la mémoire contenant le programme à exécuter par une interface de communication implémentant l'interface qui va envoyer les instructions au processeur simulé.

### 3.1.2 Système d'exploitation virtuel

La cosimulation se faisant à un haut niveau d'abstraction toutes les décisions sur le système n'auront pas été prises, c'est pourquoi nous allons utiliser un système d'exploitation virtuel.

Notre but étant de simuler des systèmes de type Linux embarqué, il faut que notre système virtuel respecte le standard POSIX (qui est aussi présent dans les systèmes Linux embarqué). Il doit aussi offrir des stratégies d'ordonnancement des processus. Pour cela, nous avons choisi de faire fonctionner la simulation sur une machine sous Linux avec les applications natives pour cette machine. Ainsi, on ne va pas modifier le code source des applications pour permettre la simulation.

L'introduction d'un ordonnancement dynamique va permettre de tester différentes politiques d'ordonnancement pour trouver la plus adaptée.

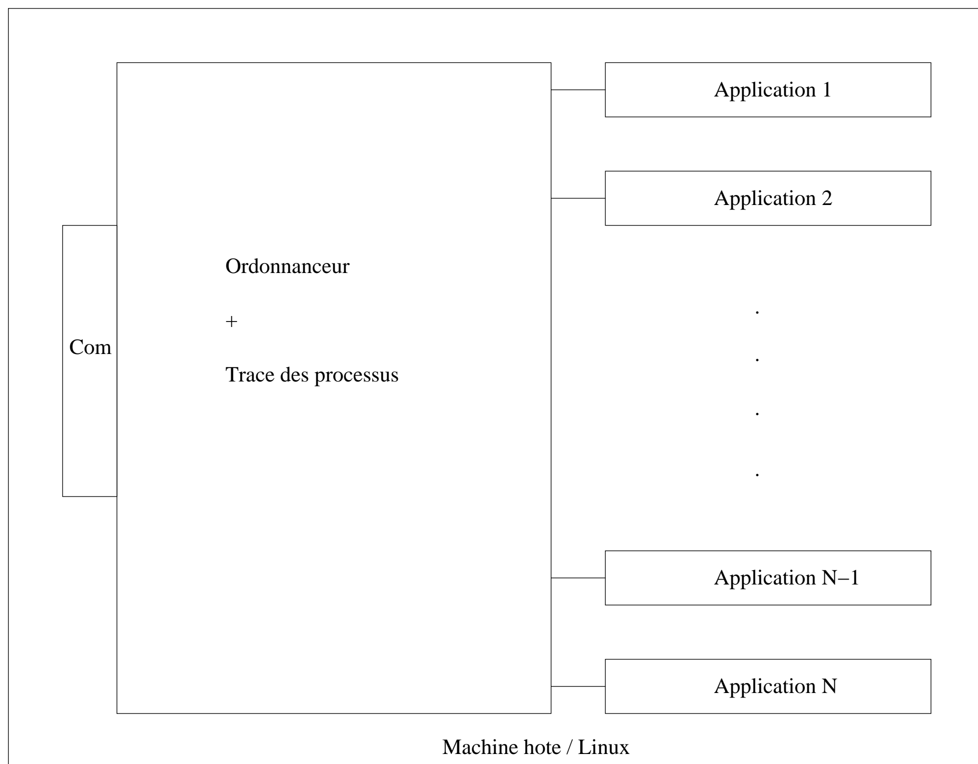


FIG. 3.1 – Organisation de la simulation Logiciel

### 3.1.3 Applications

Les applications que nous allons simuler seront écrites pour un système Linux classique. Aucune modification spéciale ne sera effectuée dans le code source de ces applications pour les faire fonctionner dans la cosimulation.

### 3.1.4 Coordination entre les entités de simulations

#### Traçage des appels systèmes

Pour permettre de simuler des fonctionnalités sur le matériel, essentiellement des lecture/écriture pour le moment, nous allons "tracer" les processus pour connaître les appels systèmes qu'ils font (figure 3.1).
Pour cela, nous allons intercepter les appels systèmes et ceux qui nous intéressent donneront lieu à une simulation sur la simulation matérielle. De même, nous allons suivre les accès en mémoire RAM pour les simuler.
Cela va permettre la coordination entre le système virtuel et l'application.

#### Synchronisation entre le système virtuel et la simulation matérielle

La synchronisation entre le système d'exploitation virtuel et la simulation matérielle se fera par l'intermédiaire du protocole de communication. Deux types de synchronisations sont possibles :

- Synchronisation obligatoire lors de l'invocation de commande sur la simulation matérielle et quand il y a une fin de commande sur la simulation matérielle,
- Synchronisation régulière facultative.

Quand la simulation logicielle invoque une fonction de la simulation matérielle, il faut se synchroniser pour que la simulation matérielle connaisse à quel temps elle doit exécuter la commande. De même, quand la simulation matérielle finit une commande, il faut dire à la simulation logicielle à quel temps la commande se finit. Cela est nécessaire pour savoir quand une application n'est plus en attente d'une fonction matérielle.
Cette synchronisation est obligatoire pour garantir le bon fonctionnement de la simulation.

La synchronisation régulière est, elle, facultative. Elle permettra d'avoir un ordonnancement plus "juste", dans le sens où il s'écoulera moins de temps entre la fin d'une commande sur la simulation matérielle et l'instant où la simulation logicielle sera au courant de la fin de cette commande.
Cette synchronisation ralentissant la simulation par sa répétition, elle est facultative. Un paramètre de la simulation permettra son activation.

### 3.1.5 Le temps dans la cosimulation

#### Temps d'exécution du logiciel sur le matériel simulé

Nous avons fait une simulation logicielle avec une notion de temps. Mais nous ne voulions pas modifier le code source de l'application en y insérant des annotations sur le temps.
Pour avoir une notion de temps dans la simulation sans annotations, nous avons considéré que le temps d'exécution de l'application sur le processeur de simulation est linéaire par rapport au temps d'exécution de celle-ci sur le processeur simulé.
Il faut donc pouvoir estimer le processeur simulé par rapport au processeur de simulation. Une fois cela effectué, nous aurons un facteur de multiplication (f) entre les 2 processeurs. Par exemple si une application met un temps T d'exécution sur le processeur de simulation, on va considérer qu'elle mettra le temps T*f pour s'exécuter sur le processeur simulé (figure 3.2).

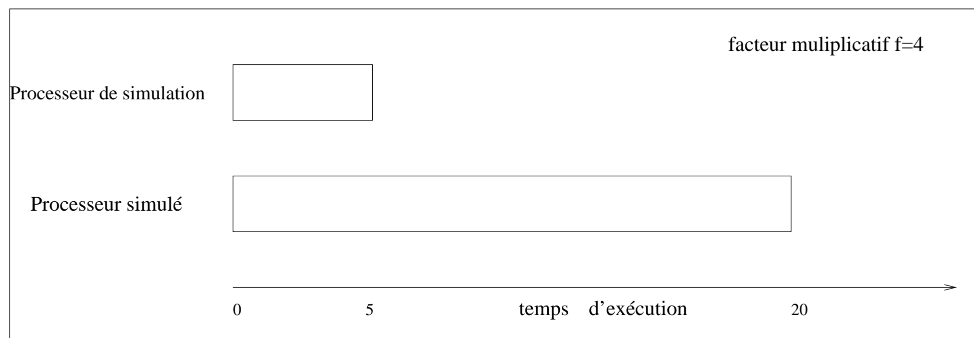


FIG. 3.2 – Illustration du temps d'exécution sur le processeur simulé par rapport au temps sur le processeur de simulation

#### Évaluation du facteur multiplicatif

L'évaluation du facteur multiplicatif f peut être faite de deux manières différentes :
- utilisation de programme de tests,
- comparaison des spécifications techniques.

Si le processeur que l'on veut simuler existe déjà, on peut imaginer écrire une série de programmes de test qui permettront de tester les différentes parties des processeurs et ainsi de calculer le facteur multiplicatif par rapport à la machine de simulation. Les constructeurs des processeurs pourraient fournir dans leurs spécifications les résultats de ces programmes pour permettre de faire une cosimulation comme la nôtre.
Une autre solution, plus complexe, peut être de comparer les spécifications techniques (architecture, jeu d'instructions, fréquence) du processeur de simulation et du processeur simulé. Grâce à ces comparaisons, on pourrait extraire les différences majeures

entre les processeurs et évaluer une valeur du facteur multiplicatif f.

L'intérêt de cette approche est principalement de ne pas avoir à modifier le code source des applications. Une fois que le facteur f est estimé, on peut alors faire beaucoup de cosimulations rapidement. On peut tester très rapidement différentes applications.
Cette approche a des limites ; la principale étant la difficulté de calculer le facteur multiplicatif. Ce calcul n'est pas trivial et la précision ne peut, pour le moment, pas être garantie. Si f est mal évalué, cela va fausser totalement la cosimulation dont les résultats ne voudront plus rien dire.

#### Unité de temps de la simulation

L'unité de temps dans la simulation peut être vue différemment selon le niveau d'abstraction de la simulation matérielle.
Différentes utilisations du temps :
- Le temps réel dans une simulation au niveau TLM,
- Le cycle d'horloge dans une simulation au niveau RTL.

Dans un niveau TLM à un haut niveau d'abstraction, où le système est encore grossier, il n'est pas nécessaire de temporiser la simulation au niveau du cycle d'horloge. C'est pourquoi on peut faire la simulation de telle sorte que le temps de la simulation soit le temps réellement pris par la simulation (secondes, millisecondes).

Mais au niveau RTL , où le système est décrit très précisément, il est logique de temporiser la simulation au niveau du cycle d'horloge. Il est également possible de modifier l'architecture pour la mettre en temps réel.

La simulation logicielle s'adaptera à l'unité de temps. Un paramètre de la simulation logicielle permettra de lui indiquer quelle unité de temps il faut utiliser.

## 3.2 Protocole de Communication

Pour faire les simulations séparées, il est nécessaire de définir une façon de communiquer entre les deux simulations. Un bus de cosimulation peut être justifié avec une simulation matérielle au niveau RTL, mais cela semble trop complexe pour une simulation à un haut niveau d'abstraction. C'est pourquoi nous avons défini un protocole de communication par socket.
Le protocole est assez simple à mettre en oeuvre et répond aux besoins actuels de communication entre les simulations. De plus une communication par socket peut permettre de faire une simulation sur des machines différentes. Cela peut être utile si les équipes de conceptions du logiciel et du matériel ne sont pas installées au même endroit.

Ce protocole par socket pourrait aussi autoriser de faire la simulation du logiciel sur une carte comportant le futur processeur cible et de communiquer avec la simulation des périphériques sur une autre machine.

### 3.2.1 Les besoins

Le protocole de communication doit satisfaire deux choses essentielles :

- la synchronisation entre les deux entités de simulation
- la simulation logicielle doit pouvoir appeler des fonctions disponibles dans la simulation matérielle

#### La synchronisation

Dans le cadre de cette cosimulation, on considère que le nombre de cycles effectués par le processeur simulé est une fonction linéaire du nombre de cycles effectués par la machine sur laquelle est faite la simulation.

Comme les simulations matérielles et logicielles sont indépendantes l'une de l'autre (au niveau de l'exécution), il faut qu'elles communiquent entre elles pour se synchroniser.

C'est pourquoi, lors de la communication par socket, les simulations doivent s'envoyer des estampilles temporelles.

#### Appels des fonctions disponibles dans la simulation matérielle

La communication doit aussi permettre l'utilisation des fonctions matérielles comme une lecture/écriture par exemple.

Ces appels prennent un certain temps qu'il faut prendre en compte dans la simulation logicielle.

C'est pourquoi à la fin de l'appel à une fonction matérielle, il faut refaire une synchronisation.

### 3.2.2 Définition du protocole

Le protocole que l'on a défini répond aux besoins exprimés au paragraphe précédent.

Pour permettre un bonne synchronisation et une bonne simulation matérielle, il est nécessaire de savoir quand le matériel doit exécuter certaines commandes. C'est pourquoi quand la simulation logicielle va envoyer une instruction à la simulation matérielle, celle-ci va attendre de recevoir l'instruction suivante pour faire le traitement jusqu'au temps où l'on doit exécuter l'instruction suivante.

### Principe du protocole

La simulation logicielle commande la simulation matérielle, car il ne faut pas que la simulation matérielle soit en avance sur la simulation logicielle.
Il ne faut pas que la simulation logicielle demande une exécution au matériel à un temps que la simulation matérielle a déjà dépassé (Figures 3.3 et 3.4)

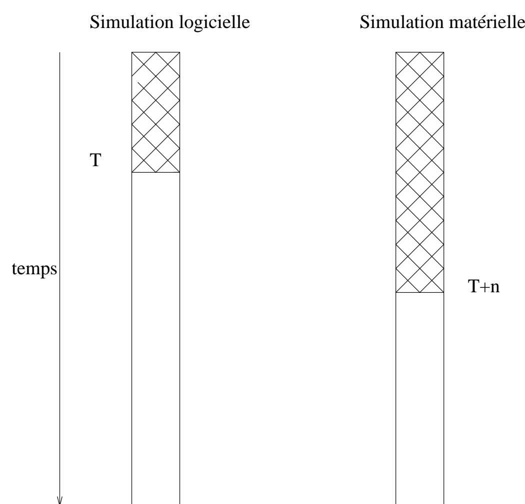


FIG. 3.3 – Ce qu'il faut éviter

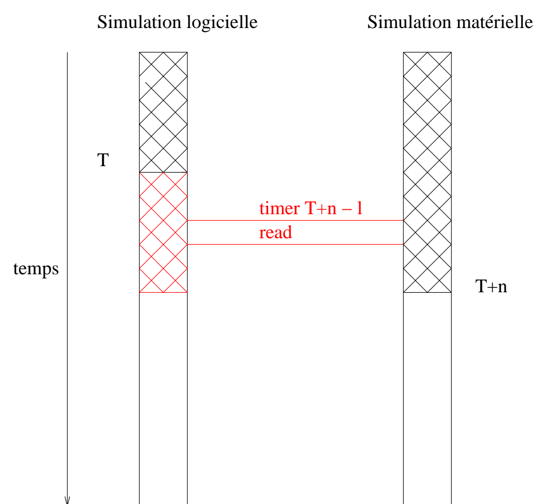


FIG. 3.4 – Ce que cela peut provoquer

Une synchronisation (avec la commande "synchro t") peut être faite régulièrement pour permettre d'avancer régulièrement dans les deux simulations. Cette synchronisation régulière permet aussi de faire un ordonnancement plus fin.

La vue du fonctionnement du protocole change selon le côté où l'on se trouve.

### Du côté de la simulation logicielle

C'est la simulation logicielle qui commande la simulation matérielle. Les commandes demandant l'exécution de quelque chose (tel que read/write) commencent par " timer t" pour indiquer à la simulation matérielle à quel temps la lecture/écriture doit être lancée. Ensuite la simulation envoie la commande "read" ou "write" (ceci est extensible) avec un identifiant de commande. Cet identifiant permet d'identifier la fin d'une commande. Par la suite, la simulation attend les réponses du matériel pour continuer.

Il y a un autre type de commande qui existe. C'est la commande "execute". Cette commande dit à la simulation matérielle de s'exécuter jusqu'à ce qu'une commande se termine. Cette commande est utile quand toutes les applications sont en attente du matériel.

Quand la simulation est finie, on envoie la commande "end" au matériel pour que la simulation s'arrête. Un exemple de gestion du protocole du côté de la simulation logicielle est présenté (Figure 3.5).

```
Tant qu'il y a des processus non finis
    p <– choisir un processus pret selon la politique d'ordonnancement
    Si p!=NULL
        exécution de p pour un quantum de temps défini
        Si on détecte un accès mémoire
            arreter le processus p
            analyse de l'accés
            envoyer timer Temps
            envoyer la commande nécessaire
            attendre la réponse du matériel
        fin si
    Sinon       //Tous les processus sont bloqués
        envoyer la commande "execute"
        attendre la réponse du matériel
    fin si
Fin Tant que
```

FIG. 3.5 – Pseudo-code de la boucle gérant le protocole coté logiciel

**Du côté de la simulation matérielle**

La simulation matérielle (figure 3.6) est, elle, toujours en attente des commandes de la simulation logicielle.

Quand la simulation matérielle reçoit une commande commençant par "timer t", elle enregistre la commande puis s'exécute jusqu'au temps t indiqué dans la commande timer. A ce temps t, la simulation matérielle connaît toutes les commandes à exécuter jusqu'au temps t, car la simulation logicielle est en avance sur la simulation matérielle, donc quand la simulation logicielle envoie "timer t", c'est que le logiciel est au temps t et qu'on ne reviendra pas en arrière. Si durant ce temps une ou plusieurs commandes se finissent on l'indique à la simulation logicielle avec la commande suivante :

– "timer t" où t est le temps auquel la commande se termine,
– "end n" où n est le numéro de la commande qui se termine.

Cela se finit par un "end" pour rendre la main à la simulation logicielle.

```
Tant que vrai
    mes <– recevoirMessage();
    Si (mes == " timer t")
        temps  <– t;  //on stocke le temps jusqu'où on doit s'exécuter
        //On attend la commande
        mes <– recevoirMessage();
        //analyse de la commande
        //sauvegarder la commande
        //le temps_courant est le temps où la simulation est rendu
        t=temps_courant;
        Tant que (t < temps)      //on s'exécute jusqu'au temps t passé en paramètre de timer
            //Si il y a une commande à lancer au temps t,          on la lance
            Si fin d'une commande
                envoyer "timer t"
                envoyer "end numeroCommande"
            fin si
            sc_start(1);//On simule un pas
            t++;
        fin tant que
        envoyer "end"
    fin si
    Si (mes == "synchro t")
        Tant que (t < temps)
        //Si il y a une commande à lancer au temps t,
        //On la lance
        sc_start(1);//On simule un pas
        t++;
        fin tant que
    fin si
    Si (mes == "execute")
        Tant qu'aucune commande ne finit
            sc_start(1);//On avance dans la simulation
        fin tant que
    envoyer "timer temps_courant"
    envoyer "end numeroCommande"
    fin si
fin tant que
```

FIG. 3.6 – Pseudo-code de la boucle gérant le protocole coté matériel

**Extension des commandes**

Pour l'instant, un nombre minimal de commandes a été défini. Mais les commandes sont extensibles. Ainsi, on peut rajouter des commandes pour simuler des accès à des périphériques, distinguer les accès à la ROM ou à la RAM et autres. De même, on peut adjoindre des arguments aux commandes si l'exécution d'une commande change selon la valeur de son paramètre (accès à des disques durs différents par exemple).

### 3.2.3 Un exemple

Dans l'exemple suivant (Figure 3.7), on peut voir comment se déroule une simulation.

Deux tâches sont ordonnancées par le système d'exploitation virtuel. C'est tout d'abord la tâche 1 qui s'exécute et elle a un write à faire. Le système intercepte le signal de write, le transmet à la simulation matérielle grâce aux commandes "timer T1","write 1". La simulation matérielle avance jusqu'au temps T1 et répond par un "end". On peut donc mettre la tâche 1 en attente. La tâche 2 peut alors commencer. Au temps T2, la tâche 2 lance un read, le signal est intercepté et la commande est envoyée au matériel avec "timer T2","read 2".

Ensuite les deux tâches sont en attente, l'ordonnanceur ne peut donc plus lancer de tâche. Il émet alors un "execute" pour signaler à la simulation matérielle de s'exécuter jusqu'à la fin d'une commande.

Le système reçoit le signal de fin de la commande 1 au temps T3. L'application 1 est exécutée jusqu'au temps T4, où elle arrive à la fin du quantum de temps qui lui était alloué.
Dans l'exemple, la synchronisation régulière est activée, donc à la fin du quantum de temps, le système se synchronise avec la simulation matérielle. A cette synchronisation, la simulation logicielle reçoit la fin de la commande 2. On donne alors la main à l'application 2 qui se finit. Ceci permet de relancer l'application 1 qui va déclencher un accès mémoire read au temps T6. Le read est lancé sur la simulation matérielle. Comme, il ne reste qu'une application, il faut faire une commande "execute" pour que la commande se finisse au temps T7 sur la simulation matérielle. Puis, on relance l'application 1.

## 3.3 Implantation

### 3.3.1 Organisation de la simulation matérielle

La simulation matérielle (figure 3.8) va avoir une partie générique quelque soit le niveau d'abstraction et une partie spécifique au niveau d'abstraction de celle-ci.

La partie générique va être la gestion du protocole de communication par socket. La gestion sera toujours la même pour recevoir et envoyer des messages à la simulation

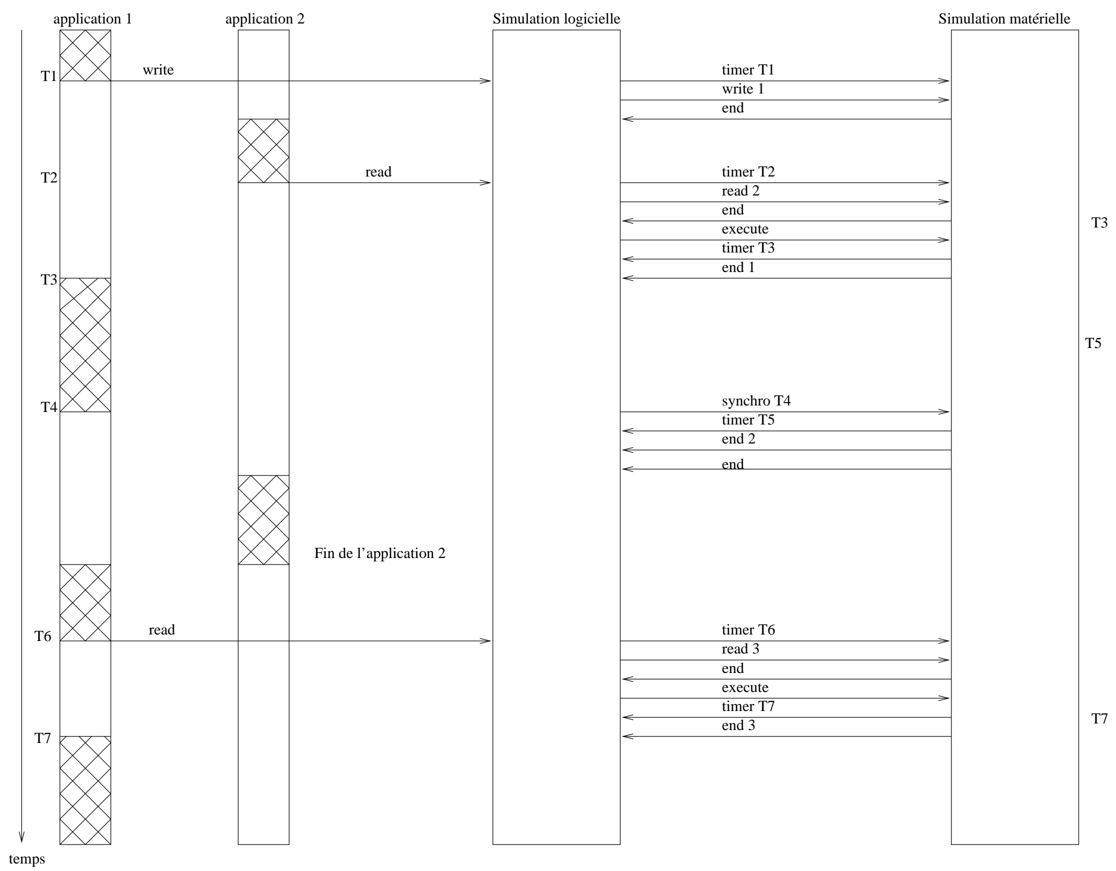


FIG. 3.7 – un exemple de fonctionnement du protocole de communication

logicielle. L'interface de cette partie sera donc toujours la même et une analyse de génie logiciel pourra rendre cette partie générique.

La partie spécifique au niveau servira à envoyer les commandes à la simulation, car cette partie dépend de l'implémentation du matériel. Au niveau RTL notamment, il faudra envoyer les commandes qui correspondent à un jeu d'instructions qui diffère pour chaque famille de processeur.

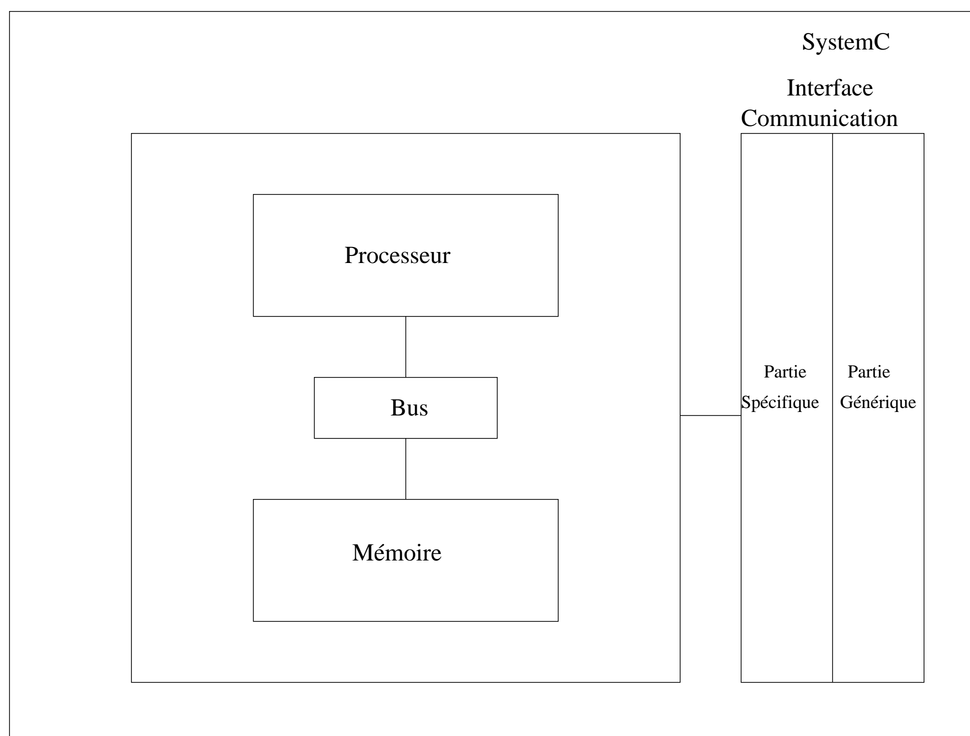


FIG. 3.8 – Organisation de la simulation matérielle

### 3.3.2 L'ordonnancement des applications

L'ordonnancement dynamique se fait grâce à un système d'exploitation virtuel. Il permet de simuler les fonctionnalités d'un système d'exploitation.Le système d'exploitation dispose des politiques d'ordonnancement classiques (FIFO et Round-Robin pour le moment).
L'ordonnancement se fait soit quand un processus a atteint le quantum de temps qu'il lui était alloué, soit quand il a un appel système bloquant (typiquement, les read/write à exécuter sur le matériel).

Pour permettre un ordonnancement au plus proche de la réalité, on peut se synchroniser régulièrement avec la simulation matérielle. Cela permet de connaître avec le

moins de retard possible la fin d'un accès au matériel dans la simulation matérielle.

Les algorithmes d'ordonnancement de bases ont été implémentés pour le moment, c'est-à-dire :

– FIFO : First In First Out, les processus sont exécutés par ordre d'arrivée,
– Round-Robin (tourniquet) : un processus est ordonnancé pendant un quantum de temps, puis un autre etc ..

Dans le futur, on pourrait implémenter la politique d'ordonnancement de Linux.

### 3.3.3 Librairie ptrace

Le traçage des appels systèmes permet de connaître tout ce que font les applications que l'on simule, ainsi de transmettre les informations qui nous intéressent à la simulation matériel, comme les accès read/write.

La librairie ptrace est une librairie système permettant à un processus père de suivre le comportement de ses processus fils.
Cette librairie comprend une fonction :

```
long ptrace(int requete,int pid,void *addr,int data)
```

L'appel à ptrace fournit au processus parent le moyen de suivre l'exécution de son fils ayant comme identifiant le pid(numéro) passé en paramètre. Cela permet notamment de contrôler l'empreinte mémoire du processus suivi. Suivant la requête passée en paramètre de ptrace, différentes actions sont possibles :

– Suivre les appels systèmes,
– Faire une exécution pas à pas (instruction par instruction),
– Aller lire/écrire des informations dans l'espace mémoire d'un processus.

Cette librairie informe par un signal le processus "superviseur" de la survenu d'un événement. De cette manière, le processus "superviseur" peut analyser ce que fait le processus suivi et agir en conséquence s'il veut modifier une action par exemple.

Pour des raisons évidentes, nous avons repris le code source de strace [?] qui est un utilitaire linux classique destiné à suivre les appels systèmes du processus suivi. Nous l'avons adapté à nos besoins, notamment pour la communication par socket mais aussi pour l'ordonnancement des processus.

### 3.3.4 Connaître les accès en RAM

Pour réaliser la simulation, il peut être intéressant de connaître les accès en mémoire RAM effectués par les applications simulées.

Pour faire cela, il faut être capable de détecter tous les accès en RAM.

Contrairement à gtrace [?] qui trace les accès aux pages mémoires, nous voulons connaître chaque accès mémoire.

Un appel système permet de gérer les droits d'accès à la RAM, il s'agit de la fonction mprotect dont la signature est la suivante :

```
int mprotect(const void * addr,int *len,int prot)
```

Cet appel système permet d'indiquer avec les droits PROT, les droits accordés sur la page contenant l'adresse addr, page dont la taille est 4096 octets sur architecture x86. Les droits pouvant être accordés sont :
- PROT_READ : droit en lecture,
- PROT_WRITE : droit en écriture,
- PROT_EXEC : droit en exécution,
- PROT_NONE : aucun droit sur la page.

Quand des droits ne sont pas respectés, par exemple si un processus essaie d'écrire sur une page alors que la page est bloquée par PROT_NONE cela provoque une segmentation fault (signal SIGSEGV).

Le principe pour connaître tous les accès mémoires est donc le suivant :
- Bloquer toutes les pages dont les applications ont besoin,
- Mettre en place un handler pour SIGSEGV (Segmentation fault),
- Quand on intercepte un SIGSEGV, on passe l'exécution en mode pas à pas,
- On débloque l'accès à la mémoire, on exécute un pas puis on rebloque l'accès à la mémoire,
- On se remet en mode PTRACE_SYSCALL.

Voilà ce que cela donne en pseudo-code (Figure 3.9)

```
void fault_handler(int sig,siginfo_t *sip,void *notused)          {

    long page;     //On recherche la page qui a provoqué la faute
     page=(long) sip->si_addr & (PAGESIZE-1);     //On récupère le numéro de page
     mprotect(page,PAGESIZE,PROT_READ|PROT_WRITE|PROT_EXEC);
    //Envoie de sigtrap
     kill( pid,SIGTRAP);        //SIGTRAP va provoquer le reveil de la boucle de strace
}

//Dans la boucle de strace
//on change le mode de ptrace
//Mode pas à pas
ptrace(PTRACE_SINGLESTEP,pid,..);
//Attente de la fin du pas
wait4();
//Remise en mode normal
ptrace(PTRACE_SYSCALL,pid,..);      //Mode appel système
```

FIG. 3.9 – Le code pour tracer la RAM

# Chapitre 4

# Test et validation

Quelques tests ont été faits pour voir si cela fonctionnera et tester l'adaptation à différents niveaux d'abstraction. Mais il est important de valider l'approche grâce à un protocole de validation brièvement décrit ici.

## 4.1 Architecture de tests

### 4.1.1 Au niveau TLM

Pour tester cette approche avec un modèle TLM haut niveau, il a fallu trouver une architecture au niveau TLM. Comme ceci n'est pas fortement développé pour le moment, nous avons mis au point une architecture très simple (figure 4.1) permettant de tester les accès en lecture/écriture en ROM.

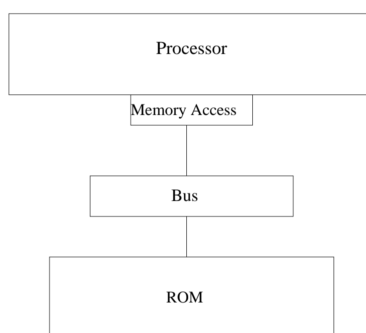


FIG. 4.1 – Une architecture simple au niveau TLM

Cette architecture comporte un processeur quelconque et permet de faire des accès en lecture/écriture.
De plus, le modèle est aisément extensible pour implémenter une architecture multi-processeur et multi-mémoire. Il a notamment permis de tester le protocole de communication et la synchronisation entre les différentes simulations.

### 4.1.2 Au niveau RTL

Au niveau RTL, nous avons choisi un processeur "académique" pour faire nos tests, le processeur DLX [?]. Ce processeur est composé de 5 niveaux de pipelines, d'une ROM qui contient les instructions et d'une RAM (figure 4.2).

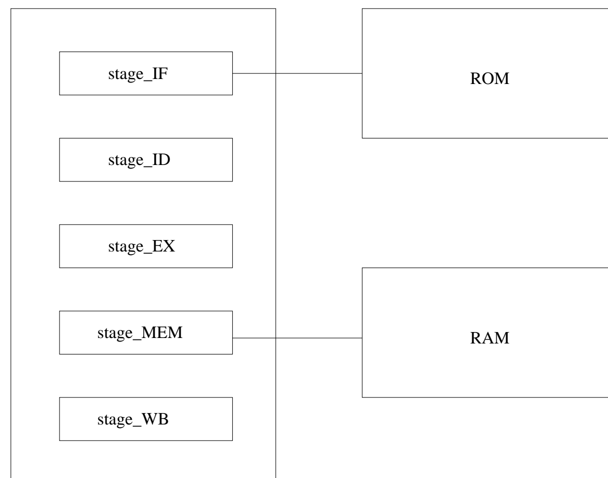


FIG. 4.2 – Le processeur DLX

Le processeur va charger une instruction contenue dans la ROM à chaque cycle d'horloge, pour utiliser le processeur il faut donc lui envoyer des instructions qu'il comprend, c'est pourquoi l'adaptation du processeur avec notre modèle consiste à modifier cette mémoire pour envoyer les instructions nécessaires au bon moment.
Le principe est le suivant, on remplace la ROM par l'interface de communication avec la simulation logicielle (figure 4.3).
Quand l'interface doit envoyer une lecture/écriture en mémoire, elle envoie l'instruction DLX requise, sinon le processeur ne fait rien suite à une instruction NOP.

## 4.2 Validation de l'approche

Il est nécessaire de pouvoir valider l'approche et d'avoir des résultats pour bien voir ce qu'une méthode de ce type peut apporter.
Les résultats à observer sont les suivants :
- Temps d'exécution estimé des applications,
- Taux d'erreur du temps d'exécution par rapport à une exécution au niveau RTL,
- Taux d'erreur par rapport à une exécution réelle,
- Validation de l'approche par système virtuel,
- Rapidité de la cosimulation.

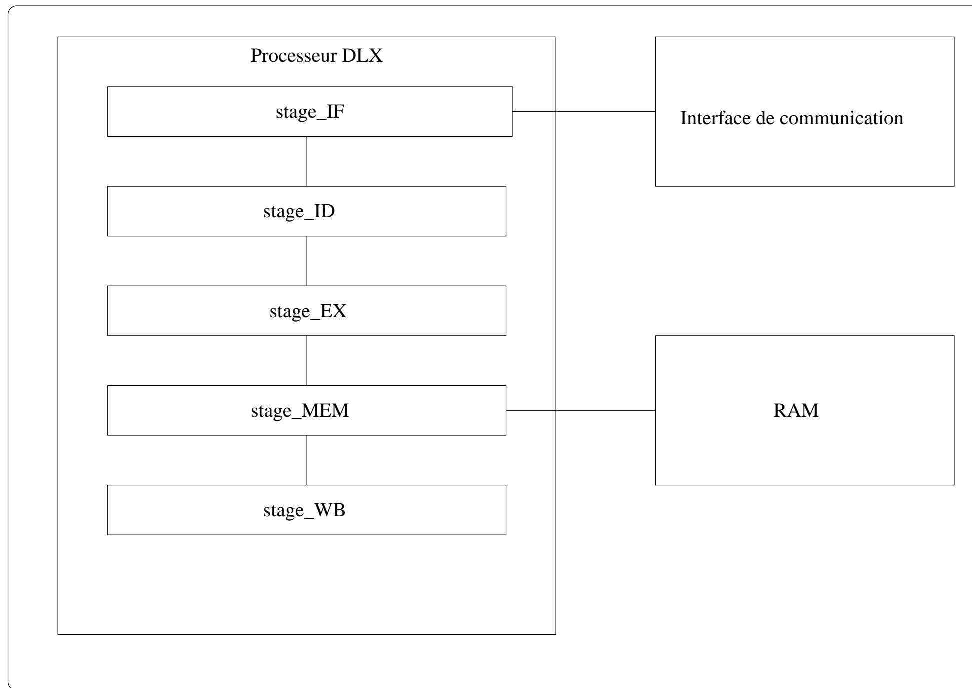


FIG. 4.3 – Le processeur DLX modifié pour être utilisé dans la cosimulation

### 4.2.1 Calcul des taux d'erreurs

L'estimation du temps d'exécution de l'application sur le processeur simulé par le facteur multiplicatif engendre forcément une erreur par rapport au temps d'exécution sur la machine réelle et sur un ISS. Il faut que ce taux d'erreur soit le plus bas possible et qu'il soit acceptable au vu du niveau d'abstraction où l'on se trouve.

Pour calculer le taux d'erreur, il faut simuler une application sur notre cosimulation, faire la même application avec un ISS et comparer les temps d'exécution. On peut faire la même chose avec l'application exécutée sur le vrai processeur. Il faut aussi vérifier que le taux d'erreur est constant quel que soit le type de l'application simulée : application de calcul intensif de nombres réels ou entiers, etc...
Il faudra donc faire un jeu de programmes test permettant de tester les différentes spécificités des processeurs. Pour cela, nous avons notamment envisagé d'écrire un produit de matrice en C et la même chose en assembleur DLX.

### 4.2.2 Validation du système virtuel

Ensuite, il faudra valider l'approche par système d'exploitation virtuel et vérifier que les effets de l'ordonnancement dynamique sont bien visibles. Pour cela, il conviendra de faire d'abord une exécution réelle avec un système Linux embarqué sur un processeur, puis la cosimulation avec le même processeur et la même politique d'ordonnancement

et enfin de comparer les moments où interviennent les changements de contexte. Il faut aussi calculer le temps pris par le système virtuel et le comparer au temps pris réellement par le système embarqué.
Tout cela servira à montrer que la simulation avec le système virtuel simule effectivement le système réel.

### 4.2.3 Temps d'exécution de la cosimulation

Il est aussi nécessaire de considérer le temps d'exécution de la cosimulation. On peut le comparer au temps d'exécution d'une simulation ISS. D'après les premières constatations, la simulation logicielle avec strace (sans tracer les accès en RAM) prend environ dix fois plus de temps que l'exécution des applications seules. L'ajout du traçage des accès en RAM rallongera le temps de simulation, car ils sont nombreux, et le fait de passer en mode pas à pas ralentit la simulation.

Par manque de temps, la validation n'a pas encore été effectuée mais cela devrait être fait prochainement.

# Chapitre 5

# Conclusion et perspectives

Ce travail a permis l'introduction d'une méthode de cosimulation de systèmes embarqués avec des systèmes d'exploitation de type Linux embarqué. L'introduction de la simulation au niveau TLM très tôt dans le flot de conception d'un système embarqué permet de le vérifier. De même, la technique permettant de faire l'évaluation du temps d'exécution des applications sans avoir à annoter les applications permet de faire les simulations très tôt. Une fois qu'un processeur que l'on veut simuler est évalué par rapport au processeur sur lequel on le simule avec le facteur multiplicatif, on peut simuler toutes les applications que l'on veut pour ce processeur. Auparavant, il aurait fallu annoter toutes les applications pour ce processeur. L'exécution d'un système virtuel de type Linux embarqué sur un système Linux permet de faire l'exécution native des applications sur le processeur. Le traçage des appels systèmes permet de simuler les fonctionnalités que l'on veut sur le matériel.

Les perspectives de ce travail sont notamment de simuler les systèmes d'exploitation temps réel et les architectures multi-processeurs. Il pourrait être intéressant de faire des tests sur une architecture définie dans la bibliothèque d'IP Soclib pour montrer l'adaptabilité du système à un niveau RTL. Il faut étendre les appels systèmes gérés par le protocole et tracés par la simulation. On pourrait également faire des mesures de consommation d'énergie sur le matériel en introduisant la consommation moyenne de chaque élément du système (processeur, mémoire, autres..) et ainsi avoir une idée générale des performances du système très tôt dans la simulation. L'insertion de ce travail dans le flot de conception Gaspard de l'équipe est également envisageable avec notamment la génération automatique de l'interface de communication du matériel. Il est aussi nécessaire de développer la méthode d'évaluation du facteur multiplicatif entre le processeur simulé et le processeur de simulation.

# Bibliographie


[1] T.Riesgo, Y.Torroja, and J.Uceda. Design process based on hardware description languages. In *Industrial Electronics*, pages 59–65, Santiago, May 1994.

[2] Laboratoire d'informatique fondamentale de Lille. Gaspard home page. `http://www.lifl.fr/west/gaspard/`, 2005.

[3] Jim Straus. Synthesis from register-transfert level vhdl. 1989.

[4] George Economakos, Petros Oikonomakos, Ioannis Panagopoulos, Ioannis Poulakis, and George Papakonstantinou. Behavioral synthesis with systemc. In *Design, Automation and Test in Europe Conference and Exhibition(DATE'01)*, pages 21–25, Munich, March 2001.

[5] Systemc. World Wide Web document, URL : `http://www.systemc.org/`.

[6] Thorsten Grötker. Modeling software with systemc 3.0. In *6th European SystemC Users Group Meeting*, Stressa, Italie, 2002.

[7] A.Bernstein, M.Burton, and F.Ghenassia. How to bridge the abstraction gap in system level modeling and design. In *Computer Aided Design ICCAD-2004*, pages 910–914, November 2004.

[8] Heinz-Josef Schlebusch, Gary Smith, Donatello Sciuto, Daniel Gajski, Carsten Mielenz, Christopher K. Lennard, Frank Ghenassia, Stuart Swan, and Joachim Kunkel. Transaction based design : Another buzzword or the solution to a design problem ? In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition(DATE'03)*, pages 876–877, 2003.

[9] Adam Donlin. Transaction level modeling : flows and use models. In *CODES+ISSS*, pages 75–80, 2004.

[10] Lukai Cai and Daniel Gajski. Transaction level modeling in system level design. Technical report, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, CA, 2003.

[11] Lukai Cai and Daniel Gajski. Transaction level modeling : An overview. In *Hardware/Software Codesign and System Synthesis*, pages 19–24, October, 2003.

[12] Vxworks. World Wide Web document, URL : `http://www.windriver.com/products/development_tools/ide/wind_river_vxworks_simulator/vxsim.pdf`.

[13] Dirk Desmet, D. Verkest, and Hugo De Man. Operating system based software generation for systems-on-chip. In *Design Automation Conference*, pages 396–401, June 2000.

[14] Sungjoo Yoo, Gabriela Nicolescu, Lovic Gauthier, and Ahmed A.Jerraya. Automatic generation of fast timed simulation models for operating systems in soc design. In *Proceedings of the 2002 Design,Automation and Test in Europe Conference and Exhibition(DATE'02)*, pages 620–627, Paris, France, March 2002.

[15] Lovic Gauthier. *Génération de système d'exploitation pour le ciblage de logiciel multitache sur des architectures multiprocesseurs hétérogènes dans le cadre de systèmes hétérogènes dans le cadre des systèmes embarqués spécifiques*. Thèse de doctorat (PhD Thesis), Institut national polytechnique de Grenoble, dec 2001. (In French).

[16] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. Rtos-centric hardware/software cosimulator for embedded system design. In *Hardware/Software Codesign and System Synthesis*, pages 158–163, September 2004.

[17] $\mu$itron. World Wide Web document, URL : `http://www.asso.tron.org`.

[18] strace. World Wide Web document, URL : `http://strace.sourceforge.net`.

[19] Galderic Punti, Marisa Gil, Xavier Martorell, and Nacho Navarro. gtrace : function call and memory access traces of dynamically linked programs in IA-32 and IA-64 Linux. 2002. `www.ac.upc.edu/pub/reports/DAC/2002/UPC-DAC-2002-51.pdf`.

[20] L.Charest, E.M. Aboulhamid, C.Pilkinson, and P.Paulin. Systemc performance evaluation using a pipelined dlx multiprocessor. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition(DATE'02)*, pages 8–12, Paris, France, March 2002.