Accélération de la simulation modulaire

Mourad Bouache, David Parello, Bernard Goossens

RESEARCH REPORT

Accélération de la simulation modulaire

Mourad Bouache, David Parello et Bernard Goossens

Univ. Perpignan Via Domitia,
Digits, Architectures et Logiciels Informatiques,
F-66860,Perpignan , France

Univ.Montpellier II,
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier,
UMR 5506,
F-34095, Montpellier, France

CNRS,
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier,
UMR 5506,
F-34095, Montpellier, France

Accélération de la simulation modulaire

Mourad Bouache[*,] — David Parello et Bernard Goossens[*]**

[*] *DALI, Université de Perpignan Via Domitia 66860 Perpignan Cedex 9 France, LIRMM, CNRS: UMR 5506 - Université Montpellier 2 34095 Montpellier Cedex 5 France,* [**] *Laboratoire Lifab, Université de Boumerdès-Algérie*

mourad.bouache@univ-perp.fr, david.parello@univ-perp.fr, goossens@univ-perp.fr

RÉSUMÉ. *Les processeurs actuels intègrent de nombreuses unités (calcul, mémoire, contrôle, cœur), dupliquées ou vectorisées. Afin de tester de nouveaux concepts, les architectes utilisent des simulateurs. La principale qualité de ces simulateurs est leur modularité. Sans cette modularité, un simulateur ne peut pas suivre l'évolution rapide des micro-architectures et devient de moins en moins représentatif. La modularité est aujourd'hui pour les simulateurs une qualité plus importante que la vitesse. Nous présentons une méthodologie de vectorisation des simulateurs se substituant à la multiplication des unités ce qui en améliore la vitesse de simulation. L'objectif de cette étude est de : (1) présenter une extension du protocole de communication UNISIM (August* et al.*, 2007) entre les modules du simulateur, (2) introduire une méthodologie de développement simple et systématique pour construire des simulateurs avec duplication des ressources, (3) étudier l'impact de notre méthodologie sur l'accélération du temps de simulation et plus précisément du temps de passage dans l'ordonnanceur du moteur SystemC*

ABSTRACT. *Actual processors integrate many units (computing units, memory, control, core), either duplicated or vectorized. In order to test new concepts, architects simulate them. The main quality for such simulators is their modularity. Without modularity, a simulator cannot follow the microarchitectural evolutions and rapidly becomes less and less representative. Modularity is today a more essential quality for simulators than speed. We present a simulator vectorization methodology replacing the multiplication of the units, which improves the simulation speed. The aim of this study is to : (1) present an extension of the UNISIM (August* et al.*, 2007) communication protocol, (2) introduce a simple and systematic developing methodology to build simulators with duplicated resources, (3) study the methodology impact on the simulator time speedup and more precisely, the time spent in the SystemC scheduler.*

MOTS-CLÉS : *Vectorisation, Simulation modulaire, protocole de communication modulaire, Accélération de la simulation.*

KEYWORDS: *Vectorization, Modular simuation, Modular communication protocol, Simulation speedup.*

unpublished, pages 1 à 20

1. Introduction

Les processeurs actuels intègrent de nombreuses unités (calcul, mémoire, contrôle, cœur). Ces unités sont dupliquées (cœur, cache) ou vectorisées (contrôle superscalaire, opérateurs vectoriels). Afin de tester de nouveaux concepts, les architectes utilisent des simulateurs dont le plus représentatif, SimpleScalar[1], a été conçu il y a 15 ans. Les simulateurs de cette génération souffrent d'une organisation en un seul gros programme (simulateurs monolithiques). Si cela correspondait à l'état de l'art des processeurs de leur époque, ce n'est plus le cas aujourd'hui. Ces simulateurs s'adaptent avec peine aux processeurs multi-cœurs actuels.

L'organisation d'un simulateur monolithique rend difficile l'évolution des parties de la micro-architecture simulée, soit pour les adapter aux produits du marché, soit pour tester l'impact d'innovations. Le développement d'un nouveau simulateur est un investissement conséquent en temps à cause d'un long processus de modélisation, de débogage et de validation. La complexité croissante des micro-architectures à modéliser et leur évolution rapide condamne à plus ou moins court terme les simulateurs monolithiques. Pour ces mêmes raisons, l'adoption d'environnements de simulation modulaires est désormais une nécessité, mais de tels environnements ne peuvent être exploités correctement sans y ajouter une méthodologie de conception.

Pour autant, les simulateurs modulaires sont sensiblement plus lents que leurs équivalents monolithiques. La vitesse de simulation est un facteur critique en particulier parce qu'elle limite l'étendue des solutions qui peuvent être évaluées par les architectes de processeurs.

Notre principale contribution est une méthodologie de développement des modules de simulateur afin d'accélérer le temps de la simulation. Cette méthodologie est basée sur la vectorisation de modules. Cet article compare la performance d'un simulateur modulaire vectorisé à celle du même simulateur sans vectorisation.

Notre étude vise trois objectifs,

1) présenter une extension du protocole de communication UNISIM entre les modules du simulateur.

2) introduire une méthodologie de développement simple et systématique pour construire des simulateurs avec duplication des ressources.

3) étudier l'impact de notre méthodologie sur l'accélération du temps de simulation et plus précisément du temps de passage dans l'ordonnanceur du moteur *SystemC* (Mueller *et al.*, 2001).

Cet article est organisé comme suit : la section 2 présente un ensemble de travaux visant des objectifs liés ou comparables aux nôtres. La section 3 rappelle ce qu'est la simulation modulaire et décrit notre proposition d'extension du protocole de communi-

1. http ://www.simplescalar.com

cation UNISIM. La section 4 explique la méthodologie de vectorisation des modules. Les résultats de la partie expérimentale sont rapportés et discutés dans la section 5.

2. Travaux relatifs

La modularité dans l'organisation d'un simulateur réduit fortement sa performance. Cela vient essentiellement des communications entre les modules (via des ports et des liens) et en ce qui concerne *SystemC*, de l'ordonnancement des processus.

De nombreux travaux ont été menés pour tenter de minimiser ce ralentissement, que nous avons classés dans quatre catégories principales :

1) réduction des entrées de la simulation,

2) optimisation des moteurs de simulation,

3) utilisation des techniques d'échantillonnages,

4) changement de granularité (passage d'un modèle précis au cycle (CLM) à un modèle à transaction avec information de temps : TTLM (Cai *et al.*, 2003)).

Réduction des entrées de la simulation : MinneSPEC [2] est une suite de benchmarks dérivée de SPEC-CPU2000, réduisant la taille du jeu de données pour faire décroître le temps de simulation. La suite MinneSPEC (Kleinosowski *et al.*, 2002) peut également être utilisée pour explorer rapidement un grand espace (différentes variantes microarchitecturales) et cibler les «zones» remarquables de cet espace. On peut ensuite appliquer la suite SPEC pour affiner l'observation sur ces «zones» remarquables.

Optimisation des moteurs de simulation : *SystemC* s'impose de plus en plus largement en tant qu'environnement de simulation pour les systèmes sur puce (SoC) et les processeurs embarqués. Alors que son principal avantage est la modularité et le fait qu'il devienne un standard de fait, l'évolution de l'environnement *SystemC* (de la version 0.9 à la version 2.0.1) montre que l'accent est mis sur l'augmentation des fonctionnalités plutôt que sur l'amélioration de la vitesse de simulation. Pour la simulation au niveau cycle, la vitesse est un facteur critique, la simulation pouvant être extrêmement lente, ce qui affecte l'étendue des solutions qu'il est possible d'évaluer.

FastSysC (Perez *et al.*, 2004) [3] accélère les simulations par un facteur de 2,13 à 3,56 dans *SystemC* 2.0.1. Ce moteur de *SystemC* est conçu pour des simulateurs au niveau cycle. Il ne supporte qu'un sous-ensemble de la syntaxe *SystemC* (signaux, méthodes) qui est le plus souvent utilisé pour ce type de simulateur. La vitesse de simulation a été améliorée d'une part grâce à la réécriture complète du moteur *SystemC* faisant emploi d'une ingénierie logicielle innovante, et d'autre part en appliquant une nouvelle technique d'ordonnancement, intermédiaire entre la technique *SystemC* d'ordonnancement dynamique et les techniques d'ordonnancements statiques. Par rapport

2. www.arctic.umn.edu/minnespec/
3. http ://www.microlib.org/FastSysC

à l'ordonnancement dynamique dans *SystemC*, la technique *FastSysC* évite le réveil de nombreux processus inutiles. De plus, *FastSysC* utilise un algorithme hybride plus simple que celui des ordonnanceurs statiques.

Utilisation des techniques d'échantillonnages : L'échantillonnage de la trace, consiste à sélectionner aléatoirement un grand nombre de petites traces, représentant une très petite fraction (habituellement quelques centaines de millions d'instructions) de la trace totale à simuler. Trois techniques d'échantillonnage ont été proposées par les architectes dans le domaine de la simulation micro-architecturale (Yi *et al.*, 2005) :

1) échantillonnage représentatif,

2) échantillonnage périodique,

3) échantillonnage aléatoire.

L'échantillonnage représentatif tend à extraire d'un benchmark un sous-ensemble de ses instructions dynamiques qui correspond à son comportement général lors de l'utilisation de l'entrée de référence. Avec par exemple la technique SimPoints (Hamerly *et al.*, 2004), un nombre relativement petit de points de simulation est choisi pour être représentatif du comportement de l'ensemble du programme.

L'échantillonnage périodique simule des portions sélectionnées de l'exécution des instructions dynamiques à intervalles fixes. La fréquence d'échantillonnage et la longueur de chaque échantillon sont utilisées pour contrôler le temps de simulation global. SMARTS (Wunderlich *et al.*, 2003) (Sampling Microarchitectural Simulation : échantillonnage de la simulation microarchitecturale) est un exemple récent d'outil basé sur l'échantillonnage périodique. Dans SMARTS, la majeure partie du temps d'exécution vient du "réchauffement fonctionnel" (produire l'effet des parties non simulées). TurboSMARTS (Wenisch *et al.*, n.d.) est une amélioration de SMARTS qui réduit fortement le coût du réchauffement fonctionnel en stockant les états issus des parties non simulées pour leur réutilisation chaque fois qu'il faut réchauffer la même séquence de code non simulée.

Dans l'échantillonnage aléatoire, les résultats de simulation à partir de "N" intervalles choisis au hasard et distribués sont combinés pour produire les résultats de la simulation globale. Pour réduire l'erreur associée à un échantillonnage aléatoire, Conte et al. (Conte *et al.*, 1996) ont suggéré d'augmenter la taille des échantillons et / ou d'augmenter leur nombre.

Le passage partiel d'un modèle précis au cycle (CLM) à un modèle à transaction avec information de temps (TTLM) : On peut passer d'une granularité de cycle à une granularité plus grossière de transactions temporisées (modèle TLM). Le niveau d'abstraction appelé "Transaction Level Modeling" (TLM) a été proposé pour modéliser les architectures SOC, dans le but de permettre le développement par avance de parties d'un logiciel embarqué et de procéder à des analyses plus tôt dans le cycle de conception. Les modules du niveau TLM correspondent à des modèles fonctionnels. Ces modèles ne sont généralement pas temporisés, ce qui ne permet pas de les utiliser

pour estimer la performance de la micro-architecture simulée. Le modèle TTLM (Timed Transaction-Level Modeling) ou TLM temporisé est également conçu à l'aide de l'interface TLM. Dans ce modèle, certaines informations de chronométrage sont ajoutées aux messages pendant la communication. Les informations de chronométrage sont ajoutées lors de la notification de la réponse. En fixant cette valeur, le module cible informe le module source du temps nécessaire pour répondre à sa requête. Le module source réagit suite à cette réponse.

L'ensemble de ces techniques est un compromis entre la précision et la vitesse de simulation. La vectorisation que nous proposons ici est une méthodologie qui peut être utilisée en complément des techniques d'échantillonnages et de réduction des programmes de test.

3. La simulation modulaire

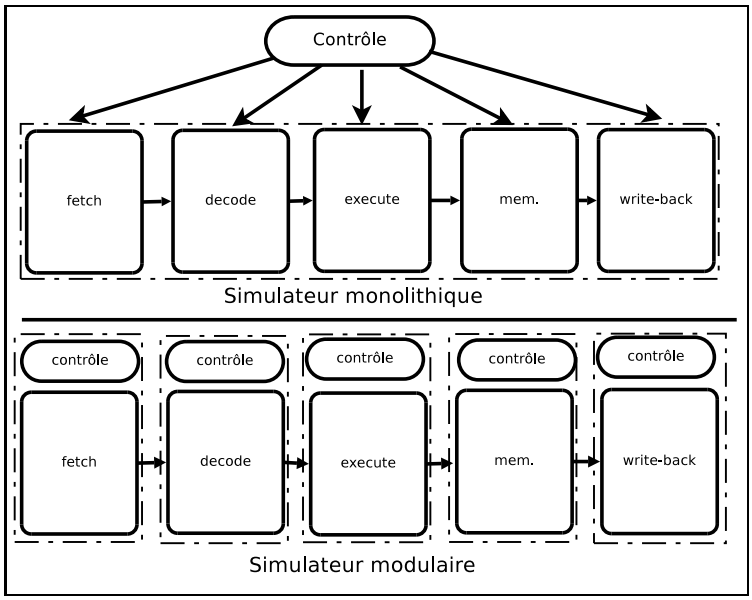

Figure 1 – Simulateur monolithique vs modulaire

Le principal problème des simulateurs monolithiques vient du contrôle du composant simulé, qui est centralisé, comme le montre la figure 1 (en haut). Cela a un impact sur la réutilisation. Si on veut ajouter une UAL ou une hiérarchie mémoire à un simulateur monolithique de processeur, il faut lui ajouter des fonctions de simulation de ces unités additionnelles et incorporer leur contrôle à l'unité de contrôle centralisée du processeur. Cet ajustement du contrôle est délicat et remet le plus souvent en cause le fonctionnement général du simulateur. La modularité vient en grande partie des interfaces de programmation (API). Afin de permettre une véritable réutilisation de module, la plateforme de simulation UNISIM propose une distribution du

code de contrôle à l'intérieur de chaque module (figure 1, en bas). Cette distribution du code de contrôle est possible grâce au protocole de communication UNISIM de niveau cycle.

3.1. *Le protocole de communication au niveau cycle*

Les modules UNISIM sont une modélisation de blocs matériels. Un module est une classe C++ héritant de la classe *module* composée d'un *état* et de *processus*. Le processus d'un module décrit le comportement local du module et emploie pour cela les méthodes de la classe.

Les modules communiquent par le biais des ports et des signaux UNISIM. Au cours d'un cycle de simulation, un module source doté d'un port de sortie (*outport*) connecté au port d'entrée (*inport*) d'un module destination échange 3 signaux : *data*, *accept* et *enable* (voir la figure 2). Le module source envoie une donnée par le signal *data*. Le module destination accepte ou refuse cette donnée par le signal *accept*. Tant qu'un module de destination refuse une donnée, il apparaît figé, comme dans le cas d'un gel de pipeline. Le signal *enable* permet au module source d'autoriser le module destination à utiliser la donnée précédemment acceptée. Ce signal est utilisé afin de synchroniser plusieurs modules destinations lorsqu'une donnée leur est envoyée. Lorsqu'un module source écrit une valeur sur un signal de sortie, le moteur de simulation copie cette valeur sur le port d'entrée associé par la connexion puis réveille le processus du module destination sensible à ce signal d'entrée. UNISIM est défini au-dessus de SystemC. Pour cette raison, une connexion UNISIM regroupe les 3 signaux SystemC *data*, *accept* et *enable*. Chaque écriture sur un signal SystemC fait appel au moteur de simulation de SystemC qui réveille les processus associés au signal.

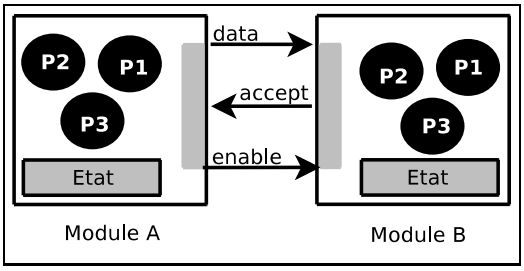

Figure 2 – Connexion des modules UNISIM

3.2. *Les tableaux de signaux*

Dans la pratique, à chaque cycle, un module source envoie plusieurs données de type identique au même module destination. Le concepteur d'un module source peut placer ces données dans un tableau avant de l'envoyer sur un unique port de sortie. Un module destination a la possibilité de n'accepter qu'un sous-ensemble des données

reçues. Les données ainsi regroupées sont soit prises toutes ensemble, soit laissées toutes ensemble. La connexion entre les deux modules se fait par autant de signaux UNISIM qu'il y a de sous-ensembles de données. Dans certains cas, le concepteur des modules ne peut regrouper aucune donnée (parce que toutes les combinaisons d'acceptation / non acceptation sont autorisées) et est contraint d'associer une connexion par donnée, ce qui augmente considérablement le nombre de liens entre modules, donc le nombre d'opérations de contrôle de ces liens.

Afin de réduire le nombre de signaux et le nombre de réveils engendrés par ces signaux, nous avons étendu les ports et signaux UNISIM. Plusieurs valeurs de données (*data*), d'acceptations (*accept*) et de confirmations (*enable*) sont temporairement stockées dans des tableaux. Lorsqu'un module source a écrit toutes ces données dans le tableau temporaire, il envoie explicitement ce dernier. Cela signifie que le code UNISIM effectue l'envoi et non pas le moteur SystemC. Le module destination procède de même avec les acceptations et le module source avec les confirmations. Ces tableaux temporaires sont intégrés aux ports et signaux UNISIM étendus (voir la figure 3).

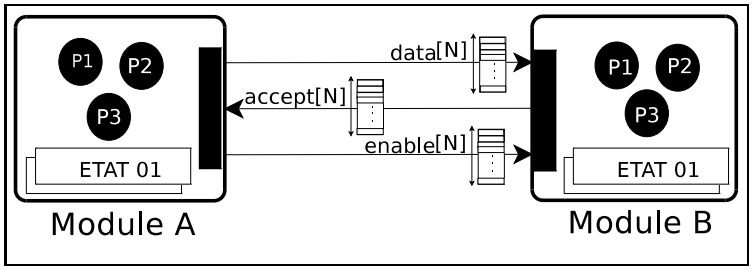

Figure 3 – Communication des modules avec tableaux de signaux

4. Vectorisation des modules

Cette section est divisée en deux sous-sections, d'une part illustrant par un exemple de quelle manière le code de contrôle est distribué dans chaque module et d'autre part présentant le même exemple après sa vectorisation.

4.1. *Distribution du code de contrôle*

La figure 4 montre un extrait de l'implémentation UNISIM d'un module simulant une unité fonctionnelle pipelinée. Les détails d'implémentation, comme par exemple la manière dont est exécutée l'instruction, sont cachés dans l'objet privé `pipeline`. Le code présenté est donc essentiellement du code de contrôle.

```
1  class FunctionalUnit: public module
2  { public:
3     inclock clock;
4     inport<instr> in;
5     outport<instr> out;
6     FunctionalUnit(const char*name): module(name)
7     { sensitive_pos_method(start_of_cycle) << clock;
8       sensitive_neg_method(end_of_cycle) << clock;
9       sensitive_method(on_data_accept) << in.data << out.accept;
10    }
11    void start_of_cycle()
12    { if (pipeline.is_ready())
13        out.data = pipeline.get();
14      else out.data.nothing();
15    }
16    void on_data_accept()
17    { if (in.data.know() && out.accept.know())
18      { if (!pipeline.is_full() || out.accept)
19          in.accept = true;
20        else in.accept = false;
21        out.enable = out.accept;
22      }
23    }
24    void end_of_cycle()
25    { if (out.accept) pipeline.pop();
26      if (in.enable) pipeline.push(in.data);
27      pipeline.run();
28    }
29   private:
30    Fifo<instr> pipeline;
31 };
```

Figure 4 – Module unité fonctionnelle

La classe C++ a trois propriétés publiques : un port d'entrée *in*, un port de sortie *out* et un signal d'horloge *clock*. Le constructeur établit la liste de sensibilité : il indique au moteur SystemC quels sont les processus sensibles aux divers signaux d'entrées.

Lors d'un front montant du signal d'horloge (`sensitive_pos_method`), le moteur *SystemC* lance le processus `start_of_cycle`. L'unité fonctionnelle envoie une instruction sur le port de sortie `out`. Le processus `on_data_accept` peut être réveillé par le moteur *SystemC* lors de la réception d'une donnée (signal `in.data`) ou lors de la réception d'un signal d'acceptation sur le port de sortie (signal `out.accept`). Il accepte la réception d'une nouvelle donnée dans le pipeline si ce dernier n'est pas plein ou si les données précédemment envoyées ont été acceptées par le module suivant. Lors d'un front descendant sur le signal d'horloge (`sensitive_neg_method`), le moteur SystemC lance le processus `end_of_cycle`. Ce dernier retire d'abord l'instruction du pipeline si celle-ci a été acceptée par le module suivant et alloue une nouvelle entrée dans le pipeline pour une nouvelle instruction si celle-ci a été confirmée par le module précédent (signal `in.enable`).

L'état du module est modélisé par l'objet `pipeline` et mis à jour à la fin de chaque cycle.

```
1  class FunctionalUnit: public module
2  { public:
3    inclock clock;
4    inport<instr, NBCFG> in;
5    outport<instr, NBCFG> out;
6    FunctionalUnit(const char*name): module(name)
7    { // sensitive list
8     sensitive_pos_method(start_of_cycle) << clock;
9     sensitive_neg_method(end_of_cycle) << clock;
10    sensitive_method(on_data_accept) << in.data << out.accept;
11   }
12   void start_of_cycle()
13   { for(int cfg=0; cfg<NBCFG; cfg++)
14     {
15      if (pipeline[cfg].is_ready())
16        out.data[cfg] = pipeline[cfg].get();
17      else out.data[cfg].nothing();
18     }
19     out.data.send();
20   }
21   void on_data_accept()
22   { if (in.data.know() && out.accept.know())
23     { for(int cfg=0; cfg<NBCFG; cfg++)
24       { if (!pipeline[cfg].is_full() || out.accept[cfg])
25           in.accept[cfg] = true;
26         else in.accept[cfg] = false;
27         out.enable[cfg] = out.accept[cfg];
28       }
29       in.accept.send();
30       out.enable.send();
31     }
32   }
33   void end_of_cycle()
34   { for(int cfg=0; cfg<NBCFG; cfg++)
35     { if (out.accept[cfg]) pipeline[cfg].pop();
36       if (in.enable[cfg]) pipeline[cfg].push(in.data);
37       pipeline[cfg].run();
38     }
39   }
40   private:
41   Fifo<instr> pipeline[NBCFG];
42 };
```

Figure 5 – Module unité fonctionnelle vectorisée

4.2. *Vectorisation*

La vectorisation de module est une procédure simple et systématique en trois étapes :

1) vectoriser les états et les ports des modules,

2) ajouter une boucle *for* autour du code des processus,

3) ajouter des appels à la méthode *send()* après les boucles *for*, si nécessaire.

La figure 5 montre le module unité fonctionnelle vectorisé.

Après la première étape, les paramètres de *template* des ports d'entrée et de sortie (`in` et `out`) sont complétés par la taille des tableaux de signaux `NBCFG` (lignes 4 et 5). L'état du module, représenté par l'objet `pipeline`, est également vectorisé (ligne 42).

Après la deuxième étape, les boucles *for* ont été ajoutées autour du code des 3 processus (lignes 14, 24 et 35).

Après la troisième étape, les appels à la méthode `send` ont été rajoutés après les boucles *for* des processus `start_of_cycle` et `on_data_accept` (lignes 20, 30 et 31). Le protocole de communication UNISIM non-étendu n'autorise qu'une seule écriture sur les ports de sortie par cycle. Cette écriture d'une valeur sur un port de sortie a pour conséquence de transmettre implicitement cette valeur au moteur SystemC. Afin de permettre l'envoi d'un tableau de signaux au lieu d'un signal unique, la sémantique d'affection des valeurs sur un port UNISIM étendu est différente de celle sur un port UNISIM non-étendu. Les écritures sur un port UNISIM étendu sont stockées temporairement dans un tableau. Lorsque toutes les écritures ont eu lieu, le tableau doit être explicitement envoyé à l'aide de la méthode `send`. Il n'est pas nécessaire de faire appel à la méthode `send` dans le processus `end_of_cycle`. En effet, lorsque ce processus est appelé, tous les signaux ont été propagés et sont connus. Ce processus peut donc mettre à jour l'état du module.

Ce processus de vectorisation peut parraître ici très spécifique à l'environnement UNISIM. Cependant, la vectorisation de modules peut s'appliquer à n'importe quel simulateur modulaire de niveau cycle basé sur *SystemC*. En effet, comme cela est indiqué dans la section 3.1, UNISIM est construit au-dessus de *SystemC* (un signal UNISIM est composé de 3 signaux *SystemC*). Les tableaux de signaux, qui facilitent la vectorisation des modules, sont bien une particularité de l'environnement de simulation cycle à cycle (CLM) d'UNISIM. Le processus de vectorisation en trois étapes sera identique hormis le fait que la gestion des tableaux de données transmis par les signaux *SystemC* devra être réalisé à l'intérieur de chaque module. Même si la méthodologie a été appliquée ici dans le contexte des multi-cœurs, elle s'applique quel que soit le contexte pour peu qu'il concerne la duplication de ressources identiques comme par exemple l'utilisation de multiples unités fonctionnelles entières. Or, les thèmes des principales conférences en micro-architecture montrent que les solutions permettant d'exploiter le nombre croissant de transistors disponibles sur une puce s'orientent vers la duplication de ressources.

5. Le processus expérimental

Le but des expériences réalisées est d'étudier l'impact de la vectorisation des modules sur la vitesse de simulation d'une part et d'autre part d'observer le temps de passage dans l'ordonnanceur du moteur de simulation. Les trois sous-sections suivantes présentent respectivement les simulateurs, les programmes de tests et les résultats obtenus.

5.1. *Les simulateurs*

Le simulateur de référence utilisé est OoOSim, un simulateur générique de processeur superscalaire de degré 4 à exécution dans le désordre avec une mémoire cache d'instructions, une mémoire cache de données, un bus et une mémoire de type DRAM. Le cœur du processeur est composé de 12 modules comme le montre la figure 6. OoOSim a été construit dans l'environnement de simulation UNISIM. L'implémentation de son cœur repose sur plus de 15000 lignes de code réparties dans les 12 modules. La connexion des 12 modules est effectuée par 187 signaux pour une configuration super-scalaire de degré 4.

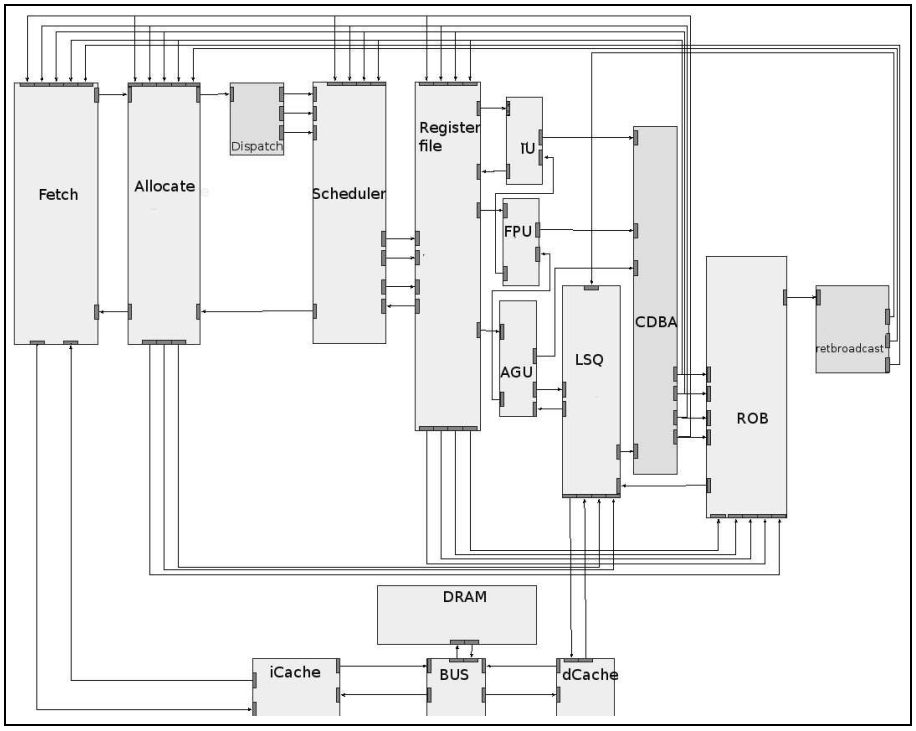

Figure 6 – Les modules OoOSim

Tous les modules du cœur ainsi que les mémoires caches ont été vectorisés. La vectorisation du bus n'est que partielle et concerne son interface avec les mémoires caches.

Sur la base des modules du simulateur OoOSim et les extensions vectorielles, nous avons construit deux types de simulateurs multi-cœurs présentés dans la figure 7 : un simulateur multi-cœurs non vectorisé (de 2 à 64 cœurs), un simulateur multi-cœurs

vectorisé (de 2 à 64 coœurs). Les simulateurs sont disponibles dans la bibliothèque UNISIM [4].

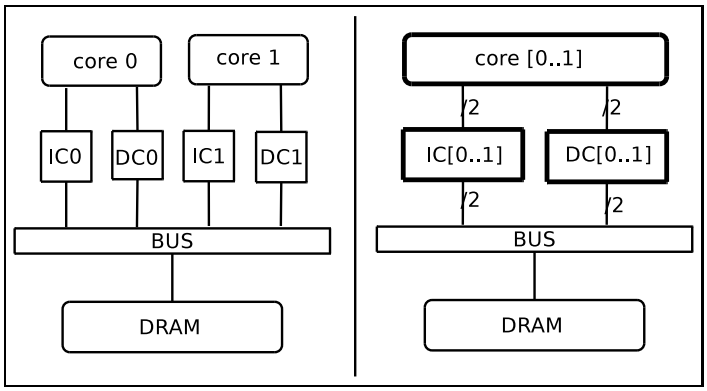

Figure 7 – Vectorisation de simulateur

5.2. *Les programmes de test*

Les simulations ont été effectuées en utilisant la suite des programmes de test MiBench (Guthaus *et al.*, 2001), divisée en six parties ciblant des domaines spécifiques du marché de l'embarqué. Les six catégories sont "véhicules automobiles"(*Automotive*), "les outils de la consommation" (*Consumer*), "la bureautique" (*Office*), "les réseaux" (*Network*), "la sécurité" (*Security*), et les "télécommunications" (*Telecomm*). La table 8 présente les 8 programmes que nous avons retenus. Tous les programmes sont disponibles en code C standard [5].

La hiérarchie mémoire du simulateur implémente un protocole de cohérence de cache de type MESI. Cependant, le simulateur ne permet pas et n'a pas l'objectif de simuler un système complet dit *full-system*. En conséquence, nous avons compilé les programmes de tests afin qu'ils résident en mémoire dans des espaces d'adresses distincts. Même s'ils sont identiques, les programmes simulés simultanément ne partagent aucune donnée ni aucune instruction. De plus, le bus connectant les mémoires caches à la mémoire principale sérialise les requêtes mémoires par un algorithme d'ordonnancement de type tourniquet (*round-robin*). La simulation des programmes de test se désynchronise dès les premiers accès mémoires et les signaux au sein d'un tableau deviennent rapidement différents. Plusieurs centaines voir milliers de cycles peuvent séparer la fin de la simulation de deux programmes identiques se trouvant sur deux cœurs différents. Le fait de simuler des programmes de tests identiquent n'introduit pas de biais dans les résultats et permet de maximiser la charge de tous les cœurs

4. http ://www.unism.org/
5. http ://www.eecs.umich.edu/mibench/

Auto./Industrial	Consumer	Office	Network	Security	Telecom.
susan (edges)	jpeg	stringsearch	dijkstra	sha	FFT
susan (corners)	-	-	-	rijndael	-

Figure 8 – Les benchmarks utilisés dans la suite "MiBench"

sur l'ensemble des cycles simulés. Nous avons réalisé une expérience en simulant différents programmes de tests simultanément. Les résultats montrent que les vitesses de simulation sont très proches de la vitesse de simulation du programme de test le plus long.

Nous n'avons utilisé que 8 programmes parmi les 38 de la suite de tests Mibench. Il y a deux raisons à cela. D'une part les simulations des versions 32 et 64 cœurs peuvent durer plusieurs mois. D'autre part, la séparation des programmes dans des espaces mémoires distincts est réalisée de manière manuelle et la garantie que les programmes (dynamiques) ne se chevauchent pas en mémoire n'est assurée qu'après la simulation.

5.3. *Résultats*

Les simulations ont été effectuées sur un cluster de 30 processeurs *Intel Xeon 5148 dual-core* avec une fréquence d'horloge de *2.33GHz* et une mémoire cache *L2* de *4MOctets*. Chaque simulation est une exécution séquentielle simulée sur un processeur.

La figure 9 montre la vitesse de simulation en kilo-cycles par seconde pour les 8 programmes de tests et pour des simulateurs non-vectorisés dont le nombre de cœurs varie de 2 à 64. Cette figure présente une chute exponentielle de la vitesse de simulation variant de 10 à 14 Kcycles/seconde pour 2 cœurs à moins de 1 Kcycles/seconde pour 64 cœurs.

La figure 10 montre la vitesse de simulation en kilo-cycles par seconde pour les 8 programmes de tests et pour des simulateurs vectorisés dont le nombre de cœurs varie de 2 à 64. La chute de la vitesse de simulation est bien moins dramatique pour les simulateurs vectorisés. Cette vitesse est quasi-linéaire variant de 13 à 23 Kcycles/seconde pour 2 cœurs à moins de 4 Kcycles/seconde pour 64 cœurs. La vectorisation du simulateur accélère considérablement la vitesse de simulation.

La figure 11 montre le gain obtenu par la vectorisation : elle présente le rapport des vitesses des simulateurs vectorisés sur les vitesses des simulateurs non-vectorisés pour un nombre de cœurs variant de 2 à 64. Cette figure permet de mieux quantifier l'impact de la vectorisation pour les simulateurs avec un grand nombre de cœurs.

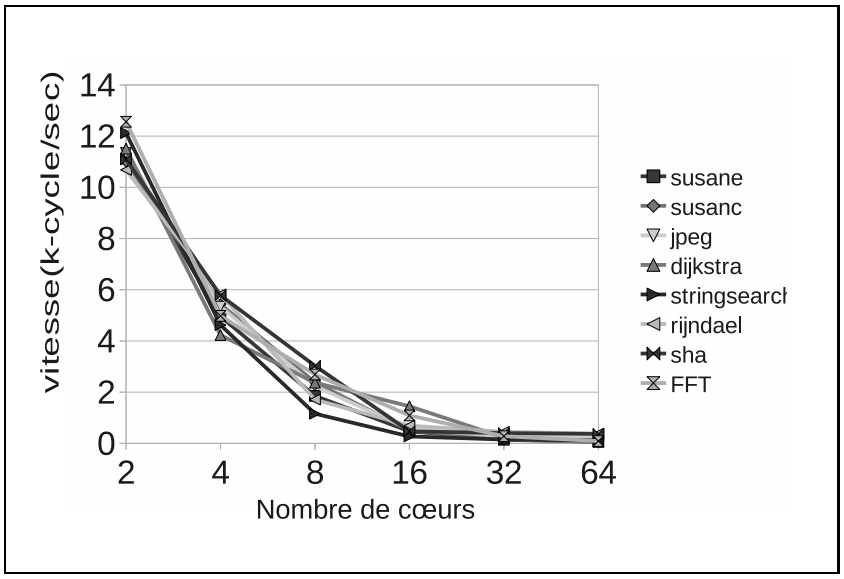

Figure 9 – Vitesse de simulation sans vectorisation

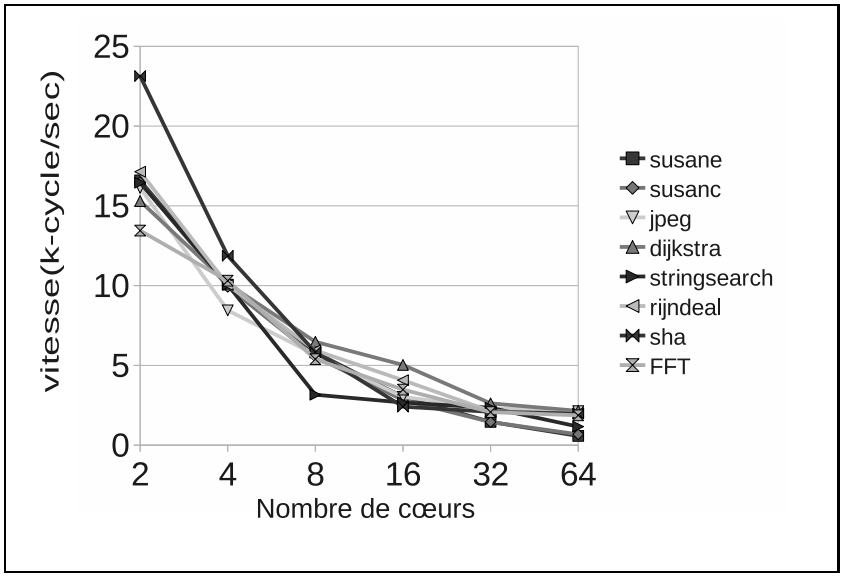

Figure 10 – Vitesse de simulation avec vectorisation

Le moteur de simulation (Panda, 2001) de SystemC ordonnance dynamiquement les processus. SystemC utilise le paradigme d'évaluation/mise à jour *(evaluate/update)*. Un processus est réveillé par SystemC lorsqu'au moins un signal de sa liste

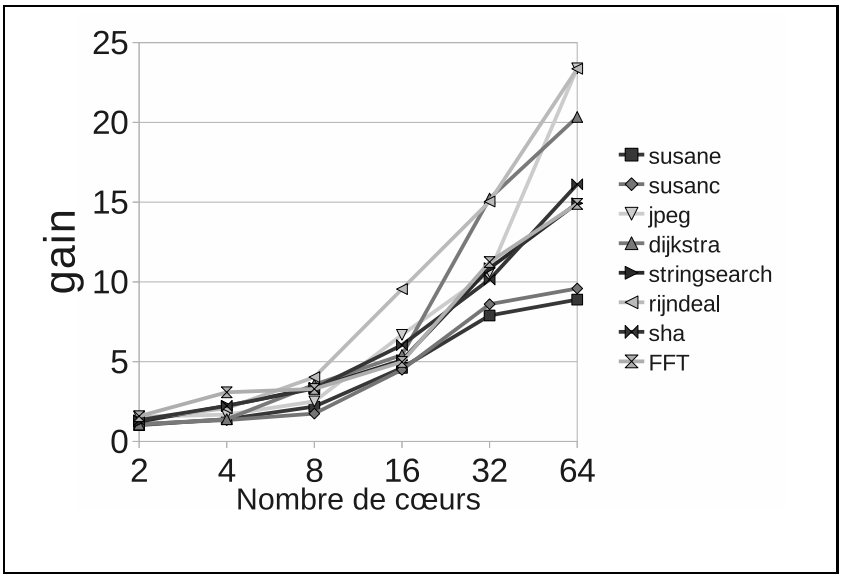

Figure 11 – Gain de la vitesse de simulation

de sensibilité a changé, de façon à régénérer les signaux de sortie du module. Quand un processus écrit sur un signal, la valeur n'est pas immédiatement disponible sur le signal. Ce n'est qu'après que tous les processus se soient réveillés, que les signaux ayant changé sont mis à jour, c'est-à-dire propagés. Leurs nouvelles valeurs sont alors disponibles pour tous les processus. De cette façon, l'ordre de réveil des processus ne détermine pas le résultat de la simulation. Chaque cycle d'évaluation/mise à jour est appelé un "*delta cycle*".

La simulation d'un cycle d'horloge est constituée de plusieurs delta cycles. Au début du cycle d'horloge, les processus sensibles à l'horloge sont réveillés en "parallèle", c'est-à-dire au même delta cycle, et les signaux sont ensuite propagés. Les signaux qui ont changé réveillent les processus qui y sont sensibles. Ces processus modifient à leur tour des signaux qui sont propagés et réveillent d'autres processus, et ainsi de suite jusqu'à ce que plus aucun processus ne soit réveillé. Le cycle simulé se termine donc lorsque le système est devenu stable. La simulation d'un cycle est constituée d'un nombre non connu à l'avance de delta cycles.

Fast SysC (Perez *et al.*, 2004) améliore le moteur SystemC en terme de vitesse, en proposant une nouvelle technique d'ordonnancement, intermédiaire entre la technique d'ordonnancement dynamique "SystemC" et les techniques d'ordonnancement statiques.

Par rapport à l'ordonnancement dynamique *SystemC*, la technique *Fast SysC* élimine des réveils inutiles de processus, tout en utilisant un algorithme d'ordonnancement statique. UNISIM utilise *Fast SysC*.

La vectorisation réduit le nombre de signaux et le nombre de réveils de processus. Dans le but de mieux comprendre l'effet de la vectorisation sur le temps de simulation, le moteur *Fast SysC* a été instrumenté pour mesurer le temps passé dans les processus des divers modules ainsi que le temps passé dans l'ordonnanceur. Cette instrumentation est basée sur l'utilisation du compteur de cycle du processeur exécutant la simulation.

La figure 12 montre la répartition du temps de simulation en nombre de cycles pour les 8 programmes de test et pour les simulateurs non-vectorisés de 2 à 64 cœurs. Cette répartion se divise en deux : le temps passé dans l'ordonnanceur (*temps scheduler*) et le temps passé dans les processus des modules (*temps processus*). Cette figure montre que, dans le cas des simulateurs non-vectorisés, le temps de simulation passé dans l'ordonnanceur *Fast SysC* représente systématiquement au moins un tiers du temps total de simulation.

La figure 13 montre la répartition du temps de simulation en nombre de cycles pour les 8 programmes de test et pour les simulateurs vectorisés de 2 à 64 cœurs. Le temps passé dans l'ordonnanceur représente au plus un dixième du temps total de simulation.

La réduction de temps passé dans l'ordonnanceur à un temps négligeable ne permet pas d'expliquer en totalité les gains importants obtenus par la vectorisation. La vectorisation change le placement des données du simulateur : les états des modules vectorisés ainsi que les signaux vectorisés sont contigus en mémoire. La vectorisation change également l'ordre d'accès à ces données (avec l'ajout des boucles *for*). Une partie du gain obtenu par la vectorisation peut donc être attribuée à une meilleure réutilisation spatiale et temporelle des données du simulateur. A l'aide de l'outil OPROFILE [6], nous avons relevé plusieurs compteurs de performance associé au processeur permettant d'obtenir des statistiques sur les accès mémoires et la prédiction de branchement. L'outil permet de configurer les compteurs de performances du processeur sur lequel s'exécute une simulation afin de compter des événements apparaissant durant l'exécution comme par exemple le nombre de défauts d'accès au cache d'instruction. Le tableau de la figure 14 contient le nombre de débordement des compteurs pour les évènements suivants : le nombre de mauvaises prédiction de branchement (défauts de BR), le nombre de défauts d'accès au cache d'instruction de niveau 1 (défauts L1I), le nombre d'accès en lecture au cache de données de niveau 1 (L1D Load), le nombre d'accès en écriture au cache de données de niveau 1 (L1D Store), le nombre d'accès au cache unifié (instuctions et données) de niveau 2 (Requêtes L2). Les compteurs ont été paramétrés pour déborder tous les 10000 événements. Il faut donc multiplier par 10000 le nombre de débordements pour obtenir le nombre total d'événements.

On constate que la vectorisation dégrade légèrement les accès au cache d'instruction. En effet, l'ajout des boucles dans le code vectorisé augmente sensiblement la taille du code. En revanche, la prédiction de branchement est sensiblement améliorée.

6. http://oprofile.sourceforge.net

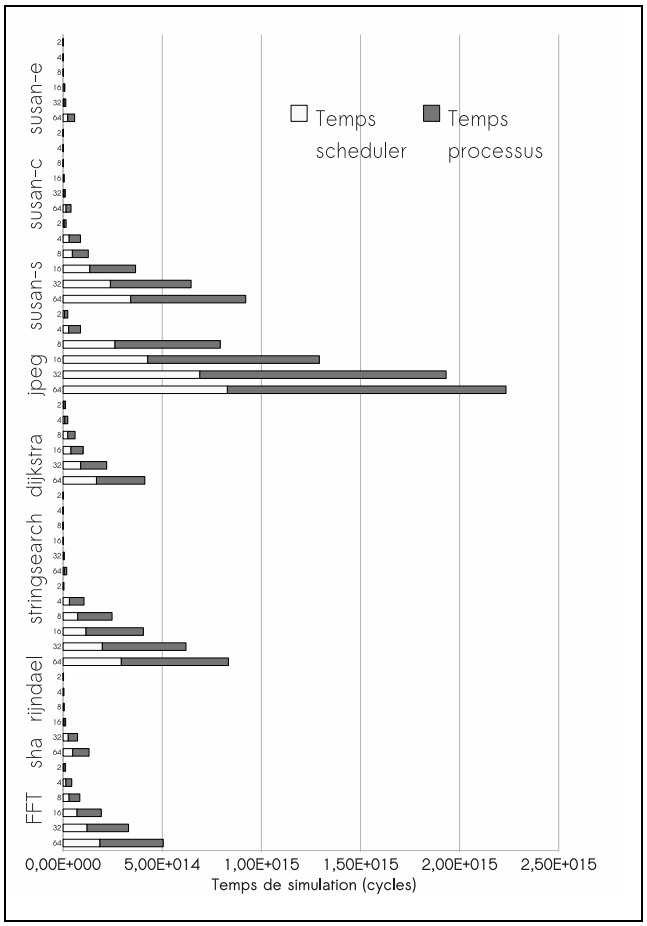

Figure 12 – sans vectorisation

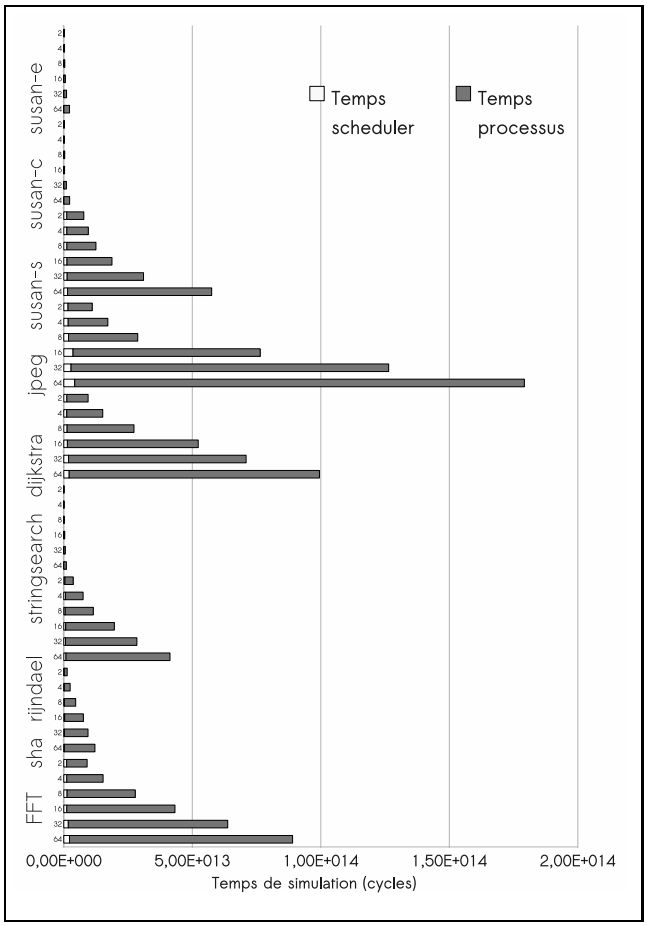

Figure 13 – avec vectorisation

Répartition du temps de simulation

La vectorisation améliore la réutilisation spatiale et temporelle des données du simulateur aussi bien au niveau des mémoires caches que dans les registres. En effet, on constate une réduction du nombre de lectures et d'écritures dans le cache de données de premier niveau.

6. Conclusion

Les environnements de simulation modulaire sont devenus incontournables. Le principal défi de tels environnements est la *vitesse de simulation*. Comme le montre

stringsearch	Défauts BR	Défauts L1I	L1D Load	L1D Store	Requêtes L2
sans vect.	31890	23574	1093610	410583	302191
avec vect.	30588	31263	754185	278311	148323

Figure 14 – Statistiques obtenues à l'aide de l'outil OPROFILE sur les simulations 2-cœurs avec et sans vectorisation pour le programme de test `stringsearch`

cette étude, les environnements de simulation modulaires ne passent pas à l'échelle face à l'augmentation en complexité des micro-architectures. Cette complexité croissante se présente, le plus souvent, sous la forme d'une duplication des ressources comme, par exemple, pour les architectures multi-cœurs.

Ces travaux proposent une méthodologie simple et systématique de développement basée sur un nouveau protocole de communication au niveau cycle : *la vectorisation de module*. Cette méthodologie améliore considérablement la vitesse de simulation et permet aux simulateurs de passer à l'échelle dans le cas d'architectures dont la complexité est basée sur la duplication des ressources.

Contrairement à de nombreuses autres techniques d'accélération de la simulation, cette méthodologie n'est pas un compromis entre vitesse et précision de la simulation. Elle est orthogonale à ces techniques et elle peut donc être utilisée conjointement avec l'une d'entre elles (comme par exemple l'échantillonnage).

Dans de futurs travaux, nous étudierons et quantifierons le passage à l'échelle de la modélisation par transaction temporisée (*timed transaction level modeling TTLM*) avec la duplication de ressources.

7. Bibliographie

August D., Chang J., Girbal S., Gracia-perez D., Mouchard G., Penry D., Vachharajani N., « UNISIM : An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development », *IEEE Computer Architecture Letters*, vol. 1, p. 2007, 2007.

Austin T., Larson E., Ernst D., « SimpleScalar : An Infrastructure for Computer System Modeling », *Computer*, vol. 35, n° 2, p. 59-67, 2002.

Cai L., Gajski D., Transaction Level Modeling In System Level Design, technical report, University of California, Irvine, 2003.

Conte T. M., Hirsch M. A., Menezes K. N., « Reducing state loss for effective trace sampling of superscalar processors », *Computer Design : VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, p. 468-477, 1996.

Guthaus M. R., Ringenberg J. S., Ernst D., Austin T. M., Mudge T., Brown R. B., « MiBench : A free, commercially representative embedded benchmark suite », *WWC '01 : Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, IEEE Computer Society, Washington, DC, USA, p. 3-14, 2001.

Hamerly G., Perelman E., Calder B., « How to use SimPoint to pick simulation points », *SIGMETRICS Perform. Eval. Rev.*, vol. 31, n° 4, p. 25-30, 2004.

Kleinosowski A., Lilja D. J., « Errata to MinneSPEC : A new SPEC benchmark workload for simulation-based computer architecture research », *Computer Architecture Letters*, 2002.

Lane B., Aboulhamid E. M., Baird M., Bhattacharya B., Black D., Dumlogal D., Ghosh A., Goodrich A., Graulich R., Groetker T., Jannsen M., Lavelle E., Kranen K., Mueller W., Schwartz K., Rose A., Ryan R., SystemC 2.0.1 Language Reference Manual Acknowledgements, Technical report, OSCI, 2008.

Mueller W., Ruf J., Hoffmann D., Gerlach J., Kropf T., Rosenstiehl W., « The Simulation Semantics of SystemC », *In Proc. of DATE 2001. IEEE CS*, Press, p. 64-70, 2001.

Panda P. R., « SystemC : a modeling platform supporting multiple design abstractions », *ISSS '01 : Proceedings of the 14th international symposium on Systems synthesis*, ACM, New York, NY, USA, p. 75-80, 2001.

Perelman E., Hamerly G., Biesbrouck M. V., Sherwood T., Calder B., « Using simpoint for accurate and efficient simulation », *ACM SIGMETRICS Performance Evaluation Review*, p. 318-319, 2003.

Perez D. G., Mouchard G., Temam O., « A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling », *Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, p. 10552, 2004.

Petrini F., Vanneschi M., « SMART : a Simulator of Massive ARchitectures and Topologies », *In International Conference on Parallel and Distributed Systems Euro-PDS'97*, 1997.

Shin D., Abdi S., Gajski D. D., « Automatic generation of bus functional models from transaction level models », *ASP-DAC '04 : Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, IEEE Press, Piscataway, NJ, USA, p. 756-758, 2004.

Wenisch T. F., Wunderlich R. E., Falsafi B., Hoe J. C., « TurboSMARTS : accurate microarchitecture simulation sampling in minutes », *SIGMETRICS '05 : Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM, New York, NY, USA, p. 408-409, n.d.

Wunderlich R. E., Wenisch T. F., Falsafi B., Hoe J. C., « SMARTS : accelerating microarchitecture simulation via rigorous statistical sampling », *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, p. 84-95, 2003.

Yi J. J., Kodakara S. V., Sendag R., Lilja D. J., Hawkins D. M., « Characterizing and Comparing Prevailing Simulation Techniques », *HPCA '05 : Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, Washington, DC, USA, p. 266-277, 2005.