Je suis pleinement conscient(e) que le plagiat de documents ou d'une partie de document constitue une fraude caractrisée.
Nom, date et signature :

Programmation parallèle pour la modélisation des systèmes embarqués

Sancassani Gaëtan

June 2014

Résumé Les travaux présentés dans ce document ont été menés dans le laboratoire de recherche de Verimag, lors d'un module Travaux et Études de Recherches de la formation Master Informatique à l'IM2AG. Ce projet a pour but de nous faire découvrir le milieu de la recherche en nous donnant la chance de pouvoir travailler avec des enseignants chercheurs durant quelques mois.
En plus de cela, il nous est donné l'opportunité de pouvoir améliorer nos connaissances sur des domaines spécifiques que l'on ne pourrait pas autant approfondir sans ce module.

Keywords SystemC · sc-during · Scheduler préemptif · temps simulés · FIFO

Table des matières

G.Sancassani
Verimag
Centre Équation - 2, avenue de Vignate 38610 GIÈRES
Téléphone : +33 4 56 52 03 41
Télécopie : +33 4 56 52 03 44

1 Introduction

1.1 Présentation du sujet

Devant la complexification croissante des systèmes sur puce, le besoin d'outils de simulation rapide est devenu une priorité. La méthode de description RTL (Register Transfer Level) utilisée par les langages de description tel que Verilog ou VHDL s'avère bien trop lente.
SystemC est une bibliothèque C++ servant à la description de matériel pour la conception de système sur puce. Ce langage haut niveau permet de modéliser chaque composant par une classe ayant son comportement, ses entrées et ses sorties. En plus de faire gagner en vitesse de simulation, SystemC permet de gagner du temps dans le flot de conception en réalisant simultanément la partie matérielle et logicielle.
Bien que les circuits fonctionnent en parallèle, le scheduler SystemC fonctionne de manière non-préemptif, ne permettant pas le parallélisme d'exécution de la simulation. La bibliothèque **sc-during** a été développée par l'équipe de Matthieu Moy afin de pouvoir accélérer les simulations en utilisants plusieurs coeurs des machines actuelles. Cette librairie propose des fonctions permettant d'exécuter des tâches en parallèle de l'exécution de SystemC.

1.2 Ma contribution

Je me suis penché sur les communications entre module par FIFO. SystemC profite de la classe **sc_fifo** permettant la communication entre modules, mais avec l'ajout du parallélisme amené par **sc-during** , cette dernière a un comportement incorrect lors d'écritures et de lectures concurrentes. J'ai donc dû réaliser l'implémentation d'une FIFO gérant ces accès concurrents.

2 Simulation de système sur puce

2.1 SystemC

SystemC est un langage de description matériel de haut niveau, qui permet d'avoir une vitesse de simulation bien plus rapide que les autres langages de description (ex : VHDL/Verilog).
SystemC, comme tout langage de simulation fonctionne sur un principe de temps simulé représentant le temps pris par la simulation. De par la notion même de simulation, ce temps est différent du temps d'exécution du programme (Wall Clock Time). Le programmeur va donc devoir attribuer arbitrairement des temps d'exécution aux différentes méthodes.
SystemC fonctionne de manière non préemptive, c'est-à-dire que l'exécution d'un module ne sera pas interrompue par le scheduler. Cette bibliothèque permet un parallélisme de simulation en exécutant durant un même temps simulé plusieurs processus, mais physiquement la simulation est séquentielle.

La bibliothèque SystemC permet de représenter les composants de notre système à l'aide de classe **SC_MODULE**. Chaque classe a son propre comportement décrit soit dans une méthode **SC_THREAD** se lançant au début de la simulation, soit dans une fonction **SC_METHOD** étant sensible à certains signaux d'entrées.
De par l'exécution coopérative de la simulation, et l'aspect non préemptif du scheduler, tout processus doit se terminer ou rendre la main au scheduler à l ?aide d ?un appel à la fonction **wait** pour permettre à la simulation de continuer.

2.2 sc-during

La bibliothèque **sc-during** propose aux utilisateurs de SystemC des primitives permettant de bénéficier du parallélisme d'exécution qu'offre leur machine multic$\frac{1}{2}$ur, pour paralléliser, au mieux, les parties gourmandes en ressources.
La contrainte principale de **sc-during** est de ne pas modifier le fonctionnement de SystemC, cependant les primitives de parallélisation que propose cette librairie peuvent engendrer des problèmes d'accès concurrents. Un appel à **during** sur une tâche gourmande va donc la réaliser dans un thread séparé, qui aura accès aux variables de son contexte d'appel. L'entrelacement aléatoire des instructions peut engendrer des comportements anormaux lors de la simulation.
Certaines primitives de SystemC ne sont donc pas utilisables hors du contexte de SystemC tel que **wait** ou **notify**.

2.2.1 During

Cette bibliothèque contient entre autres la fonction **during** qui permet d'exécuter une tâche durant un temps simulé prédéfini, dans un thread séparé de celui gérant SystemC, laissant ainsi le scheduler SystemC continuer l'exécution de la simulation. À la fin de l'exécution de cette tâche ou à la fin de la durée accordée, SystemC se synchronise avec le thread pour assurer une cohérence des données à l'aide le la fonction **join**.

2.2.2 Sc_call

Pour pouvoir appeler du code dans le contexte de SystemC à l'intérieur d'une tâche avec durée, **sc-during** fournis la primitive **sc_call**. Cet appel bloquant envoie la fonction à SystemC et attend que le scheduler SystemC puisse l'exécuter. Une fois la fonction réalisée, la tâche avec durée peut reprendre son exécution, comme on peut le voir sur la figure 1.

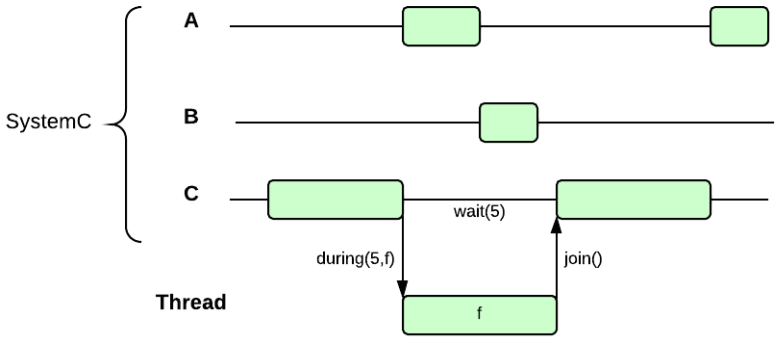

Figure 1 Schéma représentant l'appel à **during**

2.2.3 Extra_time

Le problème de l'appel à **during** est qu'il impose de connaître la durée de calcul de la tâche avant son exécution. Pour allouer un temps supplémentaire on peut utiliser la fonction **extra_time** qui va donc rajouter du temps simulé pour exécuter cette tâche.

2.2.4 Catch_up

Cette primitive permet de synchroniser le temps SystemC avec celui de la tâche avec durée. **Catch_up** bloque le thread appelant jusqu'à ce que le temps SystemC ait rattrapé le temps de la tâche avec durée.

3 Communication entre SystemC et sc-during

3.1 Sc_FIFO

L'objectif est de réaliser un moyen de communication basé sur le principe de FIFO entre des modules SystemC et **sc-during** tel que représenté sur la figure 2. Les premiers tests ont été réalisés à l'aide d'une SC_fifo. La SC_fifo est un buffer de type "premier arrivé, premier sorti" (First In, First Out) proposée par SystemC permettant les communications entre modules.
De par l'exécution séquentielle de SystemC, SC_fifo ne se préoccupe pas des accès concurrents, ce qui pose des problèmes lors des communications entre un module SystemC et un module **sc-during** , ou même entre deux modules **sc-during** .
De plus, les primitives de lectures et d'écritures bloquantes ne peuvent pas être appelées dans un **during** car elles font appel à des **wait**. En effet, ces méthodes bloquantes se doivent de rendre la main au scheduler SystemC pour

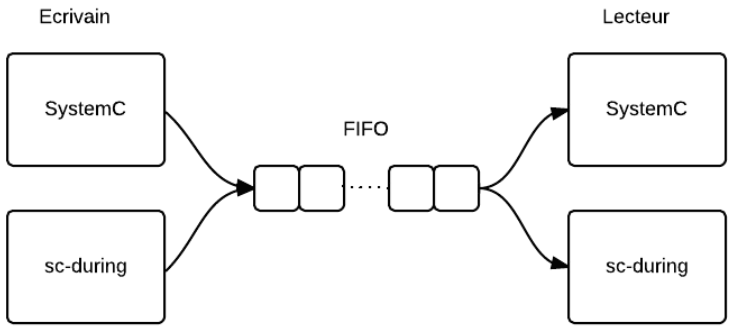

Figure 2 Principe de communication à l'aide d'une FIFO.

attendre un événement (lecture ou écriture) dans le cas d'une lecture sur une FIFO vide ou d'une écriture sur une FIFO pleine.

3.2 During_FIFO

3.2.1 Structure

Sc_fifo n'est donc pas suffisant pour gérer les communications entre nos modules SystemC et **sc-during** , car elle ne gère pas la concurrence des accès. Une FIFO classique C++ ne conviendrait pas non plus de par la spécificité de la communication entre les modules SystemC qui se doivent de communiquer par des interfaces bien définies.
During_fifo doit proposer des fonctions d'écriture et de lecture utilisable aussi bien dans le contexte de SystemC que dans celui de **sc-during** . Pour cela il ne faut pas bloquer la simulations lors de lecture ou d'écriture bloquante.

3.2.2 Protection des accès concurrents

Pour assurer une cohérence des lectures et des écritures concurrentes, les accès critiques sont protégés par des mutex. Chaque thread désirant interagir avec During_fifo devra d'abord verrouiller un mutex. La syntaxe en C++ correspondante à l'initialisation du mutex et à son verrou est :

```
1 // Initialisation de la variable lock verrouillant le mutex fifo_mutex
2 unique_lock<mutex> lock(fifo_mutex);
```

Suite à cet appel, le mutex est verrouillé, toute tentative précédent le déverrouillage du mutex sera bloquante.

3.2.3 Communication SystemC / sc-during

1. Extra_time et Catch_up :
   Ma première approche a été d'utiliser les primitives **extra_time** et **catch_up**, dans un contexte d'une lecture ou d'une écriture bloquante depuis une tâche avec durée. Ces primitives ont pour avantage de pouvoir augmenter le temps estimé lors de notre appel de **during** et de se synchroniser avec le temps SystemC. L'appel à ces deux primitives va permettre à la simulation d'avancer dans le temps et donc ne pas se bloquer.

2. Condition_variable :
   Pour éviter toute attente active, j'ai voulu ensuite utiliser les **conditions_variable**, en mettant le thread en attente lors d'une lecture sur une FIFO vide, ou d'une écriture sur une FIFO pleine. Cependant cette version bloque l'exécution de SystemC. En effet, l'appel à un **wait** sur une variable de condition dans une tâche avec durée ne termine pas le processus et ne rend donc jamais la main au scheduler SystemC bloquant la simulation, comme l'illustre la Figure 3.

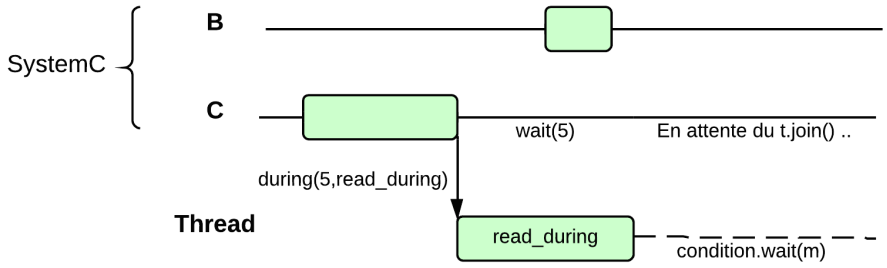

Figure 3 Problème des **condition_variable**

3. sc_call :
   Ma troisième approche consiste en l'utilisation de la primitive **sc_call**, pour permettre au temps simulé de SystemC d'avancer même dans le cas d'une lecture ou d'une écriture bloquante.

```
1  void read(T& c) {
2   unique_lock<mutex> lock(fifo_mutex);
3   while(num_readable() == 0 ) {
4          lock.unlock();
5          sc_call([this] {wait(1,SC_NS)});
6          lock.lock();
7    }
8    lecture(c);
9    // unique_lock fait en sorte de le libérer automatiquement le mutex
10   //    lorsque l'on sort de son contexte d'exécution
11   //    (appel du "unlock" dans le destructeur)
12 }
```

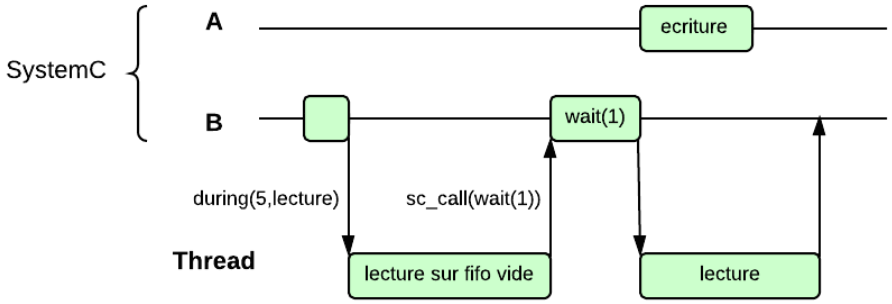

Figure 4 Séquence d'exécution d'une lecture bloquante dans une tâche avec durée.

J'ai pu améliorer cette version en me servant des **sc_event** proposés par SystemC. Au lieu d'attendre un temps prédéfini, il est donc possible d'attendre une notification sur un événement. Il faut cependant bien penser à notifier cet événement en cas de lecture ou d'écriture. Une notification d'événement n'a de sens que dans le contexte de SystemC, et pour éviter un changement de contexte coûteux (un appel à **sc_call**), la bibliothèque **sc-during** dispose d'une classe **async_event_queue** permettant de notifier un événement SystemC depuis le contexte d'une tâche avec durée.
Cette méthode permet donc de réduire considérablement le nombre d'appel à **sc_call**rendant les fonctions de lecture et d'écriture beaucoup moins lourdes. Il aurait été intéressant de trouver un moyen de notifier une tâche avec durée depuis le contexte de SystemC (principe inverse de **async_event_queue**).

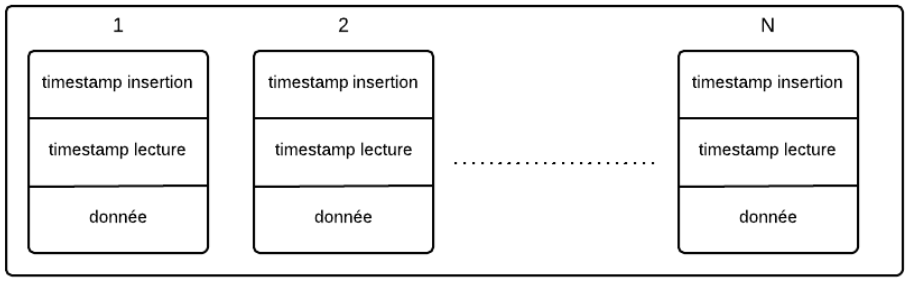

Figure 5 Structure de la **Smart_FIFO**

3.3 Smart_FIFO

3.3.1 Concept

Pour accélérer les simulations, il serait intéressant d'inclure le principe de **temporal decoupling** à la During_FIFO. Le principe du **temporal decoupling** est de stocker le temps local à chaque module dans une variable locale, la synchronisant avec le temps SystemC qu'à des moments précis de l'exécution du processus ; ainsi la simulation ne serait pas ralentie par des changements de contextes à chaque **wait**.
Chaque appel à **wait** serait remplacé par une fonction incrémentant une variable locale à chaque module. La synchronisation effectuera un **wait** de la différence entre le temps du module et le temps de SystemC.
Le principe de cette FIFO couplée au **temporal decoupling** est donc d'ajouter une notion de temps à chaque interaction avec cette dernière, comme on peut le voir dans la figure 5. Dès lors, à chaque lecture/écriture, un **timestamp** sera ajouté à la donnée sur la FIFO pour renseigner à la prochaine lecture ou écriture si cette case est disponible ou non à un instant t.

3.3.2 Algorithme d'écriture

- on vérifie que la FIFO n'est pas remplie de cellules **occupées**, si c'est le cas on attend qu'une cellule soit libérée.
- si la première cellule disponible à une date de lecture dans le futur, on attend jusqu'à cette date avant de réaliser l'insertion.
- on met à jour la donnée et le timestamp d'écriture, on incrémente la variable représentant l'indice de la première cellule disponible.
- on notifie d'une écriture pour réveillé tout lecteur éventuel.

Une cellule est considérée comme occupée à l'instant t si la date d'insertion est dans le passée (inférieur à t) ou la date de lecture est dans le futur (supérieur à t).

3.3.3 Mise en pratique avec sc-during

Le problème avec la bibliothèque **sc-during** est qu'on ne sait pas vraiment à quel instant du temps simulé la tâche avec durée va être exécutée. L'appel à **during(f,5)** au temps t signifie que l'on accorde 5 unités de temps pour l'exécution de f, mais l'exécution de f peut avoir lieu n'importe quand entre [t, t+5].

4 Conclusion

Références

1. MOY, Matthieu. Parallel programming with systemc for loosely timed models : a non-intrusive approach. In : Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2013. p. 9-14.
2. HELMSTETTER, Claude, CORNET, Jérome, GALILÉE, Bruno, et al. Fast and accurate TLM simulations using temporal decoupling for FIFO-based communications. In : Design, Automation et Test in Europe Conference et Exhibition (DATE), 2013. IEEE, 2013. p. 1185-1188.
3. WADGHIRI, Mohamed Zaim, MOY, Matthieu, et MAIZA, Claire. Simulation coopérative et parallèle : Expérimentations avec le scheduler SystemC. 2011.
4. KAUSHIK, Anirudh Mohan. Accelerating Mixed-Abstraction SystemC Models on Multi-Core CPUs and GPUs. 2014.