OPTIMISATIONS DE PERFORMANCES DE SIMULATEURS SUR MACHINES MULTI-CŒURS

Xavier Poczekajlo

16 mai 2013

Introduction

Ce travail a été mené au sein du laboratoire de recherche Verimag, au cours d'une Introduction à la Recherche en Laboratoire. J'ai été encadré pour ce travail par Matthieu Moy, membre de l'équipe de recherche Synchrone de Verimag.

Mon travail a porté sur la simulation de système sur puce. La simulation de système est devenue une partie indispensable du processus de création d'un système sur puce. SystemC [1] est un simulateur libre qui s'est imposé dans l'industrie pour ces simulations. L'outil sc-during [2][3] est développé par Verimag, dans le but d'améliorer les performances du simulateur SystemC en tirant partie du parallélisme des systèmes sur puce simulés. sc-during connaît des problèmes de synchronisation avec l'utilisation d'un très grand nombre de processeurs. Pour résoudre cela, on utilise le profiling. Le profiling est une méthode qui permet d'obtenir des informations sur le déroulement du programme.

L'objectif de mon travail d'IRL est de réaliser un outil de profiling adapté à sc-during. Son rôle est d'analyser la contention [1] sur les verrous. Cela permet à terme de trouver les points bloquants dans le programme, pour améliorer les performances.

Table des matières

1. Le temps de contention est le temps d'attente pour accéder à une ressource partagée.

1 Les limites de SystemC pour la simulation de système sur puce moderne

1.1 Systèmes parallèles : la norme moderne

De nos jours, les systèmes sur puce sont soumis aux mêmes contraintes que le reste du marché : il faut produire bien, et produire vite. La particularité du système sur puce est que pour développer la partie logicielle, il faut pouvoir travailler sur la partie matérielle. Ainsi, sans le matériel, il n'est pas possible de produire le logiciel, et donc il n'est pas possible de produire aussi vite que souhaité. Pour résoudre ce problème, les développeurs ont recours à la simulation de matériel. Simuler le système sur puce permet de lancer les deux parties -logicielle et matérielle- en production en même temps, et donc de gagner du temps.

À l'instar des ordinateurs modernes, le principal facteur de gain de performances des systèmes modernes est l'augmentation du nombre de processeurs. Développeurs comme simulateurs doivent s'adapter à cette contrainte. Simuler le parallélisme du système sur puce permet de gagner un temps très important.

L'outil majoritairement utilisé pour simuler des systèmes sur puce dans l'industrie est SystemC. Si SystemC s'est largement imposé, c'est qu'il a permis d'offrir un excellent outil de simulation pour les systèmes sur puces. SystemC est une bibliothèque pour C++, qui permet la modélisation de système embarqué et donc de système sur puce.

sc-during a permet de s'adapter à cette évolution, pour exploiter au mieux le parallélisme.

1.2 SystemC : un outil aux performances limitées

Comme cela a été précisé ci-dessus, SystemC n'est pas un langage de programmation mais une bibliothèque pour C++. Cela offre un certain nombre d'avantages. Le premier est que SystemC se base sur un langage populaire. De plus, le C++ dispose d'outils puissants pour le développement et la compilation. Ensuite, disposer d'un langage pour tout modéliser est un atout majeur. En effet, cela permet d'éviter de lourdes conversions entre les différents outils de modélisation.

Le modèle de programmation de SystemC est le suivant. Il est composé de différents modules qui communiquent via des canaux. Chaque module a un comportement, implémenté par un ou plusieurs processus. Comme une porte logique, les modules ont différents ports, d'entrée ou de sortie, qui ont pour fonction de relier le module avec l'extérieur. Les ports d'entrée permettent aux modules de réagir aux évènements externes au module. Ceci est illustré dans la figure 1.

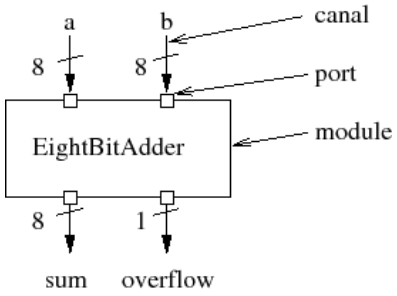

FIGURE 1 – Exemple de description d'un additionneur 8-bits, avec le vocabulaire SystemC.

Il est évidemment possible de gérer le parallélisme avec SystemC. Cependant, la simulation de code parallèle sous SystemC se fera séquentiellement. Cela permet plus de fiabilité de simulation. Si cette fiabilité est intéressante, elle oblige à retarder l'exécution des simulations, puisque le parallélisme est simulé, et non réel. Ainsi, du fait de l'augmentation du niveau actuel de parallélisme dans les systèmes sur puce -et encore plus dans les systèmes futurs, ce point de SystemC peut faire l'objet d'une sensible amélioration pour les performances.

SystemC est donc l'outil utilisé majoritairement pour les simulations. Par contre, sa gestion du parallélisme peut être améliorée pour la simulation des systèmes sur puce futur.

2 Un outil pour la programmation parallèle en SystemC : sc-during

Nous allons maintenant nous concentrer sur la parallélisation dans SystemC, en présentant une nouvelle bibliothèque : sc-during.

2.1 Fonctionnement général

Le but de sc-during est de permettre à l'utilisateur de gagner en performance sans modifier grandement ses habitudes. sc-during propose des primitives pour gérer le temps et le parallélisme. Ces primitives peuvent être utilisées dans certaines parties uniquement. Le concept est de les utiliser dans les parties particulièrement gourmandes en performances, là où le parallélisme est maximal. Ainsi, l'utilisateur n'a qu'une partie du code à retravailler : celle où le gain de temps potentiel est important.

sc-during lie temps et tâches. Il est possible d'associer un temps d'exécution aux tâches à simuler. Ceci permet de marquer des pauses dans une simulation entre des sections critiques. Cette notion vient de jTLM [4], un simulateur préemptif, écrit en Java.

La fonction `during()` permet ce lien. Cette fonction prend deux arguments : une durée et une tâche. L'appel à cette fonction va permettre l'utilisation d'un nouveau thread qui exécutera la tâche passée en paramètre. À la fin de cette tâche, ou à la fin du temps qui lui a été donné, SystemC va se synchroniser avec le thread, pour permettre une exécution cohérente.

Ceci est le cœur du fonctionnement de sc-during, et c'est ce qui le différencie des autres simulateurs de parallélisme en SystemC.

2.2 Résultats expérimentaux et perte d'efficacité mesurée

La bibliothèque sc-during est fonctionnelle. Pour des tâches hautement parallélisable, elle permet des performances accrues. Malgré cela, l'accélération mesurée devient négligeable à partir d'un certain nombre de CPUs. Ceci est dû au besoin de synchronisation grandissant.

2.3 Détection des problèmes de synchronisation

Une méthode pour détecter d'où viennent les problèmes de synchronisation est le profiling. Ma contribution à sc-during consiste en la réalisation de profiling. Mon travail s'inscrit dans le cadre de sc-during. Cependant, le travail réalisé est indépendant, et fonctionne pour tout programme utilisant la bibliothèque `boost`.

3 Outils externes et instrumentation du code

Le but est d'avoir des statistiques sur les mutex, pour pouvoir analyser entre autres le temps de contention. Un exemple de temps de contention est illustré à la figure 2. Ceci devrait permettre d'identifier la source de baisse de performance. J'ai employé différentes méthodes pour parvenir à cela.

Nous utilisons ici la bibliothèque pour C++ `boost` [5], qui fournit -entre autres- des classes pour la programmation parallèle, comme `boost::mutex` ou `boost::unique_lock`. Nous reviendrons sur ces classes par la suite.

3.1 Profiling : l'utilisation d'outils d'analyse externe

Une première tentative était d'utiliser un outil de profiling : mutrace. Après une installation relativement ardue, l'utilisation devait être plus simple.

Malheureusement, ces essais n'ont pas été concluants. En effet, il n'a pas été possible de faire des mesures sur les mutex à l'intérieur des modules sc-during. Cela s'est révélé techniquement infaisable.

Mutrace est un outil conçu pour des programmes simples, avec thread POSIX. L'utilisation pour sc-during est rendue difficile. Pour illustrer ceci, nous allons faire un exemple simplifié de ce que l'on voudrait faire. L'exemple utilise un `boost::mutex`, comme le fait la bibliothèque sc-during.

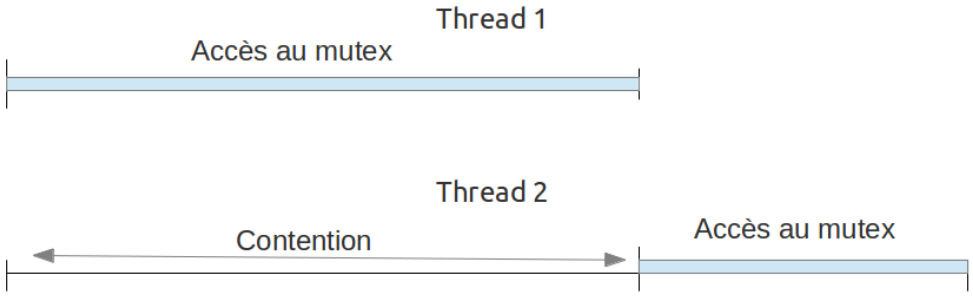

FIGURE 2 – Représentation du temps de contention lors de l'accès à un verrou par deux threads.

```
#include <boost/thread.hpp>
using namespace std;

boost::mutex m;

void foo()
{
        m.lock();
        m.unlock();
}

int main(void)
{
        boost::thread t(foo);
        t.join();
        return 0;
}
```

Voici maintenant une partie de la sortie de mutrace, lancé sur le programme précédent.

```
mutrace : 6 mutexes used.

Mutex #0 (0x0x950f6f8) first referenced by :
/usr/lib/mutrace/libmutrace.so(pthread_mutex_init+0xf7) [0x71f457]
./run() [0x804bbb2]
./run() [0x804be56]
./run() [0x804d107]
./run() [0x804d04a]
./run() [0x804cf26]
./run() [0x804cc87]
./run() [0x804c741]
./run() [0x804b40a]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0x1e94d3]
```

On note premièrement que mutrace annonce 6 mutex utilisés, alors qu'il n'y a qu'un seul `boost::mutex` déclaré dans le code. Ensuite, quelques informations sur des mutex sont affichées. On note ici qu'il n'y aucune information utilisable en l'état sur notre mutex.

Le problème vient probablement de la complexité de boost : le `boost::mutex` utilisé est compris dans une classe spécialisée de `boost`, et est donc caché à l'outil mutrace..

Cet exemple sur un programme simplifié à l'extrême montre que mutrace n'est pas adapté, en l'état, à notre

cas d'utilisation.

3.2 Instrumentation légère : héritage de la classe `boost::mutex`

Puisqu'il n'était pas possible de réaliser une analyse avec des logiciels externes, nous avons décidé d'essayer une analyse interne à la bibliothèque sc-during. J'ai donc instrumenté le code.

sc-during utilise la classe `boost::mutex`, et la solution est de surcharger cette classe. L'objectif était d'avoir des statistiques précises sur les `boost::mutex`. Cette solution permet de garder le reste du code propre en ne modifiant que le type des `boost::mutex`.

Voici le fonctionnement en détail. Hériter le type `boost::mutex`, par un type mutex_t de mutex personnalisé. Ainsi, une simple modification des types des mutex dans sc-during aurait permis de modifier tous les mutex utilisés. L'héritage permet de ne pas modifier l'API, et donc les modifications qui en découlent sont mineures. On vient ensuite surcharger les fonctions sur lesquelles ont peut récupérer des statistiques.

En voici un exemple simplifié :

```
class instrumented_mutex : public boost::mutex {
    public:
    // surcharge de la fonction de boost::mutex
    void lock() {
    // collect some stat
        ...
        super::lock();
    }
}
```

Les statistiques sont récupérées -entre autre- par l'utilisation de la fonction `boost::try_lock()`. Cette fonction essaie de verrouiller le mutex passé en paramètre. Si le mutex est déjà pris, alors elle rend la main. Cela permet de calculer le temps d'attente, en essayant d'accéder au mutex en continu.

Cette solution a de nombreux avantages, mais elle n'est pas réalisable. La raison est que la surcharge des fonctions de la classe héritée `instrumented_mutex` ne sera pas effective. La fonction de `instrumented_mutex lock()` ne sera jamais appelée à la place de la fonction de `boost::mutex lock()` car elle n'est pas définie avec le mot clef `virtual`.

Pour essayer de pallier à cela, j'ai transformé la classe mutex_t pour qu'elle n'hérite plus de `boost::mutex`, mais contienne un `boost::mutex`. Voici un exemple de l'architecture présentée.

```
class instrumented_mutex {
public:
    // surcharge de la fonction de boost::mutex
    void lock() {
        // collect some stat
        ...
        m_mutex.lock();
    }
private:
    boost::mutex m_mutex;
}
```

Le problème est que les mutex ne sont pas utilisés uniquement seuls mais aussi passés en paramètre de `boost::unique_lock`. Ainsi, il faudrait modifier les `boost::unique_lock` mais aussi tout ce qui utilise les `boost::mutex`. Ce n'est pas possible, car cela demande au final de modifier la bibliothèque boost, ce qui n'est pas une solution acceptable.

3.3 Instrumentation lourde : surcharge de la classe `boost::unique_lock`

Commençons par faire un point sur l'API de la bibliothèque `boost`. La bibliothèque `boost` propose les `boost::mutex` et les `boost::unique_lock`. La classe de `boost::mutex` permet de créer des mutex classiques. La classe `boost::unique_lock` permet de définir un verrou qui prend un `boost::mutex`, le verrouille à la

déclaration de l'objet et le déverrouille à la destruction de l'objet. Ça permet d'être sûr de déverrouiller le `boost::mutex`, malgré un évènement inattendu, comme une exception.

```
{
    boost::unique_lock u_lock(m_mutex);
    /* m_mutex est verrouille */
    /* section critique */
        ...
    /* fin de la section critique */
}
/* m_mutex est deverrouille */
```

Comme on l'a vu dans la partie précédente, il n'est pas possible de faire une classe fille de `boost::mutex`. À la place, il est possible de faire une classe contenant un `boost::unique_lock` : la classe `instrumented_unique_lock`.

Quelle est la différence avec la solution précédente ? La différence est que le `boost::unique_lock` est un objet qui verrouille à sa création. Il est possible de réaliser des mesures en créant l'objet, contrairement à la classe fille de `boost::mutex`. Nous n'aurons pas de problème de type, puisque notre classe `instrumented_unique_lock` n'est pas héritée. Il n'y a pas les problèmes de surcharge de fonction dans une classe fille, comme lors de la solution précédente.

Nous allons maintenant présenter l'implémentation de cette solution. Il faut réaliser un héritage sur les `boost::unique_lock`, et une classe qui gère les statistiques. L'accès aux statistiques doit être protégé par un mutex pour éviter les écritures simultanées. Le destructeur de la classe statistique affiche les statistiques qu'il a calculées. Les statistiques sont donc automatiquement données à la fin du programme.

```
/* Declaration d'un objet global de statistiques*/
stat_t unique_lock_stat;

class instrumented_unique_lock {
public:
    boost::unique_lock<boost::mutex> &get_unique_lock() {return m_unique_lock;}

instrumented_unique_lock(boost::mutex& m) : m_timer(), m_unique_lock(m) {
    /* Le timer a ete cree avant que le lock soit fait, donc on obtient */
    /* le temps de contention, que l'on ajoute aux statistiques. */
    unique_lock_stat.add(m_timer.get_time());
}

private:
timer m_timer;
boost::unique_lock<boost::mutex> m_unique_lock;
};
```

Cet outil permet, d'avoir une classe personnalisée qui maintient des statistiques, et les affiche à la fin de l'exécution de son programme. Pour un outil de mesure, il est important de se demander si l'expérience ne modifie pas la mesure.

J'ai réalisé un test, qui crée dix threads utilisant le même `boost::mutex`, avec une section critique qui crée de la contention en réalisant des calculs. Le protocole est de faire une moyenne du temps d'exécution du programme avec ma classe `instrumented_unique_lock` et de comparer à la moyenne du temps d'exécution avec la classe `boost::mutex`.

Voici les mesures réalisées.

opérations sur dix mesures	`instrumentation_unique_lock`	`boost::unique_lock`
moyenne	30,847s	28,506s
maximum	38,497s	34,140s
minimum	27.018s	26,025s

On constate une différence sur la moyenne de 2,3 secondes, soit 8% du temps d'exécution. Les maximums sont très éloignés, puisqu'il y a 4 secondes de différences entre les deux maximums.

Il y a peu de données, mais l'outil a l'air utilisable en l'état. La mesure n'est pas très fine, mais ce n'est pas indispensable pour l'utilisation. Je n'ai malheureusement pas eu le temps de l'exploiter pour analyser sc-during. Ceci est maintenant possible.

Quelles sont les améliorations possibles ? Il est possible tout d'abord d'améliorer la mesure, en diminuant la contention ajoutée par la mesure. Ceci nécessite des méthodes d'écriture concurrentes avancées.

De plus, il peut être intéressant de chercher à donner plus d'informations. Par exemple, des statistiques données par `boost::mutex`, et non plus pour tout le programme. Le problème est que cela risque de fausser encore plus la mesure.

Conclusion

sc-during est une bibliothèque qui permet de profiter du parallélisme des systèmes simulés. La simulation de ce parallélisme n'est pas tout à fait optimale : beaucoup de temps est perdu dans la synchronisation à partir d'un grand nombre de processeurs.

Mon travail permet d'obtenir des statistiques sur les différents endroits clefs de la synchronisation : les verrous. Il permettra d'analyser le déroulement du programme et de trouver les points de contentions importants. Ceci permettra à terme de profiter pleinement du parallélisme.

Maintenant que l'on dispose d'outils, il est possible de réaliser une analyse de la bibliothèque sc-during. Il peut être intéressant d'améliorer l'outil pour avoir de meilleures mesures sur les mutex.

Références

[1] O. S. Initiative, "Systemc 2.0. 1 language reference manual," *Revision*, vol. 1, no. 1177, pp. 95118–3799, 2003.

[2] M. Zaim-Wadghiri, "Simulation coopérative et parallèle : Expérimentations avec le scheduler SystemC," (Grenoble, France), May 2011.

[3] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models : A Non-Intrusive Approach," in *The Design, Automation, and Test in Europe (DATE)*, (Grenoble, France), Mar 2013.

[4] G. Funchal and M. Moy, "jtlm : an experimentation framework for the simulation of transaction-level models of systems-on-chip," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–4, IEEE, 2011.

[5] B. Dawes, D. Abrahams, and R. Rivera, "Boost c++ libraries," *URL http ://www. boost. org*, vol. 35, p. 36, 2009.