Techniques de compilation dédiées pour un langage spécifique à un domaine (SystemC)

Guillaume Sergent
Advisor: Matthieu Moy (VERIMAG, SYNCHRONE)

Juin 2014

This work has been partially supported by the
LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01)

1 Introduction

1.1 Systèmes sur puce et co-conception matériel-logiciel

L'évolution des techniques industrielles, en permettant une augmentation exponentielle de la densité des transistors selon la loi de Moore (doublement tous les deux ans), n'a pas seulement permis l'amélioration des performances de ces dernières mais également l'étendue de leurs fonctionnalités. Ceci s'est manifesté en premier lieu dans l'apparition du microprocesseur, et désormais de plus en plus de SoC (systèmes sur puce, qui contiennent non seulement un ou des processeurs mais aussi l'essentiel des périphériques dont l'utilisateur a besoin) sont présents sur le marché. Des contraintes pécuniaires (produire une puce personnalisée entraîne des frais fixes considérables, pouvant dépasser le million de dollars des États-Unis[1] et qui sont encourus à nouveau dès le moindre changement des plans de la puce) et mercatiques (les SoC, une fois mis sur le marché deviennent rapidement obsolètes) font qu'il est d'une importance capitale de concevoir un nouveau SoC le plus rapidement possible en produisant le moins de prototypes physiques possible. Dans ce contexte, il est hautement désirable de pouvoir tester des logiciels sur des versions préliminaires du futur SoC sans en avoir d'exemplaire matériel, mais également de vérifier formellement le

matériel ou d'estimer sa température lors de son utilisation. Il est envisageable, dans certains cas, de prototyper le SoC ou une partie de celui-ci non pas sur un ASIC [1] trop onéreux mais sur un FPGA [2], ce qui élimine les frais fixes au prix d'une diminution importante des performances. On ne s'intéressera désormais plus qu'à la simulation logicielle des SoC, laissant de côté l'utilisation de FPGAs.

1.2 Modélisation du matériel en vue d'une simulation

Pour simuler un SoC de façon logicielle, il faut avoir un moyen de décrire le matériel le composant. Parmi les langages de description du matériel se trouvent les langages suivants :

1.2.1 VHDL et Verilog

VHDL et Verilog sont les langages traditionnels, qui ont l'intérêt d'avoir un sous-ensemble synthétisable (vers des FPGAs ou des ASICs) tout en permettant une programmation plus usuelle, pour faire des tests par exemple. Des extensions de ces langages, telles que SystemVerilog, permettent de programmer à un plus haut niveau.

1.2.2 SystemC

SystemC suit l'approche inverse et part du langage généraliste C++ et l'étend avec des fonctionnalités spécifiques adaptées à la simulation de composants matériels, dont notamment un parallélisme intrinsèque et une notion de temps. Une implémentation de référence est disponible sous forme d'une bibliothèque C++ qui simule le parallélisme sus-cité en exécutant à la suite les évènements qui ont lieu simultanément selon le temps simulé. L'exécution d'un programme SystemC se déroule en deux étapes :

- Une phase d'élaboration, où des composants sont instanciés et connectés ensemble.
- Une phase de simulation, où la bibliothèque simule l'exécution simultanée (selon le temps simulé, mais pas le temps réel) des fonctionnalités des différents composants.

1.2.3 Niveau de la modélisation

On peut modéliser des composants matériels à plusieurs niveaux de détail, dont on peut citer les suivants, par ordre de performances croissantes :

1. Application-Specific Integrated Circuit : c'est un circuit intégré fait sur mesure.
2. Field-Programmable Gate Array : ce sont des ASICs déjà conçus, qui peuvent être programmés pour avoir les fonctionnalités désirées. Ils peuvent être achetés à l'unité.

– Un niveau « portes logiques », où le composant est entièrement décrit à l'aide de portes logiques, ou d'opérateurs classiques et simples (addition, opérateur ternaire `?:`, multiplexeur, etc.). Ainsi le modèle est aisément synthétisable.
– Un niveau intermédiaire, où une partie des calculs est faite de manière logicielle plutôt que par simulation d'un matériel les effectuant : on peut ainsi par exemple remplacer la partie centrale d'un processeur par un simulateur de son jeu d'instructions.
– Un niveau transactionnel : au lieu de modéliser les connexions entre les composants par les fils qui les connecteraient vraiment dans une réalisation physique, on permet aux composants de réaliser directement des transactions entre eux : par exemple, dans le bus mémoire Basic (cf. section 2.3), on a des fonctions de lecture et d'écriture et il ne reste à modéliser dans le bus que le décodage d'adresses. TLM, une extension de SystemC en faisant désormais partie et permettant un tel niveau de modélisation, est traité dans la section 2.2.

1.3 Limitations de SystemC

SystemC a comme avantages importants d'être compilable avec uniquement un compilateur C++ et une bibliothèque et de permettre la réutilisabilité des composants ; mais ces avantages ont un prix.

1.3.1 Difficultés concernant l'optimisation

L'aspect orienté object de C++ dont hérite SystemC fait que les fonctionnalités d'un composant sont implémentées par des méthodes indépendantes de l'instance du composant sur laquelle on les exécute. Cela empêche un compilateur C++ ordinaire d'optimiser ces fonctionnalités selon l'environnement dans lequel les composants sont exécutés, même lorsque le composant n'est instancié qu'une fois. De plus, certaines données ne sont modifiées que pendant la phase d'élaboration, mais le compilateur l'ignore, ce qui l'empêche d'utiliser ces données à des fins d'optimisation.

1.3.2 Difficultés concernant la vérification formelle

Lorsque dans un programme SystemC plusieurs évènements ont lieu, la sémantique de SystemC permet d'appliquer leurs effets dans n'importe quel ordre. Cependant, l'implémentation de référence choisit arbitrairement une façon d'ordonner ces évènements, ce qui fait que la vérification formelle de la combinaison d'un programme SystemC et de l'implémentation de référence de SystemC par un vérificateur de C++ ne vérifie pas le programme donné selon la sémantique de SystemC. La vérification formelle d'un programme SystemC nécessite donc l'emploi d'outils spécifiques.

1.4 PinaVM : une solution aux problèmes précédents

PinaVM [4] est un front-end SystemC utilisant LLVM [3] qui exécute la phase d'élaboration afin de connaître la disposition des composants d'un programme SystemC, le but étant à l'origine d'en faire un modèle permettant la vérification formelle. Cette approche est particulièrement adaptée pour les SoC dans lesquels la disposition des composants est par nature invariable. L'utilisation de LLVM permet à PinaVM de ne travailler que sur la représentation intermédiaire de LLVM, ce qui permet d'éviter d'analyser directement du C++ ou de devoir générer du code exécutable, ces tâches étant confiées respectivement à Clang et à LLVM. En contrepartie, cela demande à PinaVM de dépendre d'un certain nombre de détails d'implémentation du C++. L'infrastructure d'exécution partielle de PinaVM a été réutilisée [2] pour l'optimisation à la volée de programmes SystemC : la connaissance de la disposition des composants permet de faire des optimisations complémentaires s'ajoutant à celles de Clang et LLVM.

1.5 Contributions

1.5.1 Optimisation statique en deux passes

Le dispositif d'optimisation de PinaVM ne permettait précédemment d'optimiser un programme SystemC qu'en vue de son exécution immédiate, ce qui nécessitait donc d'exécuter N fois l'optimiseur pour N exécutions. Désormais il est possible pour l'optimiseur de produire un exécutable ordinaire, ouvrant la voie à de potentielles optimisations coûteuses qui prennent plus de temps qu'elles n'en économisent à l'exécution. Les deux passes sont donc désormais :

- Exécution de la phase d'élaboration du code d'origine pour obtenir la disposition des composants et de certaines structures de données SystemC.
- Ces informations sont utilisées pour produire une version optimisée du programme.

Le nouveau déroulement d'une exécution de PinaVM est décrit dans la figure 1.

1.5.2 Extension de la portée de l'optim par utilisation d'analyse statique

L'utilisation d'une analyse de valeur fournie par LLVM permet d'élargir de façon notable la portée d'une optimisation déjà présente dans PinaVM.

1.5.3 Optimisation statique des communications

Dans le cadre d'un système invariable, une fois la phase d'élaboration terminée, certaines données ont une valeur constante et prédictible. Cependant,

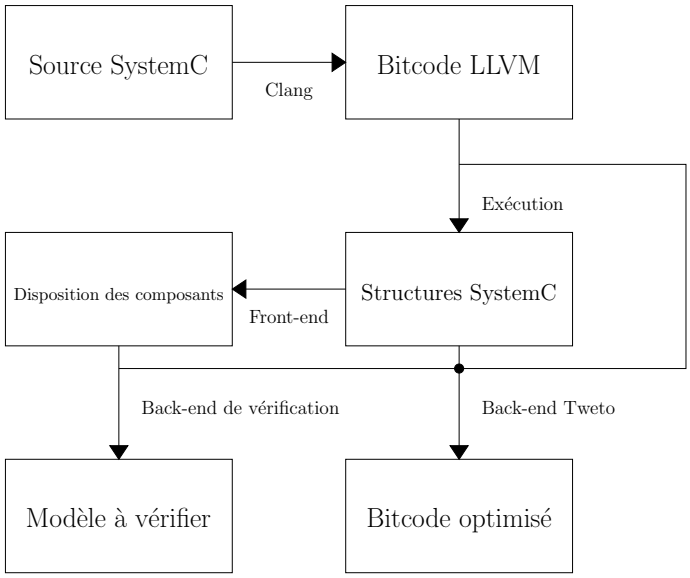

FIGURE 1 – Schéma montrant le nouveau processus d'optimisation de PinaVM.

comme elles ont été modifiées lors de la phase d'élaboration, un compilateur C++ ordinaire ne peut pas en tirer profit. L'outil Tweto, développé par Claude Helmstetter, permet d'en tirer profit avec deux étapes :

- Une fonction `tweto_mark_const`, qui signale à l'optimiseur qu'une donnée ne changera plus après l'élaboration ;
- Des optimisations en tirant profit.

Cet outil est en cours de réintégration à PinaVM et d'amélioration.

2 SystemC

2.1 Concepts de SystemC

SystemC ajoute des concepts à C++, dont voici quelques-uns d'une importance centrale :

2.1.1 Les composants

Un composant SystemC est codé comme une classe héritant de `sc_core::sc_module`. Ses entrées et sorties sont des membres publics de la classe, et ses fonctionnalités sont implémentées par des fonctions membres, généralement déclarées dans le constructeur de la classe :

- Soit comme des `SC_METHOD`s, qui s'exécutent en une fois et qui sont lancées à chaque fois que la fonctionnalité est utilisée ;

```
#include <systemc>
using namespace sc_core; using namespace std;
struct Vigenere:public sc_module {
        sc_in<sc_uint<8> > in;
        sc_out<sc_uint<8> > out;
        sc_uint<8> key;
        void maj_cle();
        void calcul();
        SC_CTOR(Vigenere) {
                SC_THREAD(maj_cle);
                sensitive << in;
                SC_METHOD(calcul);
                sensitive << in;
        }
}
void Vigenere::calcul() {
        out.write (in.read() + key);
}
```

FIGURE 2 – Un composant SystemC implémentant un chiffre de Vigenère sur des entiers. Le `SC_THREAD maj_cle` met à jour la clé de chiffrement à chaque nouvelle lettre alors que la `SC_METHOD calcul` chiffre l'entrée avec la clé pour obtenir la sortie.

- Soit comme des `SC_THREAD`s, qui peuvent se suspendre et permanents, et qu'on lance qu'une fois au début de la simulation.

La figure 2 donne un exemple de composant SystemC.

2.1.2 La plate-forme d'exécution

L'exécution d'un programme SystemC se déroule en deux étapes :

- La fonction `sc_main()` est d'abord appelée. Elle instancie des composants SystemC, qui sont des classe héritées d'un prototype commun, puis les connecte ensemble.
- `sc_main()` appelle `sc_start()` pour rendre la main à l'ordonnanceur de SystemC afin de commencer la simulation. Les seules fonctions alors exécutées par l'ordonnanceur sont les fonctionnalités des composants enregistrées comme `SC_METHOD`s ou comme `SC_THREAD`s comme décrit dans la figure. L'ordonnanceur opère un multi-tâche coopératif : un `SC_THREAD` garde le contrôle tant qu'il n'a pas explicitement rendu la main, ce qu'il peut faire par exemple en appelant `sc_wait`.

Les fonctionnalités des composants sont donc des fonctions génériques ne dépendant que de la nature du composant et non du contexte où il est placé, ce qui les rend difficiles à optimiser telles quelles. Ceci est illustré dans la figure 4.

```
struct Message:public sc_module {
        sc_out<sc_uint<8> > msg;
        void emettre();
        SC_CTOR(Message) {
                SC_THREAD(emettre);
        }
}
void Message::emettre() {
        char const* s = "Exemple";
        for (int i = 0; i < strlen(s); i++) {
                msg.write(s[i]);
                sc_wait(10,SC_NS);
        }
}
int sc_main (int argc, char* argv[]) {
        Message m;
        Vigenere v;
        sc_signal<sc_uint<8> > s;
        m.msg(s);
        v.in(s);
        return 0;
}
```

FIGURE 3 – Le composant Vigenere est relié à un composant Message émettant un message à coder. Un composant utilisant la sortie de Vigenere a été omis pour la lisibilité.

```
int sc_main (int argc, char* argv[]) {
        Message m1,m2;
        Vigenere v1,v2;
        sc_signal<sc_uint<8> > s1,s2;
        m1.msg(s1);
        v1.in(s1);
        m2.msg(s2);
        v2.in(s2);
        return 0;
}
```

FIGURE 4 – Reprise de la figure 3 avec deux instances de Message et Vigenere. La méthode `calcul` de Vigenere ne peut pas dans cette situation déterminer à quel composant `in` est connecté.

2.2 TLM

Les extensions étaient à l'origine destinées à des descriptions matérielles bas niveau, comme par exemple RTL (Register-Transfer Level). Désormais, pour faire face à la complexité croissante des systèmes qu'on veut simuler, tant du côté matériel (systèmes sur puce entiers) que du côté logiciel (systèmes d'exploitation généralistes), on cherche à simplifier tant que possible le modèle du système matériel simulé. SystemC a été à ces fins étendu par le standard TLM (Transaction-Level Modeling) permettant, conjointement avec le remplacement des composants par des modèles logiciels de leur comportement, de simuler le plus simplement possible les couches logiques des protocoles de communication entre composants, en faisant abstraction des couches physiques.

2.3 Le protocole Basic

C'est un protocole de bus simplifié utilisé à des fins pédagogiques dans les cours et TP de Matthieu Moy sur SystemC à l'Ensimag[lien]. La figure 5 donne un exemple de programme utilisant ce protocole.

2.4 PinaVM

PinaVM est, comme dit dans la section 1.4, un outil permettant de vérifier formellement et d'optimiser des programmes SystemC reposant sur un front-end qui exécute la phase d'élaboration du programme qu'il analyse.

2.4.1 Composition détaillée

PinaVM [4] utilise deux bibliothèques externes :
- Les bibliothèques LLVM, pour manipuler la représentation intermédiaire de LLVM [3].
- L'implémentation de référence de SystemC, qui a été modifiée pour les besoins de PinaVM.

Elle contient :
- Un front-end, qui reconnaît dans la représentation intermédiaire LLVM du code des appels à SystemC ;
- Des back-ends, de deux types :
  - Des back-ends d'analyse statique, pour la plupart destinés à la vérification formelle et utilisant les données du front-end pour émettre un modèle du programme SystemC prêt à être vérifié ;
  - Un back-end d'optimisation, basé [2] sur l'outil Tweto (TLM with elaboration-time optimization) de Claude Helmstetter, utilisant directement les structures de données de la bibliothèque SystemC pour

3. Cette représentation est bas niveau, typée et sous forme SSA.

```
#include "basic.h"
#include "bus.h"
#include <sysc/pinavm/permalloc.h>
using namespace std; using namespace sc_core;

struct initiator : sc_module {
        basic::initiator_socket<initiator> socket;
        void thread(void) {
            for (basic::addr_t addr = 4; addr <= 30; addr+=4)
                socket.write(addr, 42);
        }
        SC_CTOR(initiator) {
                SC_THREAD(thread);
        }
};

struct target : sc_module, basic::target_module_base {
        basic::target_socket<target> socket;
        tlm::tlm_response_status write(basic::addr_t a,
                                       const basic::data_t &d)
                                          {
                cout << "j'ai recu : " << d
                     << " a l'adresse " << a << endl;
                return tlm::TLM_OK_RESPONSE;
        }
        tlm::tlm_response_status read(basic::addr_t a,
                                       /* */ basic::data_t &d)
                                          {
                SC_REPORT_ERROR("TLM", "non implemente");
                abort();
        }
        SC_CTOR(target) {/* */}
};

int sc_main (int argc, char ** argv) {
        initiator* a = permalloc::obj<initiator>("Alice");
        target* b = permalloc::obj<target>("Bob");
        Bus* router = permalloc::obj<Bus>("Router");

        router->map(b->socket, 4, 100);
        a->socket.bind(router->target);
        router->initiator.bind(b->socket);

        sc_start();
        return 0;
}
```

FIGURE 5 – Un exemple d'utilisation du protocole Basic. Alice (`a`) peut accéder à Bob (`b`) aux adresses de 4 (incluse) à 100 (exclue). Ces adresses sont traduites par le bus en adresses de 0 (incluse) à 96 (exclue).

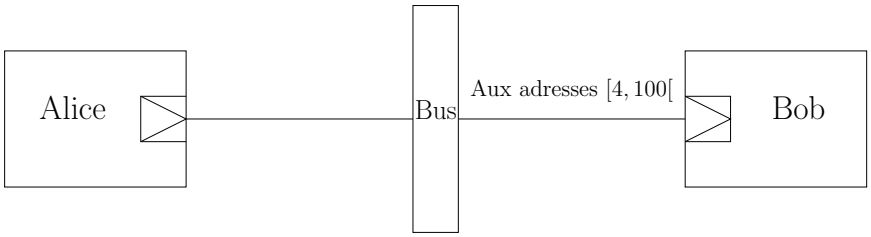

FIGURE 6 – Schéma de la disposition des composants dans la figure 5. Alice effectue des transactions mémoire sur le bus, dont le décodage d'adresses est le suivant : une transaction est renvoyée vers Bob si l'adresse mémoire est entre 4 inclus et 100 exclu, et génère une erreur sinon.

ré-optimiser le programme SystemC avant de lancer la phase de simulation.
- Un code reliant les parties précédentes ensemble et gérant l'interface utilisateur ; c'est lui qui compile le programme et exécute sa phase d'élaboration, après quoi la bibliothèque SystemC modifiée rend la main à PinaVM, qui peut alors lancer le front-end et un back-end.

Les étapes d'une exécution de PinaVM sont décrite dans la figure 1.

2.4.2 Détails du processus d'optimisation

Une fois que la phase d'élaboration a été exécutée par PinaVM, la phase d'optimisation peut avoir lieu. Les étapes suivantes sont alors successivement entreprises à cet effet :

- Créer des spécialisations de certaines fonctions membres, notamment les `SC_METHOD`s et les `SC_THREAD`s, des composants aux instanciations de ces derniers ;
- Tirer profit de la donnée de l'instance afin de mieux optimiser ces dernières ;
- Modifier les structures de données de SystemC afin d'utiliser les fonctions ainsi créées des `SC_METHOD`s et des `SC_THREAD`s au lieu de leurs versions génériques.

Par exemple, dans le code de la figure 4, deux instances de `Vigenere::calcul()` seront créées, et on saura alors que dans `v1 in` est relié à `m1` et que dans `v2 in` est relié à `m2`.

3 Optimisation permanente en deux passes

3.1 La situation initiale

L'étape d'optimisation de PinaVM décrite à la section 2.4.2 repose massivement sur l'évaluation d'expressions retournant des pointeurs, que l'on

```
int global = 42;
int main()
{
        int* pi = permalloc<int>(38);
        int** ppi = permalloc<int*>(&global);
        std::string* ps = permalloc<std::string>("Test");
        return 0;
}
```

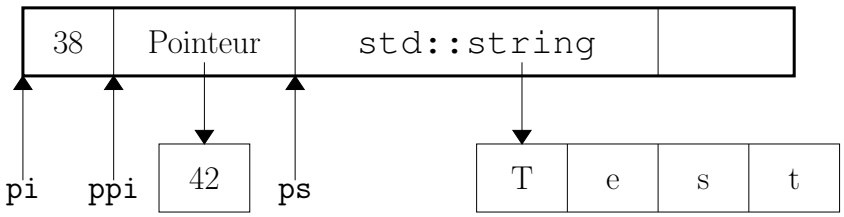

FIGURE 7 – Utilisation de `permalloc` dans une fonction `main`. Le diagramme montre l'état de la mémoire pré-allouée utilisée par `permalloc` juste avant le retour de la fonction `main`. Seules les adresses comprises dans la mémoire pré-allouée (encadrée par un trait épais dans la figure) sont prédictibles et ne dépendront que de celle du début de ce grand bloc.

considère par nécessité comme reproductible. Tandis que c'est acceptable si le code résultant est généré pour une exécution à la volée (car les objets concernés resteront alors à la même place), ce n'est pas souhaitable dans un binaire isolé [4] car cela demanderait alors d'utiliser des parties fixes de l'espace d'adressage du programme. Le travail d'optimisation doit donc être effectué à chaque exécution, ce qui renchérit lourdement l'utilisation d'optimisations avancées sur le code produit par PinaVM : en effet, il faut alors faire ces optimisations à chaque fois que le programme est exécuté.

3.2 L'allocateur `permalloc`

`permalloc` est un allocateur allouant des objets de façon permanente et prévisible [5] dans une zone mémoire pré-allouée. Les adresses des objets alloués, relativement à celle du début de la zone pré-allouée, seront toujours les mêmes comme on peut le voir sur le figure 7, illustrant sur un exemple simple le comportement de permalloc, ainsi que les adresses qui sont prédictibles et celles qui ne le sont pas.

4. voire potentiellement interdit par des systèmes de sécurité tels que SELinux.
5. si on alloue toujours les mêmes objets dans le même ordre, ce qui est suffisant pour l'élaboration d'un système sur puce.

3.3 Obtention de pointeurs lors d'une optimisation permanente

Dans du code ordinaire, lorsqu'on référence un objet, on le fait par un nom qui est résolu par le compilateur, en général soit par adresse relative au pointeur de pile (pour les variables locales) soit par une adresse relative au compteur programme ou absolue (pour les variables globales, selon que le code est indépendant de la position ou non). Dans l'implémentation actuelle, la zone pré-allouée de `permalloc` est un grand tableau déclaré comme variable globale. Si un objet pointé par l'un des pointeurs a été alloué par `permalloc` il a un décalage constant par rapport au tableau sus-cité. Il est donc adressable tel une variable globale, ce qui permet d'utiliser des pointeurs le référençant dans une optimisation permanente.

3.4 Signalement des fonctions optimisées au scheduler

Le scheduler de SystemC contient des pointeurs vers les `SC_THREAD`s et les `SC_METHOD`s qu'il doit exécuter. Or leurs versions spécialisées sont des nouvelles fonctions, donc il faut faire pointer ces pointeurs vers ces dernières. Ceci est suffisant pour lancer la simulation immédiatement, mais pas pour émettre un programme optimisé (puisque les modifications ne sont pas pérennes). Du code est donc ajouté par l'optimiseur pour modifier les pointeurs juste avant la simulation, et les structures contenant les pointeurs sont désormais allouées avec `permalloc` afin de permettre ces modifications.

4 Extension de la portée de l'optim par utilisation d'analyse statique

4.1 La passe ScalarEvolution de LLVM

ScalarEvolution est une passe d'analyse (elle donne des informations sans modifier de code) de LLVM qui fait des analyses de valeur adaptées aux variables de boucle.

4.2 Exemple de code désormais optimisable

Dans la figure 5, dans la boucle `for` de `initiator::thread()`, ScalarEvolution est capable de déterminer que addr est toujours compris entre 4 et 30, ce qui fait que le bus redirigera toujours l'écriture vers le composant Bob : l'optimiseur peut donc modifier la version de `initiator::thread()` destinée à Alice pour appeler directement Bob plutôt que le bus.

Références

[1] Anthony Cataldo. High mask costs seen impeding chip prototypes. Accessible à l'adresse `http://www.eetimes.com/document.asp?doc_id=1216736`, 13 février 2003.

[2] Si-Mohamed Lamraoui. Dedicated Compilation Techniques for SystemC (based on LLVM). Master's thesis, Verimag, Gières, 2011. Dirigé par Claire Maïza et Matthieu Moy, disponible à l'adresse `http://www-verimag.imag.fr/\home\~moy/IMG/pdf/ter-paper-lamraoui-v1-0.pdf`.

[3] Chris Lattner. LLVM : An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[4] Kevin Marquet and Matthieu Moy. PinaVM : a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010. SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic).