PRES LUNAM
École Doctorale STIM
Sciences et Technologies de l'Information et Mathématiques

Spécialité : Informatique
Laboratoire : IRCCyN
Équipe : Temps Réel

Simulation efficace d'architecture temps réel

Bullich, Adrien
Mél : Adrien.Bullich@irccyn.ec-nantes.fr

Résumé : Mon travail de recherche s'inscrit dans le domaine des systèmes embarqués temps réel. Il est souvent difficile d'effectuer des tests sur ces systèmes pour s'assurer de leur bon fonctionnement. Un simulateur présente l'avantage de pouvoir suivre en détail l'évolution de chaque composante. Malheureusement ces simulateurs demandent beaucoup de temps pour être conçus. Un temps qui n'est, de plus, pas capitalisable si des modifications dans l'architecture sont apportées. HARMLESS a été développé pour automatiser cette étape. Il s'agit d'un langage de description d'architecture, lisible par un programme (*gadl*), qui permet la génération d'un simulateur.

Dans le cadre de ma thèse, je m'attache à poursuivre le développement de cet outil de plusieurs manières. J'expose dans cette article la première d'entre elles : utiliser la technique de la simulation compilée.

Mots clés : *Temps réel, architecture matérielle, simulation CAS, simulation interprêtée, simulation compilée.*

1 Simulation d'architecture temps réel et HARMLESS

On retrouve les systèmes embarqués temps réel à de multiples emplois dans notre vie : téléphones portables, avions, voitures... Avec leur essor et leur complexification, il devient un enjeu important de s'assurer de leur bon fonctionnement. Cela pour des raisons de fiabilité (le système répond correctement aux attentes), mais aussi de sécurité (le système est prémuni de comportements dangereux). Plusieurs méthodes existent pour obtenir ce résultat : la vérification par modélisation formelle (*model-checking*), par des procédures de tests ou par la simulation. Ces trois méthodes sont complémetaires quant aux informations qu'elles peuvent apporter. Nous nous intéressons, nous, aux techniques de simulation.

1.1 Les simulateurs dans la littérature

Les simulateurs sont des outils complexes à réaliser et requièrent un long temps de développement. Il est intéressant pour alléger cette étape de pouvoir automatiser leur génération. De nombreux outils existent pour effectuer ce travail. On trouve notamment dans la littérature : le simulateur du langage *nML* [1], *MIMOLA* [2], *LISA* [3], *EXPRESSION* [4]. La thèse de Rola Kassem [5] définit elle aussi un langage de description d'architectures matérielles qui va le permettre : HARMLESS (pour *Hardware ARchitecture Modeling Language for Embedded Software Simulation*). Il présente comme avantage d'être indépendant de l'architecture, d'être incrémental, concis et facilement vérifiable (grâce à son typage). À l'aide de cette description, un compilateur spécifique (*gadl*) est capable de construire ce simulateur. Il s'agit de l'outil que je développe dans le cadre de ma thèse.

1.2 Simulateur ISS et simulateur CAS

On parle de simulateur ISS (pour *Instruction Set Simulator*) pour un outil qui se concentre sur l'aspect fonctionnel de la simulation. Il ne prend donc pas en compte les considérations temporelles, seulement l'exécution des instructions. Or, dans le cadre des systèmes temps réel, il est nécessaire de pouvoir tenir compte des contraintes de temps. Notre compilateur fournit également cette possibilité : générer un simulateur CAS (pour *Cycle Accurate Simulator*), c'est-à-dire précis au cycle près.

1.3 Le modèle du pipeline

Pour obtenir une précision au cycle près, il faut y ajouter la modélisation de la micro-architecture et notamment du pipeline, qui a une grande incidence dans le temps que prend l'exécution d'un programme

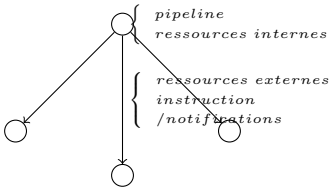

FIGURE 1 – Modélisation de la micro-architecture

(à cause de la gestion des aléas de données, de contrôle ou de structure). Nous utilisons pour cela un automate, qui possède la sémantique suivante, présentée en figure 1. Un état modélise l'état du pipeline à un instant donné, c'est-à-dire les instructions qui se trouvent dans chaque étage, ainsi que l'état des ressources internes. Une transition possède deux éléments : la condition et les notifications. La condition, elle-même, peut se décomposer en une instruction et les ressources externes. Les notifications sont des étiquettes qui permettent à l'utilisateur de recueillir les informations sur les évènements qui surviennent lors de la simulation. Les ressources externes sont des entités pour interagir avec d'autres parties non-modélisées par l'automate. C'est par exemple le cas de la mémoire. Si une instruction dans le pipeline a besoin de faire un accès mémoire, la disponibilité ou non de la ressource associée représentera la possibilité ou non d'accéder à la mémoire. Quant à l'instruction, il s'agit de celle qui se présente à l'entrée du pipeline.

L'automate est construit en prenant initialement un état représentant un pipeline vide. On fait ensuite partir de chaque état autant de transitions qu'il existe de classes d'instructions et de dispositions des ressources externes. L'opération se poursuit sur les états qui ont été rajoutés dans l'étape précédente. L'algortihme s'arrête quand tous les états ont été explorés.

Lors de la simulation, cet automate est parcouru en franchissant les transitions qui correspondent aux instructions lues sur le programme, et selon la disponibilité des ressources externes. Le chemin parcouru modélise ainsi l'évolution du système. Grâce aux notifications que l'on récupère au moment de l'exécution, il est possible pour l'utilisateur d'obtenir le suivi des évènements nécessaire à la vérification.

2 Simulation interprêtée et simulation compilée

La simulation compilée, que l'on oppose à la simulation interprêtée, est la principale contribution de ma thèse. Il s'agit d'une technique de simulation qui permet d'obtenir un gain de temps de la façon suivante. Elle déplace des tâches de l'exécution du simulateur à sa compilation. Dans la littérature, on peut trouver des exemples de simulateur d'architecture matérielle qui utilise la simulation compilée, comme c'est le cas de *MADL* [6] ou de *LISA* [7]. En revanche, dans l'état de nos connaissances, cette technique est spécifiquement utilisée pour les simulateurs ISS. Dans le cadre du domaine temps réel, il apparait important de pouvoir utiliser cette technique pour des simulateurs CAS.

Dans notre cas de figure, la tâche qui est déplacée de l'exécution à la compilation est l'analyse du programme. Le simulateur interprêté fonctionne selon un programme, qui lui est donné en entrée. L'étape de lecture et d'analyse du code se fait donc lors de l'exécution. Pour un simulateur compilé, il n'y a pas à spécifier quel programme est simulé, car ce sera nécessairement celui avec lequel il aura été compilé.

2.1 Comparaison des chaines de développement

Pour bien comprendre quelles différences existent entre les deux procédés, je propose de comparer leurs chaines de développement.

On trouve la chaine de développement associée en figure 2, qui correspond à un simulateur CAS. Pour ce faire, le compilateur (*gadl*) fournit en supplément un fichier de description de la micro-architecture du pipeline. Le programme *p2a* construit sur cette base l'automate (*a2a*) qui modélise cette micro-architecture. Transcrit en programme par le programme *a2cpp*, la compilation finale permet la prise en compte dans le simulateur du paramètre temporel.

Pour ce qui est de la simulation compilée, la chaine de développement est celle présentée en figure 3. La lecture du programme se fait lors de l'exécution de *p2ac* et permet de construire un automate qui modélise l'évolution du pipeline, ainsi que celle dans la lecture du programme. Un module (Analyseur) permet l'analyse du code pour en extraire les informations nécessaires à la construction de l'automate.

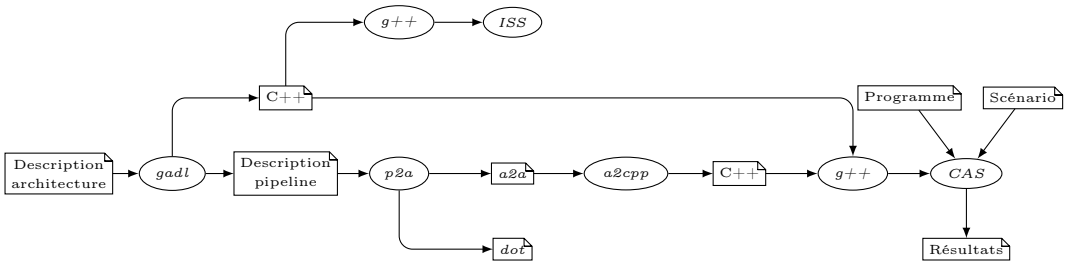

FIGURE 2 – Chaine de développement en simulation interprêtée

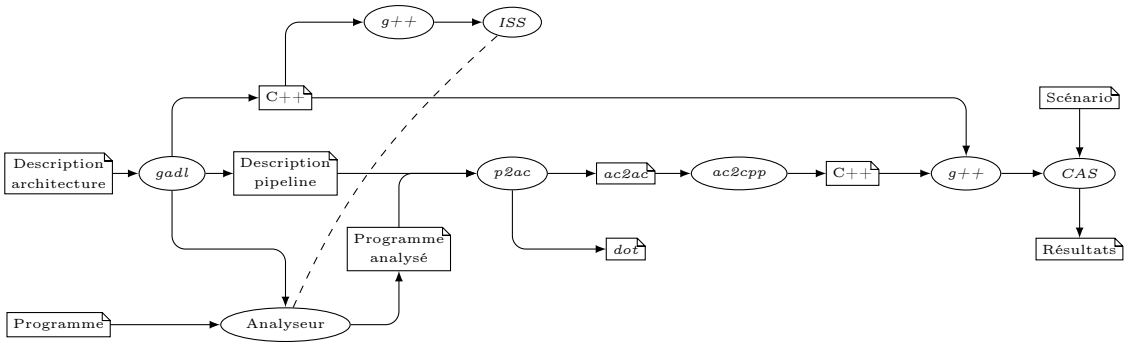

FIGURE 3 – Chaine de développement en simulation compilée

Cette analyse demande des outils propres au jeu d'instructions de l'architecture. Se trouvant présents dans le simulateur ISS, ce dernier est donc d'abord construit pour lire le programme.

On remarque de cette manière à quelle étape dans le processus, le programme est lu. Dans la simulation interprêtée, l'opération est faite lors de l'exécution du simulateur. En revanche, dans la simulation compilée, l'opération se fait lors de la construction du simulateur.

Le choix de développer la simulation compilée offre donc des avantages et des inconvénients. L'action de l'exécution est allégée, tandis que celle de compilation est davantage chargée, mais dans la mesure où cette dernière n'est effectuée qu'une seule fois pour plusieurs exécutions, cela représente un gain de temps dans l'ensemble du processus. En revanche, le simulateur obtenu est non seulement propre à une architecture matérielle donnée, mais également à un programme donné. Son utilisation en devient plus restreinte et ne fournit de résultats plus qu'en fonction d'un scénario de test.

2.2 Modèle de simulation compilée

Le modèle précédemment utilisé en simulation interprêtée pour représenter la micro-architecture n'est plus suffisant. En effet, dans notre nouvelle perspective, nous devons pouvoir suivre l'évolution de la micro-architecture et celle du programme que l'on lit. Les modifications à apporter sont ainsi les suivantes.

Un état ne représente plus seulement l'état du pipeline à un instant donné, mais également l'état dans la lecture du programme. C'est-à-dire qu'une état contient comme information les instructions présentes dans les étages du pipeline, et le *Program Counter* (qui donne l'adresse de l'instruction lue). Les transitions contiennent quant à elles les mêmes informations que dans le modèle précédent, sinon que l'instruction n'est plus considérée comme une condition, mais comme une étiquette.

La construction de l'automate présente également des différences. Elle se fait en partant d'un état représentant un pipeline vide et un *Program Counter* au début du programme. De chaque état on construit une transition pour chaque disposition possible des ressources externes. (Et non plus des dispositions des ressources externes et des instructions qui peuvent entrer dans le pipeline.) La construction se poursuit avec les nouveaux états qui sont construits, jusqu'à ce que la totalité des états accessibles aient été explorés.

L'exécution de l'automate se fait en évaluant lors de la simulation la disponibilité des ressources externes. Celles-ci permettent seules de définir la transition à franchir. Une transition franchie donne comme information les notifications pour renseigner l'utilisateur, mais également quelle instruction est lue, afin

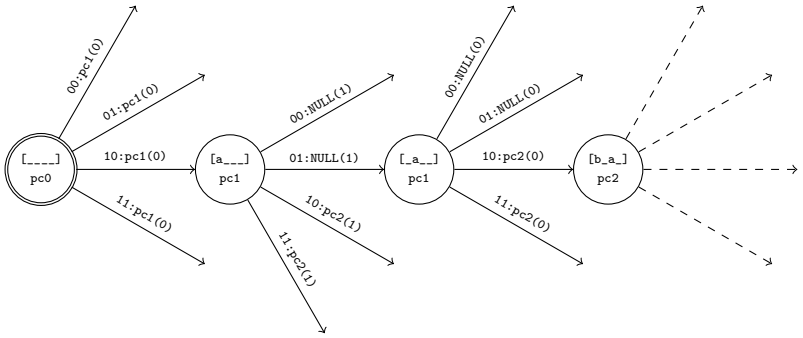

FIGURE 4 – Automate en simulation compilée

de l'exécuter.

On peut, en figure 4, voir un exemple d'automate. Dans les états, je représente deux choses : l'état du pipeline ([a_ _ _] signifie qu'une instruction a se trouve dans le premier étage, quand les autres étages sont vides), et l'état dans la lecture du programme (c'est-à-dire le PC (*Program Counter*)). Pour les transitions, j'utilise la syntaxe suivante : a :b(c), où a est l'état des ressources externes (on note 01 pour signifier que la première ressource est disponible (1) et la deuxième ne l'est pas (0)), où b est l'adresse de l'instruction (le PC) qui rentre dans le pipeline, et où c est l'état des notifications (on note 1 pour une notification qui survient et 0 pour une notification qui ne survient pas).

Dans notre exemple, l'instruction b nécessite d'utiliser la deuxième ressource pour accéder au premier étage du pipeline. La seule notification utilisée informe sur la rentrée ou non d'une instruction dans le deuxième étage du pipeline.

3 Perspectives

De mon travail sur la simulation compilée, l'avancement est près de la finalisation. Les principaux éléments ont déjà été programmés. Ne reste qu'à apporter des modifications au compilateur *gadl* pour qu'il tienne compte de ce nouveau type de fonctionnement.

Pour le reste de ma thèse, m'attendent deux autres voies pour développer notre outil de simulation. En premier lieu, la gestion des architectures multi-cœurs, et en second lieu, la prise en compte des pipelines plus modernes comme le multi-scalaire.

Références

[1] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. *EDTC'95 : Proceedings of the 1995 European Conference on Design and Test*, page 503, 1995.

[2] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, and Dietmar Voggenaur. The mimola language version 4.1. Technical report, 1994.

[3] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. *DAC'99 : Proceedings of the 36th ACM/IEEE conference on design automation*, pages 933–938, 1999.

[4] A. Halambi, P. Grun, and al. A language for architecture exploration through compiler/simulator retargetability. *European Conference on Design, Automation and Test (DATE)*, 1999.

[5] Rola Kassem. *Langage de description d'architecture matérielle pour les systèmes temps réel.* PhD thesis, Université de Nantes, 2010.

[6] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, 2004.

[7] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des Autom. Electron. Syst.*, pages 815–834, 2000.