Simulation efficace d'architectures opérationnelles

Adrien Bullich

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Adrien BULLICH

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
Label européen
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)

Soutenue le 10 décembre 2014

Simulation efficace d'architectures opérationnelles

JURY

Présidente :	**M^me^ Nathalie JULIEN**, Professeur d'Université, Lab-STICC, Lorient
Rapporteurs :	**M. Michel AUGUIN**, Directeur de Recherche CNRS, LEAT, Sophia Antipolis
	M. Pascal SAINRAT, Professeur d'Université, IRIT, Toulouse
Examinateurs :	**M. Mikaël BRIDAY**, Maître de Conférence, Université de Nantes
	M. Jean-Luc BÉCHENNEC, Chargé de Recherche CNRS, IRCCyN, Nantes
Directeur de thèse :	**M. Yvon TRINQUET**, Professeur d'Université, Université de Nantes

Remerciements

Avant tout, je remercie mon directeur de thèse Yvon Trinquet, ainsi que mes deux encadrants : Mikaël Briday et Jean-Luc Béchennec. C'est essentiellement à eux que je dois ces trois années de développement et d'épanouissement. Leurs précieux conseils m'apporteront toujours un éclairage important sur mes futurs travaux de recherche. Je leur suis reconnaissant de leur ouverture d'esprit, de leur disponibilité, de leur gentillesse et, plus que tout, d'avoir cru en moi.

C'est aussi toute l'équipe des Systèmes Temps Réel de l'IRCCyN à qui j'adresse mes remerciements pour leur accueil chaleureux et la place qu'ils m'ont accordé en leur compagnie.

Je remercie les membres du jury, avec à leur tête Madame Nathalie Julien, pour le regard bienveillant qu'ils voudront porter sur mon travail et les contributions de ce mémoire.

Une thèse est une successions de péripéties parfois douloureuses et on la supporte avec peine lorsque l'on est tout seul. J'avais à mes côtés deux collègues de bureau pour m'épauler, à la fois sympathiques et à l'écoute : Julien Tanguy et Johan Girault. Ils ont pu être les témoins privilégiés des petits tracas de la vie d'un thésard. Je les en remercie pour cela et je leur souhaite bonne chance et bonne continuation dans leurs projets respectifs.

Bien au delà du cadre restreint de mon bureau, c'est toute une joyeuse bande de thésards, de masters et de stagiaires, qui par leur bonne humeur communicative ont teinté mes trois années de thèse d'une joie revigorante. En remerciement de tous ces moments fraternels, je leur souhaite d'atteindre le bonheur, chacun dans la voie qu'il s'est destiné.

1

Introduction générale

Les systèmes informatiques embarqués occupent une grande place dans notre quotidien et sont amenés à en occuper une plus grande encore dans l'avenir du fait de la croissante informatisation de notre société. On les retrouve à tous les niveaux de notre vie, dans les avions, les voitures, les téléphones portables, etc. Leur étude est d'autant plus nécessaire que les systèmes utilisés deviennent complexes. Assurer leur bon fonctionnement représente alors un travail important et doit faire appel à des outils d'analyse poussés.

Dans le cadre de ce bon fonctionnement, les contraintes temporelles peuvent prendre une grande importance. C'est d'autant plus compréhensible pour certains domaines critiques assurant la sécurité, comme un système de contrôle de frein dans une voiture. Notre thèse s'ancre dans cette problématique.

Elle s'intéressera tout particulièrement à la *Simulation efficace d'architectures opérationnelles*. Ce mémoire rendra compte des développements proposés au cours de cette thèse pour améliorer l'efficacité de la simulation temporelle précise au cycle près, notamment par un travail sur la simulation compilée et l'abstraction de programme.

Cette thèse s'est déroulée au sein de l'équipe *Temps réel* de l'IRCCyN, sous la direction d'Yvon Trinquet, avec l'encadrement de Mikaël Briday.

1.1 Système temps réel

Les systèmes temps réels sur lesquels nous allons baser notre étude se caractérisent principalement par les contraintes temporelles qui pèsent sur eux. Un système informatique peut être soumis à des contraintes temporelles de différents degrés, en fonction de la problématique à laquelle il répond. On utilise habituellement trois catégories pour les distinguer [HP85] :

Systèmes transformationnels : le résultat est donné sans contraintes temporelles. C'est le cas pour les calculs scientifiques. Même si un temps humainement raisonnable peut être apprécié, les gains du résultat espéré vis-à-vis de l'avancement d'une recherche justifie les concessions faites sur le temps que demande ces calculs. Les compilateurs font un autre bon exemple à cette catégorie.

Systèmes interactifs : le système interagit avec le procédé commandé, il est demandé un temps de réaction court. On peut imaginer le cas d'un système de réservation. L'interaction avec

un emploi du temps et des utilisateurs demandent une certaine synchronisation.

Systèmes réactifs : le système interagit avec le phénomène qu'il mesure, son délai de réaction doit s'approcher du temps réel, pour que ce délai soit adapté à l'environnement (le système informatique gérant le pilotage d'un avion, comme exemple). Les systèmes embarqués (ou enfouis), dont la caractéristique est l'autonomie et la spécialisation pour réaliser une tâche donnée, sont souvent des systèmes réactifs. Ces systèmes embarqués sont généralement soumis à des contraintes d'ordre énergétique et spatial (espace réduit).

Le cadre de la thèse se situe dans cette troisième catégorie : les applications temps réel. On entend usuellement par temps réel une notion d'immédiateté de la réponse. C'est, en réalité, une notion à relativiser. Une définition précise du temps réel repose sur deux concepts [Ta06] :

- En premier lieu, un système temps réel réagit à son environnement, c'est-à-dire qu'il est sensible à l'avènement de certains évènements et peut infléchir son comportement en fonction de ceux-ci. (Voir Figure 1.1.) C'est un système réactif.

- En second lieu, un système temps réel a un temps de réponse adapté à l'environnement sur lequel il agit, *i.e.* son temps de réaction doit être du même ordre de grandeur que le temps caractéristique du phénomène qu'il mesure. Le caractère réel du temps n'a donc que peu à voir avec la perception humaine de l'instantanéité. Il peut même poser une contrainte plus lâche que celle-ci, si l'environnement avec lequel le système est en interaction a un temps propre d'évolution lent.

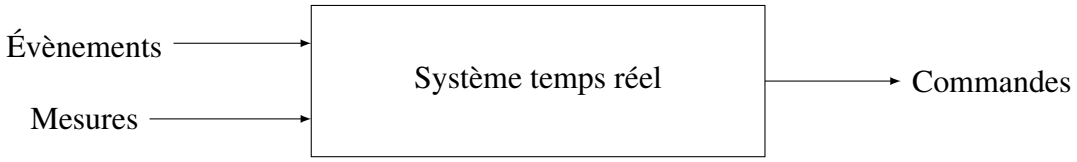

FIGURE 1.1 – Schéma du fonctionnement d'un système temps réel. Sur la mesure de son environnement et par sa sensibilité à certains évènements le système fournit les commandes adaptées.

La problématique que pose les contraintes des systèmes temps réel est un champ très vaste d'études. Il recouvre un ensemble de domaines englobant l'informatique, l'électronique et l'automatique. Ces contraintes sont à la fois d'ordre fonctionnel (par l'exactitude logique des résultats donnés) et d'ordre temporel (le temps auquel il donne ses résultats).

Un système temps réel peut être soumis à différents degrés de contraintes que pose le temps de réponse. L'impact d'une défaillance pour le système ne présente alors pas la même criticité, et n'entraînera pas les mêmes types de moyens pour assurer le respect des échéances auxquelles il est soumis.

On définit généralement une défaillance comme un évènement qui dévie le service rendu par le système du service correct, c'est-à-dire du service qui est attendu. C'est un concept de transition, caractérisant le passage d'une phase de bon fonctionnement à un mauvais fonctionnement. La panne est, quant à elle, l'état du système dans lequel le service rendu est incorrect, c'est-à-dire qu'il ne remplit plus la fonction du système.

Pour articuler la variété du niveau de contraintes des systèmes temps réel, on fait appel à trois catégories.

Contraintes souples (*soft real time*) : une défaillance n'entraîne qu'une dégradation de l'intérêt du système. Un service multimédia de diffusion de vidéos peut présenter cette propriété. En cas de ralentissement de la connexion, la qualité de la vidéo se dégrade progressivement, causant un désagrément pour l'utilisateur. Si un tel comportement n'est pas désirable, il ne remet pas en cause la fonction du système et ne présente aucun danger. La problématique peut se présenter alors comme une question de qualité du service.

Contraintes fermes (*firm real time*) : quoique la contrainte doive être respectée pour le bon fonctionnement du système, des défaillances rares et ponctuelles peuvent être tolérées.

Contraintes strictes (*hard real time*) : de la même manière que les contraintes fermes, le bon fonctionnement du système est engagé par le respect des contraintes. S'ajoute à cela des considérations sur la gravité des conséquences que peut causer une défaillance, que cela soit pour l'environnement contrôlé, ou l'utilisateur. On peut évoquer tout ce qui est système de contrôle dans une centrale nucléaire.

Tenir compte de ces contraintes demande de bien comprendre qu'une défaillance peut provenir de son utilisation ou de sa conception. On considérera qu'il y a défaillance si le système cesse son aptitude à fournir un service correct ou si les spécifications fonctionnelles attendues ne correspondent pas au système conçu. De cette manière, il est possible de répondre à cette problématique par des notions de sûreté de fonctionnement ou par le développement de systèmes tolérants aux fautes [ACD+06]. Cette dernière passe notamment par l'utilisation de composants résistants aux défaillances, ou encore par leur redondance. Mais il est également nécessaire de valider les spécifications fonctionnelles dans la conception des systèmes, d'autant plus au vu de la complexité des systèmes informatiques actuels. Cette validation rend obligatoire l'utilisation de modèles d'application vérifiables.

1.2 Validation des systèmes temps réel

La validation d'applications temps réel est un vaste et difficile problème. Elle doit être menée le long du cycle de développement sur les aspects fonctionnels et extra-fonctionnels (les aspects temporels, la sécurité, etc.). Cette validation peut se faire de deux façons : sur modèle ou sur cible (avec des tests). Dans ce mémoire, notre travail se place à la dernière étape du processus de conception, avant la phase finale de test sur la cible réelle.

1.2.1 Validation sur modèle

Dans la validation sur modèle, nous retrouvons trois techniques complémentaires. Tout d'abord, l'analyse formelle, qui utilise des outils de modélisation pour vérifier des propriétés. L'analyse d'ordonnançabilité vérifie le bon ordonnancement des tâches. Enfin, l'analyse par simulation utilise la simulation pour effectuer des scénarios de tests.

On retrouve sur la Figure 1.2, le positionnement de chacune de ces analyses dans le cycle de développement en V. L'analyse d'ordonnançabilité s'appuie sur les pires temps d'exécution (WCET) et nécessite, par conséquent, pour être effectuée que le codage soit terminé. Quant à l'analyse par simulation, elle travaille directement sur le code binaire et ne peut donc être faite avant sa conception.

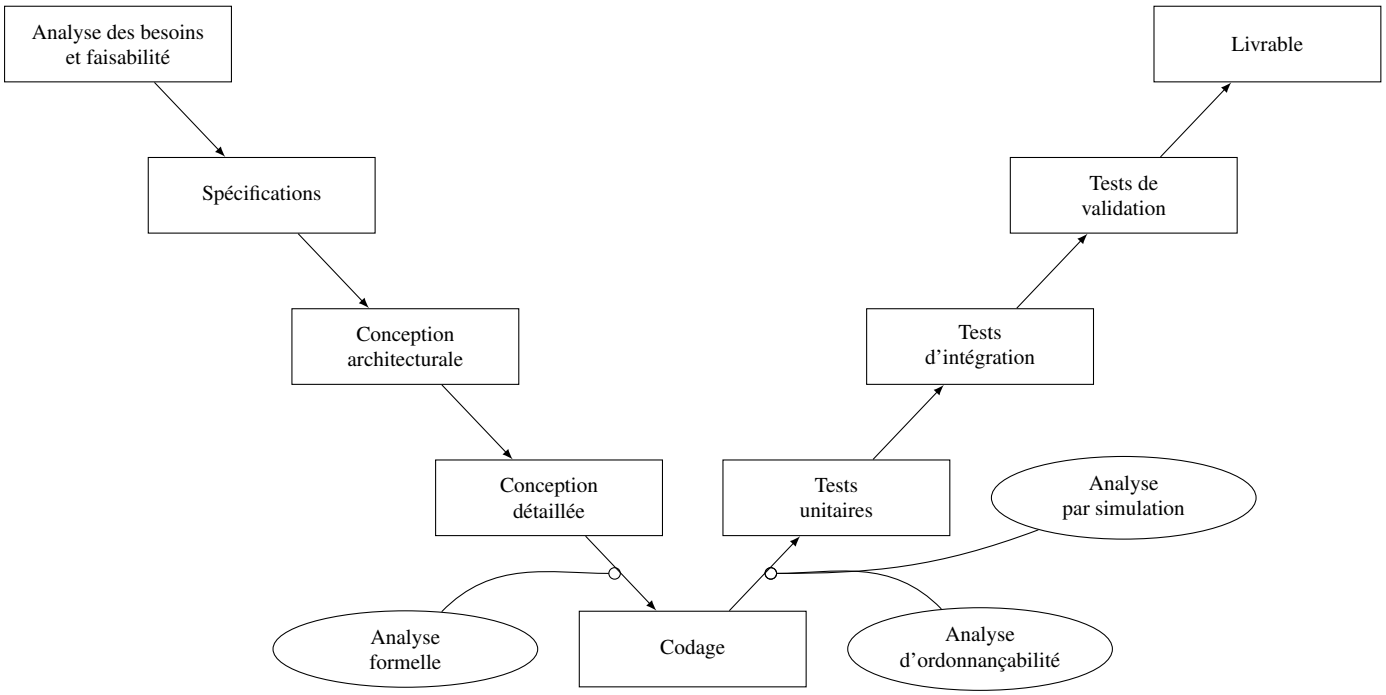

FIGURE 1.2 – Le Cycle en V. Notre travail trouve son intérêt après le codage avant les séries de tests.

Analyse formelle

On parle de vérification de modèle (*model-checking*) [Sc99]. Par un travail d'abstraction du système, il est possible de construire une représentation du système dans un formalisme particulier. Ce modèle offre ensuite la possibilité d'être exploré par des techniques automatiques de manière à vérifier des propriétés sur le système, telles que l'atteignabilité (assurant qu'un état du système peut être atteint), la sûreté (il est sûr que rien de mauvais ne peut se produire) ou la vivacité (il est sûr que quelque chose de bon peut inévitablement se produire).

Dans les formalismes utilisés, on retrouve généralement les automates ou les réseaux de Petri, pour ce qui est des modèles discrets. Mais il existe de nombreuses extensions et, notamment, pour intégrer les propriétés temporelles : les automates temporisés et les réseaux de Petri T-temporels.

Située durant la phase de conception, cette approche permet d'anticiper et de répondre au plus tôt aux problèmes constatés. Elle présente aussi l'avantage d'offrir un caractère exhaustif aux propriétés vérifiées. Les algorithmes d'exploration assurent que les propriétés sont vérifiées ou non, dans tous les cas de figure possibles.

Ces résultats, qui s'appuient sur des preuves mathématiques, peuvent toutefois être limités dans leur portée du fait de la distance incertaine entre l'implantation réelle et le modèle étudié. L'analyse formelle se confronte à une autre difficulté : les limitations de puissance de calcul. Quand le système est complexe à modéliser, la construction du modèle peut faire face à une explosion combinatoire des états, empêchant toute étude.

Analyse d'ordonnançabilité

Une application temps réel se divise en une série de tâches qui doivent être exécutées par le processeur (ou les processeurs). Comme elles ne peuvent pas l'être simultanément sur le même processeur, celui-ci a la charge de gérer leur ordonnancement, c'est-à-dire d'organiser la ré-

partition de son temps de calcul entre les différentes tâches concurrentes. Le problème se pose également avec plusieurs processeurs (architectures multiprocesseurs), bien qu'un certain parallélisme puisse être alors appliqué. Pour assurer les propriétés temporelles, l'analyse d'ordonnançabilité permet de garantir qu'une répartition des tâches permet bien de respecter les contraintes temporelles. Elle renseigne également sur les performances qu'elle induit.

On peut caractériser une tâche par plusieurs données : son caractère périodique ou sporadique, sa période, son offset, son échéance, son temps d'exécution, sa priorité, ses relations de dépendance, etc. L'état de la tâche entre aussi en considération : elle peut être inactive, en attente ou en cours d'exécution. Différentes techniques d'ordonnancement existent pour résoudre le problème de leur répartition, comme le *Round-Robin* répartissant les tâches à tour de rôle, où l'*Earliest Deadline First* (EDF), faisant passer en priorité la tâche dont l'échéance est la plus proche.

Pour s'assurer qu'une telle répartition des tâches permet de respecter les contraintes temporelles, l'analyse d'ordonnançabilité peut faire appel à des formules analytiques ou des algorithmes complexes de calcul des temps de réponse. Elle est effectuée sur la base d'une configuration particulière des tâches et la méthode d'ordonnancement. Elle peut être opérée soit de façon hors ligne (déterminée avant l'exécution) soit en ligne (déterminée au cours de l'exécution) [DC06].

Analyse par simulation

Contrairement aux méthodes précédentes, la simulation n'offre pas de caractère exhaustif aux résultats. En effet, elle n'apporte qu'une assurance sur les cas simulés, qui ne peuvent pas prétendre à la généralité.

L'analyse par simulation consiste à se donner la possibilité d'effectuer des tests au travers d'un modèle qui simule fidèlement le fonctionnement de l'application. Dans le domaine du temps réel, effectuer des tests sur la cible pose de nombreuses difficultés. Le matériel, nécessairement réduit en embarqué, n'offre pas toujours les fonctionnalités requises pour la vérification. L'environnement en interaction n'est pas toujours à portée d'expérimentation ou manipulable pour mettre en œuvre des tests prédéfinis et répétables. Enfin, effectuer des tests sur cibles peut représenter un certain coût, du fait des machines utilisés (prenons, par exemple, les *crash tests* de voiture). La simulation répond à ces différents problèmes. Elle a notamment la capacité d'offrir des résultats complémentaires aux phases de tests, grâce à la large palette de résultats qu'elle peut fournir.

Il existe plusieurs moteurs de simulation. On peut, par exemple, citer parmi les plus connus : Matlab, Simulink, SystemC. Ces simulateurs peuvent être à temps discret ou continu. Un simulateur à temps continu effectue un rafraîchissement de l'état du système à chaque pas du temps. Tandis qu'un simulateur à temps discret attend la venue d'un évènement pour effectuer ce rafraîchissement. Matlab et Simulink utilisent la simulation à temps continu. SystemC utilise la simulation à temps discret.

L'analyse par simulation fait face aux mêmes défauts que l'utilisation de phases de tests. Elle peut notamment demander un nombre conséquent de scénarios à simuler pour valider le système. Cela implique un certain choix dans les scénarios de telle manière à faire apparaître les différentes contraintes du système. À ceci s'ajoute que, ayant recours à une modélisation du système, le résultat d'une simulation peut différer sensiblement de la réalité si la distance entre le modèle et le système est trop grande.

La validation par simulation de systèmes temps réel impose de travailler sur les temps d'exécution. On distingue des simulateurs fonctionnels et des simulateurs temporels. Dans le cadre de cette thèse, ce sont ces seconds qui nous intéressent.

1.2.2 Validation sur cible

La validation sur cible se passe en fin de développement, sur le produit fini. Elle consiste principalement en des procédures de tests. Elle permet de vérifier la qualité, le respect des exigences techniques et l'absence de défauts.

De la même manière que l'analyse par simulation, elle ne garantit pas l'exhaustivité des résultats, et se confronte à la complexité de la couverture des différents cas de contraintes du système. Des générateurs de tests existent toutefois pour répondre à cette difficulté.

1.3 Simulation d'architectures matérielles

La simulation d'architectures matérielles est le cœur de notre travail de thèse. Dans cette section, nous mettons en place les éléments pour cerner cet environnement : quels types d'architectures matérielles on peut rencontrer, les deux types de simulateurs que nous avons manipulés et les langages de descriptions des architectures matérielles.

1.3.1 Architectures matérielles

Plusieurs types d'architectures matérielles existent. Ils n'ont pas tous été exploré au cours de cette thèse. Pour bien cerner le cadre de notre travail, il faut distinguer, dans les systèmes temps réel, plusieurs types d'architectures :

Architecture monoprocesseur : le système possède un seul processeur. Il a la charge de toutes les tâches de l'application, qui se partagent donc le temps de calcul du processeur. Plusieurs tâches peuvent s'exécuter dans un pseudo-parallélisme : leur exécution est décomposée en sous-partie pour être rapidement alternée.

Architecture multiprocesseur : le système est composé de plusieurs processeurs. Ils peuvent disposer de mémoires partagées et d'autres privées. Ce type d'architecture permet de mettre en œuvre un véritable parallélisme dans l'exécution des tâches.

Architecture multicœur : le système est composé de plusieurs cœurs, disposant chacun d'un ou plusieurs caches. En revanche, les différents cœurs ont accès à la même mémoire.

Architecture répartie : le système est composé de plusieurs machines : processeurs et mémoires. À la question de la répartition des tâches par les processeurs, s'ajoute la question de la gestion du réseau.

Le simulateur associé à HARMLESS ne permet la simulation que des architectures monoprocesseurs, en l'état actuel des choses. Dans le travail de cette thèse, nous nous sommes concentrés sur la simulation de ce type d'architectures, bien que des pistes aient été abordées pour généraliser nos contributions aux architectures multiprocesseurs.

1.3.2 Instruction Set Simulator et Cycle Accurate Simulator

Comme nous l'avons dit, les systèmes temps réel présentent une double exigence : la fiabilité logique des résultats délivrés (l'aspect fonctionnel) et le respect des échéances pour délivrer ces résultats (l'aspect temporel). C'est la raison pour laquelle, dans le cadre de cette thèse, on distingue deux types de simulateurs : les simulateurs fonctionnels et les simulateurs temporels.

Instruction Set Simulator

Un *Instruction Set Simulator* (ISS) est un simulateur reproduisant le comportement de l'architecture, et délivrant des informations fonctionnelles sur son exécution. Dans les informations qu'il peut délivrer, on peut citer les résultats des calculs effectués ou l'évolution du contenu des registres. Ce type de simulateurs est le plus rapide à l'exécution. Il a fait l'objet d'un important travail d'optimisation dans la littérature, comme nous le verrons dans le chapitre 2 sur l'état de l'art.

Comme son nom l'indique, un ISS simule un jeu d'instructions. Un jeu d'instructions est l'ensemble des instructions qu'utilise une processeur. On y retrouve des opérations logiques ou arithmétiques, mais aussi des opérations de manipulation des données dans le matériel. La description d'un jeu d'instructions, nécessaire pour implanter un ISS, se compose du codage de ces instructions. Elles permettent à l'ISS d'interpréter le code d'un programme dans le langage machine de l'architecture simulée, d'évaluer les opérandes des instructions et de reproduire dessus les opérations de l'instruction.

Cycle Accurate Simulator

Un *Cycle Accurate Simulator* (CAS) est un simulateur permettant de fournir des informations temporelles sur l'exécution du code dans l'architecture. De plus, ces informations sont précises au cycle processeur près. Il permet ainsi d'obtenir des résultats comme le pire temps de calcul (WCET), ou le timing d'apparition d'évènements donnés.

Un simulateur temporel est aussi capable de donner des résultats fonctionnels, dans une certaine mesure. En effet, manipuler des informations fonctionnelles sur l'exécution permet de déterminer entre autres le flot de contrôle du programme. Celui-ci est fondamental pour déterminer les informations temporelles, telle que le pire temps d'exécution.

Si un CAS présente l'avantage d'offrir des résultats temporels sur la simulation, il présente en contrepartie une relative lenteur dans le temps de calcul, en comparaison notamment d'un ISS. Dans [KBB+11], le CAS est environ 7 fois plus lent que le simulateur fonctionnel sur un simple processeur PowerPC avec un pipeline à 5 étages. Ainsi, on peut constater qu'il existe une certaine marge d'amélioration de l'efficacité d'un CAS. Dans cette thèse, nous proposons d'explorer des techniques pour améliorer la vitesse d'exécution du CAS.

Les propriétés temporelles d'une architecture sont grandement déterminées par le matériel utilisé. On peut citer l'impact du pipeline dans la détermination du timing et, plus en particulier, le rôle des différents aléas, comme nous le verrons dans le chapitre 3 traitant du modèle de processeur utilisé.

Pour le travail de ce mémoire, la chaîne de compilation autour du langage HARMLESS [KBB+11] a été utilisée pour construire les simulateurs. Elle permet de générer à la fois un *Instruction Set Simulator* (ISS), qui fournit des résultats fonctionnels, et un *Cycle Accurate Simulator* (CAS), qui fournit des résultats temporels. De par l'intérêt du timing des applications dans les systèmes temps réels, c'est sur ce dernier type de simulateur que nous focaliserons la thèse.

1.3.3 Langage de description d'architecture

Il n'est pas possible de tout représenter dans le modèle de simulation. Un certain niveau d'abstraction est nécessaire pour éviter que la taille du modèle ne porte préjudice aux performances de la simulation. Un modèle de simulation doit donc être choisi en fonction du domaine étudié, des objectifs poursuivis et du niveau d'abstraction requis, de manière à mettre en avant les éléments pertinents à l'étude.

La simulation d'architectures matérielles demande un niveau d'abstraction bas, par conséquent son implantation en est complexe et peut prendre beaucoup de temps. De plus, dans un développement « à la main », des erreurs sont difficiles à éviter et rendent la distance vis-à-vis du système réel plus incertaine. On peut compter près d'une année à la conception d'un simulateur avec ce type de développement. En outre, le travail effectué pour une cible donnée est difficilement capitalisable. Si l'architecture évolue un peu, c'est tout le simulateur qu'il faut réécrire.

Pour résoudre les problèmes de développement des simulateurs, des techniques existent pour automatiser cette tâche. On fait pour cela appel à des langages de description d'architecture matérielle, *Hardware Architecture Description Language* (HADL). On retrouve plus usuellement le terme de *Architecture Description Language* (ADL), mais il peut prêter à confusion avec les ADL logiciels. Les HADL sont nés de l'intérêt croissant pour les *Application Specific Instruction set Processors* (ASIP), les *Digital Signal Processors* (DSP), les *System On Chips* (SOC). La spécialisation de processeurs demandait de pouvoir manipuler dans un langage la description d'une architecture.

Les HADL présentent de nombreux avantages. Ils permettent :

- de vérifier l'architecture, grâce au niveau d'abstraction qu'ils offrent,
- d'apporter des modifications rapides et évaluer les performances qu'elles entraînent,
- d'accélérer ainsi l'obtention d'un modèle de processeur.

On trouve de nombreux HADL dans la littérature. Ceux-ci peuvent avoir différents emplois et décrire différents niveaux d'une architecture. C'est pour cette raison que l'on utilise une double classification : selon l'objectif et selon le contenu [MD08].

On distingue généralement deux niveaux dans la description d'une architecture : la structure et le jeu d'instructions. Selon ce qui est décrit, il n'est pas possible de remplir les mêmes objectifs. On peut toutefois décrire les deux niveaux.

HADL orientés structure : Il décrit la structure interne de l'architecture, pour décrire les connexions entre les différents composants et leur fonctionnement. (MIMOLA [BBH+94])

HADL orientés jeu d'instructions : Il décrit le jeu d'instructions, mais pas les composantes matérielles. On retrouve donc le codage des instructions et leur comportement. (nML [FVPF95] ou son extension SIM-nML [Raj96], ISDL [HHD97])

HADL mixtes : Il décrit à la fois le jeu d'instructions et la structure interne et permet ainsi de s'adapter à tous les objectifs : validation, synthèse, simulation, compilation. (LISA [PHZM99], EXPRESSION [HGa99], MADL [QRM04])

Un HADL peut trouver de multiples applications, dans la conception d'architectures ou la simulation, par exemple. Sa structure varie nécessairement selon la fonction qu'il veut porter, aussi faire une distinction des HADL suivant leur objectif a son importance.

HADL orientés validation : ils ont l'ambition de permettre aux concepteurs de générer des tests pour valider fonctionnellement les processeurs.

HADL orientés synthèse : ils représentent un outil flexible pour la conception d'architectures matérielles. Un de leurs avantages, à ce niveau-là, est d'offrir la possibilité de mettre en place des techniques de vérification sur les architectures proposées en analysant sa description.

HADL orientés compilation : ils permettent de générer automatiquement des compilateurs « reciblables ».

HADL orientés simulation : ils offrent la possibilité de construire des simulateurs d'architectures sur la base de leur description. Ce sera cette utilisation qui nous intéressera dans ce mémoire.

On peut noter que ces deux classifications ne sont pas indépendantes. Faire de la synthèse ou de la validation, nécessite de travailler sur la structure de l'architecture et donc de détenir des informations s'y rapportant dans le HADL. En revanche, un HADL ayant pour fin la compilation ou la simulation focalise son travail sur le fonctionnement des programmes et passe donc par la description du jeu d'instructions. Un HADL mixte est adapté à tous les objectifs, puisqu'il renferme toutes les informations requises pour chacun des objectifs. (Voir la Figure 1.3.)

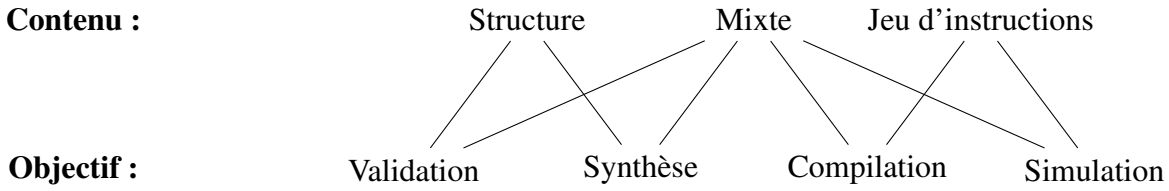

FIGURE 1.3 – Classification des HADL

Comme c'est l'aspect de la simulation qui nous intéresse dans ce mémoire, il est utile de noter que, bien qu'un simulateur fonctionnel (ISS) ne nécessite qu'un HADL décrivant le jeu d'instructions, dans le cadre d'un simulateur temporel, il est nécessaire de posséder une description de la structure de l'architecture. En effet, le comportement temporel est fortement influencé par la structure de l'architecture (dont le pipeline, comme on le verra), et sa simulation passe par une modélisation de celle-ci. Un simulateur temporel ne peut donc être généré qu'à partir d'un HADL mixte.

1.4 Motivation de la thèse et contributions

Dans le cadre des systèmes temps réel, où le respect des contraintes sur le temps est critique, la validation du comportement temporel reste primordiale. C'est dans cette perspective que nous fixons notre thèse.

Cette validation est cependant bornée par la difficulté de simuler temporellement ces systèmes. En effet, la simulation temporelle (CAS) peut s'avérer très lente en comparaison de la simulation fonctionnelle (ISS). Les architectures matérielles sont des systèmes complexes et longs à simuler finement. Pour donner un ordre de grandeur, le simulateur associé à HARMLESS sur des benchmarks donne la vitesse d'exécution du CAS 3 fois plus lente que celle de l'ISS.

Pour répondre à cette difficulté, nous nous attacherons à développer, dans le cadre de cette thèse, des techniques pour rendre plus rapide l'exécution des CAS. Notre travail portera sur trois éléments en particulier :

- la simulation compilée : elle consiste à déplacer des tâches d'analyse du temps d'exécution à celui de compilation ;
- l'abstraction de programme : le modèle ne manipule plus des instructions mais des blocs d'instructions ;

- une autre forme d'abstraction du programme : on retire de la simulation les instructions n'intervenant pas dans le flot de contrôle.

1.5 Organisation du mémoire

L'organisation du mémoire est la suivante. Il est découpé en huit chapitres et une annexe.

Au cours de ce chapitre 1 nous avons établi le contexte dans lequel s'inscrit le travail de thèse. Il s'agit de la validation des systèmes temps réel par des méthodes de simulation. La génération de simulateurs se fait sur la base de la description du processeur, grâce à un langage de description d'architecture.

Le chapitre 2 dresse un état de l'art pour présenter les différentes techniques existantes dans la littérature pour améliorer le temps d'exécution des simulateurs d'architectures.

Le chapitre 3 donne les bases du fonctionnement de la simulation interprétée sur HARMLESS et permettra de comprendre la modélisation utilisée au cours de ce mémoire. Il y est exploré le formalisme que reprennent les contributions de ce mémoire, ainsi que les méthodes algorithmiques utilisées pour la construction du modèle interprété.

Le chapitre 4 présente la première contribution du travail de thèse. Il s'agit du premier modèle qui a été construit pour reprendre les principes de la simulation compilée dans le modèle du processeur. Il repose sur l'introduction du programme à simuler dans le modèle processeur.

Le chapitre 5 présente la deuxième contribution du travail de thèse. Il s'agit d'un nouveau modèle plus poussé que le précédent sur la simulation compilée. Celui-ci permet notamment de résoudre certaines limitations du premier modèle, et ouvre notamment la voie à la simulation de programmes utilisant la gestion logicielle des calculs sur les nombres réels.

Le chapitre 6 présente une technique d'optimisation du modèle précédent, ce qui est la troisième contribution du mémoire. La méthode consiste dans une forme d'abstraction du code. Ainsi le modèle utilisé ne manipule plus simplement des instructions, mais des suites d'instructions, ce que l'on appelle des macro-instructions.

Le chapitre 7 dresse le bilan d'une technique qui a été explorée au cours de la thèse. Elle consiste aussi en une forme d'abstraction du code : le simulateur n'exécute que les instructions nécessaires à l'exploration du modèle.

Enfin, dans le chapitre 8 une conclusion synthétisera l'ensemble des travaux effectués et fera le point sur les perspectives qu'offre ce travail.

L'annexe A fournit des informations sur le fonctionnement du pipeline.

2

État de l'art

Dans ce chapitre, nous présentons la littérature existante sur la simulation d'architectures matérielles, et plus précisément les différentes techniques qui existent pour l'implanter. Cet état de l'art a pour objectif de dresser un premier bilan des stratégies d'implantation existantes, de leurs avantages et de leurs inconvénients, notamment dans le cadre de l'amélioration du temps d'exécution. Elle permettra ainsi de tracer des pistes aux contributions sur notre travail pour rendre plus efficace la simulation des architectures matérielles des systèmes temps réel.

Cet état de l'art se focalise sur les deux types de simulation que nous avons vus : l'ISS et la CAS. Nous verrons que les stratégies d'amélioration du temps d'exécution sont plus poussées pour les ISS. On retrouve donc cinq principales techniques que nous allons développer dans ce chapitre :

- la simulation interprétée, la plus lente des techniques ;
- la simulation compilée, plus rapide mais au prix de plusieurs restrictions ;
- la simulation compilée en *Just In Time*, qui se veut un compromis entre les deux précédentes ;
- la *Binary Translation*, que l'on peut implanter de façon statique ou dynamique ;
- l'échantillonnage, une approche basée sur des analyses statistiques.

2.1 Simulation interprétée

Le terme de simulation interprétée provient du vocabulaire de la programmation, où l'on trouve des langages interprétés (BASIC ou LISP, par exemple). Ces langages se distinguent des autres par l'absence d'une phase de compilation, ce que nous verrons plus en détail dans la section suivante. Les instructions sont directement interprétées et exécutées durant l'exécution.

Dans le cadre de la simulation fonctionnelle, le terme se transpose facilement. Un simulateur interprété se distingue par son schéma de développement, présenté sur la Figure 2.1. Un simulateur interprété simule l'exécution d'un code binaire en reproduisant les mêmes étapes que l'architecture simulée. Ainsi, pour chaque instruction le simulateur effectue les étapes suivantes :

- chargement, le code binaire est lu en mémoire à partir de la valeur courante du compteur programme (PC) ;
- décodage, le code binaire est décodé pour extraire l'instruction et ses opérandes ;
- exécution, l'instruction est exécutée.

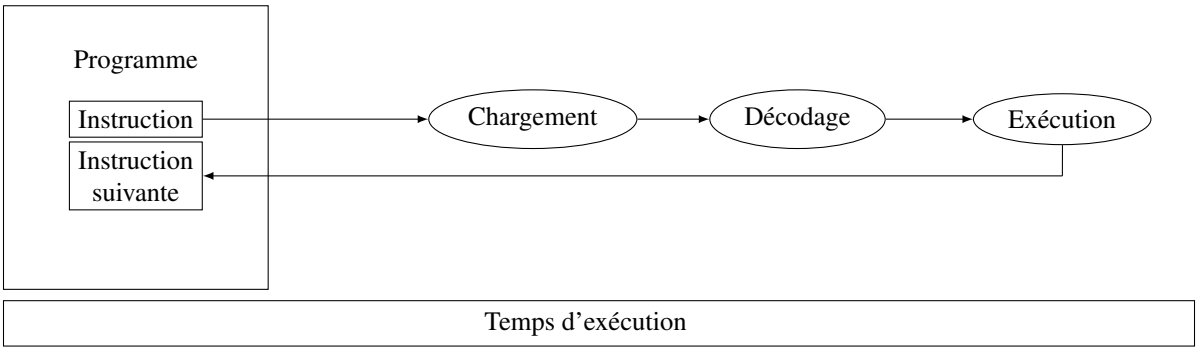

FIGURE 2.1 – La simulation interprétée charge, décode et exécute les instructions durant le temps d'exécution.

Une grande partie des simulateurs actuels utilise cette technique et plus particulièrement les simulateurs CAS. Notamment, c'est la technique utilisée par le simulateur associé à HARMLESS . Nous étudierons son fonctionnement plus en détail dans le chapitre 3.

Bien qu'elle s'avère relativement peu performante, du point de vue de la vitesse d'exécution, elle présente l'avantage d'offrir un développement flexible. Contrairement à des approches compilées, le simulateur obtenu n'est pas spécifique à la simulation d'un seul programme. Il permet de tous les simuler, sans avoir besoin de compiler un nouveau simulateur. Comme autre caractère de flexibilité, il peut simplement tenir compte de types de programmes variés, comme les branchements indirects, les codes récursifs, les programmes auto-modifiants.

La faible efficacité de ce type d'implantation de simulateurs s'explique par l'ensemble des tâches réalisées lors de l'exécution. Entre les opérations effectuées, c'est généralement la phase de décodage qui coûte le plus en termes de vitesse [BNH+02]. En effet, elle implique l'utilisation de nombreuses opérations bit-à-bit pour vérifier et extraire : l'instruction, ses opérandes et les modes d'exécution. Avec la propriété de localité d'un programme, ce type d'opérations devient, de plus, redondant. C'est-à-dire que les programmes possèdent comme caractéristique de tendre à revenir sur les mêmes instructions (typiquement, dans le cas des boucles), l'opération de décodage pour une même instruction est alors effectuée plusieurs fois. On parle alors de localité temporelle (qui inclut donc la réutilisation des instructions, mais aussi des données) et que l'on distingue de la localité spatiale (caractéristique voulant que les programmes utilisent des données et des instructions situés à proximité de celles accédées récemment).

Cette propriété de localité temporelle rend intéressante l'utilisation d'un cache d'instructions. Il permet la mémorisation des instructions décodées récemment, en considérant toutefois une politique de remplacement. De cette manière, lors du retour sur ces instructions, dans le cas d'une boucle par exemple, celles-ci peuvent directement être réutilisées sans réeffectuer les calculs du décodage. De même, le temps utilisé à l'allocation en mémoire des fonctions de l'instruction est capitalisé. Ce procédé permet ainsi d'alléger sensiblement l'exécution, comme cela a été utilisé dans le cadre de HARMLESS [KBB+09].

2.2 Simulation compilée

De même que pour la simulation interprétée, le terme de simulation compilée provient du langage de la programmation. Pour certains langages de programmation, une compilation est nécessaire pour exécuter le programme (comme C++ ou ADA). Lors de cette phase, le compilateur traduit le code source pour produire le code binaire exécutable.

Cependant, le terme ne se transpose pas tel quel dans la simulation. En effet, il y a dans celle-ci deux processus qu'il faut distinguer et que l'on peut compiler : la conception du simulateur et la simulation du programme. La conception d'un simulateur repose usuellement sur des langages de programmation compilés, réputés plus efficaces, et demande donc une compilation. Dans le cas d'un simulateur compilé, la simulation du programme passe également par une phase de compilation, mais celle-ci est intégrée à la compilation du simulateur. Ainsi, en une seule phase de compilation, le simulateur est conçu et certaines tâches de simulation du programme sont traitées.

Avec ce type de chaîne de développement, il est à noter que le simulateur ainsi conçu est propre à un programme donné : celui avec lequel la compilation a été effectuée. Pour en simuler un nouveau, il faut une autre compilation pour concevoir un autre simulateur. C'est ce qui rend la chaîne de développement moins flexible.

D'une façon générale, on peut décrire la compilation comme un procédé visant à traduire en amont le programme dans un langage intermédiaire, plus simple à exécuter. Dans le cas de la programmation, il s'agit de traduire en code machine. Dans le cas de la simulation, il s'agit d'une représentation du comportement des instructions, pour les exécuter plus simplement. C'est la raison pour laquelle elle se révèle une technique efficace de simulation. Comme elle déplace des tâches de la phase d'exécution à celle de compilation, la simulation compilée [MAF91] présente naturellement de meilleures performances durant l'exécution.

Il faut toutefois faire remarquer que ces tâches ne disparaissent pas de la phase de développement complète, puisqu'elles se retrouvent durant la compilation. Pareille configuration peut s'avérer utile pour certains types de développement et moins pour d'autres. En effet, s'il est nécessaire de compiler le simulateur pour chaque exécution, comme c'est le cas quand l'on souhaite simuler des programmes différents, alors le gain peut se trouver nul, voire plus mauvais. Dans le cas d'une phase de tests contenant plusieurs scénarios par programme, on retrouve un cas de figure d'une compilation pour plusieurs exécutions, ce qui rentabilise le coût de la compilation. Ainsi, pour ce qui est d'un développement où chaque programme est testé avec un ensemble de scénarios de manière à mettre en évidence les différentes utilisations du programme et obtenir des résultats statistiques consistants, la simulation compilée représente un grand intérêt.

Nous avons représenté le schéma de développement d'un simulateur compilé sur la Figure 2.2. On y voit la distinction entre les deux phases de compilation et d'exécution. La première coïncidant avec la compilation du simulateur.

Les tâches qui sont traitées durant la compilation sont variables selon les simulateurs. On retrouve généralement trois tâches se retrouvant durant la phase de compilation :

- Le décodage des instructions : interpréter le code opérationnel des instructions pour en extraire les informations sur son comportement de telle manière à ce qu'elle soit directement utilisable par le simulateur.
- Le séquençage des instructions : organiser la gestion des ressources matérielles par les instructions, ce qui est notamment utile en cas d'exécution des instructions dans le désordre.
- L'instanciation des instructions : générer le code exécutable des différentes opérations qui forment le comportement des instructions.

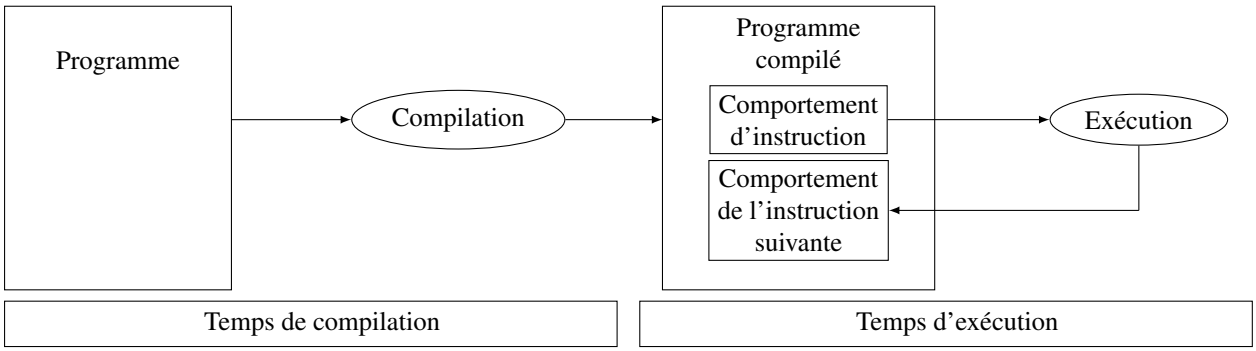

FIGURE 2.2 – La simulation compilée distingue deux phases : la compilation durant laquelle le programme est traduit dans un langage intermédiaire et l'exécution durant laquelle il suffit d'exécuter les instructions.

En fonction du nombre de tâches qui sont traitées durant la compilation, on peut parler de simulateur compilé à un plus ou moindre niveau. Comme le présente la Figure 2.3, on retrouve habituellement les degrés suivant pour une simulation compilée. Lorsque le séquençage des instructions est effectué à la compilation, on qualifie le simulateur d'ordonnancement dynamique. Si, en outre, l'instanciation des instructions est opérée à la compilation, on parle d'ordonnancement statique.

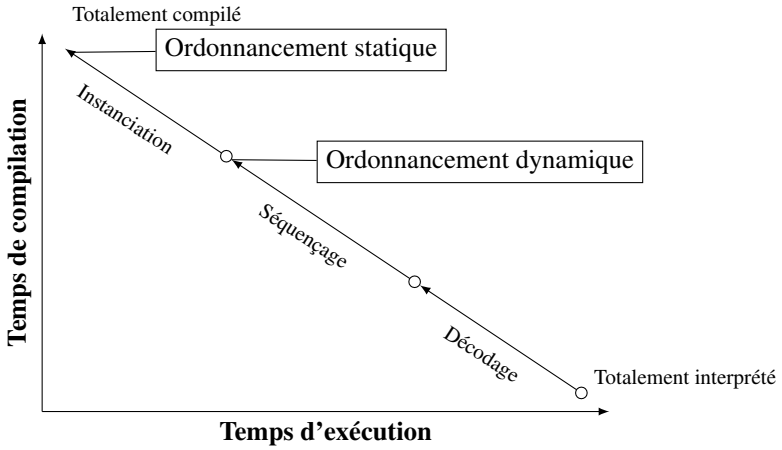

FIGURE 2.3 – Le niveau de compilation, d'après [PHM00]. On parle d'ordonnancement statique et dynamique pour les deux cas les plus compilés. Ces termes ne sont pas à confondre avec l'ordonnancement des tâches dans l'analyse d'ordonnançabilité des systèmes temps réel.

Si la simulation compilée présente une meilleure vitesse d'exécution, cela se fait au prix d'une perte de flexibilité. Comme nous l'avons dit, cette perte en flexibilité s'exprime d'abord par la spécificité du simulateur à un programme particulier. Du fait de cette propriété, on peut comprendre que la simulation de programmes auto-modifiants n'est pas envisageable. Cependant, ce genre de programmes ne se trouve pas dans le contexte de l'embarqué temps réel.

Le fait de traiter des tâches de simulation durant la compilation impose une autre contrainte : hors de l'exécution, certaines informations ne peuvent pas être extraites. Par exemple, sans exécuter un programme, il n'est pas possible de calculer l'évolution du contenu de la mémoire. On parle d'analyse statique dans la compilation, que l'on oppose à l'aspect dynamique de l'exécution. Du fait de cette restriction, il n'est pas possible de simuler certains codes, comme les

branchements indirects, par exemple (de façon générale, car nous verrons par la suite que certains branchements indirects sont traitables). On utilise le terme de branchement indirect lorsque la cible du branchement est calculée au cours de l'exécution. On l'oppose donc aux branchements directs, dont la cible est fixe (même si le branchement peut être conditionnel). Pour les branchements indirects, la cible se trouve dans un registre, pour les branchements directs, elle est codée dans l'instruction. Par conséquent, pour déterminer le flot de contrôle avec des branchements, il faut être en mesure de calculer le contenu d'un registre, ce qui ne peut se faire par une analyse statique, dans le cas général.

Comme exemple d'utilisation, un tel simulateur a été conçu à l'aide du langage LISA [PHM00]. Le niveau de compilation de ce simulateur correspond à un ordonnanceur dynamique, *i.e.* le décodage et le séquençage des instructions sont effectués durant la compilation. La contribution du papier présente une vitesse de simulation allant de 37 à 170 fois plus grande que les simulateurs commerciaux.

La simulation compilée est une technique propre à améliorer le temps de simulation de l'exécution des instructions. En ce sens, c'est une technique de simulation fonctionnelle : elle accélère le temps de calcul pour simuler le comportement des instructions. Dans la mesure où le fonctionnement d'un CAS peut reposer sur les services d'un ISS, typiquement pour calculer le flot de contrôle, on peut retrouver cette technique dans la simulation temporelle. Cependant, cela en fait une technique d'un moindre intérêt. En effet, elle ne permet alors de réduire que la part spécifiquement fonctionnelle du simulateur et pas le calcul des informations temporelles. Ainsi, du fait de la prédominance en temps de calcul de la manipulation du modèle du processeur pour obtenir les informations temporelles, si une partie des techniques utilisées par les ISS peuvent se retrouver dans les CAS, elles ne permettent généralement plus d'améliorer significativement le temps d'exécution.

2.3 Just In Time

La technique du *Just In Time* (JIT) (ou compilation à la volée) est souvent présentée comme un compromis entre la simulation interprétée et la simulation compilée. Elle combine, en effet, la flexibilité du premier et la vitesse du second. Le principe qui est mis en œuvre est d'effectuer les tâches de compilation lors de l'exécution et, pour rentabiliser cette opération, de mémoriser les résultats dans un cache. De cette manière, grâce à la caractéristique de localité temporelle d'un programme, lors d'un retour sur une partie du code, l'utilisation du cache permet de faire appel à un code déjà compilé, avec les gains d'efficacité que nous avons vus.

On trouve dans la littérature l'exemple du *Just-In-Time Cache-Compiled-Simulation* (JIT-CCS) [BNH+02], sur le HADL LISA. Braun *et al.* propose la technique de compiler les instructions durant l'exécution, et d'utiliser un cache pour en mémoriser les résultats. C'est ce processus que représente la Figure 2.4. Le JIT-CCS repose sur la simulation compilée avec le langage LISA, qui utilisait comme niveau de compilation celui d'un ordonnanceur dynamique. Par conséquent, les informations qui sont mémorisées dans le cache sont celles du décodage et du séquençage des instructions.

Un tel procédé, parce qu'il ne nécessite pas de phase en amont pour la compilation, maintient la souplesse d'utilisation d'un simulateur interprété, *i.e.* le simulateur conçu n'est pas spécifique à un seul programme. Tandis que l'utilisation du cache permet de se rapprocher des gains de vitesse de la simulation compilée. Les résultats de Braun *et al.* montrent qu'il est possible d'atteindre 95% des performances de la simulation compilée. Ce gain est toutefois à moduler en fonction de la taille du cache de simulation utilisé. Plus celui-ci est grand, plus la compilation à la volée est rentabilisée et plus les performances se rapprochent de la simulation compilée.

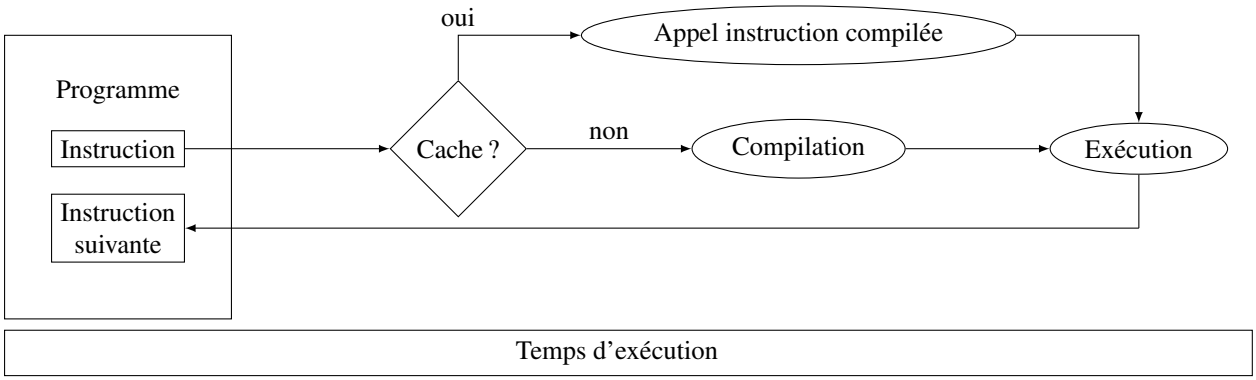

FIGURE 2.4 – Le *Just In Time* compile chaque instruction durant l'exécution. Il dispose d'un cache pour rentabiliser cette opération.

2.4 Binary Translation

Le principe de la *Binary Translation* (BT) [CM96], [Bel05] est de traduire le code à simuler directement dans le langage de la machine hôte, pour ensuite l'exécuter. Dans le cadre de la simulation, elle peut être comparée à une simulation compilée poussée à son terme. Pour rappel, nous disions que la simulation compilée traduisait le programme dans un langage intermédiaire plus simple à exécuter. La BT se passe d'un langage intermédiaire pour traduire le programme directement dans le code machine.

Cette traduction est des plus efficaces. En effet, les opérations de chargement, de décodage et d'exécution, s'avèrent coûteuses lorsqu'elles sont effectuées à un niveau logiciel par le simulateur lui-même, tandis qu'elles représentent un coût minime lorsque c'est l'architecture qui l'effectue au niveau matériel. C'est ce qui explique la grande efficacité de cette technique.

Dans la littérature, on trouve deux types de BT :

- *Static Binary Translation*, où la traduction du code se fait avant l'exécution
- *Dynamic Binary Translation*, où la traduction du code se fait durant l'exécution

On peut à nouveau faire une analogie avec les techniques vues précédemment : la traduction avant l'exécution correspondrait à la simulation compilée et la traduction durant l'exécution correspondrait à la simulation JIT.

Ces techniques présentent toutefois le défaut de nécessiter un grand travail d'adaptation à l'architecture hôte. Comme il est difficile d'effectuer ce travail pour toutes les architectures de manière générale, cela conduit ce types de simulateurs à n'être le plus souvent pas reciblables.

2.4.1 Static Binary Translation

La *Static Binary Translation* (SBT) [AS92] traduit le code dans le langage de la machine, durant une phase de compilation, d'où le terme de statique. On peut voir sur la Figure 2.5 un schéma de son fonctionnement.

Un compilateur permet d'analyser le programme pour en extraire tous les chemins possibles. Le simulateur généré reproduit le comportement du programme dans l'architecture simulé sur la machine hôte. On perd néanmoins et naturellement l'adéquation des temps de calcul, il ne s'agit alors que d'un ISS dont un des objectifs est l'efficacité.

Il peut arriver que des instructions ne puissent être reconnues lors de la phase de compilation, ou ne soient pas traduisibles. Pour pallier ce problème, certains SBT intègrent une dose de simulation interprétée dans laquelle l'exécution bascule pour traiter ces cas.

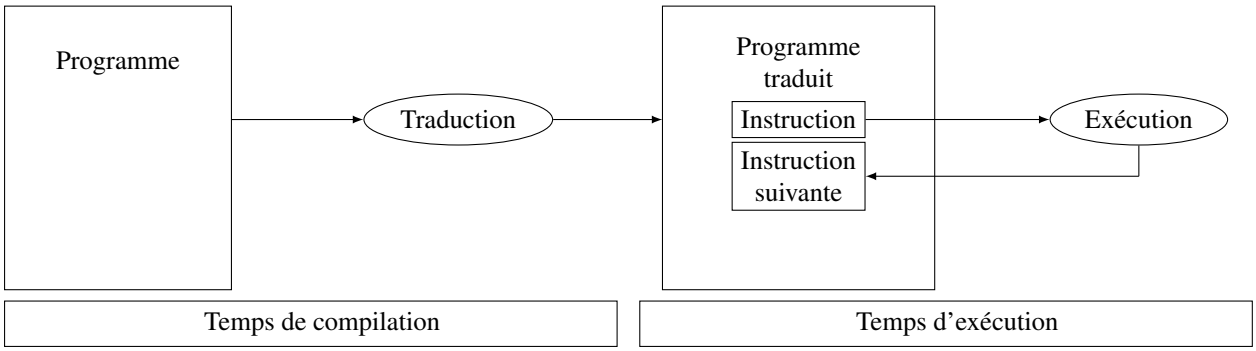

FIGURE 2.5 – La *Static Binary Translation* traduit le programme dans le langage de la machine hôte dans une phase de compilation.

C'est la technique la plus efficace dans les simulateurs fonctionnels. Son efficacité n'atteint toutefois pas celle d'une compilation native, *i.e.* une compilation du code source faite sur la machine hôte. Cela s'explique par le fait qu'un compilateur possède une vue une vue du code plus globale et à un niveau d'abstraction plus élevé, ce qui lui permet de meilleures optimisations.

De la même manière que la simulation compilée, puisqu'elle fait appel à une phase de compilation, elle présente l'inconvénient d'être peu flexible dans sa chaîne de développement. Le simulateur conçu est spécifique à un programme, une nouvelle compilation est nécessaire pour simuler un nouveau programme.

Cette technique présente un autre désavantage qui l'empêche d'être utilisée dans le cadre de la simulation temporelle. En effet, lorsque l'on traduit le programme du code machine de la cible à celui de l'hôte, on perd les spécificités de la machine cible ce qui modifie le comportement temporel. Il n'est plus possible d'observer la succession des états de l'architecture comme elle se retrouverait sur la machine cible. Compter le temps dans une telle approche est impossible.

2.4.2 Dynamic Binary Translation

La *Dynamic Binary Translation* (DBT), à la manière du JIT, traduit le code à la volée. Elle utilise également un cache pour rentabiliser cette opération. Elle couple de cette manière la simulation interprétée avec la SBT, comme le montre le schéma de la Figure 2.6.

Comme nous l'avons vu, dans le cas d'une boucle, le code déjà exploré est simplement exécuté depuis le cache, sans besoin de traduction. Des techniques d'optimisation plus poussées sont possibles pour les sections du code fréquemment appelées. Dans ces cas, la DBT peut inclure dans le cache des informations supplémentaires, comme le contenu des registres ou le flot de contrôle, dans le but d'appliquer à ce niveau des optimisations plus brutales. Ainsi, la cohérence de ces informations est préservée dans les premières exécutions, et les ayant mémorisées elles peuvent ne plus être respectées pour améliorer la vitesse de simulation. Ce type d'optimisations n'est, par exemple, pas possible dans le cadre d'une SBT.

Parmi les simulateurs utilisant cette technique, on peut citer SHADE [CK94] et le projet EMBRA [WR96].

La SBT traduit le programme suivant des blocs de code, semblables à des pas de simulation optimisés. Le simulateur SHADE enchaîne directement ces blocs sans passer par la boucle de simulation, du moment qu'aucun contrôle n'est nécessaire. Le simulateur vérifie l'appartenance

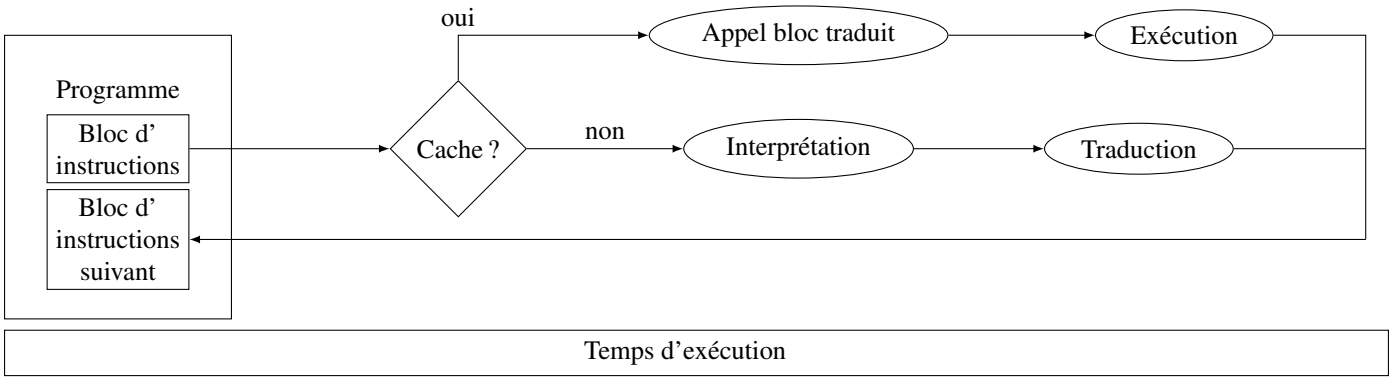

FIGURE 2.6 – La *Dynamic Binary Translation* traduit le programme dans le langage de la machine hôte à la volée.

des instructions à suivre dans les blocs présents dans le cache. S'il ne les trouve pas, il peut faire appel à des blocs fragmentés dans un cache spécifique ou construire lui-même un fragment de code à traduire.

le simulateur EMBRA fonctionne sur les mêmes bases. Il est en outre capable de simuler les codes auto-modifiants. Lorsqu'une telle opération est détectée, il vide simplement la partie du cache correspondante.

Plus les blocs peuvent être de taille importante et plus la vitesse de simulation augmente, en effet moins de contrôle est nécessaire. Des pistes d'optimisation de cette technique reposent sur l'amélioration de la taille des blocs traduits par la BT. Ces blocs définissent un pas dans la simulation Des techniques existent pour améliorer la taille des blocs traduits par la BT : *Large Translation Unit DBT* [JT09].

Par la flexibilité qui la distingue de la SBT, cette technique est utilisable par la simulation temporelle. En effet, puisqu'elle utilise en première approche les techniques de la simulation interprétée (qu'elle optimise ensuite lors d'un retour sur la section du code), on peut retrouver les informations nécessaires à la manipulation d'un modèle du processeur pour obtenir des résultats temporels. Elle constitue ainsi une bonne alternative à la simulation interprétée, et permet d'atteindre une vitesse d'exécution dix fois supérieure à celle-ci.

2.5 Échantillonnage

Une autre technique, spécifique à la simulation temporelle, est la *Cycle Approximate Simulation*. Elle repose sur la collecte d'informations temporelles sur des sous-ensembles du programme pour en déduire des propriétés sur l'ensemble général, de la même manière qu'une étude statistique dispose d'échantillons à interroger pour en déduire des propriétés sur l'ensemble de la population. Comme tout le programme n'est pas étudié sur sa temporalité, de grands gains de performance peuvent être obtenus.

Cette démarche n'est, cependant, pas exactement équivalente à un simulateur temporel précis au cycle près, à cause des marges d'erreur qu'elle induit.

Le simulateur SMARTS [WWFH03] utilise la *Cycle Approximate Simulation*. Il permet d'obtenir des temps d'exécution pouvant aller jusqu'à 60 fois plus courts qu'une simulateur précis au cycle près. Dans une première approche, la simulation est effectuée de façon fonctionnelle. Pour obtenir des informations temporelles, le simulateur sélectionne des sections particulières du code pour les simuler finement, et recueillir des données sur la temporalité de l'exécution. Ces sections sont nombreuses, mais de taille réduite. Les informations extraites

permettent, à l'aide d'une analyse statistique, de déterminer le profil général des propriétés temporelles de l'exécution.

Entre deux sections de simulation temporelle, l'état du modèle du processeur est gelé. Par conséquent, il peut arriver que cet état perde de sa cohérence entre deux échantillons. Un tel problème peut entraîner de grandes divergences dans les résultats. Pour lutter contre cela, une phase de mise à jour du modèle du processeur est effectuée avant la simulation temporelle de chaque échantillon. Cependant, il n'est pas simple d'évaluer la taille de ces phases de mise à jour pour approcher suffisamment la précision des simulateurs précis au cycle près.

2.6 Conclusion

Le développement de simulateurs d'architecture à grande vitesse d'exécution est largement étudié et présente de nombreuses techniques. Nous avons étudié dans ce chapitre les cinq principales. Parmi elles, nous insistons sur la simulation interprétée, utilisée par le langage HARMLESS que nous verrons dans le chapitre 3 et qui sert de base à nos travaux, ainsi que la simulation compilée qui représente la voie que prennent les contributions de cette thèse.

Mise à part la technique d'échantillonnage, il faut bien noter que celles présentées dans ce chapitre sont propres à améliorer la vitesse de simulation du comportement des instructions, autrement dit la simulation fonctionnelle. Comme nous l'avons dit, la simulation temporelle peut reposer sur les calculs de la simulation fonctionnelle, notamment pour obtenir le flot de contrôle du programme. De ce point de vue, les techniques étudiées dans ce chapitre sont seulement propres à réduire une partie du temps d'exécution total dans la simulation temporelle.

De plus, au vu des nombreuses contributions faites à l'optimisation des simulateurs fonctionnels, cette réduction devient de moins en moins significative sur l'ensemble. En effet, la part de manipulation du modèle de processeur, permettant d'obtenir les informations temporelles, est à présent prépondérante sur les autres tâches de simulation. Il en découle que les pistes de réduction du temps d'exécution en simulation temporelle précise au cycle près reposent davantage sur un travail sur le modèle de processeur utilisé, comme son optimisation et sa simplification.

3

HARMLESS

Dans ce chapitre, nous nous intéressons au HADL HARMLESS et plus particulièrement à l'outil de génération de simulateurs qui lui est associé. HARMLESS se distingue des HADL que l'on rencontre usuellement dans la littérature par le découpage de sa description en trois vues, mais aussi par l'utilisation qu'il fait d'un automate pour modéliser le fonctionnement du pipeline. Nous verrons en détail ce que représente ces deux avantages.

Comme cet outil sert de socle aux contributions proposées dans ce mémoire, les explications techniques qui sont données au cours de ce chapitre permettront de mieux comprendre les chapitres suivants. Nous détaillerons le fonctionnement de son modèle de pipeline, et la manière dont il est construit. Le chapitre se termine sur une analyse des performances du simulateur, en vue d'orienter les diverses réponses que l'on peut apporter pour son optimisation.

3.1 Le Langage de description d'architectures matérielles

Le HADL HARMLESS est né au sein de l'équipe Système Temps Réel de l'IRCCyN, au travers de la thèse de Rola Kassem [Kas10], intitulée *Langage de description d'architecture matérielle pour les systèmes temps réel*, et qui fut encadrée par Jean-Luc Béchennec, Mikaël Briday et Yvon Trinquet.

Cette thèse avait pour objectif de concevoir un langage de description d'architectures, qui s'orienterait vers la conception de simulateurs temporels précis au cycle près. Comme nous l'avions expliqué, pour construire un tel simulateur, il est nécessaire de disposer d'une description du jeu d'instructions et de l'architecture du processeur. C'est ce qui fait de HARMLESS un langage mixte et orienté simulation.

Le langage HARMLESS se distingue des langages existants notamment par l'utilisation de trois vues pour la description du jeu d'instructions [KBB+09] :

- la vue binaire, pour la description de l'op code, qui permet la conception d'un décodeur ;
- la vue comportementale, pour décrire les différentes opérations à effectuer lorsque l'instruction s'exécute ;
- la vue syntaxique, pour la description des instructions en assembleur, à des fins de débogage.

Ce découpage permet notamment de travailler avec des descriptions partielles (une seule vue sur les trois, par exemple). Cela offre aussi la possibilité de concevoir des architectures incomplètes. On parle de langage incrémental. Cet aspect modulaire est enfin utile si l'on souhaite reprendre des éléments de descriptions dans un autre projet similaire.

HARMLESS apporte en outre de vraies facilités dans la description des architectures. Parmi ses avantages, on peut citer :

- Une description concise. Il met à disposition différents opérateurs permettant de manipuler simplement des champs au bit près. Il donne des outils de vérifications pour gérer les champs utilisés selon une taille précise au bit près. Enfin, il est possible de mutualiser le code commun à plusieurs parties.

- Des descriptions du jeu d'instructions indépendantes de l'architecture. C'est ce qui permet de découper le travail de description entre d'une part celle du jeu d'instructions et d'autre part celle de la micro-architecture du processeur. Un tel découpage offre au langage de permettre séparément la conception d'un ISS (en écartant les parties de description de l'architecture) et d'un CAS (en utilisant la description de l'architecture pour construire le modèle temporel). Enfin, c'est un point important pour pouvoir mutualiser des descriptions au sein d'une famille de micro-architecture.

- Une vérification plus facile, grâce à deux choses : un typage fort, qui permet de s'assurer de la compatibilité des formats renseignés, et l'impossibilité d'inclure des blocs en langage C qui, bien que plus simple d'analyse syntaxique, limite la vérification du langage.

3.2 Modèle du pipeline simple

Le modèle du pipeline utilisé par HARMLESS en simulation interprétée sert de base aux contributions de la thèse. Nous en dressons ici une explication.

Dans une perspective de modélisation pour un simulateur temporel, un élément incontournable de la micro-architecture est le pipeline : il a une influence décisive sur le timing du processeur, et c'est l'élément le plus coûteux en temps de calcul. On trouve, en effet, cet élément sollicité à chaque cycle processeur, pour chaque étage de pipeline. Il génère des calculs complexes, à cause notamment de la gestion des aléas.

Un pipeline est un élément dans la micro-architecture du processeur qui permet une certaine parallélisation dans l'exécution des instructions et donc d'améliorer la vitesse de calcul. Un pipeline est composé de plusieurs étages, correspondant à l'utilisation d'une ressource matérielle (voir Figure 3.1). L'exécution d'une instruction se fait en traversant tous les étages du pipeline. Idéalement, chaque instruction dans chaque étage du pipeline évolue à l'étage suivant pour chaque cycle du processeur. Cependant cette évolution peut être perturbée par des aléas. Le fonctionnement de cet élément est détaillé dans l'annexe A.

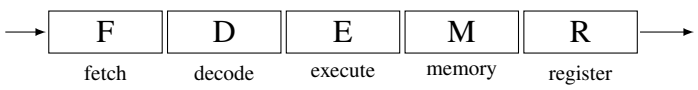

FIGURE 3.1 – Le type de pipeline utilisé dans la plupart des tests de ce mémoire. Il comporte cinq étages. F : l'instruction est chargée, D : l'instruction est décodée et les registres sont lus, E : l'instruction est exécutée, M : les appels mémoire sont faits et W : le résultat est écrit dans un registre.

On considère dans ce mémoire les pipelines séquentiels. Dans notre cas d'étude, il n'y a pas de pipelines travaillant en parallèle, ni de pipelines non-linéaires.

Dans [KBB+11], les auteurs utilisent le modèle des automates finis pour le modéliser, parce que le système peut être considéré comme un système de transitions discret. En effet, une transition est prise à chaque cycle. De plus, comme à chacune de ces transitions c'est tout le pipeline qui est actualisé, cette modélisation est préférable à une plus parcellaire, où chaque étage du pipeline serait géré indépendamment.

3.2.1 États du modèle

Sur la Figure 3.2, on considère l'évolution du pipeline dans le temps. Celle-ci est généralement décrite par un graphe présentant le temps en abcisse et les instructions en ordonnées. Sur la Figure, l'instruction i se trouve dans l'étage F au temps t-3, puis arrive dans l'étage W au temps t. De même, à l'instant t, trois instructions se trouvent dans le pipeline : i dans l'étage W, i+1 dans l'étage D et i+2 dans l'étage F. On peut remarquer l'influence d'un aléa par l'attente qu'il impose à l'instruction i+1 pour utiliser l'étage D, celle-ci décale la suite des instructions. On parle de bulle dans le pipeline, quand cette attente génère un étage vide.

Un état de l'automate associé correspond à l'état du pipeline à un cycle donné, *i.e.* le type d'instruction présent dans chaque étage du pipeline pour le cycle donné. Le modèle évolue d'un état à un autre en rendant compte des différents cas d'aléas rencontrés.

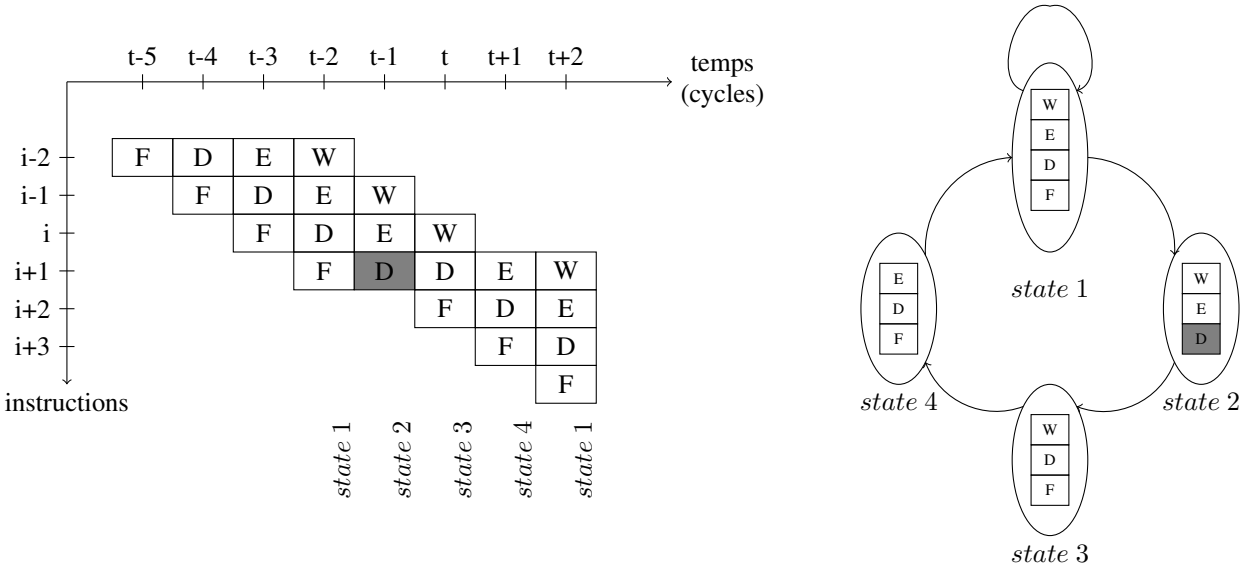

FIGURE 3.2 – Un état de l'automate représente l'état du pipeline à un instant donné. Sur la gauche, on trouve une représentation du pipeline et, sur la droite, l'automate utilisé pour le modéliser. Ici, le pipeline a 4 étages. F : l'instruction est chargée, D : l'instruction est décodée et les registres sont lus, E : l'instruction est exécutée, et W : le résultat est écrit dans un registre.

Pour modéliser les aléas (*i.e.* les bulles qui peuvent survenir dans l'évolution du pipeline), il faut pouvoir rajouter des contraintes sur l'évolution normale du pipeline. On modélise ces contraintes en utilisant ce que l'on appelle des ressources. On distingue alors deux types de ressources :

- Les *ressources internes*, qui peuvent être calculées de façon statique durant la construction de l'automate.

- Les *ressources externes*, qui ne peuvent être déterminées que durant l'exécution de façon dynamique.

Une ressource représente une contrainte de type binaire : soit elle est disponible, soit, au contraire, occupée. À l'aide de cette ressource, il est possible d'orienter l'évolution du pipeline.

Les ressources internes permettent notamment de modéliser la disponibilité ou non de ressources matérielles, répondant à la gestion des aléas structurels. Comme la résolution de ces aléas est statique, elle ne présente donc pas d'impact sur le temps d'exécution. Comme exemple de ces ressources, on peut citer simplement les étages du pipeline.

Les ressources externes peuvent modéliser les contraintes des ressources matérielles qui sont calculés à l'exécution, comme celle de la mémoire. Par exemple, le modèle du cache d'instructions interagit avec le modèle de l'architecture par l'intermédiaire d'une ressource externe. Elles sont aussi très utiles pour répondre au problème de la gestion des aléas de données. En effet, comme le modèle utilise des classes d'instructions qui ne contiennent pas d'informations sur les registres utilisés, il n'est pas possible d'effectuer cette résolution statiquement. C'est un point qui sera abordé dans le chapitre 4.

Un état du modèle est donc défini non seulement par l'état du pipeline, mais aussi par les ressources internes. On retrouve donc deux informations dans les états :

- quelle classe d'instructions se trouve dans chaque étage du pipeline ;
- l'état de disponibilité des ressources internes.

Dans le cadre de la simulation interprétée, il est seulement utile de distinguer dans le pipeline les instructions de comportement différent. Les instructions qui utilisent les mêmes ressources dans le même étage du pipeline sont regroupées pour former des *classes d'instructions*. Ainsi, par exemple, les instructions de calculs algorithmiques qui lisent deux registres, effectuent un calcul, et écrivent leur résultat dans un troisième registre appartiennent à la même classe d'instructions.

3.2.2 Transitions du modèle

Dans ce modèle, une transition représente un évènement discret qui amène le système d'un état à un autre. Il est susceptible d'évoluer à chaque cycle processeur. On trouve donc une transition par cycle. Celle-ci est déterminée par deux choses :

- l'état de disponibilité des ressources externes
- et la classe d'instructions qui se présente à l'entrée du pipeline.

Au cours de l'exécution, le noyau du simulateur effectue les calculs autour des ressources externes (comme déterminer l'état du cache d'instructions ou la dépendance des données). Sur cette base, il renseigne les ressources externes, qui à leur tour font évoluer le modèle du pipeline, pour faire apparaître des bulles si un aléa se présente.

Dans la perspective de pouvoir observer des évènements particuliers au cours de la simulation, on utilise ce que l'on appelle une *notification*. De même que pour les ressources, ce champ d'information est binaire : il peut être vrai pour désigner un évènement survenant, ou faux si l'évènement ne survient pas. Les notifications rendent, par exemple, possible le compte des instructions exécutées : la notification est alors associée à l'évènement de sortie d'une instruction du pipeline. Les transitions sont étiquetées avec les notifications. De cette manière, lors de l'exécution du modèle, on peut suivre l'apparition des évènements suivis.

Ces informations sont toutefois bien présentes dans les états, et observables de ce point de vue. Mais, elle sont rendues par ce système plus accessibles, ce qui améliore le temps d'exécution.

3.2.3 Formalisation

Nous pouvons maintenant formaliser le modèle interprété. Soit AI un automate défini par $\{S, s_0, RE, CI, N, T\}$, où :

- S est l'ensemble des états ;
- s_0 est l'état initial (pipeline vide) dans S ;
- RE est le premier alphabet d'actions (ressources externes) ;
- CI est le second alphabet d'actions (classes d'instructions) ;
- N est l'alphabet des étiquettes (notifications) ;
- T est la fonction de transitions dans $S \times RE \times CI \times N \times S$.

Si $(s, re, ci, n, s') \in T$, on note $s \xrightarrow{re:ci(n)} s'$. On fait remarquer que l'élément entre parenthèses n'est qu'une étiquette, tandis que les autres sont des conditions de franchissement de la transition. Dans la représentation que nous utiliserons au cours de ce mémoire, les ressources externes et les notifications seront décrites par une notation binaire. Chaque ressource externe (respectivement notification) est représentée sous la forme d'un bit, dont la valeur répond à la question : est-ce que la ressource est prise ? (respectivement : est-ce que l'évènement survient ?).

- Si le bit est à 0 : la ressource n'est pas prise (respectivement, l'évènement ne survient pas).
- Si le bit est à 1 : la ressource est prise (respectivement, l'évènement survient).

Les ressources (respectivement notifications) sont ensuite écrites du poids faible vers le poids fort. Ainsi, `00111` signifie que les trois premières ressources sont prises et les deux dernières libres.

Dans l'exemple de la Figure 3.3, nous avons seulement une notification qui représente *l'entrée d'une instruction dans le second étage du pipeline*. Elle vaut ainsi 1 au passage de l'état ② à l'état ③. Il y a une ressource externe. L'instruction de classe `b` nécessite de prendre la ressource externe pour rentrer dans le pipeline. Si la ressource externe vaut 1, cela signifie qu'elle est occupée : l'instruction est bloquée. C'est ce qui arrive pour l'état ⑤. En revanche, la classe d'instructions `a` n'ayant pas cette limitation, elle peut entrer dans le pipeline à l'état ④.

Durant la simulation, les états des ressources externes et la classe d'instruction de la prochaine instruction qui rentrera dans le pipeline sont requis pour déterminer le prochain état de l'automate. Quand une transition est prise, les notifications attachées à la transition sont transmises au moteur de simulation pour interagir avec le reste de la microarchitecture. Ces opérations sont réitérées en boucle autant de temps que défini par l'utilisateur.

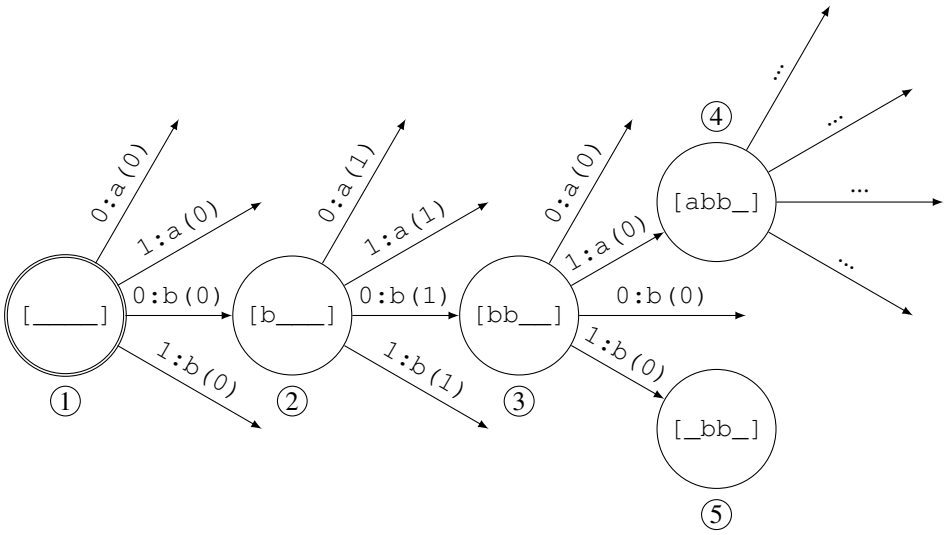

FIGURE 3.3 – Automate en simulation interprétée. `0:b(1)` signifie que la ressource externe est libre (`0`), qu'une instruction de classe `b` peut entrer dans le pipeline et que la notification peut arriver (`1`). On note entre crochets l'état du pipeline : les classes d'instructions sont rapportées dans l'ordre des étages (un tiret bas représente une absence d'instructions). Ainsi `[b____]` signifie que nous avons un pipeline à 4 étages, et que seule l'instruction de classe `b` se trouve dans le pipeline, au premier étage.

3.2.4 Estimation théorique de la taille du modèle

Le modèle en simulation interprétée regroupe l'ensemble des états possibles du pipeline. Il est aisé de faire une estimation dans ce cas de la taille du modèle.

Si le nombre de classes d'instructions est ci, pour chaque étage du pipeline il existe $ci + 1$ possibilités (pour inclure le cas supplémentaire d'un étage vide). Aussi, si le nombre d'étages est e, on peut affirmer que le nombre d'états différents possibles, et donc une estimation de la taille du modèle, est :

$$(ci + 1)^e \tag{3.1}$$

En réalité, à cause des contraintes que peuvent modéliser les ressources internes, le modèle peut être plus petit : des cas de figure du pipeline peuvent ne pas apparaître dans la construction de l'automate.

Par exemple, si le jeu d'instructions regroupe seulement trois classes d'instructions, et si le pipeline contient cinq étages, on peut borner la taille du modèle par 1 024 états.

3.2.5 Construction du modèle

La construction du modèle se base sur l'algorithme de parcours en profondeur de graphe.

L'algorithme 1 de construction du modèle manipule deux listes pour stocker les états : un pour les états à traiter, un autre pour les états traités. Le traitement des états consiste à calculer leurs successeurs. D'un état est calculé autant de successeurs qu'il y a de disponibilités possibles des ressources externes et qu'il y a de classes d'instructions qui puissent rentrer dans le pipeline.

Sur l'exemple de la Figure 3.4, le modèle dispose de deux ressources externes (sur la flèche en pointillés) et de quatre classes d'instructions (flèche en trait plein). En combinant les deux,

on construit tous les cas de figure.

Si le successeur créé n'a jamais été traité, il est ajouté à la liste des états à traiter. Le programme tourne jusqu'à ce qu'il n'existe plus d'états à traiter.

Algorithme 1 Construction du modèle interprété

```
1: Initialisation
2: while la liste des états à traiter n'est pas vide do
3:     s ← extraction(liste des états à traiter)
4:     for all état de disponibilité des ressources externes do
5:         for all classe d'instructions do
6:             Calcul de s', le successeur de s en fonction des ressources externes et de la classe
   d'instructions
7:             if s' n'appartient ni à la liste des états traités, ni à la liste des états à traiter then
8:                 Ajout de s' à la liste des états à traiter
9:             end if
10:        end for
11:    end for
12:    if s n'appartient pas à la liste des états traités then
13:        Ajout de s à la liste des états traités
14:    end if
15: end while
```

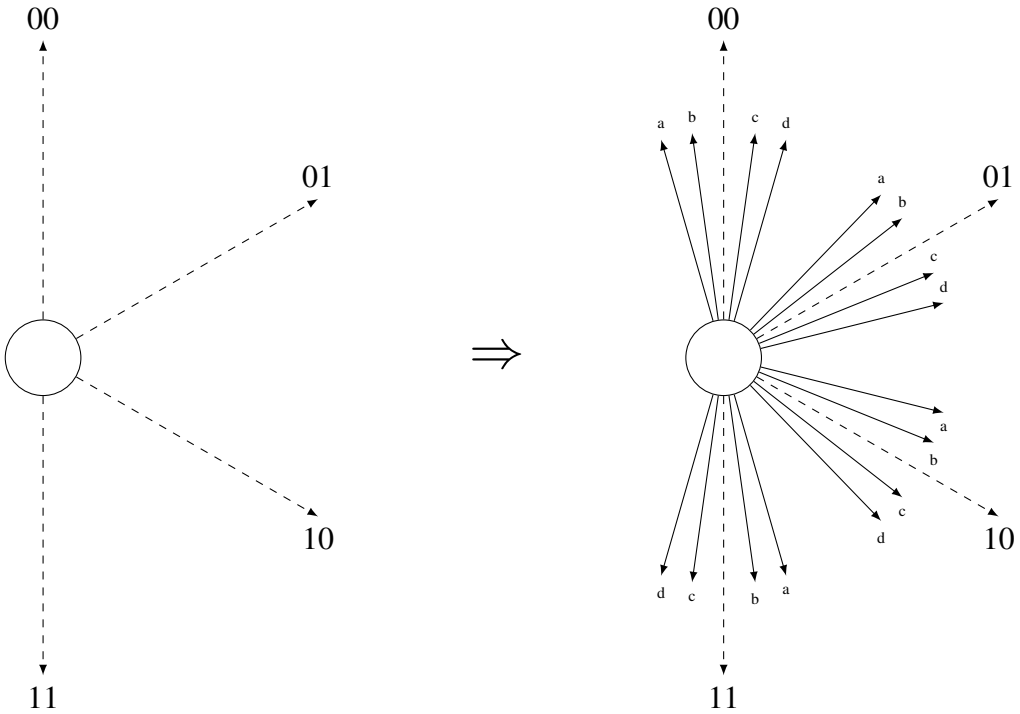

FIGURE 3.4 – La construction de l'automate : pour chaque état, l'on considère toutes les disponibilités possibles des ressources externes (00,01,10,11), et dans une deuxième étape, l'on considère pour chacune d'elles les classes d'instructions possibles (a,b,c,d). Dans chaque cas, on construit le successeur.

Le calcul du successeur se fait en remontant le pipeline du dernier jusqu'au premier étage, comme on peut le voir sur la Figure 3.5. Pour chaque étage, la disponibilité des ressources est

évaluée. Si elles le permettent, l'instruction est retirée de son étage, libérant la ressource interne associée, et ajoutée à l'étage suivant, occupant la ressource interne associée.

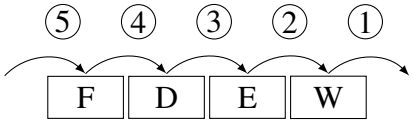

FIGURE 3.5 – Dans le calcul du successeur, l'évolution du pipeline. Partant du dernier étage, on fait avancer les instructions. Si au cours de ce processus, une ressource empêche cette évolution, l'instruction reste dans son étage, bloquant éventuellement les instructions en amont.

3.3 Le Simulateur

Les contributions de ce mémoire reposent sur le modèle, mais aussi l'implantation qui est faite du simulateur de HARMLESS. C'est pourquoi nous allons détailler le fonctionnement de cet outil. On pourra aussi se reporter à l'article [KBB+08] pour plus d'explications.

Le langage de description d'architectures HARMLESS permet la construction de simulateurs, par l'intermédiaire d'un compilateur (*gadl*[1]). C'est ce que l'on peut observer sur la Figure 3.6. Ce compilateur rend possible la construction d'un simulateur uniquement fonctionnel (ISS), et celle d'un simulateur dit mixte, *i.e.* prenant en charge l'aspect comportemental (ISS) et temporel (CAS).

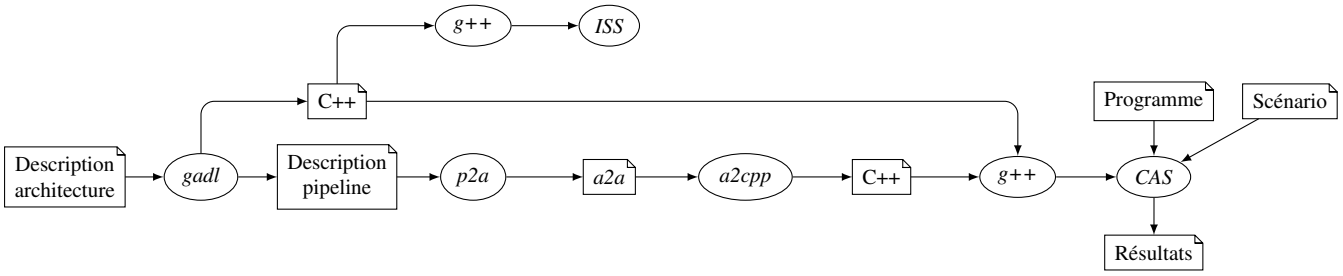

FIGURE 3.6 – Chaîne de développement de l'outil HARMLESS pour générer l'ISS et le CAS.

Pour ce deuxième simulateur, le modèle est établi par le programme *p2a* sur la base de la description du pipeline. Le modèle généré se trouve sous la forme d'un automate (*a2a*). Le programme *a2cpp* traduit simplement cet automate en code compilable. Pour obtenir un simulateur mixte, le code du modèle temporel et du simulateur fonctionnel sont utilisés.

Une fois constitué, le simulateur interprète le programme et, suivant un scénario, délivre les résultats temporels demandés. De par le traitement qu'il effectue sur le programme, on parle de simulation interprétée, comme cela a été vu dans l'état de l'art.

3.3.1 Exécution du simulateur

Pour pouvoir analyser les avantages en terme de temps d'exécution, il faut comprendre le fonctionnement de l'exécution du simulateur. Celui-ci est présenté sur l'algorithme 2. Il correspond à l'exécution d'un cycle, *i.e.* le franchissement d'une seule transition du modèle.

[1] *gadl* provient de la contraction de *galgas* et HADL. *galgas* est un générateur de compilateur conçu par Pierre Molinaro.

À chaque cycle, le noyau de simulation calcule les conditions (disponibilités des ressources externes, classe de la prochaine instruction dans le programme). Connaissant l'état courant et les conditions de la transition, le noyau de simulation récupère deux informations : les étiquettes de la transition (notifications) et l'état successeur. À l'aide des notifications, il est capable d'effectuer différents calculs nécessaires à la simulation, dont la gestion des aléas de données.

C'est également à l'aide d'une notification, qu'il peut détecter l'entrée d'une instruction dans le pipeline. Dans ce cas, une instruction est exécutée, et la suivante est décodée.

Algorithme 2 Boucle de simulation

```
1: Calcul des ressources externes
2: Recherche de l'état successeur
3: if entrée d'une instruction dans le pipeline then
4:     Exécution d'une instruction
5:     Décodage de la prochaine instruction
6: end if
7: Gestion des aléas de données
```

La gestion des aléas de données mérite de plus amples explications. Lors de l'exécution, l'état du système des registres est simulé par une file, contenant les registres « occupés », c'est-à-dire les registres dans lesquels une écriture est prévue. Les instructions qui suivent doivent attendre que l'écriture se fasse pour lire le registre. Lorsqu'une instruction arrive dans l'étage du pipeline ou elle peut lire les registres, le noyau du simulateur s'assure que ceux qu'elle doit consulter ne sont pas occupés. Si c'est le cas, pour modéliser la bulle de pipeline qui doit apparaître, le noyau du simulateur par une ressource externe bloque l'évolution de l'instruction.

Si ce n'est pas le cas, la ressource externe permet le passage de l'instruction. Les registres dans lesquels doit écrire l'instruction sont alors extraits et stockés dans la file. De cette manière, le noyau du simulateur a accès à cette information. Lorsque l'instruction quitte l'étage d'écriture dans les registres, la file perd l'élément correspondant. Le registre est à nouveau disponible pour être lu.

Cette organisation demande de gérer plusieurs structures de données à chaque cycle de la simulation. C'est ce qui en fait une tâche très consommatrice en temps de calcul.

3.3.2 Analyse des performances du simulateur

Pour comprendre quelles voies doivent emprunter des contributions visant à améliorer le temps d'exécution de la simulation, il est utile de s'intéresser au détail des performances que donne le simulateur interprété.

Le temps d'exécution a été analysé pour en extraire le poids des différentes tâches principales qui le composent. Nous comptons ainsi trois parties, correspondant à :

- la simulation fonctionnelle, en exécutant seule l'ISS ;
- la gestion des aléas de données, en retirant les opérations la calculant (comme les aléas de données ne sont plus respectés, le nombre de cycles de l'exécution n'est cependant plus le même) ;
- la gestion de l'automate, en considérant le temps de calcul total moins le temps des opérations précédentes.

Pour obtenir les résultats de ces analyses, nous avons simulé une architecture semblable au PowerPC 5516 de Freescale, avec un cœur *e200z1*. Le pipeline a été redimensionné de 4

étages à 5 étages de manière à augmenter la taille du modèle. Il est utilisé trois ressources externes : une pour le cache d'instruction, une pour la dépendance des données et une dernière pour la gestion de la mémoire. Les tests ont été exécutés avec les benchmarks de Mälardalen [GBEL10]. Les simulations ont été effectuées avec un Intel *Core 17@3,4GHz*. Pour faciliter la mesure des temps de calcul, chaque programme est exécuté 50 000 fois.

On peut ainsi, en moyenne, avancer les résultats présentés sur la Figure 3.7 :

- Le temps consacré aux résultats fonctionnels représente 33,4% du temps d'exécution total.
- Le temps consacré à la gestion des aléas de données représente 45,1%.
- Le temps consacré à la gestion de l'automate représente 21,5%.

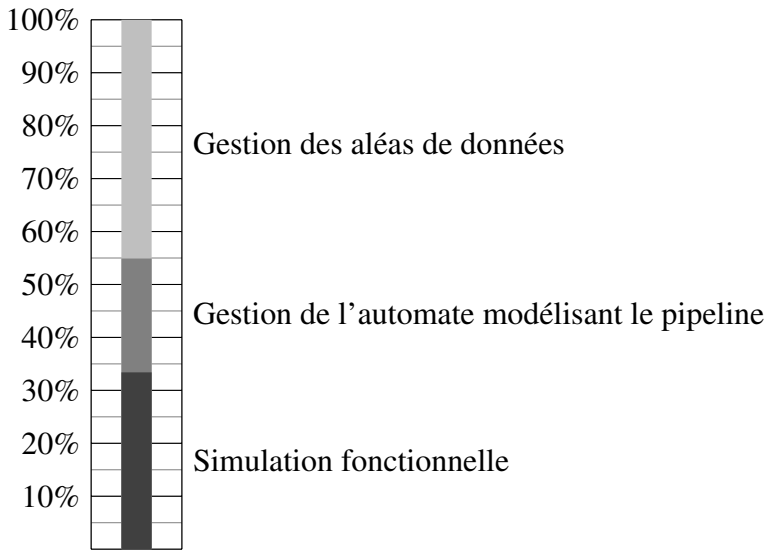

FIGURE 3.7 – Répartition moyenne des tâches dans le temps d'exécution du simulateur CAS

Une telle analyse ouvre les réflexions vers les stratégies à adopter dans le cadre d'un travail sur la réduction du temps d'exécution. On note de cette manière que la gestion des aléas de données est la plus consommatrice lors de l'exécution. Dans une visée d'efficacité, c'est la tâche en premier lieu à optimiser. Cela sera fait à l'aide de la simulation compilée.

Les différentes contributions qui suivent dans ce mémoire explorent les manières de réduire le temps d'exécution de chacune de ces trois tâches.

4

Simulation compilée — 1er modèle

4.1 Objectifs

Si la simulation compilée a fait ses preuves comme technique efficace d'amélioration du temps d'exécutions des simulateurs ISS, dans la limite de nos connaissances, elle n'a pas été traduite pour servir aux simulateurs CAS. En le faisant, l'objectif de notre première contribution est d'ouvrir une porte à des techniques de simulation capables d'améliorer les performances de la simulation temporelle. Les contributions qui suivront, dans ce mémoire, s'appuient sur les différentes possibilités qu'elle offre.

Dans l'analyse que nous faisons du temps d'exécution de la simulation interprétée, nous pouvons distinguer trois parties :

- une partie spécifique à la simulation du pipeline grâce à l'automate ;
- une partie spécifique à la résolution dynamique des aléas de données ;
- une partie spécifique à la simulation fonctionnelle.

Ces différentes tâches, en moyenne, sont réparties comme cela est représenté par la Figure 4.1.

Nous avons noté que la partie de résolution des aléas de données occupe près de la moitié du temps d'exécution (45%). Il apparaît ainsi légitime, dans une démarche de réduction du temps d'exécution, de se focaliser sur cette première tâche.

Pour ce faire, une idée est avancée : transposer les tâches associées à cette résolution dans le temps alloué à la phase de compilation. C'est cette même idée qui explique pourquoi, dans la simulation compilée, le temps d'exécution est réduit. Comme la compilation n'est effectuée qu'une seule fois dans le cycle de développement, un gain en ressort, voir Figure 4.2.

Dans le chapitre 2, nous avons vu ce qui fait la particularité de la simulation compilée et sous quelle forme nous la retrouvons dans la littérature. Nous avons souligné en quoi il s'agit d'une technique propre à la simulation fonctionnelle. En effet, elle optimise des tâches qui sont associées à l'exécution des instructions : décodage, séquençage ou instanciation.

Dans notre approche focalisée sur les aspects temporels, les tâches allouées dans la phase de compilation se trouvent être des tâches spécifiques au modèle du processeur. Les aléas de

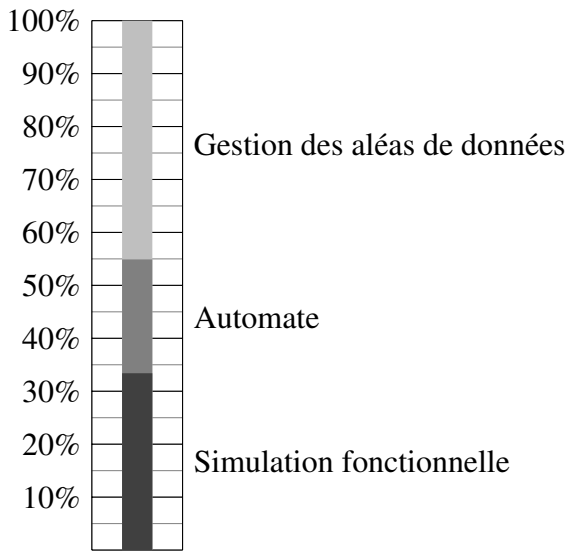

FIGURE 4.1 – Répartition moyenne des tâches dans le temps d'exécution du simulateur CAS. La simulation fonctionnelle occupe 33,4% du temps d'exécution, la gestion de l'automate 21,5% et la gestion de la dépendance des données 45,1%.

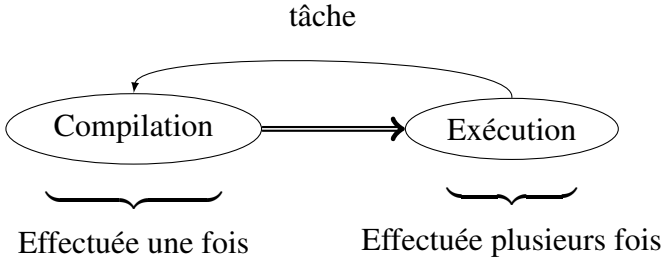

FIGURE 4.2 – Intérêt de la simulation compilée : des tâches sont déplacées de la phase d'exécution à celle de compilation.

données sont des évènements internes au fonctionnement du pipeline et donc appartiennent au modèle du processeur.

Cette contribution amène au développement d'un outil de simulation que nous avons baptisé ComCAS, pour Compiled CAS.

4.2 Modélisation

Pour effectuer l'analyse des aléas de données, ou autrement dit des dépendances de données, dans la phase de compilation, il faut deux choses :

- d'une part, disposer des informations nécessaires à ce moment de la chaîne de développement, *i.e.* pour que le programme puisse être analysé durant la compilation, il faut le fournir en entrée du compilateur ;
- et, d'autre part, se doter des structures qui permettent d'effectuer ces calculs lors de la construction du modèle, *i.e.* l'état du modèle doit être plus précis et connaître quelle instruction est lue.

4.2.1 Chaîne de développement

La mise en œuvre de la simulation compilée passe par une modification majeure de la chaîne de développement. En effet, pour que l'analyse des dépendances de données puissent se faire durant la compilation, il est nécessaire de disposer, à ce niveau, du programme. C'est la raison pour laquelle le programme est placé en entrée du compilateur dans le processus, non plus lors de l'exécution, mais lors de la compilation. On obtiendrait ainsi, de façon schématique, la chaîne de développement de la Figure 4.3.

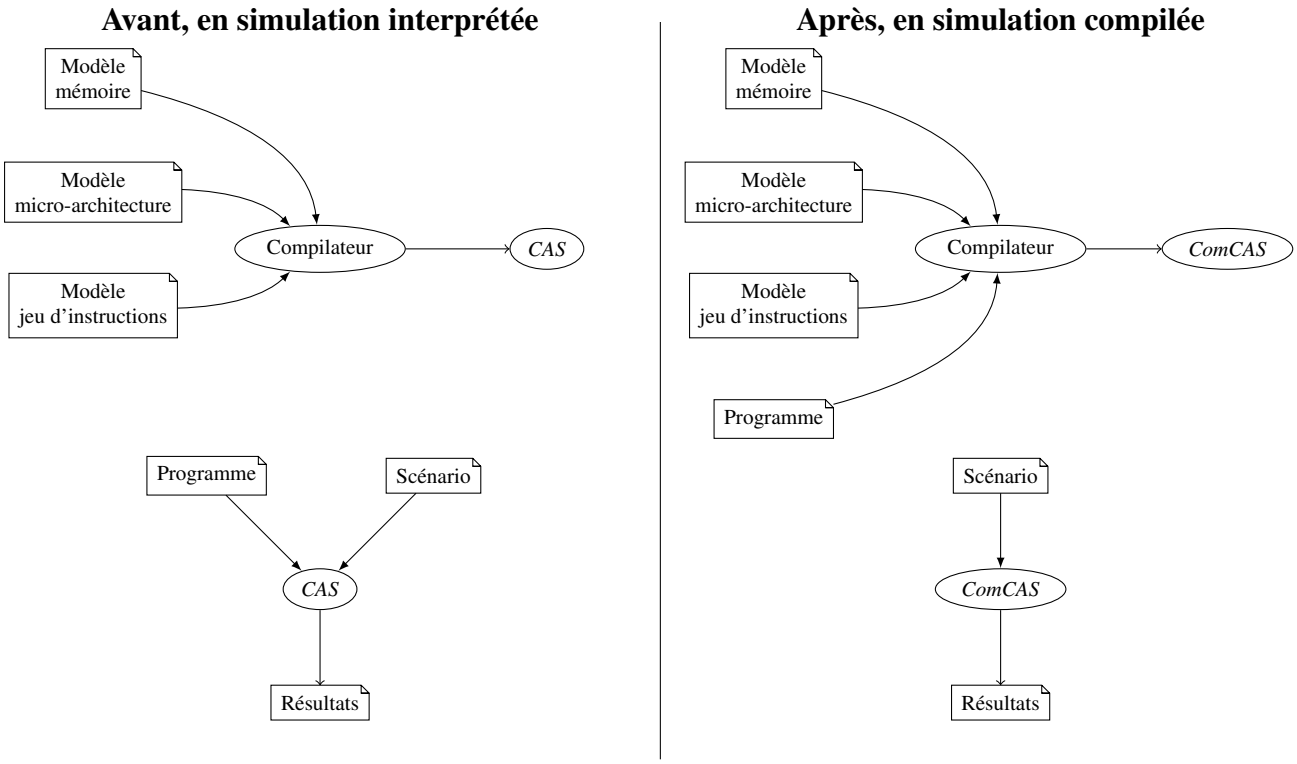

FIGURE 4.3 – Le développement de ComCAS nécessite de déplacer l'analyse du programme dans la compilation. Sur la partie gauche, en simulation interprétée, le programme est analysé durant l'exécution par le simulateur. Sur la partie droite, en simulation compilée, le programme est analysé durant la compilation.

Il faut préciser que ce schéma s'avance sur un développement complet du simulateur, où le programme ne serait plus nécessaire à l'exécution, car le processus serait totalement compilé. Ce n'est pas encore le cas. Il a toutefois l'avantage de mieux montrer le mouvement général du travail. En l'état actuel de l'implantation qui a été faite, le programme se situe en entrée du compilateur et de l'exécutable. En entrée du compilateur pour analyser les dépendances de données dans le code, et en entrée de l'exécutable pour exécuter les instructions lors de la simulation.

4.2.2 Modèle

Le modèle du processeur dans la simulation interprétée, parce qu'il n'inclut aucune information sur le programme à simuler, ne permet pas d'effectuer les calculs nécessaires à la gestion des dépendances de données. C'est le rôle de cette section de montrer ce qui doit être modifié dans le modèle pour le permettre.

Comme expliqué dans l'annexe A, les cas de figure EAE[1] et EAL[2] ne sont pas rencontrés, car nous nous concentrons seulement sur les pipelines linéaires. Seuls les aléas LAE[3], que l'on retrouve sur ce type de pipelines, seront donc modélisés. Pour tenir compte de cet aléa, il faut pouvoir disposer des registres lus et écrits par les instructions qui transitent dans le pipeline. La modélisation de l'état du pipeline par les classes d'instructions qu'elle contient n'est pas suffisante pour cela.

Il ne peut être envisageable de modéliser tous les agencements possibles d'utilisation des registres dans le pipeline, comme l'amènerait la simulation interprétée, au risque de faire exploser la taille du modèle. Pour cette raison, il est nécessaire que seuls les agencements effectivement rencontrés dans l'exécution du programme à simuler soient modélisés. Cela implique pour le modèle qu'il soit spécifique à un programme et qu'il inclut plusieurs informations sur le programme à simuler. C'est la raison pour laquelle l'adresse de l'instruction lue (le *Program Counter* ou PC) est ajoutée au modèle, mais également l'état du pipeline n'est plus simplement renseigné par des classes d'instructions mais directement par les adresses de ces instructions.

La connaissance du PC de l'instruction lue n'est cependant pas suffisante pour déterminer l'état de lecture du programme, il peut s'avérer nécessaire d'inclure également des informations sur le passé de la lecture du programme. C'est pour cela que l'état de la pile des appels est incluse dans le modèle. L'intérêt de ce choix est précisé dans la sous-section 4.2.4. En effet, nous verrons que cela permet de résoudre le problème des branchements indirects dans les retours de fonction.

On obtient ainsi un modèle, dont les états contiennent les éléments suivants :

- quelle instruction (quel PC) se trouve dans chaque étage du pipeline ;
- l'état des ressources internes ;
- le PC de l'instruction lue ;
- l'état de la pile des appels.

Il est à noter que le premier et le troisième point ne sont pas totalement redondants. S'il est possible de déterminer quelles instructions se trouvent dans le pipeline, on peut effectivement en déduire quelle instruction est actuellement lue. Cependant, il est possible que le pipeline se vide, et, dans ces cas là, cette information supplémentaire est nécessaire.

Comme le modèle évolue de façon discrète, à chaque cycle, il est possible d'avoir recours aux automates finis dans la modélisation. À chaque cycle, le modèle évolue d'un état à un autre en fonction de l'état de disponibilité des ressources externes. C'est la seule condition que l'on trouvera pour nos transitions. En effet, comme le programme à simuler est fixé, il n'est pas utile de conditionner l'évolution de l'automate sur les classes d'instructions qui entrent dans le pipeline, comme c'est le cas dans la simulation interprétée.

Cela a une certaine influence sur la taille du modèle comme on peut le voir sur la Figure 4.4. L'exemple utilise deux ressources externes (représentées en binaire) et quatre classes d'instructions (représentées par des caractères alphabétiques). Pour la simulation interprétée, chaque état a donc seize transitions sortantes, tandis qu'on n'en compte que quatre en simulation compilée.

Pour interagir avec le simulateur, nous utilisons le même principe de notifications que pour la simulation interprétée.

Nous pouvons maintenant formaliser notre modèle ComCAS comme suit. Soit AC un automate défini par $\{S, s_0, RE, N, T\}$, où :

[1] EAE pour Écriture Après Écriture : deux écritures sur un même registre sont inversées.

[2] EAL pour Écriture Après Lecture : une écriture écrase une donnée dans un registre qui devait être lu.

[3] LAE pour Lecture Après Écriture : une donnée est lue dans un registre avant qu'elle ait été écrite.

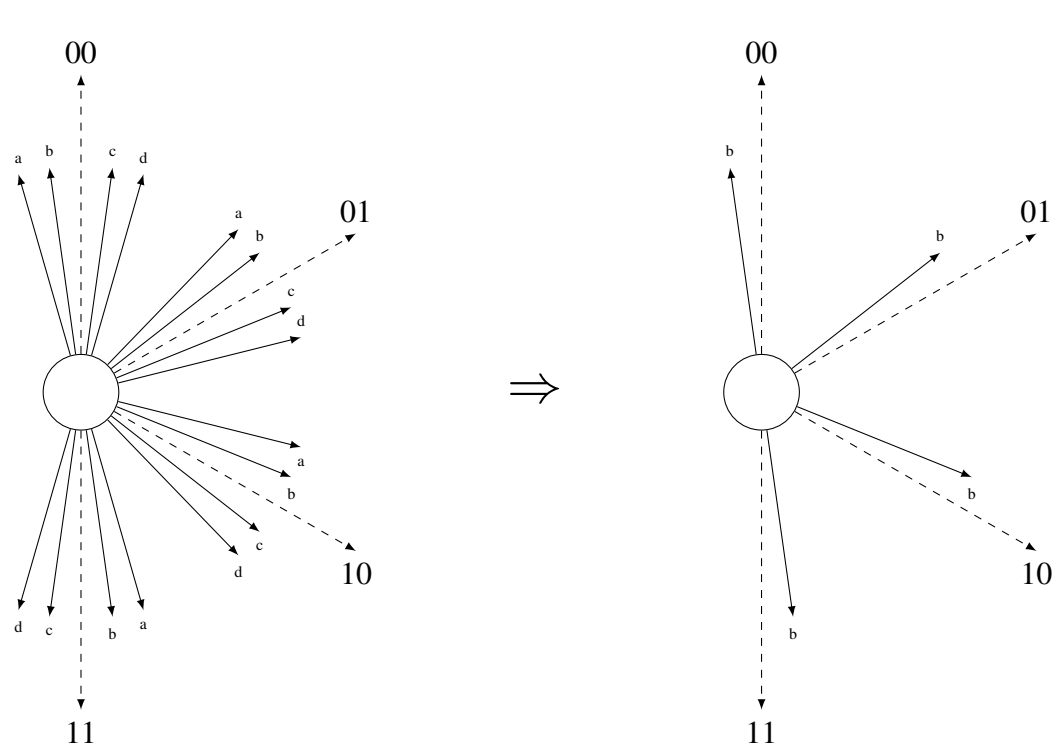

FIGURE 4.4 – La construction de l'automate par rapport à la simulation interprétée. En simulation interprétée, il faut prévoir l'entrée possible de n'importe quelle classe d'instructions. En simulation compilée, comme l'on connaît exactement l'instruction suivante, un seul cas est nécessaire.

- S est l'ensemble des états ;
- s_0 est l'état initial (pipeline vide, PC initial, pile vide) dans S ;
- RE est l'alphabet des actions (ressources externes) ;
- N est l'alphabet des étiquettes (notifications) ;
- T est la fonction de transition dans $S \times RE \times N \times S$.

Pour la formalisation des transitions, nous reprenons la même notation binaire qu'en simulation interprétée. À ceci près qu'elle est simplifiée, puisque nous n'utilisons pas de classes d'instructions. Pour $(s, re, n, s') \in T$, on note $s \xrightarrow{re(n)} s'$, avec re et n en notation binaire. La notation signifie qu'une transition part de s et arrive à s', en étant conditionnée par re et permettant de récupérer l'information n comme notification. On retrouve ce formalisme sur la Figure 4.5. Dans cet exemple, nous avons seulement une notification qui représente la sortie d'une instruction du pipeline. Il y a deux ressources externes. L'instruction `b`, avec pour PC `pc2`, nécessite de prendre la deuxième ressource externe pour entrer dans le pipeline.

On peut noter que l'état ③ laisse apparaître une bulle dans le pipeline. En effet, la seconde ressource externe étant prise, l'instruction suivante ne peut prendre la ressource en entrant dans le pipeline.

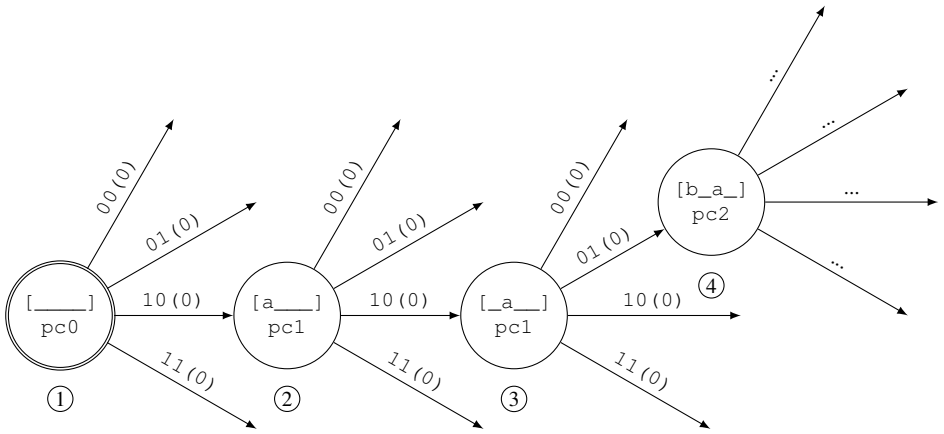

FIGURE 4.5 – Un automate en simulation compilée : `10(0)` signifie que la première ressource externe est libre (0) et la seconde est prise (1) (`10`), et qu'il n'y a pas de notification (`0`).

4.2.3 Gestion des branchements

En simulation interprétée, la manipulation du flot de contrôle est entièrement traitée à l'exécution. En simulation compilée, il faut disposer de mécanismes dans le modèle pour cet usage.

Dans le cas d'un branchement conditionnel, un état, via le même état de disponibilité des ressources externes, peut avoir deux successeurs (si le branchement est pris ou non). Pour que le modèle conserve un caractère déterministe, une ressource externe spécialement allouée aux branchements est utilisée. En effet, l'utilisation des ressources externes est le seul moyen permettant de distinguer deux transitions dans notre modèle.

On lui donne la signification suivante : si la ressource externe est prise, le branchement l'est également (*i.e.* le programme poursuit sur la cible du branchement), et inversement, si la ressource externe n'est pas prise, le branchement ne l'est pas.

De la même manière, il est possible de modéliser la latence d'un branchement, due aux aléas de contrôle, à l'aide d'une ressource externe spécifique. En fonction de la politique de branchement, celle-ci laisse ou non les instructions suivantes entrer dans le pipeline. Si la microarchitecture utilise un prédicteur de branchement, il peut être simulé à l'exécution pour alimenter la valeur de cette ressource externe. Ces éléments sur la latence de branchement ont fait l'objet de réflexions, mais ils n'ont pas été implantés.

Un exemple est donné en Figure 4.6. La seconde ressource externe représente la gestion des branchements (utilisé dans ce cas pour le branchement `b` à `pc0`). S'il est pris, alors le modèle va à la cible de PC (`pc3`), sinon il va au prochain PC (`pc1`). Ainsi l'on peut voir entre l'état ① et ② que le branchement est pris. Entre l'état ① et ③ le branchement n'est pas pris.

Remarque : Cette technique permet de modéliser le comportement des branchements conditionnels directs. Elle n'est pas suffisante pour les branchements indirects, où la cible est calculée lors de l'exécution. Un pareil cas poserait effectivement de plus importants problèmes, car, de façon générale, un état peut avoir un nombre considérable de successeurs (selon toutes les cibles possibles qui peuvent être calculées).

4.2.4 Gestion des retours de fonction

L'utilisation d'une ressource externe spécifique ne suffit pas pour traiter le cas des branchements indirects. Or, ils sont essentiels pour modéliser la grande majorité des programmes de systèmes

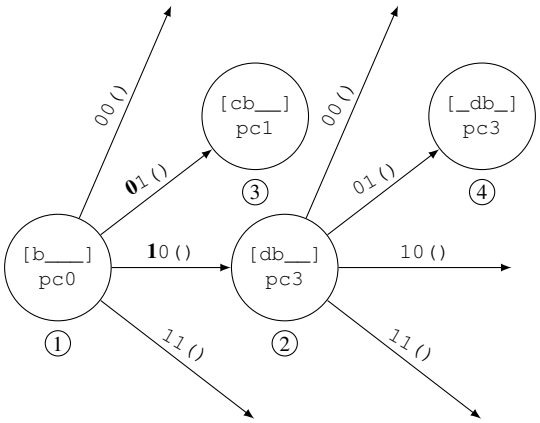

FIGURE 4.6 – La seconde ressource externe spécifie si un branchement (comme `b`) est pris ou non.

embarqués : les retours de fonction opèrent à l'aide de branchements indirects.

Un retour de fonction est un branchement indirect particulier, dont la cible dépend de la valeur du compteur programme au moment de l'appel de la fonction. Durant l'exécution du programme, une pile de données est manipulée. On l'appelle la pile des appels. Comme on peut le voir sur la Figure 4.7, lorsqu'une fonction est appelée ① ou ②, l'adresse de l'instruction qui suit l'appel est empilée. Lors du retour de fonction ③ ou ④, la cible du branchement est l'adresse qui est dépilée. Ainsi le retour de fonction prendra pour cible l'instruction qui suit l'appel de fonction.

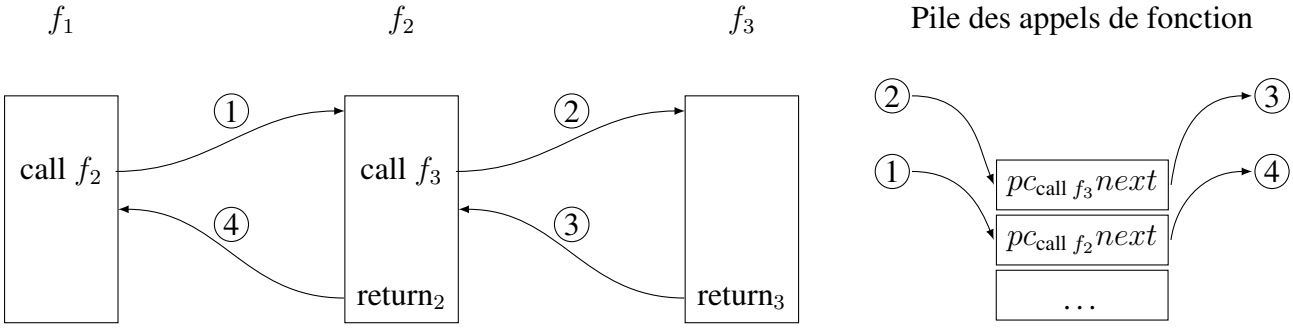

FIGURE 4.7 – Manipulation des fonctions. À chaque appel d'une fonction, l'adresse suivante est empilée. À chaque retour de fonction, l'adresse est dépilée.

On peut s'appuyer sur cette particularité pour simuler ce type de branchements, en intégrant la pile des appels à notre modèle, comme cela a été fait dans la sous-section 4.2.2. Cela revient à considérer que notre système n'est pas seulement déterminé par l'état du pipeline et l'état courant de lecture du programme, mais également par une certaine mémoire sur le passé du programme : par la suite de quels appels de fonction le programme en arrive à l'adresse courante.

Une telle modélisation a un impact sur la taille de l'automate. En effet, s'il est possible d'accéder au même endroit dans le code par des appels de fonction différents, plusieurs états y correspondront, puisque l'état de la pile sera différent. La taille du modèle augmentera donc d'autant qu'une fonction sera appelée plusieurs fois.

Par ce procédé, le branchement du retour de fonction dans l'automate perd toute ambiguïté

sur sa cible. Un état qui possède une certaine pile des appels n'aura pour successeur, par un retour de fonction, qu'un état avec une pile des appels.

Cette solution ne permet toutefois pas de traiter tous les types de branchements indirects. Il ne rend pas non plus possible la modélisation de programmes récursifs, car cela amènerait un nombre d'états infini, l'étude statique du programme ne pouvant pas déterminer la condition de terminaison de la récursion. Cependant, ce type de programme est rare dans les systèmes embarqués temps réel. Notre méthode rend seulement possible la simulation des programmes usuels dans ce domaine.

La démarche pour gérer cette pile est la suivante : quand un appel de fonction entre dans le pipeline, le prochain PC est empilé, et le prochain état a pour PC la cible du branchement. Quand un retour de fonction entre dans le pipeline, un PC est dépilé, et le prochain état a pour PC celui dépilé dans la pile des appels. Cette procédure est présentée sur la Figure 4.8. Il s'agit, en réalité, de la même procédure que celle qui est effectivement suivie dans le processeur.

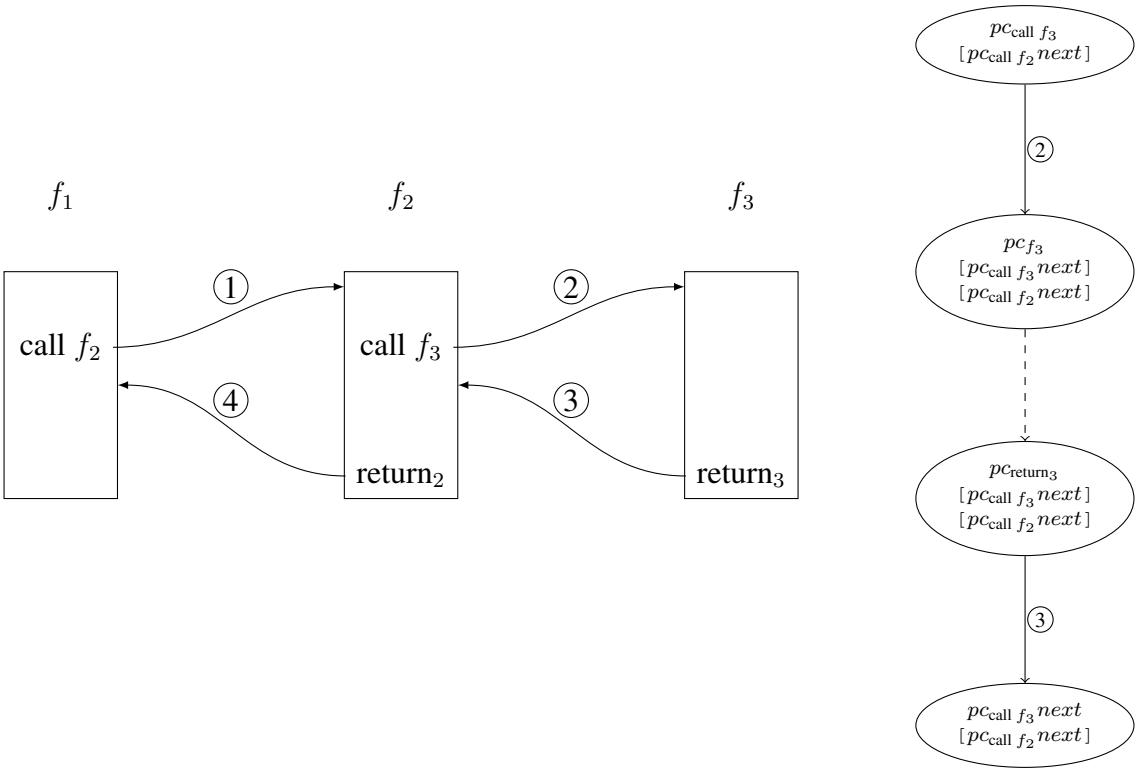

FIGURE 4.8 – Un appel de fonction empile l'adresse de l'instruction suivante, et un retour de fonction dépile cette adresse.

4.3 Algorithme

Dans cette section, nous présentons les éléments pour implanter ce modèle. Cela inclut les réflexions et les choix qui ont été faits sur la chaîne de développement, les principaux algorithmes pour la construction du modèle et comment opère l'exécution du simulateur.

4.3.1 Chaîne de développement

La mise en œuvre de la simulation compilée nécessite de disposer lors de la compilation d'outils pour analyser le code. Ces outils d'analyse sont disponibles grâce au simulateur fonctionnel

(ISS). Un module analyseur a été conçu pour extraire du code les informations nécessaires à son analyse, via l'utilisation de l'ISS.

La chaîne de développement qui en ressort est présentée sur la Figure 4.9.

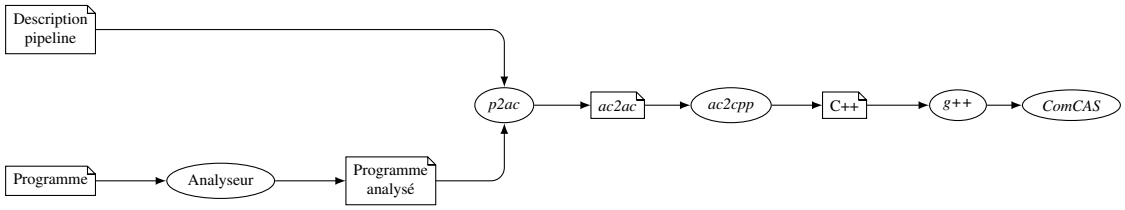

FIGURE 4.9 – Chaîne de développement de l'outil HARMLESS en simulation compilée avec module analyseur

4.3.2 Analyse du code

Le module analyseur traite le programme à simuler pour fournir au programme de construction du modèle (*p2ac*) les informations suivantes :

- l'adresse mémoire de l'instruction,
- le mnémonique de l'instruction (à des fins de débogage),
- le nombre de morceaux de l'instruction (*chunk*),
- si l'instruction est un branchement, et dans ce cas :
  - si le branchement est conditionnel,
  - si le branchement est un appel de fonction,
  - si le branchement est indirect,
  - quelle est la cible du branchement,
- les registres lus,
- les registres écrits,
- si l'instruction est un accès mémoire, et dans ce cas :
  - l'offset pour accéder à l'emplacement mémoire,
  - quel registre est utilisé pour accéder,
  - quel registre est utilisé en tant qu'offset (si c'est le cas).

Nous nous arrêtons sur les trois derniers points qui méritent quelques clarifications. Les modes d'adressage que nous gérons sont du type $[rX] + Y$ ou $[rX] + [rY]$, c'est-à-dire que l'adresse est constituée d'un registre (rX), dont on récupère la valeur, auquel on ajoute un offset, qui peut être une valeur fixe (Y) ou le contenu d'un autre registre (rY).

Toutes ces informations sur le code sont récupérables en puisant dans la description du jeu d'instructions et grâce à l'ISS. En effet, celui-ci dispose déjà d'outils pour décoder les instructions. Certains points ont cependant fait l'objet de quelques développements. La détermination des branchements, par exemple, se fait en analysant les opérations faites par l'instruction. Si celle-ci affecte le PC, c'est qu'il s'agit nécessairement d'un branchement.

4.3.3 Construction de l'automate

L'automate est construit sur le principe d'un parcours en profondeur de graphe, via le programme *p2ac* (pour *pipeline to automaton compiled*. Il reprend la structure existante de la construction du modèle en simulation interprétée.

On dispose de deux listes, gérés en LIFO comme des piles, pour stocker les états : une contient les états à traiter et une autre les états traités. Durant l'initialisation, on stocke dans la première pile l'état initial.

Tant que la pile des états à traiter n'est pas vide, on peut extraire un élément de celle-ci et le traiter. Pour chaque état, il y a autant de successeurs à calculer que de combinaisons possibles de disponibilité des ressources externes. Une fois le traitement de l'état effectué, il est empilé dans la pile des états traités. C'est ce que présente l'algorithme 3.

On voit ici apparaître ce que nous avons remarqué entre la construction de la simulation compilée et de la simulation interprétée, car cette dernière calcule, en plus, ses successeurs selon les possibilités de classes d'instructions qui peuvent entrer dans le pipeline. Ainsi dans le programme de *p2ac*, pour chaque état on parcourt seulement l'ensemble des ressources externes.

Algorithme 3 Construction du modèle — p2ac

```
1: Initialisation
2: while la liste des états à traiter n'est pas vide do
3:     s ← pop(liste des états à traiter)
4:     for all état de disponibilité des ressources externes do
5:         Calcul de s', le successeur de s pour toutes les valeurs de ressources externes possibles
6:         if s' n'appartient ni à la liste des états traités, ni à la liste des états à traiter then
7:             Ajout de s' à la liste des états à traiter
8:         end if
9:     end for
10:    if s n'appartient pas à la liste des états traités then
11:        Ajout de s à la liste des états traités
12:    end if
13: end while
```

Le nouvel état se calcule en faisant évoluer les instructions dans le pipeline, selon la disponibilité des ressources internes et externes. Si une instruction est un appel de fonction, le PC de l'instruction suivante est empilé sur la pile des appels. S'il s'agit d'un retour de fonction, il est dépilé et permet d'actualiser le PC courant. On fait donc également évoluer le PC courant : il s'agit du PC suivant ou de la cible de branchement. C'est ce que présente l'algorithme 4, que nous avons un peu simplifié.

Une fois le modèle construit, l'automate est stocké dans un fichier intermédiaire sous le format *ac2ac*, qui rassemble les propriétés générales de l'automate, et le code des différentes transitions.

4.3.4 Génération du simulateur

Le modèle est traduit, grâce au programme *ac2cpp*, dans un code C++ qui sera utilisé pour compiler la partie CAS du simulateur.

L'automate dans le format *ac2ac* est analysé. Il est retranscrit sous la forme d'un tableau à l'intérieur d'un code C++. L'index du tableau est fonction de l'état d'origine et des ressources

Algorithme 4 Calcul du successeur

```
1: for e étage du pipeline, partant de la fin jusqu'au début do
2:     e' ← l'étage suivant
3:     if e étage de lecture then
4:         isDataConflict ← isDataConflict(instruction i de l'étage e)
5:     end if
6:     if pas de conflits de ressources ET NON isDatatConflict then
7:         Libération de la ressource de l'étage e
8:         Stocker l'instruction de l'étage e dans l'étage e'
9:         Prendre la ressource de l'étage e'
10:        Vidage de l'instruction de l'étage e
11:    end if
12: end for
```

externes. La valeur retournée permet d'extraire l'état successeur et les notifications. Ce code, une fois compilé avec celui du simulateur, permet l'obtention du CAS.

À la suite de cela, l'exécution du simulateur diffère peu de celle de la simulation interprétée. Il n'est plus besoin d'extraire la classe d'instructions du programme à simuler pour calculer le nouvel état dans l'automate. Seul le calcul des ressources externes est suffisant.

4.4 Gestion des dépendances de données

En l'état, le simulateur obtenu ne présente pas de gain de vitesse par rapport au simulateur interprété. Mais, comme nous le disions, l'intérêt que présente la simulation compilée est de permettre le déplacement de tâches d'analyse du code de la phase d'exécution à la phase de compilation. C'est ce qui a été fait avec la gestion des dépendances de données et que nous allons explorer dans cette section.

Dans la simulation compilée, ce traitement est effectué lors de la construction du modèle. La structure de l'automate est réduite des états qui ne tiennent pas compte des conflits de données, simplifiant par là l'exploration de l'automate lors de l'exécution.

Pour effectuer cette tâche, on dispose, grâce au module analyseur :

- des registres dans lesquels écrit chaque instruction ;
- des registres que lit chaque instruction.

La description de l'architecture est également enrichie pour inclure les étages du pipeline qui sont consacrés à l'écriture et à la lecture de registres. Avec ces différentes informations, il est possible de gérer les dépendances de données.

Lors du calcul du successeur, on conditionne l'évolution du pipeline avec la règle suivante qui permet de tenir compte des aléas LAE : une instruction ne peut entrer dans le registre de lecture qu'à la condition qu'elle ne lise pas un registre dans lequel on prévoit d'écrire les instructions en aval.

Cette condition donne le code présenté dans l'algorithme 5, qui pour une instruction i renvoie comme booléen si elle ne lit pas un registre occupé. Cette fonction se retrouve ensuite dans l'algorithme 4 de calcul du successeur.

On l'a vu, avec un tel processus, les conflits de données sont directement inclus dans la structure de l'automate. Il n'est ainsi nullement besoin de disposer d'une ressource externe pour déterminer à l'exécution ce fait. Cela permet, d'une part, de retirer toutes les tâches allouées à

Algorithme 5 Y a-t-il un conflit de données ?

```
1: function ISDATACONFLICT(instruction i)
2:     isDataConflict ← faux
3:     for all étage e entre l'étage de lecture et l'étage d'écriture do
4:         if l'instruction à l'étage e a pour registres d'écriture des registres de lecture de l'instruction i then
5:             isDataConfict ← vrai
6:         end if
7:     end for
8:     return isDataConflict
9: end function
```

la détermination d'un aléa de données dans le cycle de simulation et, d'autre part, de réduire la complexité du modèle. En effet, le nombre de transitions sortantes pour chaque état est égal à 2^n, si n est le nombre de ressources externes. En retirant la ressource externe allouée aux dépendances de données, on divise donc par deux le nombre de transitions sortantes.

4.5 Estimation théorique de la taille du modèle

Dans le but de donner une idée de la complexité du modèle ComCAS, nous proposons d'évaluer le nombre d'états. Cette estimation a aussi permis, par ailleurs et dans une certaine mesure, de s'assurer de la cohérence de nos résultats.

Comme la simulation compilée se contente de modéliser les états qui sont effectivement rencontrés au cours de la simulation, une première réflexion peut aboutir à estimer que la taille du modèle est réduite devant celle de la simulation interprétée. Cette réflexion est en partie erronée et cette évaluation théorique permet de le comprendre.

Le dénombrement des états dans la simulation compilée est plus complexe qu'en simulation interprétée. En effet, le modèle ne regroupe pas tous les états possibles, mais ceux qui sont effectivement explorés lors de l'exécution du programme. La méthode de dénombrement utilisée pour la simulation interprétée sur-évaluerait donc largement la taille du modèle.

Notons, de plus, qu'une estimation exacte n'est pas nécessairement à portée, et que nous ne donnerons dans cette sous-section qu'une borne maximale à la taille du modèle.

En simulation compilée, un état de l'automate est composé de l'état du pipeline et des ressources internes, de l'adresse de la dernière instruction lue et la pile des appels de fonction. Dans une première étape, nous allons considérer qu'il n'y a pas de pile dans les états. Nous la rajouterons par la suite.

Pour présenter d'abord notre méthode, elle consiste à compter les états du pipeline pour un PC donné.

Nous travaillons sur trois différentes situations dans le flot de contrôle : une configuration linéaire du programme, *i.e.* sans branchement, le début du programme et une configuration de branchement. Pour terminer, nous étudierons quelles modifications impliquent l'utilisation de la pile des appels de fonction.

4.5.1 Configuration linéaire

Dans la configuration parfaite d'un programme linéaire, *i.e.* sans branchement, pour un PC donné, un seul passé existe. En conséquence, l'état du pipeline est seulement déterminé par l'emplacement des éventuelles bulles entrées dans le pipeline pour résoudre les aléas.

On peut voir ce cas sur la Figure 4.10. Comme l'ordre des instructions présentes dans le pipeline est fixé par le programme, les seules variations dans le pipeline sont dues aux bulles qui peuvent apparaître.

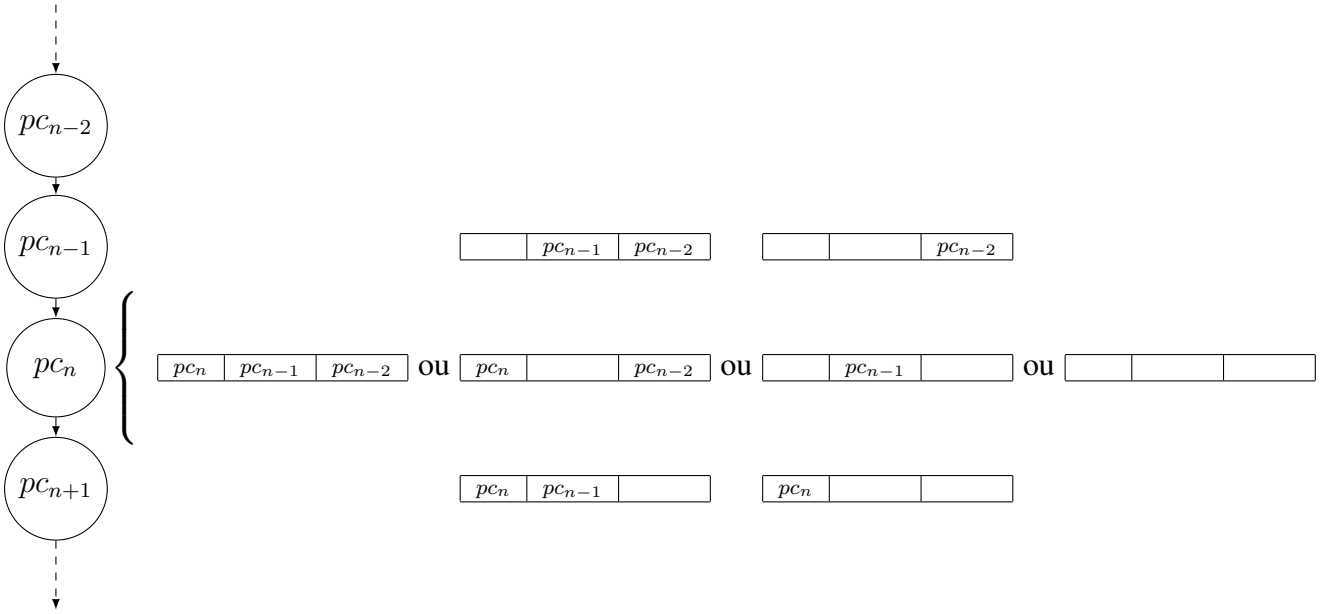

FIGURE 4.10 – Un programme dans une configuration linéaire parfaite avec les pipelines possibles pour l'adresse pc_n. (Le pipeline a 3 étages.)

Comme l'ordre des instructions est fixé par la configuration du programme, il est possible de faire abstraction de quelles instructions sont dans le pipeline pour ne considérer que l'emplacement des bulles (voir Figure 4.11). Ce changement de perspective permet de faire apparaître le dénombrement comme un problème de combinatoire classique, sans ordre.

Si e est le nombre d'étages du pipeline, et pour k instructions présentes dans le pipeline ($k \in [0; e]$) déterminer les états possibles du pipeline revient à choisir l'emplacement des $e - k$ bulles dans le pipeline. C'est-à-dire choisir $e - k$ éléments parmi e, donc $\binom{e}{e-k}$, ce qui est équivalent à $\binom{e}{k}$ (combinaison de k parmi e). Ainsi, pour obtenir le nombre total d'états du pipeline, quel que soit le nombre d'instructions k, il suffit de sommer l'expression sur k. On appelle n_n le nombre d'états du pipeline pour l'adresse pc_n.

$$n_n = \sum_{k=0}^{e} \binom{e}{k} \tag{4.1}$$

4.5.2 Début du programme

Si l'instruction en question se trouve au début du programme, pc_n avec $n < e$, il est impossible d'avoir plus de n instructions dans le pipeline. Dans ce cas, la somme précédente se retrouve tronquée à $\sum_{k=0}^{n} \binom{e}{k}$.

Pour simplifier le calcul, à partir de maintenant, nous utilisons la fonction $f_e : n \to \sum_{k=0}^{n} \binom{e}{k}$. Et, de par la formule du binôme de Newton, nous savons que $f_e(e) = 2^e$.

À ce moment du raisonnement, nous pouvons évaluer le nombre total d'états dans un programme parfaitement linéaire (sans branchement). Soit i le nombre d'instructions dans le programme. Les états possibles du pipeline pour les e premières instructions sont dénombrés grâce

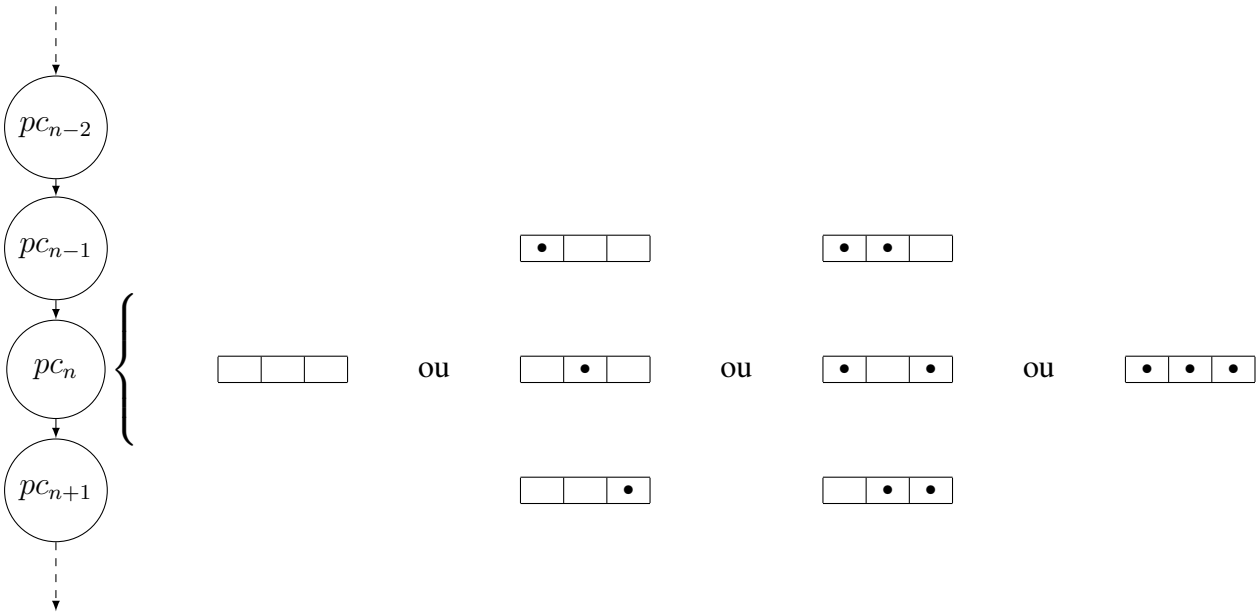

FIGURE 4.11 – Un programme dans une configuration linéaire parfaite avec les pipelines possibles pour l'adresse pc_n. On dénombre les dispositions possibles de bulles plutôt que les dispositions d'instructions. (Le pipeline a 3 étages.)

à la somme tronquée que nous avons vue, tandis que les états possibles du pipeline des autres $i-e$ instructions sont dénombrés par la somme complète. En regroupant ces deux cas, on obtient la formule suivante. On appelle n_Σ le nombre d'états total.

$$n_\Sigma = \sum_{k=0}^{e-1} f_e(k) + (i-e).2^e \quad \left| \begin{array}{l} n_\Sigma \text{ : nombre total d'états} \\ e \text{ : nombre d'étages dans le pipeline} \\ i \text{ : nombre d'instructions dans le programme} \end{array} \right. \tag{4.2}$$

Cette formule chiffre la taille maximale du modèle pour un programme sans branchement. Ce maximum est atteint si tous les états du pipeline sont explorés. C'est notamment le cas quand une ressource externe gère les entrées dans le premier étage du pipeline (accès au bus ou cache d'instructions), permettant de cette manière la formation de toutes les combinaisons possibles de bulles dans le pipeline.

4.5.3 Configuration de branchement

Le nombre d'états du pipeline est plus grand si l'on inclut les branchements dans le flot de contrôle.

Considérons le cas de la Figure 4.12, avec $k < e$. Dans cette situation, si nous mettons j instructions dans le pipeline avec $j \leq k$, aucune instruction précédant le branchement n'est présente dans le pipeline. Ainsi, nous restons dans le même cas de figure que précédemment : $\binom{e}{j}$ états du pipeline.

Mais, si nous mettons j instructions dans le pipeline avec $j > k$, alors pour chaque combinaison de bulles deux états de pipeline existent, selon que l'on provienne de la branche verte ou de la branche rouge. Ainsi, nous pouvons compter $2.\binom{e}{j}$ états de pipeline.

Le nombre total d'états de pipeline pour l'adresse pc_n est la somme de ces différents cas :

$$n_n = \sum_{j=0}^{k} \binom{e}{j} + \sum_{j=k+1}^{e} 2.\binom{e}{j} \tag{4.3}$$

Cette expression est équivalente à :

$$n_n = 2^{e+1} - f_e(k) \tag{4.4}$$

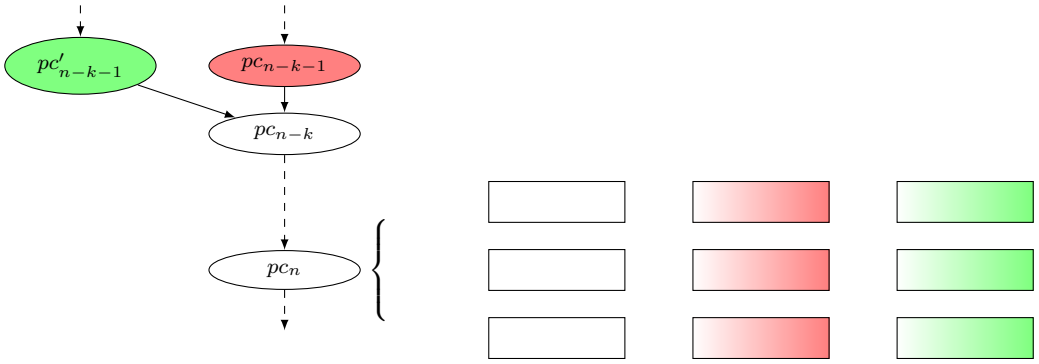

FIGURE 4.12 – Une configuration de branchement dans un flot de contrôle. Si $k < e$ alors à l'adresse pc_n, on peut retrouver trois types de pipelines : en blanc, les pipelines qui ne contiennent pas le branchement ; en rouge, les pipelines qui conservent des instructions de la branche rouge ; en vert, les pipelines qui conservent des instructions de la branche verte.

On fait intervenir à ce moment un nouveau paramètre b qui est le nombre de fois où ces cas se présentent. Il est bien sûr lié au nombre de branchements, mais, pour être précis, il correspond au nombre de branchements ayant pour cible une instruction qui peut déjà être explorée, de telle manière à ce qu'on retrouve bien la perturbation étudiée.

Pour une situation de branchement, nous avons k qui peut varier dans $[0; e-1]$. Il faut tenir compte de tout cet intervalle pour décrire l'ensemble des perturbations du branchement. Nous retrouvons donc dans notre décompte final la somme de l'expression de n_n, multipliée par b. Ceci est ajouté dans l'expression n_Σ, à laquelle on a pris soin de soustraire du calcul le nombre d'instructions qui sont perturbées par le branchement. Alors, le nombre total d'états devient :

$$n_\Sigma = \sum_{k=0}^{e-1} f_e(k) + (i - e - e.b).2^e + b.(\sum_{k=0}^{e-1} 2^{e+1} - f_e(k)) \tag{4.5}$$

C'est équivalent à :

$$n_\Sigma = (1-b).\sum_{k=0}^{e-1} f_e(k) + (i - (1-b).e).2^e \tag{4.6}$$

L'expression est valide si la configuration des branchements reste la même que celle donnée en Figure 4.12. Cela signifie que deux conditions doivent être respectées :

- les cibles de branchement sont séparées par plus de e instructions, pour ne pas rencontrer d'éventuelles interférences ;
- aucun branchement ne se situe à moins de e instructions d'une cible de branchement.

La seconde condition exclut les boucles trop courtes dans le programme. Pour ce qui est de la première condition, il s'avère qu'elle ne dégrade pas notre estimation. En effet, on peut concevoir que le dénombrement ne change pas en déplaçant virtuellement les branchements pour ne plus qu'ils interfèrent.

Comparaison avec la simulation interprétée L'analyse de notre évaluation révèle que le nombre d'états est linéaire par rapport au nombre d'instructions, et exponentiel par rapport au nombre d'étages. Nous pouvons comparer l'expression avec le nombre d'états en simulation interprétée : $(ci+1)^e$, qui augmente de façon plus importante en fonction de la taille du pipeline (ci étant le nombre de classes d'instructions).

4.5.4 Utilisation de la pile des appels

Dans notre évaluation, nous n'avons pas encore considéré l'utilisation de la pile des appels dans les états. Pour comprendre quelles modifications cette considération va apporter à l'évaluation, nous pouvons commencer par observer dans la Figure 4.13 l'effet sur le flot de contrôle de l'ajout de cette pile des appels.

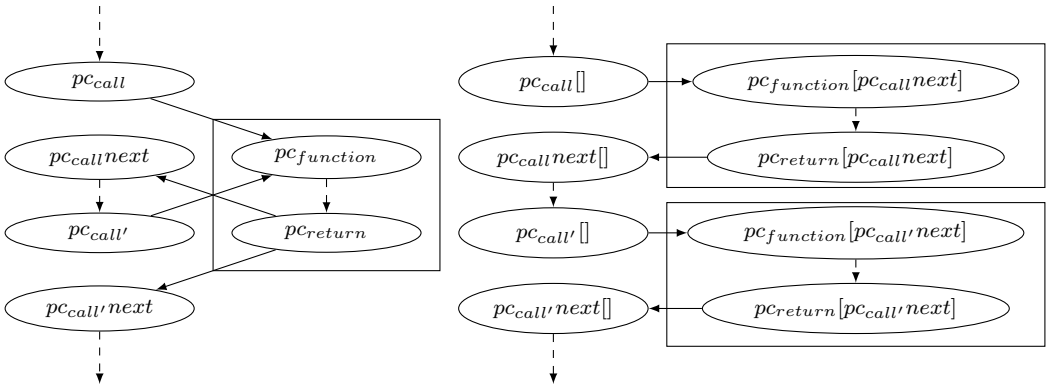

FIGURE 4.13 – Ces deux automates sont équivalents. La considération de la pile revient à dupliquer les fonctions selon les fois où elles sont appelées.

Cette modification peut être comprise comme ce que l'on appelle de l'*inlining* : le code des fonctions est dupliqué aux endroits où elles sont appelées. Si nous considérons ce nouveau flot de contrôle, le raisonnement qui a été fait jusque ici peut être à nouveau appliqué. Considérons donc un flot de contrôle où chaque état correspond à un couple : l'adresse mémoire de l'instruction courante et le contenu de la pile des appels. De cette manière, on retrouve bien la duplication des fonctions (un code appelé par des fonctions différentes sera représenté par des états distincts). Dans ce nouveau flot de contrôle, fonctionnellement équivalent au précédent, la pile des appels n'a plus d'importance et l'on peut s'en remettre à nos calculs précédents. Une chose diffère cependant : le nombre d'instructions n'est plus le même, ainsi que le nombre de branchements. Pour cette raison, nous utilisons la notation i' et b', pour les nouveaux paramètres de notre code. Il n'existe pas de relation simple entre i, b, i' et b'. Ainsi, le nombre d'états devient simplement :

$$n_\Sigma = (1-b').\sum_{k=0}^{e-1} f_e(k) + (i' - (1-b').e).2^e \quad \left| \begin{array}{l} n_\Sigma \text{ : nombre total d'états} \\ e \text{ : nombre d'étages dans le pipeline} \\ i' \text{ : nombre d'instructions dans le programme avec } \textit{inlining} \\ b' \text{ : nombre de branchements dans le programme avec } \textit{inlining} \end{array} \right. \tag{4.7}$$

Grâce à un programme construisant le flot de contrôle, il est possible de retrouver les valeurs attribuées à i et b, et également celles de i' et b'. À partir de ces valeurs, on peut se faire une idée de l'influence de l'*inlining*, qui s'avère très dépendante du code (voir la Table 4.1).

Ce programme de construction du flot de contrôle est obtenu simplement avec *p2ac* si l'on fixe le nombre d'étages du pipeline à 1. Dans ce cas, et si l'on retire tous les aléas possibles, le nombre d'états correspond au nombre d'instructions avec l'*inlining*. Tandis que si l'on fixe le nombre d'étages du pipeline à 2, et toujours si l'on écarte les aléas, on obtient un nouveau nombre, qui contient des doublons (un même PC pour deux pipelines différents) dus aux perturbations des branchements. Par une soustraction, on récupère ainsi le nombre de cibles de branchements avec l'*inlining*. Le calcul peut se faire sans l'*inlining* en filtrant les résultats des états obtenus pour en retirer la pile des appels.

Il est rendu ainsi possible de déterminer une borne maximale à la taille du modèle. Il nous permet aussi de vérifier que, dans le cas particulier d'une ressource externe sur le premier étage, cette borne est atteinte.

4.6 Résultats

Dans cette section, nous présentons les résultats expérimentaux sur les performances du modèle ComCAS en comparaison avec celles de la simulation interprétée. Ces résultats sont obtenus dans les mêmes conditions que celles exposées dans la section 3.3.2.

4.6.1 Taille des automates

Nous donnons dans la Table 4.1 une illustration de l'influence de *l'inlining* et le nombre d'états obtenu par l'outil ComCAS.

Cela nous permet de confirmer plusieurs idées intuitives. Premièrement, si une fonction n'est appelée qu'une seule fois durant l'exécution, l'*inlining* n'a pas d'influence sur notre modélisation. C'est ce qu'on peut voir avec les programmes comme bs ou cover. Plus généralement, tous les programmes où i=i' et b=b' ne font que des appels uniques à leurs fonctions. Deuxièmement, nous vérifions que le nombre d'états de notre modèle est bien borné par l'évaluation que nous avons faite. Il ne donne pas le même résultat que cette évaluation, parce qu'il n'explore pas tous les états du pipeline, en raison des contraintes que posent les aléas. Troisièmement, on a pu vérifier qu'avec des ressource externes particulières (rendant chaque état du pipeline accessible) nous retrouvons les mêmes résultats que dans notre évaluation.

Pour permettre une comparaison, le modèle interprété, pour une même configuration, donne un nombre de 1 024 états. Cette taille est indépendante du programme simulé, ce qui n'est pas le cas de notre modèle : plus le code à simuler est petit, plus le modèle l'est également et inversement.

Il a été remarqué que certains programmes donnaient de façon surprenante des tailles de modèle très grandes, si grandes qu'il n'était pas possible de les calculer en entier. Il s'agissait de programmes opérant des calculs flottants de façon logicielle, c'est-à-dire que ces opérations

TABLE 4.1 – Influence de *l'inlining* : **i** est le nombre d'instructions, **b** le nombre de cibles de branchements, **i'** le nombre d'instruction avec *l'inlining* et **b'** le nombre de cibles de branchement avec *l'inlining*

Programme	i	b	i'	b'	Borne théorique	nb d'états
adpcm	2 243	79	3 308	79	112 096	75 588
bs	84	4	84	4	2 928	2 061
compress	867	40	1 027	43	36 224	24 586
cover	145	7	145	7	5 120	3 434
crc	322	11	584	19	20 128	13 022
duff	88	3	88	3	2 976	2 101
expint	185	8	185	8	6 480	4 544
fdct	692	3	692	3	22 304	14 638
fibcall	58	3	58	3	2 016	1 447
fir	144	5	144	5	4 928	3 398
janne_complex	76	6	76	6	2 832	1 974
jfdctint	551	4	551	4	17 872	11 605
lcdnum	74	4	74	4	2 608	1 768
matmult	203	7	274	8	9 328	6 844
ndes	1 009	31	1 377	47	47 744	32 976
ns	116	8	116	8	4 272	2 907
prime	147	8	268	9	9 216	6 706

ne sont pas traitées directement par le matériel et passe par une série de sous-fonctions, appelées et réappelées, pour être exécutées. Le code élémentaire de ces sous-fonctions est minime, mais il se retrouve tant dupliqué que la taille de notre modèle explose. Cette multiplication de l'utilisation des fonctions montrent les limites de notre approche basée sur l'*inlining*.

4.6.2 Temps d'exécution

La Figure 4.14 présente les performances du modèle ComCAS en comparaison avec la simulation interprétée. Dans l'approche compilée, la génération du simulateur est plus complexe, comme elle requière de générer l'ISS, d'analyser le programme et de construire le simulateur. Cette compilation, lourde en temps d'exécution, est largement contrebalancée par un temps d'exécution plus court, lequel est effectué plusieurs fois.

Le temps de compilation varie en fonction de la taille du modèle généré. Il n'a pas été spécialement étudié au cours de cette thèse, car ne faisant pas l'objet d'optimisation. On dira simplement qu'il relevait de l'ordre d'une minute pour les petits modèles ($\sim$1 000 états), tandis que le programme adpcm, plus volumineux, nécessitait un temps de l'ordre de la dizaine de minutes de calcul.

Le principal intérêt de notre modèle vient de la possibilité de gérer les tâches d'analyse pendant la compilation. En particulier, le traitement des dépendances de données dans la phase de compilation a été implanté pour ComCAS. Cela réduit le temps d'exécution de 45,1% en moyenne comme nous pouvons le voir sur la Figure 4.14, et avec un maximum de 49,5% avec le benchmark *prime*. Cette amélioration significative montre l'intérêt de la simulation compilée pour la validation des systèmes embarqués temps réel.

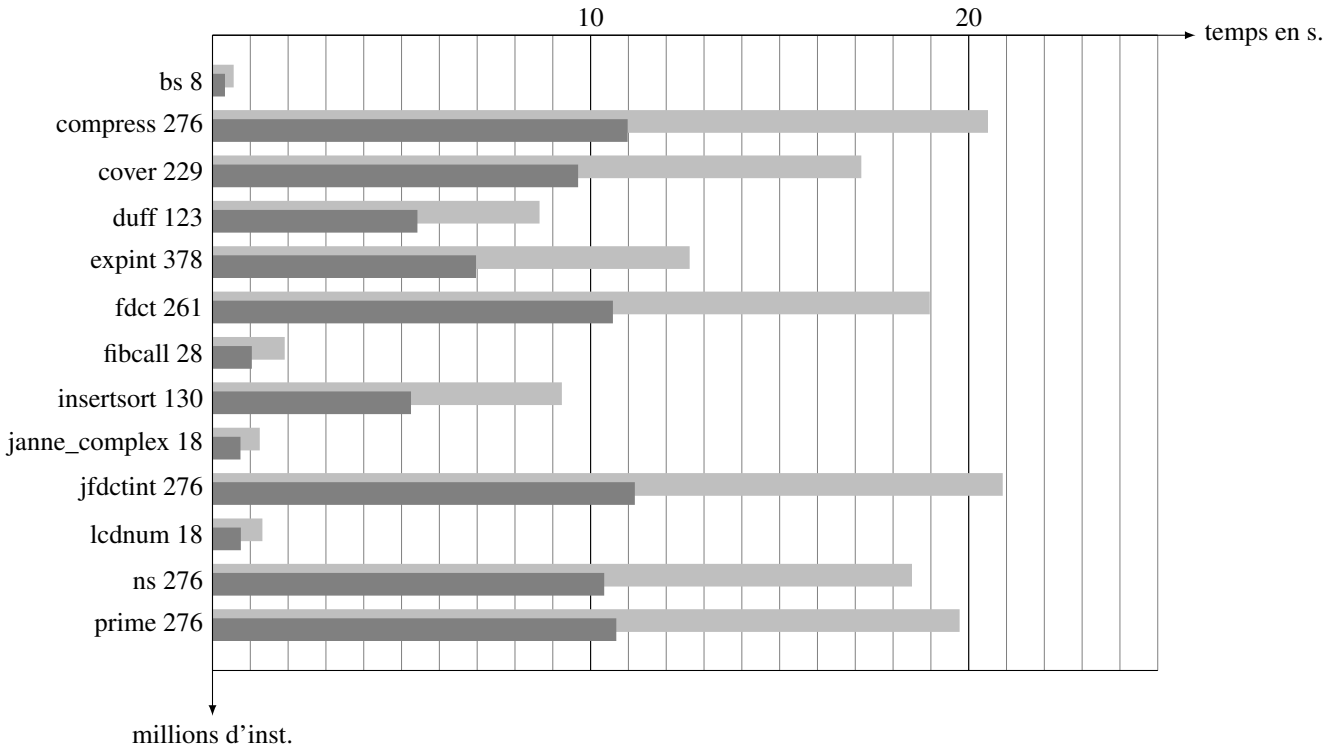

FIGURE 4.14 – La comparaison du temps d'exécution en secondes pour 50 000 exécutions. Le gris est pour la simulation interprétée, et le noir est pour la simulation compilée.

4.7 Conclusion

Nous avons développé un premier modèle pour adapter l'approche de la simulation compilée à la simulation temporelle et nous l'avons implanté dans l'outil ComCAS.

Cet outil ne permet cependant pas la simulation de tous les types de programmes. À cause de l'aspect statique de l'analyse du programme, il n'est pas possible de simuler :

- les programmes auto-modifiants
- les programmes récursifs
- les programmes utilisant des branchements indirects (hors retours de fonction)

De plus, l'implantation particulière que nous donnons pour tenir compte des retours de fonction limite une conséquente utilisation des fonctions. Cela empêche notamment la construction de modèle pour un programme gérant de façon logicielle les calculs flottants.

Dans le domaine des systèmes embarqués temps réel, on trouve peu de programmes faisant intervenir les trois premiers points. Le dernier est en revanche plus problématique et fait l'objet du chapitre suivant.

Nous avons étudié la taille maximale théorique de notre modèle et nous avons comparé les performances de notre modèle avec la méthode interprétée associée. Ces résultats montrent que le temps de calcul est réduit de 45% en comparaison de la simulation interprétée (voir Figure 4.15).

La simulation compilée est efficace parce qu'elle permet de retirer certaines tâches d'analyse de la phase d'exécution. Même si la technique ne tient pas encore compte des branchements

indirects, les appels et retours de fonctions sont considérés, ce qui permet de simuler la majorité des programmes de systèmes embarqués.

Ces travaux ont fait l'objet d'une publication scientifique dans la conférence internationale SIMUL'13 [BBBT13].

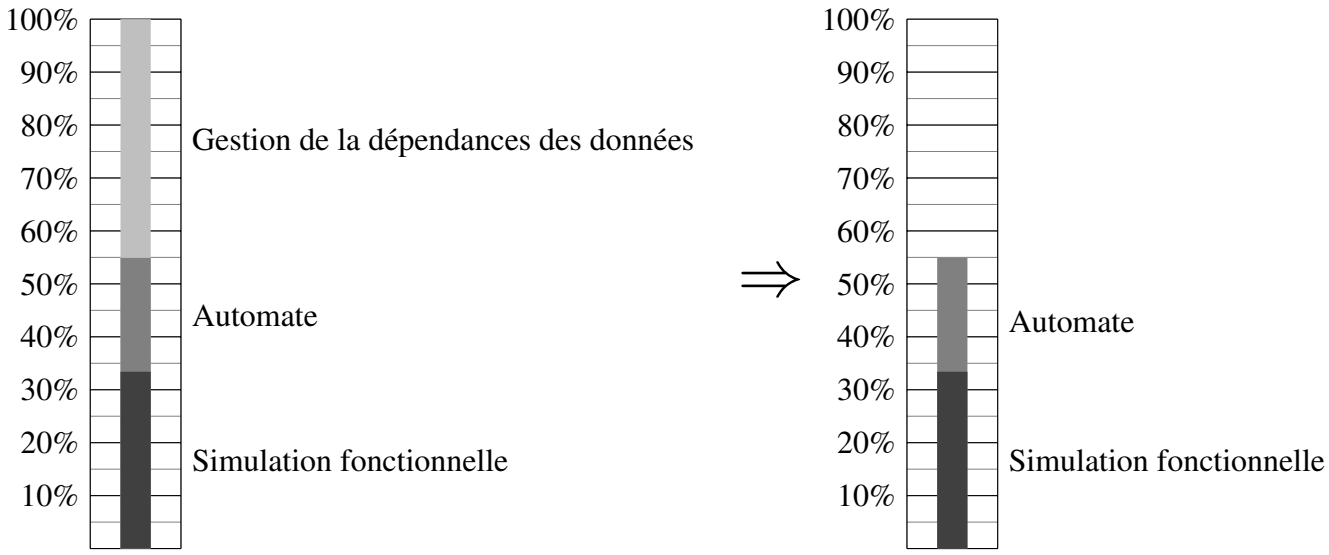

FIGURE 4.15 – Répartition moyenne des tâches dans le temps d'exécution du simulateur CAS. La gestion des aléas de données est supprimée grâce à la simulation compilée.

5

Simulation compilée — 2e modèle

5.1 Objectifs

La technique de la simulation compilée a permis de réduire le temps d'exécution de 45% en moyenne, grâce à la suppression de la gestion des aléas de données (voir Figure 5.1). Elle présente néanmoins certaines limitations qui peuvent être préjudiciables dans le domaine des systèmes embarqués temps réel, notamment la grande difficulté à construire les modèles pour simuler les programmes gérant les calculs flottants de façon logicielle, et plus généralement les programmes qui utilisent beaucoup de fonctions.

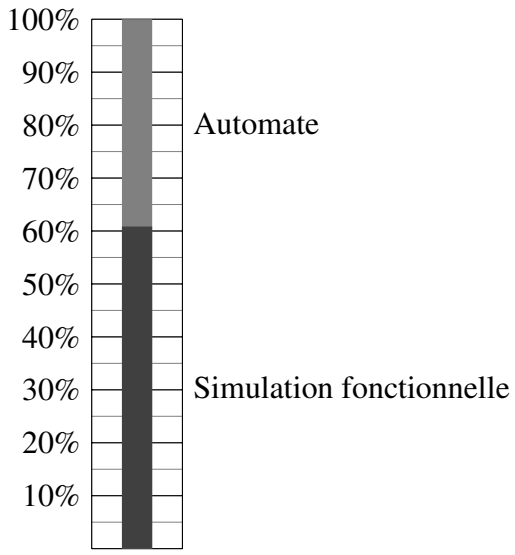

FIGURE 5.1 – Répartition moyenne des tâches dans le temps d'exécution du simulateur CAS. La simulation fonctionnelle occupe 60,8% du temps d'exécution et l'automate 39,2%.

Pour pallier ces défauts, nous proposons, dans ce chapitre, un modèle amélioré qui prend en compte les retours de fonction, sans pénalité sur la taille de l'automate. Cette amélioration permet de simuler les programmes utilisant la gestion logicielle des calculs flottants. De plus, il

rend possible la simulation des programmes récursifs, même si ces programmes ne présentent que peu d'intérêt dans le cadre des systèmes embarqués temps réel.

5.2 Modélisation

L'objectif de cette nouvelle modélisation est de synthétiser dans les états les informations qui permettent la manipulation du flot de contrôle, sans qu'il n'y ait de duplication des états.

Si l'on rejette comme méthode de traiter statiquement la pile des appels de fonction, il faut donc le faire dynamiquement. Ce sera le rôle d'un automate à pile.

5.2.1 Automate à pile

L'exécution du flot de contrôle des programmes, appel des fonctions compris, peut être considérée comme un langage non-contextuel (ou langage algébrique) déterministe, c'est-à-dire qu'il peut être reconnu à l'aide d'un automate à pile déterministe (PA). Grâce à ce modèle, il est possible de décrire naturellement l'évolution de notre système en y incluant les appels de fonction, sans passer par l'astuce lourde de l'*inlining*.

Qu'est-ce qu'un automate à pile ? Un automate à pile est une automate fini qui dispose d'une mémoire, organisée sous la forme d'une pile de données. Le tir d'une transition peut être conditionné par le sommet de la pile, et il est capable d'ajouter du contenu sur la pile (voir Figure 5.2).

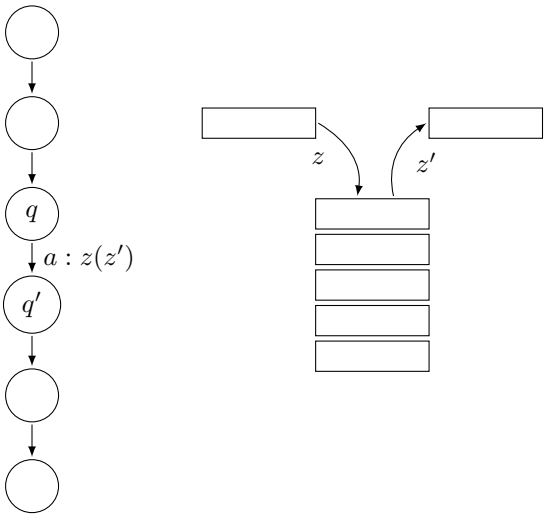

FIGURE 5.2 – Fonctionnement d'un automate à pile. La transition étiquetée a dépile z' et empile z. Dans notre modèle, la condition a sera équivalente aux ressources externes.

Cette modélisation offre la possibilité de traiter les branchements indirects que sont les retours de fonction de manière naturelle, puisque l'emploi de la pile de l'automate peut être directement équivalente à celui de la pile des appels.

5.2.2 Modèle

Grâce au modèle des automates à pile, il n'est plus nécessaire d'inclure dans le modèle l'état de la pile des appels, puisqu'elle est modélisée à l'exécution. L'état du système se retrouve

seulement décrit par :

- l'état du pipeline : quelle instruction se trouve dans chaque étage ;
- l'état des ressources internes ;
- et la position dans le programme (le Program Counter).

Le système change son état à chaque cycle selon la disponibilité des ressources externes et ce qui se trouve au sommet de la pile des appels.

Cette modélisation est formalisée de la façon suivante. Soit PA un automate à pile défini par $\{S, s_0, RE, N, Z, z_0, T\}$, où :

- S est l'ensemble des états ;
- s_0 est l'état initial dans S ;
- RE est l'alphabet d'entrée (ressources externes) ;
- N est l'alphabet des étiquettes (notifications) ;
- Z est l'alphabet de la pile ;
- z_0 est le symbole de fond de pile, dans Z ;
- T est la fonction de transition dans $S \times RE \times (Z \cup \epsilon) \times S \times N \times (Z \cup \epsilon)$.

$(s, re, z, s', n, z') \in T$ signifie qu'il y a une transition de l'état s à l'état s' avec re comme entrée et z au sommet de la pile. Cette transition dépile z, puis empile z'. (On parle de condition indirecte pour z.) La transition est étiquetée avec la notification n. Nous utilisons la notation $re : z(n : z')$ sur la transition.

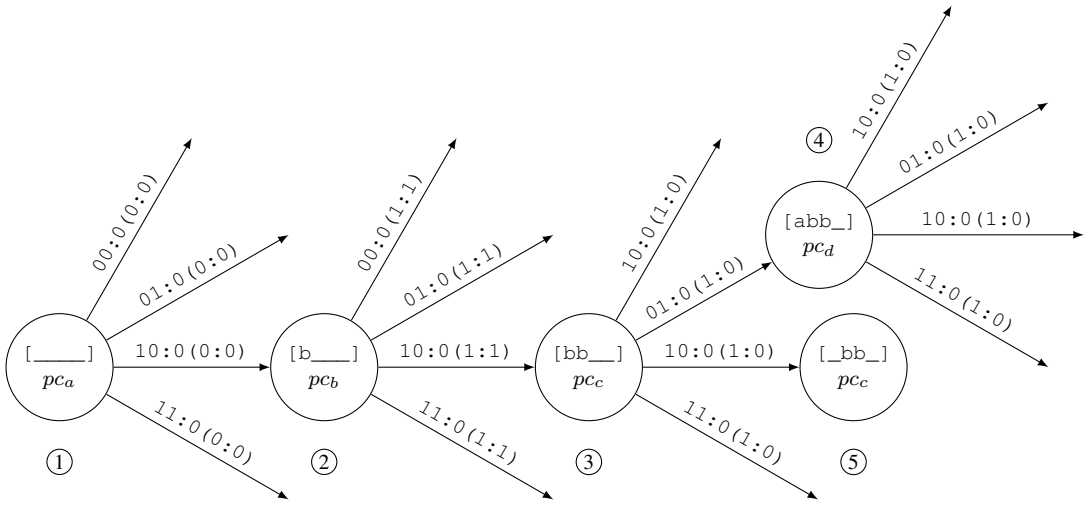

FIGURE 5.3 – Un automate en simulation compilée avec le deuxième modèle. `10:0(1:1)` signifie que la première ressource externe est libre et la seconde prise (`10:0(1:1)`, en partant du bit de poids faible), que rien n'est dépilé `10:0(1:1)`, que la notification arrive `(1:1)` et que l'on empile 1 `(1:1)`.

Dans la Figure 5.3, nous donnons un exemple du modèle avec automate à pile. Seulement une notification est représentée, indiquant l'*entrée d'une instruction dans le second étage du*

pipeline. Deux ressources externes sont présentes. La seconde ressource externe permet à l'instruction `a`, avec pour PC pc_d, d'entrer dans le pipeline. C'est pourquoi l'état ⑤ demeure au PC pc_c, la ressource ne permettait pas l'entrée de l'instruction. Comme l'instruction présente à l'emplacement pc_c empile 1 à son entrée dans le pipeline (état ③), cela signifie qu'il s'agit d'un appel de fonction.

Gestion des fonctions On se sert de cette pile pour déterminer quelle transition tirer lorsqu'un retour de fonction est rencontré. La Figure 5.4 montre un exemple de manipulation de la pile en modélisant le programme. La fonction f est appelée à deux reprises au cours du programme (pc_a et pc_m). Selon que l'automate provienne de ces deux états différents, un identifiant différent est ajouté à la pile (1 ou 2, respectivement). Cet identifiant permet, lors du dépilement au retour de fonction, d'orienter l'automate vers l'instruction qui suit l'appel de fonction précédent (pc_b ou pc_n, respectivement).

label	pc	inst
	pc_a	call f
	pc_b	
		...
	pc_m	call f
	pc_n	
		...
f	pc_y	
	pc_z	return

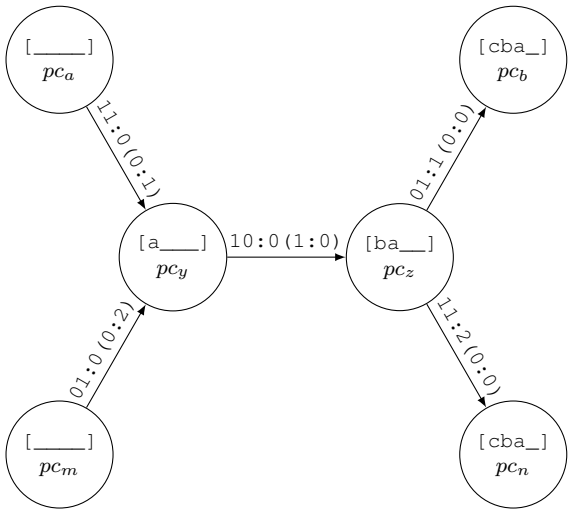

FIGURE 5.4 – La gestion de la pile : `11:0(0:1)` signifie que 1 est empilé, `01:1(0:0)` signifie que la transition est tirée si 1 est dépilée. `10:0(1:0)` signifie que rien n'est empilé et `10:0(1:0)` signifie que rien n'est dépilée. Si l'automate arrive depuis pc_a au retour de fonction, 1 est dépilé, donc l'automate poursuit sur pc_b. De même en provenance de pc_m avec l'identifiant 2 pour poursuivre l'exécution en pc_n au retour de fonction.

5.3 Algorithme

La construction de l'automate présente certaines difficultés, du fait de la nécessité de coordonner différentes informations qui se trouvent tantôt dans des états déjà construits, tantôt dans des états à construire de l'automate. Un simple algorithme de parcours en profondeur n'est pas suffisant pour cela.

La construction de cet algorithme est complexe, par les structures de données qu'elle utilise. Dans les sous-sections qui suivent, nous avons tenté de dresser une démarche par étapes pour montrer l'intérêt de chacun des éléments abordés et faciliter leur compréhension. Nous expliquons les différents problèmes que posent l'automate à pile et par quelle façon ils sont résolus.

5.3.1 Manipulation de la pile

Pour que l'automate à pile fonctionne de la manière dont nous l'avons modélisé, il faut que les identifiants qui sont dépilés lors du retour de fonction soient les mêmes que ceux empilés lors de l'appel de fonction correspondant. Cela nécessite une certaine harmonisation des identifiants.

Durant la construction de l'automate, lors de la création des transitions des appels de fonction, on ne dispose pas de l'ensemble des informations sur les appels de fonction qui interviennent, ce qui rend impossible d'attribuer des identifiants dans un premier temps.

Pour être capable de traiter, ultérieurement, les identifiants sur les transitions, on se dote de la structure de données suivante : un dictionnaire qui relie chaque appel de fonction à la liste des transitions qui lui sont attachées. Dans les programmes, il est fait référence à ce dictionnaire sous la dénomination $transVectorMap$.

Lors de la construction de l'automate, ce dictionnaire est renseigné. Lors du traitement des retours de fonction, il est possible d'y revenir pour faire coïncider les transitions entrantes de la fonction avec les sortantes.

Nous avons fait le choix de manipuler directement dans l'automate des identifiants pour la pile, plutôt que les adresses des appels de fonction, qui aurait été plus simple mais aurait demandé un travail ultérieur pour réduire cette notation. Pour tenir compte de cela, nous utilisons un dictionnaire qui permet de relier chaque adresse d'appel de fonction à l'indice qui lui est attaché. On y fait référence sous la dénomination $condMap$.

Pour manipuler ces dictionnaires, on se dote également d'un vecteur dans lequel sont rangés les adresses des appels de fonction $callPCVector$. En Figure 5.5, on en voit une illustration. L'exemple retrace la construction d'une partie d'un automate, que nous poursuivrons par la suite. Un premier appel de fonction est effectué, représenté par l'état e0. C'est le deuxième appel de fonction, après e1 et avec pour adresse pc1, qui nous intéresse. Les états suivants sont associés au code de la fonction et disposent des renseignements spécifiés.

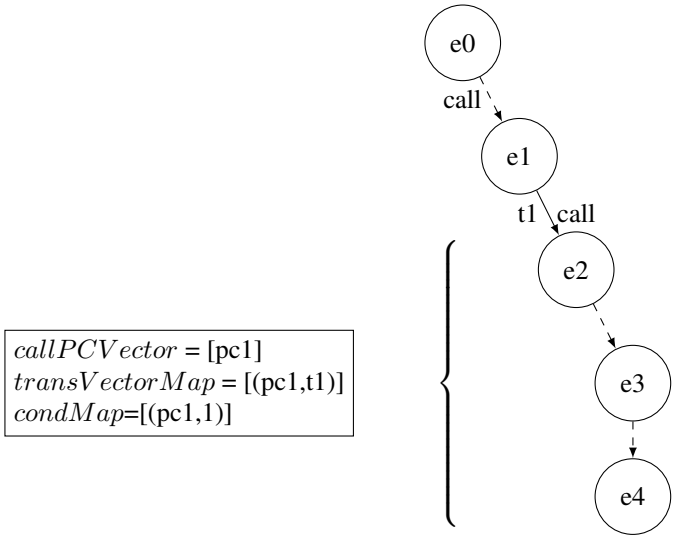

FIGURE 5.5 – Manipulation de la pile. Les états e2, e3 et e4 modélisent la fonction appelée. L'adresse de l'appel de fonction est stockée dans $callPCVector$. Les transitions correspondant aux appels de la fonction sont stockées dans le dictionnaire $transVectorMap$. La condition servant à la pile est stockée dans le dictionnaire $condMap$.

5.3.2 Factorisation de l'automate

Une autre difficulté se trouve dans le parcours de l'automate. Lorsqu'une fonction est appelée pour la seconde fois, les états de la fonction ont déjà été calculés et ne doivent pas l'être une nouvelle fois. Il faut donc être capable de reprendre la construction du modèle depuis les états du retour de fonction.

Cela se fait en renseignant quels états représentent les retours de fonction. On utilise un simple vecteurs contenant les indices des états modélisant un retour de fonction ($returnVector$). Lors de la construction de l'automate, ce vecteur est renseigné. Quand une fonction est appelée une nouvelle fois, en utilisant ce vecteur, on peut spécifier où la construction doit reprendre et sauter ainsi la partie déjà calculée de l'automate. (Voir une illustration en Figure 5.6)

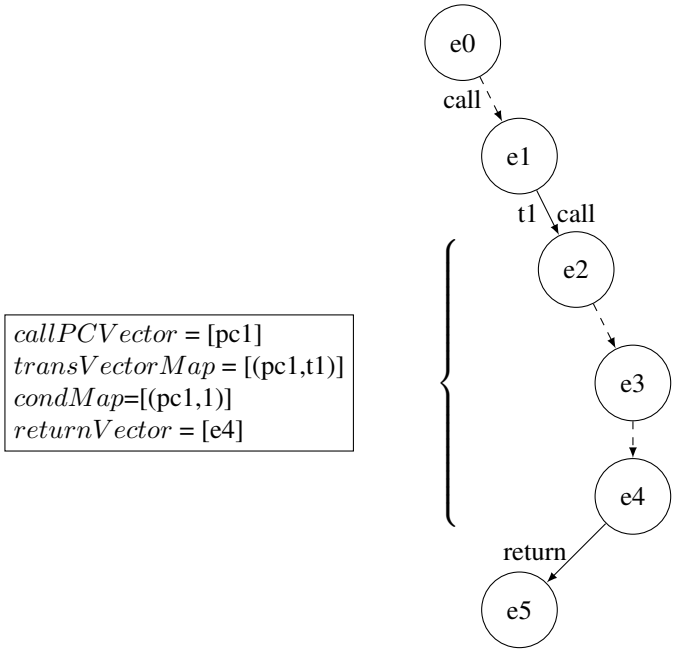

FIGURE 5.6 – Factorisation de l'automate. Les états correspondants aux retours de la fonction sont stockés dans le dictionnaire $returnVector$. Cette information permet, lors d'un nouvel appel de la fonction, d'en retourner directement à ces états sans avoir besoin de calculer une nouvelle fois les états intermédiaires.

5.3.3 Gestion de la structure de données

Toutes les structures de données que nous avons introduites ne sont pas propres à un état, mais à une fonction. Il faut donc être capable de diffuser ces informations à tous les états de la fonction.

Plutôt que d'assigner cette structure de données à chaque état de la fonction, ce qui demanderait un travail de réédition conséquent, nous avons décidé de l'externaliser. Pour profiter de la structure de l'automate, il a été choisi comme lieu de stockage un état particulier de la fonction : l'état appelant. (On utilisera la désignation de $stateCalling$.) Dans le reste de l'automate, il suffit de diffuser l'index de cet état, plutôt que toute la structure de données, pour permettre l'accès aux informations. (Voir une illustration en Figure 5.7.)

Lorsque une fonction est appelée une nouvelle fois, les successeurs créés ne sont pas immédiatement identiques aux états existants (e2 $\neq$ e8). On observe une latence correspondant à celle du pipeline. Pour éviter que, dans de telles situations, les états appelants n'amènent à

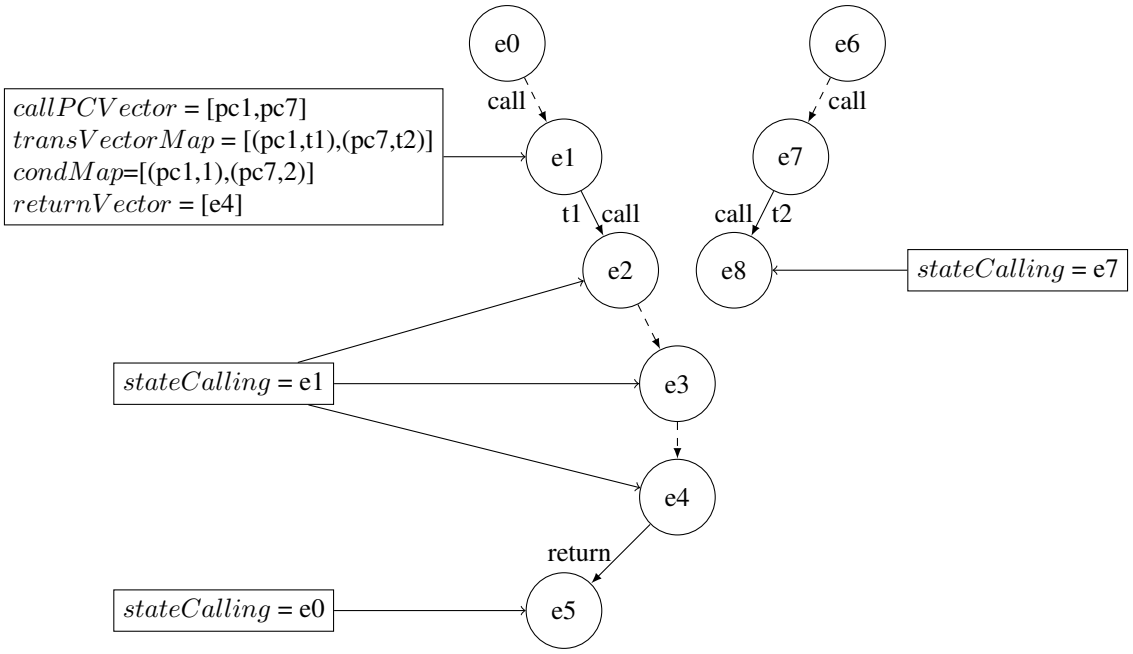

FIGURE 5.7 – Fonctionnement de l'état appelant. La structure de données est stockée dans l'état appelant e1. Tous les états de la fonction sont renseignés via la connaissance de cet état appelant. Lorsque la fonction est appelée une nouvelle fois, on remarque que, lors de la construction de l'automate, l'état appelant peut être différent. En effet, pour e8, l'état appelant est e7. Notons également que pour e5, l'état appelant est e0, puisque c'est cet état qui a appelé la fonction dans laquelle il appartient.

deux structures de données indépendantes, on se dote d'un nouvel indicateur qui désigne le premier état appelant ($firstStateCalling$). Avant que les états ne fusionnent, les états appelants diffèrent (vers l'appel de fonction qui leur correspond). Lorsque la construction de l'automate arrive à un point de fusion, il est possible de rassembler les structures de données dans le premier état appelant, servant de référence. Pour ne pas que les états intermédiaires, comme e8, considèrent toujours leur état appelant, e7, comme la référence des structures de données de la fonction, nous indiquons dans cet état appelant quel est l'état qui fait référence, e1, le premier état appelant. De cette manière, à la fin de la construction, il est partout possible sur les états de la fonction d'accéder à la bonne structure de données, cela même si on passe par un mauvais état appelant comme intermédiaire.

L'utilisation des états appelants pour stocker les différentes informations permet de résoudre le problème de l'imbrication des fonctions. En effet, l'état appelant connaît lui-même son état appelant, c'est-à-dire quel état représente l'appel de la fonction à laquelle il appartient, et ainsi de suite. De cette manière, il est possible, dans les états d'une fonction, de remonter aux structures de données de toutes les fonctions précédentes. C'est notamment utile pour construire les successeurs des retours de fonction. Cependant, du fait de la centralisation des informations d'une fonction sur un seul état, alors même que la fonction peut être appelée à de nombreux endroits, ce mécanisme n'est pas directement opérationnel. Il faut pouvoir disposer d'un dictionnaire reliant chacun des appels de fonction à son état appelant ($stateCallingMap$) pour qu'il le devienne.

5.3.4 Structure de données

Pour synthétiser ces différents points, il a été ajouté comme informations aux états :

- $stateCalling$: l'état appelant,
  Pour savoir où récupérer les données.

- $firstStateCalling$: le premier état appelant,
  Pour déterminer, entre les différents états appelants, quel est le premier qui a été traité.

- $callPCVector$: un vecteur contenant les PC des appels de fonction amenant à la fonction courante,
  Pour connaître les différents appels de la fonction.

- $transVectorMap$: un dictionnaire qui associe à chaque PC d'appel de fonction le vecteur des transitions entrantes dans la fonction,
  Pour connaître les transitions sur lesquelles éditer l'identifiant à empiler.

- $condMap$: un dictionnaire qui associe à chaque PC d'appel de fonction son identifiant,
  Pour connaître l'identifiant qui correspond à un appel de fonction.

- $stateCallingMap$: un dictionnaire qui associe à chaque PC d'appel de fonction son état appelant,
  Pour connaître à chaque appel de fonction, son état appelant et télescoper la structure.

- $returnVector$: un vecteur des états de retour de fonction,
  Pour poursuivre les calculs directement aux retours de fonction.

Dans l'exemple de la Figure 5.8, la structure de données est renseignée de la façon suivante :

- Pour e2, e3 et e4, $stateCalling$ = e1
- Pour e8, $stateCalling$ = e7
- Dans e7, $firstStateCalling$ = e1
- Dans e1, $callPCVector$ = [pc1,pc7]
- Dans e1, $transVectorMap$ = [(pc1,t1),(pc7,t2)]
- Dans e1, $condMap$ = [(pc1,1),(pc7,2)]
- Dans e1, $stateCallingMap$ = [(pc1,e0),(pc7,e6)]
- Dans e1, $returnVector$ = [e4]

5.3.5 Algorithme

La structure du programme reste la même que dans le premier modèle. Seules des modifications sont apportées pour permettre de traiter la structure de données et effectuer la manipulation de l'automate. Celles-ci sont présentées dans l'algorithme 6.

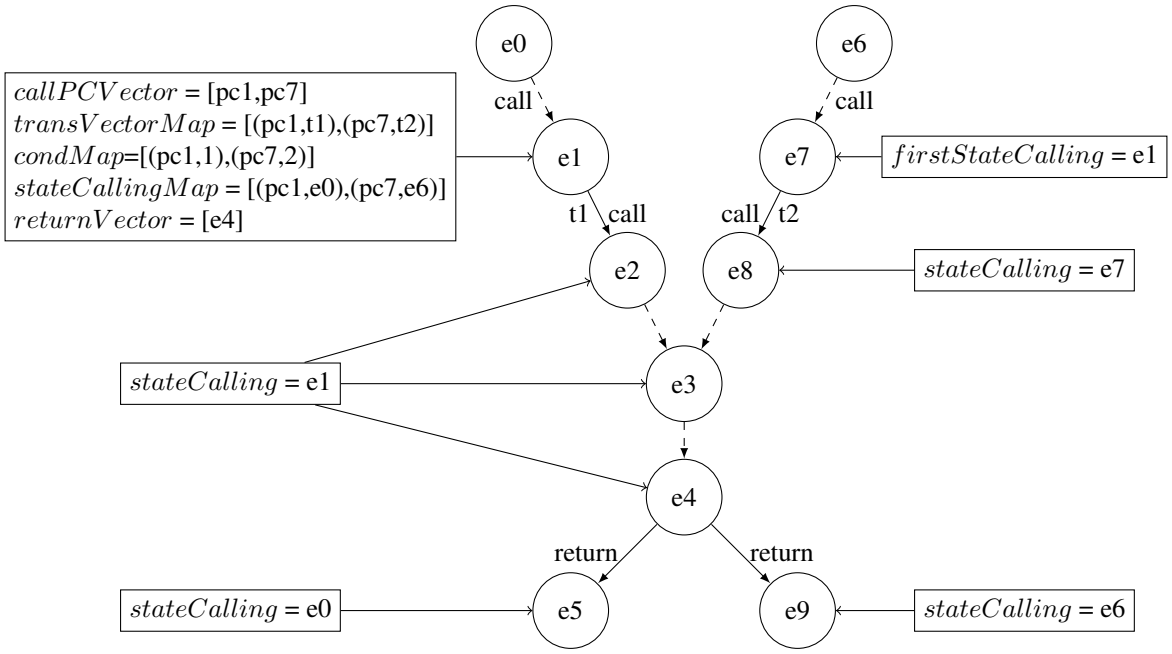

FIGURE 5.8 – La structure de données utilisée pour le deuxième modèle de simulation compilée. Pour relier l'état e8 à la structure de données en e1, on renseigne e7 avec un $firstStateCalling$: e1. Cependant, au cours de la construction, cette information ne peut être trouvée que lorsque les états des deux branches fusionnent en e3. Après cette fusion, il est possible de poursuivre la construction à l'état e4. À la construction du retour de fonction manquant (celui de pc7, représenté en e9), il est possible de consulter le dictionnaire $stateCallingMap$ pour obtenir le bon $stateCalling$: e6.

5.3.6 Exécution

Dans cette sous-section, il est expliqué comment la pile est manipulée durant l'exécution. On utilise l'algorithme 7.

Si l'empilement est une action simple (on extrait l'information de la transition et on l'ajoute au sommet de la pile), ce n'est pas le cas pour un dépilement. En effet, l'ordre des actions sur la transition et la pile est inversée. D'abord il faut dépiler pour connaître la transition à tirer. Cela impliquerait que l'instruction de dépilement devrait être donnée en amont, sur la transition précédente. Cependant, une telle pratique n'est pas envisageable, parce qu'il n'existe pas systématiquement de successions de transitions menant exclusivement à un retour de fonction (un retour de fonction conditionnel, par exemple). Cela présenterait le risque de dépiler inutilement, ou plusieurs fois.

Pour répondre à ce problème, les deux opérations sont concentrées sur la seule transition du retour de fonction. Par défaut, les transitions sont explorées avec une condition indirecte nulle. Si aucun successeur n'est trouvé, cela peut signifier deux choses : il s'agit d'un retour de fonction et la condition indirecte est non nulle, ou il s'agit de la fin du programme. Dans ce cas, on retourne à l'état précédent, on dépile la pile et on explore à nouveau les transitions avec cette nouvelle condition indirecte.

Algorithme 6 Module de manipulation de la structure de données

Création du successeur $succ$
Recherche des différents états appelants (soit e l'état appelant de $succ$)
if l'instruction courante est un appel de fonction **then**
    if $succ$ n'existe pas **then**
        Initialisation d'une structure de données dans le prédécesseur $pred$
        $succ.stateCalling \leftarrow pred$
    else
        Ajout de la transition courante à $pred.transVectorMap[PC_{call}]$
    end if
end if
if l'instruction courante est un retour de fonction **then**
    Ajout de l'état dans le $returnVector$ de l'état appelant e
    Télescopage : $succ.stateCalling \leftarrow e.stateCallingMap[PC]$
    Édition du dépilement sur la transition courante : $trans.pop = e.condMap[PC]$
end if
if $succ$ est identique à un état existant ($exist$) mais avec des états appelants différents **then**
    Fusion des structures de données
    Traitement de tous les retours de fonction connus par l'état appelant de $exist$
else
    Ajout des transitions connues par $next$ à $exist$
    Harmonisation des identifiants à empiler sur les transitions
end if

Algorithme 7 Exécution d'un cycle

Calcul des ressources externes
Condition indirecte fixée à 0
Calcul du successeur
if aucun état trouvé **then**
    if la pile est vide **then**
        return true ;
    else
        Retour à l'état précédent
        Condition indirecte fixée avec le sommet de la pile
        Calcul du successeur
        Dépilement
    end if
end if
Lecture de l'identifiant à empiler
if l'identifiant est valide **then**
    Empilement de l'identifiant
end if
Suite de la procédure

5.4 Résultats

Ce nouveau modèle permet de simuler, en plus des programmes simulables par le premier modèle, ceux utilisant les calculs flottants de façon logicielle et les fonctions récursives.

On observe que les temps d'exécution sont similaires à ceux du premier modèle, à une

augmentation de 8% près. Ceci est dû à la gestion dynamique de la pile des appels.

5.4.1 Taille des automates

Ce nouveau modèle permet de réduire la taille de l'automate en comparaison de la version précédente. Nous donnons dans le tableau 5.1 le gain de la réduction. Le calcul de la section 4.5 nous donnait la valeur théorique du nombre d'états, si la pile d'appels n'est pas utilisée. On retrouve précisément ce résultat avec ce modèle.

Nous pouvons noter que certains programmes ont le même nombre d'états dans les deux modèles. La raison est que la pile d'appels n'est pas pleinement utilisée : les fonctions sont appelées une fois durant toute l'exécution.

Pour le cas spécial des programmes utilisant la gestion logicielle du calcul flottant, la réduction est considérable, comme on peut le voir avec le programme basicMathVerySmall, une version simplifiée du mibench basicMathSmall.

Programme	**États avec inlining (Modèle 1)**	**États avec PA (Modèle 2)**	**Gain**
adpcm	18 229	12 096	33,6%
basicMathVerySmall	322 615	4 302	98,6%
bs	587	587	0%
compress	5 901	5 059	14,3%
cover	1 123	1 123	0%
crc	3 616	2 038	43,6%
duff	669	669	0%
expint	1 395	1 395	0%
fdct	3 707	3 707	0%
fibcall	479	479	0%
fir	1 016	1 016	0%
janne_complex	645	645	0%
jfdctint	3 134	3 134	0%
lcdnum	568	568	0%
matmult	1 941	1 516	21,9%
ndes	8 378	6 114	27%
ns	837	837	0%
prime	1 871	1 138	39,2%

TABLE 5.1 – Influence de l'automate à pile dans la taille du modèle

5.5 Conclusion

Dans ce chapitre nous avons proposé un nouveau modèle pour la simulation compilée. Son objectif était de réduire les restrictions qui pesaient sur son exécution, notamment la difficulté de construire le modèle pour les programmes utilisant de trop nombreuses fonctions.

Cette réduction des restrictions a été rendue possible au prix d'une augmentation du temps d'exécution de 8% en moyenne, en raison de la gestion dynamique de la pile des appels de fonction (voir Figure 5.9).

Les programmes restants qu'il n'est pas possible de simuler sont :

- les codes auto-modifiants
- les programmes utilisant des branchements indirects (hors retours de fonction)

Cependant, ces deux cas sont peu fréquents dans les programmes présents dans les systèmes embarqués.

Nous avons présenté le résultats de ces travaux à la conférence internationale SIMUTOOLS'14 [BBBT14].

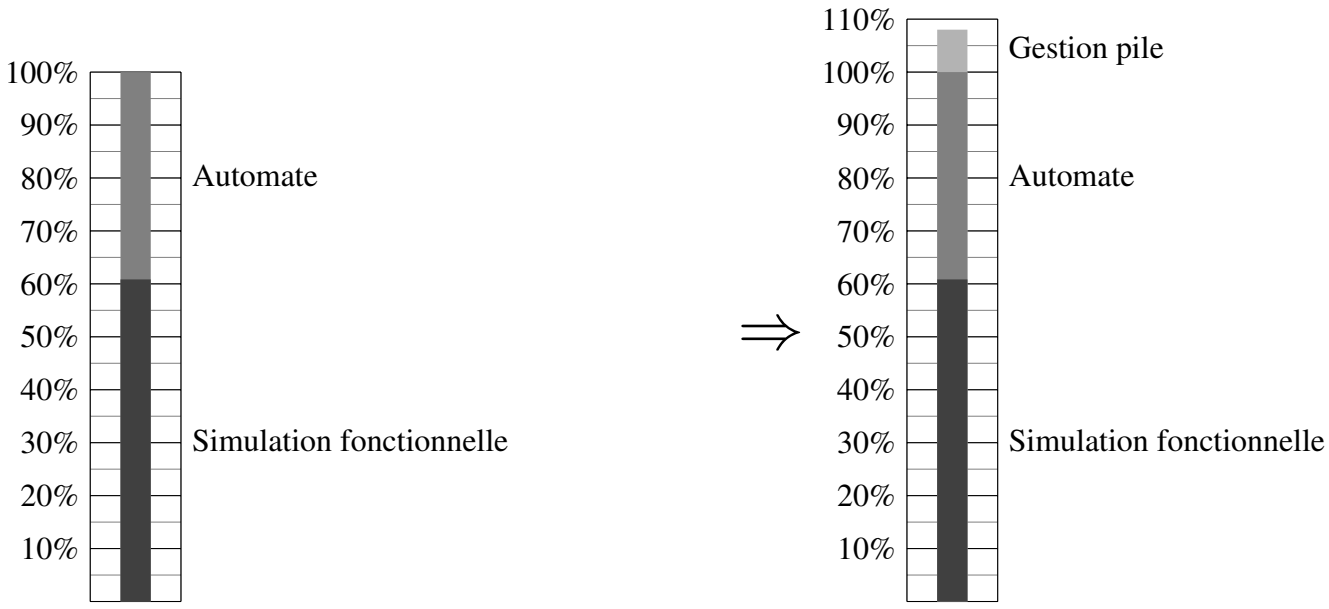

FIGURE 5.9 – Répartition moyenne des tâches restantes dans le temps d'exécution du simulateur CAS. La suppression des restrictions de programmes implique la manipulation d'une pile à l'exécution, augmentant le coût de traitement de l'automate de 8%.

6

Macro-instructions

6.1 Objectifs

Nos travaux ont permis de montrer que l'on pouvait déporter certaines tâches (gestion de la dépendance des données) de la phase d'exécution à celle de compilation, en intégrant le programme au modèle temporel. Cette opération a permis de réduire le temps d'exécution de près de 45%.

Deux tâches principales demeurent dans la phase d'exécution, comme on peut le voir sur la Figure 6.1 : la simulation fonctionnelle du programme et la gestion de l'automate.

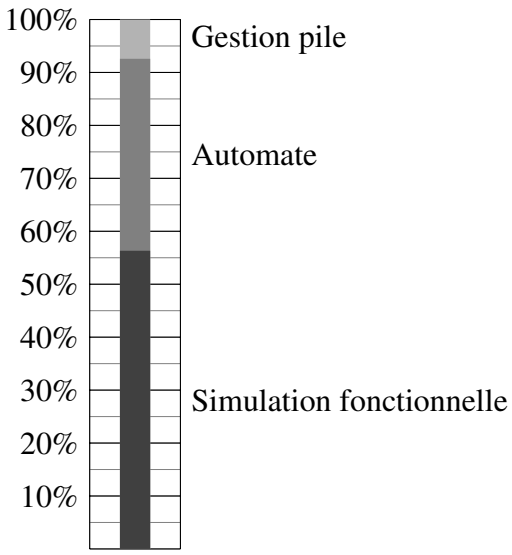

FIGURE 6.1 – Répartition des tâches restantes dans le temps d'exécution du simulateur CAS. La simulation fonctionnelle occupe 56,3% dans le temps d'exécution, la gestion de l'automate 36,3% et la gestion de la pile 7,42%.

Près de la moitié du temps d'exécution (42%) est consacrée à la gestion de l'automate. C'est l'objectif de la technique des macro-instructions d'améliorer la vitesse de cette tâche.

Nous expliquons son principe. À chaque itération du simulateur, une transition est tirée pour simuler un cycle. Le temps de calcul d'une transition est quasiment constant : interrogation de la structure de données des transitions et extraction des différentes étiquettes. Pour rentabiliser ce temps de calcul, il est possible de rassembler les informations de plusieurs instructions dessus. De cette manière simuler un certain nombre de cycles demande de franchir moins de transitions et donc d'économiser du temps.

Dans ce modèle, le programme est abstrait : une transition ne modélise plus une instruction, mais un bloc de plusieurs instructions. Nous utilisons le terme de macro-instructions pour ces blocs. Une séquence de transitions peut être réduite à une seule transition, contenant la concaténation de l'information des transitions élémentaires. Nous appelons une macro-transition une transition modélisant plusieurs cycles (voir Figure 6.2).

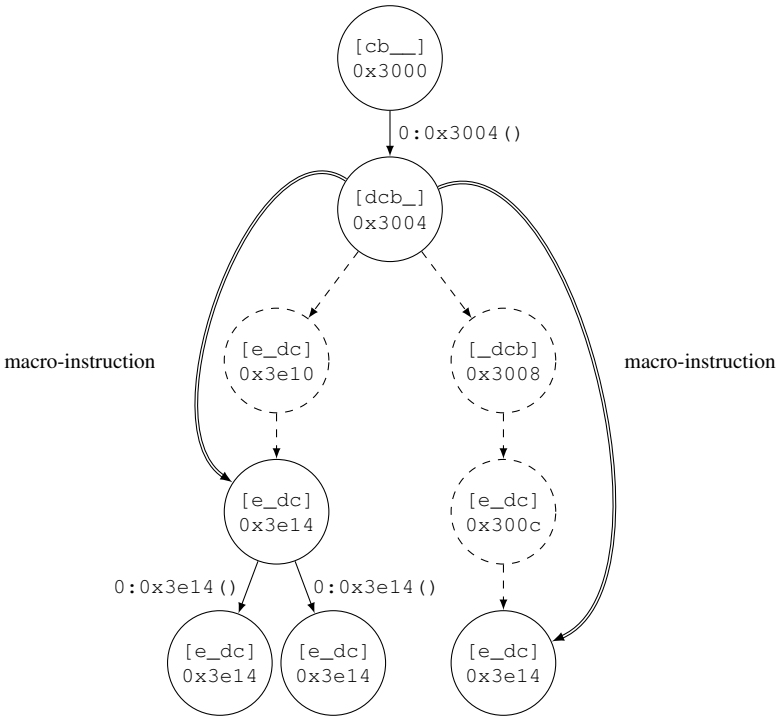

FIGURE 6.2 – Exemple de l'utilisation de macro-instructions : entre crochets, l'état du pipeline est représenté, le nombre en hexadécimal pointe l'adresse de l'instruction en cours. La macro-instruction sur la gauche représente 2 cycles processeurs, celle sur la droite 3 cycles processeurs.

6.2 Modélisation

L'intégration des macro-instructions ne demande pas de profondes modifications du modèle. Il est seulement nécessaire de gérer la concaténation des informations sur les étiquettes des transitions et d'en ajouter une : le nombre de cycles représentés par la macro-transition.

La principale difficulté réside dans la délimitation des macro-instructions. Pour comprendre les restrictions qui pèsent dessus, il faut observer finement le fonctionnement du simulateur. À l'exécution, l'exploration de l'automate est opérée en deux étapes successives, que l'on itère : d'abord le calcul des conditions (ressources externes et conditions indirectes) pour déterminer la transition à prendre, ensuite l'extraction des informations présentes sur l'étiquette de la transition. Ces informations permettent à leur tour de calculer les conditions de la prochaine transition. On réitère le processus pour cette prochaine transition.

Dans le cadre de cette procédure, il faut toujours pouvoir calculer les conditions d'une transition avant de la tirer. Si c'est un pré-requis acquis pour une transition modélisant un seul cycle processeur, ça ne l'est pas nécessairement pour une macro-instruction. Sur la Figure 6.3, déterminer quelle macro-transition emprunter peut nécessiter des informations présentes sur la macro-transition elle-même. Si la transition ② apporte les informations nécessaires à la résolution du branchement ③, alors il n'est pas possible de déterminer quelle macro-transition emprunter en ①.

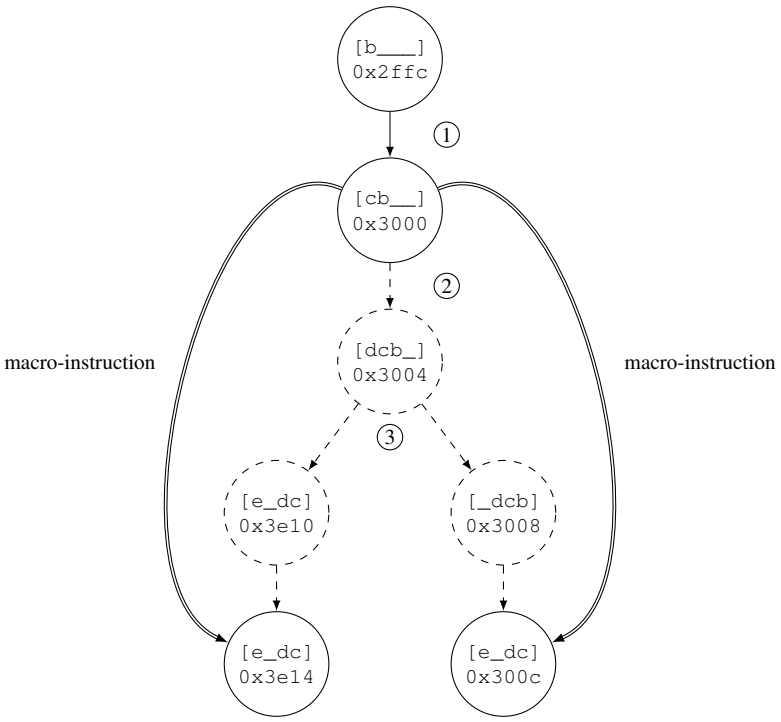

FIGURE 6.3 – Cas de figure d'un branchement présent au milieu d'une macro-instruction.

Ainsi, cette procédure nous limite au cas où les conditions pour déterminer la macro-transition sont indépendantes des informations présentes sur les étiquettes de la macro-transition. En effet, il faut pouvoir disposer de toutes les informations pour déterminer la séquence de transitions à tirer en amont de celle-ci.

C'est typiquement le cas pour une séquence linéaire de transitions : comme, à chaque fois, il n'existe qu'un seul successeur, calculer les conditions n'est pas nécessaire.

Pour les ressources externes, il n'est pas simple de déterminer à faible coût les dépendances entre étiquettes et conditions. Pour cette raison et dans un premier temps, nous nous limitons à une approche simple : celle de restreindre les macro-transitions aux séquences linéaires de transitions, comme on le voit sur la Figure 6.4. De plus, pour des raisons d'implantation, la taille d'une macro-instruction est nécessairement bornée.

Cette restriction présente l'avantage de réduire la taille du codage des transitions. En effet, pour déterminer la macro-transition suivante, il devient seulement nécessaire de connaître la disponibilité des ressources externes au début de la macro-transition, et non pas celle de chaque cycle.

Pour ce qui est des conditions indirectes, il est plus simple de déterminer les dépendances, car elles se font seulement entre empilement et dépilement. Il s'avère, cependant, que la restriction précédente rend vain tout raffinement sur les conditions indirectes. Pour cette raison, il a été décidé de privilégier une approche qui réduise la taille du codage des transitions.

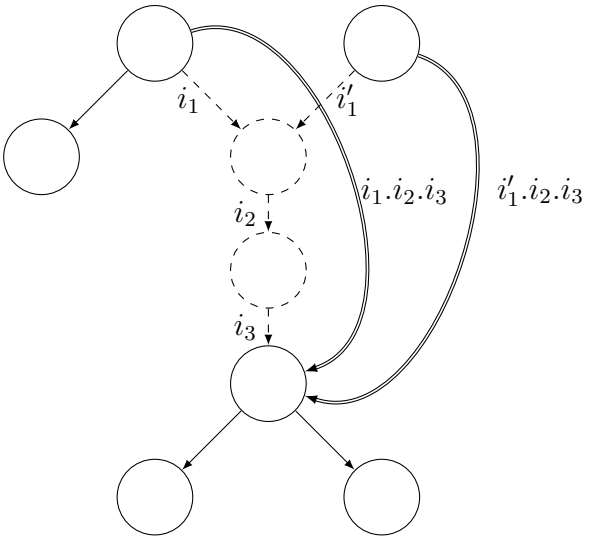

FIGURE 6.4 – $i_1.i_2.i_3$ est appelée une macro-instruction. Elle est délimitée par deux embranchements. On peut noter avec la macro-instruction $i'_1.i_2.i_3$ que les réunions ne sont pas une restriction, quitte à dupliquer les instructions i_2 et i_3 sur plusieurs transitions.

Ainsi, nous imposons comme règle qu'une macro-transition ne puisse procéder à plus d'un dépilement et plus d'un empilement. Une autre simplification dans le codage de la transition peut être appliquée. Il est fait abstraction du moment précis où ces opérations sont effectuées : elles sont reportées au début de la macro-transition. Pour conserver le comportement du modèle, seule la chronologie de ces évènements est importante (voir Figure 6.5). Nous imposons donc que le dépilement précède toujours l'empilement, sur une macro-transition.

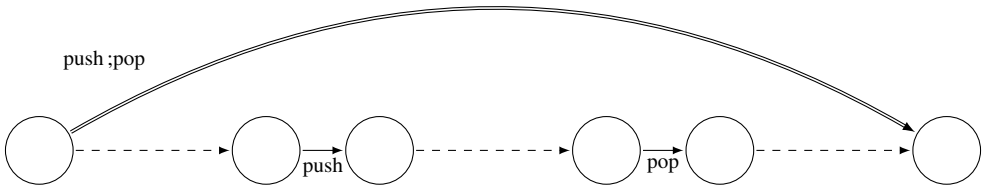

FIGURE 6.5 – Justification de la manipulation des piles en amont. À la fin de la macro-instruction, en double trait, l'état de la pile est identique.

Ainsi pour résumer, les conditions auxquelles on soumet les macro-instructions :

- chaque état intermédiaire n'a qu'un seul successeur ;
- il n'y a pas plus d'un dépilement sur la macro-instruction ;
- il n'y a pas plus d'un empilement sur la macro-instruction ;
- le dépilement précède toujours l'empilement sur la macro-instruction ;
- la taille des macro-instructions est bornée, en fonction de l'implantation.

Nous pouvons maintenant formaliser. Soit MA un automate défini par $\{S, s_0, IC, RE, NC, P, N, T\}$, où :

- S est l'ensemble des états ;
- s_0 est l'état initial (pipeline vide, PC initial) dans S ;
- IC est le premier alphabet d'actions (condition indirecte) ;
- RE est le second alphabet d'actions (ressource externe) ;
- NC est le premier alphabet d'étiquettes (nombre de cycles) ;
- P est le second alphabet d'étiquettes (identifiant à empiler) ;
- N est le troisième alphabet d'étiquettes (notifications) ;
- T est la fonction de macro-transition dans $S \times IC \times RE \times NC \times P \times N^* \times S$.

Si $(s, ic, re, nc, p, n, s') \in T$, nous notons $re : ic(n : p : nc)$ sur la transition de s à s'.

6.3 Contrôle du cache d'instructions

6.3.1 Objectifs

Le souhait de recourir à des macro-instructions appelle un travail préliminaire pour rendre la technique pleinement efficace. La réduction de l'automate ne peut se faire que si des chaînes de transitions se trouvent dans l'automate. Plus le taux de chaînes trouvées dans l'automate est grand et plus la réduction de l'automate peut être forte. Dans le but de rendre le gain des macro-instructions optimal, il est donc nécessaire de limiter les embranchements intervenant dans l'automate.

Dans le cas général, un embranchement survient lors de l'utilisation d'une ressource externe. En effet, l'utilisation d'une ressource externe produit deux successeurs : si la ressource externe est prise ou libre, l'évolution du pipeline est bloquée ou non. En limitant l'utilisation des ressources externes, on facilite le travail de réduction.

La ressource externe pour le cache d'instructions entraîne un embranchement dans l'automate à chaque fois qu'une instruction est chargée. Ce fonctionnement restreindrait la taille des macro-instructions de façon drastique. C'est pour cette raison que nous cherchons une manière d'optimiser la modélisation du cache d'instructions.

Quelle piste d'optimisation du modèle du cache d'instructions ? Dans cette optique, nous nous intéressons à la modélisation de la gestion du cache d'instructions, qui présente un intérêt particulier.

Le cache d'instructions permet le chargement rapide des instructions depuis la mémoire. Lorsqu'une instruction est appelée, l'instruction est chargée dans le cache avec toute la ligne de mémoire qui lui correspond. Ainsi, si l'instruction suivante se trouve sur la même ligne de cache, il est inutile de procéder à un nouvel appel mémoire. (On parle de *hit*, pour le cas où l'élément recherché se trouve dans le cache, et *miss* dans le cas contraire.) Cette technique s'appuie ainsi sur les propriétés de temporalité et de localité d'un programme.

La zone mémoire associée au cache d'instructions est modélisée par le simulateur. Durant l'exécution, le simulateur reproduit le fonctionnement de cette zone mémoire au cycle près[1]. Il permet ainsi de répondre facilement si une instruction est présente ou non dans le cache. Si elle

[1]La modélisation de la mémoire cache a déjà été réalisée dans le cadre de HARMLESS mais cela dépasse le contexte de cette thèse.

ne l'est pas, le nombre de cycles nécessaires pour la charger est calculé. Durant cette intervalle de temps, la ressource externe, qui lui est attribuée, est considérée comme occupée. Le pipeline est ainsi bloqué le temps du chargement de l'instruction.

Cette ressource externe a un grand impact sur l'automate, car chaque instruction l'utilise. Elle est également systématiquement utilisée, à chaque entrée dans le pipeline. Cela rend impossible toute réduction de l'automate.

C'est la raison pour laquelle, nous proposons d'alléger le travail de cette ressource externe.

Attention : Contrairement au travail qui a été fait sur la dépendances de données, il ne s'agit pas, ici, de supprimer ou réduire le temps de calcul nécessaire à la ressource externe, mais seulement de réduire son impact sur la complexité de l'automate.

6.3.2 Principes

Durant la compilation, il est possible de vérifier si l'instruction qui doit être chargée se trouve sur la même ligne du cache d'instructions que l'instruction précédente.

Si l'instruction est sur la même ligne de cache que la précédente (et cette information peut être déterminée statiquement), nous pouvons en déduire que l'accès au cache fera un *hit*. Cela permet d'élaguer l'automate en supprimant les transitions qui ne pourront pas être prises à l'exécution.

Dans le cas contraire, rien n'est assuré. En effet, il pourrait s'agir d'un *miss*, mais également d'un *hit* si l'instruction se trouve sur une autre ligne du cache déjà chargée. Cette ambiguïté n'est pas soluble, sauf à simuler le comportement du cache, ce qui nécessiterait d'obtenir des informations dynamiques sur l'exécution. De plus, dans le cas d'un *miss*, il n'est pas possible de déterminer statiquement le temps que prendra le chargement de l'instruction, *i.e.* le nombre de bulles à introduire dans le pipeline.

Ces deux points rendent impossible une gestion purement statique de la ressource externe.

Il reste néanmoins possible d'effectuer quelques simplifications dans l'automate. Si l'on connaît les cas de *hit*, on peut retirer du modèle des états inaccessibles.

6.3.3 Impact théorique sur la taille du modèle

Pour évaluer la réduction maximale du modèle que peuvent produire les macro-instructions, nous nous intéressons à un cas de figure idéal, c'est-à-dire qui maximise l'utilisation des macro-instructions. Ce cas de figure est celui d'un programme limitant au maximum les embranchements dans l'automate : il est linéaire, il ne comporte aucun aléa de données. Tant et si bien que la seule cause d'embranchement possible est la modélisation du cache d'instructions.

Avec la simplification qui a été opérée dans cette section, la modélisation du cache d'instructions génère un embranchement dans l'automate à chaque nouvelle ligne de cache. On obtient donc un automate de forme linéaire, sur lequel on retrouve à chaque nouvelle ligne de cache le motif présenté sur la Figure 6.6 (où l'on considère un pipeline à seulement 3 étages).

Construisant les macro-instructions sur ce cas idéal, on obtient ce que l'on peut voir en Figure 6.7.

On peut ainsi en déduire le taux de réduction maximal que permettent les macro-instructions. Soient e le nombre d'étages du pipeline et t la taille de la ligne de cache.

Avant réduction, pour un motif, on peut énumérer les transitions, elles sont au nombre de :

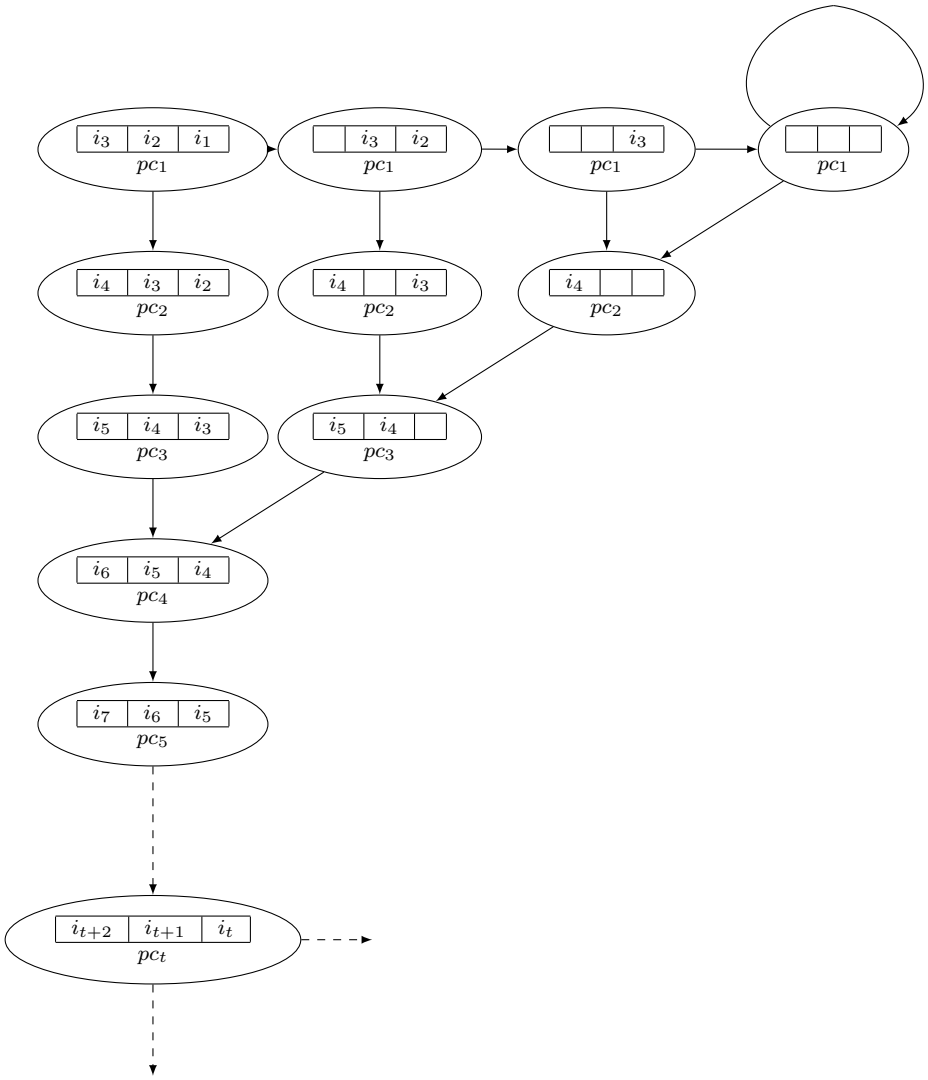

FIGURE 6.6 – Dans le cas d'un programme idéal (linéaire, sans aléas de données), le modèle est seulement complexifié par la gestion du cache d'instructions. L'automate généré prend la forme d'une succession du motif présenté, réitéré toutes les t instructions (avec t le nombres d'instructions par ligne de cache).

$$(t-e) + 2e + 1 + \sum_{k=1}^{e} k \tag{6.1}$$

Pour détailler cette énumération, dans le premier terme $(t-e)$ on trouve le nombre de transitions qui relient chaque motif. Dans le second terme $(2e)$, on retrouve les deux côtés du motifs (du dessus et du dessous). Le troisième terme (1) compte la boucle. Enfin, la somme énumère les transitions internes au motif.

Si l'on réduit l'automate grâce aux macro-instructions, on ne dénombre que :

$$2e + 2 \tag{6.2}$$

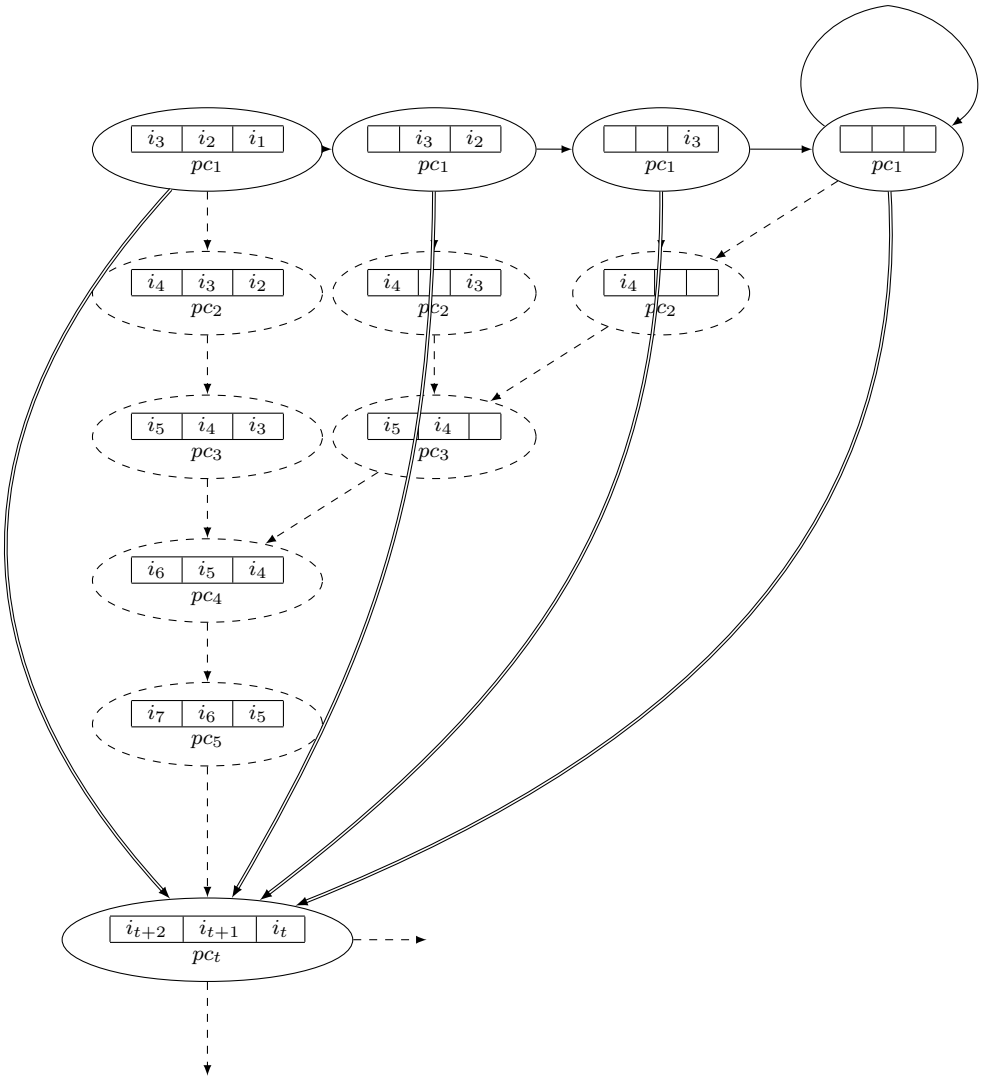

FIGURE 6.7 – Dans le cas d'un programme idéal (linéaire, sans aléas de données), les macro-instructions se développent entre chaque chargement du cache. Ces macro-instructions ont une taille égale à celle de la ligne de cache (t).

Le taux de réduction est la différence du nombre de transitions sur le nombre initial. Il se chiffre, par conséquent, à :

$$\frac{(t-e)+2e+1+\sum_{k=1}^{e} k-2(e+1)}{(t-e)+2e+1+\sum_{k=1}^{e} k} \tag{6.3}$$

Ce qui est équivalent à l'expression suivante :

$$\frac{e^2-e+2t}{e^2+e+2t+2} \tag{6.4}$$

Si l'on prend le modèle que nous utilisons pour les tests, avec un pipeline à 5 étages ($e = 5$) et une ligne de cache de 8 instructions ($t = 8$), on obtient une réduction maximale de l'ordre de 75%.

6.3.4 Algorithme

Le cache d'instructions utilise les bits de poids faible du PC pour déterminer quelle ligne de cache est utilisée. De cette manière, il est simple de comparer ce champ de bits statiquement pour savoir si l'instruction qui suit est sur la même ligne de cache ou non.

De façon générale, pour une ligne de cache de taille 2^n octets, il faut s'assurer que l'adresse des instructions ne diffère que sur les n premiers bits de poids faibles. Si c'est le cas, les instructions appartiennent à la même ligne de cache.

Cette condition est introduite dans la procédure de calcul du nouveau pipeline. À chaque étage, il est déterminé si un conflit avec les ressources externes et internes empêche l'évolution du pipeline. Lorsque la ressource externe considérée est celle du cache d'instructions et que la condition est vérifiée on pose qu'elle est nécessairement disponible.

6.4 Algorithme

6.4.1 Réduction de l'automate

La procédure de réduction de l'automate se fait en recherchant les séquences de transitions réductibles et en les concaténant.

Sur la Figure 6.8, nous donnons le schéma de construction d'une macro-instruction. Il est supposé que la première transition dépile ic_1, et que la seconde transition empile p_2. L'état intermédiaire n'a qu'un successeur. Ainsi, les conditions sont réunies pour construire une macro-transition.

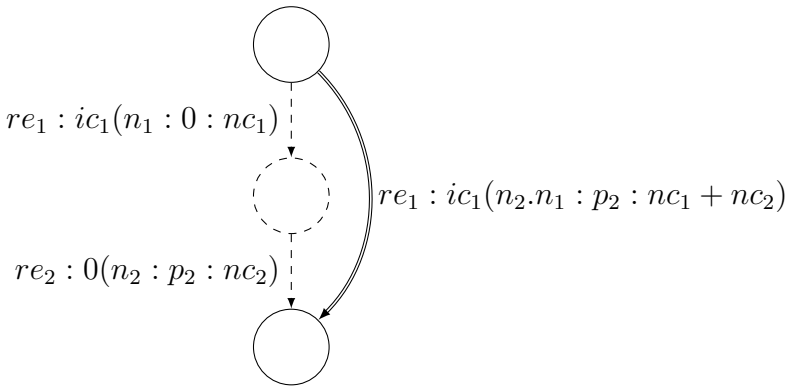

FIGURE 6.8 – La construction d'une macro-transition

La condition de concaténation est :

$$un\ seul\ successeur \wedge (ic_1 = 0 \vee ic_2 = 0) \wedge (p_1 = 0 \vee (ic_2 = 0 \wedge p_2 = 0))$$

Si cette condition est remplie, il est possible de concaténer les deux transitions. La construction de la macro-transition résultante suit les règles suivantes :

- toutes les transitions en entrée sont remplacées par une macro-transition partant de l'état prédécesseur jusqu'à l'état successeur ;
- la ressource externe est celle de la première transition ;
- la notification est la concaténation des deux notifications ;

- la condition indirecte est celle de la première transition, si elle existe, sinon de la deuxième ;
- l'élément à empiler est celui de la deuxième transition, s'il y en a un, sinon de la première ;
- le nombre de cycles est la somme de ceux des deux transitions.

La réduction se fait en parcours successifs de l'automate. Dans chaque parcours, les transitions que l'on peut concaténer le sont. Le parcours est recommencé tant que des concaténations sont effectuées, jusqu'à ainsi obtenir un point fixe. Ce point fixe est nécessairement atteint, puisque la taille des macro-instructions, croissante, est bornée par la taille de la plus grosse chaîne de transitions initiales.

6.4.2 Exécution

L'exécution de l'automate est opérée comme précédemment, à ceci près que les opérations pour chaque transition sont itérées plusieurs fois. Pour chaque cycle modélisé par la transition, le simulateur extrait les notifications associées au cycle, puis décode et exécute des instructions. C'est ce que l'on peut voir sur l'algorithme 8.

Algorithme 8 Exécution d'un cycle

```
1: Calcul des ressources externes
2: Calcul du successeur
3: Traitement de la pile
4: Extraction du nombre de cycles (n) de la macro-instruction sur la transition
5: for i allant de 1 à n do
6:     Extraction des notification du cycle i sur la macro-instruction
7:     Décodage d'une instruction
8:     Exécution d'une instruction
9: end for
```

6.5 Résultats

Nous simulons à nouveau une architecture similaire au PowerPC 5516 de Freescale, avec un cœur *e200z1*. Le pipeline est redimensionné de 4 à 5 étages pour augmenter la taille du modèle. Nous exécutons les benchmarks de Mälardalen [GBEL10]. Comme présenté dans la section 3.3.2, les simulations sont faites sur un Intel *Core i7@3,4GHz*. Chaque programme est exécuté 50 000 fois pour faciliter les mesures temporelles.

6.5.1 Taille des automates

L'utilisation de macro-instructions permet d'abord une réduction du modèle. Les macro-instructions sont principalement limitées par les embranchements dans l'automate. Ceux-ci sont causés par les ressources externes. Dans le meilleur des cas, *i.e.* pour un flot de contrôle linéaire sans dépendances de données, un embranchement apparaît à chaque fois que le bord d'une ligne du cache d'instruction est franchie. Dans notre exemple, une ligne du cache d'instructions contient 8 instructions. Par conséquent, la réduction est bornée à 75% (pour un pipeline à 5 étages une ligne de cache à 8 instructions). Dans le Tableau 6.1, nous observons une réduction de 47,1% en moyenne.

Programme	# états sans macro-instructions	# états avec macro-instructions	Gain
adpcm	12 096	5 729	52,6%
bs	587	317	46%
basicMathVerySmall	4 302	2 588	39,8%
compress	5 059	2 370	52,8%
cover	1 123	599	46,7%
crc	2 038	1 060	48%
duff	669	378	43,5%
expint	1 395	764	45,2%
fdct	3 707	1 984	46,5%
fibcall	479	240	49,9%
fir	1 016	574	43,5%
janne_complex	645	356	44,8%
jfdctint	3 134	1 727	44,9%
lcdnum	568	279	50,9%
matmult	1 516	863	43,1%
ndes	6 114	3 239	47%
ns	837	471	44,4%
prime	1 138	566	50,3%

TABLE 6.1 – Influence des macro-instructions sur la taille du modèle

6.5.2 Temps d'exécution

La réduction du temps d'exécution n'est pas nécessairement du même ordre que celle de l'automate. En effet, elle dépend avant tout de la réduction que permettent les macro-instructions qui seront rencontrées lors de l'exécution.

La technique des macro-instructions réduit le temps alloué à la manipulation de l'automate. Il permet d'améliorer la réduction du temps d'exécution de 53% en moyenne, avec un maximum de 57%, comme nous pouvons le voir sur la Figure 6.9. La Figure 6.10 montre une large comparaison entre les temps d'exécutions, spécialement avec l'ISS. Cette comparaison montre la réduction sur la part spécifique du CAS, et quelle amélioration reste possible sans améliorer la vitesse de l'ISS.

6.6 Conclusion

Dans ce chapitre, nous avons introduit les macro-instructions qui consistent à un rassemblement d'instructions dans un seul bloc. Cette technique permet d'augmenter le nombre de cycles par lequel avance le simulateur à chaque évolution du modèle temporel.

Nous avons également apporté une optimisation dans la gestion du cache d'instructions qui permet aux macro-instructions d'être plus efficaces.

Ces deux contributions conduisent à une réduction de la taille de l'automate de 47% en moyenne, ainsi qu'à une réduction du temps d'exécution de 53%, en comparaison avec le simulateur interprété (voir Figure 6.11).

Cette technique a été présentée à la communauté scientifique dans un article la regroupant au deuxième modèle, à la conférence internationale SIMUTOOLS'14 [BBBT14].

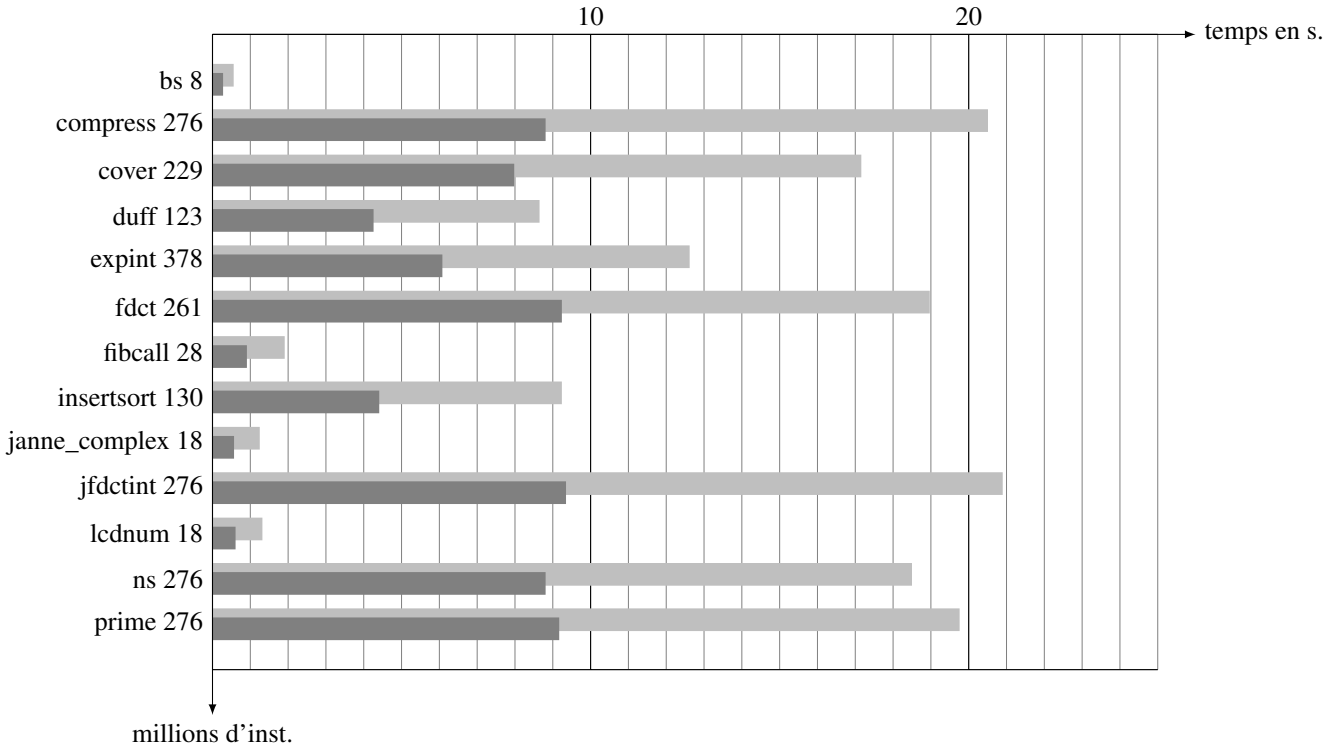

FIGURE 6.9 – Comparaison du temps d'exécution en secondes pour 50 000 exécutions. Le gris est pour la simulation interprétée, et le noir est pour la simulation compilée.

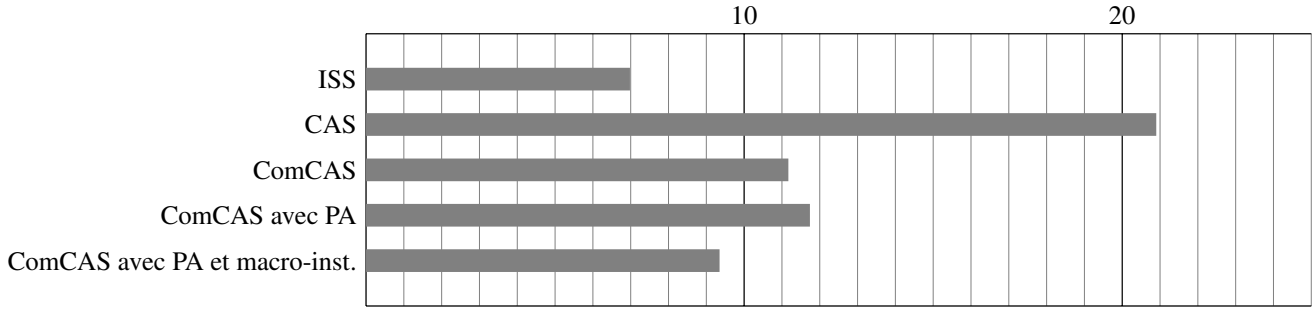

FIGURE 6.10 – Comparation du temps d'exécution de différentes techniques de simulation en secondes pour jfdctint. Si nous considérons seulement la part spécifique du CAS, notre modèle amène une réduction de 83%.

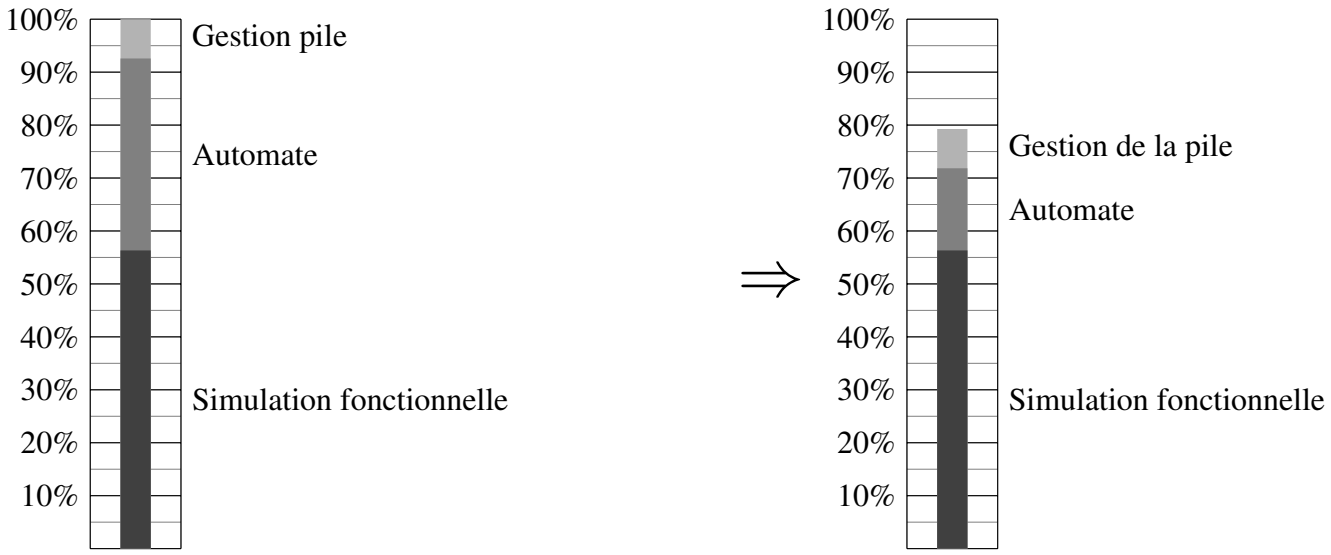

FIGURE 6.11 – Répartition des tâches dans le temps d'exécution du simulateur CAS. L'utilisation des macro-instructions réduit de près de la moitié le temps de gestion de l'automate.

7

Abstraction du code simulé

7.1 Objectifs

À la suite de l'analyse du temps d'exécution de la simulation de la Figure 7.1, on note que la tâche associée à la simulation fonctionnelle est à présent fortement prépondérante sur celle gérant l'automate. Dans une perspective de réduction du temps d'exécution, il devient pertinent de concentrer ses efforts pour réduire celle-ci. Ce chapitre y est consacré, en présentant une technique d'abstraction du code, nommée le *slicing*.

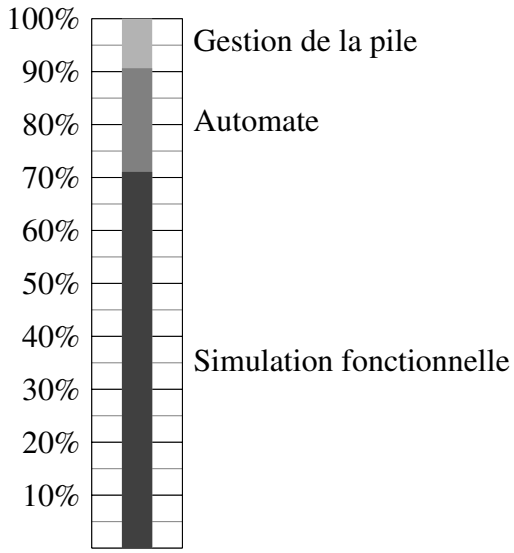

FIGURE 7.1 – Analyse du temps d'exécution : la simulation fonctionnelle occupe à présent 71% du temps d'exécution, la manipulation de l'automate 19,5%.

Comme on peut le voir sur l'algorithme 9, qui représente les opérations effectuées à chaque cycle de la simulation : à chaque transition, si une instruction entre dans le pipeline, elle est décodée puis exécutée. Les informations qui sont récupérées de cette exécution servent au calcul du moteur de simulation.

Algorithme 9 Boucle de simulation interprétée

```
1: Calcul des ressources externes                    ▷ Optimisé par la simulation compilée
2: Recherche de l'état successeur                    ▷ Optimisé par les macro-instructions
3: if entrée d'une instruction dans le pipeline then
4:     Exécution d'une instruction
5:     Décodage de la prochaine instruction          ▷ Optimisé par un cache
6: end if
7: Gestion des aléas de données                      ▷ Pré-calculé par la simulation compilée
```

Si chaque étape a été étudiée pour réduire son coût, que cela soit au cours de cette thèse ou avant elle (comme l'utilisation du cache dans le décodage), une des tâches ne l'a pas encore été : l'exécution des instructions.

Dans le cadre des systèmes temps réel, on peut se retrouver seulement intéressé par les résultats temporels de la simulation. Dans un tel cas, il devient envisageable et profitable de se priver des résultats fonctionnels pour économiser du temps d'exécution, dans la mesure où l'on reste capable de préserver les informations temporelles.

L'exécution des instructions ne se montre pas seulement importante pour obtenir des résultats fonctionnels, elle permet également de déterminer certaines informations dynamiques, nécessaires au calcul du flot de contrôle (le caractère pris ou non d'un branchement conditionnel, par exemple). Pour préserver les informations temporelles, il est essentiel de pouvoir effectuer le même chemin dans l'automate. Pour que cela soit possible, il faut être capable de distinguer les instructions liées au flot de contrôle de celles liées aux données. Les premières sont fondamentales pour préserver le flot de contrôle et obtenir des informations sur les temps de calcul. Les secondes ont un intérêt limité à la simulation fonctionnelle : elles garantissent la cohérence des valeurs obtenues.

On retrouve ainsi la technique du *slicing*, telle qu'elle est utilisée dans le calcul du WCET [BC11].

Principe Le *slicing* est une technique d'abstraction du code. Elle consiste dans l'abstraction des instructions qui n'influencent pas le calcul de certains registres prédéfinis. Ainsi, il est possible d'exécuter le code sans ces instructions tout en conservant la cohérence des résultats sur ces registres particuliers.

Cette technique est utilisée notamment dans le calcul du pire temps d'exécution (WCET) d'un programme. Il suffit de désigner comme registres à calculer ceux qui servent à la détermination du flot de contrôle. Ces instructions peuvent être alors simulées simplement comme un temps d'attente, correspondant à son temps de calcul, sans fausser le calcul du WCET.

Ces instructions sont déduites de la détermination des dépendances de données. Il est pris comme situation initiale un ensemble de données (ou registres) dont on souhaite préserver la cohérence, à un endroit précis du code. Sur cette base, le flot de contrôle est remonté. Lorsqu'une instruction influe sur cet ensemble de données, elle est ajoutée au *slice* : il faut l'exécuter pour assurer le résultat final. Lorsqu'une telle instruction fait appel à d'autres données, elles sont ajoutées au processus : il faut alors préserver la cohérence de leurs résultats à cet endroit du code.

7.2 État de l'art

7.2.1 Bibliographie

Le calcul des instructions influençant les registres a été l'objet d'une importante littérature. Elle a commencé avec Weiser [Wei84]. De nombreux algorithmes ont été proposés pour améliorer ce calcul. Cela passe par la recherche de *dominators* dans le flot des dépendances de données : [LT79], [CHK01] et [GTW06]. Les *dominators* représentent les points de passage obligatoires dans le parcours d'un graphe.

L'utilisation de la technique du *slicing* pour l'optimisation du calcul du WCET est une contribution de Béchennec et Cassez [BC11]. Elle consiste à modéliser les informations n'intervenant pas dans le flot de contrôle comme de simples temps d'attente. De cette manière, la modélisation du programme est allégée tout en préservant le WCET.

Pour calculer le pire temps d'exécution, Béchennec et Cassez utilisent un double modèle : un premier pour le programme, son flot de contrôle, et un deuxième pour l'architecture sur lequel il est exécuté. Ces modèles s'expriment sous la forme d'automates. Ils sont synchronisés pour modéliser le système complet.

Il est possible alors d'utiliser les outils du *model-checking* pour obtenir les informations sur le temps d'exécution. C'est ce qui est fait grâce à l'outil UPPAAL [1].

Dans ce cadre, Béchennec et Cassez utilisent la technique du *slicing* pour réduire la modélisation du programme. Elle permet notamment de :

- calculer rapidement le flot de contrôle du programme,
- et déterminer un programme réduit, mais équivalent du point de vue du WCET.

Pour le premier objectif, la technique consiste à dégager l'ensemble minimal d'instructions à exécuter pour déterminer la cible des branchements. Cette opération est effectuée en dépliant progressivement le flot de contrôle, ce qui n'est possible qu'à la condition que les branchements puissent être calculés. Lorsqu'ils ne le peuvent pas, la technique du *slicing* déterminent les instructions nécessaires pour cela. Une exécution rapide calculant les branchements est possible sur cette base, et le dépliage peut se poursuivre.

Une fois le flot de contrôle construit, le calcul d'un programme réduit peut se faire par la technique du *slicing*. Dans ce calcul, la détermination des branchements n'est plus nécessaire, puisqu'ils l'ont été dans l'étape précédente.

Exemple Pour bien comprendre le principe de l'abstraction du code opérée par le *slicing*, on donne l'exemple de la Figure 7.2. Le programme permet le calcul de la 20^{e} itération de la suite de Fibonacci ($u_{n+2} = u_{n+1} + u_n$). Le registre r1 contient le numéro d'itération (n), tandis que les registres r2 et r3 contiennent les valeurs successives de la suite (u_{n-1} et u_n).

Pour parcourir le flot de contrôle de ce programme, il est inutile de calculer les valeurs de la suite. Seul le calcul du nombre d'itérations est nécessaire (n). C'est ce qui amène à ne retenir que les instructions en gras.

7.2.2 Analyse

La méthode mise en place par Béchennec et Cassez permet bien de fournir le résultat qui conviendrait au simulateur. Il utilise néanmoins des éléments qui ne sont pas disponibles dans le cadre de notre implantation.

[1] uppaal.org

inst. n°	mnémo	comportement	algo
1	**mv r1,#1**	r1 ← 1	$n = 1$
2	mv r2,#1	r2 ← 1	$u_{n-1} = 1$
3	mv r3,#1	r3 ← 1	$u_n = 1$
4	mv r4,r3	r4 ← r3	$x = u_n$
5	**add r1,r1,#1**	r1 ← r1+1	$n = n + 1$
6	add r3,r3,r2	r3 ← r3+r2	$u_n = u_n + u_{n-1}$
7	mv r2,r4	r2 ← r4	$u_{n-1} = x$
8	**cmp r1,20**	r ? ← r1 < 20	$n < 20$?
9	**bc 4**		

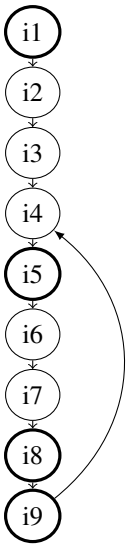

FIGURE 7.2 – Exemple de *slicing* avec une suite de Fibonacci. Les instructions en gras sont dans le *slice*. La méthode proposée consiste à n'exécuter que ces instructions pour accelérer la simulation et obtenir les temps de calcul de l'algorithme.

Tout d'abord, nous nous plaçons dans un cadre générique, tandis que le modèle de Béchennec et Cassez est dédié à une architecture particulière.

Ensuite, dans le cadre de la simulation compilée, lors de la compilation, on ne dispose pas d'informations dynamiques. C'est-à-dire que l'on ne se permet pas d'exécuter le programme ou toute instruction. Cette restriction nous empêche :

- de calculer la cible d'un branchement indirect ;
- de déterminer finement les dépendances de données transitant par la mémoire, qui ont besoin de calculs pour déterminer les emplacements mémoire visés.

La méthode utilisée pour construire le flot de contrôle n'est donc pas applicable dans notre cas. Cependant, le premier point ne pose pas de problème particulier, puisque nous disposons déjà, grâce aux modèles précédents, des outils nécessaires à la construction du graphe. Notamment, nous pouvons calculer la cible de branchements indirects particuliers : les retours de fonction, en simulant le comportement de la pile des appels.

Principalement à cause des accès à la mémoire, il n'est pas possible de remonter les dépendances de données de façon aussi optimale que dans le modèle de Béchennec et Cassez.

Comme on peut le voir sur l'algorithme 10, une donnée peut transiter dans un emplacement mémoire. Cet emplacement mémoire est déterminé par la somme du contenu d'un registre et d'un *offset*. Sans pouvoir calculer le résultat de cette somme (qui nécessite de connaître le contenu d'un registre) il est impossible de localiser l'emplacement mémoire concerné et donc de retrouver les écritures mémoires correspondantes.

Pour résoudre ce problème, nous proposons dans la section suivante de nouveaux modèles.

7.3 Modélisation

Dans cette section, nous allons explorer une approche qui nous a paru la plus raffinée pour le traitement de la mémoire. Mais avant cela, nous allons détailler comment fonctionne le *slicing*.

Algorithme 10 Exemple d'utilisation de la mémoire

```
1: stw r0,r1,8          ▷ Mem[r1+8] ← r0
2: ...
3: ldw r3,r2,8          ▷ r3 ← Mem[r2+8]
4: cmp r3,0
5: bc 0x320c
```

Comment fonctionne le *slicing* ? La technique du *slicing* prend pour entrée ce qu'on appelle un *slice criterion*. Il se compose d'un ensemble de couples :

- une instruction pour point de départ au *slicing* ;
- les registres demandés pour cette instruction.

Sur cette base, le calcul revient à remonter depuis ces instructions le flot de contrôle et à faire jouer les dépendances de données pour calculer pour chaque instruction les registres demandés. Si une instruction affecte un des registres demandés, elle est incluse au *slice*.

La Figure 7.3 propose un exemple. Supposons que le registre r1 serve à déterminer un branchement dans une instruction ultérieure, nous souhaitons donc préserver sa cohérence à l'instruction ⑤.

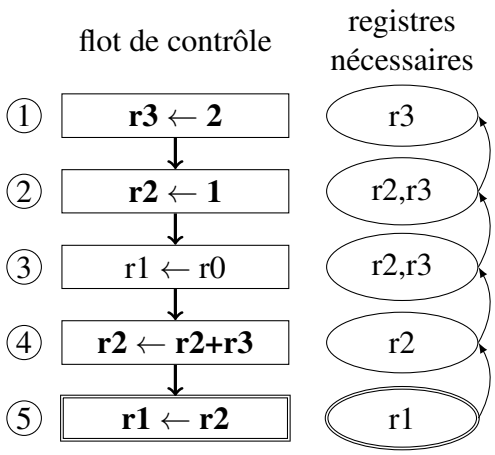

FIGURE 7.3 – Fonctionnement du *slicing*. Le flot de contrôle est à gauche. Les registres nécessaires à droite. On considère que le registre r1 doit être connu en 5, c'est le *slice criterion*, *i.e.* l'état initial du *slicing*. Les instructions en gras permettent d'obtenir la valeur de r1 et doivent donc faire partie du *slice*.

Sur cette figure, on peut voir que, par exemple, l'instruction ④ ajoute r3 à l'ensemble des registres nécessaires (le registre r2 étant écrit et lu, il est conservé).

On note que l'instruction ③ n'entre pas dans le *slice*. En effet, au moment de l'instruction, le registre r1 n'est plus un registre nécessaire au flot de contrôle.

Dans le cas de branchements, la remontée des dépendances de données est similaire : les dépendances de données sont explorées en remontant dans chaque branche.

Dans le cas d'une boucle, la question de la terminaison du programme peut se poser. Cependant, un point fixe est nécessairement trouvé (en moins d'itérations que la boucle ne contient d'instructions). En effet, dans le pire des cas, une instruction de la boucle est ajoutée au *slice* par itération et le nombre d'instructions dans une boucle est borné.

Utilisation pour le flot de contrôle Il faut être capable de déterminer quelles instructions influent sur le flot de contrôle. Les branchements conditionnels, puisque leur cible est déterminée en fonction d'un registre (celui contenant le résultat de la condition), nécessitent la bonne exécution de toutes les instructions qui permettent le calcul de ce registre. Celles-ci s'obtiennent en remontant les dépendances de données.

La remontée des dépendances de données s'opèrent comme suit. Dans chaque état du flot de contrôle est associé un ensemble de registres : les registres nécessaires à la bonne exploration du flot de contrôle. Partant de l'état initial, le *slice criterion*, on remonte l'ensemble des registres nécessaires dans le flot de contrôle. Si une instruction écrit dans un des registres nécessaires, celle-ci est incluse dans le *slice*. De plus, dans l'ensemble des registres nécessaires, on retire le ou les registres écrits, pour le remplacer par l'ensemble des registres lus par l'instruction. C'est ce qui arrive lors de l'écrasement d'une ancienne valeur, dont on ne souhaite pas préserver la cohérence.

Problème de la gestion des accès mémoire Dans les moyens que nous avons mis à disposition pour effectuer la simulation compilée, il n'est pas possible d'avoir des informations dynamiques sur le programme. Le problème se pose avec les accès mémoire, par lesquels peuvent transiter les dépendances de données.

Pour résoudre cette difficulté, nous avons proposé plusieurs modèles du plus grossier au plus fin :

- considérer toute la mémoire, comme une unique ressource ;
- distinguer dans la mémoire la pile des appels ;
- repérer les cas où les registres d'accès à la mémoire ne sont pas modifiés.

Le raffinement de ce modèle a été un enjeu important pour réduire le nombre d'instructions dans le *slice*.

Modèle final Dans la littérature, on trouve une branche qui s'intéresse au calcul de la correspondance entre lecture et écriture mémoire : la *memory disambiguation*. L'objectif de ce champ de recherche est de repérer les dépendances de données pour éviter des incohérences dans le cas d'une exécution dans le désordre. En l'état de nos connaissances, celle-ci ne dégage cependant pas de techniques pour résoudre notre problème de façon purement statique.

Si, en toute généralité, le calcul des emplacements mémoire nécessite des informations dynamiques (le contenu des registres), il existe des situations où une analyse statique peut suffire.

Par exemple, si une écriture en mémoire se fait à l'aide d'un registre rX et d'un offset Y, on peut être assuré que la lecture de la mémoire dans le registre rX avec l'offset Y correspond à l'écriture en question à la condition que le registre rX n'ait pas été modifié entre les deux opérations.

Cependant, selon toute généralité, rien n'empêcherait que la lecture en mémoire corresponde davantage à une écriture mémoire, moins en amont, avec un couple (rX,Y) différent, mais dont l'adresse effective donnerait le même résultat. Dans notre première approche nous nous en tenons à cette limitation. Le but étant d'obtenir un recouvrement minimal des instructions dans le *slice*, nous nous prêtons alors à toutes les hypothèses restrictives qui nous le permettent.

Cette propriété permet de raffiner la gestion de la mémoire dans une certaine mesure. Du moment que le registre, qui sert de base au calcul de l'adresse effective en mémoire, est modifié, il n'est plus possible avec les outils à disposition de déterminer l'écriture mémoire correspondante. Pour ne pas perdre l'écriture en mémoire, il redevient alors nécessaire de considérer la mémoire comme une unique ressource.

Les fonctions Pour optimiser le nombre de correspondances détectables, on peut s'intéresser à un autre procédé. La gestion de la mémoire se fait en grande partie via un espace particulier : la pile de contexte d'une fonction. À chaque appel d'une fonction, un espace est alloué pour stocker les variables internes à la fonction. Le processus gère alors un registre de pile, qui sert de références pour les appels à l'emplacement mémoire d'une variable interne (typiquement, les emplacements mémoire sont la somme du registre de cette pile et d'un *offset*). À l'appel d'une fonction, ce registre s'incrémente d'une certaine valeur (dépendante de l'architecture) pour offrir l'espace mémoire nécessaire aux variables internes. Au retour de la fonction, ce registre est décrémenté.

Les programmes utilisent souvent des alias du registre de pile. Ceux-ci peuvent facilement être reconnus : il sont initialisés à l'aide du registre de pile ou d'un alias de celui-ci.

De cette manière, si une fonction est appelée entre deux accès mémoire (écriture puis lecture correspondante) le registre de référence va être incrémenté (à l'appel de la fonction) et décrémenté de la même valeur (au retour de la fonction). Ces opérations sur le registre peuvent donner l'impression qu'il n'est plus sûr que la lecture corresponde bien à l'écriture, or c'est assuré. Pour intégrer cette connaissance, on gère les emplacements mémoire correspondant à la pile de contexte dans un système de pile.

À chaque appel de fonction, un nouvel ensemble de couples (registre, *offset*) est empilé. Il est dépilé au retour de la fonction. Avec un pareil procédé, l'appel d'une fonction ne perturbe plus la recherche de correspondance entre lecture et écriture mémoire.

7.4 Algorithme

Dans cette section, on donne les bases pour mettre en œuvre cette technique.

7.4.1 Calcul du slicing

Dans un premier temps, on calcule les instructions qui se trouvent dans le *slice*. Ce calcul se fait en deux temps :

- Construction du flot de contrôle
- Remontée des dépendances de données

Construction du flot de contrôle

Avant de pouvoir effectuer les calculs sur les dépendances de données, il faut pouvoir disposer d'un flot de contrôle.

Celui-ci est construit sur la même base que l'automate des modèles précédents. Au cours de la construction, les branchements sont ajoutés au *slice*. Dans le même mouvement, les branchements conditionnels sont repérés : ils serviront de *slice criterion*.

Remontée des dépendances de données

L'algorithme 11 montre comment les dépendances de données sont remontées. On définit un ensemble de registres $\mathcal{R}$ les registres utiles au flot de contrôle. Chaque état du flot de contrôle en possède une valeur particulière (dans l'algorithme, on a noté $s.registres$ si s est l'état du flot de contrôle, comme $s.inst$ est l'instruction à ce moment du flot de contrôle).

On sait que, de façon minimale, sur les branchements conditionnels, $\mathcal{R}$ contient les registres lus par l'instruction.

De ces premiers prédicats (*slice criterion*), on fait remonter aux instructions précédentes ces registres dans le $\mathcal{R}$.

Dans cette remontée, on peut écrire les instructions sous la forme : $r_{write} = f(r_{read})$ avec r_{write} les registres écrits par l'instruction, et r_{read} les registres lus par la fonction. f est une fonction quelconque.

On peut affirmer que si $r_{write} \in \mathcal{R}$ alors $\mathcal{R}' = ((\mathcal{R}' \cup \mathcal{R}) \cap \overline{r_{write}}) \cup r_{read}$. C'est-à-dire que l'on retire les registres lus et que l'on ajoute les registres écrits.

Lorsque cette opération est effectuée, l'instruction est ajoutée au *slice*. Cela signifie qu'elle joue un rôle dans la détermination du flot de contrôle.

Et, sinon, $\mathcal{R}' = \mathcal{R}' \cup \mathcal{R}$.

L'algorithme tourne jusqu'à ce que $\mathcal{R}$ reste stable. Un point fixe est alors atteint.

Algorithme 11 Calcul des dépendances

```
1:  procedure DEPENDENCESCOMPUTATION
2:      while taille de deptoProcess > 0 do
3:          s ← depToProcess.pop()
4:          for all p prédécesseur de s do
5:              if s.inst écrit dans s.registres then
6:                  p.registres ← s.registres - registres écrits + registres lus de s.inst
7:                  s.inst dans le slice
8:              else
9:                  p.registres ← s.registres
10:             end if
11:         end for
12:     end while
13: end procedure
```

Dans le cas de l'algorithme 11, la mémoire n'est pas modélisée. Celle-ci est toutefois manipulée de la même manière, sinon qu'elle dispose d'une structure de données particulière et qu'elle est organisée dans une pile.

7.4.2 Exécution

Dans un souci d'optimisation, le tri entre les instructions présentes dans le *slice* ne se fait pas dans la boucle de simulation, mais avant elle. Durant l'initialisation, les instructions absentes du *slice* sont remplacées par des `nop` (*i.e.* des instructions vides).

7.5 Conclusion

Les résultats sur les cas réels que nous avons testés semblent montrer que le *slicing* recouvre la quasi totalité des instructions, en dépit de tous nos efforts d'optimisation. Il en découle que le temps d'exécution ne présente aucun gain particulier en face des modèles précédents, puisque aucune instruction n'est retirée du temps d'exécution.

Une étude plus poussée semble étayer que cette technique n'apporte pas de gain particulier. Même dans un découpage idéal du programme, faisant fi des restrictions de notre modèle, où le *slicing* serait donc optimal, le programme utilise la quasi-totalité des instructions.

La différence de traitement avec le travail de Béchennec et Cassez peut se trouver dans l'utilisation qu'ils font de ce *slicing*. En effet, entre autres, ils n'incluent pas les branchements (car le flot de contrôle est déjà construit).

8

Conclusion

8.1 Résultats synthétiques

Au cours de ce mémoire, nous avons étudié la problématique que pose l'efficacité de la simulation temporelle des architectures matérielles, dans le cadre des systèmes temps réel. Nous avons mis en exergue l'écart de méthodes d'optimisation existant entre la simulation fonctionnelle et la simulation temporelle.

Pour répondre à cette demande, nous avons développé tout le long de cette thèse des techniques pour améliorer l'efficacité des simulateurs temporels précis au cycle près. Nos travaux se sont appuyés sur le simulateur associé au langage de description d'architectures HARMLESS. Ces travaux se sont décomposés en trois techniques différentes.

La première a tenté d'introduire la technique de la simulation compilée à la simulation temporelle, en intégrant des tâches d'analyse du code dans la structure du modèle du processeur. Pour tâche d'analyse du code, c'est la gestion des aléas au cœur du pipeline que nous avons optimisée. Ce procédé a permis de réduire le temps d'exécution du simulateur de près de 45% en moyenne.

La seconde a répondu aux restrictions les plus gênantes du précédent. En utilisant un modèle d'automate à pile, il a permis la simulation de code utilisant un nombre consistant d'appels de fonction. À la suite de cette amélioration, la taille du modèle du processeur a été fortement réduit, selon les appels de fonction utilisés. En contrepartie, le temps d'exécution du simulateur a légèrement été impacté de l'ordre de 8%.

La troisième technique que nous avons développée faisait appel à un modèle d'abstraction du code. L'automate du processeur ne manipule plus des instructions, mais des macro-instructions de plusieurs cycles, représentant des ensembles d'instructions. Cette technique simplifie le modèle du processeur utilisé et permet de réduire le temps d'exécution d'un total de 53%, toujours en comparaison de la simulation interprétée.

Enfin, nous avons également exploré une méthode pour alléger la part de simulation fonctionnelle intégrée au simulateur CAS. Celle-ci consistait à limiter son utilisation aux calculs des instructions pertinentes pour déterminer le flot de contrôle. Cette piste de développement ne s'est pas avérée fructueuse pour améliorer le temps d'exécution. En raison de la nécessité d'exécuter les appels à la mémoire, trop d'instructions étaient exécutées.

Pour synthétiser les contributions de ce mémoire, la Figure 8.1 retrace l'évolution du temps

d'exécution moyen au fil de nos propositions.

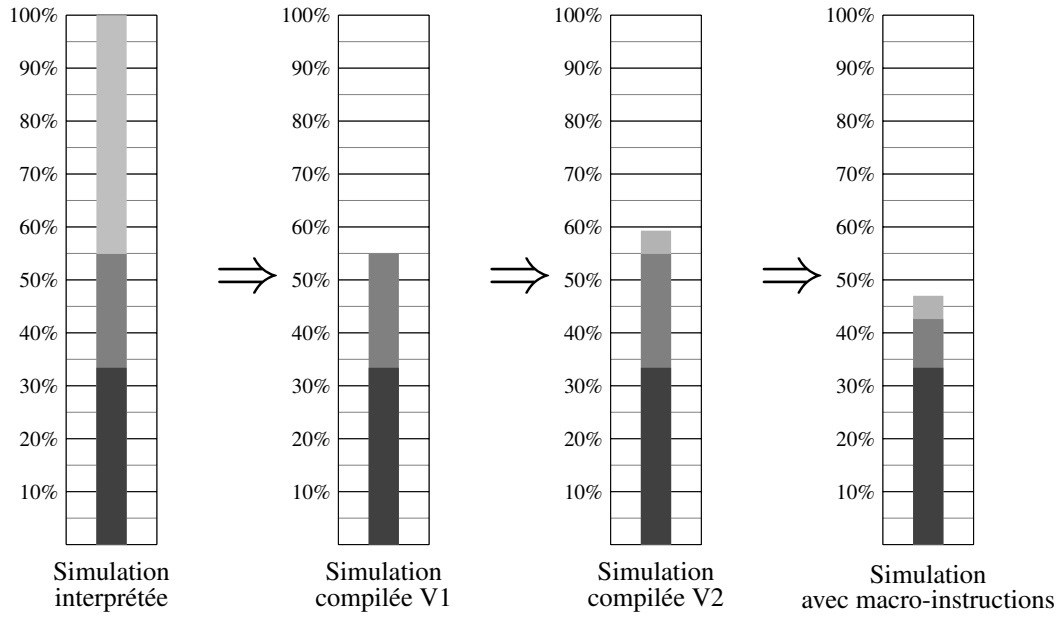

FIGURE 8.1 – Évolution moyenne des tâches dans le temps d'exécution du simulateur CAS au fil des contributions du mémoire.

8.2 Perspectives

Cette thèse ouvrait de nombreuses perspectives pour améliorer l'efficacité des simulateurs temporels précis au cycle près. Toutes n'ont pas été explorées et demanderaient de plus amples recherches pour se concrétiser.

Tout d'abord, nous n'excluons pas de pouvoir trouver un moyen pour mettre en œuvre la technique du *slicing*, notamment en explorant davantage le domaine de la *memory disambiguation*. Celle-ci clorait la suite logique de nos travaux d'optimisation en réduisant la dernière tâche prépondérante existante dans le temps d'exécution.

Au travers de cette thèse, la possibilité de faire appel au principe du *Just in Time* a été évoquée et dans une certaine mesure abordée. Par le compromis qu'elle offrirait entre la simulation interprétée et la simulation compilée, elle représenterait une bonne solution pour construire un simulateur temporel efficace, tout en préservant la flexibilité d'utilisation. Nous fondons aussi des espoirs à se servir de cette technique pour simuler des applications temps réel concrètes, utilisant notamment des changements de contextes, ces derniers ne pouvant pas être simulés par la simulation compilée.

Les contributions proposées peuvent également faire l'objet d'améliorations. Deux axes peuvent se dessiner pour ce travail. D'abord, la réduction des restrictions causées par notre approche d'analyse statique, qui comprendrait principalement la possibilité de simuler les branchements indirects. Ensuite, l'optimisation des techniques utilisées, dans laquelle on pourrait aborder la modélisation du cache d'instructions.

Une dernière perspective est d'étendre le champ d'utilisation de notre simulateur à des architectures plus diverses et modernes. La possibilité de simuler des architectures multicœurs a été évoquée au cours de notre thèse et des pistes ont été tracées. L'utilisation de pipelines superscalaires est également une autre possibilité de développement.

A

Le Pipeline

A.1 Les Bases

Les pipelines sont un élément incontournable des micro-architectures. Ils permettent une certaine parallélisation de l'exécution des instructions, ce qui améliore grandement la vitesse des processeurs. Je propose dans cette annexe un petit point d'explication sur leur fonctionnement, de telle manière à ce que le lecteur encore profane pour le sujet puisse en avoir une idée intuitive et mieux suivre les contributions faites au cours de ce mémoire.

Sans l'utilisation d'un pipeline, une instruction pour être exécutée doit attendre que la précédente ait terminé son exécution, comme on peut le voir sur la Figure A.1.

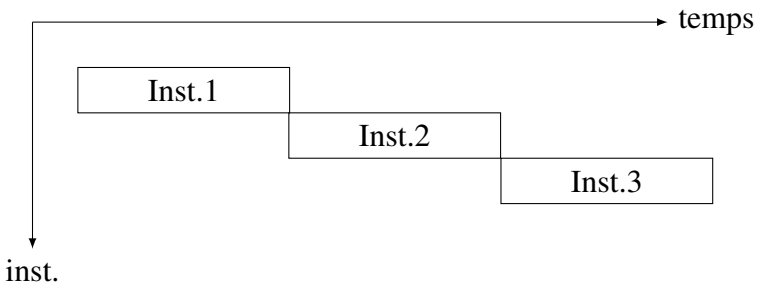

FIGURE A.1 – Sans pipeline, les instructions sont exécutées en série.

Cette exécution est cependant constituée de plusieurs tâches élémentaires. Elles ont basiquement une durée d'exécution d'un cycle processeur, bien que des cas de pipelines avec des étages plus consommateur en temps existent (comme la gestion des opérations flottantes). La nature de ces étapes est variable en fonction des architectures étudiées. Pour fixer un exemple, on peut considérer que l'exécution d'une instruction se déroule en quatre étapes :

Fetch : récupération du code de l'instruction dans la mémoire ;

Decode : décodage du code ;

Execute : exécution du code ;

Write Back : écriture du résultat dans la mémoire.

On parlera d'étages du pipeline pour chacune de ces étapes. L'exécution d'une instruction se fait en parcourant à la suite tous les étages du pipeline. Ainsi notre exécution se retrouve fragmentée comme il est indiqué sur la Figure A.2.

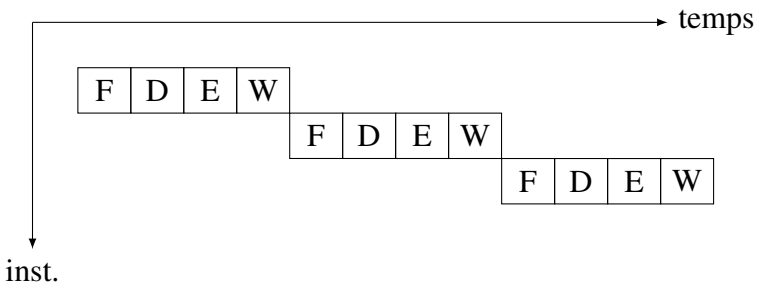

FIGURE A.2 – L'exécution d'une instruction se découpe en sous-tâches élémentaires, gérées par une ressource matérielle.

En règle générale, chacune de ces étapes fait appel à des ressources matérielles différentes (bien que des conflits puissent exister comme on le verra par ailleurs). Ainsi, quand l'une de ces étapes est terminée sa composante matérielle associée est disponible et pourrait prendre en charge l'instruction suivante directement sans attendre la fin du processus. C'est ce fait qui est exploité dans l'architecture des pipelines.

Un pipeline permet aux instructions d'accéder aux différents étages dès que ceci sont disponibles. Ainsi, de manière idéale, l'exécution de deux instructions se suit d'un étage, c'est-à-dire d'un cycle processeur. Il faut relever qu'auparavant ce temps était de l'ordre de la taille du pipeline, quatre cycles processeur dans notre exemple.

Une telle organisation peut être comparée à un travail à la chaîne. Elle permet de rendre plus performante l'exécution d'un code. Sur la Figure A.3, on peut voir le gain de temps obtenu à exécuter les instructions de cette manière.

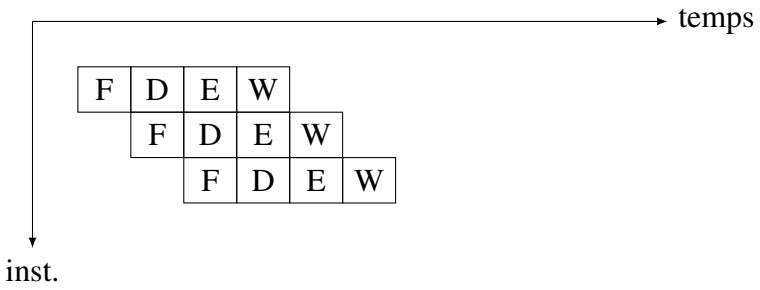

FIGURE A.3 – L'exécution des instructions est parallélisée grâce à l'utilisation d'un pipeline.

Comme le gain de temps obtenu est dépendant de la taille du pipeline, on peut comprendre que la segmentation de l'exécution d'une instruction en des tâches les plus élémentaires possibles est un atout important. C'est ce qui explique l'utilisation de pipelines très grands dans les architectures modernes.

On rencontre différentes sortes de pipelines : non-linéaires, superscalaires, etc. Dans le cadre de cette thèse, nous nous sommes focalisés sur les pipelines simples.

A.2 Les Différents types d'aléas

Idéalement, et comme nous l'avons vu, le *Cycle Per Instruction* (CPI) peut atteindre 1 en utilisant un pipeline. En pratique, le bon fonctionnement d'un pipeline est altéré par ce que l'on appelle des aléas [HP01]. On en distingue de trois espèces.

Aléas structurels : un recouvrement parfait des instructions n'est pas possible à cause d'un conflit entre les ressources matérielles. Nous l'avions dit, il peut arriver que les étapes élémentaires d'une exécution fassent appel à des ressources communes. Un exemple à cela, est l'utilisation de la mémoire, avec laquelle il n'est pas possible de faire simultanément une écriture et une lecture.

Aléas de données : le code présente quelques fois des dépendances de données entre des instructions : une instruction peut nécessiter de lire le résultat d'une instruction précédente, par exemple. Avec le recouvrement que permet le pipeline, plusieurs configurations peuvent apparaître qui rendent la cohérence des données problématiques. Pour reprendre notre exemple, si la lecture des opérandes se fait dans un des premiers étages du pipeline et l'écriture dans un des derniers, il est possible avec le recouvrement qu'une instruction soit capable de lire ses données avant que la précédente ait pris le temps de les écrire.

Aléas de contrôle : il s'agit d'un aléa particulier spécifique à la manipulation du flot de contrôle, via les branchements. Ceux-ci posent de nombreuses problématiques, notamment la suivante : avant d'avoir calculé la cible d'un branchement, l'on ne peut pas déterminer avec assurance quelles sont les instructions suivantes qui peuvent entrer dans le pipeline. Cette difficulté crée une sorte d'indétermination sur les instructions qui suivent un branchement et qui, grâce au pipeline, aurait déjà dû pénétrer dans le pipeline.

Pour résoudre ces aléas, plusieurs techniques existent. La plus basique est d'introduire dans le pipeline des temps d'attente. Ils permettent à l'exécution des instructions de reprendre une séquencialité plus large autant que c'est nécessaire. Comme ce procédé mène à l'apparition d'étages vides à l'intérieur du pipeline, on utilise le terme de bulles.

Des techniques plus complexes sont mises en pratique, pour éviter les pertes de temps que causent ces bulles. Pour les aléas de contrôle, on peut, par exemple, recourir à ce que l'on appelle la prédiction de branchement (de telle manière à limiter les cas où les instructions entrées ne correspondent pas au branchement pris), l'exécution spéculative (exécution supposée de l'instruction suivante).

Pour la gestion des aléas de données, on peut citer l'ordonnancement dynamique (réagencer les instructions pour éviter des aléas), le renommage de registres (changer de registres pour éviter les aléas).

A.2.1 Aléas de données

Dans le cadre de cette thèse, nous nous sommes principalement intéressés aux aléas de données qui font l'objet d'une phase d'optimisation particulière dans notre travail.

Ces aléas peuvent être de trois natures :

Lecture après écriture (LAE) : une instruction essaie de lire une donnée avant qu'elle n'ait été écrite.

Écriture après écriture (EAE) : une instruction écrit une donnée avant une autre qui la précède, son résultat sera écrasé alors qu'il est le plus récent. C'est le cas d'un pipeline où

plus d'un étage peuvent écrire dans le registre. On le trouve encore dans les architectures faisant recours à l'exécution dans le désordre.

Écriture après lecture (EAL) : une instruction écrit une donnée dans un registre, écrasant par là une donnée que devait lire une instruction précédente. Ce cas de figure est seulement possible si un étage d'écriture précède un étage de lecture.

Les aléas de type EAE et EAL ne se rencontrent que dans le cas de pipeline non-linéaires ou d'exécution dans le désordre. Ces deux types d'architectures n'ont pas été étudié dans le cadre de cette thèse. C'est la raison pour laquelle, le seul type d'aléa de données qui sera traité au cours de ce mémoire est le type LAE, que l'on rencontre dans les pipelines linéaires.

Table des matières

Liste des tableaux

Table des figures

Bibliographie

[ACD+06] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, and all. *Section "Fault Tolerance", Encyclopedia of Computer Science and Information Systems*. Vuibert, 2006. 7

[AS92] Kristy Andrews and Duane Sand. Migrating a cisc computer family onto risc via object code translation. *Proceedings ASPLOS V*, pages 213–222, 1992. 20

[BBBT13] Adrien Bullich, Mikaël Briday, Jean-Luc Béchennec, and Yvon Trinquet. Comcas, a compiled cycle accurate simulation for hardware architecture. *The Fifth International Conference on Advances in System Simulation (SIMUL'13)*, 2013. 54

[BBBT14] Adrien Bullich, Mikaël Briday, Jean-Luc Béchennec, and Yvon Trinquet. Improving processor hardware compiled cycle accurate simulation using program abstraction. *7th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS'14)*, 2014. 66, 77

[BBH+94] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, and Dietmar Voggenaur. The mimola language version 4.1. Technical report, Lehrstuhl Informatik XII, University of Dortmund, 1994. 12

[BC11] Jean-Luc Béchennec and Franck Cassez. Computation of wcet using program slicing and real-time model-checking. Non publié, 2011. 82, 83

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. *Translator*, 394 :41–46, 2005. 20

[BNH+02] Gunnar Braun, Achim Nohl, Andreas Hoffmann, Oliver Schliebusch, Rainer Leupers, and Heinrich Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *DAC'02 : Proceedings of the 39th annual Design Automation Conference*, pages 22–27, 2002. 16, 19

[CHK01] Keith Cooper, Timothy Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software – Practice and Experience*, 4 :1–10, 2001. 83

[CK94] Bob Cmelik and David Keppel. Shade : A fast instruction-set simulator for execution profiling. *SIGMETRICS 1994*, pages 128–137, 1994. 21

[CM96] Cristina Cifuentes and Vishv Malhotra. Binary translation : Static, dynamic, retargetable ? *Proceedings International Conference on Software Maintenance*, pages 340–349, 1996. 20

[DC06] Anne-Marie Déplanche and Francis Cottet. *Section "Ordonnancement temps réel". Encyclopédie de l'informatique et des systèmes d'informations*. Vuibert, 2006. 9

[FVPF95] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nml. *EDTC'95 : Proceedings of the 1995 European Conference on Design and Test*, pages 503–507, 1995. 12

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147. OCG, 2010. 34, 76

[GTW06] Loukas Georgiadis, Robert E. Tarjan, and Renato Fonseca F. Werneck. Finding dominators in practice. *Graph Algorithms Appl.*, 10 :69–94, 2006. 83

[HGa99] Ashok Halambi, Peter Grun, and al. Expression : A language for architecture exploration through compiler/simulator retargetability. *European Conference on Design, Automation and Test (DATE)*, pages 485–490, 1999. 12

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srivinas Devadas. Isdl : an instruction set description language for retargetability. *DAC'97 : Proceedings of the 34th annual conference on Design automation*, pages 299–302, 1997. 12

[HP85] David Harel and Amir Pnueli. *On the development of reactive systems*. Springer-Verlag, 1985. 5

[HP01] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach-Second Edition*. Morgan Kaufmann Publishers, Inc., 2001. 93

[JT09] Daniel Jones and Nigel Topham. High speed cpu simulation using ltu dynamic binary translation. *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, pages 50–64, 2009. 22

[Kas10] Rola Kassem. *Langage de description d'architecture matérielle pour les systèmes temps réel*. PhD thesis, Université de Nantes, 2010. 25

[KBB+08] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Guillaume Savaton, and Yvon Trinquet. Simulator generation using an automaton based pipeline model for timing analysis. In *International Multiconference on Computer Science and Information Technology (IMCSIT'08)*, pages 657–664, 2008. 32

[KBB+09] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet, and Guillaume Savaton. Instruction set simulator generation using HARMLESS, a new hardware architecture description language. *Simutools '09 : Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 24 :1–24 :9, 2009. 16, 25

[KBB+11] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Guillaume Savaton, and Yvon Trinquet. HARMLESS, a hardware architecture description language dedicated to real-time embedded system simulation. *Journal of Systems Architecture*, pages 318–337, 2011. 11, 27

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1 :121–141, 1979. 83

[MAF91] Christopher Mills, Stanley C. Ahalt, and James E. Fowler. Compiled instruction set simulation. *Software - Practice and Experience*, 21 :877–889, 1991. 17

[MD08] Prabhat Mishra and Nikil Dutt, editors. *Processor description languages*. Morgan Kaufmann Publishers, 2008. 12

[PHM00] 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. 18, 19, 101

[PHZM99] 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. 12

[QRM04] 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)*, pages 47–56, 2004. 12

[Raj96] V. Rajesh. A generic approach to performance modeling and its application to simulator generator, 1996. 12

[Sc99] Philippe Schnoebelen (coordinateur). *Vérification de logiciel : Techniques et outils du model-checking — Livre collectif*. Vuibert, 1999. 8

[Ta06] Yvon Trinquet and al. *Section « Systèmes temps réel ». Encyclopédie de l'informatique et des systèmes d'information*. Vuibert, 2006. 6

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10 :352–357, 1984. 83

[WR96] Emmet Witchel and Mendel Rosenblum. Embra : Fast and flexible machine simulation. *SIGMETRICS 1996*, pages 68–79, 1996. 21

[WWFH03] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts : Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–95, 2003. 22

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Adrien BULLICH

Simulation efficace d'architectures opérationnelles

Efficient Simulation of Operational Architectures

Résumé
Le travail de cette thèse s'inscrit dans le contexte des systèmes embarqués temps réel. Ces systèmes exigent d'importants efforts pour leur validation et leur vérification. Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à la validation par simulation. Celle-ci peut se faire à de nombreux niveaux d'abstraction, depuis le haut niveau du modèle de l'application jusqu'au code binaire. Nous nous situons dans ce dernier cas, seul capable de donner des résultats précis au cycle près.
HARMLESS est un langage de description d'architectures. Il permet de générer un simulateur fonctionnel (ISS) et un simulateur temporel précis au cycle près (CAS). Dans le cadre des systèmes temps réel, c'est ce deuxième type de simulateurs qui nous intéresse. Relativement à l'ISS, un CAS présente l'inconvénient majeur de se montrer très lent à l'exécution.
Un moyen d'améliorer la rapidité d'exécution est d'utiliser la simulation compilée. On distingue deux grands types d'implantation de la simulation : la simulation interprétée et la simulation compilée. Si un simulateur interprété se comporte comme un interpréteur du programme à exécuter, un simulateur compilé nécessite une phase de compilation du programme. Il permet une meilleure vitesse d'exécution, au détriment de la souplesse dans la chaîne de développement.
C'est cette technique que nous introduirons dans le cadre des CAS. Couplé à une technique d'abstraction du programme, nous montrerons que l'on peut améliorer la vitesse de simulation de plus de 50% en comparaison du CAS interprété.

Mots clés
Systèmes temps réel, Architectures matérielles, Simulation.

Abstract
The work of this thesis lies in the context of real-time embedded systems. These systems require significant effort for validation and verification. As part of this thesis, we are particularly interested in validation through simulation. This can be done at many levels of abstraction, from high-level model of the application to binary code. We are in the latter case, only able to give accurate results close to the processor cycle.
HARMLESS is a Hardware Architecture Description Language (HADL). It generates a functional simulator (ISS) and a temporal simulator cycle-accurate (CAS). As part of real-time systems, it is this second type of simulators that interests us. With respect to the ISS, CAS has the major drawback to be very slow at runtime.
One way to improve the speed of execution is to use the compiled simulation. There are two main types of implementation of the simulation: interpreted simulation and compiled simulation. If interpreted simulator behaves as an interpreter for the program to run, a compiled simulator requires a compilation phase of the program. It allows a better execution speed to the detriment of flexibility in the development chain.
It is this technique that we introduce in the context of CAS. Coupled with a technique of abstraction of the program, we will show that we can improve the simulation speed by more than 50% compared to the interpreted CAS.

Key Words
Real-Time Systems, Material Architectures, Simulation.