Calcul du pire temps d'exécution : méthode formelle s'adaptant à la sophistication croissante des architectures matérielles

Bilel Benhamamouch

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Spécialité Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Bilel Benhamamouch

Thèse dirigée par **Jean-Claude Fernandez**
et codirigée par **Bruno Monsuez**

préparée au sein **UEI ENSTA-ParisTech et VERIMAG**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Calcul du pire temps d'exécution

Méthode formelle s'adaptant à la sophistication croissante des architectures matérielles

Thèse soutenue publiquement le **02 mai 2011**,
devant le jury composé de :

M, Eric Goubault
Directeur de recherche au Comissariat de l'Énergie Atomique, Rapporteur

M, Marc Pouzet
Professeur à l'Université Pierre et Marie Curie, département d'informatique de l'Ecole Normale Supérieure, Rapporteur

M, Jean-Claude Fernandez
Directeur de l'Unité de Formation et de Recherche IMAG, Directeur de thèse

M, Bruno Monsuez
Directeur de l'Unité Electronique-Informatique ENSTA-ParisTech, Co-Directeur de thèse

Calcul du pire temps d’exécution : Méthode d’analyse formelle s’adaptant à la sophistication croissante des architectures matérielles

Benhamamouch Bilel

02 mai 2011

Remerciements

Cette thèse a été effectuée au sein de l'Unité d'Informatique et d'Electronique (UEI) de l'Ecole Nationale Supérieur de Techniques Avancées (ENSTA).

J'exprime ma profonde reconnaissance à Mr Bruno Monsuez, qui m'a dirigé tout au long de cette thèse. Mes multiples intrusions dans son bureau ont toujours été conclues par des réponses et des remarques pertinentes. Ses critiques constructives m'ont toujours éclairé et m'ont permis de mener à bien cette thèse.

Je tiens à remercier particulièrement le Professeur Jean-Claude Fernandez, membre de l'équipe VERIMAG, pour m'avoir soutenu et permis de mener mes travaux à terme.

J'exprime ma vive reconnaissance à Mr Franck Védrine (CEA Saclay), qui m'a été d'une aide très précieuse durant cette thèse.

Je remercie le Professeur Alain Sibille, responsable de l'unité d'électronique et d'informatique (UEI), de m'avoir accueilli au sein de son laboratoire.

Je remercie également messieurs : Eric Goubault, Marc Pouzet et Denis Barthou, d'avoir accepté mon invitation et je leurs suis profondémment reconnaissant pour le temps et l'énergie qu'ils ont consacré à l'évaluation de cette thèse.

J'exprime ma profonde gratitude à Philippe Baufreton (Sagem défence) pour sa collaboration au projet APE.

Je voudrais enfin remercier tous mes proches : mes parents, ma soeur, mon épouse (pour son immense patience) et mes ami(e)s de m'avoir soutenu et surtout $SUPPORTÉ$ durant mes études.

Résumé

Pour garantir qu'un programme respecte toutes ses contraintes temporelles, nous devons être capables de calculer une estimation fiable de son temps d'exécution au pire cas. Cependant, vu la sophistication croissante des processeurs cette tâche devient un travail très laborieux.
Afin de palier à ce problème, nous proposons une nouvelle approche formelle qui permet de calculer une estimation précise des pires temps d'exécution et qui peut facilement s'adapter aux évolutions des dispositifs matériels. En effet, la méthode consiste à exécuter symboliquement un modèle exécutable du dispositif matériel, puis, à fusionner les états générés en fonction de la précision souhaitée.

Abstract

To ensure that a program will respect all its timing constraints we must be able to compute a safe estimation of its worst case execution time (WCET). However with the increasing sophistication of the processors, computing a precise estimation of the WCET becomes very difficult.
In this purpose, we propose a novel formal method to compute a precise estimation of the WCET that can be easily parameterized by the hardware architecture. Assuming that there exists an executable timed model of the hardware, we first use symbolic execution to precisely infer the execution time for a given instruction flow. Then, we merge the states relying on the loss of precision we are ready to accept.

Table des matières

Table des figures

Introduction générale

Les systèmes temps réel jouent un rôle de plus en plus important dans nos vies quotidiennes. Ces systèmes sont particuliers du fait que l'exactitude des résultats qu'ils fournissent n'est pas leur seule contrainte. En effet, ils doivent en plus fournir ces résultats dans un intervalle de temps précis. Ainsi, la conception de ces systèmes se démarque par la prise en compte de contraintes temporelles. Le respect de ces contraintes est tout aussi important que l'exactitude des résultats produits [35, 28].

En terme général, est qualifié de temps réel le comportement d'un système informatique lorsqu'il est assujetti à l'évolution dynamique d'un procédé qui lui est connecté et qu'il doit piloter (ou suivre) en réagissant à tous ses changements d'états. Dans cette définition l'expression "réagissant à tous ses changements d'états" renvoie à une notion de temporalité. Ceci implique que la détermination du pire temps d'exécution est un pré-requis indispensable à la validation des systèmes temps réel. Cependant, il ne faut pas confondre temps réel et rapidité d'exécution. En effet, une tâche temps réel n'est pas forcement une tâche qui s'exécute rapidement et vice-versa. La notion de temps réel exprime uniquement que le temps d'exécution de la tâche ne doit pas excéder une certaine durée (longue ou courte).

On distingue deux types de systèmes temps réel : les systèmes temps réel stricts ou durs (hard real-time systems) et les sytèmes temps réel souples ou mous (soft real-time systems). Cette classification est faite suivant l'importance accordée aux contraintes temporelles. Le temps réel strict ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques. A l'inverse le temps réel souple s'accommode des dépassements des contraintes temporelles dans certaines limites au-delà desquelles le système devient inutilisable. Ainsi, un système temps réel strict doit respecter des limites temporelles données même dans les pires situations d'exécution. En revanche un système temps réel souple doit respecter ces limites pour une moyenne de ses exécutions. Nous tolérons un dépassement exceptionnel qui sera *peut-être* rattrapé à l'exécution suivante.

Nous dirons donc qu'un système temps réel **strict** [1] est un système qui réalise un ensemble de fonctions sujettes à des contraintes d'intégrité qui se traduisent par le fait que tout traitement doit : (1) produire des résultats corrects et (2) être réalisé dans des délais impartis, sous peine de conséquences graves.

De nos jours, les challenges concernant la détermination des temps d'exécution sont apparus, pour la plupart, suite à la sophistication des processeurs. En effet, ces derniers comportent actuellement, de nombreux dispositifs permettant d'augmenter la vitesse d'exécution des programmes. Ces dispositifs se matérialisent par au moins un pipeline et une mémoire cache. Les travaux concernant l'analyse de WCET (de l'anglais *worst case execution time*) portent principalement sur la prise en compte de ces éléments architecturaux durant l'analyse, en vue de réaliser des systèmes temps-réel fiables et plus performants. Ainsi, le défi le plus important à l'heure actuelle, se situe au niveau matériel. Il s'agit de s'adapter au rythme effréné, suivant lequel, la structure interne des processeurs se complexifie.

Pour cela, différentes techniques ont été développées [59, 19, 26, 63, 62, 68, 31, 8, 56, 39, 2]. Un résumé des contributions les plus significatives dans ce domaine est proposé dans [49, 65]. Certaines de ces techniques ont donné lieu à la réalisation de logiciels issus aussi bien du milieu académique

1. Nos travaux concernent uniquement les systèmes temps réels stricts

qu'industriel.

Nous proposons de classer ces différentes techniques en deux grandes familles : les méthodes dynamiques et les méthodes statiques.

Concernant la première famille, ces méthodes sont dites dynamiques parce qu'elles se basent sur une série de mesures [15, 34]. Elles consistent, dans un premier temps, à injecter un jeu de données en entrée et, dans un second temps, à mesurer le temps d'exécution.

Leur facilité d'utilisation fait que les méthodes dynamiques sont largement utilisées dans l'industrie. Néanmoins, ces méthodes souffrent d'un manque de fiabilité car la détermination des pires temps d'exécution implique :

- soit de définir pour chaque programme le jeu d'entrée adéquat [2].
- soit d'explorer tous les chemins d'exécution.

La satisfaction de l'une de ces conditions est nécessaire. Cependant, cette tâche demeure particulièrement difficile (voir impossible) à réaliser. Ce dilemme a conduit beaucoup de chercheurs vers des méthodes totalement formelles apparues sous le nom de méthodes statiques [24, 59, 54, 21]. Ces techniques déterminent la séquence de code qui sera exécutée :

- soit en utilisant la programmation linéaire en nombre entier (ILP : interger linear programming) afin de considérer implicitement toutes les traces d'exécution [60, 51, 44, 32]
- soit, comme pour les analyses à base d'arbres, en énumérant explicitement toutes les traces d'exécution du programmes [46, 33]

Les performances de ces méthodes peuvent être améliorées. Pour cela, des informations supplémentaires (comme les compteurs de boucles) sont nécessaires. Ce type d'information peut être obtenu de différentes façons : (1) grâce à des annotations rajoutées au code source [45], ou (2) automatiquement après analyse de flot [20, 23, 26].

Le point fort de ces approches est la fiabilité des résultats fournis. En effet, ces méthodes calculent une $sur - approximation$ des pires temps d'exécution. En contrepartie, d'une part, elles fournissent globalement une sur-approximation exagérée des pires temps d'exécution [3], et d'autre part, leurs mises en œuvre peut se révéler être une tâche extrêmement longue et complexe.

Afin de palier aux différentes lacunes des méthodes existantes, nous proposons dans notre travail de recherche, une nouvelle méthode d'analyse formelle qui permet de calculer une sur-approximation acceptable du pire temps d'exécution (méthode semi-dynamique). Notre but a été de développer une approche qui, d'une part, soit aussi simple à utiliser qu'une méthode dynamique et qui, d'autre part, bénéficie du côté formel propre aux approches statiques.

Cette méthode doit donc s'adapter aisément aux évolutions croissantes des processeurs tout en fournissant des sur-approximations très proches des pires temps d'exécution réels.

2. Produisant le pire scénario d'exécution.
3. A cause des sur-approximations successives.

Organisation de la thèse

Ce rapport de thèse est organisé comme suit :

- **Chapitre 1 :** Le premier chapitre est dédié à l'état de l'art. Dans ce chapitre, nous présentons plusieurs techniques qui permettent de déterminer des approximations des pires temps d'exécution. Une première partie porte sur certaines méthodes dynamiques, et une seconde partie porte sur des exemples de méthodes semi-dynamiques (méthodes dynamiques formelles). Nous concluons ce tour d'horizon par une présentation des méthodes statiques (méthode AbsInt).

- **Chapitre 2 :** Dans ce chapitre, nous exposons les grandes lignes de notre approche en mettant en évidence les points forts de notre analyse et en argumentant nos choix.

- **Chapitre 3-5 :** Ces chapitres sont consacrés à la présentation détaillée de notre méthode de calcul des WCETs [5, 3]. Dans le chapitre 3, le modèle exécutable du proceseur est présenté. Puis, un cas d'étude, représentant un exemple de processeur (PowerPC 603), est introduit afin d'expliquer comment le modèle du processeur est construit. Le chapitre 4, est consacré à la partie exécution symbolique de l'approche. Dans cette partie, nous expliquons comment cette technique est adaptée au calcul des temps d'exécution. L'exécution symbolique étant source d'explosion combinatoire, une politique de fusionnement est proposée dans le chapitre 5 . Dans ce chapitre, nous présentons la méthode utilisée pour gérer l'explosion du nombre d'états, ainsi que l'impact des fusions sur les performances globales [4] de notre méthode.

- **Chapitre 6 :** Ce chapitre porte sur les résultats pratiques obtenus lors de la phase de tests. En effet, nous avons utilisé notre approche afin d'analyser différentes structures de code. Le but de cette partie étant de montrer que la méthode peut être utilisée sur des exemples de code contenant des instructions de branchement et des boucles.

- **Chapitre 7 :** Dans ce chapitre une conclusion ainsi que des perspectives de travaux futurs sont proposées.

4. la perte de précision introduite au niveau des temps d'exécution

CHAPITRE 1

Pires temps d'exécution : Etat de l'art

1.1 Introduction

Le temps d'exécution d'un programme qui s'exécute de façon ininterrompue sur un unique processeur dépend autant des caractéristiques du programme que du processeur sur lequel il tourne. Concernant les processeurs actuels, leurs états sont fortement corrélés à leurs passifs, et donc aux instructions exécutées durant les phases précédentes. Pour comprendre l'influence de cet historique d'exécution nous pouvons prendre l'exemple du cache d'instruction. Si une instruction, requise par le processeur à un moment donné, se trouve déjà dans le cache d'instruction (cache hit) du fait d'une exécution antérieure, l'exécution se poursuivra sans blocage alors que si cette même instruction ne se trouve pas dans le cache (cache miss) cela peut se solder par une interruption de l'exécution. La durée de cette interruption est dictée par la vitesse à laquelle cette donnée va être récupérée de la mémoire. Ces deux scénarios donneront des temps d'exécution différents. Cette différence représente un exemple de l'impact qu'à un historique sur les temps d'exécution finaux. C'est pourquoi le calcul du pire temps d'exécution d'un programme est défini comme étant le plus long temps d'exécution possible en considérant simultanément toutes les combinaisons de données en entrée et tous les historiques d'exécution.

Ce pire temps d'exécution a un impact considérable sur les performances des logiciels qui doivent satisfaire des contraintes temps réel. Pour cela, différentes techniques ont été mises au point. Ces techniques peuvent être classées en trois grandes familles qui sont l'estimation dynamique, l'estimation dynamique formelle et l'analyse statique.

1.2 Estimation dynamique

Cette méthode consiste simplement à injecter un jeu de données en entrée du programme, puis à mesurer les temps d'exécution obtenus. Cette mesure peut se faire de trois façons différentes [69] :

- (1) de façon matérielle, en utilisant des dispositifs de mesure comme des chronomètres ou des oscilloscopes,
- (2) de manière logicielle grâce, par exemple, aux fonctions de temps qui sont mises à disposition par les systèmes d'exploitation,
- (3) de façon hybride, en combinant les deux méthodes précédemment citées. Exemple : en insérant dans le programme des petits bouts de code (partie logicielle) qui seront utilisés pour lancer ou arrêter l'appareil de mesure (partie matérielle).

L'estimation dynamique est très utilisée dans l'industrie, ce qui explique le nombre important de méthodes développées en vue de mesurer les temps d'exécution [57]. Chacune de ces méthodes est mise en œuvre dans le but de réaliser le meilleur compromis entre plusieurs caractéristiques tel que :

- la résolution qui représente les limites du matériel utilisé lors de la mesure. Exemple, si un temps d'exécution est mesuré en utilisant un chronomètre celui-ci mesurera avec une certaine résolution. En dessous de celle-ci toutes les mesures seront identiques.
- la précision qui correspond à la différence entre la mesure effectuée et la mesure obtenue dans le cas idéal [1]. Donc si une mesure est répétée plusieurs fois, celle-ci contiendra un pourcentage d'erreur et elle sera donc représentée sous la forme suivante : $x \pm y$, où y est la marge d'erreur.
- la granularité qui représente la taille du code qui peut être mesurée selon la méthode de mesure choisie. Certaines méthodes serviront à mesurer le temps d'exécution d'une procédure ($coarse - grain$), alors que d'autres pourront être utilisées pour mesurer le temps d'exécution d'une instruction ($fine - grain$).
- la difficulté qui représente les efforts fournis afin d'obtenir les mesures souhaitées.

1.2.1 Choisir la méthode de mesure

Le premier critère de sélection, lors du choix de la méthode de mesure, est dicté par la raison qui pousse à effectuer ces mesures. Prenons l'exemple du développement d'un système qui doit respecter des contraintes temps réel. La problématique du dimensionnement se pose généralement dès le départ. Cette phase permet de répondre à d'importantes questions telles que : comment choisir le processeur adéquat ou comment obtenir, pour une fonction, le nombre maximal d'itérations pouvant être effectuées chaque seconde.

A ce niveau, l'utilisation de méthodes de mesures grossières (coarse-grain) peut permettre d'obtenir plusieurs réponses facilement et assez rapidement. Par contre, lorsque les mesures sont faites dans le but d'optimiser du code ou d'analyser les performances temps réel d'un système, les méthodes grossières se retrouvent vite inappropriées et doivent, ainsi, être remplacées ou du moins conjointement utilisées avec des méthodes de mesure plus précises mais également plus coûteuses.

Nous proposons, ci-dessous, un bref état de l'art des différentes méthodes de mesures existantes. Nous commençons ce tour d'horizon par des méthodes grossières qui sont les plus simples à développer, puis, nous présenterons des méthodes de plus en plus complexes pour enfin mettre en évidence que la quête de précision est indissociable de la difficulté de mise en œuvre.

1.2.1.1 Méthode Stop-watch (chronomètre)

C'est sans conteste, la méthode de mesure la plus simple. Un chronomètre est utilisé pour mesurer la durée d'exécution d'un programme. Evidemment, cette méthode ne peut être utilisée que pour mesurer les temps d'exécution de tâches qui s'exécutent sans interruption sur un unique processeur. Cette méthode est généralement appropriée pour des utilisateurs dont les besoins se limiteraient à des approximations grossières du temps d'exécution.

1.2.1.2 Commandes date et time (UNIX)

Ces deux méthodes sont utilisables si les mesures sont faites à l'aide d'un système d'exploitation fournissant une commande permettant d'afficher la date et le temps. La manière la plus commune d'utiliser ces commandes est : (1) soit de mettre la commande qui lance le programme que nous souhaitons analyser entre deux appels de la commande $date$ (2) soit d'utiliser directement la commande $time$. comme suit :

1. Mesure qui ne contient aucun pourcentage d'erreur.

```
$date
$program
$date
```

```
$ time  program
```

Comme pour la méthode utilisant le chronomètre, ces mesures de temps d'exécution vont aussi donner une estimation grossière. La cause étant qu'elles ne prennent pas en considération plusieurs phénomènes qui peuvent se produire durant une exécution, telles que les préemptions ou les interruptions.

1.2.1.3 Timer et Compteur

La plupart des systèmes embarqués possèdent des timers/compteurs qui peuvent être programmés par l'utilisateur. Si ce type de matériel est disponible il pourra être utilisé pour obtenir des mesures très fines et très précises de temps d'exécution de petits segments de code. Le temps d'exécution du segment de code est obtenu en calculant la différence entre les valeurs lues dans le timer/compteur au début et à la fin de l'exécution, et en multipliant cette différence par le temps que prend chaque tic du timer/compteur.

Lors de l'utilisation de cette méthode les erreurs dues au dépassement de capacité (overflow) sont à prendre en compte. Il faut donc bien choisir les segments de code en fonction de la taille des registres du timer/compteur et de la résolution souhaitée.

Cette méthode fournit des temps d'exécution qui incluent les temps de préemptions et les temps des différentes interruptions. Une façon simpliste de calculer uniquement le temps d'exécution de la tâche serait d'invalider les interruptions durant les mesures. Cependant, ceci pourrait se révéler catastrophique au regard des performances temps réel. Cette solution demeure, néanmoins, réaliste durant les phases de tests lorsque nous souhaitons juste obtenir les temps d'exécution des différents segments d'un programme.

1.2.1.4 Prof et Gprof (UNIX)

Les méthodes que nous avons vu jusqu'ici ne permettaient que de mesurer les temps d'exécution d'une séquence d'intructions. Néanmoins, il arrive souvent que nous ne nous intéressions qu'à une toute petite partie du programme (petite fonction : nombre minime d'instructions) et pour cela il existe d'autres méthodes de mesure. Certaines de ces méthodes se basent sur les mécanismes de profilage $prof$ et $gprof$.

Le profilage signifie : obtenir un ensemble de mesures de temps couvrant tout (ou une grande partie) du code. La grandeur de ce code dépend de la méthode de mesure utilisée.

Les méthodes $prof$ et $gprof$ sont similaires, mais la méthode $gprof$ fournit plus d'informations. Ces méthodes sont généralement utilisées pour mesurer les temps d'exécution des fonctions. De plus, contrairement aux méthodes de mesure vues jusqu'ici, celles-ci fournissent les temps d'exécution en prenant en compte la préemption. Ces méthodes peuvent donc être utilisées pour optimiser du code. Dans ce contexte, nous pourrions identifier la fonction qui s'accapare le plus du processeur et donc identifier la partie du code qui a besoin d'être optimisée.

Pour utiliser $prof$, il faut compiler le programme en utilisant l'option $-p$ et ensuite l'exécuter, comme suit :

```
$ gcc -p -o programme programme.c
$ programme
```

Une fois que l'exécution du programme se termine, le fichier $mon.out$ est automatiquement généré. C'est un fichier binaire qui contient des données temporelles concernant les fonctions du programme. Pour afficher ces données il suffit d'utiliser la commande suivante :

```
$ prof programme
```

Si nous souhaitons avoir plus de précision concernant le comportement temporel des fonctions du programme, nous pouvons réitérer exactement les mêmes étapes en remplaçant lors de la compilation l'option $-p$ par $-pg$. Nous obtenons ainsi un fichier $gmon.out$ que nous pouvons visualiser grâce à la commande :

```
$ gprof programme
```

1.2.1.5 Analyseur logiciel

L'expression analyseur logiciel englobe généralement tous les outils logiciels fournis, soit par une société spécialisée tel que : WindView de Wind River systems soit directement par le système d'exploitation, qui permettent de mesurer les temps d'exécution. Dans la littérature nous constatons qu'un bon analyseur logiciel est un analyseur qui à l'instar des méthodes précédemment définies ($prof$ et $gprof$) ne se limite pas seulement à mesurer le temps d'exécution d'une fonction mais peut également être utilisé dans des analyses temporelles à plus petite échelle. Exemple : les boucles ou les petits segments de code qui peuvent se limiter à une seule instruction.

Certains analyseurs fournissent aussi des traces d'exécution permettant de connaitre précisément et à tout moment la tâche qui a été exécutée. Ceci peut se révéler extrêmement intéressant pour tous les développeurs qui souhaitent déterminer des erreurs de synchronisation entre processus.

Néanmoins, lors de l'utilisation de ce type d'analyseur il faut prêter une attention particulière à l'allocation des ressources. La plupart de ces analyseurs requiert beaucoup de mémoire nécessaire à la sauvegarde des données. Ceci rend l'outil moins performant lorsqu'il est utilisé dans un environnement dont les ressources sont limitées.

1.2.2 Méthodes dynamiques : avantages et inconvénients

Les méthodes dynamiques sont des méthodes de mesure, qui sont, pour la plupart, faciles à mettre en œuvre et à utiliser. Néanmoins, le principal inconvénient de ces méthodes est que pour déterminer le pire temps d'exécution on doit être capable soit de tester tous les jeux de données, soit de calculer le jeu de données en entrée qui mène au pire temps d'exécution.

Nous pouvons donc conclure que ce type de méthode, bien que largement utilisé dans l'industrie, manque d'exhaustivité. Le plus long temps d'exécution mesuré peut ne pas être le pire.

1.3 Estimation semi-dynamique

Dans cette section, nous allons présenter quelques exemples de méthodes récentes qui se placent a mi chemin entre l'estimation dynamique et l'analyse statique.

1.3.1 Méthode de Lundqvist et Stenstrom

Cette approche, développée par LUNDQVIST T. et STENSTRÖM P. en 1998, introduit une nouvelle façon d'estimer les WCETs [37, 36]. Elle consiste à simuler les comportements d'un programme tout en produisant des résultats aussi fiables que ceux fournis par les méthodes statiques.

Elle se définit par une exécution symbolique bas niveau des instructions à analyser. Cette exécution implique d'étendre la sémantique des instructions afin qu'elles puissent supporter des données dont les valeurs sont inconnues (unknow). Pratiquement, le résultat d'une opération sera de valeur inconnue (voir tableau 1.1) à chaque fois que l'une des opérandes aura cette valeur (valeur inconnue).

Type de l'instruction	Exemple	Sémantiques
ALU	ADD T,A,B	$T\{$ $inconnue$ si $A =$ inconnu $\vee B =$ inconnu $A + B$ sinon
Comparer	CMP A,B	$A - B$ et mettre à jour la condition : registre (cc). Peut mettre les bits du registre cc à inconnue.
Branchement conditionnel	BEQ L1	Tester les bits du registre cc afin de savoir si la branche doit être prise. Si les bits sont à inconnue, simuler les deux traces d'exécution
Load	LD R,A	Copier les données de l'adresse mémoire A vers le registre R (les données peuvent être à la valeur inconnue). Si l'adresse est à inconnue, mettre R à inconnue
Store	ST R,A	Copier les données du registre R vers l'addresse mémoire A (les données peuvent être à inconnue). Si l'adresse est à inconnue, toute la mémoire est mise à inconnue.

FIGURE 1.1: Extention de la sémantique des instructions.

En plus de cette analyse, le comportement de l'architecture est également pris en compte mais malheureusement de façon arbitraire [39]. Plus précisément, les impacts de la mémoire cache et du pipeline sur les temps d'exécution sont juste "sur"-estimés séparément grâce à une série de comparaisons. Ces estimations sont, par la suite, additionnées aux temps déterminés durant la première phase de l'analyse. Par ailleurs, pour palier à l'explosion combinatoire des traces d'exécution, cette méthode implémente une politique de fusionnement assez pessimiste. Cette dernière ne prend pas en compte l'historique d'exécution dans son intégralité. En effet, elle consiste à comparer deux états (ou plus) et à mettre à la valeur inconnue tous les paramètres qui ne seraient pas totalement identiques. Ce qui introduit évidemment une perte de précision supplémentaire sur les temps d'exécution.

1.3.2 Méthode RapiTime [1]

Rapitime est un outils commercial né d'un projet de recherche académique qui donna lieu a un prototype appelé pWCET [7, 6]. Ainsi, cet outil est le fruit de la collaboration des chercheurs GUILLEM BERNAT, ANTOINE COLIN et de STEFAN M. PETTERS. Cette méthode est une combinaison de mesures faites dynamiquement et d'analyses de traces d'exécutions effectuées statiquement. En général, cet outil permet de :

- Mesurer les temps d'exécution des blocs de code : la décision d'effectuer des mesures est motivée par le fait que les concepteurs de RapiTime reconnaissent que le meilleur modèle d'un processeur est le processeur lui même. Ainsi, cet outil effectue des tests en ligne pour mesurer le temps d'exécution de chaque chemin reliant deux instructions de branchement conditionnel.
- Identifier les traces d'exécution faisables : durant cette étape une analyse statique est effectuée. Celle-ci a pour but d'analyser les traces d'exécutions possibles de façon à trouver la combinaison des chemins qui mène à une trace d'exécution complète et faisable.
- Combiner les résultats obtenus durant les précédentes étapes (mesures et analyses des chemins) de façon à calculer le pire temps d'exécution (WCET).

Cet outil calcule les temps d'exécution en six étapes (voir figure 1.2). A la fin de chacune d'elles, RapiTime fournit également certaines informations utiles pour une étude de performances.

Ces six étapes sont :

– (1) Construction
– (2) Analyse structurelle
– (3) Test et génération des traces
– (4) Etude des traces
– (5) Calcul du WCET
– (6) Rapport d'analyse

1.3.2.1 Construction

La première étape consiste à instrumenter le code source. Ceci nécessite que ce code contienne toutes les macros ($\sharp include$) indispensables à sa compilation.

L'outil CINS prend en entrée le programme et génère deux fichiers : (1) le code instrumenté (programme.i) et (2) un fichier contenant des informations sur la structure du code analysé (programme.xsc)

1.3.2.2 Analyse structurelle

L'outil Xstutils est utilisé pour analyser les informations structurelles contenues dans le fichier (.xsc). Xstutils produit une description de la structure générale du code. Cette description est appelée arbre syntaxique étendue (programme.xse). Ce fichier sera exploité par la suite, durant les phases consacrées :

– à l'étude des traces d'exécution (étape 4),
– au calcul du pire temps d'exécution (étape 5).

1.3.2.3 Test et génération des traces

L'exécutable (programme.exe) est exécuté sur le processeur cible (ou sur un simulateur si le processeur n'est pas disponible) pour être après soumis à une variété de tests. Le but de ces tests est d'étudier toutes les traces d'exécution atteignables. Durant cette étape, tous les points d'exécution impliquant une prise de décision sont enregistrés dans un fichier appelé trace.txt.

1.3.2.4 Etude des traces

Le traceutils traite les fichiers contenant les traces générées pendant les tests (trace.txt). Il produit des fichiers contenant les traces d'exécution dans un format binaire (trace.rpz). L'outil traceparser prend le fichier généré par xstutils, ainsi que celui contenant les traces d'exécution (trace.rpz) et produit une distribution des temps d'exécution (.rtd database). Ainsi, ce fichier contient les temps d'exécution de chaque élément du programme. Ces éléments peuvent être : des fonctions de haut niveau, des routines, des boucles, des blocs de base, ou des traces d'exécution reliant deux instructions de branchement conditionnel.

1.3.2.5 Calcul du WCET

Une fois les traces traitées, la prochaine étape de l'analyse porte sur le calcul du pire temps d'exécution. L'outil wcalc traite les informations contenues dans la base de données (dans le fichier .rtd database) conjointement avec celles qui représentent la structure générale du programme (programme.xse). , afin de produire le pire temps d'exécution. Ce temps est, par la suite, stocké dans la base de données (.rtd database).

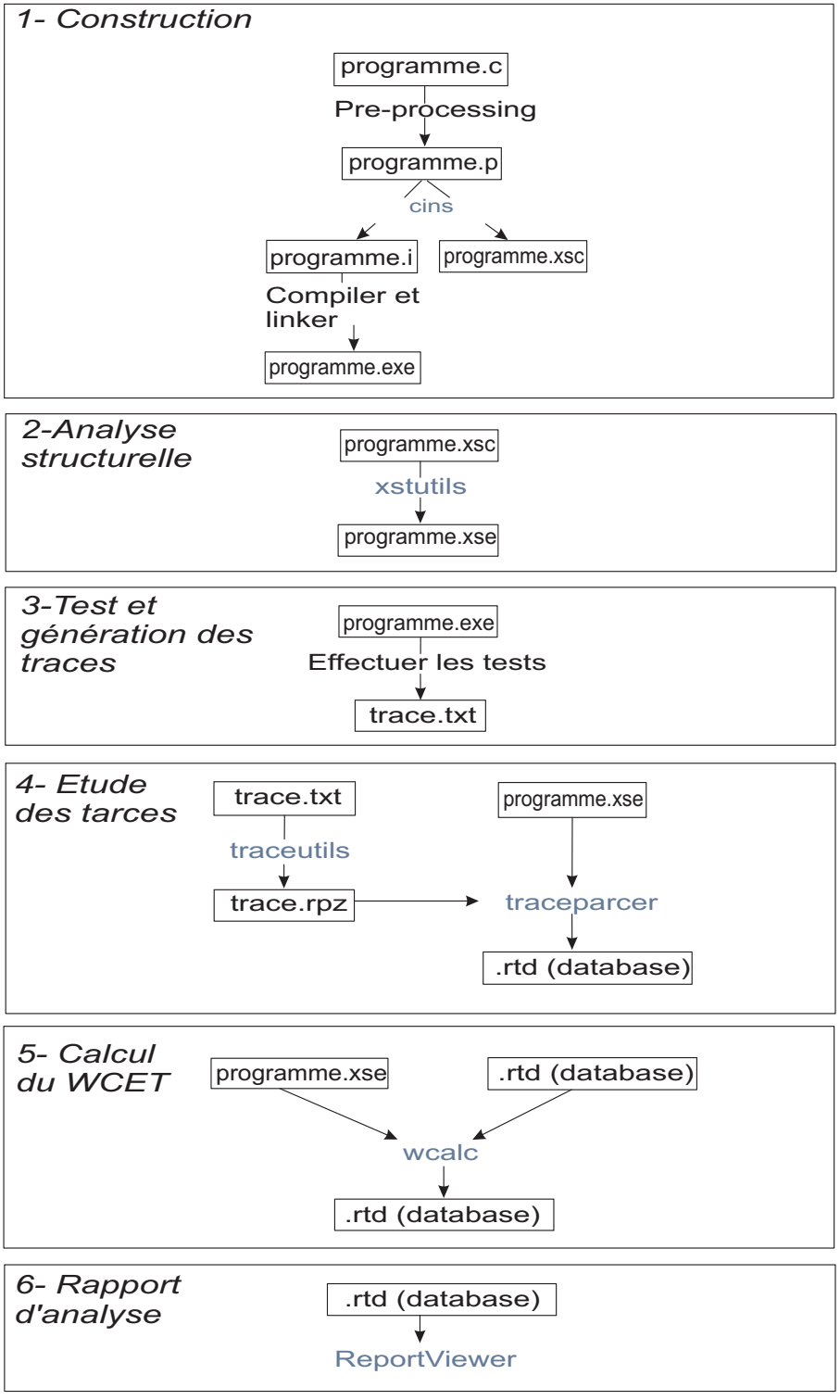


FIGURE 1.2: Méthode RapiTime

1.3.2.6 Rapport d'analyse

L'étape finale consiste à utiliser le ReportViewer pour afficher le pire temps exécution calculé, ainsi que certaines informations concernant les performances de l'application.

1.3.3 Méthode HEPTANE (Hades Embedded Processor Timing ANalyzEr)

En 2001, la collaraboration de COLIN, A., et PUAUT, I. pour la mise au point de cet outil [14] fut motivé par la volonté de disposer d'une méthode de calcul des WCETs complète. Ainsi, cet outil englobe une analyse de mémoire cache, de pipeline et de prédiction de branchement.

1.3.3.1 Représentation du programme à analyser

L'outil calcule les WCETs des tâches grâce à une analyse successive.

En premier lieu, le code assembleur du programme est divisé en plusieurs petites parties appelées : bloc de base. Ce dernier est une séquence d'instructions avec un unique point d'entrée et un seul point de sortie. Les instructions de saut sont présentes uniquement à la fin du bloc de base. Ainsi ces blocs de base sont conjointement utilisés avec les informations collectées durant la compilation du programme de façon à générer deux représentations du code source :

- un arbre syntaxique.
- un graphe de flot de contrôle (CFG : control flow graph) .

Une petite contrainte est rajouter au langage source de manière à ce que ces représentations soient équivalentes. Autrement dit, toutes les traces d'exécution qui apparaissent au niveau du graphe de flot de contrôle doivent apparaitre au niveau de l'arbre syntaxique et vice versa.

1.3.3.2 Analyse des boucles

Le comportement d'une instruction est fortement corrélé à son environnement d'exécution. Ainsi, dans le cas où une instruction ferait partie du corps d'une boucle, la prise en compte de cette boucle, comme environnement d'exécution, est primordiale. A cet effet, la méthode présentée définit une structure de données hiérarchique. Ceci lui confère la possibilité d'identifier chacune des boucles du programme ainsi que son niveau d'imbrication. Durant l'analyse, à chaque boucle, est associé un niveau d'imbrication appelé In-Level. Les In-level $[0], [1]$, ..., sont associés aux grandes boucles du programme, alors que le In-Level $[]$ représente le niveau qui englobe tout l'arbre syntaxique. Ainsi, une boucle dont le In-Level est égal à $[0.2.7.1]$ représente une partie du corps de la boucle dont le In-Level est égal à $[0.2.7]$. Ce raisonnement est appliqué jusqu'à la dernière étape qui implique que la boucle dont le In-Level est égal à $[0]$ est inclus dans la boucle dont le In-Level est égal à $[]$.

1.3.3.3 Analyse de la mémoire cache

La prise en compte du cache d'instructions est faite par simulation statique du cache [43]. Cette technique nécessite le classement des instructions en fonction de leurs comportements vis-à-vis du cache d'instructions. Ce classement s'appuie sur la présence (absence) probable de l'instruction dans le cache, au moment où elle est requise par le processeur, ainsi que de l'impact du comportement de cette instruction sur les présences (absences) probables des autres instructions. Ainsi, le but de cette classification est d'identifier les instructions susceptibles de causer un conflit provoquant un cache miss.

1.3.3.4 Analyse des instructions de branchement conditionnel

Le mécanisme de prédiction de branchement a pour but de réduire les délais d'attente lorsqu'une instruction de branchement conditionnel est rencontrée (voir section 3.4.1.4). Cela permet au processeur de poursuivre l'exécution en attendant que l'instruction de branchement soit traitée. Ainsi, ce mécanisme prédit si une branche sera traitée ou pas. Pour ce faire, l'unité de branchement se base généralement sur un historique. Ce dernier est rangé dans une pile de taille limitée. Un conflit est susceptible de se produire si deux instructions de branchement partagent le même emplacement.

Ainsi, la technique appliquée durant l'analyse des mémoires caches est également utilisée pour classer les instructions de branchement conditionnel. Le but est d'identifier les instructions pour lesquelles l'unité de branchement ne disposerait pas de leurs historiques au moment de leurs traitements.

1.3.3.5 Analyse du pipeline

Les résultats obtenus suite aux analyses de mémoires caches et des instructions de branchement sont injectés en entrée du simulateur de pipeline ($PipeSim$). Ce dernier peut ainsi fournir une estimation des temps d'exécution dans le pire cas.

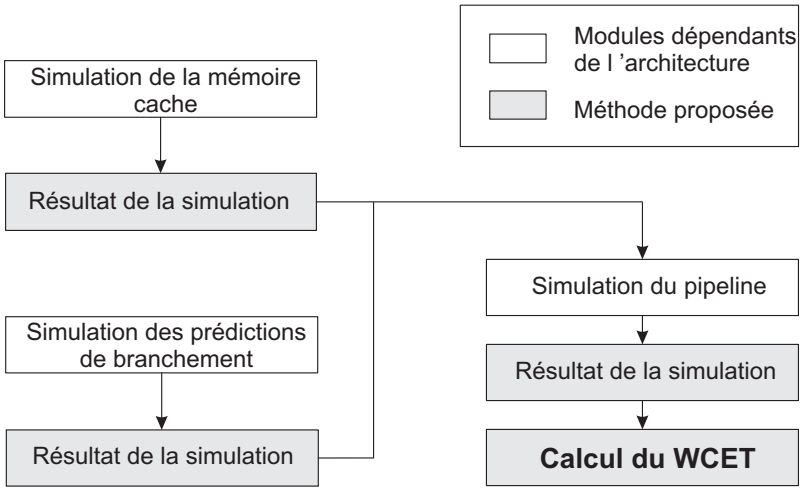


FIGURE 1.3: Méthode HEPTANE

1.3.4 Méthodes semi-dynamiques : avantages et inconvénients

Les méthodes semi-dynamiques présentées ci-dessus sont toutes modulaires et adaptables à plusieurs types d'architectures modernes. Néanmoins, ces analyses sont classées parmi les analyses compartimentées (analyse de mémoire cache, analyse de pipeline, etc.) qui ne capturent pas avec exactitude les multiples interactions intra-processeurs. En conséquence ces méthodes sont susceptibles d'introduire une perte de précision, sur les temps d'exécution, qui augmente selon la complexité du processeur.

1.4 Analyse statique

1.4.1 Motivations

Les méthodes d'analyse statique sont des méthodes formelles. Cependant, comparées aux méthodes dynamiques, elles demeurent beaucoup plus difficiles à mettre en œuvre.

Elles consistent :

- (1) à identifier les boucles et les instructions de branchement afin de les analyser séparément des autres instructions.
- (2) à découper le code restant en séquence et à déterminer le temps d'exécution de chaque bloc.
- (3) à exploiter conjointement les résultats des deux étapes précédentes pour construire les différents chemins d'exécution afin d' identifier le plus long.

L'avantage des méthodes statiques réside essentiellement dans leur capacité à fournir une couverture totale des traces d'exécution. Prenons un exemple simple et supposons que durant une exécution le processeur requiert une donnée. Nous savons qu'à chaque fois que cela se produit, cette donnée est logiquement localisée grâce à son adresse. La valeur de cette adresse peut dépendre des données fournies au programme en début d'exécution. Dans cette situation, il apparait clairement qu'un calcul sûr des temps d'exécution ne peut se faire que si la méthode d'analyse utilisée permet de prendre en compte toutes les valeurs possibles fournies en entrée du programme, de telle façon que toutes les adresses possibles soient considérées.

A présent, intéressons-nous aux inconvénients de cette approche. Hormis la difficulté de mise en oeuvre, son désavantage le plus apparent demeure dans la précision des résultats. En effet, cette méthode, utilisée pour analyser des architectures complexes (contenant mémoire cache et pipeline), aura tendance à sur-approximer (parfois un peu trop) les WCETs. La raison est que durant l'analyse tous les chemins d'exécution sont explorés et qu'une partie d'entre eux sont infaisables (contenant des états inatteignables). De plus, avec l'apparition de nouvelles architectures destinées à augmenter les performances des processeurs, ces derniers disposent tous actuellement d'(au moins) une mémoire cache, un pipeline, et peuvent être conçus pour faire de l'exécution spéculative ou du ré-ordonnancement d'instructions (out of order execution). Ces innovations font que le comportement des processeurs est de plus en plus imprévisible. Ce qui rend le calcul des WCETs de plus en plus laborieux.

Nous avons pu ainsi constater, pendant longtemps, que les méthodes statiques ont peiné à s'adapter aux multiples innovations des architectures matérielles. Ce qui a logiquement freiné leurs utilisations mais qui a aussi fait de cette dicipline un domaine prisé par de nombreux chercheurs. Cet intérêt croissant a mené à la mise en œuvre de nombreuses méthodes d'analyse capables de fournir des estimations fiables des WCETs.

1.5 Méthode OTAWA

OTAWA(Open Tool for Adaptative WCET Analysis) est un outil développé par deux chercheurs : Hugues Cassé and Pascal Sainrat [11]. Cet outil « open source » permet une estimation de temps d'exécution pire-cas (WCET) par analyse statique, utilisable sous la forme d'une bibliothèque C ++ (licence LGPL). Il intègre un certain nombre de facilités pour la manipulation de code binaire et a été conçu pour supporter différentes architectures.

Grâce à son architecture modulaire, OTAWA est adaptable à diverses techniques de calcul de WCET. Il est basé sur un ensemble d'analyseurs pouvant réaliser des tâches diverses, de la construction du CFG à partir du binaire jusqu'au calcul final du WCET. Un sustème d'annotation (propriété) permet de stocker des informations (données d'entrée, résultats intermédiaires ou finaux) sur le programme à analyser. Un système de dépendance entre analyseurs permet à OTAWA d'enchainer l'utilisation des analyseurs

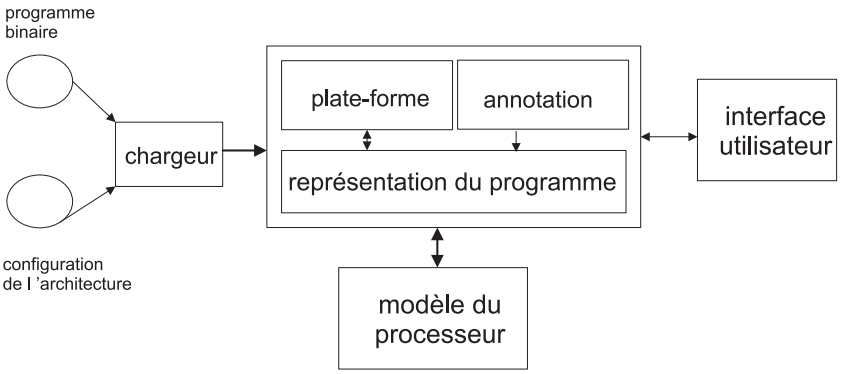


FIGURE 1.4: Méthode otawa

dans le bon ordre pour aboutir au WCET. Pour le calcul final du WCET, OTAWA utilise actuellement $lp - solve$, un logiciel libre de résolution de systèmes ILP.

L'outil OTAWA comporte de nombreux avantages comme sa rapidité d'exécution et sa facilité d'adaptation aux nouvelles architectures. En effet, cette adaptation consiste uniquement à fournir les caractéristiques du processeur sous forme d'un fichier XML (voir figure 1.4). Cependant, cette prise en compte du processeur est partielle, ainsi les résultats fournis par l'outil surapproximent généralement un peu trop les résultats réels.

1.6 Méthode AbsInt

Ces dernières années, la méthode statique la plus aboutie est celle développée par l'équipe AbsInt.

Cette méthode apparue en 1999, refléte les travaux de différents chercheurs : FERDINAND, C., KASTNER, D., LANGENBACH, M., MARTIN, F., SCHMIDT, M., SCHNEIDER, J., THEILING, H., THESING, S., qui ont tous collaboré sous la direction du Pr WILHELM, R [24, 64].

Les WCETs sont calculés, comme le montre la figure 1.5, par une succession d'analyses.

1.6.1 Description générale de la méthode

En premier lieu, un graphe de flot de contrôle (CFG) est extrait à partir du code binaire à analyser. Ensuite, sur ce CFG une analyse de valeurs est effectuée dont le but est de produire une sur-approximation des zones mémoires (intervalles) qui seront accédées. Ce résultat est, à son tour, exploité par l'étape suivante représentée par l'analyse du cache qui permet de classer les références mémoires en :

- Always hit : bloc toujours présent dans le cache.
- Always miss : bloc jamais présent dans le cache.
- Persistent : bloc chargé au plus une fois dans le cache, ce qui implique que pour ce bloc on aura tout au plus un seul cache miss.
- Not classified : bloc ne pouvant pas être classé dans un des groupes ci-dessus

Une fois cette classification effectuée pour tous les blocs, ce résultat est exploité par l'analyse suivante, dont le but est de permettre de définir les états éventuels du pipeline à chaque point d'exécution du programme à analyser. Cela permet d'identifier les temps d'exécution des différent blocs d'instruction formant le programme à analyser. Notons que les mises en œuvre de ces différentes analyses s'appuient sur l'interprétation abstraite [16, 17]. Cette technique est largement utilisée dans la vérification de programmes, et permet d'associer des valeurs concrètes à des valeurs abstraites (un ensemble de valeurs

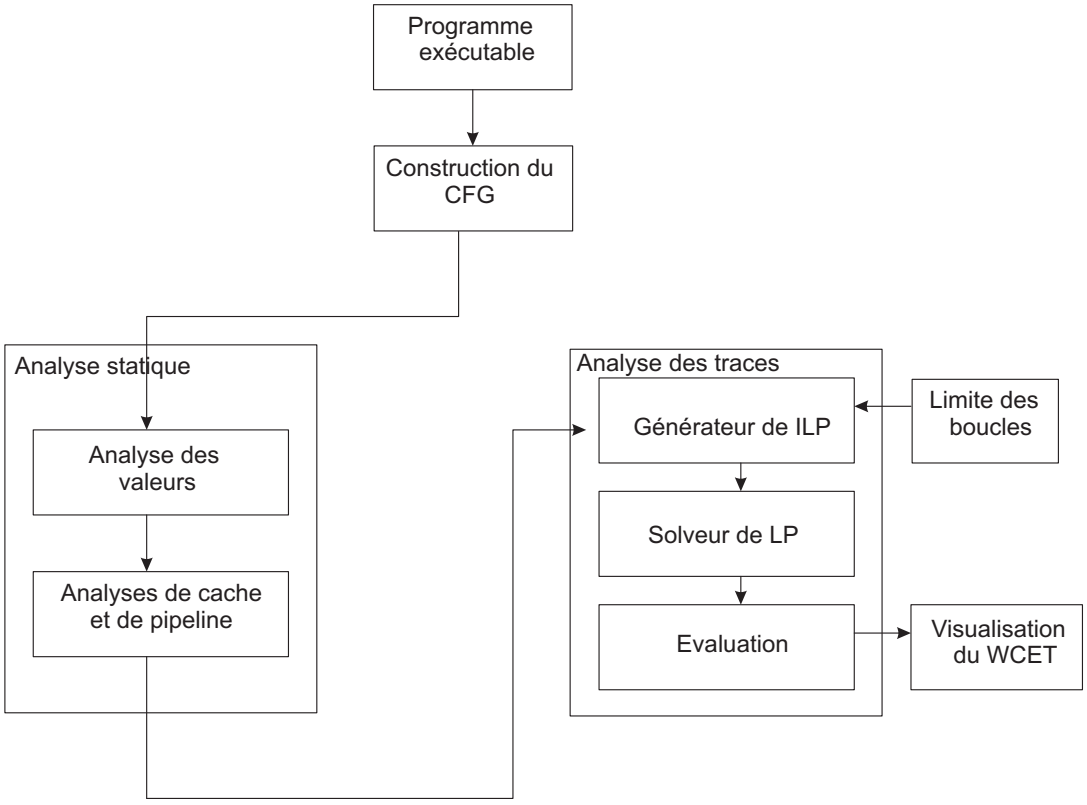


FIGURE 1.5: Méthode AbsInt

concrètes peut être représenté par une valeur abstraite).

Les différents résultats obtenus, durant les étapes précédentes, sont enfin exploités conjointement au code source par une dernière analyse appelée analyse des chemins d'exécution. Cette dernière analyse conduit à la formulation d'un programmation linéaire en nombres entiers (appelés ILP pour « Integer Linear Programming »). La résolution du ILP permet de connaitre le plus long chemin d'exécution (WCET).

1.6.2 Etude détaillée de l'approche

Cette méthode est basée sur l'interprétation abstraite et la programmation linéaire en nombres entiers. Elle se compose de deux phases :

- Prédire fiablement le comportement du processeur exécutant un programme donné.
- Déterminer une trace d'exécution qui représente le pire temps d'exécution. Ces traces d'exécution sont exprimées sous forme de fonctions objectives à maximiser sous certaines contraintes.

1.6.2.1 Construction du graphe de flot de contrôle

La première phase de cette analyse vise à traduire le programme à analyser dans un formalisme exploitable. Ainsi, le programme est représenté par un graphe de flot de contrôle qui se compose de noeuds et d'arcs. Les noeuds représentent les blocs de base. Un bloc de base est une séquence d'instructions contigues qui s'exécute sans interruptions et qui ne contient pas d'instructions de branchement. Ainsi, le

découpage du programme est fait de manière à placer chaque instruction de saut à la fin du bloc auquel elle appartient. L'intérêt de ce graphe est de faciliter les analyses suivantes. Ainsi, comme nous allons le voir par la suite, pour simplifier l'analyse de la mémoire cache, chacun de ces blocs mémoires peut être contenu dans une même ligne de cache. De cette façon, chacun des noeuds du CFG est associée à une seule référence mémoire. Ceci permet durant l'analyse de cache de classer chaque noeud du CFG dans la même catégorie (noeud associé à un : bloc always hit ou bloc always miss).

1.6.2.2 Analyse des valeurs

Une analyse statique du comportement de la mémoire cache nécessite de connaitre les adresses des données du programme [66]. Cependant les adresses effectives ne sont connues qu'à l'exécution. Dans [17] une analyse d'intervalles est proposée. Cette analyse permet de calculer les adresses possibles de registres ou de variables. Ainsi, un intervalle est déterminé pour un point de l'exécution du programme. Cet intervalle contient l'ensemble des valeurs possibles des adresses.

Le domaine des intervalles est donné par :

$$\mathbb{I} = \{[l,u] | l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

Cette définition n'admet que les intervalles qui représentent des ensembles non-vide d'entiers. L'ensemble des intervalles est ordonné par $\sqsubseteq$ défini comme suit :

$$[l_1,u_1] \sqsubseteq_{\mathbb{I}} [l_2,u_2] \text{ si et seulement si } l_2 \leq l_1 \wedge u_1 \leq u_2$$

la plus petite limite supérieure et la plus grande limite inférieure de deux intervalles sont définies par :

$$[l_1,u_1] \sqcup [l_2,u_2] = [min\{l_1,l_2\}, max\{u_1,u_2\}]$$

$$[l_1,u_1] \sqcap [l_2,u_2] = [max\{l_1,l_2\}, min\{u_1,u_2\}] \text{ , si } max\{l_1,l_2\} \leq min\{u_1,u_2\}$$

Une fonction $\beta_{\mathbb{I}}$ qui associe un entier à un intervalle singleton est définie par : $\beta_{\mathbb{I}}(z) = [z,z]$. La fonction d'abstraction $\alpha_{\mathbb{I}}$ associe un sous-ensemble $M \subseteq \mathbb{Z}$ d'entiers à un intervalle comportant un élément inférieur et un élément superieur de M : $\alpha_{\mathbb{I}}(M) = [inf_{z \in M} z, sup_{z \in M} z]$. La fonction de concrétisation $\gamma_{\mathbb{I}}$, représentant les valeurs et les intervalles concrets, est définie par :
$\gamma_{\mathbb{I}}([l,u]) = \{z \in \mathbb{Z} | l \leq z \leq u\}$.

Pour obtenir la sémantique abstraite, quelques opérations arithmétiques sont définies :

- L'addition de deux intervalles : $[l_1,u_1] +^{\sharp} [l_2,u_2] = [l_1 + l_2, u_1 + u_2]$ où $-\infty + \_ = -\infty$ et $+\infty + \_ = +\infty$ (le symbole _ représente n'importe quelle valeur).
- l'opérateur unaire moins : $-^{\sharp}[l,u] = [-u,-l]$.
- la multiplication et la division d'intervalles sont très difficiles à exprimer. Pour ce type d'opérations ainsi que pour la comparaison d'intervalles, se référer à [66].

La terminaison de l'analyse d'intervalles requiert des effort supplémentaires car l'ordre partiel du domaine $\mathbb{I}$, représente une chaine ascendante infinie, exemple :

$$[0,0] \sqsubset [0,1] \sqsubset [0,2] \sqsubset [-1,2] \sqsubset .....$$

Ainsi, dans le but de forcer la terminaison du calcul du point fixe, l'opérateur d'élargissement est utilisé (widening). Dans l'analyse de valeur proposée par AbsInt, une limite supérieure d'un intervalle qui augmente est immédiatement mise à ∞. De la même manière une limite inférieure qui diminue est mise à $-\infty$. Cette analyse produit des informations pertinentes qui servent à l'analyse de la mémoire cache. Plus les intervalles identifiés seront petits et meilleures seront les performances de l'analyse.

1.6.2.3 Abstraction des composants

Description de la mémoire cache

- Sa capacité : le nombre d'octets qu'elle peut contenir
- La taille de ses lignes (aussi appelée blocs mémoires) : le nombre d'octets transférés lors d'un défaut de cache. Ainsi, la mémoire cache peut contenir au plus n=($capacité/taille\ de\ ses\ lignes$) blocs.
- L'associativité : le nombre d'emplacements dans le cache qu'un bloc mémoire est susceptible d'occuper.

Si un bloc peut occuper n'importe quel emplacement, la mémoire cache est qualifiée de totalement associative (fully associative). Mais s'il ne peut en occuper qu'un seul alors la mémoire cache est qualifiée de directement associée (direct mapped). Si un bloc mémoire peut occuper n (nombre entier positif) emplacements, alors la mémoire cache est qualifiée de n voies associatives (n-way set associatve). Ainsi, dans le cas d'une mémoire cache totalement associative, une ligne du cache doit être choisie pour être remplacée à chaque fois que le cache est plein et que le processaur requiert une nouvelle donnée. Pour cela une statégie de remplacement doit être définie. Les stratégies les plus communes sont : la LRU (least recent used), la FIFO(first in first out) et le random.

Dans le but de décrire l'analyse de cache proposée par AbsInt, nous nous restreindrons à la sémantique des caches totalement associatifs qui implémentent une stratégie de remplacement de type LRU.

Sémantique de la mémoire cache
Dans la suite, nous considérons une mémoire cache (totalement associative) comme un ensemble de lignes $L = \{l_1, l_2...., l_n\}$ et son contenu (Store) comme un ensemble $S = \{s_1, s_2...., s_m\}$. Pour indiquer l'absence de blocs mémoires dans une ligne de cache, l'élément I est introduit ; $S' = S \cup I$.

Définition 1 ***état concret du cache*** *Un état concret de la mémoire cache est une fonction $c : L \rightarrow S'$. C_c représente l'ensemble de tous les états concrets de la mémoire cache. L'état initial du cache c_I associe toutes les lignes du cache à I.*

Si $c(l_i) = s_y$ pour un état concret du cache c, alors i représente l'âge relatif du bloc mémoire selon la stratégie de remplacement LRU et ne représente pas nécessairement sa position physique au niveau de la mémoire cache. La fonction de mise à jour décrit les changements qui surviennent au niveau de la mémoire cache lorsqu'un bloc mémoire est requis. Ainsi, le bloc mémoire s_x est transféré à l'emplacement l_1 si ce dernier était déjà dans le cache et tous les blocs mémoires les plus anciennement utilisés augmentent leurs âges respectifs de 1. Autrement dit, ils sont décalés d'une ligne (voir figure 1.6). A présent, si le bloc mémoire requis n'est pas présent dans le cache, il est chargé à l'emplacement l_1 après que les blocs mémoires présents dans le cache soient décalés d'une ligne. Dans ce cas, si la mémoire cache est pleine, le bloc le plus anciennement référencé (utilisé) est retiré du cache.

Définition 2 ***mise à jour du cache*** *Une fonction de mise à jour du cache $U : C_c \times S \rightarrow C_c$ détermine le nouvel état de la mémoire cache en fonction d'un état initial du cache et du bloc mémoire requis.*

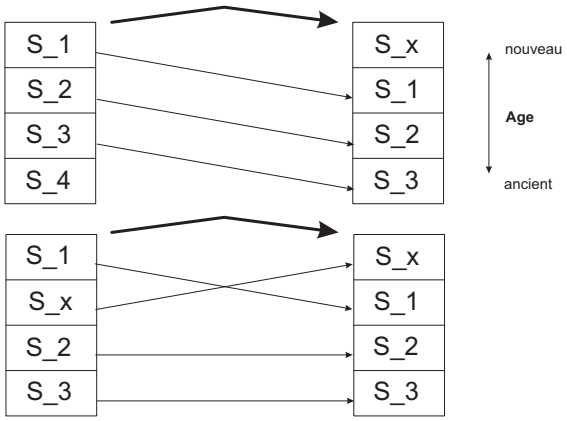


FIGURE 1.6: Mise à jour (d'une partie) d'une mémoire cache totalement associatif

Représentation du flot de contrôle Comme mentionné précédemment, le programme est représenté par un flot de contrôle qui se compose de noeuds et d'arcs. Les noeuds représentent les blocs de base. Un bloc de base est une séquence continue d'instructions qui s'exécute sans interruptions. Pour une analyse de mémoire cache, il est plus pratique, de représenter au niveau de chaque noeud une seule référence mémoire. Ainsi, les noeuds pourraient représenter les différentes instructions qui interagissent avec la mémoire. Il existe donc une relation associant les noeuds du flot de contrôle aux blocs mémoires. $F : V \rightarrow S^*$ (où V est un noeud du CFG et S^* le contenu du cache à ce point d'exécution) . Les effets de ces blocs mémoires sur la mémoire cache peuvent être décrit à l'aide de la fonction de mise à jour. Ainsi, la définition de la fonction U est étendue aux blocs mémoires par une séquence de compositions $U(c, (s_{x_1}, ..., s_{x_y})) = U(....(U(c, s_{x_1}))..., s_{x_y})$
L'état du cache pour une trace $(k_1, ..., k_p)$ dans le graphe du flot de contrôle est donnée en applicant U à l'état initial c_I de la mémoire cache. La concaténation de tous les blocs mémoires le long de cette trace est donnée par : $U(c_I, (F(k_1). \ ... \ .F(k_p))$.

La *collecting semantics* d'un programme rassemble, à chaque point d'exécution, l'ensemble des états que le programme est susceptible d'atteindre. Une sémantique sur laquelle se base une analyse de cache doit modéliser le contenu du cache au niveau des états du programme. Si, la collecting sémantics de la mémoire cache pouvait être calculée, l'ensemble des contenus possibles de la mémoire cache pourrait être obtenu, en remplaçant chaque état du programme par le contenu possible de sa mémoire cache. Cependant, la collecting sémantics n'est en général pas calculable.

Dans [64], les auteurs expliquent qu'une solution envisageable pour obtenir une sémantique permettant l'analyse du cache, est de se restreindre aux influences qu'a le programme sur cette mémoire (exemple les références mémoires). Ainsi, seuls ces effets seront modélisés par la fonction de mise à jour du cache U. Cette sémantique plus grossière (**coarser semantics**) peut contenir des traces d'exécution infaisables. C'est pourquoi, la collecting sémantics d'un programme se limite à calculer, à chaque point d'exécution, un sur-ensemble de tous les états concrets de la mémoire cache.

Définition 3 ***Collecting semantics de la mémoire cache***
La collecting semantics de la mémoire cache, pour un pogramme donné, est :
$C_{coll}(p) = \{U(c_1, F(k_1), ..., F(k_n))|(k_1, ..., k_n)$ *traces d'exécution dans le CFG qui mènent à p* $\}$

Cette collecting semantics peut être calculée mais est souvent de taille importante. Ainsi, une autre étape d'abstraction est nécessaire pour rendre le modèle du cache plus compact. Cette nouvelle représen-

tation est appelée "état abstraits du cache". Selon [25], les informations contenues dans un état abstrait permettent la déduction d'une information fiable concernant les états concrets que cet état représente. Seule la précision de ces informations peut être diminuée (sur-emsemble d'états).

Sémantique abstraite
L'analyse d'un programme se fait par la déterminaison d'un domaine abstrait et par des fonctions d'abstraction appelées *fonctions de transfert*.

L'analyse de la mémoire cache se compose de trois analyses :

- La must analysis : détermine l'ensemble des blocs mémoires présents dans le cache pour chaque point de l'exécution.
- La may analysis : détermine tous les blocs mémoires qui pourraient être dans le cache à un point précis de l'exécution. Cette analyse est utilisée pour identifier l'absence d'un bloc mémoire de la mémoire cache.
- la persistence analysis : détermine si un bloc mémoire est susceptible d'être retiré du cache. Ainsi, si un bloc mémoire est identifié comme étant persistent (jamais retiré du cache), il ne sera chargé qu'une seule fois dans le cache.

Ces analyses sont utilisées pour classer les différents blocs mémoires selon la classification exposées dans la partie 1.6.1.

Définition 4 ***Etat abstrait du cache***
Un état abstrait du cache $\hat{c} : L \rightarrow 2^S$ associe des lignes de cache à un ensemble de blocs mémoires. $\hat{C}$ représente l'ensemble des tous les états abstraits du cache.

Comme pour le cas du modèle concret de la mémoire cache, il existe une correspondance entre la position d'un bloc mémoire dans un modèle abstrait du cache et son âge. Cependant les domaines abstraits auquels appartiennent les états de la mémoire cache ont un ordre partiel différent des domaines concrets. Ainsi, l'interprétation d'un état abstrait est différente durant l'analyse.

Les fonctions suivantes relient les domaines concrets aux domaines abstraits :

- Fonction d'extraction (extr) : associe un état concret à un état abstrait.
- Fonction d'abstraction (abstr) : associe un ensemble d'états concret à leur meilleure représentation dans le domaine abstrait. Cette fonction est déduite de la fonction d'extraction.
- La fonction de concrétisation (concr) : associe un état abstrait du cache à l'ensemble des états concrets qu'il représente.

Définition 5 ***Fonctions d'extraction, d'abstraction et de concrétisation***
la fonction d'extraction $extr : C_c \rightarrow \hat{C}$ forme un singleton à partir d'ensembles d'images d'états concrets du cache. $extr(c)(l_i) = \{s_x\}$ si $c(l_i) = s_x$.
La fonction abstraction abstr :$2^{C_c} \rightarrow \hat{C}$ est définie par $abstr(C) = \sqcup\{extr(c)|c \in C\}$.
La fonction de concrétisation concr :$\hat{C} \rightarrow 2^{C_c}$ est définie par $concr(\hat{C}) = \{c|extr(c) \sqsubseteq \hat{c}\}$.

Les fonctions de transfert (les fonctions de mise à jour abstraite voir figure 1.7), seront toutes représentées par $\hat{U}$. Ces fonctions décrivent les effets des noeuds du flot de contrôle sur les éléments du domaine abstrait. Elles ont deux tâches à réaliser :

- Rafraichîr les blocs mémoires (les insérer dans la ligne de cache la plus récente).
- Augmenter l'âge des blocs mémoires présents dans le modèle abstrait du cache.

La Must analysis. Cette analyse détermine l'ensemble des blocs mémoires présents dans le cache pour chaque point d'exécution. Ainsi, plus cet ensemble est grand et plus l'analyse est performante.

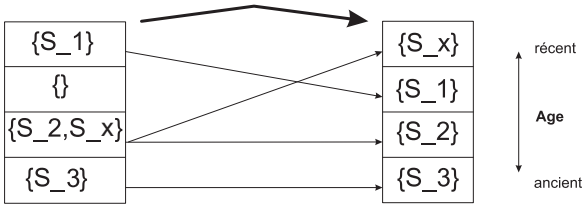


FIGURE 1.7: Mise à jour (d'une partie) d'un modèle abstrait de mémoire cache totalement associatif

L'âge associé à chaque bloc correspond à la durée de présence du bloc dans le cache. Ainsi, l'ordre partiel du domaine utilisé est comme suit : Soit un état abstrait du cache $\hat{c}$. Les éléments avant $\hat{c}$ dans ce domaine (les éléments moins précis) sont les états qui, soit, ne contiennent pas certains blocs mémoires présents dans $\hat{c}$, soit, contiennent ces blocs associés à des âges plus grands. Ainsi, l'opérateur $\sqcup$ appliqué à deux états abstraits du cache $\hat{c_1}$ et $\hat{c_2}$, produit un état $\hat{c}$ contenant uniquement les blocs mémoires présents dans $\hat{c_1}$ et $\hat{c_2}$. A chaque bloc mémoire est associé le maximum des âges qui leur étaient associés dans $\hat{c_1}$ et $\hat{c_2}$ (voir figure 1.8). Les positions des blocs mémoires dans le modèle abstrait du cache sont donc les limites supérieures des âges de ces blocs mémoires dans le modèle concret du cache. La concrétisation d'un état abstrait $\hat{c}$, produit l'ensemble de tous les états concrets du cache. Cet ensemble contient tous les blocs mémoires présents dans $\hat{c}$ associés à des valeurs d'âge plus petites que dans $\hat{c}$.

La solution de la must analysis s'exprime ainsi : admettons que $\hat{c}$ soit un état abstrait du cache à certains points d'exécution. Si $s_x \in \hat{c}(l_i)$ pour une ligne l_i alors s_x est toujours présent dans le cache à chaque fois que l'exécution atteint ce point de programme. s_x est donc classé comme étant toujours présent dans le cache (always hit). Par ailleurs, si $s_x \in \hat{c}(l_i)$, cela signifie également que s_x est présent dans le cache au moins pour les $n - i$ (n nombre maximal de blocs contenus dans le cache) prochains accès au cache qui font intervenir, soit, des blocs mémoires absents du cache, soit des blocs plus anciens que ceux présents dans $\hat{c}$. La définition de "plus ancien" est donnée par : s_a plus ancien que s_b : $\exists l_i, l_j$: $s_a \in \hat{c}(l_i), s_b \in \hat{c}(l_j), i > j$

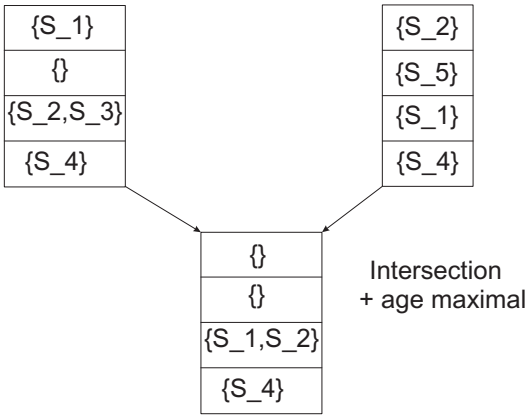


FIGURE 1.8: Conbinaison pour la must analysis

La may analysis Pour déterminer si un bloc mémoire s_x n'est jamais présent dans le cache, l'ensemble des blocs mémoires qui *pourraient* être dans le cache est calculé. L'information pertinente est qu'un bloc

mémoire ne soit pas contenu dans cet ensemble. Ainsi, ce bloc pourra être classé comme étant toujours absent du cache à chaque fois que l'exécution atteint ce point du programme. Donc plus cet ensemble est petit et plus l'analyse est performante. Les blocs les plus anciens sont plus rapidement retirés du cache que les nouveaux. Ainsi, l'ordre partiel du domaine utilisé est comme suit : Soit un état abstrait du cache $\hat{c}$, les éléments avant $\hat{c}$ dans ce domaine (les éléments mois précis) sont les états qui, soit, contiennent des blocs mémoires qui se sont pas présents dans $\hat{c}$, soit, contiennent des blocs présents dans $\hat{c}$ mais associés à des âges plus petits.

Ainsi, l'opérateur $\sqcup$, appliqué à deux états abstraits du cache $\hat{c_1}$ et $\hat{c_2}$, produit un état $\hat{c}$ contenant uniquement les blocs mémoires présents dans $\hat{c_1}$ et $\hat{c_2}$. A chaque bloc mémoire est associé le minimum des âges qui lui étaient associés dans $\hat{c_1}$ et $\hat{c_2}$ (voir figure 1.9).

Les positions des blocs mémoires dans le modèle abstrait du cache sont donc les limites inférieures des âges de ces blocs mémoires dans le modèle concret du cache.

La solution de la may analysis s'exprime ainsi : s_x est contenu dans l'état abstrait du cache $\hat{c}$ signifie que s_x peut être dans le cache à certains points d'exécution.

Le bloc mémoire $s_x \in \hat{c}(l_i)$ est sûrement retiré de la mémoire cache après, au plus, $n - i + 1$ (n nombre maximal de blocs contenus dans le cache) accès au cache qui font intervenir, soit, des blocs mémoires absents du cache, soit des blocs plus anciens ou du même âge que s_x. Ceci dans l'éventualité qu'aucun accès mémoire ne fasse intervenir s_x.

La définition de "plus ancien ou du même âge" est donnée par : s_a plus ancien ou du même âge que s_b : $\exists l_i, l_j : s_a \in \hat{c}(l_i), s_b \in \hat{c}(l_j), i \geq j$.
Si s_x n'est pas contenu dans $\hat{c}(l_i)$ quelque soit l_i, alors s_x est classé comme étant toujours absent du cache (always miss).

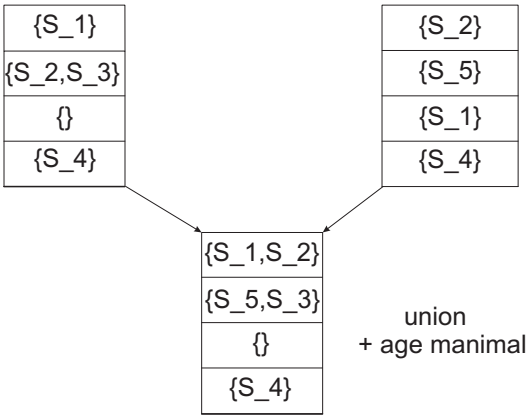


FIGURE 1.9: Conbinaison pour la may analysis

La persistent analysis Pour améliorer les performances de l'analyse de cache [58], la "classe persistent" a été rajoutée. Elle contient les blocs mémoires dont la première exécution peut résulter en un cache hit ou cache miss, mais dont toutes les exécutions suivantes résulteront en un cache hit. Un état abstrait du cache $\hat{c}$ contenant le bloc mémoire s_x est interprété de la façon suivante : Si $s_x \in \hat{c}(l_y)$ pour $y \in \{1, ..., n\}$ alors s_x n'est pas remplacé si ce bloc était présent dans le cache. Ainsi, si ce bloc n'a pas été classé comme étant toujours dans le cache, il sera classsé comme étant persistent.

Lorsqu'un bloc mémoire est trop ancien, il est envoyé vers une ligne virtuelle du cache $l_\top$. Cette ligne contient les blocs mémoires qui ont été retirés du cache.

Dans cette analyse la fonction qui permet de fusionner les lignes du cache est semblable à la fontion d'union. Si un bloc mémoire s a deux âges différents dans deux états abstraits du cache, alors cette fonction selectionnera l'âge maximal (voir figure 1.10).

La solution de la persistent analysis s'exprime de la façon suivante : admettons que $\hat{c}$ soit un état abstrait du cache à certains points d'exécution, si s_x n'est pas contenu dans $\hat{c}(l_\top)$ alors ce bloc n'est jamais retiré du cache. Ainsi, s_x est classé comme étant (persistent).

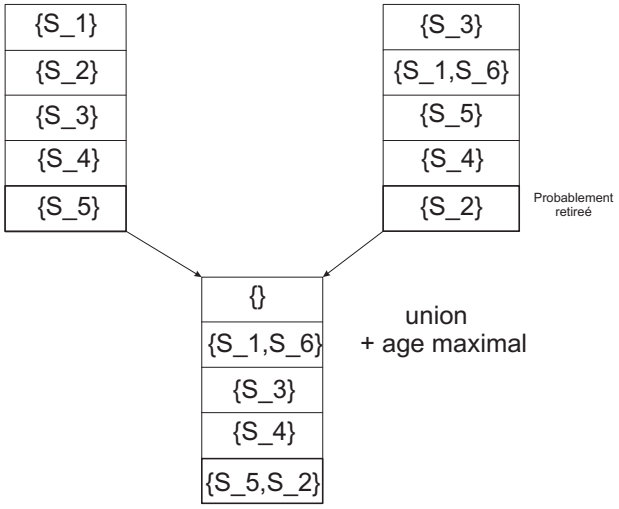


FIGURE 1.10: Conbinaison pour la persistent analysis

Terminaison de l'analyse Le cache a un nombre fini de lignes et à chaque point d'exécution du programme le nombre de blocs mémoires représentés est également fini. Ainsi, le domaine abstrait du cache $\hat{c} : L \rightarrow 2^S$ est fini. Ce qui assure que toutes les chaînes ascendantes sont finies. De plus la fonction de mise à jour abstraite $\hat{U}$ est monotone. Et donc la terminaison de l'analyse est garantie.

Analyse du pipeline

Un pipeline peut être considéré comme une machine à états finis (FSM : finit state machine). Ainsi, l'analyse du pipeline est définie comme un calcul des ensembles d'états du FSM [59]. L'efficacité de l'approche se base sur la représentation du FSM en diagramme de décisions binaires (BDD : binary decision diagram).

Définition 6 *Une machine à états finis M est un triplet (Q, I, T) où Q est l'ensemble d'états, I l'ensemble des valeurs d'entrée et T=$(Q \times I \times Q)$ l'ensemble des transitions.*

Chaque ensemble d'états du FSM $A \subseteq Q$ peut être associé à une fonction qui le caractérise : $\mathbf{A}$: $Q \rightarrow \{0,1\}$; $\mathbf{A}(x) = 1 \Leftrightarrow x \in \mathbf{A}$. De la même façon, la fonction de transition peut être caractérisée par $T : Q \times I \times Q \rightarrow \{0,1\}$; $T(x,i,y) = 1 \Leftrightarrow (x,i,y) \in T$. Il est commun de représenter l'ensemble des états du FSM ainsi que ses transitions par des fonctions représentées sous forme de BDDs. Cette représentation a l'avantage d'être plus compacte.

De plus, une représentation matérielle (Hardware design) se compose d'un ensemble interconnecté de bascules à deux états (bascule D) et de portes. Une représentation comportant n bascules avec m connections en entrée est associée à un FSM dont les états appartiennent à $Q = \{0,1\}^n$ et les entrées appartiennent à $I = \{0,1\}^m$.

Partant d'une modélisation du matériel en FSM, l'analyse du pipeline permet de calculer un point fixe sur le domaine $P(Q)$ des états du pipeline. Le plus petit point fixe (LFP :Least fixed point) contient tous les états du FSM qui sont atteignables à un point du programme. Le FSM contient un compteur de cycles pour chaque bloc de base (un nombre de cycle d'exécution). Ainsi, le WCET d'un bloc de base B est identifié en sélectionnant, lorque la dernière instruction appartenant à B termine son exécution, l'état dont le compteur de cycle a la plus grande valeur.

Définition 7 *Partant d'un FSM (Q, I, T) et d'un ensemble d'états $A \subseteq Q$, l'image de A : Img$(A) \subseteq Q$ est l'ensemble des états atteignables par A suivant T.*

Simplification du modèle de pipeline

Verilog ou VHDL sont des languages qui permettent de décrire de façon concise le hardware, sous forme de bascules et de logiques de mise à jour. Il a été démontré que ces spécifications peuvent être compilées dans une machine d'états finis temporisées [67].

Dans l'exemple de la figure 1.11, un exemple de code Verilog décrivant un FSM est présenté. Notons que reg[k :g] représente un ensemble de $(g - k) + 1$ variables d'états du FSM ayant les valeurs 0 ou 1. Admettons que ce FSM soit un modèle très simplifié de pipeline et que son pointeur d'instructions $instr$, puisse référencer 3 instructions. Le compteur $delay$ doit donc être initialisé avec le délais d'exécution de chaque instruction. Le compteur de cycles, $cycles$, compte les cycles d'exécution de blocs de base qui ne dépasse pas 7 cycles. L'exécution d'un cycle se fait en 3 étapes : clk_first, clk_second et clk_third . A chaque fois que le compteur $delay$ atteint 0 et que le signal clk_second est activé, une nouvelle valeur de $delay$ est lue grâce à la fonction $get\_delay()$. Similairement, la nouvelle valeur de $instr$ est obtenue grâce à la fonction $get\_next\_instr()$ lorsque la valeur de $delay$ atteint 0 et que clk_third est activé.

Lorsqu'un FSM est généré pour cette description, les fonctions d'entrée sont modélisées comme étant des transitions non-déterministes. Ainsi, le calcul de l'image d'un état où $delay$ atteint 0, mène à 3 successeurs possibles ($delay$ prend les valeurs 0,1 ou 2).

Fonction de transfert

La fonction de transfert, durant l'analyse de pipeline, calcule les successeurs des états du FSM. En général ceci représente un calcul d'images avec la restriction d'identifier uniquement les états atteignables. Le calcul d'images détermine l'ensemble des états atteignables pour toutes les valeurs d'entrée possibles. Admettons que A_0 soit l'ensemble d'états initiaux du FSM (Q, I, T). Alors la formule $A_{k+1} = A_k \cup Img(A_k)$ permet le calcul d'un point fixe (ensemble contenant tous les états possibles).

La difficulté de représenter les entrées concrètes du programme peut être surmontée en divisant le problème en deux parties :

- (1) Construire des BDDs contenant les états où ces entrées sont lues.
- (2) Construire d'autres BDDs pour ces entrées concrètes.

Il est important de garder à l'esprit que les variables du BDD sont les bascules de la représentation du matériel. Admettons que . représente la conjonction de BDD et que ¬ représente la négation d'une variable. Pour l'exemple de la figure 1.11, l'état où le $delay$ de l'instruction 1 est lu par $get\_delay()$ peut être représenté par :

$J_1 = instr < 0 > .\neg instr < 1 > .\neg delay < 0 > .\neg delay < 1 > .clk\_second$

Dans cet état, le pointeur d'instructions est égal à 1, la valeur du $delay$ à 0 et le signal clk_second est actif. La table 1.12 spécifie que le $delay$ de l'instruction 1 est égal à 2.

$C_1 = \neg delay < 0 > .delay < 1 >$

```
reg [0:3] cycles;
reg [0:1] instr;
reg[0:1] delay;

initial cycles=0;
initial instr=0;
initial delay=0;

always @(clk_first) begin
    if(cycles == 7)
        cycles = 0;
    else
        cycles = cycles + 1;
end

always @(clk_second) begin
    if(delay == 0)
        delay = get_delay;
    else
        delay = delay - 1;
end

always @(clk_third) begin
    if(delay == 0)
        instr = get_next_instr();
end
```

FIGURE 1.11: Exemple de code Verilog représentant un simple FSM

instruction	delay
0	1
1	2
2	$\{0, 1\}$

FIGURE 1.12: Exemple d'entrée pour le modèle exposé par la figure 1.11

Les BDDs $\mathbf{J}_1$ et $\mathbf{C}_1$ peuvent être considérés comme des fonctions qui caractérisent l'ensemble des états J_1 et C_1. Ces derniers contiennent des variables dont les valeurs sont représentées par des BDDs. Pour un ensemble A_k d'états du FSM, les états successeurs, pour des entrées concrètes à ce point de l'exécution du programme, est calculé par la formule suivante :
$A_{k+1,1} = (Img(A_k \cap J_1)) \cap C_1$
Pour n entrées concrètes, les états successeurs qui ne requièrent pas d'informations sur les entrées concrètes sont calculées comme suit :
$A_{k+1,-} = Img(A_k \setminus \bigcup_{l \in [0,n]} J_l)$

Finalement, le point fixe de l'analyse du pipeline, comprenant n entrées concrètes du programme, peut être calculé de la façon suivante : $A_{k+1} = (\bigcup_{l \in [0,n]} A_{k+1,l}) \cup A_{k+1,-}$

Notons que les entrées indéterminées peuvent également être représentées en BDDs. Exemple le $delay$ de l'instruction 2 dans la figure 1.12, peut prendre les valeurs 0 ou 1. Le BDD de cette entrée est simple : $C_2 = \neg delay < 1 >$. L'ordre dans lequel les instructions sont analysées est également une entrée du modèle de la figure 1.11. Cette entrée peut être déduite du graphe de flot de contrôle du

programme et représentée de la même façon que l'entrée $delay$.

Le succès de cette analyse dépend de la taille des BDDs qui évolue durant l'analyse. Cette taille dépend principalement de l'ordre donné aux variables des BDDs. Identifier un BDD de taille minimale pour une fonction logique est algorithmiquement impossible. Cependant, il existe plusieurs heuristiques qui permettent d'identifier un ordre convenable à donner aux variables. Pour chaque modèle du pipeline, cet ordre ne doit être identifié qu'une seule fois.

Ainsi, cette analyse produit un sur-ensemble des états atteignables par le pipeline. A chacun de ces états est associé un temps maximal (le WCET d'un bloc de base). Ce temps est déduit en combinant les résultats des analyses du pipeline et de la mémoire cache. Ce temps est donc l'addition du temps maximal pris par un bloc de base pour s'exécuter au niveau du pipeline et des délais de ses accès mémoires.

1.6.2.4 Analyse des boucles

Généralement, les limites des boucles peuvent être déterminées que si les variables du programme sont rangées dans des registres du processeur [12]. La raison est que les données stockées dans la mémoire peuvent être altérées par les accès indéterminés à la mémoire.

Un exemple commun de boucles est donné par : $while(x > (x\_table + +))$.....;
Dans cet exemple, il subsiste le risque que x soit plus grand que tous les autres éléments de la table. Ainsi, l'analyse continuera à examiner les valeurs contenues dans la table même après avoir atteint le dernier élément de la table. C'est pourquoi avant cette analyse, il est important que le code soit modifié de façon à rajouter à la fin de la table la plus grande valeur possible. Ainsi, le nombre maximal d'itérations de la boucle sera limité par la taille de la table.

L'analyse de boucle doit résoudre différentes situations conflictuelles comme celle présentée ci-dessus. C'est pourquoi, cette analyse nécessite le plus possible l'intervention de l'utilisateur. Ainsi, l'utilisateur est encouragé à rajouter un maximum d'annotations de façon à améliorer les performances de cette analyse.

1.6.2.5 Analyse des traces d'exécution : ILP

Les différents résultats obtenus, durant les étapes précédentes, sont enfin exploités conjointement au code source par une dernière analyse appelée analyse des chemins d'exécution. Cette analyse se base sur la programmation linéaire (appelés ILP pour « Integer Linear Programming »). L'utilisation de l'ILP permet de décrire la structure du programme ainsi que l'ensemble de ses traces de manière très naturelle. La résolution de l'ensemble des contraintes décrivant la structure du programme permet d'obtenir des résultats précis.

Un problème formulé en ILP est composé de deux parties :
- la fonction objective (à maximiser)
- les contraintes associées aux variables de cette fonction.

Dans l'analyse proposée par AbsInt la fonction objective représente un nombre de cycles d'horloge. Les variables de cette fonction sont les temps d'exécution des blocs de base du programme. Admettons que t_b^v soit le temps d'exécution d'un bloc de base b s'exécutant dans un contexte v. Afin de déterminer le WCET, le temps d'exécution doit être maximiser. Ainsi pour une pire trace d'exécution $(b_1, v_1)...(b_k, v_k)$ associée aux durées $t_{b_i}^{v_i}$ pour $i = 1...k$, le temps correspond à : $f_{max} = \Sigma_{i=1}^{k} t_{b_i}^{v_i}$

Les contraintes décrivent le nombre de fois qu'un bloc de base est exécuté. Elles peuvent donc être générées automatiquement à partir du graphe de flot de contrôle. Cependant, l'intervention de l'utilisateur est souvent nécessaire dans le but d'améliorer la précision de l'analyse [64] (préciser certaines contraintes). De plus, les valeurs des limites des boucles doivent aussi être fournies par l'utilisateur car elles ne sont pas toujours automatiquement identifiables.

L'utilisation de l'ILP dans l'analyse des traces d'exécution a l'avantage de permettre à l'utilisateur de décrire de façon précise toutes les informations qu'il détient sur le programme. Dans l'approche AbsInt, des contraintes arbitraires peuvent être rajoutées par l'utilisateur de façon à améliorer les performances de l'analyse. L'analyseur de trace commence par générer les contraintes évidentes, puis rajoute toutes les contraintes fournies par l'utilisateur de façon à obtenir une estimation plus précise des WCETs.

1.6.2.6 Exécution de séquences d'instruction

Dans la méthode AbsInt, l'abstraction est très utilisée. Ceci permet *apparemment* d'analyser de longue séquence de code. Cependant, comme nous l'avons exposé dans la partie 1.6.2.1, le graphe de flot de contrôle découpe le programme en blocs mémoires. La taille de chacun de ces blocs n'exède pas celle d'une ligne de cache. La longueur du programme à analyser a donc un impact direct sur le nombre de noeuds du CFG. Plus le programme est grand et plus le nombre de noeuds est important. Par ailleurs, ces noeuds sont associés à des contraintes. Ainsi, lors de l'analyse des traces par ILP, le nombre de contraintes à considérer pour maximiser la fonction objective, sera conséquent. Ce qui risque de rendre l'identification de la plus longue trace d'exécution impossible.

1.6.2.7 Limites de la méthode

L'approche AbsInt se compose de plusieurs analyses. Ceci est un des points forts de cette approche, vu que cela permet d'utiliser différentes techniques à différents niveaux de l'étude. Exemple : l'utilisation de l'interprétation abstraite pour l'analyse des valeurs, du cache et du pipeline, alors que l'analyse des traces d'exécution se fait par ILP (Integer Linear Programming). Mais cette force peut aussi se révéler être une faiblesse. En effet, une imprécision, introduite par une analyse, est injectée à l'entrée de l'analyse suivante. C'est pourquoi, les analyses successives sont généralement apparentées à des accumulations d'erreurs. Afin d'éclaircir ce point, prenons l'exemple de l'exécution d'une instruction I (le même raisonnement est applicable à un bloc mémoire). L'analyse statique de valeurs identifie, pour cette instruction I, un intervalle d'adresse $[I_{adr_{min}}, I_{adr_{max}}]$, dans lequel est susceptible de se trouver l'instruction I. Cet intervalle est utilisé par l'analyse de cache en vu de classer l'instruction I dans l'une des catégories présentées dans la partie 1.6.1. Pour que l'analyse de cache soit performante il faut qu'à un point de l'exécution toutes les valeurs présentes dans l'intervalle soient dans le cache (cache hit) ou qu'elles soient toutes absentes du cache (cache miss). Autrement, l'instruction ne pourra pas être classée (not classified).

A présent, si durant l'analyse de valeurs, l'opérateur d'élargissement (widening) est utilisé, les limites de l'intervalle sont susceptibles d'appartenir à l'infini ($I_{adr_{max}} = \infty$, $I_{adr_{min}} = -\infty$). Rappelons, que l'utilisation de l'opérateur d'élargissement est utilisé systématiquement de façon à garantir la terminaison de l'analyse. Donc, même si ce cas ne se produit pas, l'intervalle calculé est potentiellement grand. Ce qui rend l'analyse de la mémoire cache imprécise.

De la même façon, l'analyse de cache a un impact sur l'analyse du pipeline. En effet, si une l'instruction I ne peut être classée, l'analyse de pipeline mènera à la définition d'un sur-ensemble important d'états de façon à prendre en considération tous les cas d'exécution possibles (instruction I : dans le cache, absente du cache, persistente). Evidemment, ce sur-ensemble contiendra plusieurs états inatteignables. Ainsi, lorsque l'analyse des chemins est réalisée, les temps que la fonction objective utilise pour identifier le WCET, seront des temps impossibles à obtenir durant une exécution réelles du programme (temps associés à des états inatteignables). Maximiser cette fonction revient à calculer un pire temps d'exécution assez éloigné de la réalité.

Par ailleurs, l'analyse des boucles n'est pas dénouée de problèmes. En effet, cette analyse sollicite une fréquente intervention de l'utilisateur, ce qui représente une source supplémentaire d'erreurs.

Ainsi, les désavantages de cette méthode apparaissent généralement lorsque la sûreté des résultats n'est plus un critère suffisant [29], mais que nous souhaitions aussi que ces résultats soient au plus près de la réalité. En effet, cette méthode travaille sur des sur-approximations d'ensembles (domaines abstraits), et a du mal à fournir des temps d'exécution précis (valeurs exactes non approximées).

Par ailleurs, concernant le cache et le pipeline, il est à noter que leurs analyses sont étroitement liées au hardware. Ainsi, pour chaque processeur, une contraignante étude est exigée (description du matériel, définition de sémantique concrète et abstraite, détermination de fonctions de transfert...). Ainsi, vu l'importance accordée actuellement à la sophistication des architectures, la phase de modélisation risque d'être un frein à l'utilisation de cette méthode. De plus, certains comportements, intégrés récemment dans les processeurs, ne peuvent pas être représentés correctement, exemple : le réodonnancement d'instructions (voir partie 2.2). Ce problème complexe a incité certaines recherches récentes au sein de l'équipe AbsInt. Ces travaux ont pour but d'adapter une partie des comportements du processeur à leurs analyses [47].

1.7 Récapitulatif des méthodes existantes

Un récapitulatif des méthodes de calculs des WCETs est présenté par les tableaux 1.1 et 1.2. Le tableau 1.1 présente une rapide comparaison concernant la facilité d'utilisation des méthodes présentées précédemment.

TABLE 1.1: " Récapitulatif 1"

Méthodes	Données en entrée	Facilité d'utilisation
Stop-watch	Jeux de tests.	Disposer d'un chronomètre.
date et time	Jeux de tests.	Disposer d'un OS UNIX.
Timer et Compteur	Jeux de tests.	Le processeur doit disposer d'un compteur interne.
Prof et Gprof	Jeux de tests.	Disposer d'un OS UNIX.
Analyseur logiciel	Jeux de tests.	Disposer d'un analyseur.
Lundqvist Stenstrom	Code exécutable.	Disposer d'un simulateur. Etendre la sémantique du simulateur.
RapiTime	Code source. Jeux de tests. Le modèle est le matériel effectif que les clients ont	Générer des jeux de tests pertinents permettant d'atteindre le pire nombre de tours de boucles.
HEPTANE	Code exécutable.	Disposer d'un simulateur.
OTAWA	Code exécutable. Description partielle de l'architecture	Description de l'architecture simple fichier XML.
AbsInt	Code exécutable. Description sommaire du contexte pour la fonction à analyser. (valeur du registre pointeur de pile, bornes de certaines boucles)	Fournir les bornes des boucles, une description de la mémoire pertinente Nécessite une version différente d'aiT pour chaque modèle – cela nécessite un coût de développement important)

Le tableau 1.2 présente un récapitulatif des capacités ainsi que des limites de chaque méthode. Les symboles N et O signifient respectivement NON et OUI. Ce tableau montre que :

- 1) La première famille de méthode (méthodes dynamiques) prend en compte systématiquement tous les mécanismes internes d'un processeur. Cependant, vu qu'elle se base essentiellement sur des mesures, la fiabilité des résultats n'est pas garantie.
- 2) Les méthodes semi-dynamiques donnent, quant à elles, des résultats fiables (à l'exception de la méthode RapiTime), néanmoins leurs utilisations sont contraintes par la disponibilité soit d'un simulateur soit du processeur lui même. Ainsi, ces méthodes sont difficilement utilisables en phase de conception lorsque l'analyse a pour unique but de choisir le processeur adéquat (le processeur, à plus faible coût, capable d'exécuter un ensemble de taches temps réelles). Par ailleurs, du fait de leurs prise en compte partielle de l'architecture (Lundqvist-Stenstrom et HEPTANE), elles donnent des résultats très sur-approximés.
- 3) Concernant la dernière méthode (méthode AbsInt), qui est la méthode de référence actuellement, nous constatons son impossibilité à s'adapter aux évolutions du processeur (effets du pipeline). De plus, cette analyse se base sur un mélange de sur-ensembles (domaines abstraits) et de traces implicite (ILP). Ainsi, cette méthode n'a pas la capacité de fournir des résultats exacts et cela même si son utilisation est restreinte à l'analyse d'une simple séquence d'instructions linéaires.

TABLE 1.2: " Récapitulatif 2"

Méthodes existantes	prise en compte des boucles	prise en compte des effets du pipeline	prise en compte de l'architecture	fiabilité des résultats	Sur-approximation < à 2 fois le résultat réel
Stop-watch	O	O	O	N	O
date et time	O	O	O	N	O
Timer et Compteur	O	O	O	N	O
Prof et Gprof	O	O	O	N	O
Analyseur logiciel	O	O	O	N	O
Lundqvist Stenstrom	O	O (si simul. dispo)	N (partielle)	O	N
RapiTime	O	O (si proc. dispo.)	O (si proc. dispo.)	N	O
HEPTANE	O	O (si simul. dispo)	N (partielle)	O	N
OTAWA	O	N	O (partielle)	O	N
AbsInt	O	N	O	O	N (jusqu'à 2.5 fois)

1.8 Conclusion

Cet état de l'art nous a permis de présenter deux familles de méthodes portant sur le calcul des WCETs : méthodes dynamiques et statiques. Par ailleurs, nous avons également présenté une nouvelle famille de méthodes qui se place à mi-chemin entre les méthodes statiques et dynamiques : l'analyse semi dynamique. Notre approche appartient également à cette nouvelle famille de méthodes. Cependant, elle prend en compte l'architecture matérielle sans qu'un simulateur ou que le processeur analysé soit disponible. Dans le chapitre suivant, nous exposons les motivations qui nous ont amené au développement d'une nouvelle méthode. Nous mettons en évidence les apports de notre approche dont l'objectif principal est de calculer précisément les WCETs des programmes qui s'exécutent sur des architectures complexes.

CHAPITRE 2

Motivation

Dans le chapitre précédent nous avons mis en évidence que la seule méthode capable de prendre en compte la complexité des processeurs est celle développée par AbsInt. Dans la suite, nous nous référerons souvent à cette approche afin de mettre en évidence l'intérêt de notre méthode.

- Dans l'analyse que nous proposons, la complexité des processeurs est aussi prise en compte. Cependant, contrairement à la méthode AbsInt qui nécessite une laborieuse phase de modélisation pour toute analyse d'un nouveau processeur, nous proposons un modèle exécutable (Time-accurate) d'une architecture, simple à mettre en oeuvre.
- Par ailleurs, notre méthode permet de prendre en compte les interactions entre les composants d'un processeur plus finement. En effet, l'utilisation de notre extension de l'exécution symbolique nous permet de considérer les multiples effets du pipeline (effets à court terme, effets à long terme et anomalies temporelles : voir partie 4.6) comme des comportements possibles du processeur.
- Au lieu d'adopter une logique d'abstraction qui consiste à implémenter une politique de fusionnement uniforme, nous proposons une méthode qui permet de contrôler dynamiquement la perte d'information en guidant les fusions durant l'exécution.

2.1 Intérêt du modèle exécutable : Cycle Accurate

De façon informelle notre modèle est un simple automate dont les états abstraient les composants du processeur. Les transitions sont, quant à elles, labélisées par une information temporelle. Ce label(i-i+1) représente le temps maximal pour que l'automate évolu de l'état(i) à l'état(i+1).

Durant l'exécution, cet automate évolu en fonction de ses entrées :

- 1) Exécuté avec des entrées standards (valeurs concrètes en entrée du modèle) le modèle prend en entrée un état courant $\mathcal{SC}$ du processeur et retourne deux informations :
  - l'état suivant $\mathcal{SC}'$ qui est différent de l'état courant.
  - le nombre de cycles d'horloge requis pour atteindre ce nouvel état.
- 2) Exécuté symboliquement (valeurs abstraites en entrée : voir partie 4.3), le modèle fournira en sortie tous les états directement atteignables par le processeur (couverture totale), ainsi que les temps associés à chaque changement d'état.

Contrairement à la méthode développée par AbsInt qui nécessite une phase de modélisation laborieuse (définition de sémantiques concrète et abstraite, détermination de fonctions de transfert...) à chaque fois qu'un nouveau processeur doit être analysé, nous proposons une modélisation beaucoup plus simple à mette en oeuvre. En effet, notre modèle de processeur peut être vu comme un simple automate composé d'états et de transitions, où chaque état représente l'ensemble des composants actifs [1] du

1. actifs dans le sens traitent des instructions.

processeur à ce point de l'exécution, et chaque transition est labélisée par le temps nécessaire à son tir [2].

Ainsi, l'intérêt d'utiliser un modèle exécutable dans notre analyse est double. D'une part, la méthode s'adapte rapidement aux changements d'architecture car la phase de modèlisation n'est pas très coûteuse. D'autre part, la finesse apportée par ce type de modélisation permet de capter avec précision les multiples comportements intra-processeur de bas niveau.

2.2 Prise en compte des interactions entre les composants du processeur

Notre modélisation s'appuie sur une représentation détaillée des unités du processeur. Ainsi, toutes les interactions intra-processeur peuvent être modélisées. Ces interactions sont très importantes car elles influent directement sur les temps d'exécution. En effet, grâce à ce type de modélisation nous pouvons estimer précisément les temps de transfert des données entre la mémoire cache et le pipeline ou représenter exactement ce qu'il se passe quand un processeur exécute des instructions dans le désordre. Sans cette finesse de modélisation, de multiples effets internes au processeur ne peuvent pas être pris en compte. Ce premier point illustre un des points forts de notre approche. En effet, aucune méthode existante ne prend en compte, avec une telle précision, les comportements des architectures. En règle générale, les comportements du processeur sont considérés partiellement ou généralisés grâce à des sur-ensembles d'états.

Prenons l'exemple du ré-ordonnancement d'instructions (exécution dans le désordre) traité avec la méthode AbsInt. Le ré-ordonnancement d'instructions est un mécanisme qui permet au processeur de décider dynamiquement, en fonction des disponibilités de ses unités et des dépendances de données, de l'instruction du programme à exécuter en premier. Ainsi, le processeur est rarement bloqué (en attente de données) et le temps d'exécution d'un groupe d'instructions est réduit. Identifier qu'à un instant précis le processeur est susceptible de changer le flot d'instructions qu'il exécute, nécessite l'énumération de tous ces états et de connaitre très précisemment l'historique de chaque trace. Ainsi, une analyse statique de la mémoire cache et du pipeline ne peut prendre en compte ce type de comportement. Ceci s'explique par le fait qu'une fois l'analyse du cache effectuée, le résultat est injecté en entrée du pipeline. Ainsi, durant l'analyse du pipeline si un possible ré-ordonnancement d'instructions est identifié, il ne pourra pas être pris en compte. En effet, l'analyse du cache est réalisée **en amont** en suivant un ordre de précédence des instructions dicté par le programme. Ainsi, dans le cas où le ré-ordonnancement serait pris en compte, l'ordre des instructions se retrouverait changé, ce qui compromettrait les résultats fournis par l'analyse du cache. Donc, le gain de temps apporté par ce type de mécanisme n'est jamais considéré par la méthode AbsInt, ce qui nuit à la précision des résultats.

Ceci n'est pas un cas isolé vu qu'actuellement la tendance pousse les concepteurs de processeurs, de plus en plus, à user de ce type de méchanisme pour améliorer les performences des architectures matérielles. En effet, la parallélisation des tâches est très prisée depuis que l'augmentation de la fréquence des processeurs montre ses limites. Ainsi, les méthodes d'analyse se doivent de s'adapter à ces évolutions matérielles. En ce sens les méthodes semi-dynamiques qui fournissent une couverture totale des comportements possibles des processeurs sont actuellement l'unique solution.

2.3 Maitrise de la perte de précision

Prendre en compte tous les comportements bas niveau n'est pas dénoué de problèmes. En effet, ceci nous contraint à faire une énumération d'états et implique qu'une explosion combinatoire du nombre d'états générés est possible. Une solution envisageable pourrait être, comme pour AbsInt, d'utiliser l'in-

2. le tir d'une transition signifie la franchir : passer de l'état dont sort cette transition à celui dans lequel elle arrive.

terprétation abstraite afin de faire correspondre à plusieurs états concrets, un ou plusieurs états abstraits. Ceci réduirait le nombre d'états mais serait très préjudiciable vis-à-vis de la précision des résultats.

Le problème, avec les politiques de fusionnement totalement statiques, est que la correspondance entre états concrets et abstraits est définie dès le départ. Ainsi, quelque soit l'historique d'exécution, la fusion se fera de la même façon. Nous proposons un exemple simple pour éclaircir ce point. Admettons que durant une exécution le processeur requiert une instruction. Cette dernière va d'abord être recherchée dans la mémoire cache. Si le cache la contient (cache hit) elle sera directement traitée. Dans le cas contraire une transaction mémoire est déclanchée de manière à récupérer cette instruction de la mémoire principale. A présent, analysons ce comportement plus en détail. Pour ce type de scénario une analyse de la mémoire cache classe la zone mémoire contenant l'adresse de l'instruction comme étant "inconnue" ("not classified" si nous nous référons à la méthode AbsInt), car en début d'analyse aucune information n'est disponible. Ainsi, les deux scénarios (cache miss et hit) sont possibles. Donc, une fois que cette information est injectée à l'entrée de l'analyseur de pipeline deux traces sont générées. Après quelques cycles, nécessaires au transfert de l'instruction, ces deux traces sont susceptibles d'être fusionnées. Cette fusion se fait en remplaçant les deux états finaux des deux traces précédentes par un état générique (abstrait). Le problème est que pour la trace reflétant le cache miss, la transaction mémoire qui a permis de récupérer l'instruction a tranféré également toutes les instructions contenues dans le même bloc mémoire (chargement du cache par ligne et non par instruction : voir partie 3.4.1.1). Cependant, cette information étant perdu, l'analyse se poursuit par un nouveau défaut de cache (cache miss). Ce type d'approche est donc très efficace en terme de réduction du nombre d'états générés mais mène à des abbérations (deux caches miss qui se succédent) qui sont très préjudiciables à la précision des résultats.

Partant de ce constat, nous avons choisi de définir une politique de fusionnement, certe moins aggressive en terme de réduction du nombre d'états, mais permettant de fournir des résultats plus proches de la réalité. Notre méthode à été élaborée de façon à prendre en compte le coté dynamique d'une exécution. Ainsi, contrairement à la méthode AbsInt qui impose une méthode générale uniforme quelque soit le contexte d'exécution (l'historique d'exécution), nous avons opté pour une méthode capable de s'adapter à l'évolution de l'analyse.

Ainsi, notre méthode de fusionnement a pour principale contrainte de préserver l'intégrité des états. Le but est de réduire au maximum le nombre d'états générés sans pour autant modifier la logique d'exécution suivie par le processeur.

2.4 L'approche proposée

Nous proposons une approche semi-dynamique formelle [5, 3, 4]. Cette méthode s'adapte à la sophistication croissante des architectures matérielles. Comme le montre la figure 2.1, la première phase de cette analyse consiste à mettre au point un modèle de l'architecture cible. Ce modèle doit respecter certaines contraintes. La plus importante est qu'il doit contenir des informations temporelles. Concrètement, ces informations temporelles correspondent aux temps associés à chacune des opérations de base que le processeur effectue. Ainsi, nous devons associer à chacune des opérations basiques effectuées par le processeur, le temps nécessaire à sa réalisation. L'expression "opérations basiques" englobe :

- les opérations effectuées par les unités de traitement (addition, multiplication, etc.),
- les tâches réalisées par les étages du pipeline (Fetcher, Dispatcher, etc.),
- la recherche de données/instructions au niveau des mémoires caches (hit :succes, miss : échec).

Associer à chaque opération basique, le temps nécessaire à son exécution est une tâche essentielle pour pouvoir fournir des valeurs exactes des temps d'exécution.

Une fois la modélisation achevée, nous exécutons symboliquement ce modèle (voir la partie 4.2). La partie la plus importante de cette analyse est de capter précisément toutes les interactions intra-processeur

(entre le pipeline, l'unité de branchement et la mémoire cache) de façon à obtenir une estimation précise des temps d'attente.

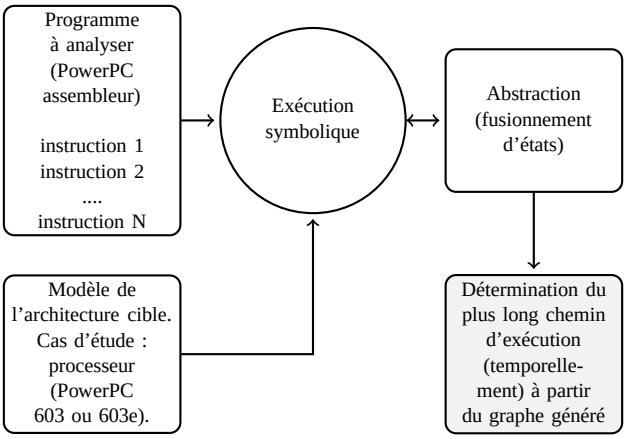


FIGURE 2.1: Approche proposée : Estimation des WCETs

L'approche s'exécute donc, principalement en deux phases :

Exécution symbolique conjointe du programme et du modèle du processeur :
Durant l'exécution symbolique du programme le modèle du processeur est utilisé pour calculer pour chaque point d'exécution tous les états que le processeur peut atteindre. La recherche de ces états se fait évidemment en respectant l'environnement d'exécution courant et l'historique.

Fusion maitrisée des états :
Afin d'éviter l'explosion du nombre des états engendrés durant l'exécution symbolique, les états similaires doivent être fusionnés. Sachant que l'approche proposée n'impose aucune restriction à la sémantique du modèle de l'architecture utilisée, une fusion mal étudiée pourrait mener à une très grande imprécision des résultats. Notre politique de fusionnement fonctionne comme suit : après plusieurs pas d'exécution symbolique, la méthode de fusionnement est lancée. Celle-ci commence par une analyse arrière qui lui permet d'identifier les états fusionnables. Ces états sont qualifiés de "fortement similaires". Ce qui signifie que plusieurs de leurs paramètres respectifs sont totalement similaires. Cette similitude entre les états garantit que leurs évolutions respectives futures seront quasiment identiques. Une fois ces états identifiés, ils sont fusionnés. Dès que toutes les fusions sont terminées, la méthode de fusionnement est stoppée de façon à ce que l'exécution symbolique puisse reprendre.

Lorsque le nombre d'états généré par l'exécution symbolique n'est plus compatible avec les ressources mémoires disponibles (dans ce cas le nombre d'états dépasse une valeur seuil), nous utilisons en complément de la méthode précédente, une méthode qui fait appel à la notion d'états faiblement similaires.

Cette fusion, plus agressive en terme de réduction du nombre d'états, a le désavantage d'introduire une imprécision sur les temps d'exécution calculés mais a l'avantage de garantir la terminaison de l'analyse.

L'analyse s'achève lorsque toutes les instructions ont été symboliquement exécutées. Ainsi, le graphe résultant peut être analysé afin d'en extraire le pire temps d'exécution.

2.5 conclusion

Dans cette partie nous avons, par de simples comparaisons, mis en évidence l'intérêt de notre approche. Ceci nous a aussi permis d'argumenter les choix pris durant l'élaboration de notre méthode. Dans le prochain chapitre, nous allons nous intéresser à la première étape de notre analyse qui consiste à fournir un modèle Time-accurate du processeur.

CHAPITRE 3

Modélisation du processeur

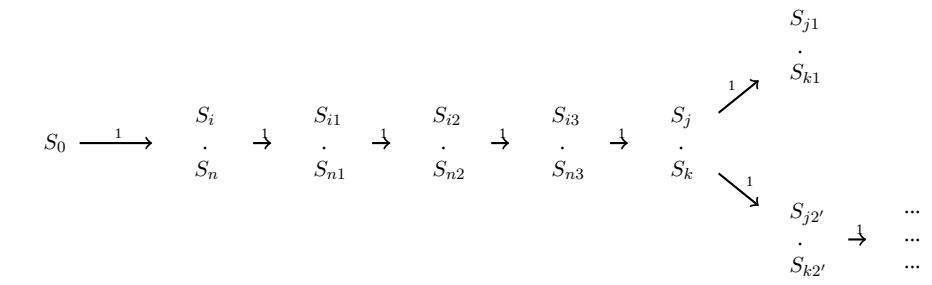
3.1 Définition du modèle "Time-accurate"

Le modèle de l'architecture doit être temporisé. Ceci signifie qu'il doit prendre en compte le temps (nombre de cycle d'horloge) nécessaire pour exécuter une opération. La façon la plus simple d'implémenter ce modèle est d'utiliser les tics de l'horloge comme cycle de base. Autrement dit le modèle évolu à chaque front montant de l'horloge. Néanmoins, en présence de défaut de cache (cache miss) et délais d'attente au niveau du pipeline, ce raisonnement mènerait à engendrer un nombre important d'états intermédiaires qui reflèteraient ces moments d'attentes (voir figure 3.1). Ainsi, l'ensemble des états intermédiaires ne serait pas pertinent au regard des performances de l'analyse. Une approche plus efficace et moins coûteuse consiste à sauter ces états intermédiaires. Le modèle proposé permet de générer directement l'ensemble des états futurs du processeur (différents de l'état initial), ainsi que le nombre de cycle d'horloge requis pour leurs évolutions respectives. Nous avons appelé ce type : modèle temporisé "Time-accurate model" (pour insister sur la priorité donnée au temps de passage d'un état à l'autre)

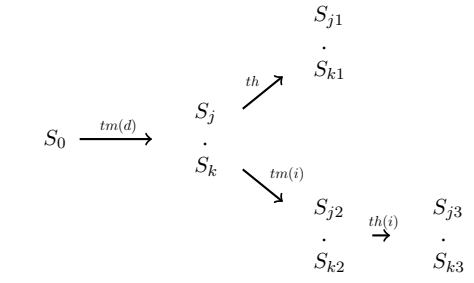
Définition 8 ***Modèle se basant sur les tics de l'horloge & Time-accurate model*** *Un modèle exécutable* basé sur les tics d'horloge *est une fonction qui associe un ensemble d'états s à un autre ensemble d'états s' à chaque front montant de l'horloge. Un modèle exécutable* time-accurate model *est une fonction qui associe un ensemble d'états s à un autre ensemble d'états s' ainsi que le temps $t \in \mathcal{T}$ nécessaire pour les atteindre.*

Le tableau suivant présente les temps de transfert entre la mémoire cache et un étage du pipeline

Temps des opérations effectuées par la mémoire cache
th : temps associé à un cache hit.
tm : temps associé à un cache miss.
trl : temps associé a un rechargement



(1. Tics de l'horloge)



(2. Time-accurate)

FIGURE 3.1: (1) Modèle d'architecture se basant sur les tics de l'horloge et (2) modèle Time accurate

Le modèle proposé est réalisé en C++. Néanmoins, d'autres extensions peuvent être faites en SystemC TLM-T, SystemC Verilog, ou en VHDL. Ce modèle tente de reprendre précisément les comportements du processeur que nous souhaitons analyser. Nous ne nous intéressons qu'aux comportements qui ont des impacts sur les temps d'exécution. Ainsi, ce modèle est constitué principalement de deux grandes parties (deux composants) :
1) La première représente la mémoire cache. Celle-ci sera donc instanciée deux fois, une fois pour simuler le cache de données et une autre pour représenter le cache d'instructions.
2) La seconde partie représente le pipeline. Ce composant contient l'ensemble des sous-composants suivants : F, D, LSU, IU, FPU, SRU, BPU, CU. Pour calculer les temps d'exécution et en déterminer le pire, nous devons développer un modèle d'architecture cible contenant certaines informations telles que :

- le temps nécessaire à chaque micro opération.
- les temps que prennent les caches miss et les caches hit.
- les différentes unités par lesquelles transite le flot d'instructions durant l'exécution.
- les interactions entre unités de traitement pouvant survenir durant l'exécution.

3.2 ASM : abstract state machine

La dynamique de notre modèle d'architecture est inspirée des travaux de Mueller qui portent sur la simulation de la sémantique de SystemC. Ainsi, dans [9, 55], un ASM est défini comme un pseudo-code comportant des données abstraites.

Dans la logique du premier ordre, une structure est un ensemble non-vide avec des fonctions et des relations. En terme d'ASM, ces ensembles non-vides sont appelés des univers. Les éléments d'un univers représentent des objets de la réalité. Par exemple, l'univers $ENTREES\_CONCRETES$ contient les éléments $nombre$, $chiffre$ et $caractere$. Les fonctions et les relations de la structure sont regroupées dans un ensemble fini appelé signature. Ces fonctions représentent les relations entre les objets de la réalité. Par exemple, une signature β contient le nom de la fonction
$cablage : PORT \rightarrow ENTREES\_CONCRETES$

De plus, le formalisme ASM ajoute des symboles true, false et undef, l'opération d'égalité = et les opérations standards sur les booléens.

Un état (state ou static algebra) s de signature β est un univers X combiné à une interprétation des noms de fonctions de β sur X. Par exemple, X est un super-univers composé des univers $PORT$ et $ENTREES\_CONCRETES$. Prenons la signature β déjà définie. Supposons que $PORT$ contient un élément t dont la valeur est $INPUT1$. L'état s peut être représenté par X avec l'interprétation : $cablage(t) = nombre$.

Pour simuler un comportement dynamique, le formalisme ASM introduit un mécanisme qui permet de mettre à jour les valeurs des fonctions pour certains arguments. Ce mécanisme d'update (:=) réalise les transitions entre états (exécution de Fonctions de Mises à Jour "FMJ" : $Univers \rightarrow Univers$). Dans ce principe des "evolving algebras", une transition est alors une règle dont l'exécution, pour transformer un état, dépend d'une condition. Ainsi, un ASM évolu en satisfaisant un ensemble de règles Ri. Chaque règle est exprimée sous la forme suivante :

$$if\ Condition\ then < mises\_a\_jour1 > else < mises\_a\_jour2 > endif$$

Le tir d'une transition se fait après la satisfaction des conditions qui sont associées à son état de départ. Par exemple la règle
$if\ cablage(t) = nombre\ then\ cablage(t) := chiffre$
décrit l'activité du système modélisé. Pour notre état s, la condition est satisfaite et le corps de la règle est donc exécuté. Après cette exécution, la signature β peut être interprétée sur X comme $cablage(t) = chiffre$ ce qui représente un nouvel état s'.

Un modèle ASM s'exécute en séquentiel car le modèle traite seulement une règle à la fois. Remarquons qu'il est possible que des conditions de plusieurs règles soient vérifiées à un moment. Dans ce cas, l'ordre dans lequel ces règles s'exécutent n'est pas déterministe. Pour simuler explicitement un comportement parallèle, le formalisme ASM possède une notion d'agent ou de module. Chaque agent ou module représente un modèle ASM (séquentiel) qui s'exécute en parallèle avec les autres agents. Ainsi, il existe un modèle ASM multi-agent pour simuler les modèles parallèles.

3.3 Modélisation du comportement du processeur

Dans notre modélisation, le processeur est divisé en composants (P, IC, DC, countime). Ces composants sont des objets réels, il prennent donc leurs valeurs respectives dans des univers. Ainsi, un composant P (pipeline) prendra ses valeurs dans l'univers $\mathbb{P}$ qui contient l'ensemble des états accessibles par le

pipeline. Exemple : $P = F\_i$ avec $F\_i \in \mathbb{P}$ indique que le pipeline récupère l'instruction i. L'évolution de chaque composant est définie par un ensemble de règles. Ces règles comportent des fonctions de mises à jour qui font évoluer l'état d'un composant. Ainsi, toutes les fonctions de mises à jour appliquées au composant P sont définies de l'univers $\mathbb{P}$ vers lui même ($P\_FMJ : \mathbb{P} \rightarrow \mathbb{P}$).

3.3.1 ASM : Etat

Intéressons nous à présent aux états du modèle Time-accurate. Basiquement un processeur peut être défini comme un ensemble de composants. Chacun d'entre eux exécute un nombre de tâches durant un cycle d'horloge. L'état courant d'un processeur est donc le produit de tous les états de ses composants ainsi que des interractions susceptibles d'apparaitre entre eux.

Définition 9 ***Etat d'un composant*** *un état $\mathcal{SC}[u]$ est un emsemble minimal de propriétés qui permettent de définir quel sera la prochaine opération à effectuer par ce composant.*
Etat du processeur *l'état du processeur s est le produit de tous les états de ses composants s : $s = (\otimes_{\mathbf{u}\ \textit{composant de}\ \mathbf{s}} \mathcal{SC}[\mathbf{u}])$.*

Donc, un état du modèle du processeur reflète les états des différents composants (P, IC, DC, countime) qu'il contient. Comme le montre la figure 3.2, un état du modèle de l'architecture inclut les paramètres suivants :

- (1) Pr : l'état du processeur (pipeline et caches instructions/données),
- (2) $countime$: le compteur interne.

De plus, durant l'exécution, un nouvel ensemble EC apparait, représentant l'univers des valeurs des entrées concrètes.

Définition 10 *Un état S est un quadruplet ($P, IC, DC, countime$) où :*

- *P : l'état du pipeline.*
- *IC : l'état du cache d'intructions.*
- *DC : l'état du cache données.*
- *countime : compteur de temps interne.*

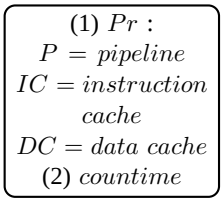


FIGURE 3.2: Etat

3.3.2 ASM : Transition

Le comportement de chaque composant est représenté par un ensemble de règles. Cependant, il existe une forte interaction entre les composants d'un même processeur. Ainsi, lors de l'établissement des règles associées à chaque composant, nous constatons qu'à chaque point d'exécution différentes règles, appliquées à des composants différents, partagent plusieurs conditions. Autrement dit, la même condition permet la mise à jour de plusieurs composants. Ainsi, dans notre modèle un ensemble de

composants est mis à jour à chaque fois qu'une condition est vérifiée. Ces fonctions de mise à jour sont séparées par des virgules, et sont excutées simultannément.

Définition 11 *Une transition $TR : S \rightarrow S$ est une fonction de transfert qui teste les états des composants internes du processeur (P, IC, DC) et exécute, en fonction du résultat des tests, le bloc de fonctions de mise à jour <MAJ> appropié.*

Un bloc de fonctions de mise à jour est donc activé à chaque tir d'une transition. De plus, ces transitions sont labélisées par le temps maximal que prennent ces mises à jour. Exemple, concernant le composant P, le passage de son état de $P = empty$ (pipeline vide) à l'état $P = F\_i$ prend un temps $t1$. Ce temps sera le label de la transition qui relie ces deux états.

3.3.3 ASM : Temporalité

La modélisation que nous proposons a pour but de nous permettre un calcul précis des temps d'exécution. Pour cela, à chaque point d'exécution, tous les temps d'exécution possibles doivent être représentés. A cet effet, chaque état du modèle du processeur a été enrichi par un composant $countime$. Ce composant $countime$ est mis à jour à chaque tir d'une transition t à laquelle est associée un temps $t1$. Cette mise à jour se fait en incrémentant sa valeur du label de la transition ($countime := countime + t1$).

3.3.4 Illustration

Pour expliquer l'évolution du modèle, prenons l'exemple simple d'une transaction mémoire. Admettons qu'à un point d'exécution le processeur requiert une donnée d. Ce processeur exécute donc une instruction I d'écriture ou de lecture mémoire. Ainsi, l'état courant du processeur S_i est défini par l'état de l'ensemble de ces composants comme suit : pipeline $P = LSU_I$, mémoire cache d'instruction IC, mémoire cache de donnée DC et compteur interne $countime = t$. Dans le cas où $d \in DC$, la donnée est transférée en un temps th au processeur. Dans le cas contraire, la donnée est récupérée de la mémoire principale, placée dans le cache et ainsi n'atteint le processeur qu'au bout d'un temps tm. La règle qui définit l'évolution du modèle dans ce cas est formulée de la façon suivante :

```
if d=cache_hit
then
    P:=CU_I,
    IC:=IC,
    DC:=DC,
    countime:=t+th
else
    P:=CU_I,
    IC:=IC,
    DC:=DC /\ d;d+7,
    countime:=t+tm
endif
```

$P := CU\_I$ signifie que l'instruction I a achevé son exécution et qu'elle peut donc être retirée du buffer d'exécution. Un ensemble de fonctions de mises à jour est activé en fonction de l'évaluation de la condition $d = cache\_hit$. Chaque fonction est associée à un composant particulier. Ainsi, l'activation simultannée de ces fonctions tranforme l'état S_i en un état S_{i+1}. Notons que la valeur qui incrémente le composant $countime$ représente le label de la transition qui relie l'état courant S_i à son succeseeur S_{i+1}.

La figure 3.3 montre le modèle Time-accurate de la mémoire cache de données. Ce modèle est déduit de la règle précédente et représente les évolutions possibles du cache (nous n'avons volontairement pas représenté le composant IC car ce composant n'a aucune influence durant ce pas d'exécution). La

modélisation que nous proposons permet de capter systématiquement l'impact qu'a le comportement de la mémoire cache sur les autres composants du processeur (interactions intra-processeur). Ainsi, lorsque la mémoire cache répond à la requête, le pipeline passe de l'état $P = LSU_I$ à l'état $P = CU_I$ alors que le composant $countime$ est incrémenté dans le cas (a) de th cycles et dans le cas (b) de tm cycles.

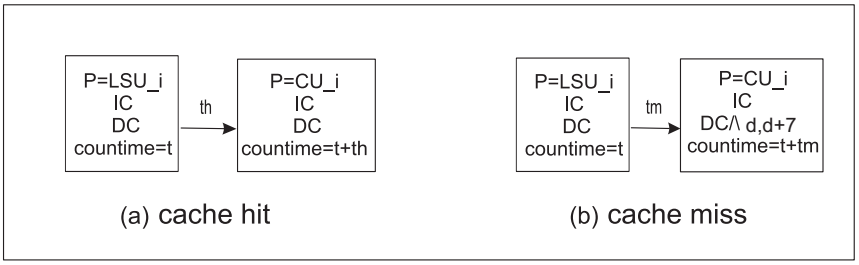


FIGURE 3.3: Evolution du modèle Time-accurate de la mémoire cache de données

3.3.5 Combinaison des modèles de composants

Chaque évolution d'un composant du processeur est modélisée par une règle. Ainsi, le modèle du processeur évolue en combinant les règles des composants qu'il contient. Nous proposons une définition qui représente l'évolution d'un processeur contenant deux composants. Cette définition est extensible à un ensemble de composants.

Définition 12 *Si R_a : $if\ Ca\ then < MAJa1 > else < MAJa2 > endif$ est la règle qui définit le comportement d'un composant a, et*
si R_b : $if\ Cb\ then < MAJb1 > else < MAJb2 > endif$ est la règle qui définit le comportement d'un composant b.
Alors, un modèle de processeur comportant ces 2 composants (a et b) évolue en combinant l'ensemble de leurs règles R_i avec $i \in \{a, b\}$. de la façon suivante :

```
if Ca
then
    if Cb
    then
        <MAJa1> U <MAJb1>
    else
        <MAJa1> U <MAJb2>
    endif
else
    if Cb
    then
        <MAJa2> U <MAJb1>
    else
        <MAJa2> U <MAJb2>
    endif
endif
```

L'application de l'opérateur d'union $\cup$ est guidée par le composant $countime$. Ainsi, L'évaluation de <MAJa1> $\cup$ <MAJb1> est faite de la façon suivante : la première étape consiste à identifier le bloc contenant le composant $countime$ dont l'incrément est le plus petit : $countime_{a1b1} =$

$min\{countime_{a1}, countime_{b1}\}$. La seconde étape consiste à activer uniquement le bloc identifié durant l'étape précédente et à sauvegarder la différence entre les compteurs associés, d'une part, au bloc qui n'a pas été exécuté, et d'autre part, au bloc qui a été exécuté : Si $countime_{b1} > countime_{a1}$, alors le bloc <MAJa1> est activé et la valeur $countime_{b1} - countime_{a1}$ est sauvegardée, sinon, $countime_{a1} > countime_{b1}$, le bloc <MAJb1> est activer et la valeur $countime_{a1} - countime_{b1}$ est sauvegardée. Au fur et à mesure que l'exécution avance, cette différence entre les valeurs des deux compteurs est décrémentée. Lorsqu'elle atteint, la valeur zéro, le bloc de fonctions de mises à jour qui n'avait pas été exécuté est activé.

A présent, si $countime_{a1b1} = countime_{a1} = countime_{b1}$, un nouveau bloc, représentant la conjonction des blocs <MAJa1> et <MAJb1> est activé. En d'autres termes, si <MAJa1>= $\{maj_{Pa1}, maj_{ICa1}, maj_{DCa1}\}$ et <MAJb1>= $\{maj_{Pb1}, maj_{ICb1}, maj_{DCb1}\}$ alors le bloc activé est :
<MAJa1> $\wedge$ <MAJb1>=$\{maj_{Pa1} \wedge maj_{Pb1}, maj_{ICa1} \wedge maj_{ICb1}, maj_{DCa1} \wedge maj_{DCb1}\}$
Pour illustrer nos propos, choisissons de reprendre le modèle représentant l'évolution de la mémoire cache de données et de le combiner avec le modèle représentant l'évolution de la mémoire cache d'instructions.

```
Ra:
    if d=cache_hit
    then <MAJa1>
        P:=CU_i,
        IC:=IC,
        DC:=DC,
        countime:=t+th
    else <MAJa2>
        P:=CU_i,
        IC:=IC,
        DC:=DC /\ d;d+7,
        countime:=t+tm
    endif
Rb:
    if i''=cache_hit
    then <MAJb1>
        P:=F_i'',
        IC:=IC,
        DC:=DC,
        countime:=t+th
    else <MAJb2>
        P:=F_i'',
        IC:=IC /\ i'';i''+7,
        DC:=DC,
        countime:=t+tm
    endif
```

Admettons que la condition Ca : $d = cache\_hit$ soit vraie et que la condition Cb : i"$= cache\_hit$ soit fausse. La combinaison de ces deux règles résulte donc en l'évaluation de l'expression <MAJa1> $\cup$ <MAJb2>. Ainsi, vu que $t+th < t+tm$, <MAJa1> est activé et le bloc <MAJb2> est ingnoré durant ce pas d'exécution. La valeur $tm - th$ est sauvegardée, puis décrémenté à chaque pas d'exécution de façon à activer le bloc <MAJb2> lorsque cette valeur atteindra 0.

A présent, si les deux conditions Ca : $d = cache\_hit$ et Cb : i"$= cache\_hit$ sont vraies , alors le bloc activé est <MAJa1> $\cup$ <MAJb1>. Le composant $countime$ de <MAJa1> est égale à celui de <MAJb1>. Ainsi, un nouveau bloc, comportant les fonctions de mises à jour suivantes, est activé :
<MAJa1> $\cup$ <MAJb1>=

```
P:=CU_i /\ F_i'',
IC:=IC,
DC:=DC
```

Notons que la conjonction de deux composants identiques résulte en l'un de ces deux composants. Exemple, $IC \wedge IC = IC$

L'intérêt de cette modélisation réside donc dans sa capacité à représenter l'évolution des composants de façon précise. Ainsi, les temps de transitions d'un état à l'autre sont des valeurs exactes (aucune approximation n'est introduite).

3.4 Etude de cas : PowerPC 603 et 603(e)

Les processeurs selectionnés pour tester l'approche sont très communs aux systèmes embarqués. Le PowerPC 603 et 603(e) [52, 53] ont des fonctionnements très similaires. C'est la raison pour laquelle nous ne nous intéresserons qu'à l'une de ces deux architectures [1].

Le PowerPC 603 est un processeur qui contient deux mémoires caches, un cache d'instructions et un autre de données. Ces mémoires caches implémentent des stratégies de remplacement LRU (last recent used). Ces stratégies sont utilisées afin de déterminer la ligne de cache concernée à chaque fois que nous sommes en présence d'une opération de lecture/écriture dans le cache.

Ce processeur contient un pipeline intégrant cinq unités d'exécution : l'unité d'entiers (IU : integer unit), l'unité des flottants (FPU : floating point unit), l'unité des instructions de branchement (BPU : branch processing unit), l'unité d'écriture et de lecture mémoire (LSU : load/store unit) et l'unité de registre système (SRU : system register unit). Additionnellement à ces unités d'exécution, le pipeline contient aussi : un Fetcher (F), un dispatcher (D) et une unité chargée d'achever l'exécution des instructions appelée completion unit (CU)

Le tableau suivant reprend les différentes notations utilisées pour représenter les unités d'exécution du PowerPC 603 :

Unités du processeur
F : Fetcher
D : Dispatcher
BPU : Unité de branchement
LSU : Unité de lecture et d'écriture mémoire de ligne de cache.
IU : Unité entière
FPU : Unité flottante
SRU : Unité des registres système
CU : Unité d'achèvement
RS : registre temporaire
IQ : pile d'instructions
IC : cache d'instructions
DC : cache de données

1. Le powerPC 603(e) est partiellement présenté dans la section 4.6.1

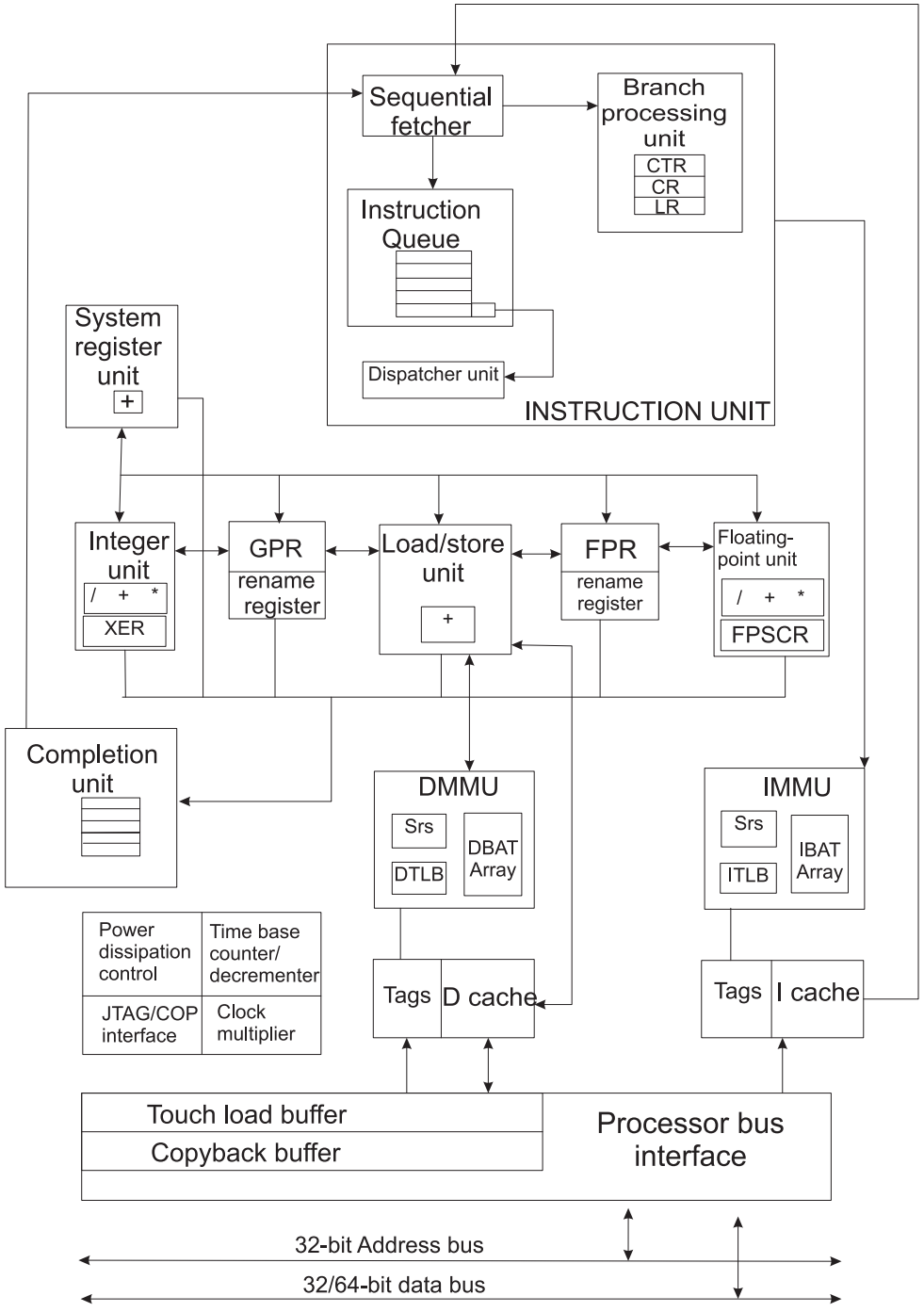


FIGURE 3.4: Le PowerPC 603

3.4.1 Description des unités composant le processeur

Pour construire ce modèle exécutable, la description détaillée de l'architecture est nécessaire.

Cette description doit non seulement couvrir les comportements des unités qui composent le processeur mais aussi contenir les temps de toutes les micro-opérations réalisées par ce processeur. Nonobstant

l'abstraction de certains comportements du processeur, ce modèle doit néanmoins mettre en valeur toutes les interactions inter et intra unités ayant un impact aussi petit soit-il sur les temps d'exécution.

Dans le but de mettre au point un modèle précis du processeur, nous donnons, ci dessous, le fonctionnement précis du PowerPC 603.

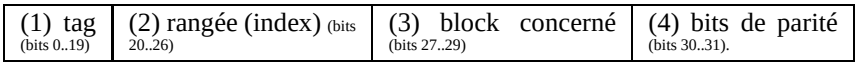
3.4.1.1 Mémoire cache : (MEM)

La mémoire cache est une des principales évolutions de ces dernières années. Elle a contribué à la sophistication des processeurs. Cette $petite$ mémoire permet un gain de performance considérable. En effet, grâce à ce dispositif, le processeur peut accéder aux données beaucoup plus rapidement. Les délais d'accès à la mémoire cache étant minimes, si une donnée est placée dans le cache et que le processeur la requiert par la suite, il pourra la récupérer immédiatemment (en un cycle d'horloge).

Fonctionnement de la mémoire cache

Le fonctionnement de la mémoire cache a un lien direct avec la façon dont les instructions/données sont placées dans le cache. Nous allons donc nous intéresser, à présent, à ce mécanisme susceptible d'octroyer un gain de temps considérable à chaque fois que le processeur requiert une instruction/donnée.

Le PowerPC 603 contient deux mémoires caches, un cache instructions (IC) et autre de données (DC), de 8 Koctet chacune. Ces deux mémoires ont globalement la même structure. Elles sont toutes deux composées de 128 rangées, chaque rangée contient deux blocs. Un bloc est composé d'une suite de bits appelée tag (étiquette) et de 32 octets d'instructions/données, ce qui signifie que chaque bloc peut contenir 8 instructions/données de 32 bits. Pour retrouver une instruction/donnée dans le cache, il suffit de connaitre son adresse. En effet, l'adresse d'une instruction/donnée est divisée, comme le montre la figure, en (1) 20 bits de tag, (2) 7 bits d'$index$ permettant de calculer la rangée du cache dans laquelle doit être stockée cette instruction/donnée, (3) trois bits qui permettent d'identifier, parmi les 8 instructions/données contenues dans chaque bloc, l'instruction/donnée qui a déclenchée le transfert du bloc mémoire et (4) 2 bits de parité.



La mémoire cache du PowerPC 603 est 2 voies associatives (elle peut stocker deux lignes de cache ayant le même index). Cette mémoire cache implémente une stratégie de remplacement de type LRU (Least Recently Used) détaillée ci-dessous.

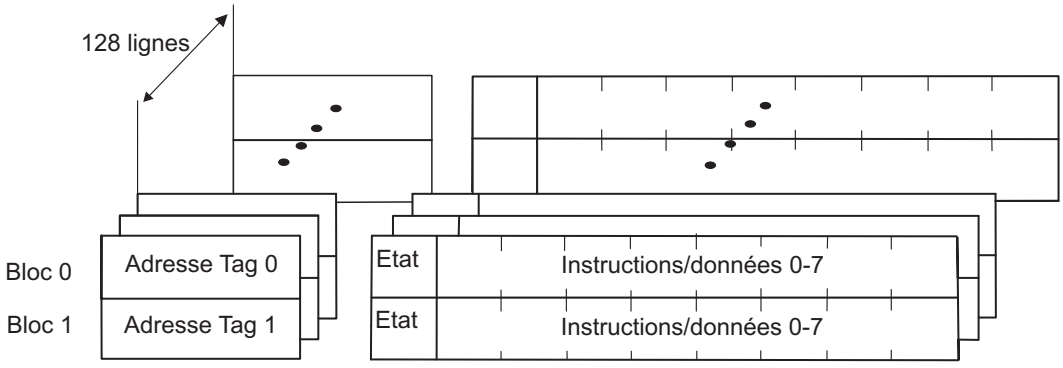


FIGURE 3.5: Organisation de la mémoire cache du PowerPC 603 (instructions et données)

Stratégie de remplacement LRU
Cette stratégie consiste à remplacer la ligne la plus anciennement référencée. Elle est parfaitement adaptée au PowerPC 603 du fait que l'associativité de ses mémoires caches est petite (2 voies associatives voir figure 3.5). En effet, la méthode LRU donne de bons résultats mais devient assez coûteuse lorsque l'associativité augmente. Ceci est dû à l'augmentation du nombre de tests indispensables avant de sélectionner la bonne ligne de cache.

3.4.1.2 Pipeline : (P)

Le pipeline est un mécanisme qui divise l'exécution des instructions en plusieurs étapes. Chaque étape est appelée $étage\ de\ pipeline$. Cette technique ne permet pas d'exécuter une instruction plus rapidemment. L'exécution étant divisée en plusieurs étapes, si une seule instruction est traitée par un pipeline, aucun changement ne sera observé quant au temps d'exécution. La force de ce mécanisme est d'octroyer au processeur la possibilité de commencer l'exécution des instructions suivantes avant d'achever l'exécution de l'instruction courante. Ainsi, la vitesse d'exécution d'un groupe d'instructions augmente.

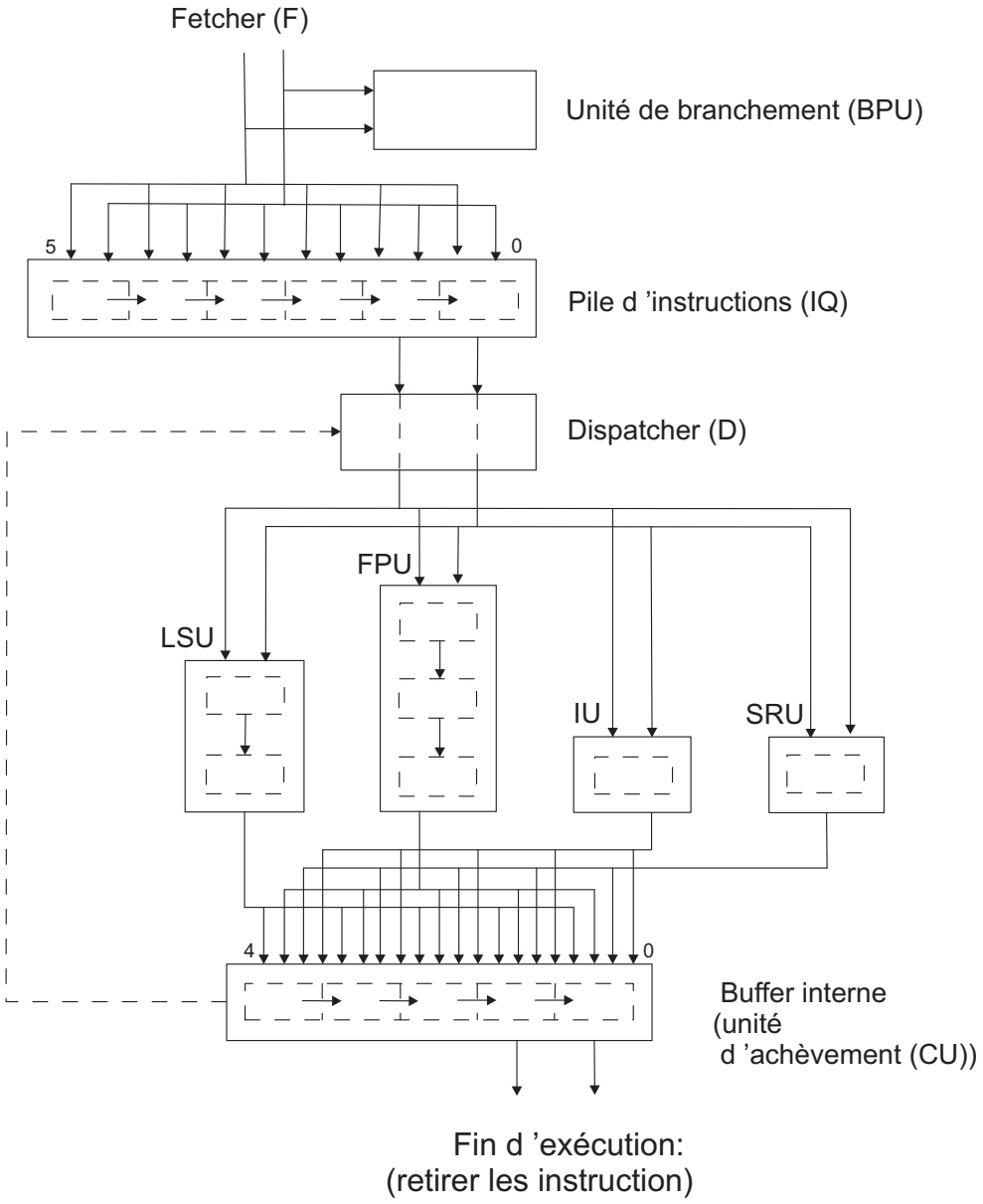


FIGURE 3.6: Evolution du flot d'instructions à l'intérieur du pipeline

Fetcher : (F)

Le fetcher peut récupérer à chaque top d'horloge jusqu'à deux instructions de la mémoire et localiser l'adresse de l'instruction suivante. Ces instructions sont stockées dans une pile interne (IQ) , appelée pile d'instructions (Instruction queue).

Durant cette étape l'unité de branchement détecte les instructions de saut afin de les traiter au plus vite. Cela permet de réduire les délais d'attente. Le but est d'exécuter les instructions de branchement conditionnel en zéro cycle (voir section 3.4.1.4).

Dispatcher : (D)

Le dispatcher récupère les instructions de la pile d'instructions (IQ), les décode afin de déterminer :

- (1) les unités de traitement qui doivent recevoir ces instructions.
- (2) si ces instructions sont éligibles à être dispatchées au cours du cycle courant.

Unités d'exécution : (EX)

Le pipeline du PowerPC 603 contient 5 unités d'exécution : une unité entière (IU), une flottante (FPU), une de lecture et d'écriture (LSU), une de branchement (BPU) et une de registre système (SRU). Chaque unité d'exécution commence le traitement d'une instruction dès sa réception. Ce traitement peut se faire durant un ou plusieurs cycles, puis l'unité écrit le résultat de l'opération dans un de ses registres (registres intermédiaires). Ensuite, elle notifie à l'unité d'achèvement (CU) que l'exécution de l'instruction est terminée.

Les instructions flottantes sont exécutées par la FPU en parallèle. Cette unité peut exécuter jusqu'à 3 instructions simultanément. La LSU peut exécuter jusqu'à 2 instructions en même temps. Ces exécutions se font grâce à la structure interne micro-pipelinée de ces unités. Exemple concernant la LSU, son micro-pipeline contient 2 étages : le premier étage convertit l'adresse virtuelle en une adresse physique alors que le second se charge d'accéder à la mémoire pour en récupérer ou y stocker une donnée. Notons que les micro-pipelines, au même titre que le pipeline, permettent d'augmenter localement (au niveau des unités d'exécution (EX)) la vitesse d'exécution d'un ensemble d'instructions.

Par ailleurs, ces unités peuvent fonctionner indépendamment les unes des autres. Ce qui confère au pipeline la possibilité d'exécuter des instructions "partiellement dans le désordre". En effet, dans le cas où aucune dépendance de données n'est détectée, une unité qui reçoit une instruction, l'exécute sans se préoccuper de l'avancement des autres instructions. Ainsi, une unité peut avoir fini de traiter une instruction du programme avant que les prédécésseurs de cette instruction ne finissent de s'exécuter.

Exemple (voir figure 3.7) : supposons que durant l'exécution nous ayons à traiter les instructions I_1 et I_2 (selon l'ordre imposé par le programme I_1 précéde I_2). A présent, si I_1 est une addition de flottant alors que I_2 est une addition d'entier, I_1 prendra 3 fois plus de temps que I_2 à s'exécuter [2]. Ainsi, I_1 sera dispatchée vers son unité d'exécution un cycle avant I_2, cependant le traitement de I_2 s'achèvera avant celui de I_1.

2. Addition de flottants = 3 cycles d'horloge. Addition d'entier = 1 cycle d'horloge.

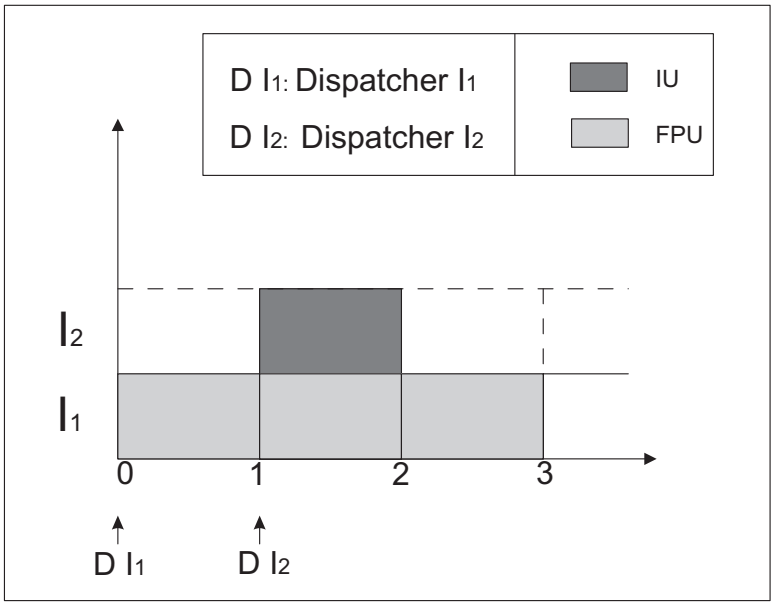


FIGURE 3.7: Exemple d'exécution partiellement dans le désordre

Notons que le PowerPC 603 n'exécute pas les instructions "totalement dans le désordre". En effet, le mode "totalement dans le désordre" impose que le dispatcher (D) réordonnance les instructions en fonction des disponibilités des unités de traitement. Autrement dit, les instructions présentes dans la pile d'instructions (IQ) sont envoyées vers leurs unités de traitement respectives dans un ordre différent de celui imposé par le programme. L'ordre d'exécution des instructions est, à présent, dicté par la disponibilité des unités d'exécution.

Or, concernant le PowerPC 603, l'exécution dans le désordre n'est susceptible d'apparaitre qu'au niveau des unités d'exécution (EX) et non du dispatcher (D) (pas de ré-ordonnancement d'instructions). Ainsi, nous dirons que ce processeur est capable d'exécuter les instructions dans un désordre partiel.

Unité d'achèvement : (CU)
L'unité d'achèvement est alertée en fin d'exécution. En réalité, cette unité commence à traquer les instructions aussitôt qu'elles atteignent le dispatcher afin de maintenir, dans un buffer interne, une trace de leurs exécutions. Ainsi, cette unité peut s'assurer qu'aucune exception (problème durant l'exécution) ne s'est produite. Cette unité a, par ailleurs, d'autres rôles :

- (1) de transférer les résultats obtenues par les unités d'exécution, des registres temporaires vers des registres appelés registres généraux (GPR : general purpose register) ou vers des registres flottants (FPR : floatingpoint purpose registre). Ce transfert se fait juste avant que l'unité d'achèvement ne retire l'instruction concernée de son buffer interne. Ceci certifie que l'exécution s'est terminée sans qu'aucune exception ne soit laissée sans traitement.
- (2) de maintenir une cohérence d'exécution. Ce qui implique de remettre les instructions dans le bon ordre (l'ordre imposé par le programme) avant d'achever leurs exécutions.

3.4.1.3 Exécution d'une séquence d'instructions

Dans cette partie nous allons montrer comment une séquence d'instructions est exécutée par le PowerPC 603. Durant cette exécution, nous supposons que toutes les instructions du programme sont dans

la mémoire cache (IC). Comme nous pouvons le voir sur la figure 3.8 le traitement du flot d'instructions s'effectue de la manière suivante :

- cycle 1 : les instructions 1 et 2 sont récupérées de la mémoire cache (F).
- cycle 2 : deux opérations sont effectuées en parallèle : (1) les instructions 1 et 2 sont décodées (D), (2) les instructions 3 et 4 sont récupérées de la mémoire cache (F).
- cycle 3 : les instructions 1 et 2 sont traitées par leurs unités d'exécution respectives (EX) alors que les instructions 3 et 4 sont décodées (D).
- cycle 4 : l'instruction 1 a terminé son exécution. Le résultat obtenu est donc transféré par l'unité d'achévement vers un registre du processeur (WB). L'instruction 2 (instruction flottante) effectue son deuxième cycle d'exécution (EX) (deuxième étage du micro-pipeline de la FPU). Les instructions 3 et 4 sont traitées par leurs unités de traitement respectives (EX) (instruction 3 est traitée par l'IU, instruction 4 effectue son premier cycle d'exécution : première étage du micro-pipeline de la FPU)
- cycle 5 : l'intruction 1 est retirée du buffer d'exécution (DL). L'instruction 2 exécute son troisième cycle d'exécution (EX). L'instruction 3 a terminé son exécution, le résultat qu'elle a produit est transféré vers un registre du processeur (WB). L'instruction 4 effectue son deuxième cycle d'exécution (EX).
- cycle 6 : L'instruction 2 a terminé son exécution, le résultat qu'elle a produit est transféré vers un registre du processeur (WB). Du fait que l'instruction 2 n'a toujours pas été retirée du buffer d'exécution (DL), l'intruction 3 reste bloquée (WB). L'instruction 4 exécute son troisième cycle d'exécution (EX).
- cycle 7 : les instructions 2 et 3 sont retirées du buffer d'exécution. L'instruction 4 a terminé son traitement, ainsi le résultat qu'elle a produit est transféré vers un registre du processeur (WB).
- cycle 8 : l'instruction 4 est retirée du buffer d'exécution.

Ce qui marque la fin de l'exécution de la séquence de code.

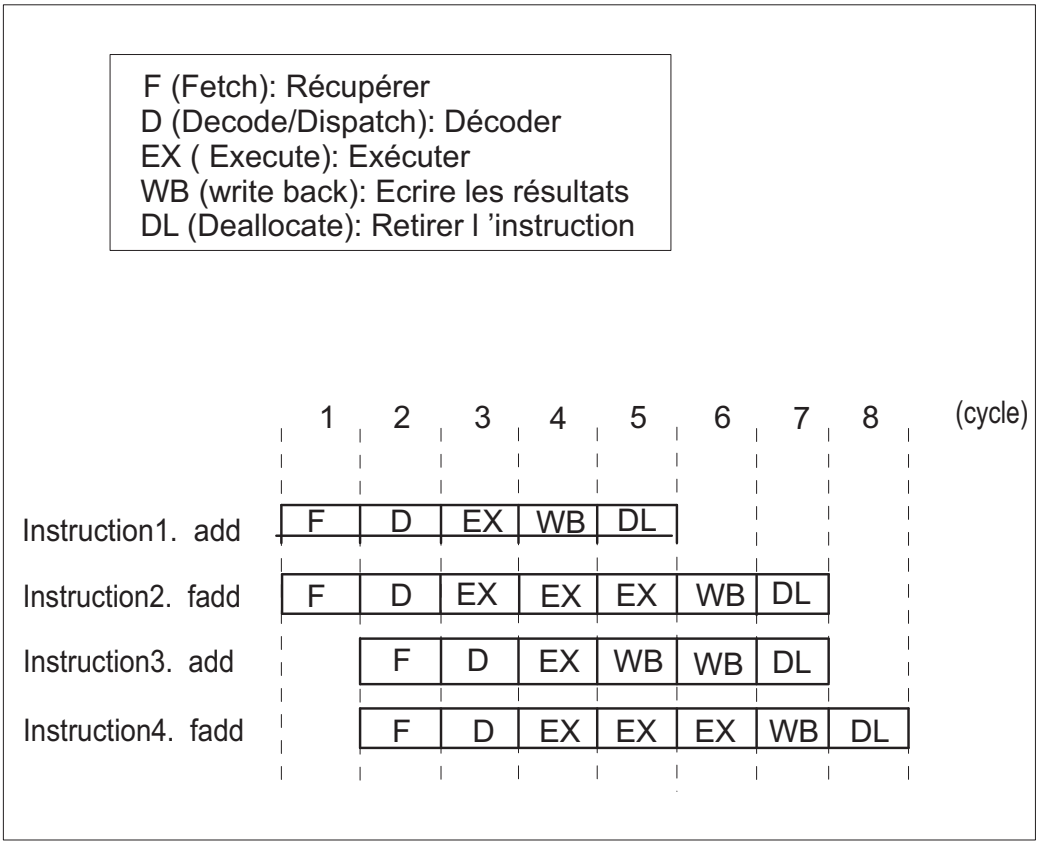


FIGURE 3.8: Exemple d'exécution d'une séquence d'instructions

3.4.1.4 Unité de branchement : (BPU)

Cette unité traite une partie très importante du comportement du programme. En effet, une instruction conditionnelle peut altérer le flot d'instructions traitées (l'ordre dans lequel les instructions du programme sont traitées). Cette altération a un impact direct sur les temps d'exécution et doit donc être traitée avec beaucoup de précaution.

Le PowerPC 603 contient une BPU qui extrait les instructions de branchement directement du flot d'instructions qui arrive au pipeline (directement du fetcher (F) sans passer par la pile d'instructions (IQ)) afin de les traiter le plus rapidement possible. Le but est de ne pas introduire de délais d'attente au sein du pipeline. La BPU tente de traiter les instructions de branchement conditionnel avant que leurs successeurs n'atteignent l'unité de traitement suivante (le dispatcher (D)). Ce type d'instruction est donc exécuté en parallèle des instructions qui le précédent. Ainsi, son temps d'exécution peut être ramené à zéro cycle d'horloge.

Prédiction de branchement

Le principe de ce mécanisme est qu'à chaque fois qu'une instruction de branchement comporte une condition qui ne peut être traitée au moment où l'unité reçoit cette instruction, la BPU va alors faire une prédiction. Cette approche possède un immense avantage : si la prédiction est juste, aucun délais d'attente ne sera introduit. Le revers de la médaille est que si la prédiction est fausse, l'exécution s'interrompt. Ensuite, toutes les instructions qui succèdent l'instruction de branchement conditionnel sont retirées du

pipeline et l'exécution reprend à partir de l'adresse de branchement calculée.

Exécution spéculative
L'exécution spéculative est dans la continuité de la prédiction de branchement. En effet lorsqu'un branchement est prédit, le processeur va tout naturellement commencer à exécuter les instructions suivant cette branche. Cependant, la condition n'étant pas encore traitée, les résultats, des exécutions des instructions qui succèdent à l'instruction de branchement, ne peuvent pas être exploités par le processeur. Ceci implique que ces instructions soient traitées par leurs unités respectives mais que leurs exécutions ne soient pas achevées (ces instructions ne sont pas retirées du buffer d'exécution). Ces instructions resteront donc au niveau de l'unité d'achèvement (CU) jusqu'à ce que la condition de branchement soit traitée.

L'intérêt de l'exécution spéculative est d'éviter les périodes d'inactivité du pipeline. Sans ce mode d'exécution, le fetcher (F) serait inactif dès le moment où l'instruction de branchement conditionnel serait récupérée du cache et cela jusqu'à ce que le résultat du traitement de la condition soit connu. Le point fort de cette technique est que si la condition est vérifiée, le traitement des instructions se poursuit, suivant le chemin prédit, sans interruption. Dans le cas contraire, comme mentionné ci-dessus, le pipeline est vidé (toutes les instructions qui suivent l'instruction de branchement sont retirées du pipeline), et la nouvelle adresse de branchement est utilisée pour déterminer le chemin adéquat.

3.4.1.5 Exécution d'instructions de branchement

Dans cette partie, nous présentons un exemple d'exécution d'instructions de branchement. L'exemple proposé dans la figure 3.9 est introduit afin d'expliquer les mécanismes présentés ci-dessus (prédiction de branchement et exécution spéculative).

L'exemple contient deux instructions de branchement (instructions 2 et 6). Lorsque le processeur traite cet exemple, il effectue donc deux prédictions : (1) la première est correcte et (2) la seconde est incorrecte.

Prédiction correcte
Au cycle 2, l'unité de branchement (BPU) reçoit la deuxième instruction du programme et effectue une prédiction de branchement. Les instructions 3 et 4 sont, donc, récupérées de la mémoire cache (IC). C'est durant ce cycle que l'exécution spéculative commence. En effet, le processeur n'a toujours pas exécuté l'instruction de branchement (instruction 2), cependant il poursuit l'exécution selon l'une des deux branches (exécution spéculative). Au cycle 4, l'instruction 2 est exécutée. Ainsi, au cycle suivant (cycle 5), il en résulte que la prédiction du processeur est correcte. L'exécution peut donc se poursuivre normalement. Le PowerPC 603 peut donc transférer le résultat obtenu suite à l'exécution de l'instruction 3 vers le registre approprié (GPR).

Prédiction incorrecte
Au cycle 4, l'unité de branchement (BPU) reçoit la sixième instruction du programme et effectue une nouvelle prédiction de branchement. Les instructions 7 et 8 sont, donc, récupérées de la mémoire cache (IC). L'exécution spéculative apparait donc une seconde fois. Cependant au cycle 7, après que l'instruction 6 ait été exécutée, la prédiction se révèle incorrecte. Le processeur doit donc : (1) stopper les exécutions des instructions 7 et 8, (2) effacer les résultats de ces exécutions de ses registres intermédiaires, et (3) récupérer le flot d'instructions correctes (instructions 9 et 10).

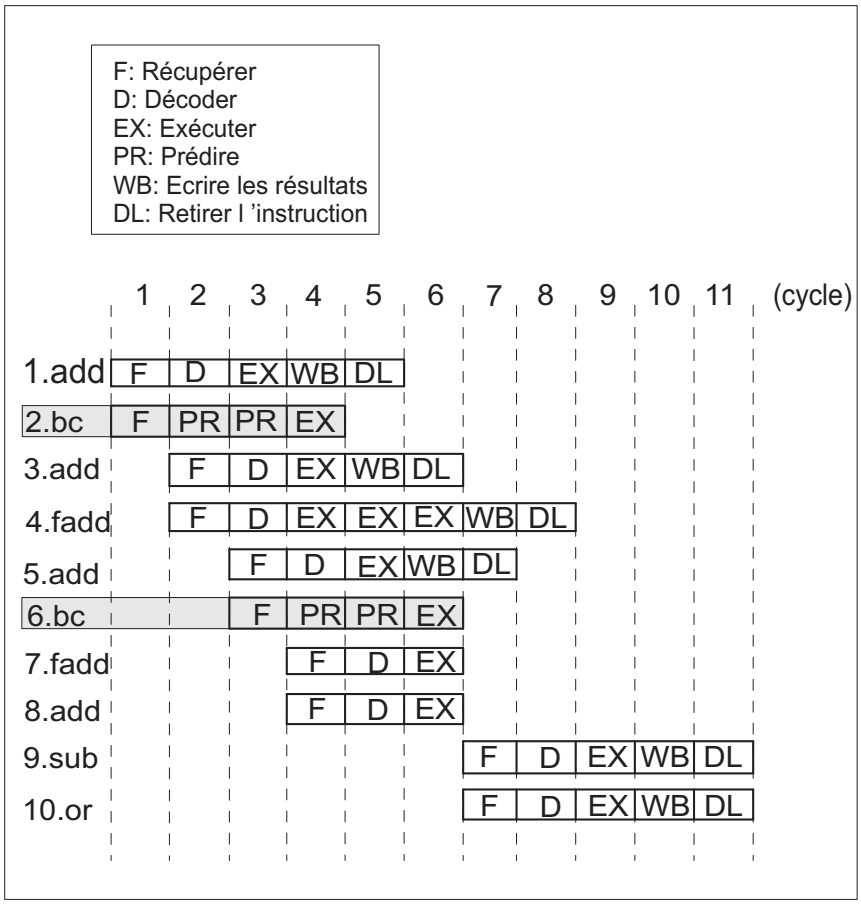


FIGURE 3.9: Exemple d'exécution d'instructions de branchement

3.4.2 Micro-opérations et temporalités

A présent, intéressons-nous au fonctionnement interne des unités d'exécution. Comme nous l'avons mentionné précédemment, le PowerPC 603 contient 5 unités d'exécution. Certaines ont un fonctionnement séquentiel simple, comme les unités entière (IU) et de registre système (SRU). Ceci, leurs confèrent la possibilité d'exécuter chaque instruction en un cycle d'horloge. D'autres ont une structure plus complexe. Prenons l'exemple de l'unité flottante (FPU) (voir figure 3.10) ou de celle de lecture et d'écriture (LSU). Ces unités ont des fonctionnements parallèles (micro-pipeline), ce qui leur permet de traiter respectivement 3 (A,B,C) et 2 instructions par cycle. Mais cela signifie également que pour l'exécution d'une instruction ces unités consommeront, au mieux, respectivement 3 et 2 cycles. Durant ces types d'exécution, l'instruction concernée est transférée d'un étage du micro-pipeline à l'autre.

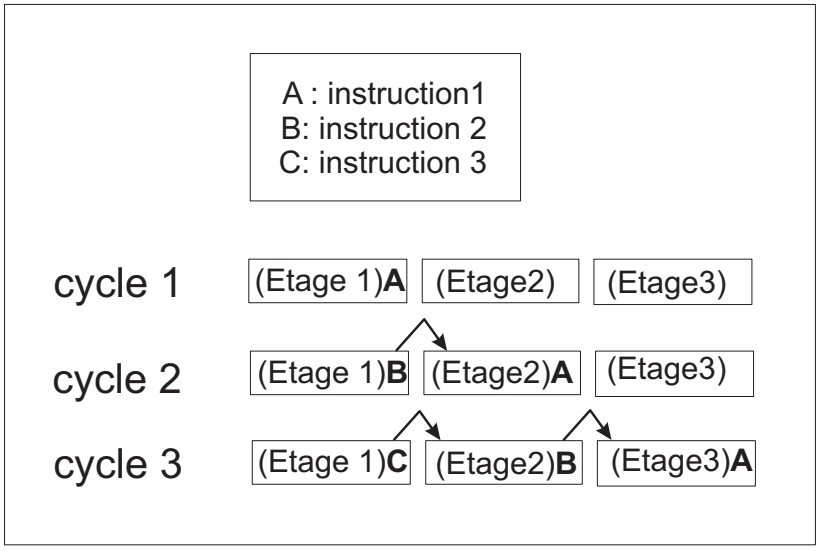


FIGURE 3.10: Micro-pipeline de l'unité flottante (FPU)

3.5 Modélisation du PowerPC

La modélisation du processeur cible passe par une phase où les comportements de ce processeur sont répertoriés et étudiés. Cette étude vise à décider des comportements et des paramètres ayant un impact direct ou indirect sur les temps d'exécution. L'intérêt de cette étape est donc de mettre en évidence la partie pertinente du processeur [3]. Présentons, à présent, pas à pas le modèle du processeur.

Par souci de clarté, les modèles de chaque unité de traitement sont présentés sous une forme algorithmique (organigramme). Le but étant d'expliquer le raisonnement suivi durant la phase de modélisation et de montrer que les modèles peuvent être développés indépendemment du formalisme d'abstraction choisi. Nous montrons également, pour chaque unité, comment le Time-Accurate modèle est construit.

3.5.1 Modèle de cache

la modélisation de la mémoire cache est composée de plusieurs classes. Les plus importantes sont les classes :

- $ControlCache$ qui contient différentes méthodes permettant de gérer cette mémoire cache, (gérer les transactions entre le cache et la mémoire principale, stratégie de remplacement du cache LRU, etc.).
- $CacheMemory$ qui représente au niveau du modèle le cache physique (tableau à deux dimensions).

La figure 3.11 présente une version simplifiée de la fonction permettant de rechercher un ensemble d'adresses dans la mémoire cache de données/instructions. Cette recherche est faite suite à l'envoi d'une requête par la LSU/F. Cette dernière (classe LoadStoreUnit/Fetcher) reçoit, en réponse à sa demande, une valeur entière représentant le temps requis pour récupérer la donnée/instruction.

3. Pertinente du point de vue calcul des temps d'exécution.

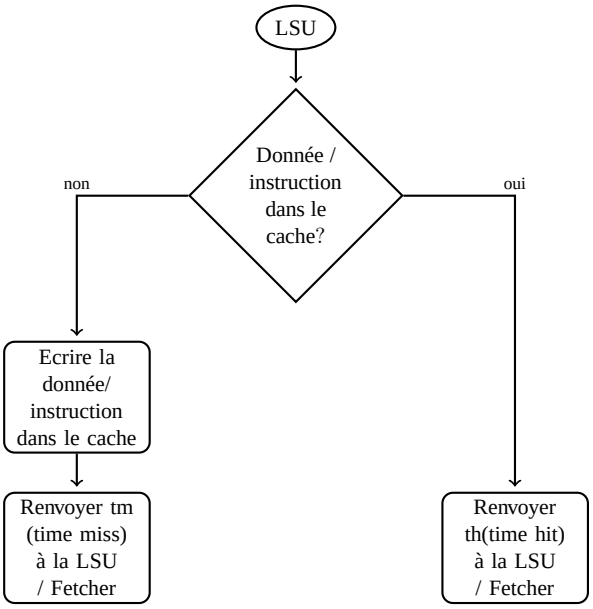


FIGURE 3.11: Fonction gérant les interactions avec le cache de données/instructions : ControlCache

La règle qui représente l'évolution de la mémoire cache instruction [4] se compose de deux blocs contenant des fonctions de mises à jour. En fonction du résultat obtenu, suite à l'évaluation de la condition, un seul bloc sera activé. Ainsi, dans le cas où la condition est vérifiée, le F récupèrera l'instruction i après th cycles et dans le cas contraire, cette même instruction sera placée dans l'IC et le F ne la récupèrera qu'après tm cycles.

```
if i=cache_hit
then
    P:=F_i,
    IC:=IC,
    DC:=DC,
    countime:=t+th
else
    P:=F_i,
    IC:=IC /\ i;i+7,
    DC:=DC,
    countime:=t+tm
endif
```

Comme le montrent les figures 3.12 et 3.13 , les modèles utilisés dans l'analyse doivent refléter respectivement le principe de fonctionnement des mémoires caches d'instructions et de données . Ces modèles sont directement déduits de l'ensemble des règles présentées précédemment. Ainsi, le modèle de la mémoire cache (instructions ou données) répond à une requête de deux façons différentes : soit cache hit, soit cache miss et renvoie le temps associé à chaque scénario.

4. le modèle de la mémoire cache de données est présenté dans la partie 3.3

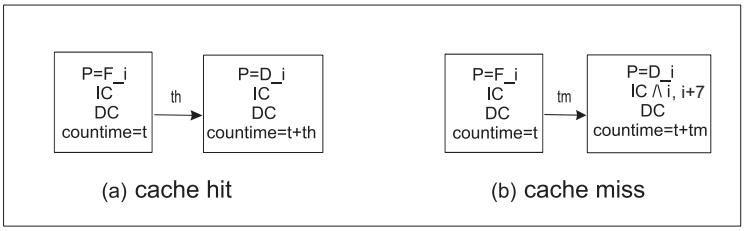


FIGURE 3.12: Evolution du modèle Time-accurate de la mémoire cache d'instructions

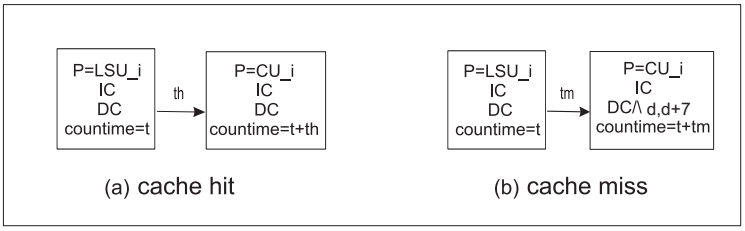


FIGURE 3.13: Evolution du modèle Time-accurate de la mémoire cache de données

3.5.2 Modèles de pipeline

Pour la partie concernant le pipeline, celle-ci reprend dans son intégralité la description du pipeline PowerPC 603. Elle contient donc toutes les unités et doit, par ailleurs, prendre en compte toutes les interactions intra-processeurs. Le flot d'instructions à l'intérieur du pipeline évolue d'un étage à l'autre selon le schéma proposé dans la figure 3.6 (voir section 3.4.1.2).

Ainsi, une fois que le F récupère des instructions de l'IC, ces instructions sont stockées dans IQ, puis, elles sont dispatchées, par le D, vers leurs unités de traitement respectives (EX) . Dès que le traitement d'une instruction se termine, la CU est alertée. Cette unité se charge alors de transférer le résultat de l'opération vers des registres du processeur. Toutes les instructions du programme seront traitées de cette façon à l'exception des instructions de branchement qui sont extraites, par la BPU, du flot d'instruction immédiatement après avoir été placées dans l'IQ.

3.5.2.1 Modèle du fetcher

La classe *Fetcher* récupère continuellement des instructions de l'IC et les place dans l'IQ.

La principale fonction du fetcher est représentée par la figure 3.14. Cette fonction a pour but de déterminer le temps nécessaire pour récupérer une instruction de la mémoire cache.

Les instructions stockées dans l'IQ ont pour point commun d'être ordonnées de façon séquentielle (maintiennent l'ordre fixé au niveau du programme). Il existe cependant d'autres types d'instructions qui interrompent cet ordre. Ces instructions sont des instructions de branchement. A cet effet, une composant BPU (voir la partie 3.5.3) est implémentée. Ce composant repère les instructions de branchement dès l'instant où elles atteignent l'IQ afin de les traiter séparément.

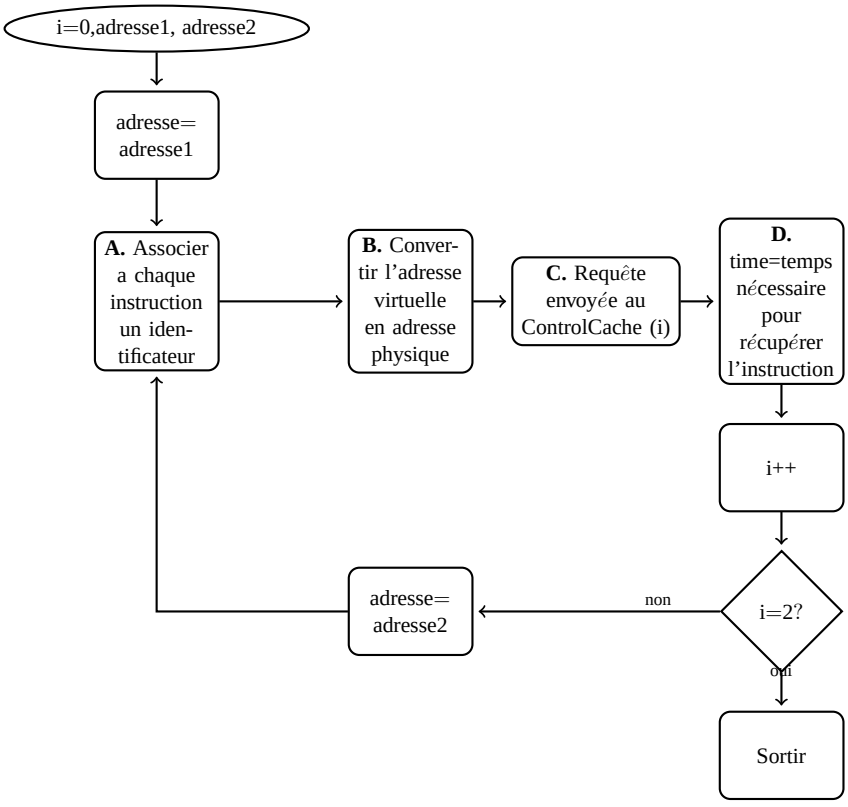


FIGURE 3.14: Récupérer une instruction : Fetcher(F)

Le PowerPC 603 peut récupérer simultannément jusqu'à deux instructions par cycle. La règle reflétant son évolution doit donc être étendue de façon à prendre en compte ce traitement parallèle. La structure de la règle est ainsi modifiée : le simple $if - then - else$ est transformé en deux $if - then - else$ impriqués. Cette nouvelle structure permet de considérer les traitements parallèles des deux instructions en une seule opération. En effet, nous associons, aux traitements des deux instructions, un seul temps (un seul composant $countime$). Ainsi, les fonctions de mises à jour appliquées au composant, sont la conjonction ($\wedge$) des fonctions de mises à jour qui lui auraient été appliquées successivement suite à l'exécution des deux instructions. Exemple, si les conditions $i = cache\_hit$ et $i + 1 = cache\_hit$ sont toutes deux vérifiées, la fonction de mise à jour activée pour le composant P est la conjonction de sa première mise à jour $P := F\_i$ et sa deuxième mise à jour $P := F\_i + 1$. De cette façon, la fonction de mise à jour activée pour le composant P est $P := F\_i \wedge F\_i + 1$

```
if i=cache_hit
then
    P:=F_i,
    IC:=IC,
    DC:=DC
    if i+1=cache_hit
    then
        P:= P /\ F_i+1,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+th
    else
        P:= P /\ F_i,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+th
    endif
else
    P:=F_i;i+1,
    IC:=IC /\ i; i+7,
    DC:=DC,
    countime:=t+tm
endif
```

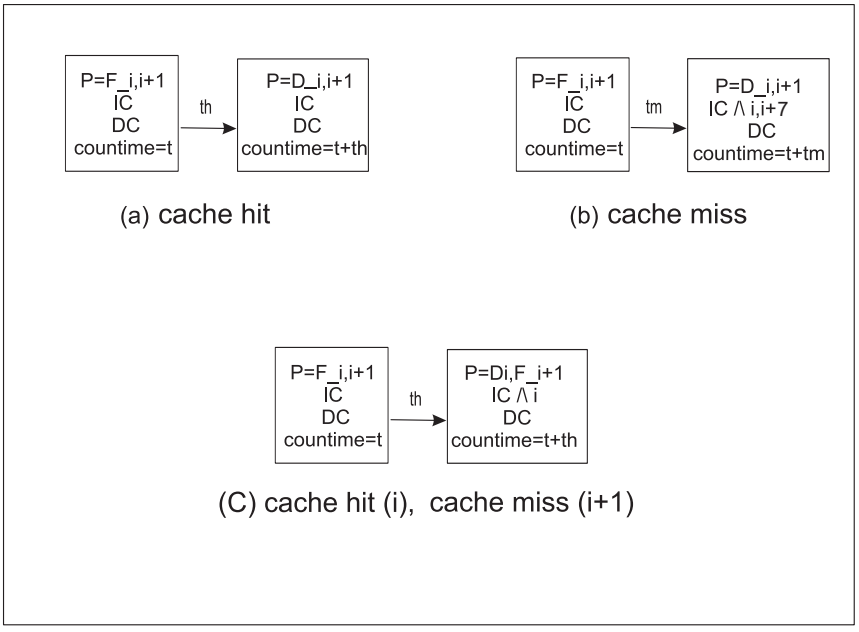


FIGURE 3.15: Evolution du modèle Time-accurate du Fetcher

3.5.2.2 Modèle du dispatcher

La figure 3.16 montre une fonction qui permet d'envoyer les instructions vers une unité d'exécution. En effet, cette unité doit, à chaque cycle, décoder les deux premières instructions qui sont entrées dans l'IQ afin de les transférer vers l'unité d'exécution appropriée. Comme le montre la figure 3.16, le dispat-

cher prend la première instruction qui est entrée dans l'IQ. Ensuite, le D identifie le type d'instruction et en fonction du résultat l'envoie vers l'unité d'exécution appropriée. Le même traitement est appliquée pour la deuxième instruction. Le PowerPC603 effectue cette opération dans l'ordre dans lequel les instructions sont rangées au niveau du programme (in-order). Néanmoins, nous disons que le PowerPC603 exécute les instructions partiellement dans le désordre, du fait que si une instruction, après avoir été envoyée vers son unité, n'a toujours pas fini de s'exécuter, les instructions suivantes sont tout de même envoyées vers d'autres unités et peuvent donc terminer leurs exécutions avant l'instruction en question.

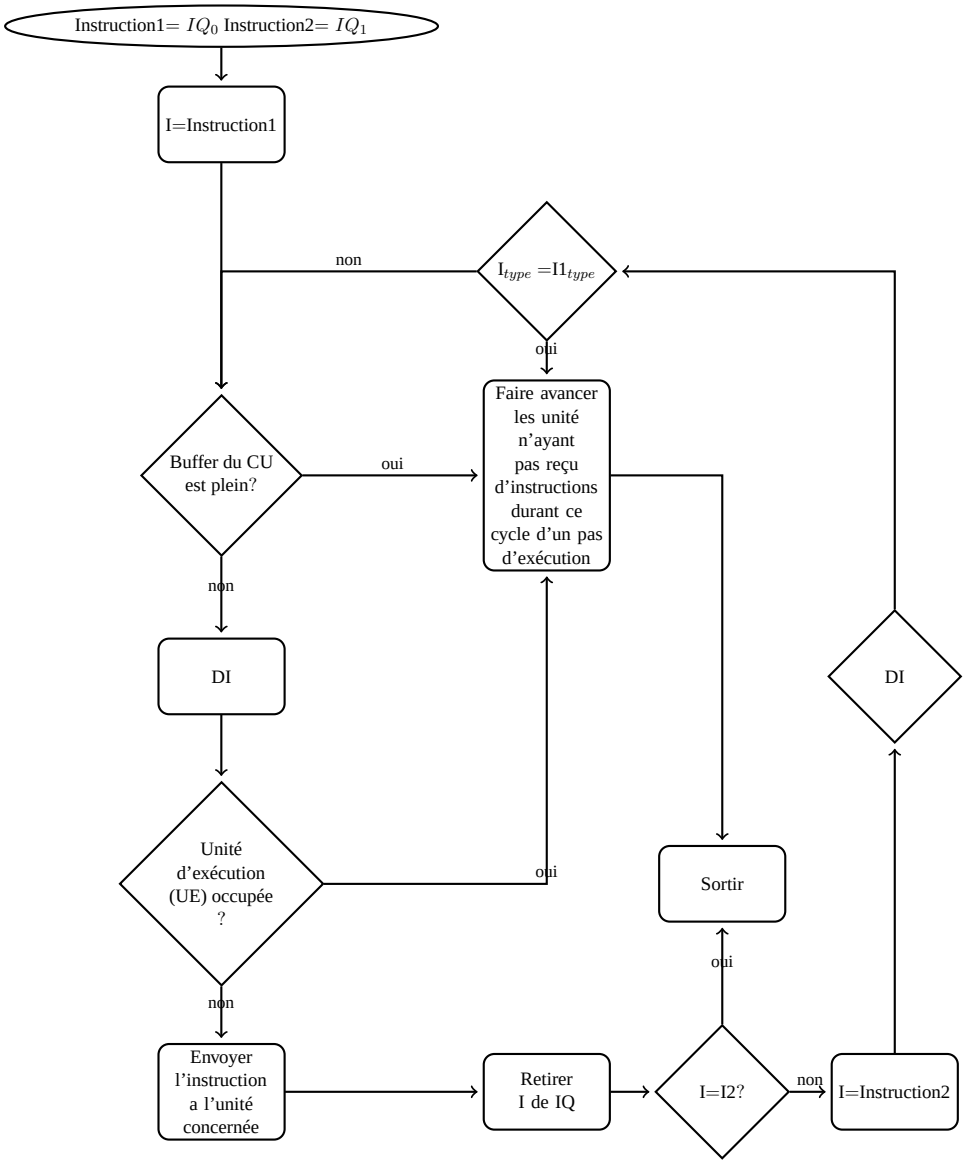


FIGURE 3.16: Dispatcher une instruction : Dispatcher(D)

Comme pour le modèle du composant F, le D traite également deux instructions simultannément. Ainsi, le raisonnement, concernant la conjonction des fonctions de mises à jour du composant F, est également utilisé pour définir l'évolution du composant D. Notons que $D[1]$ signifie que le D traite

au $maximum$ une seule instruction lorsqu'il est sollicité pour traiter l'instruction i. Si cette condition est vérifiée le décodage de i est lancé. Ensuite, le même test est fait, une seconde fois, pour décider si l'instruction $i+1$ peut être traitée durant ce cycle.

A présent, si lors de l'exécution de l'instruction i, la condition D[1] n'est pas vérifiée, alors cette instruction ne sera dispatchée qu'après $t1$ cycles. La valeur de $t1$ est déduite du compteur interne du composant D (le temps nécessaire pour qu'il termine le traitement d'au moins une de ses instructions). A ce point de l'exécution (après $t1$ cycles), la règle ci-dessous est à nouveau appliquée de façon à savoir si le traitement de l'instruction $i+1$ est également possible.

```
if D[1]
then
    P:=D_i,
    IC:=IC,
    DC:=DC
    if D[1]
    then
        P:= P /\ D_i+1,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+1
    else
        P:=P /\ F_i+1,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+1
    endif
else
    P:= D_i,
    IC:=IC,
    DC:=DC,
    countime:=t+t1
endif
```

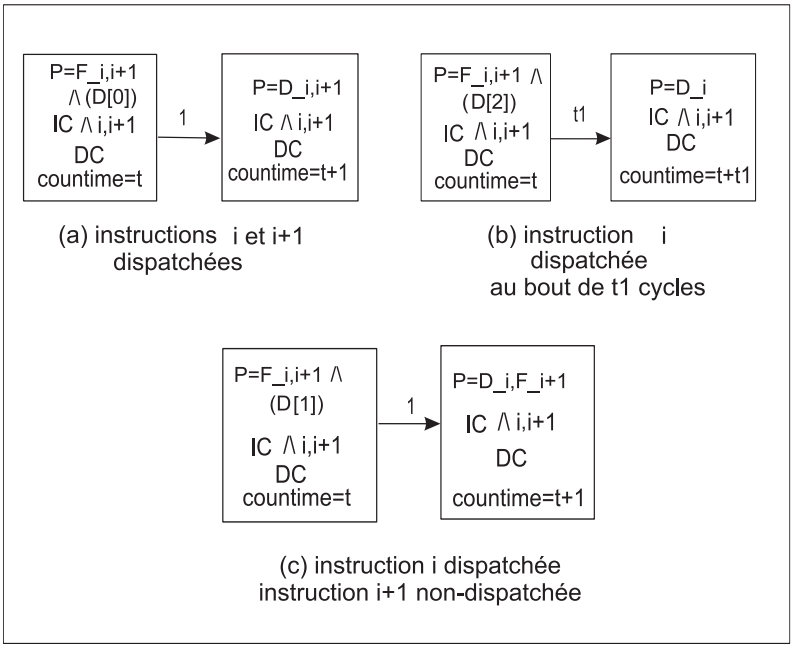


FIGURE 3.17: Evolution du modèle Time-accurate du Dispatcher

3.5.2.3 Modèles des unités d'exécution

Dans cette partie, une distinction est faite entre les unités d'exécution et l'unité de branchement. Cette dernière, ayant un comportement particulier, sera traitée séparément dans la partie 3.5.3.

Ainsi, Le modèle (EX) comprend quatre types d'unité d'exécution :*unité entière (IU), unité flottante (FPU), unité de lecture et d'écriture mémoire (LSU), unité de registres système (SRU).*

Chacune de ces unités, durant l'exécution, associe à chacune des opérations qu'elle exécute, un nombre de tops d'horloge précis. Ces nombres sont appelés les durées des micro-opérations et sont répertoriés dans [52].

La LSU ainsi que la FPU exécutent des instructions en parallèle, alors que l'IU et la SRU ont toutes deux un comportement séquentiel. Ce qui nous a conduit à présenter deux exemples d'unité :

- le premier séquentiel est représenté par l'IU. Son fonctionnement est modélisé par la figure 3.18,
- le second pipeliné est illustré par la figure 3.19. Elle montre le fonctionnement de la LSU qui comprend un micro-pipeline de deux étages et peut ainsi exécuter jusqu'à deux opérations au même moment.

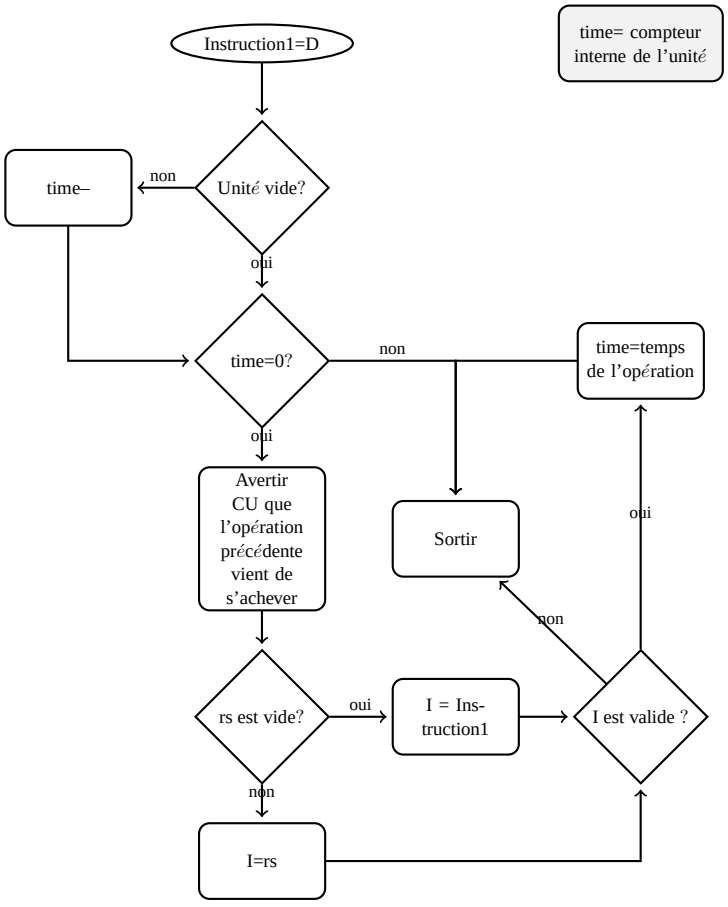


FIGURE 3.18: Exécuter une opération : Unité entière (IU)

De façon générale, une unité ayant un comportement séquentiel exécute les instructions du programme une à une et ne peut donc commencer le traitement d'une instruction que si l'exécution de l'instruction précédente est terminée. Alors qu'une unité d'exécution pipelinée place les instructions dans le premier étage de son micro-pipeline (si ce dernier est libre). Ensuite, l'exécution se poursuit par le transfert de l'instruction qu'elle contient du premier au second étage de son micro-pipeline. Le premier étage étant à présent libre, il peut être occupé par l'instruction suivante. L'exécution de chaque instruction se termine lorsque celle-ci sort du dernière étage du micro-pipeline. Ainsi, ce type d'unité peut commencer l'exécution de l'instruction suivante avant d'achever le traitement de l'instruction courante.

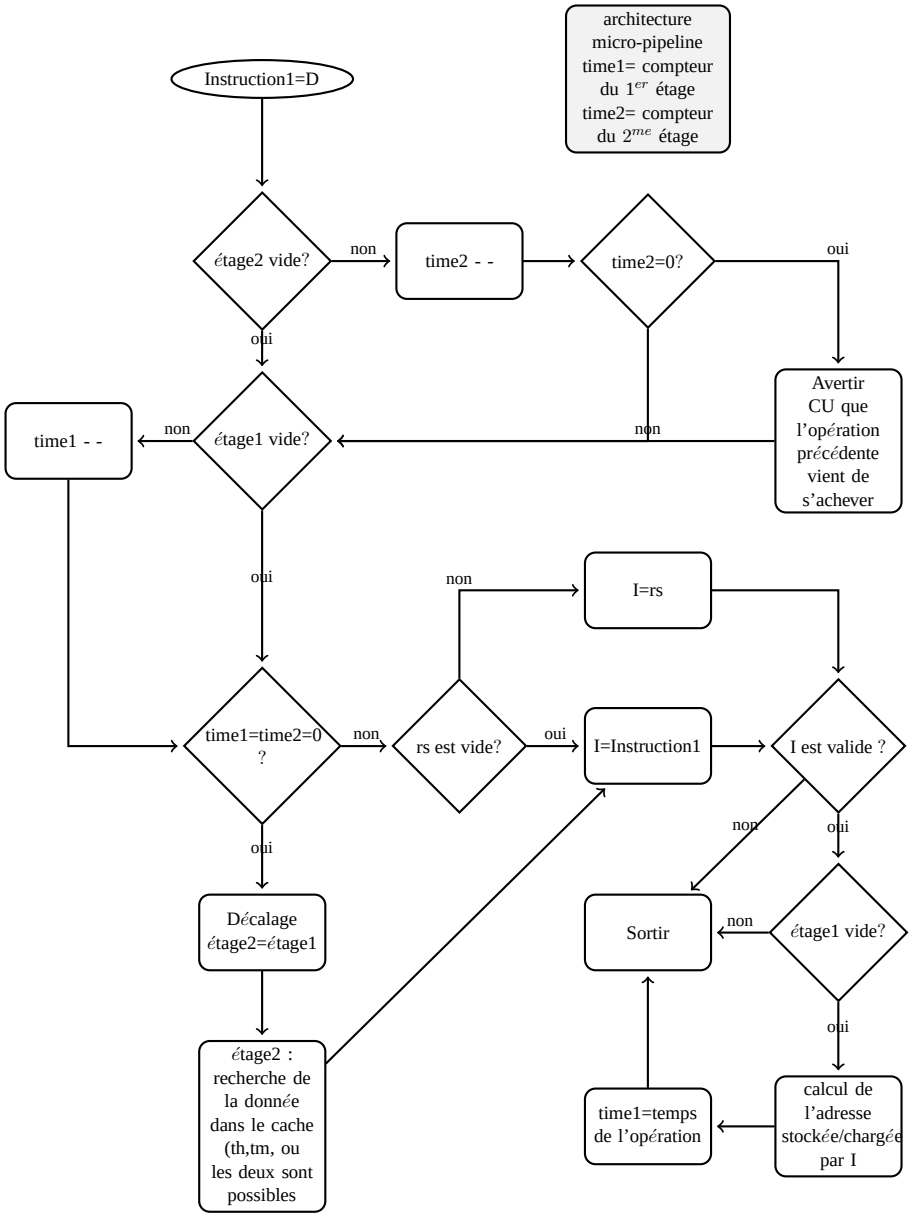


FIGURE 3.19: Exécuter une opération : Unité de lecture et d'écriture mémoire (LSU)

L'intérêt de cette étape est que les deux composants (pipeliné ou séquentiel) $EX(i)$ et $EX(i+1)$ reçoivent respectivement les instructions i et $i+1$. Notons que l'expression $EX(i)$ désigne le composant en charge de traiter l'instruction i. Ainsi, l'instruction i est traitée que si $EX(i)$ est libre et qu'il demeure au moins une place de libre dans le buffer d'exécution du CU (voir le modèle ci-dessous). Notons que l'expression $CU[n-1]$ signifie que : pour un buffer d'exécution comportant n emplacements, au mois un emplacement demeure vide.

A présent, admettons que l'une des conditions précédentes ne soit pas vérifiée ($EX(i) = busy \cup CU[n]$). Ainsi, le prochain état vers lequel évoluera ce composant sera l'exécution de l'instruction i. Cette exécution débutera après $t1$ cycles. La valeur de $t1$ est directement déduite du compteur interne du

composant $EX(i)$. A ce point de l'exécution (après $t1$ cycles), la condition $EX(i) = free \wedge CU[n-1]$, sera vérifiée et donc l'état vers lequel évolura ce composant sera conditionné uniquement par le résultat de l'évaluation de la condition $EX(i+1) = free \wedge CU[n-1]$. Ainsi, un bloc de mise à jour sera selectionné pour traiter l'instruction $i+1$.

```
if EX(i)=free /\ CU[n-1]
then
    P:=EX(i)_i,
    IC:=IC,
    DC:=DC
    if EX(i+1)=free /\ CU[n-1]
    then
        P:=P /\ EX(i+1)_i+1,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+1
    else
        P:=P /\ D_i+1
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+1
    endif
else
    P:=EX(i)_i,
    IC:=IC,
    DC:=DC
    countime:=t+t1
endif
```

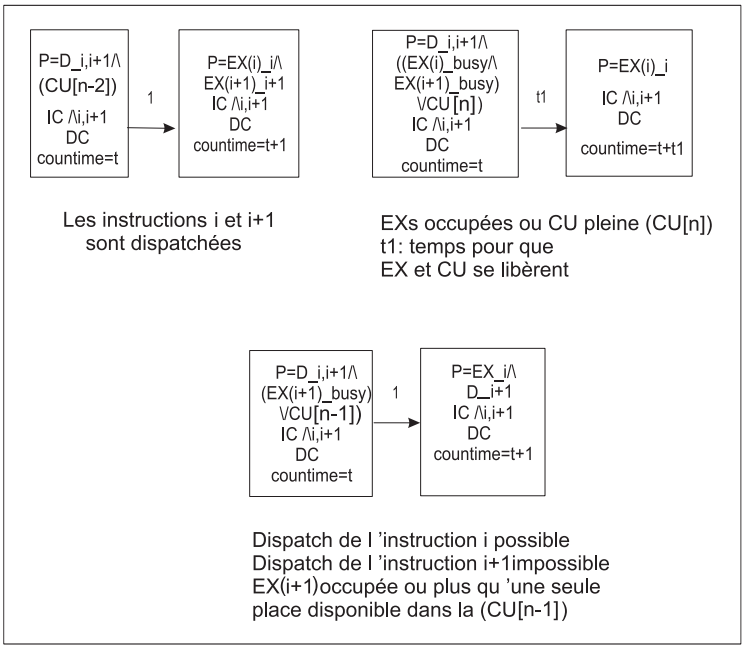


FIGURE 3.20: Evolution du modèle Time-accurate d'une unité d'exécution

3.5.2.4 Modèle de l'unité d'achèvement

Ce composant CU permet de sauvegarder dans un registre approprié les résultats obtenus à la suite du traitement d'une instruction par l'une des unités d'exécution (EX). Son fonctionnement est présenté dans la figure 3.21. Lorsqu'une unité d'exécution a terminé d'exécuter une opération, l'instruction concernée est retirée du buffer d'exécution.

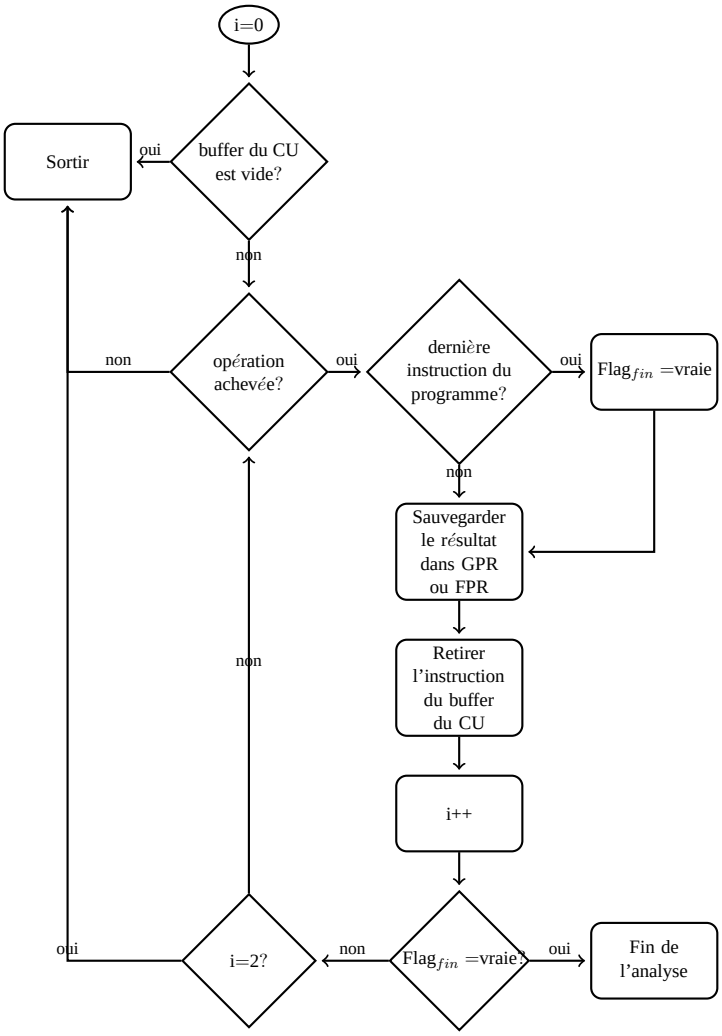


FIGURE 3.21: Retirer une instruction : Unité d'achèvement

Ces instructions sont retirées dans l'ordre dans lequel elles sont entrées dans le buffer (FIFO). Ainsi, le composant CU teste l'instruction i avant l'instruction $i+1$. Ce composant permet donc de maintenir l'ordre imposé par le programme analysé et cela même dans le cas où l'exécution de l'instruction $i+1$ se termine avant ses prédécesseurs (instruction i).

A présent, si lors de l'exécution de l'instruction i, la condition $i\_finish = true$ n'est pas vérifiée, alors cette instruction ne sera retirée qu'après $t1$ cycles. La valeur de $t1$ est déduite du compteur interne du composant $EX(i)$ (le temps nécessaire pour qu'il termine le traitement de l'instruction i). A ce point de l'exécution (après $t1$ cycles), la règle, ci-dessous, est à nouveau appliquée de façon à savoir si l'ins-

truction $i+1$ peut, également, être retirée.

```
if i_finish=true
then
    P:=CU_i,
    IC:=IC,
    DC:=DC,
    if i+1_finish=true
    then
        P:=P /\ CU_i+1,
        IC:=IC /\ IC,
        DC:=DC /\ DC,
        countime:=t+1
    else
        P:=P /\ EX(i+1)_i+1,
        IC:=IC,
        DC:=DC,
        countime:=t+1
    endif
else
    P:=CU_i,
    IC:=IC,
    DC:=DC,
    countime:=t+t1
endif
```

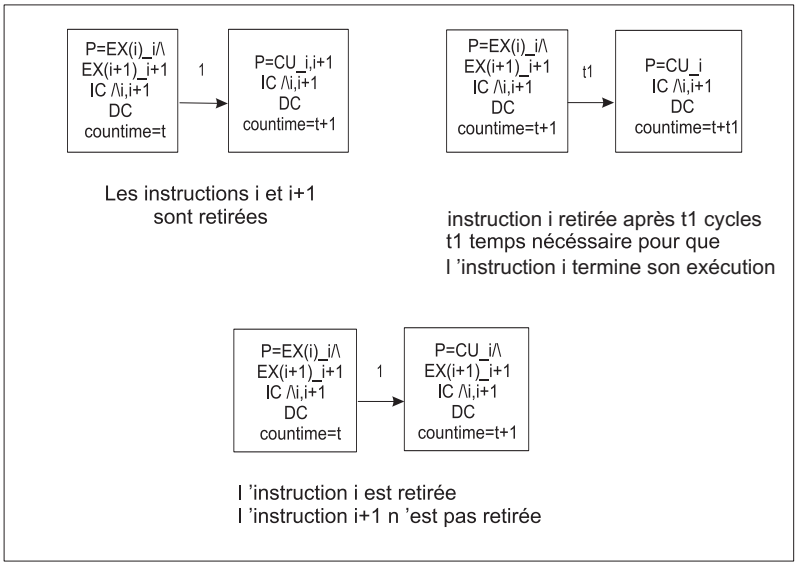


FIGURE 3.22: Evolution du modèle Time-accurate de l'unité d'achèvement

3.5.3 Modèle de l'unité de branchement

Une représentation précise du comportement de l'unité de branchement est d'une importance cruciale. Vu que les boucles et les instructions de saut ont un impact considérable sur les temps d'exécution, l'unité de branchement requiert une étude approfondie. Cette unité doit être modélisée de façon à reproduire avec précision plusieurs comportements complexes. Exemple, dans le cas d'une exécution spécu-

lative (voir section 3.4.1.4), le modèle doit être capable de gérer le scénario représentant une prédiction correcte et celui représentant une prédiction incorrecte. De plus, dans le cas d'une prédiction incorrecte le modèle devra entreprendre toutes les actions prévues, comme : (1) effacer toutes traces d'exécution des instructions qui succèdent à l'instruction de branchement (2) réorienter l'exécution en utilisant l'adresse calculée.

Ainsi, la règle qui définit l'évolution de ce composant s'exprime de la façon suivante : lorsqu'une instruction de branchement conditionnel est fetchée, une prédiction est immédiatement faite. Le composant BPU choisit une adresse de saut et applique la règle du composant F grâce à la fonction $call\_F\_Rules(instruction1, ins$ De cette façon, l'exécution se poursuit suivant l'instruction sélectionnée.

```
if flag_prediction
then
    call_F_Rules(i_jump1, i_jump1+1)
else
    call_F_Rules(i_jump2, i_jump2+1)
endif
```

Ensuite, le composant BPU est de nouveau sollicité pour vérifier si la prédiction faite est correcte. Si ce n'est pas le cas, un ensemble de fonctions est appellé ($erase(i > i\_jump$). Chaque fonction est appliquée à un composant spécifique. Son rôle est d'effacer du composant toutes les instructions qui lui sont parvenues suite à la prédiction de branchement. Ensuite, le composant BPU appelle de nouveau la règle associée au composant F en lui fournissant les adresses calculées.

```
if prediction_incorrecte
then
    P:=P_erase(i>i_jump),
    IC:=IC_erase(i>i_jump),
    DC:=DC_erase(i>i_jump),
    countime=countime+1,
    call_F_Rules(i_computed, i_computed+1)
endif
```

3.5.4 Coordination et cohérence entre les unités

Le composant History est l'élément central de l'abstraction proposée. En effet, comme le montre la figure 3.23, cette classe est en liaison permanente avec toutes les unités de traitement. Ainsi, à travers cette classe, il nous est possible de connaitre à tout moment l'état du processeur.

Ce composant virtuel a été rajoutée au modèle du processeur dans le but de répondre à plusieurs besoins dont voici une liste non-exhaustive :

- Garder durant l'exécution une vue générale de l'évolution du modèle.
- Gérer les problèmes d'incohérence inhérents aux modèles implémentés de manière séquentielle et voulant simuler le comportement d'unités évoluant en parallèle.

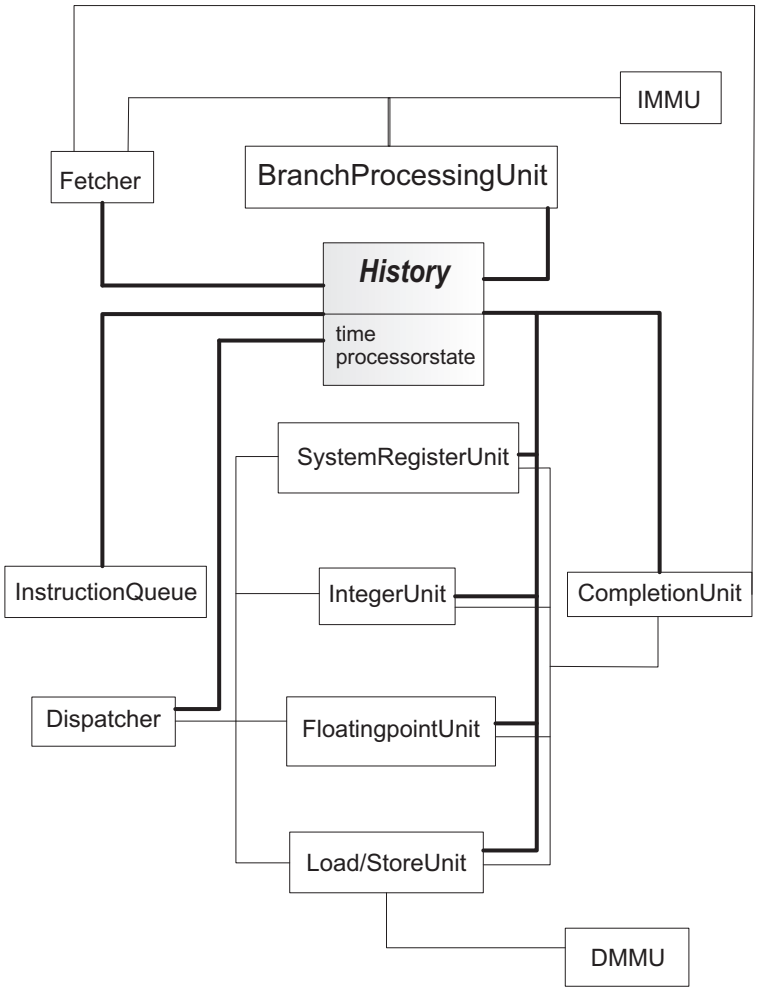


FIGURE 3.23: Maintenir une cohérence entre les unités de traitement : classe History

3.6 Pré-requis de l'approche

La mise en œuvre de l'approche implique de disposer, d'une part, d'un modèle exécutable "Time-accurate" de l'architecture cible, et d'autre part, d'un interpréteur de code.

3.6.1 Modélisation

Le modèle exécutable est une représentation du processeur cible comportant tous les comportements propres au processeur.

3.6.2 Analyse du binaire

L'interpréteur, indépendant de l'architecture cible, sert a décoder les instructions de façon à identifier certaines informations comme : les compteurs de boucles, l'ordre des instructions dans le programme, leurs adresses, etc.

Comme nous pouvons le constater sur la figure 3.25, cet interpréteur est réactif. En effet, il s'adapte aux exigences de l'exécution du modèle Time-accurate de manière à fournir des informations plus précises durant le calcul du WCET. Dans ce contexte, les traces sont utilisées afin de sauvegarder la dernière adresse qui a été écrite. Ce qui est semblable à un graphe de dépendances [27] qui permet de récupérer des informations, en fonction des besoins, à une adresse dans la mémoire. Le mécanisme de récupération d'informations est défini par une analyse arrière qui permet de déterminer la dernière fois qu'une adresse a été écrite.

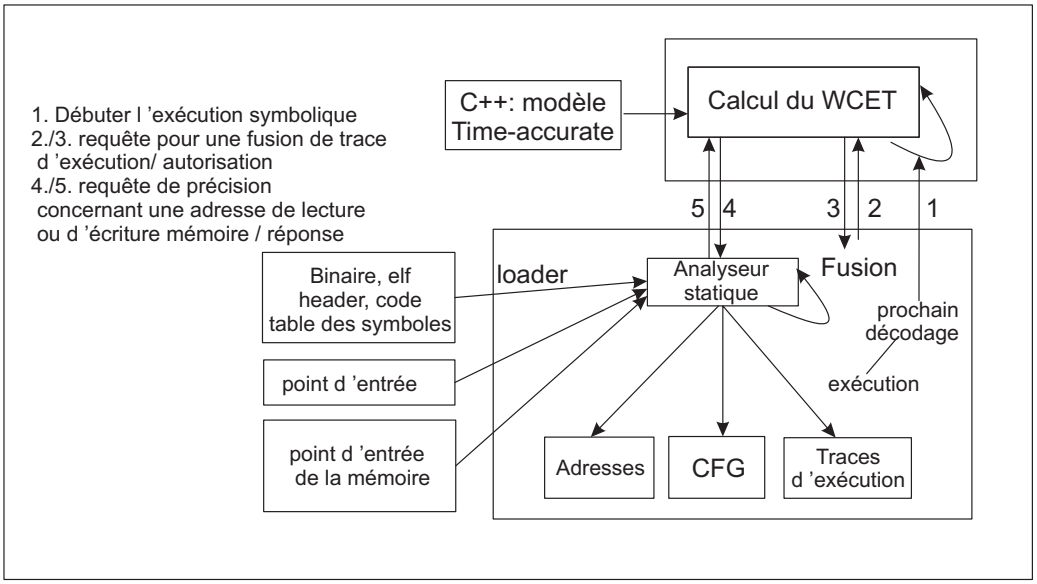


FIGURE 3.24: Interpréteur de code (CEA Saclay)

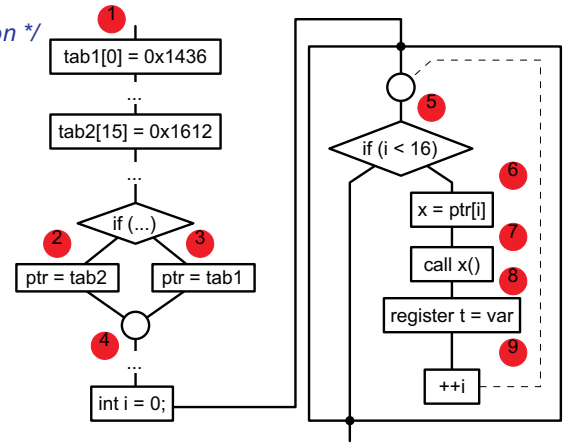
L'algorithmique général qui calcule le modèle d'évolution présente le principal problème de l'analyse statique, à savoir l'adéquation entre la précision des calculs et leur utilisation. Plus une interprétation abstraite est précise, plus elle met de temps à être manipulée et plus les convergences sont lentes. Au pire, l'interprétation abstraite ne se termine pas et explose son modèle mémoire. À l'inverse, plus une interprétation abstraite est imprécise, plus elle est rapide, mais plus elle donne des résultats surévalués. Au pire l'interprétation abstraite doit être arrêtée car nous ne pouvons plus savoir quelle est la prochaine instruction qui doit être exécutée.

La stratégie adoptée est une interprétation abstraite très approchée pour l'interprétation du code binaire. Cette stratégie est associée à un mécanisme de récupération d'informations qui est lancé en fonction des besoin de l'analyse. L'interprétation du code binaire construit un graphe de contrôle et un graphe de dépendance. Les pertes de précisions sont gérées à posteriori avec un mécanisme de récupération d'informations basé sur le graphe de dépendance. L'expression "très approché" signifie une propagation des constantes sur les valeurs par défaut. Les appels fonctionnels et les pointeurs dans le code sont traités de manière exacte avec des variables de branchement. La récupération d'information implique d'avoir une connaissance exacte des branchements non résolus et d'avoir une information sous la forme d'un intervalle pour les lectures et les écritures sur des adressages indirects. Elle implique également d'avoir une information sous forme d'un intervalle (avec des bornes exactes) pour toutes les variables correspondant à des compteurs de boucle.

Le fonctionnement de cette algorithmique est présenté sur un exemple représentatif. Il est écrit en C, mais sa compilation en code binaire est relativement directe :

```
/* initialisation des tables de pointeurs de fonction */
1  tab1[0] = 0x1436;      tab2[0] = 0x1522;
   tab1[1] = 0x1444;      tab2[1] = 0x1542;
   tab1[2] = 0x1458;      tab2[2] = 0x1562;
   …
   tab1[15] = 0x1488;     tab2[15] = 0x1612;

   …
   /* ptr est un objet */
   if (…)
2    ptr = tab1;
   else
3    ptr = tab2;
4
   for (int i =0; 5 i < 16; ++i) {
     /* i est une méthode associée à l'objet ptr */
6    x = ptr[i];
7    call x();
8    register t = var; /* variable affectée par une des méthodes */
9    …
   }
```

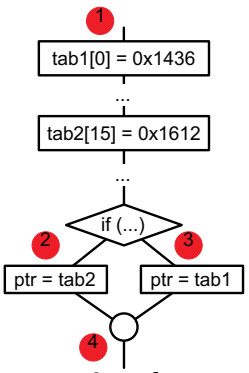


version finale du graphe de contrôle hiérarchisé

FIGURE 3.25: Exemple de code

La construction du graphe de contrôle a lieu simultanément avec l'interprétation des instructions. Dans ce contexte, voici comment fonctionne l'algorithmique générale sur l'exemple. L'interprétation abstraite débute avec le point 1 . Elle se sépare sur les points 2 et 3 et est fusionnée au point 4 . A ce point :

- ptr = $\top$, indiquant que nous n'avons plus d'information quant au contenu de ptr, étant donné que la première phase ne fait qu'une propagation des constantes.
- Le graphe de contrôle construit est défini ci-contre :



La propagation des constantes permet ensuite de savoir que i vaut 0 et de passer le 1^{er} test avec succès. L'interprétation se trouve alors au point 6 à vouloir lire le contenu du pointeur. La stratégie est d'imposer une valeur sous forme d'intervalle au pointeur, même si le fait que la valeur lue soit $\top$ ne gène pas l'émulation. A ce moment, les graphes de contrôle et de dépendance sur le binaire sont représentés par le premier schéma de la Figure 3.26. Le mécanisme de récupération d'information revient en arrière sur les graphes de dépendances et propage en avant les informations requises pour avoir le niveau de précision souhaité comme le précise la Figure 3.26.

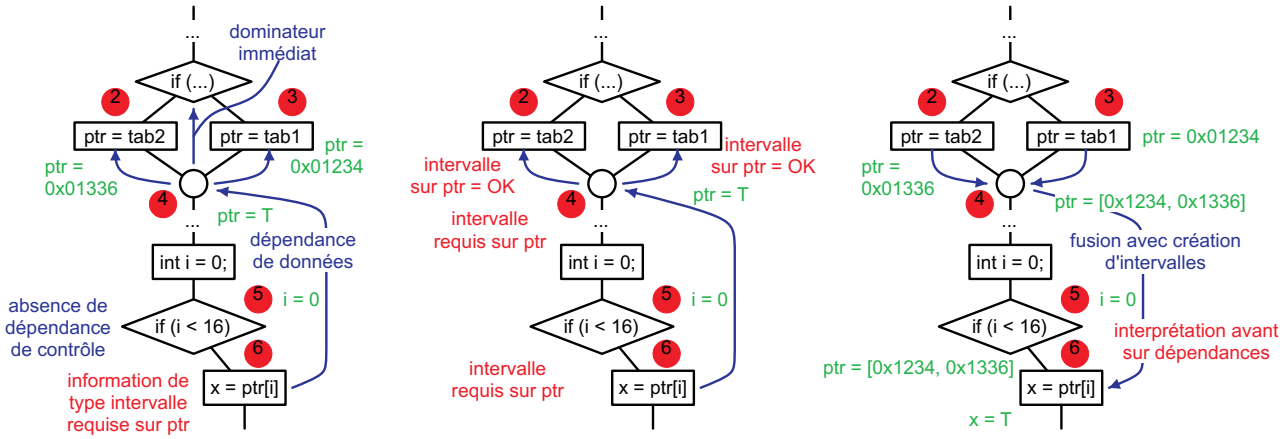


FIGURE 3.26: Récupération d'informations sur adressage indirect

Nous atteignons donc le point 7 correspondant à l'appel fonctionnel. Cet appel requiert une information exacte sur la variable x, ce qui implique une information exacte sur ptr et non plus une information sous la forme d'intervalle. L'algorithmique de récupération d'information est donc reproduite avec l'introduction de la dépendance de contrôle *cond_br* indiquant de quelle branche du test on provient (voir la Figure 3.27).

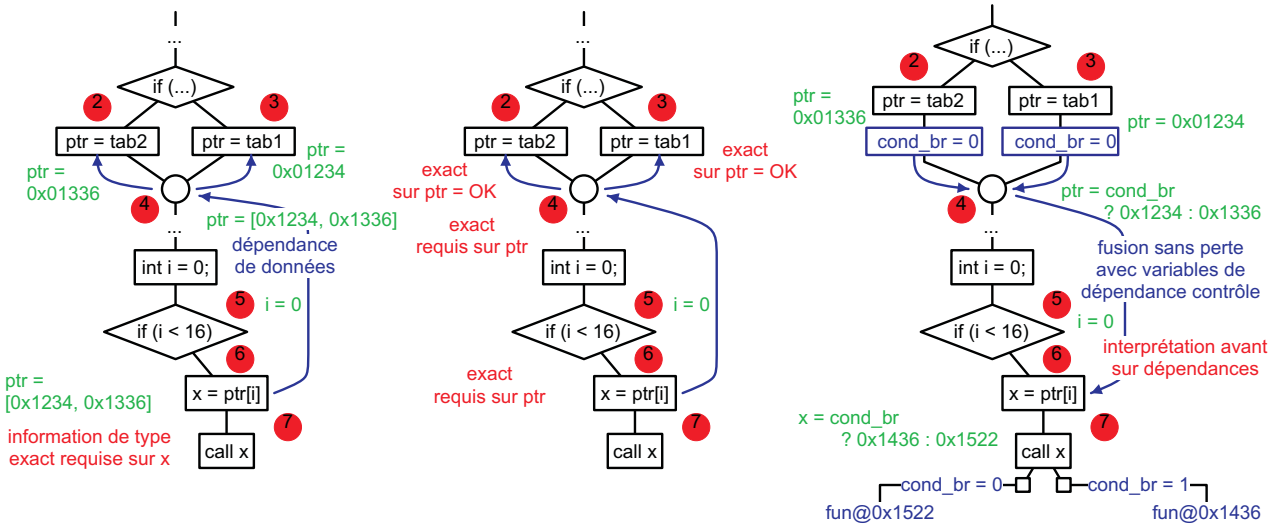


FIGURE 3.27: Récupération d'informations sur appel fonctionnel indirect

Les appels fonctionnels sont traités comme le reste du code. L'instruction suivante incrémente le registre i et reboucle via un jump inconditionnel sur une instruction déjà existante. Une sous-numération est alors automatiquement créée. Cette dernière a pour objectif d'identifier formellement l'ensemble des chemins partant de la cible du saut et qui atteignent la source du saut. Cet ensemble de chemin est nécessairement organisé en Directed Acyclic Graph en rouge sur le graphe de la Figure 3.28.

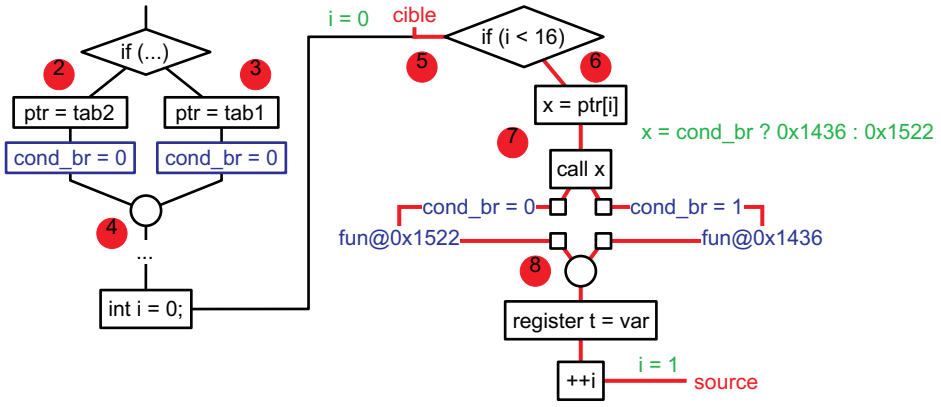


FIGURE 3.28: Chemins allant de la cible du saut à l'origine du saut

Une interprétation abstraite est alors lancée sur le chemin (en rouge sur la Figure 3.28) correspondant à une boucle.

Dans le cadre de notre exemple, max est borné par la valeur 15, indiquant un nouveau comportement lorsque compteur = 15+1. Ce nouveau comportement sort nécessairement de la boucle.

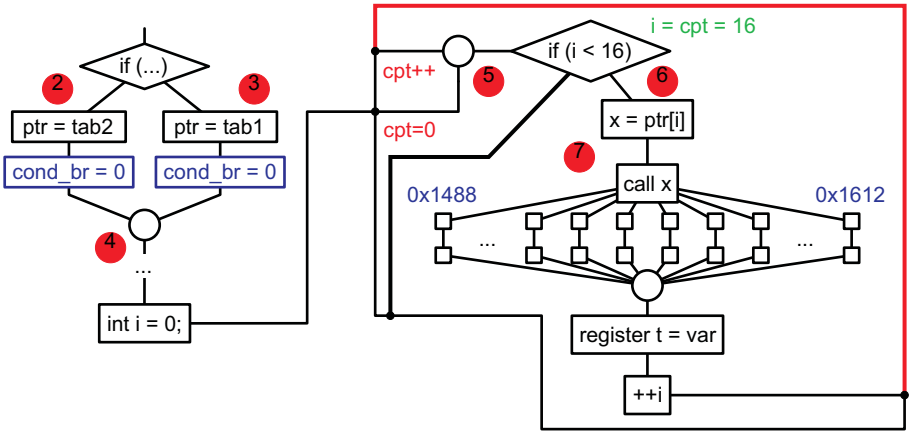


FIGURE 3.29: Passage à la limite et sortie de boucle

Ainsi, grâce à cet exemple nous avons pu montrer la logique suivie par l'interpréteur de code. Celui-ci nous a été fourni par le CEA Saclay, pour plus de détails sur cette partie vous pouvez vous référer à [42].

3.7 Conclusion

La mise en œuvre de l'approche impose qu'un modèle exécutable du processeur cible soit disponible. Ce modèle doit être construit de façon à pouvoir simuler tous les comportements du processseur qui ont un impact sur les temps d'exécution. Ainsi, la capacité du modèle à représenter finement le fonctionnement du processeur, garantit que les temps d'exécution obtenus suite à la construction de l'arbre

symbolique, seront précis (idéalement des valeurs exactes). Dans le chapitre suivant, en utilisant notre extension de l'exécution symbolique, nous décrivons la construction de l'arbre.

CHAPITRE 4

Calcul du pire temps d'exécution

4.1 Introduction

Notre approche se compose essentiellement de deux grandes parties : (1) exécution symbolique d'un modèle Time-accurate et (2) fusion des états générés. Dans ce chapitre nous allons nous intéresser à la première partie de notre analyse. La seconde est traitée dans le chapitre suivant.

L'exécution symbolique dans sa forme actuelle est une méthode formelle qui permet d'obtenir une couverture totale des comportements d'un **programme**. Le principe consiste à remplacer les données réelles injectées habituellement à l'entrée des programmes par des données symboliques (des données dont les valeurs sont des symboles) et de lancer l'exécution. Nous avons choisi de nous inspirer de la logique de cette technique afin d'obtenir tous les comportements possibles d'un **processeur**. En effet, notre raisonnement consiste à effectuer deux analyses successives : la première est une exécution symbolique normale qui vise à identifier tous les comportements d'un programme (voir partie 4.2) et la deuxième consiste à considérer chaque instruction, comportant ou pas des opérandes symboliques, comme une donnée symbolique que nous injectons à l'entrée du modèle Time-accurate du processeur (voir partie 4.3) .

Ainsi, à chaque pas d'exécution symbolique du modèle Time-accurate, nous générons un ou plusieurs états. Ces états sont, à leurs tours, le point de départ (l'environnement de départ) des prochains pas d'exécution. Les transitions reliant les états sont respectivement étiquetées par une information temporelle traduisant le temps que prend leurs tirs.

L'analyse se poursuit jusqu'à l'exécution de la dernière instruction du programme. Ainsi, un arbre symbolique est construit. Cet arbre contient tous les états atteignables par le processeur durant l'exécution.

L'expression "tous les états atteignables" implique que même les effets du pipeline sont pris en compte. La partie 4.6 montre que notre analyse traite systématiquement ces comportements imprévisibles comme des comportements possibles (états atteignables) sans besoins d'études supplémentaires.

4.2 Exécution symbolique

Comme nous l'avons précédemment spécifié, le principe de l'exécution symbolique [30, 18, 13, 48] consiste à remplacer les données réelles injectées habituellement à l'entrée des programmes par des données symboliques. Ainsi, les valeurs produites en sortie par le programme sont exprimées en fonction de ces données symboliques. L'exécution des opérations se fait naturellement : la partie gauche de l'expression reçoit une expression symbolique qui est sous forme d'un polynôme.

Par contre, l'évaluation des expressions conditionnelles est un peu plus complexe. En début d'analyse, nous définissons un "path condition" PC - une expression booléenne de données symboliques. Le PC

est un accumulateur de contraintes. Ces contraintes représentent l'historique d'exécution. Ainsi, durant l'exécution, à chaque fois qu'une expression conditionnelle devra être analysée, PC servira à identifier le chemin selon lequel l'exécution devra se poursuivre.

Au début de toute exécution symbolique, le PC est initialisé à "true". Si durant l'exécution une expression conditionnelle est rencontrée, l'évaluation commence par remplacer les variables par leurs valeurs respectives. Cependant, ces valeurs étant des polynômes de symboles, la condition prend donc la forme d'une expression de type : $P > 0$, où P est un polynôme. Soit R l'expression $P > 0$. Ainsi, trois scénarios sont possibles :

- **PC** $\supset R$ et **PC** $\not\supset \neg R$: L'expression est toujours vraie, l'exécution se poursuit avec le code conditionnel (le code associé au bloc $then$)
- **PC** $\supset \neg R$ et **PC** $\not\supset R$: L'expression est toujours fausse, l'exécution se poursuit avec le code associé au $else$ si ce code existe, sinon elle saute le code conditionnel.
- Sinon, la condition Booléenne peut être vraie ou fausse (**PC** $\supset R$ et **PC** $\supset \neg R$). Dans ce cas, le PC est divisé en deux parties : la première est le PC vrai $\mathbf{PC}_{\text{true}} = \mathbf{PC} \wedge R$ et le second est le PC faux $\mathbf{PC}_{\text{false}} = \mathbf{PC} \wedge \neg R$. Dans ce cas, l'exécution se poursuit le long des deux chemins d'exécution. L'une exécute le code conditionnel avec un $\mathbf{PC}_{\text{true}}$ et l'autre exécute le code du bloc "else" (ou le code localisé après le bloc "then" dans le cas où l'expression "if" ne contiendrait pas de "else") avec le "path condition" $\mathbf{PC}_{\text{false}}$.

Le point fort de cette méthode réside principalement dans le rôle que joue le PC durant l'exécution. En effet, grâce entre autre à ce PC, cette méthode palie aux inconvénients habituellement introduits par les méthodes d'analyse statique. Cette simple accumulation d'expressions booléennes permet de connaitre à chaque moment une partie de l'historique d'exécution. Cette partie contient toutes les décisions de branchement prises précédemment. Ainsi, les instructions de branchement conditionnel sont traitées avec plus de précision et les chemins infaisables ne sont jamais pris en compte durant l'analyse.

4.3 Extension de l'exécution symbolique

Concernant l'approche proposée, l'exécution symbolique est utilisée afin de déterminer à chaque pas d'exécution tous les comportements possibles du processeur.

Comme le montre la figure 4.1, cette exécution symbolique apparait à deux endroits dans l'analyse. Le premier, que nous appelerons "haut niveau", sert à engendrer tous les comportements possibles du programme. En effet, une exécution symbolique normale est lancée : des valeurs symboliques sont attribuées aux données en entrée et nous obtenons en sortie un arbre représentant tous les comportements que le programme est susceptible d'avoir. Cette première partie montre la pertinence de l'exécution symbolique : nous débutons l'analyse par des données d'entrée dont les valeurs sont inconnues et au fur et à mesure que l'analyse avance ces valeurs se précisent. Ce raisonnement est à la base de l'extension de l'exécution symbolique que nous proposons (voir figure 4.1). Ainsi, le second endroit où l'exécution symbolique apparait est appelé "bas niveau" et sert à identifier tous les comportements que le processeur peut avoir durant l'exécution. La logique suivie s'inspire directement de l'exécution symbolique normale. En supposant qu'un modèle de processeur soit disponible, considérons le comme un simple programme. Ce programme prendra donc en entrée les instructions de l'application que nous souhaitons analyser. Ainsi, en initialisant les paramètres des unités du modèle avec des valeurs symboliques (inconnues) et en l'exécutant, nous obtenons comme pour une exécution symbolique normale tous les comportements possibles du modèle et donc du processeur. Pour bien comprendre le raisonnement proposé, prenons l'exemple de la mémoire cache. Supposons que nous initialisons cette mémoire au début de l'analyse avec la valeur "inconnue". Autrement dit, toutes les lignes du cache contiennent la valeur inconnue et donc aucune information n'est connue concernant l'état du cache. Ensuite, nous lançons l'analyse. La

première étape consiste évidemment à exécuter la première instruction du programme. Nous allons donc tenter de récupérer cette instruction du cache. Mais, vu que ce dernier est à l'état inconnu nous sommes dans l'obligation de générer deux cas : (1) instruction dans le cache (hit), (2) instruction pas dans le cache (miss).

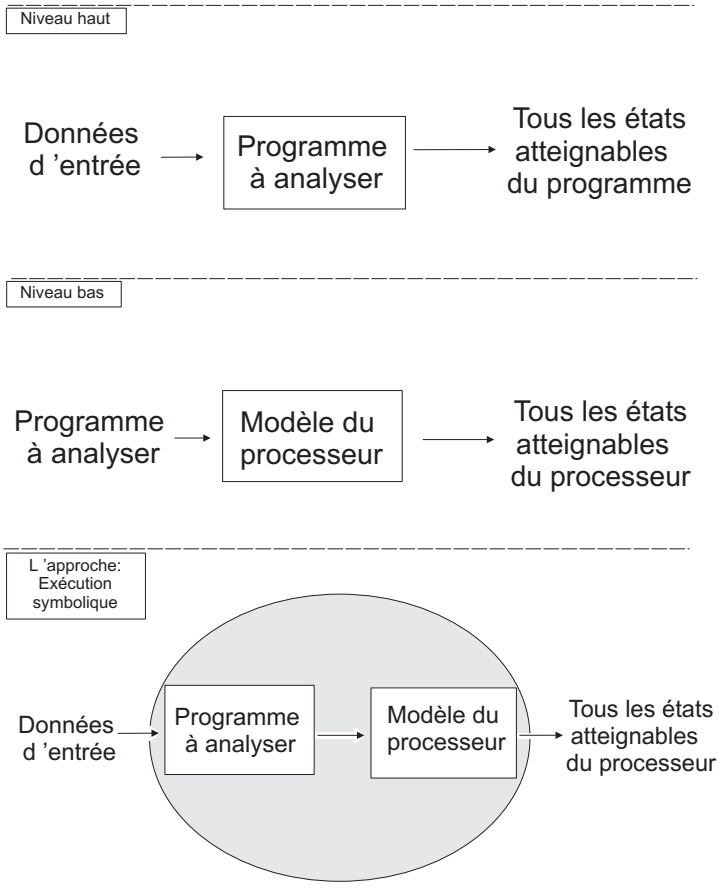


FIGURE 4.1: Extension de l'exécution symbolique

Ainsi, la trace d'exécution initiale est divisée en deux branches qui doivent être analysées séparément. En appliquant ce raisonnement aux instructions suivantes et en conservant une historique d'exécution propre à chaque trace d'exécution, nous pouvons générer tous les comportements possibles de la mémoire cache durant l'exécution. Notons par ailleurs que l'état du cache se précise au fur et à mesure que l'analyse avance.

A présent, si nous généralisons ce raisonnement à l'ensemble des unités du processeur nous pouvons non seulement identifier tous les comportements du processeur mais aussi prendre en compte toutes les interactions susceptibles d'apparaitre entre les unités durant l'exécution (voir chapitre 4.4).

4.4 Détermination des traces d'exécution possibles

Afin de construire l'arbre symbolique nous considérons les instructions du programme à analyser comme des entrées symboliques que nous injectons au modèle du processeur. Ainsi, l'arbre d'exécution généré contient tous les états atteignables par le processeur durant l'exécution. Ces états sont reliés par

des transitions où chaque transition est associée à un temps qui représente la durée nécessaire à son tir. De plus, chaque état contient également un compteur de temps. Au moment où cet état est généré, il incrémente son compteur interne de la durée associée à sa transition entrante (le label de sa transition entrante). Ainsi, à la fin de l'analyse, une simple comparaison entre les états finaux nous permet d'identifier le pire temps d'exécution (WCET).

4.4.1 Analyse de données

L'analyse de données est faite en amont par l'interpréteur de code (voir partie 3.6.2). Ainsi, notre outil de calcul des temps d'exécution s'appuie sur cette analyse pour accéder aux adresses des instructions et des données.

A cet effet, l'interpréteur de code renvoie, sur requête de notre outil, soit :
- une valeur d'adresse.
- un intervalle d'adresse.
- un $\top$ (valeur inconnue).

Dans le dernier cas le modèle de la mémoire cache est remis à son état initial (IC= $\top$ ou DC= $\top$). Ce qui signifie que nous affectons une valeur symbolique (inconnue) à toutes les lignes du modèle de la mémoire cache.

A présent, concernant la sémantique des instructions exécutées symboliquement, celle-ci est similaire à la sémantique des instructions exécutées naturellement. Notre approche s'inspire des travaux de Lundqvist et Stenstrom [37, 36], ainsi la sémantique des instructions est étendue de façon à manipuler des valeurs inconnues (voir tableau 1.1). Autrement dit, l'exécution d'une opération dont une de ses opérandes est symbolique (valeur inconnue) produit un résultat de valeur inconnu. Exemple :
c= a+b $\Rightarrow$ c= unknow if (a= unknow $\vee$ b=unknow).

4.4.2 Exécution du modèle temporisé

Le modèle temporisé de l'architecture est exécuté symboliquement pour un programme donné.

L'exécution symbolique du modèle prend en entrée l'état courant $\mathbf{s}$ du processeur et retourne un ensemble d'état finaux ainsi que le temps nécessaire pour atteindre ces états finaux : $\{(\mathbf{s_1}, \mathbf{t_1}), \ldots, (\mathbf{s_n}, \mathbf{t_n})\}$.

Définition 13 ***Etats intermédiaires*** *Si* $\mathbf{s}$ *est un état valide, on appelle "états intermédiaire" les états* $\{(\mathbf{s'_1}, \mathbf{t'_1}), \ldots, (\mathbf{s'_n}, \mathbf{t'_n})\}$ *générés suite à une seule exécution symbolique (un seul pas d'exécution) du modèle processeur en commençant cette exécution par l'état* $\mathbf{s}$*.*

Définition 14 ***Arbre symbolique temporisé*** *C'est un graphe* $\mathcal{G} = (\mathcal{N}, \mathcal{TR}, \mathcal{M})$ *qui représente le résultat de l'exécution symbolique d'une séquence de code et qui est défini comme suit :*
- *les noeuds* $\mathcal{N}$ *du graphe sont des états symboliques,*
- *les transitions* $\mathcal{TR}$: $(\mathcal{N} \times \mathcal{N})$ *relient respectivement un noeud* $\mathcal{N}$ *à un autre noeud* $\mathcal{N}$*.*
- *les fonctions de labélisation* $\mathcal{M}$ *relient* $\mathcal{TR} \longrightarrow \mathcal{T}$ *et associent à chaque transition* $tr = (n_s \times n_e)$ *le nombre de top d'horloge dont le système a besoin pour aller de l'état* n_s *où commence cette transition à l'état* n_e *où elle se termine .*

Comme le montre la figure 4.2, l'état symbolique $\mathbf{s}$ du modèle de l'architecture exécuté symboliquement inclut les paramètres suivants :
- (1) $\mathcal{SC}$: l'état du processeur (pipeline et caches instructions/données),
- (2) PC : le "path condition",
- (3) $countime$: le compteur interne.

De plus, durant l'exécution, un nouvel ensemble $\mathbf{SV}$ apparait, représentant les valeurs symboliques.

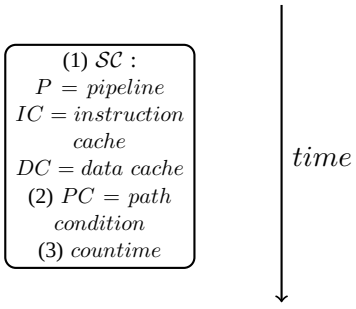


FIGURE 4.2: Etats et transitions symboliques

4.4.3 Algorithme d'exécution

Nous pouvons à présent introduire un algorithme qui construit *l'arbre symbolique temporisé* pour une séquence d'instructions binaires.

$Init\_state \leftarrow$ { (P = $\emptyset$, IC = DC = $\top$), PC = true, countime=0 } ;
$\mathbf{s}_0 \leftarrow \{Init\_state\}, current\_states \leftarrow \mathbf{s}_0$;
$\mathcal{G} \leftarrow \{Init\_state\}$;
while $current\_states \neq \emptyset$ **do** (Propagation)
  $\mathbf{s}_d \notin current\_states$ (retirer $\mathbf{s}_d$ de $current\_states$) ;
  $\mathcal{G}(\mathbf{s}_d)$ = succ$_1$ ($\mathbf{s}_d$) (construire l'arbre symbolique partant de $\mathbf{s}_d$ que nous obtenons suite à un pas d'exécution symboliques);
  $\mathcal{G} \leftarrow \mathcal{G}(\mathbf{s}_d) \cup \mathcal{G}$ (mise à jour de l'arbre symbolique $\mathcal{G}$) ;
  foreach $\mathbf{s}_d \in \mathcal{G}$ **do** (Fin de l'analyse ?)
    if ($\mathbf{s}_d \notin S_{fin\_analyse}$) **then**
      $current\_states \leftarrow \mathbf{s}_d$;

avec :
$\mathcal{G}$: arbre symbolique.
$\mathbf{s}_d$: état final de $\mathcal{G}$ vis-à-vis du point d'exécution courant et d $\in$ [0..h] indice de l'état final.
$S_{fin\_analyse}$: ensemble des états finaux de $\mathcal{G}$ vis-à-vis de l'analyse (états finaux générés par la dernière instruction du programme).

Remarque
Notons que la fonction $succ(\mathbf{s})$ génère non seulement les états successeurs ($\mathbf{s}_i^s$) mais également les transitions qui relient ces états successeurs à l'état courant ($\mathbf{s} \xrightarrow{\mathbf{t_i}} \mathbf{s}_i^s$)

Fonctionnement de l'algorithme
Cet algorithme formalise la construction de l'arbre symbolique. L'analyse débute par la création d'un état initial. Cet état a la particularité d'être le plus général possible :
- mémoires caches = inconnu ($\mathbf{IC} = \mathbf{DC} = \top$).
- pipeline = vide ($\mathbf{P} = empty$).

– path condition = true (**PC** $= true$).
– countime=0.

Au moment où l'analyse est lancée, le pipeline doit nécessairement être considéré comme vide [1].

Cet état initial est injecté conjointement aux instructions à analyser en entrée du modèle exécutable (modèle du processeur). A ce moment, ce modèle s'exécute symboliquement de manière à générer l'ensemble des états atteignables par le processeur en ce point d'exécution. Ces états intermédiaires ainsi que les transitions qui les relient sont par la suite sauvegardés respectivement dans les ensembles $current\_states$ et $\mathcal{G}$. Ces états intermédiaires seront au prochain pas d'exécution considérés comme des états initiaux.

L'analyse se poursuit jusqu'à ce que les derniers états générés par la dernière instruction du programme soient placés dans $current\_states$.

4.5 Illustration

Les premières parties de ce chapitre ont été consacrées à la présentation des grandes lignes de l'approche. A présent, nous allons montrer l'évolution de l'analyse sur un exemple composé de quelques instructions (voir figure 4.3).

Commençons par reprendre une partie du modèle du processeur. Cette partie introduite dans la section 3.5.2.1 représente le Fetcher (F). Pour rappel, cette unité est en liaison permanente avec la mémoire cache et permet donc de récupérer les instructions du programme à analyser.

Nous allons injecter, en entrée de cette partie du modèle, la séquence d'instructions de la figure 4.3. Chacune des instructions du programme effectue une opération précise :

```
1. add     r2,r2,r3
2. lwz     r9,(r31)
3. lwz     r8, (r1)
4. add     r7,r8,r9
5. stw     r7,10(r3)
6. lwz     r0,12(r9)
7. addi    r7,r0,1
8. stw     r7,10(r1)
```

FIGURE 4.3: Exemple d'une séquence d'instructions

– L'instruction 1 additionne le contenu du registre numéro 2 avec celui du registre numéro 3 et range le résultat dans le registre numéro 2.
– L'instruction 2 transfère le contenu de l'adresse mémoire présente dans le registre numéro 31 vers le registre numéro 9.
– L'instruction 3 transfère le contenu de l'adresse mémoire présente dans le registre numéro 1 vers le registre numéro 8.
– L'instruction 4 additionne le contenu du registre numéro 8 avec celui du registre numéro 9 et range le résultat dans le registre numéro 7
– L'instruction 5 place le contenu du registre numéro 7 à l'adresse mémoire correspondante au contenu du registre numéro $3 + 10$.

1. Supposons que le pipeline soit initialisé de la façon la plus générale possible qui est à l'état inconnu. Ceci ne serait pas pertinent du fait que les temps d'exécution calculés seraient obligatoirement des sur-approximations des temps réels. Chaque temps étant la somme du temps consommé par les instructions présentes au niveau du pipeline au moment où l'analyse débute, et du temps d'exécution du programme que l'on souhaite analysé.

– L'instruction 6 transfère le contenu de l'adresse mémoire correspondante au contenu du registre numéro $9 + 12$ vers le registre numéro 0.
– L'instruction 7 additionne 1 au contenu du registre numéro 0 et range le résultat dans le registre numéro 7.
– L'instruction 8 place le contenu du registre numéro 7 à l'adresse mémoire correspondante au contenu du registre numéro $1 + 10$.

Chaque instruction est identifiée grâce à un numéro représentant sa position dans le programme. Durant l'analyse, l'état d'une unité est représenté par l'instruction qu'elle exécute. Exemple, $IU1$ indique que l'unité entière exécute la première instruction du programme. De plus, d'autres notations sont introduites au niveau des temps de transfert entre les mémoires caches et le pipeline. Ainsi, à chaque temps (th, tm, trl) les lettres (d) ou (i) peuvent être associées, pour différencier les transferts de données des transferts d'instructions.

Rappelons que l'analyse débute par l'initialisation des mémoires caches à "inconnu" ($\mathbf{IC} = \mathbf{DC} = \top$) et que le pipeline est considéré comme vide ($\mathbf{P} = empty$) (aucune instruction n'est en cours d'exécution).

Première étage du pipeline : F

En se basant exclusivement sur la partie du modèle représentant le fetcher (voir figure 3.14), et en débutant l'analyse par l'état initial introduit ci-dessus, le résultat que nous obtenons est : à chaque top d'horloge deux identificateurs sont associés respectivement aux deux instructions que nous souhaitons récupérer de l'IC (modèle du fetcher : bloc A). Ensuite, l'adresse virtuelle de chaque instruction est convertie en adresse réelle (modèle du fetcher : bloc B) et Le F envoie une requête à l'IC (modèle du fetcher : bloc C). A ce stade, trois scénarios sont possibles : (1) les deux instructions sont dans le cache, (2) seule la première instruction est dans le cache, (3) les deux instructions ne sont pas dans le cache. Nous devons, par ailleurs, prendre en compte deux autres comportements propres à l'état courant de l'IC :

– l'IC est libre : le cache répond immédiatement à la requête.
– l'IC est occupée : le cache répond à la requête après avoir rechargé une de ses lignes.

Ceci implique que pour le premier pas d'exécution (voir figure 4.4) 6 scénarios sont possibles :

– (1) cache hit et le cache est libre, ceci prend $\mathbf{th}$(ii) cycles (état S1).
– (2) cache hit et le cache est occupé, ceci prend $\mathbf{th} + \mathbf{trl}$ (ii) cycles (état S2).
– (3) cache hit pour la première instruction, cache miss pour la seconde et le cache est libre, ceci prend $\mathbf{th}$(i) cycles (état S3).
– (4) cache hit pour la première instruction, cache miss pour la seconde et le cache est occupé, ceci prend $\mathbf{th} + \mathbf{trl}$(i) cycles (état S4).
– (5) cache miss et le cache est libre, ceci prend $\mathbf{tm}$(ii) cycles (état S5).
– (6) cache miss et le cache est occupé , ceci prend $\mathbf{tm} + \mathbf{trl}$ (ii) cycles (état S6).

Le graphe présenté dans la figure 4.4 montre le résultat obtenu après un pas d'exécution symbolique. A présent, détaillons une des branches du graphe. Prenons pour exemple le scénario numéro 1 (de l'état S_0 à S_1), après une exécution symbolique : le pipeline récupère les deux premières instructions du programme. Son état passe de ($\mathbf{P} = empty$) à ($\mathbf{P} = F1, 2$). Le cache d'instruction (IC) contient alors ces instructions et ainsi son état passe de ($\mathbf{IC} = \top$) à ($\mathbf{IC} = \top \wedge I1, 2$) [2]. Le compteur de temps $countime$ est incrémenté de th. Le cache de données ($\mathbf{DC}$) ainsi que le PC conservent respectivement les mêmes états initiaux.

Cette analyse se base sur les modèles proposés tout au long de la partie 3. En pratique, ces modèles sont plus volumineux car ils traitent deux instructions en parallèle et, de plus, il doivent respective-

2. Notons que, durant l'analyse, chaque instruction est repérée par un numéro indiquant sa position dans le programme. Nous précédons ce numéro par la lettre I uniquement au niveau du cache d'instruction (IC). Exemple : ($\mathbf{IC} = \top \wedge I1, 2$) signifie les instruction 1 et 2 sont dans l'**IC** .

ment gérer leurs problèmes de cohérences internes. Exemple : deux instructions de la même famille [3] ne peuvent pas être dispatchées simultanément [4]. En effet, le PowerPC 603 ne contient aucune unité redondante et toutes ses unités ne peuvent accueillir qu'une seule instruction par cycle.

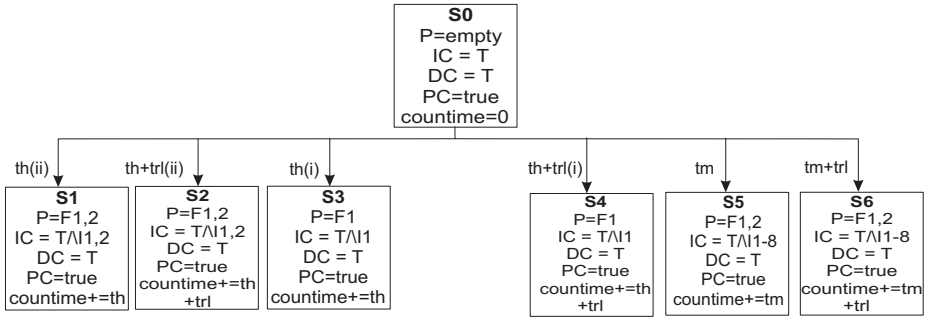


FIGURE 4.4: Premier pas d'exécution symbolique : (F)

Deuxième étage du pipeline : D

Durant cette étape (voir figure 4.5), les scénarios possibles imposés par le F demeurent les mêmes. Autrement dit, le F tente de récupérer deux nouvelles instructions du programme. Cependant, nous pouvons ignorer certains comportements du fait que l'historique d'exécution est pris en compte. Cet historique nous permet donc de contraindre le générateur de scénarios à éliminer les comportements infaisables. Exemple : prenons le scénario qui mène à l'état (S_5). Ce scénario représente le cas où les instructions 1 et 2 n'étaient pas présentes dans le cache. Une transaction mémoire a donc été déclanchée, ce qui a permis de récupérer une ligne de cache ($IC = \top \wedge I1 - 8$). Ainsi, au cours du deuxième pas d'exécution seul le scénario, indiquant que les instructions 3 et 4 sont dans le cache, sera généré. Néanmoins, ce scénario ne retranscrit, pour l'instant, que les comportements possibles du F. Pour obtenir une couverture totale des comportements possibles du pipeline nous devons aussi appliquer ce raisonnement au D.

Les comportements possibles du D, sans prise en compte de l'historique d'exécution ou de la nature des instructions traitées, sont :

- (1) "instruction1 dispatchée, instruction2 dispatchée",
- (2) "instruction1 non dispatchée, instruction2 non dispatchée",
- (3) "instruction1 dispatchée, instruction2 non dispatchée".

Le cas "instruction1 pas dispatchée ,instruction2 dispatchée" étant considéré comme infaisable du fait que les instructions sont dispatchées selon l'ordre fixé par le programme (in-order).
Concernant notre exemple, à l'instant où les instruction 1 et 2 sont envoyées vers leurs unités de traitement respectives, le pipeline est supposé être vide (toutes ses sous-composants sont vides). Ainsi, durant le deuxième pas d'exécution symbolique, seul le scénario (1) "instruction1 dispatchée, instruction2 dispatchée", sera considéré. Ce scénario est combiné avec ceux provoqués par le F. Ainsi, le résultat représente l'interaction des comportements des deux premiers étages du pipeline. Chaque exécution débute par un des états générés à l'étape précédente (premier pas d'exécution). Ces états de départ contiennent respectivement un historique d'exécution particulier. Le deuxième pas d'exécution représente donc une

3. Exemple deux instructions dont les opérandes sont des entiers.

4. Exception faite des processeurs contenant une unité entière multi-cycles ou ceux qui utilisent la SRU pour traiter des instructions d'additions d'entiers tels que le PowerPC 603(e).

multitude d'exécutions s'effectuant en parallèle, chacune introduisant un certain nombre de comportements. Tous ces comportements doivent être représentés par l'arbre d'exécution généré.

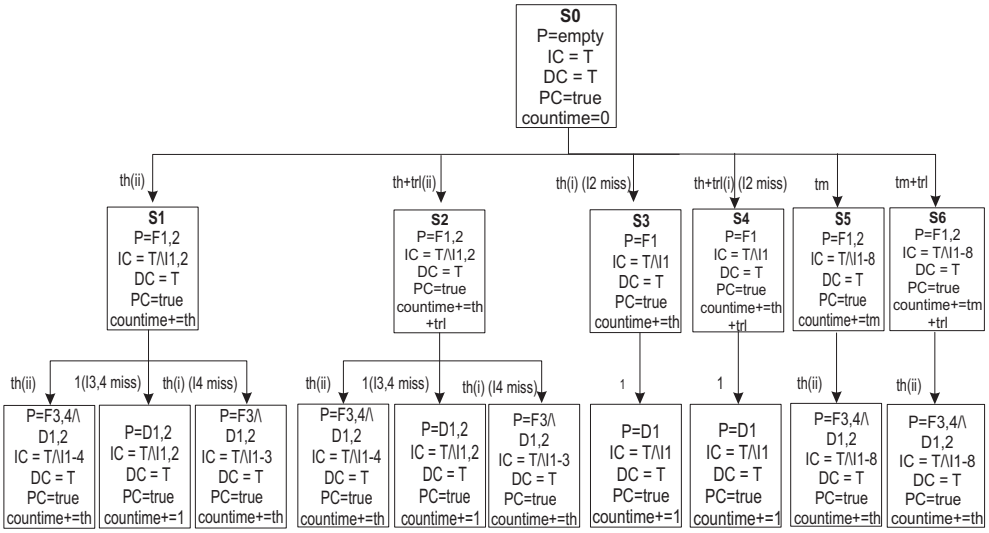


FIGURE 4.5: Deuxième pas d'exécution symbolique : (D)

Troisième étage du pipeline : EX

Reprenons, à présent, l'arbre généré suite au deuxième pas d'exécution symbolique. Pour une représentation claire des comportements du processeur, cet arbre est simplifié. Cette simplification se limite juste à retirer les scénarios d'exécution qui ont été introduits par l'état de l'IC en début d'analyse (sans trl). Autrement dit, tous les scénarios dont la première transition est labélisée avec un temps contenant une durée trl (Temps de rechargement d'une ligne de cache) sont retirés de l'arbre. Ces scénarios n'introduisent pas de nouveau comportements. Chacun d'entre eux a une trace similaire.

Ceci apparait clairement sur la figure 4.4, les états S1, S3 et S5 sont respectivement identiques aux états S2, S4 et S6 (S1=S2, S3=S4 et S5=S6).

Ainsi, la représentation de l'arbre symbolique est plus concise (voir fig. 4.6). Néanmoins, en début d'exécution s'il n'est pas possible de s'assurer que l'IC est libre (n'effectue aucune opération), les temps d'exécution obtenus (à la fin de l'analyse) devront être révisés à la hausse ($countime+ = trl$).

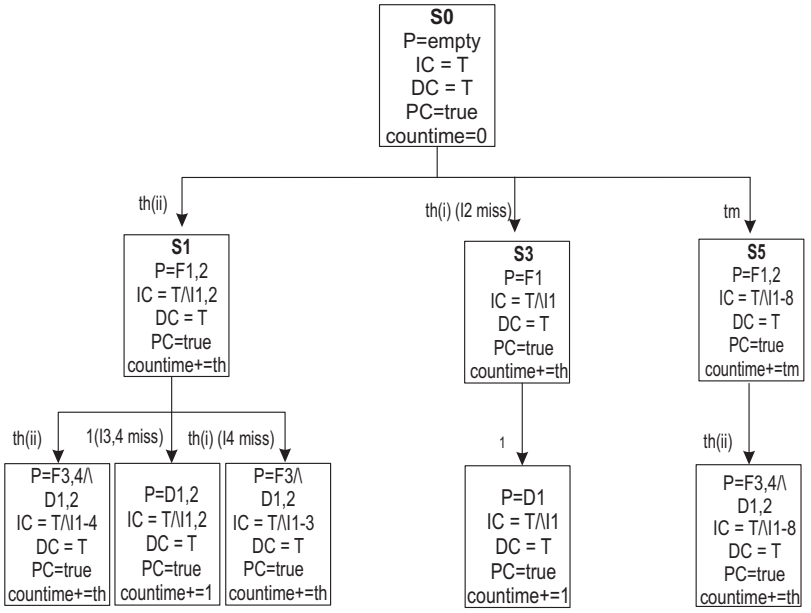


FIGURE 4.6: Deuxième pas d'exécution symbolique (D) ne contenant pas les scénarios introduits par l'état de l'IC en début d'analyse (sans trl)

Au troisième pas d'exécution symbolique, les deux premières instructions du programme ont atteint le D. A ce niveau, pour chaque instruction, deux scénarios sont possibles : (1) soit toutes ses opérandes sont prêtes, auquel cas le traitement de l'instruction commence dès qu'elle atteint son unité d'exécution. (2) Soit le D place l'instruction dans le registre temporaire (RS) de l'unité concernée de manière à ce que le traitement de l'instruction débute dès la disponibilité de toutes ses opérandes . Concernant la LSU, en plus de ces comportements, celle-ci en introduit d'autres. En effet, lorsque'une instruction atteint le deuxième étage du micro-pipeline de la LSU, celle-ci interagit avec la DC afin d'effectuer une opération de lecture ou écriture. A ce niveau, des scénarios semblables à ceux introduits par le F peuvent apparaitre :

– Donnée présente dans DC.
– Donnée absente de DC.

Concernant notre exemple de code, les opérandes des deux premières instructions sont disponibles. L'exécution commence donc : l'IU exécute la première instruction du programme alors que la seconde est placée dans le première étage du micro-pipeline de la LSU. Ainsi, durant ce cycle, la LSU va convertir l'adresse virtuelle de la seconde instruction en une adresse physique et la transfère vers le deuxième étage de son micro-pipeline.

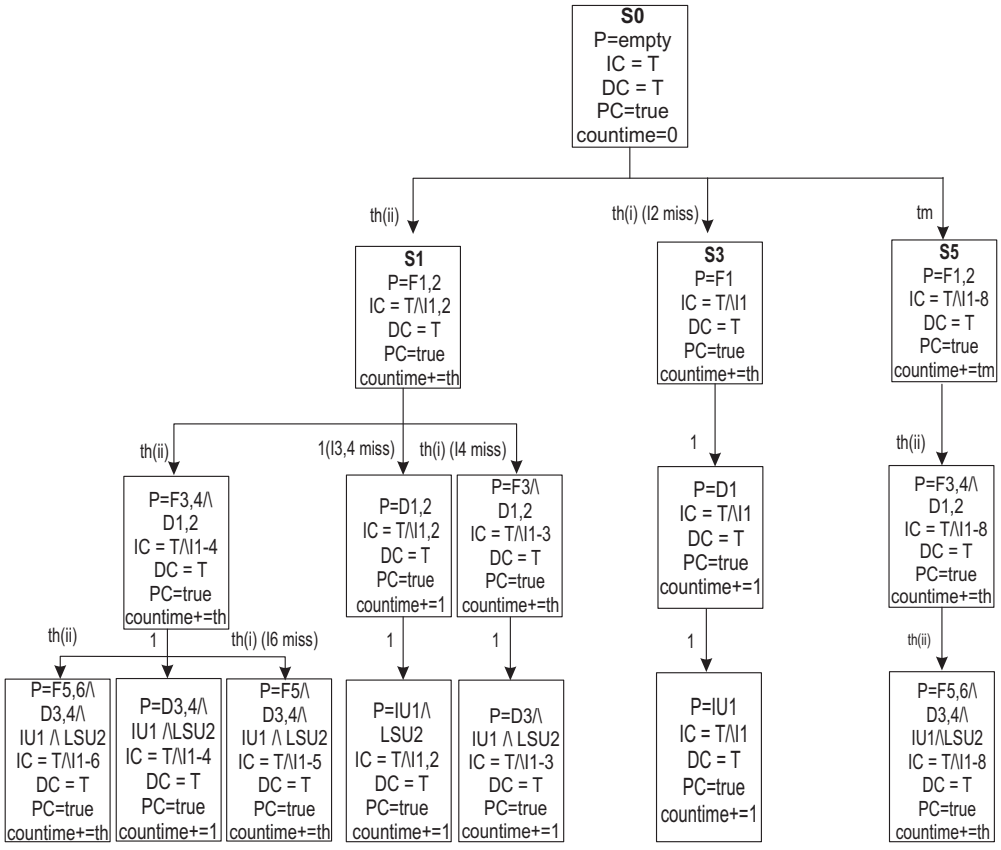


FIGURE 4.7: Troisième pas d'exécution symbolique (EX)

Quatrième étage du pipeline : CU

Le traitement de la première instruction du programme est terminé. La dernière étape consiste donc à s'assurer que cette exécution s'est effectuée correctement : aucune exception apparue au cours de l'exécution n'a été laissée sans traitement.

Par ailleurs, cette étape sert également à sauvegarder le résultat obtenu dans les registres appropriés (GPR ou FPR).

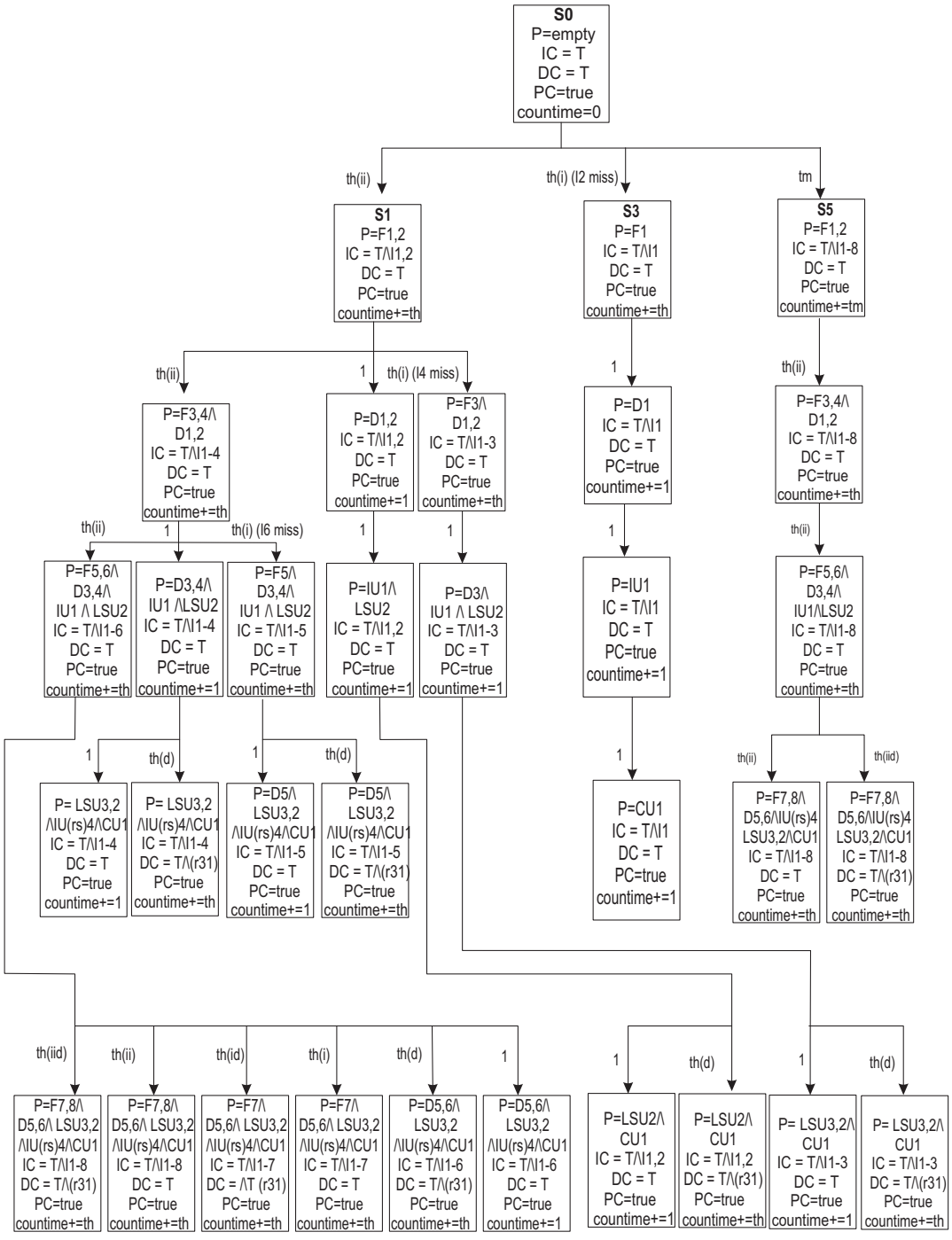


FIGURE 4.8: Quatrième pas d'exécution symbolique (CU)

Suite à l'exécution de la première instruction du programme, la CU transfère donc le résultat obtenu d'un registre intermédiaire vers le registre numéro 2.

4.6 Effets du pipeline

Dans cette partie nous allons nous intéresser à certains effets temporels apparus suite aux évolutions des processeurs. En effet, l'ajout de dispositifs, se matérialisant essentiellement par le pipeline, a introduit de l'indéterminisme. Cet indéterminisme se manifeste par des comportements du processeur dont la prise en compte demande une étude détaillée [5]. Cette étude est nécessaire, car négliger ces comportements serait préjudiciable aux performances de la méthode de calcul.

Ces effets temporels sont de deux types [22] : les effets négatifs et les effets positifs :

– Les effets négatifs représentent le gain de temps obtenu grâce à l'exécution "pipelinée" qui permet de débuter l'exécution d'une instruction avant la fin de l'exécution de l'instruction qui la précède. Ainsi, la prise en compte de ces effets durant le calcul des WCETs permet de produire des résultats plus précis.
– Les effets positifs sont, quant à eux, des temps qui viennent s'ajouter au temps d'exécution. Ces effets doivent être obligatoirement pris en compte lors du calcul du WCET. Leur négligence serait une source probable de sous-estimation des temps d'exécution.

L'analyse de ces comportements nous amène à nous intéresser aux différentes interactions entre d'une part, les unités du processeur et d'autre part, les instructions du programme à analyser. Ces interactions conduisent à une forte relation entre la rapidité de traitement d'une instruction [6] et l'histrorique d'exécution. Ainsi, l'exécution d'une instruction est liée aux exécutions des instructions précédentes. Ce lien apparait sous forme d'effets temporels que nous pouvons classer en : (1) effets à court terme, (2) effets à long terme, (3) effets introduits par la capacité du pipeline à faire du ré-ordonnancement d'instructions (anomalies temporelles).

La méthode IPET [10, 61] (Implicite Paths Enumeration Technique) est actuellement la seule méthode capable de prendre en compte certains effets intra-processeur. Cette méthode se base sur la programmation linéaire en nombres entiers (ILP). En effet, elle exprime le temps d'exécution du programme à partir des compteurs d'exécution des blocs de base (nœuds du graphe de flot de contrôle) et de leurs temps d'exécution individuels. Le recouvrement de deux blocs de base dans le pipeline produit un gain de temps qui est modélisé par un temps d'exécution négatif associé à l'arête liant les deux blocs. Le WCET est calculé en maximisant le temps d'exécution global tout en respectant un certain nombre de contraintes déduites : (1) de la structure du graphe de flot de contrôle (2) du flot de données (comme les bornes des boucles).

Ainsi, dans ce qui suit, nous nous baserons sur cette méthode de façon à montrer, d'une part, que notre méthode prend pas en compte tous les effets temporelles (IPET ne considère précisément que les effets à court terme) et, d'autre part, que concernant les effets pris en compte par IPET, notre approche donne les mêmes résultats sans besoins d'analyses supplémentaires.

4.6.1 Processeur cible

Afin d'illustrer nos propos, concernant les effets du pipeline, nous avons choisi de traiter les différents exemples présentés dans cette partie selon la logique d'exécution dictée par le processseur PowerPC603e. Ce processeur est très similaire au PowerPC 603. Cependant, il dispose d'un avantage non négligeable : son unité de registre système (SRU) contient un additionneur et peut donc traiter des instructions d'addition lorsque l'unité entière est occupée.

Comme le montre la figure 4.9 le PowerPC 603e est composé de différentes unités.

5. En fonction du type de pipeline utilisé, nous devons soit définir des conditions sous lesquelles ces comportements n'ont pas un impact catastrophique sur les temps d'exécution estimés ou sinon analyser précisement chaque cas.
6. Plus généralement d'un bloc d'instructions

- unité d'instruction (IU) : cette unité est composée (1) d'un fetcher qui récupère les instructions de la mémoire cache et qui les stocke dans la pile d'instructions (instruction queue), (2) d'une unité qui traite les instructions de branchement (BPU).
- dispatcher : cette unité récupère des instructions de la pile d'instruction, les décode puis les envoie à l'unité d'exécution appropriée. Par la suite nous supposerons que cette unité n'est pas bloquante. Ce qui signifie, que le dispatcher est capable de réordonnancer une séquence d'instructions en fonction des disponibilités des unités de traitement.
- Les unités d'exécution : Chaque unité d'exécution traite les instructions qui lui parviennent durant un ou plusieurs cycles, puis écrit le résultat de l'opération dans un registre intermédiaire. Ensuite, elle notifie à l'unité d'achèvement que l'exécution de l'instruction est terminée. Parmi ces unités celles qui nous intéressent sont : l'unité entière (IU), l'unité de regitre système (SRU) et l'unité de lecture et d'écriture mémoire (LSU).
- unité d'achèvement (CU) : Une fois l'exécution terminée, l'unité d'achèvement est alertée. Cette unité sauvegarde les résultats obtenus par les unités d'exécution.
- mémoire cache : ce modèle contient deux mémoires caches, une de données et une autre d'instructions. Le cache de donnée (DC) est en liaison avec l'unité de lecture et d'écriture mémoire alors que le cache d'instructions (IC) est relié au fetcher. Ces deux mémoires ont des structures ainsi que des fonctionnements similaires.

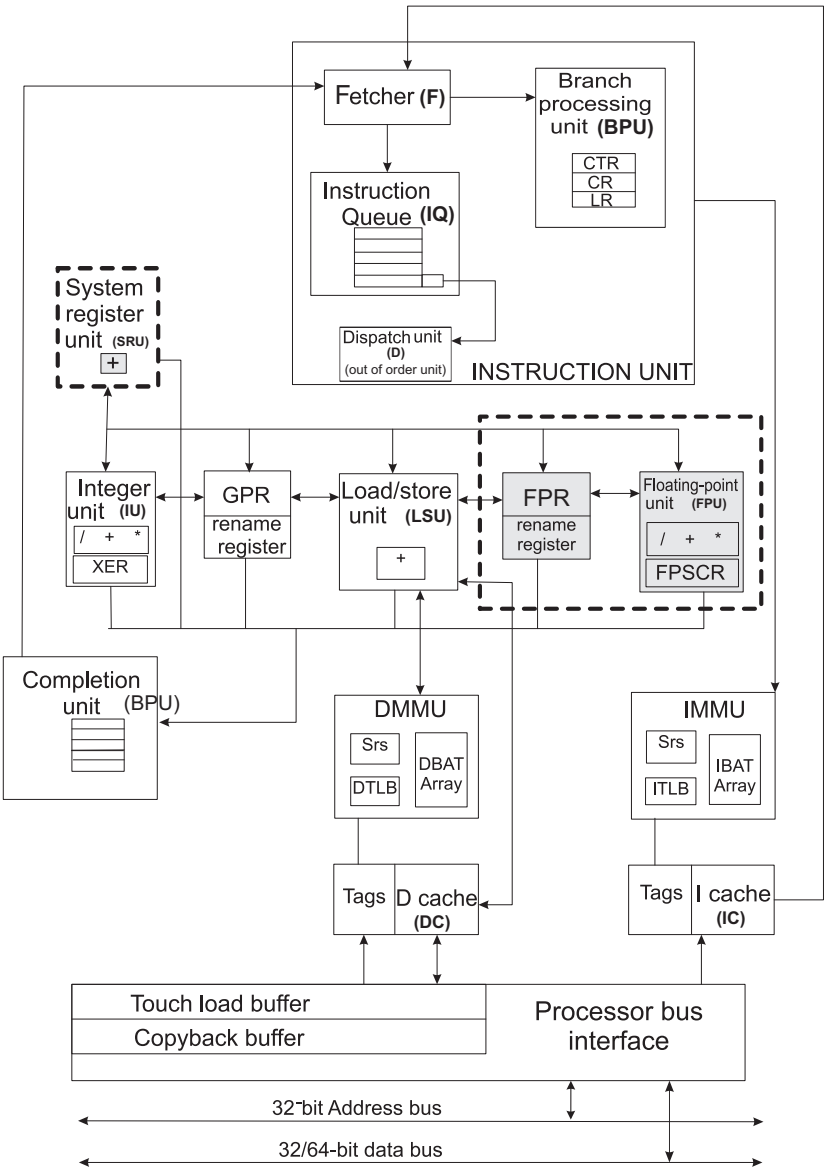


FIGURE 4.9: Processeur PowerPC 603e (*ne contient pas d'unité flottante*)

4.6.2 Effets à court terme

Ces effets apparaissent au niveau du pipeline et se définissent comme suit : si aucun conflit n'est détecté, le temps d'exécution de deux instructions (blocs de base) qui se succèdent est inférieur à la somme de leurs temps d'exécution. Cette différence de temps d'exécution est due au paraléllisme introduit par le pipeline. Cet effet est une conséquence directe de l'exécution pipelinée. Le gain de temps étant dû à l'entrelacement des exécutions des instructions.

Pour illustrer cet effet prenons l'exemple le plus simple qui est l'exécution de deux instructions [7]. Comme le montre la figure 4.10, la somme des temps d'exécution des deux instructions (sans pipeline) est égale à $t_a + t_b = 8$ cycles d'horloge. En présence de pipeline, l'entrelacement des exécutions est pris

7. le même raisonnement est appliqué lorsqu'il s'agit de deux blocs d'instructions.

en compte. Le calcul devient $t_{a-premierétagevide} + t_b = 1 + 4 = 5$. Ainsi, l'exécution de l'instruction (du bloc) B commence quand l'instruction (bloc) A libère le premier étage du pipeline. Ainsi, le gain de temps $\delta t_{ab} = 5 - 8 = -3$ est appelé effet à court terme temporel négatif.

Les méthodes d'analyse statique actuelles telle que le modèle IPET présenté dans la figure 4.10, prennent en compte ce type d'exécution. Ces méthodes tendent à traiter la recherche du WCET comme un problème de programmation linéaire. Ce problème consiste à en maximiser le temps d'exécution sous certaines contraintes relatives aux nombres d'exécutions des blocs de base.

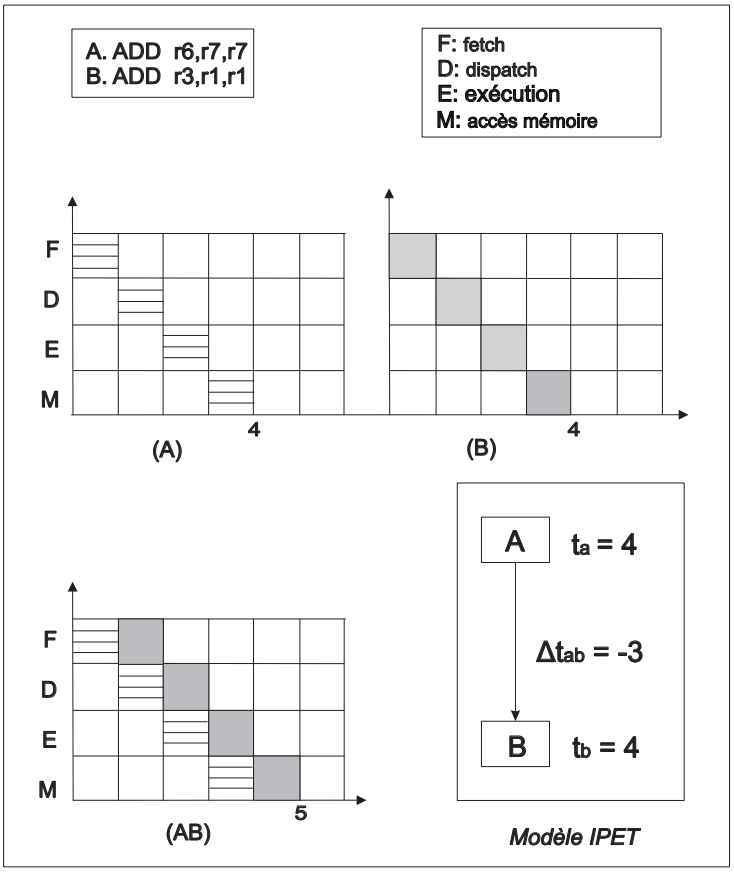


FIGURE 4.10: Pipeline et effet à court terme

4.6.3 Effets à long terme

Le temps d'exécution d'une instruction (bloc de base) peut dans certains cas être fortement lié à l'exécution d'une instruction (bloc) distante. Cela se produit généralement lorsqu'un conflit sur une ressource apparait entre ces deux instructions (blocs). Cet effet traduit donc l'impact qu'ont les exécutions antérieures sur une exécution courante. Engblom [22, 19, 21] est le premier à avoir découvert cet effet.

Prenons l'exemple présenté dans la figure 4.11. Dans cet exemple nous pouvons voir l'exécution de trois instructions (A,B,C) et les influences de chaque exécution sur les autres.

les temps d'exécutions des deux blocs (AB) et (BC) sont correctement calculés suivant le modèles IPET. La prise en compte des effets à court terme est donc vérifiée. A présent, si nous appliquons la méthode IPET afin de calculer le temps d'exécution des trois instructions (blocs), le temps d'exécution est égal à $(t_a + t_b + \delta t_{ab}) + t_c + \delta t_{bc} = (8 + 4 + -3) + 9 + -3 = 15$. Ce temps, bien qu'étant une

sur-estimation fiable du temps d'exécution, est très éloigné du temps d'exécution réel, qui est égale à 11. Cette différence entre le temps réel et le temps calculé est due aux "effets à long terme".

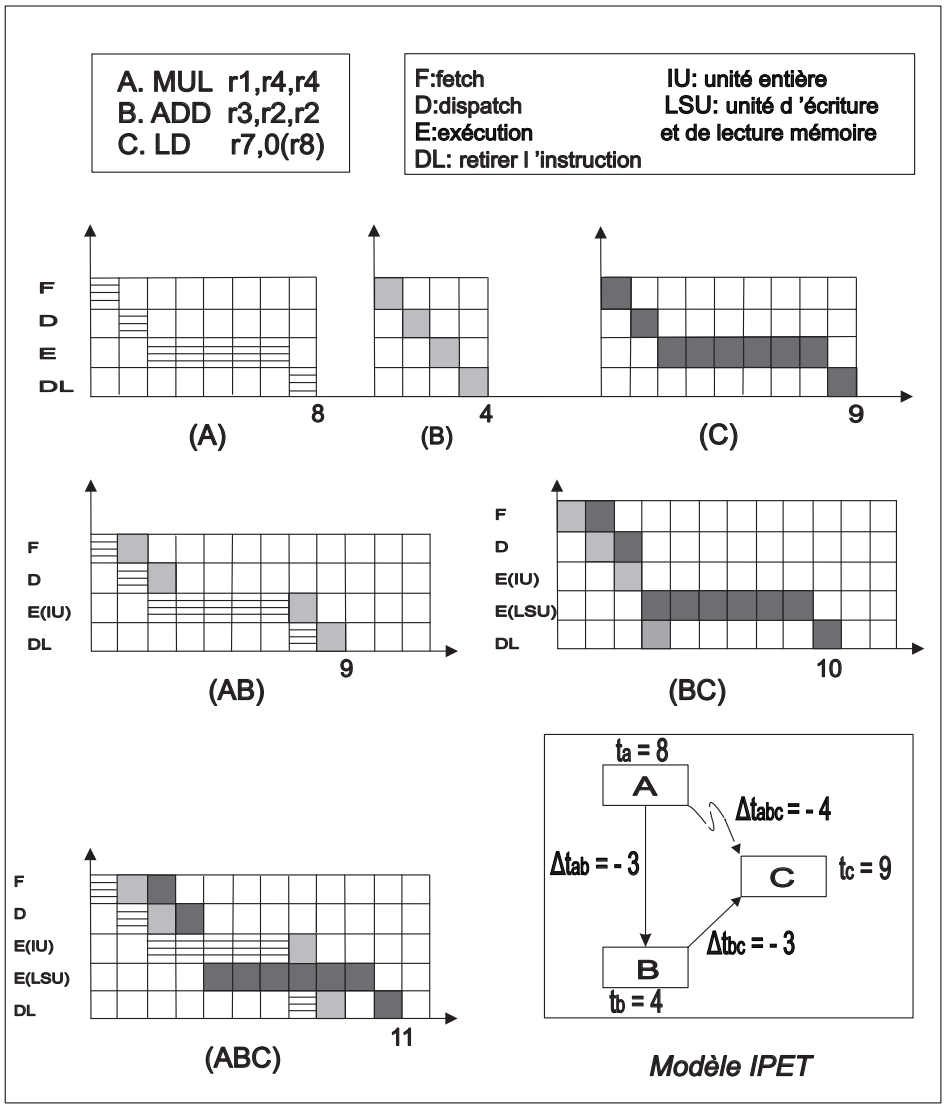


FIGURE 4.11: Pipeline et effet à long terme

4.6.4 Anomalies temporelles : ré-ordonnancement d'instructions

Ces anomalies temporelles constituent une source supplémentaire d'indéterminisme. En effet, elles apparaissent lorsqu'un conflit sur une ressource se produit. Lundqvist and Stenstrom [36, 40] sont les premiers à les avoir identifié suite à l'analyse du comportement d'architectures contenant des unités de traitement pouvant effectuer un ré-ordonnancement d'instructions (out of order execution). Ce ré-ordonnancement apparait lorsqu'une unité d'exécution effectue des traitements sans respecter l'ordre dans lequel sont rangées les instructions au niveau du programme. Ainsi, une instruction (i) peut-être traitée avant l'instruction (i-1). Cette décision est prise dynamiquement par l'unité de traitement dans le but d'améliorer la vitesse d'exécution globale du pipeline. Les unités du pipeline peuvent donc être

classées en deux groupes : (1) unités respectant l'ordre d'exécution imposé par le programme, (2) unités pouvant réordonnancer les instructions dynamiquement en fonction de l'environnement d'exécution.

Cette deuxième famille d'unité introduit un nouveau type d'indéterminisme durant l'exécution appelé "anomalies temporelles". Pour démontrer l'impact de ces anomalies sur le temps d'exécution nous donnons un exemple. Supposons dans un premier temps que le processeur présenté dans la figure 4.9 soit capable de réordonnencer dynamiquement des instructions.

A présent, intéressons nous aux unités de traitement. Considérons les unités, entière (IU) et de registre système (SRU), comme étant des unités acceptant des instructions dans le désordre (par rapport à l'ordre fixé par le programme). L'unité de lecture et d'écriture mémoire (LSU) est considérée comme traitant les instructions dans l'ordre afin de préserver la cohérence de la mémoire de données (DC).

Nous proposons dans la figure 4.12 une séquence de code montrant que dans certains cas un cache miss se révèle moins pénalisant, en termes de temps d'exécution, qu'un cache hit. Dans le but de mettre en évidence cette anomalie temporelle nous n' allons exposer que la partie de l'exécution effectuée par les unités de traitement (EX). Ainsi, nous faisons abstraction des autres étages du pipeline (Fetch (F), Dispatch (D), accès mémoire (M)).

Afin d'expliquer l'exécution de la séquence de code proposée nous allons poser trois hypothèses :

- à l'instant où la première instruction du programme atteint l'unité de lecture et d'écriture mémoire (LSU), l'unité entière (IU) est occupée (ceci est représenté par un rectangle gris sur la figure 4.12).
- les instructions (A,B,C,D,E,F) sont toutes présentes dans la mémoire cache d'instructions (IC).
- Les instants où les instructions atteignent leurs unités de traitement respectives sont ceux représentés par des flèches dans la figure 4.12.

A présent, détaillons les deux scénarios d'exécution possibles.

Donnée présente dans le cache :
Durant le cycle 1, l'instruction (A) atteint son unité d'exécution. Cette dernière a besoin de deux cycles pour charger la donnée dans le registre $r1$ (cache hit). Au cycle 3, l'instruction (A) a donc terminé son exécution. Par conséquent, l'instruction (B) peut être exécutée. Le processeur teste alors les deux unités capables de traiter une addition d'entiers (IU et SRU). L'unité entière étant occupée, le processeur place l'instruction (B) dans la SRU. Au cycle 4, le processeur exécute l'instruction (C). L'instruction (D) qui était en attente du résultat fourni par l'instruction (C), peut s'exécuter durant le cinquième cycle. Au cycle 6, commence l'exécution de l'instruction (E). Elle est suivie par celle de l'instruction (F). Cette dernière termine son exécution au quinzième cycle.
Quand la donnée requise par l'instruction (A) est présente dans la mémoire cache, le traitement de cette séquence d'instructions prend 15 cycles.

Donnée non présente dans le cache :
Durant le cycle 1, l'instruction (A) atteint son unité d'exécution. Cette dernière nécessite sept cycles pour charger la donnée dans le registre $r1$ (cache miss). Au cycle 3, l'instruction (A) n'a donc toujours pas terminé son exécution. Par conséquent, l'instruction (B) ne peut être exécutée (ses opérandes ne sont pas toutes disponibles). Le processeur place donc l'instruction (C) dans la SRU. Au cycle 4, le processeur exécute l'instruction (D). L'instruction (E) qui était en attente du résultat fourni par l'instruction (D), peut débuter son exécution au cycle 5. Au cycle 7, les opérandes de l'instruction (B) sont toutes disponibles. Ainsi, le processeur l'exécute durant ce cycle. Au cycle 10 commence l'exécution de l'instruction (F).Cette dernière termine son exécution au quatorzième cycle.

Cet exemple montre que l'exécution de la séquence d'instructions est plus rapide lorsqu'un cache miss se produit ($T_{hit} = 15$ et $T_{miss} = 14$). Cette différence de temps d'exécution, en contradiction avec nos attentes, apparait en présence d'anomalies temporelles.

Ainsi, ce résultat prouve que les méthodes d'analyses qui considèrent les accès mémoires indéterminés comme étant des caches miss sont, dans certain cas, inadaptées.

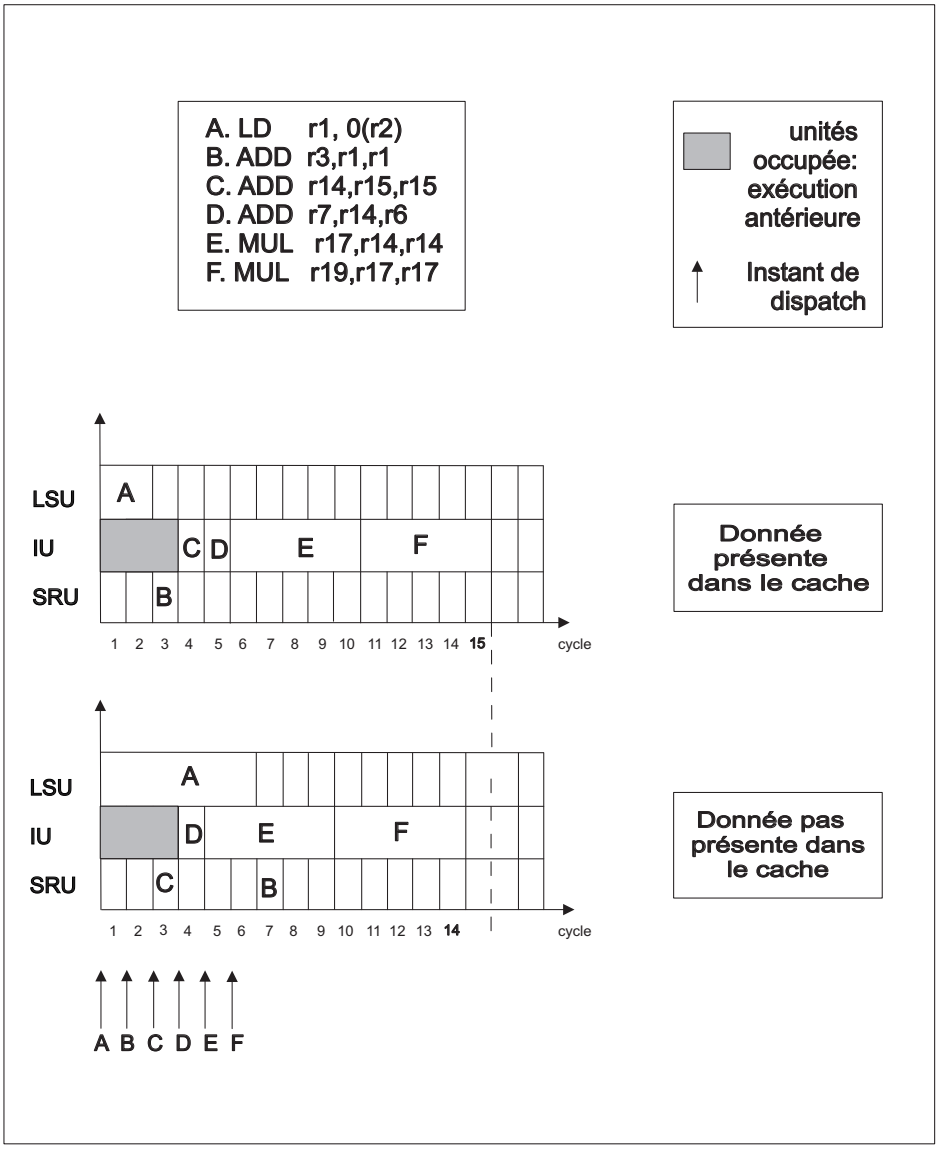


FIGURE 4.12: Anomalies temporelles

4.7 Prise en compte des effets temporels

Dans cette partie nous allons voir comment notre approche permet de prendre en considération ces différents effets du pipeline.

4.7.1 Prise en compte des effets à court terme

Notre approche consiste à générer systématiquement tous les comportements possibles du processeur. Ainsi, les effets à court terme sont logiquement pris en compte par l'analyse.

Reprenons l'exemple exposé dans la figure 4.10 ((voir partie 4.6.2) :

```
1. ADD   r6 , r7 , r7
2. ADD   r3 , r1 , r1
```

Notons que, durant l'analyse, chaque instruction n'est plus identifiée par une lettre mais par un numéro représentant sa position dans le programme.

Durant l'exécution proposée (voir figure 4.10), nous avions supposé que les instructions 1 et 2 (précédemment A et B) étaient présentes dans le cache d'instructions au début de l'exécution.

Comme le montre la figure 4.13, nous débutons l'analyse en prenant les mêmes hypothèses. Le cache d'instruction est donc initialisé à $IC = T \wedge I1, 2$. Durant le premier cycle, l'instruction 1 est récupérée de la mémoire cache. Durant le second cycle l'instruction 2 est à son tour récupérée du cache d'instruction pendant que l'instruction 1 est décodée. Le troisième cycle sert à décoder l'instruction 2 et à exécuter l'instruction 1. Cette dernière termine son exécution au quatrième cycle alors que l'instruction 2 s'exécute. Enfin, au cinquième cycle l'instruction finit de s'exécuter ce qui marque la fin de l'analyse.

Le temps d'exécution obtenu (5 cycles) est donc égal au temps d'exécution réel. Ce résultat prouve que notre approche prend systématiquement en compte les effets à court terme.

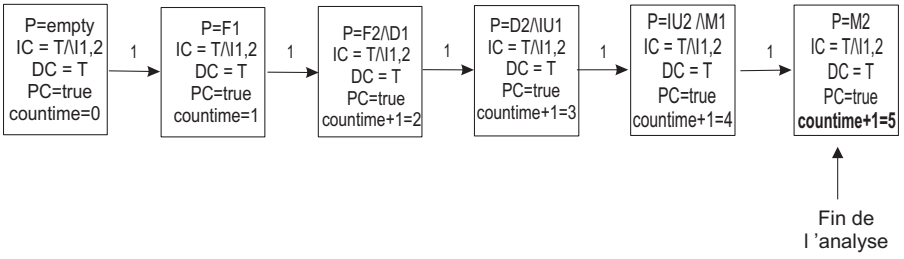


FIGURE 4.13: Prise en compte des effets à court terme

4.7.2 Prise en compte des effets à long terme

Les effets à long terme sont ceux qui font apparaitre l'impact que peut avoir un historique sur une exécution courante. Dans le but de démontrer que notre méthode de calcul prend bien en compte ces effets, nous allons reprendre l'exemple présenté dans la figure 4.11 (voir partie 4.6.3) :

```
1. MUL   r1 , r4 , r4
2. ADD   r3 , r2 , r2
3. LD    r7 , 0 ( r8 )
```

Durant l'exécution de cet exemple, nous avions supposé que : (1) les instructions (A,B,C) sont dans le cache d'instructions et (2) la donnée chargée par l'instruction C n'est pas dans le cache de données. Notre analyse ne s'intéressera qu'à ce cas.

L'analyse débute donc par l'initialisation des caches d'instructions et de données respectivement comme suit : $IC = T \wedge I1 - 3$, $DC = T \wedge \neg(r8)$. A partir de cet état une trace d'exécution est générée. Comme le montre la figure 4.14, à la fin du cinquième cycle, l'instruction 3 (LD) tente de récupérer une donnée de la mémoire cache. Cette donnée étant initialement supposée absente du cache de donnée, une transaction mémoire est lancée. Ainsi, durant le dixième cycle, la donnée arrive dans le cache ce qui permet à l'instruction 3 d'achever son exécution au cycle suivant. La trace d'exécution générée donne un temps d'exécution égal au temps d'exécution réel (11 cycles).

A travers cet exemple il apparait clairement que notre exécution symbolique permet de construire des traces d'exécution représentant exactement l'évolution réelle du processeur. Notre méthode prend donc en considération les effets à long terme sans besoin d'étude au préalable visant à les identifier.

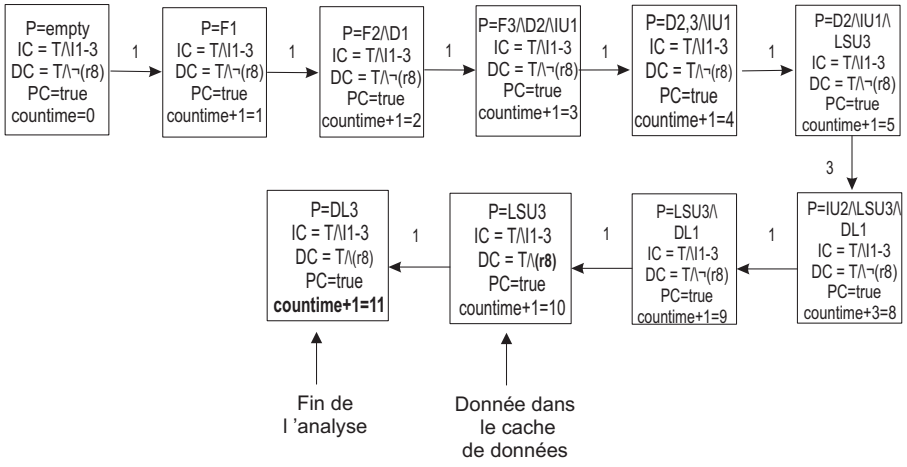


FIGURE 4.14: Prise en compte des effets à long terme

4.7.3 Prise en compte des anomalies temporelles

Dans le but de prouver que notre approche prend en considération les anomalies temporelles, intéressons nous de nouveau à l'exemple présenté dans la figure 4.12 (voir partie 4.6.4).

```
1. LD   r1 , 0(r2)
2. ADD  r3 , r1 , r1
3. ADD  r14 , r15 , r15
4. ADD  r7 , r14 , r6
5. MUL  r17 , r14 , r14
6. MUL  r19 , r17 , r17
```

Avant d'exécuter cet exemple, nous avions émis deux hypothèses : (1) les instructions (B,C,D,E,F) ont

été récupérées du cache d'instructions (F) et (2) l'instruction A est sur le point d'être envoyée vers son unité d'exécution (D).

Nous allons donc débuter notre analyse avec les mêmes hypothèses : (1) $P = F2 - 6$ et (2) $D1$. Notons que ces conditions initiales impliquent évidemment que les instructions (1,2,3,4,5,6) sont présentes dans le cache d'instructions ($IC = T \wedge I1 - 6$).

Comme nous pouvons le voir dans la figure 4.15, durant le premier cycle, l'instruction 1 est exécutée. Cette instruction doit ranger, dans le registre $r1$, une donnée dont l'adresse est placée dans le registre $r2$. Une des opérandes de l'instruction 2 est la donnée stockée dans le registre $r1$ (dépendance de donnée). Ainsi, tant que la donnée n'a pas été transférée vers le registre $r1$, l'instruction 2 ne peut être traitée. Ce même conflit existe aussi entre :

- les instructions 3 et 4 : l'instruction 4 attend la fin de l'exécution de l'instruction 3 ($r14$ en commun)
- les instructions 4 et 5 : l'instruction 5 attend la fin de l'exécution de l'instruction 4 ($r14$ en commun)
- les instructions 5 et 6 : l'instruction 5 attend la fin de l'exécution de l'instruction 6 ($r17$ en commun)

Durant le cycle 2, l'instruction 1 recherche la donnée (dont l'adresse est dans $r2$) dans le cache de donnée. Cette mémoire cache étant initialisée à l'état inconnu ($DC = T$), la méthode d'analyse génère deux traces d'exécution : (1) La donnée est dans le cache (th(d)), ainsi l'instruction 2 sera exécutée au prochain cycle. (2) La donnée n'est pas dans le cache (tm(d)), l'instruction 2 doit attendre le temps d'une transaction mémoire avant de s'exécuter. Néanmoins, grâce à la capacité du dispatcher à réordonnancer les instructions, les exécutions suivant ces deux traces peuvent se poursuivre sans blocage. Ainsi, suivant la première trace d'exécution (th(d)), le processeur exécute les instructions séquentiellement à cause des dépendances de données cités ci-dessus. Alors que dans le cas de la deuxième trace d'exécution (tm(d)), le processeur :

- poursuit l'exécution de l'instruction 1 en parallèle des exécutions des instructions 3,4,5 (cycles 2 à 6).
- poursuit l'exécution de l'instruction 5 en parallèle du traitement de l'instruction 2 (cycle 7).

Cette exécution simultannée de plusieurs instructions permet au processeur d'exécuter la séquence d'instructions plus rapidement que lorsqu'un cache hit se produit ($T_{hit} = 15$ et $T_{miss} = 14$).

A travers l'analyse de cet exemple, nous avons démontrer que notre méthode de calcul, du fait de sa capacité à générer toutes les traces d'exécution possibles, permet de prendre en compte tous les effets (aussi inattendus soit-ils) internes d'un processeur.

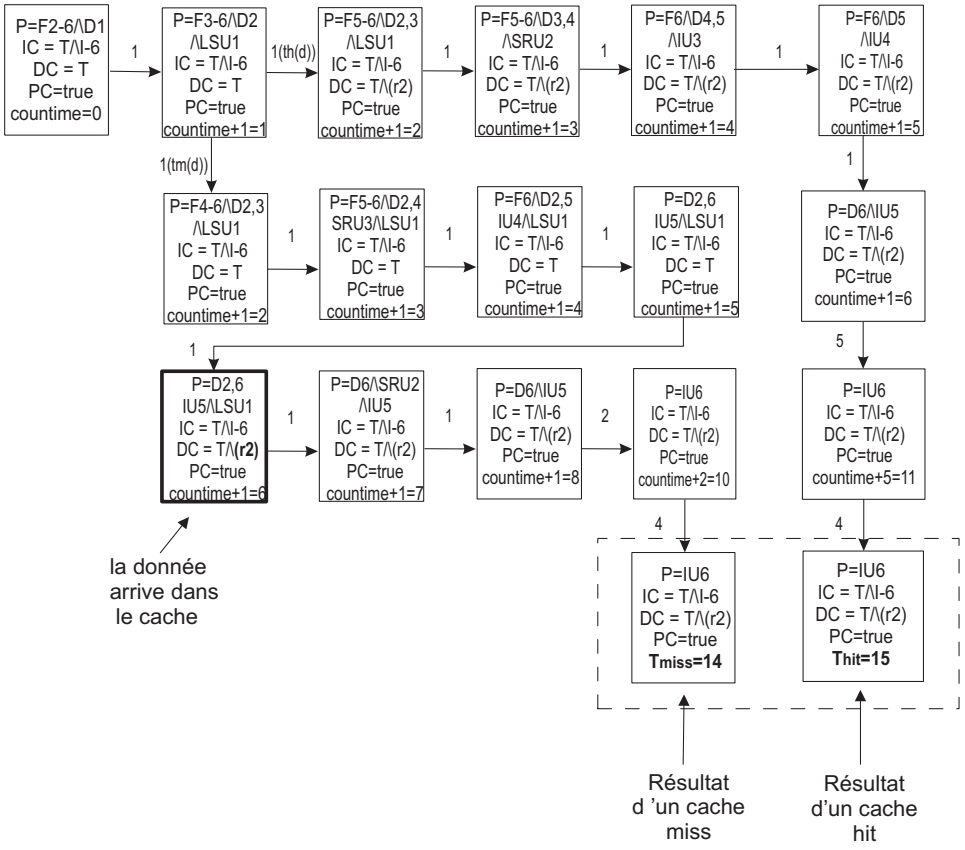


FIGURE 4.15: Prise en compte des anomalies temporelles

4.8 Impact de l'exécution symbolique sur les performances de la méthode

L'exécution symbolique comparée aux analyses statiques reste une approche assez facile à mettre en œuvre. D'une part, elle produit des résultats fiables du fait qu'elle assure une couverture totale des traces d'exécution. Cette méthode est donc une méthode formelle au même titre que les méthodes statiques [48]. D'autre part, cette méthode d'analyse palie à certaines lacunes propres aux méthodes statiques comme la prise en compte de traces d'exécution infaisables. Cette prise en compte a un impact direct et parfois considérable sur la précision des résultats. Concernant le calcul des pires temps d'exécution, cette prise en compte peut dans certains cas mener à des aberrations comme : la mise au point d'un système temps réel surdimensionné dont les performances seraient limitées [8]. Ces résultats sur-approximés et trop éloignés des valeurs réelles sont également obtenus parce que les méthodes statiques ont du mal à considérer plusieurs effets propres à l'architecture matérielle (voir partie 2.2).

En effet, les méthodes basées sur l'abstraction, comme AbsInt, sont des méthodes dont le principe consiste à trouver une formulation qui correspond à plusieurs comportements. Ces méthodes tentent donc d'identifier **implicitement** l'ensemble des traces d'exécution. Tandis que, l'exécution symbolique permet une énumération de tous les états atteignables. Cette méthode identifie donc **explicitement** toutes les traces d'exécution possibles d'un programme.

La figure 4.16 représente une séquence d'instructions utilisée pour montrer la différence entre les mé-

8. Exemple : les temps alloués aux tâches seraient largement au dessus de leurs besoins respectifs.

thodes explicites et implicites. Ainsi, au lieu de comparer l'exécution symbolique à plusieurs méthodes formelles existantes, nous avons choisi de faire un parallèle entre calcul implicite et explicite des traces d'exécution de manière à mettre en valeur la pertinence de l'exécution symbolique.

```
#define M 4
#define N 6
int main()
{
    int i, j;
    int mat[M][N];
    for (i=0; i<M; i++)
    {
        mat[i][0]=1;
        for (j=0; j<N; j++)
            mat[i][j]=i+j;
    }
}
```

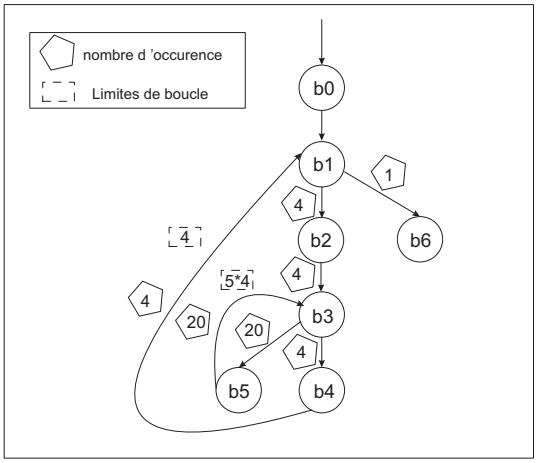


FIGURE 4.16: Exemple de code et du CFG associé

Calcul explicite des traces d'exécution
Une trace d'exécution explicite est une liste ordonnée de blocs de base. Dans notre exemple la seule trace d'exécution possible est définie par la séquence de blocs :
$(b1 - b2 - (b3 - b4)_{\times 5} - b3 - b5)_{\times 4} - b1 - b6$

Les analyses explicites [26, 38, 50] explorent toutes les traces d'exécution mais cela peut se révéler couteux si le programme a plusieurs traces d'exécution possibles.

Calcul implicite des traces d'exécution
Certaines techniques simplifient l'exploration des traces. Ces techniques se basent sur une représentation implicite [61] des traces d'exécution. En effet, une méthode implicite considère le programme comme une liste de blocs de base associés respectivement à des compteurs d'exécution. Ainsi, la trace implicite déduite de l'exemple est :
$(b1_{\times 5}, b2_{\times 4}, b3_{\times 24}, b4_{\times 20}, b5_{\times 4}, b6_{\times 1})$
De la même manière qu'un état abstrait correspond à un ensemble d'états concrets, une trace d'exécution implicite représente plusieurs traces d'exécution explicites possibles. Cependant, dans le cas général, beaucoup d'entre elles sont infaisables. Exemple, la trace implicite ci-dessus contient la trace explicite suivante :
$(b1 - b2 - b3 - b4 - b3 - b5)_{\times 3} - b1 - b2 - (b3 - b4)_{\times 17} - b3 - b5 - b1 - b6$
qui n'est pas cohérente avec la sémantique du programme.

Ainsi, utiliser une méthode qui considère un programme comme un ensemble de traces d'exécution implicites, pour calculer des valeurs précises de temps d'exécution, implique la prise en compte :

- des contraintes structurelles du CFG,
- des contraintes sur les boucles,
- des annotations de l'utilisateur.

Ceci dans le but de réduire au maximum le nombre de traces infaisables.

Cette approche reste donc valable si elle est utilisée uniquement pour calculer un temps d'exécution global en additionnant les temps d'exécution des différents blocs. Les résultats obtenus sont certes des sur-approximations néanmoins la marge d'erreur demeure acceptable. Cependant, si des mécanismes

basés sur l'historique d'exécution doivent être considérés (exemple prédiction de branchement), les méthodes implicites ne seront plus adéquates car elles donneront des résultats qui seront trop éloignés de la réalité.

Notre extension de l'exécution symbolique a donc pour principal avantage de fournir des résultats exacts indépendamment de la complexité de l'architecture analysée. Cependant, ce point fort peut se révéler être un frein à l'analyse si certaines précautions ne sont pas prises. En effet, les calculs exacts impliquent que toutes les traces d'exécution soient considérées séparément (explicitement). Ceci, rend l'efficacité de l'analyse intimement liée au nombre des traces d'exécution. Plus ce nombre augmentera, plus les besoins de l'analyse en mémoire vont s'accroitre. Par conséquent, l'utilisation de cette méthode implique de prendre en considération une possible explosion combinatoire. D'où l'intérêt d'implémenter des méthodes de fusionnement efficaces pouvant contenir le nombre d'états générés sans pour autant introduire une imprécision importante sur les temps d'exécution.

4.9 Conclusion

Notre extension, au même titre que l'exécution symbolique, permet d'obtenir une couverture totale des traces d'exécution. Nous avons ainsi démontré que, sans besoin d'études supplémentaires, tous les effets intra-processeur sont naturellement pris en compte par l'analyse.

Cependant, ce dépliage automatique des scénarios d'exécution fait que le nombre de comportements à gérer durant l'exécution explose. C'est pourquoi la deuxième partie de l'analyse, présentée dans le chapitre suivant, est consacrée à la fusion d'états. Notre principal objectif est de diminuer le nombre de traces d'exécution sans que cela ne nuise à la précision des résultats fournis.

CHAPITRE 5

Politique de fusionnement

5.1 Introduction

Notre extension de l'exécution symbolique permet de produire :
- tous les comportements du programme que nous souhaitons analyser.
- tous les comportements possibles du processeur lorsqu'il exécute ce programme.

L'explication est que cette exécution suit le principe du dépliage systématique et donc, à chaque point d'exécution tous les scénarios possibles sont générés. Cet avantage, apporté par ce type d'exécution, peut, s'il n'est pas contrôlé, se révéler être un élément bloquant mieux connu sous le nom d'explosion combinatoire. Ainsi, les performances de l'algorithme de fusionnement ont un impact considérable sur l'efficacité de l'approche. En effet, une méthode de fusionnement grossière mènerait inéluctablement vers un résultat trop imprécis. Cet algorithme doit durant l'exécution vérifier deux conditions :
- Eviter l'explosion combinatoire.
- Avoir le moins d'impact possible sur la précision du résultat (introduire pas ou peu de perte de précision).

Ces deux contraintes sont à la limite du paradoxe. En effet, si aucune fusion n'est effectuée et que les ressources mémoires diponibles sont illimitées, le résultat de l'exécution sera exact. A présent, si les ressources mémoires disponbibles sont limitées, la décision de fusionner les états sera prise. Si cette fusion est effectuée sans précaution, le nombre d'états générés diminuera mais l'imprécision du résultat dépassera surement le seuil toléré. Cet algorithme est donc implémenté de façon à identifier durant l'exécution le bon compromis "perte de précision/nombre d'états". Afin de trouver ce compromis, l'algorithme doit évidemment disposer de certaines informations comme : les paramètres définissant l'état courant du processeur.

Cet algorithme s'exécute plusieurs fois durant la construction de l'arbre symbolique. Ce mode d'exécution en alternance permet de réduire la taille de l'arbre dynamiquement. Ainsi, il nous donne l'avantage de prendre des décisions de fusionner ou de ne pas fusionner les états en ayant une idée précise de l'environnement d'exécution. Nous pourrons donc mesurer avec, plus ou moins de précision, l'impact d'une fusion sur les exécutions futures.

5.2 Explosion combinatoire

L'explosion combinatoire signifie que le nombre des comportements possibles du processeur n'est plus compatible avec les ressources mémoires disponibles. Chaque comportement possible du processeur est une combinaison des comportements de ses unités. Ainsi, plus la structure interne du processeur est complexe (exemple : plusieurs unités de traitement, unité de traitement ayant plusieurs comportements possibles) et plus le nombre de ses comportements est grand. Dans cette partie, nous allons nous inté-

resser à l'impacte de la structure interne du PowerPC sur l'explosion combinatoire. Autrement dit, nous allons déterminer une borne supérieure du nombre d'états atteignables par le PowerPC durant l'exécution d'un programme.

5.2.1 Quantification de l'explosion combinatoire : détermination d'une borne supérieure dans le cas du PowerPC

Durant l'exécution symbolique le nombre d'états générés va s'accroitre de façon exponentielle.

Théorème
Notons par :

- ρ représente la profondeur du pipeline,
- σ est la capacité maximale du processeur (le nombre maximal d'instructions que le processeur peut exécuter durant un cycle),
- η représente le nombre d'instructions composant le programme que l'on souhaite analyser.

Une borne supérieur du nombre d'état généré est : $3^{\sigma\ \eta\ \rho}$.

Preuve du théorème
Si le processeur est le PowerPC 603, un pas d'exécution symbolique, lorsque le processeur requiert des instructions [1] de la mémoire cache, va générer en pratique 3 états :

- (1) les deux instructions sont dans le cache (th(ii) : la récupération des deux instructions prend un temps "time hit"),
- (2) seule la première instruction est dans le cache (th(i) : la récupération de la première instruction prend un temps "time hit"),
- (3) les deux instructions ne sont pas dans le cache (tm : la récupération des deux instructions prend un temps "time miss").

Le nombre d'états générés sera 3^g. La valeur de g est ce que nous allons tenter de déterminer.

Notons que ce nombre d'états générés à chaque cycle d'exécution est une moyenne. Ceci signifie que durant l'exécution ce nombre varie, cependant il dépasse rarement la valeur 3, exemples :

- le tout premier pas d'exécution symbolique génère 6 états. Conséquence de l'initialisation du cache d'instructions à l'état inconnu ($IC = \top$) et donc, aux 3 états précédemment cités, nous devons en rajouter 3 autres ($trl + th(ii)$, $trl + th(i)$, $trl + tm$) pour prendre en compte le cas où la mémoire cache serait occupée au moment où elle recevrait notre requête. Ceci retardant sa réponse d'une durée trl qui correspond au temps de rechargement d'une ligne de cache.
- une fois l'instruction dans le cache nous obtenons, au plus, deux scénarios : instruction dispatchée ou instruction en attente de dispatch.

A présent supposons que nous n'exécutons qu'une seule instruction. Cette dernière génère 3^1 états quand elle atteint le premier étage (unité) du pipeline, 3^2 états quand elle atteint le second et 3^ρ états quand elle atteint le dernier étage (voire figure 5.1.**(1)**).

Généralisons ce raisonnement en commençant par lever l'hypothèse que le pipeline n'exécute qu'une seule instruction. A présent le pipeline est donc plein d'instructions, le nombre d'états générés par toutes ces instructions est borné par : $3^\rho * 3^{\rho-1} * ..... * 3^1$.

Notons que ce résultat inclut les états générés suite aux dépendances qui apparaissent durant l'exécution entre les instructions (résultat d'une instruction représentant l'opérande d'une autre instruction).

1. Le PowerPC 603 récupère à chaque cycle jusqu'à deux instructions de la mémoire cache.

Supposons que σ_i soit le nombre d'instructions gérées durant 1 cycle par l'unité $unit_i$ [2], suivant le même raisonnement, le résultat précédent devient (voir figure 5.1.**(2)**) : $3^{\rho\ \sigma_\rho} * 3^{(\rho-1)\ \sigma_{\rho-1}} * \ldots\ldots\ldots * 3^{1\ \sigma_1}$. Ce nombre peut être borné par $3^{\rho(\ \sigma_\rho + \ \sigma_{\rho-1} + \ldots\ldots + \ \sigma_1)}$. En supposant que la capacité maximale du processeur est la somme des capacités maximales de ces unités, nous obtenons : $3^{\sigma\ \rho}$.

Ce résultat représente une borne supérieure au nombre d'états générés quand la première instruction atteint le dernier étage du pipeline. Donc la largeur maximale de l'arbre symbolique quand la dernière instruction du programme finie son exécution est : $3^{\sigma\ \eta\ \rho}$

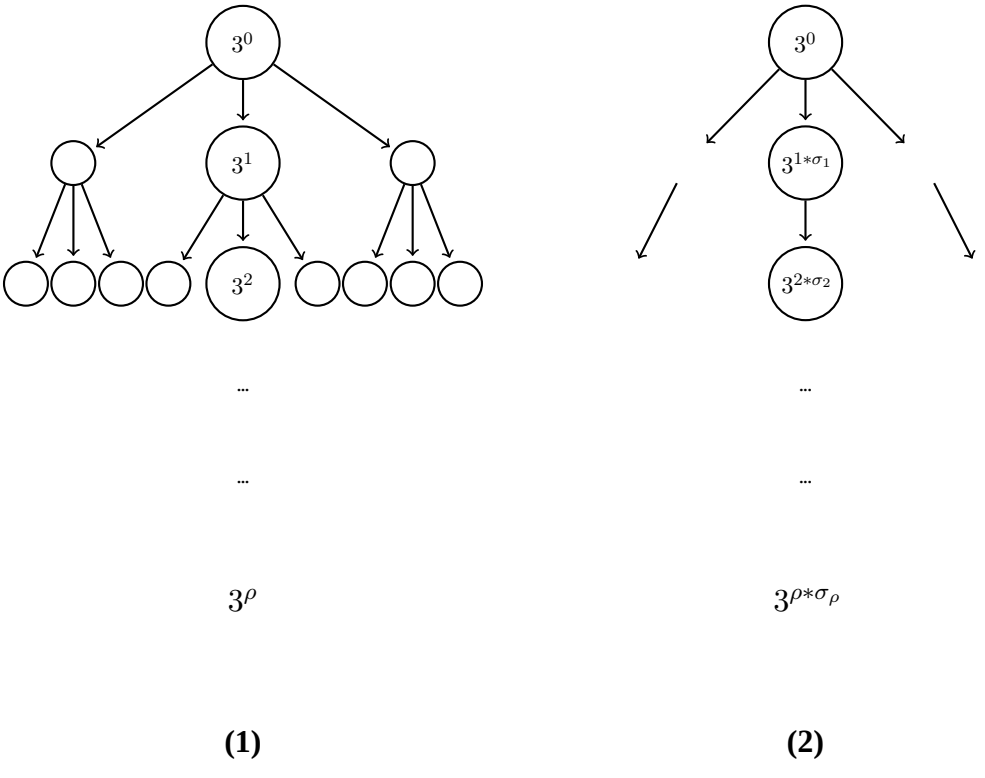


FIGURE 5.1: Calcul d'une borne supérieure au nombre d'états générés : **(1)** chaque unité peut gérer juste une instruction par cycle, **(2)** chaque unité peut gérer un nombre σ d'instructions par cycle.

5.2.2 Nombre d'états atteignables par le processeur

Dans la pratique, le résultat obtenu ci-dessus ($3^{\sigma\ \eta\ \rho}$) est une sur-estimation lointaine de la valeur réelle. Cette énorme différence est dûe au fait que durant ce calcul de complexité les limites du processeur ne sont pas prises en compte. Ces limites représentent le nombre total d'états que peut prendre un processeur lorsqu'il exécute un programme de longueur finie.

Théorème
La borne supérieure au nombre d'états maximal qu'un processeur peut atteindre est η^σ.

Preuve du théorème
Supposons que σ_i soit le nombre maximal d'instructions que l'unité i peut traiter durant 1 cycle. Ceci

2. Exemple, si le processeur est le PowerPC 603e : $\sigma_{unitentire}$=1, $\sigma_{unitflottante}$=3, etc.

implique que le nombre maximal d'états que l'unité i peut atteidre durant l'exécution est η^{σ_i}. Le processeur étant un ensemble d'unités communiquant entre elles, nous devons prendre en compte les états générés suite aux interactions entre ces unités. Ainsi, le nombre maximal d'états atteignales par le processeur est : $\eta^{\sigma_\rho} * \eta^{\sigma_{\rho-1}} ..... * \eta^{\sigma_1}$. En supposant que la capacité maximale du processeur est la somme des capacités maximales de ces unités, nous obtenons : η^{σ}.

Ce résultat prouve qu'après un certain nombre de pas d'exécution symbolique le nombre d'états générés va naturellement se stabiliser autour d'une valeur que nous pouvons voir comme un "seuil de saturation".

5.3 Fusion par abstraction

La section 5.2 a permis de montrer que le nombre d'états générés se stabilise nécessairement autour d'une valeur de saturation après une certains temps. Ce seuil de saturation pouvant atteindre des valeurs assez importantes, nous avons mis au point une méthode de fusionnement afin de s'assurer que le nombre d'états générés durant l'analyse n'explose pas.

Dans le cadre général de l'abstraction, les propriétés d'un programme sont définies dans une sémantique concrète du programme, exprimée en général avec un ensemble d'équations mathématiques. La sémantique d'un programme P est un modèle de calcul décrivant le comportement de ce programme pour toutes les exécutions possibles. La sémantique de traces [41] représente la trace d'exécution du programme qui est une séquence d'états consécutifs possibles d'exécution. Ainsi, utiliser cette technique dans une méthode de fusion revient soit, à fusionner des traces reflétants un comportement récurssif (utilisation d'opérateurs d'élargissement [59]), soit à fusionner des états en faisant correspondre plusieurs états concrets à un état abstrait.

5.4 Fusion par abstraction de traces

L'utilisions de l'abstraction pour fusionner des traces d'exécution, implique généralement d'identifier les noeuds du CFG sur lesquels reboucle l'exécution. (noeud b1 en rouge, voir figure 5.2).

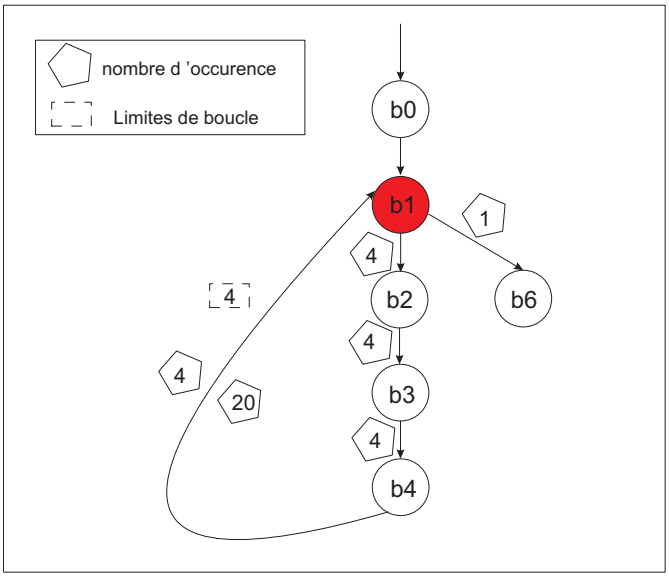


FIGURE 5.2: Fusion par abstraction de traces d'exécution

Ces noeuds sont les points à partir desquels l'abstraction (opérateur d'élargissement) est lancée. Cette abstraction a pour objectif de représenter les traces associées aux mutiples itérations de la boucle par une trace implicite. Ainsi, les boucles sont transformées en simples traces d'exécution linéaires.

Concernant notre problématique, cette fusion par abstraction de traces n'est pas une solution car l'arbre symbolique que nous générons risquerait d'exploser avant d'atteidre le noeud sur lequel l'architecture reboucle. Ainsi, pour surmonter ce problème, nous devons en premier lieux procéder à des fusions d'états.

5.5 Fusion par abstraction d'états

Afin d'expliquer clairement le principe de la fusion par abstraction d'états, nous allons analyser le programme P écrit en C ci-dessous :

```
pc0->   int i,j
pc1->   i=3;
pc2->   j=i+1;
pc3->   i=j;
```

Nous associons une sémantique concrète T à ce programme, chaque état est donné par un triplet (pc, v_i, v_j) où pc est le point de programme atteint et v_i, v_j sont les valeurs respectives des variables i et j. Cette sémantique est une trace :
$[pc0, i \leftarrow \top, j \leftarrow \top]$, $[pc1, i \leftarrow 3, j \leftarrow \top]$, $[pc2, i \leftarrow 3, j \leftarrow 4]$, $[pc3, i \leftarrow 4, j \leftarrow 4]$
Notons que $\top$ désigne que la variable a une valeur indéfinie.

Pour passer de la sémantique concrète T vers la sémantique abstraite t et vice versa, l'interprétation abstraite se base sur une correspondance de Galois, via la fonction d'abstraction $\alpha : T \rightarrow t$ et la fonction de concrétisation $\gamma : t \rightarrow T$. Par l'abstraction de cette sémantique nous obtenons une sémantique abstraite $t = \alpha(T)$, moins précise. Par exemple, en associant à la trace d'exécution l'ensemble des états qui la constitue, nous obtenons :

$\{(pc0, \top, \top), (pc1, 3, \top), (pc2, 3, 4), (pc3, 4, 4)\}$.
Une abstraction consiste à déterminer l'ensemble des valeurs des deux variables i et j :
$i \in \{\top, 3, 4\}, j \in \{\top, 4, 4\}$.
Nous remarquons que l'abstraction provoque une perte d'information comme par exemple l'ordre des valeurs à chaque point du programme.

Ainsi, utiliser ce type d'abstraction pour réduire le nombre d'états est très efficace si le but se limite à contenir l'explosion du nombre d'états. Cependant, du fait que cette abstraction introduit des comportements imprévisibles, cette méthode est très préjudiciable quant à la précision des temps calculés.

C'est pourquoi, nous avons développé une méthode de fusionnement dynamique qui agit sur l'augmentation exponentielle du nombre d'états de façon à ce que cette augmentation soit, en pratique, le plus souvent possible linéaire. Cette méthode est exécutée en parallèle avec l'analyse et nous permet de fusionner des états en ayant un impact minime sur les comportements futurs du processeur et donc sur la perte de précision concernant les temps d'exécution.

5.6 Notre méthode de fusionnement

5.6.1 Identification des états

La méthode de fusionnement proposée effectue une abstraction des traces d'exécution guidée par une identification dynamique des points repliage [3] de l'arbre symbolique. Cette méthode se base donc sur l'étude des traces d'exécution.

Définition 15 *Une trace d'exécution TE est une liste ordonnée d'états $\mathbf{s} \in \mathcal{N}$ reliés par des transitions $\mathbf{t} \in \mathcal{TR}$. Une trace d'exécution n'a qu'un seul état initial $\mathbf{s}_{ini}$ et un seul état final $\mathbf{s}_{fin}$*
Si $\mathbf{s} \neq \mathbf{s}_{ini} \wedge \mathbf{s} \neq \mathbf{s}_{fin}$ alors $\mathbf{s}$ n'a qu'une transition entrante $\mathbf{t_e}$ et une transition sortante $\mathbf{t_s}$

Une trace d'exécution reflète donc une évolution possible du processeur. Comme nous l'avons montré précédemment, un processeur qui exécute un programme donné a une limite supérieure au nombre d'états qu'il peut atteindre. Cette limite est une conséquence du nombre précis des comportements possibles de l'ensemble des unités qui composent le processeur. Les différentes combinaisons des comportements des unités, qui respectent la sémantique dictée par le processeur, représentent les traces d'exécution faisables.

Ainsi, il existe un nombre limite de combinaisons. Il est donc possible d'envisager qu'un ensemble de traces d'exécution vont, à un point de l'exécution, évoluer de façon similaire. Cette évolution similaire se matérialise par l'apparition d'un état $\mathbf{s}_{fs}$ (point de repliage de l'arbre symbolique) sur plusieurs traces d'exécution. D'une trace à l'autre, cet état $\mathbf{s}_{fs}$ a évidemment un historique d'exécution différent (PC et $countime$ différents). Ceci implique qu'il existe des points d'exécution où le processeur se comporte de la même façon quelque soit son historique.

5.6.2 Preuve d'existence d'états équivalents

Le traitement d'une instruction I implique qu'un environement d'exécution particulier soit disponible : $T(I) \Rightarrow P \in \mathbb{S}_I$ avec $\mathbb{S}_I$ est l'ensemble des états du processeur P permettant l'exécution de l'instruction I. Ainsi, il existe un ensemble de contraintes C_I que l'environnement d'exécution doit satisfaire pour traiter l'instruction I.

Si nous généralisons ce raisonnement à un flot d'instructions $\{I_1, I_2, ...I_k\}$, les contraintes s'additionneront. Ainsi, l'environnement d'exécution doit satisfaire la conjonction des ensembles des contraintes associées aux instructions : $C_{I_1} \wedge C_{I_2} \wedge ...C_{I_k}$.

3. l'expression "points de repliage " ne signifie pas le point d'entrée d'une boucle mais plutôt une équivalence d'états.

Afin d'exécuter un flot d'instructions le processeur doit donc appartenir à un ensemble d'états restreint : $P \in \mathbb{S}_{I_1} \cap \mathbb{S}_{I_2} \cap ...\mathbb{S}_{I_k}$, ce qui assure l'apparition de comportements équivalents durant une exécution.

5.6.3 Principe de fonctionnement de notre méthode d'identification d'états similaires

Durant l'exécution symbolique nous construisons un arbre symbolique qui contient l'ensemble des états atteignables. En d'autres termes, en se basant sur l'historique d'exécution ainsi que sur le modèle du processeur (voir section 3), à chaque pas d'exécution nous générons tous les scénarios (traces) qui peuvent se produire. L'extension proposée de l'exécution symbolique peut être vue comme une séquence d'expressions conditionnelles ("IF-THEN-ELSE"). Pour illustrer cela, reprenons l'exemple d'un processeur qui tente de récupérer une instruction. Cette instruction peut être ou ne pas être dans le cache (Cache HIT or cache MISS). L'arbre symbolique résultant contiendra donc deux traces d'exécution : (1) instruction dans le cache et (2) instruction absente du cache. Rajoutons à présent l'hypothèse que durant l'exécution symbolique toutes les traces d'exécution seront visitées, alors un des trois scénarios suivants est possible :

- Les traces générées mènent à des états différents. La fusion est donc formellement interdite, voir la figure 5.3.**(1)**. Ce cas est similaire à un bloc d'instructions "IF-THEN-ELSE" dont un des chemins d'exécution contient une instruction de saut. Lorsque cela se produit deux comportements totalement différents sont possibles : (1) nous exécutons les instructions du bloc "THEN" auquel cas un saut vers des instructions du programme est effectué, (2) nous explorons la branche "ELSE" et nous continuons l'exécution en traitant les instructions placées en sortie du bloc contionnel (les instructions placées tout de suite après le bloc "IF-THEN-ELSE")
- Les traces générées mènent à des **états fortement similaires** (voir section 5.6.5), donc ces états sont fusionnés automatiquement et sans introduire de perte de précision (voire la figure 5.3.**(2)**). Ce cas ressemble au traitement d'un bloc "IF-THEN-ELSE" dont l'exécution suivant ses deux branches n'affectent pas les instructions venant après le point de sortie du bloc.
- Les traces générées mènent approximativement aux mêmes états (**états faiblement similaires** : voir section 5.6.8). Ainsi, la décision de fusionner ou pas ces états dépend de la perte de précision introduite par cette fusion, (voir figure 5.3.**(3)**). Pour estimer cette perte de précision il est primordial de faire une prédiction. Cela consiste à aller plus en avant dans l'exécution et à générer pour chaque trace d'exécution les états successeurs aux états faiblement similaires afin de les comparer. Ce cas peut donc s'apparenter à un bloc d'instructions "IF-THEN-ELSE" qui affecterait une valeur à une donnée suivant la trace "THEN" qui serait différente de celle qui lui aurait été affecté selon le chemin "ELSE". En admettant que cette donnée ait un faible impact sur les exécutions futures, ces deux traces peuvent être fusionnées.

Notre méthode de fusionnement est implémentée de façon à identifier, durant l'exécution, les états sur lesquels l'arbre généré est susceptible de se replier (les états fortement similaires). Cela implique qu'à un moment, l'analyse doit s'interrompre afin que la méthode de fusionnement parcourt l'arbre symbolique. Cette méthode visite toutes les traces récemment générées [4] afin de créer des classes d'équivalence. Ces classes contiennent respectivement des états similaires. Ensuite, dans chaque classe une comparaison entre les états qu'elle contient est faite. Ainsi, les états fortement similaires sont identifiées puis fusionnés.

5.6.4 Notion générale de similarité

La similarité entre deux états d'un processeur peut d'être définie de différentes façons :

4. Les traces générées après la dernière fusion.

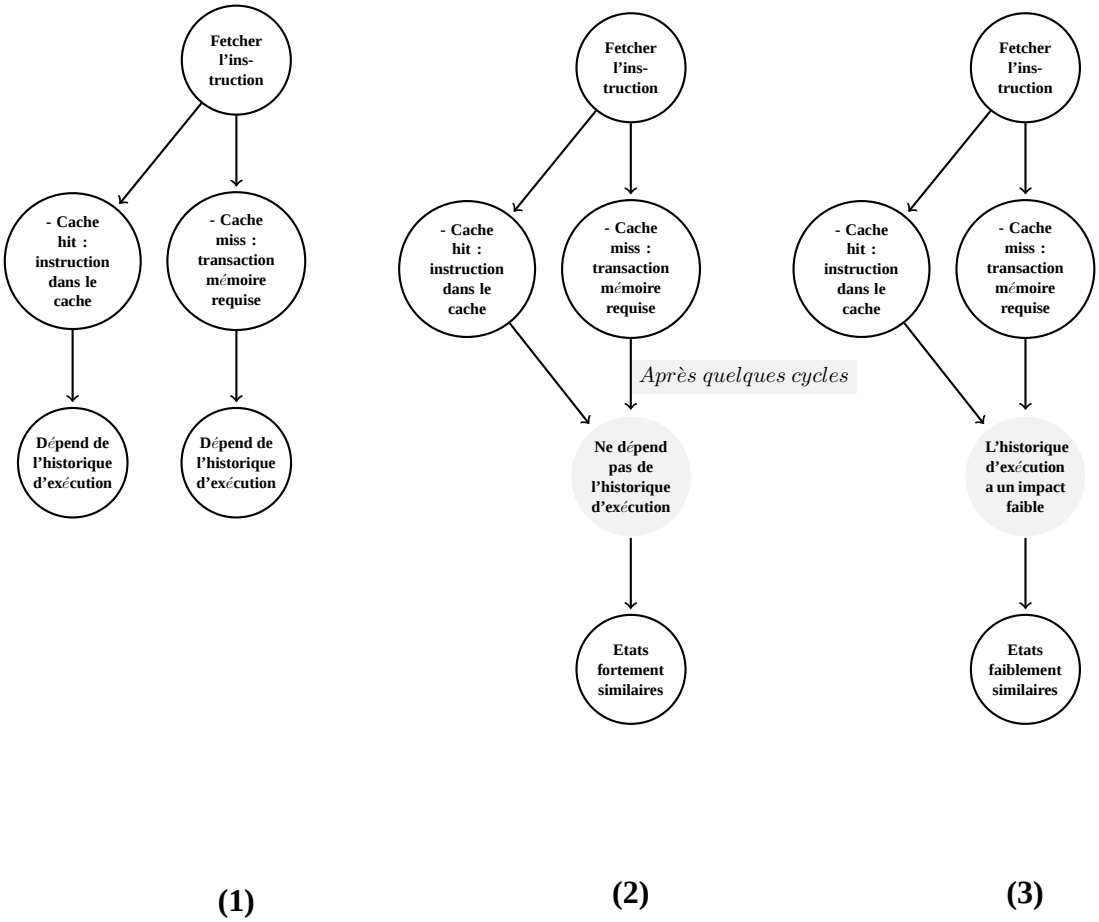


FIGURE 5.3: **(1)** Fusion interdite, **(2)** Fusion sans perte de précision, **(3)** Fusion avec perte de précision.

- Similarité au niveau des sous-composants Sct : équivalence entre certains sous-composants, exemple : équivalence au niveau des unités d'exécution (EX).
- Similarité au niveau du processeur Pr : équivalence entre tous les sous-composants.

Partant de ces deux notions, nous pouvons définir un ordre partiel entre les ensembles d'états similaires EES.

Cet ordre partiel est défini comme suit : un ensemble EES_2 qui contient des états dont deux de leurs sous-composants Sct_2^1 et Sct_2^2 sont similaires est inclut dans l'ensemble EES_1 qui contient des états dont un de leurs sous-composants Sct_1 est similaire, si et seulement si un des deux sous-composants similaires des états contenus dans EES_2 conrrespond au sous-composant similaire des états contenus dans EES_1.

$$EES_2 \subset EES_1 \Rightarrow Sct_2^1 = Sct_1 \vee Sct_2^2 = Sct_1.$$

Ainsi, le plus grand ensemble est l'ensemble de tous les états contenus dans l'arbre d'exécution et le plus petit ensemble est celui contenant des états similaire au niveau du processeur Pr (tous les composants sont similaires). Cet ensemble est contenu dans tous les autres ensembles et introduit la notion de forte similarité entre états.

Nous définissons donc cette notion, comme suit :

Définition 16 *Deux états (S_1 et S_2) sont fortement similaires si :*

- *Tous leurs composants sont identiques*
  $\mathcal{SC}_1 = \mathcal{SC}_2$ *:* $\mathbf{P}_1 = \mathbf{P}_2 \wedge \mathbf{IC}_1 = \mathbf{IC}_2 \wedge \mathbf{DC}_1 = \mathbf{DC}_2$
  (la comparaison entre les unités respectives des deux états (S_1 et S_2) résulte en une parfaite équivalence.
- *Leurs* **PC***s peuvent être différents :* $\mathbf{PC}_1 \neq \mathbf{PC}_2$
- *Leurs compteurs internes peuvent être différents :* $\mathbf{countime_1} \neq \mathbf{countime_2}$

Afin de montrer l'existence de ces états $\mathbf{s}_{fs}$, reprenons l'exemple de code introduit à la section 4.5. Dans cette partie, nous avions exposé l'évolution de l'analyse. Intéressons nous au quatrième pas d'exécution symbolique. A ce niveau de l'exécution, admettons que nous interrompions la construction de l'arbre afin de comparer les états qu'il contient deux à deux.

Phase d'identification

Comme le montre la figure 5.4, les traces d'exécution (S0 - S1 - S1.1 - S1.2) et (S0 - S3 - S3.1 - S3.2) mènent à des états fortement similaires. Ainsi, une fois l'arbre symbolique parcouru, nous pouvons construire deux classes d'équivalence :

- la première contient (S1.3a et S3.3a).
- la seconde contient (S1.3b et S3.3b).

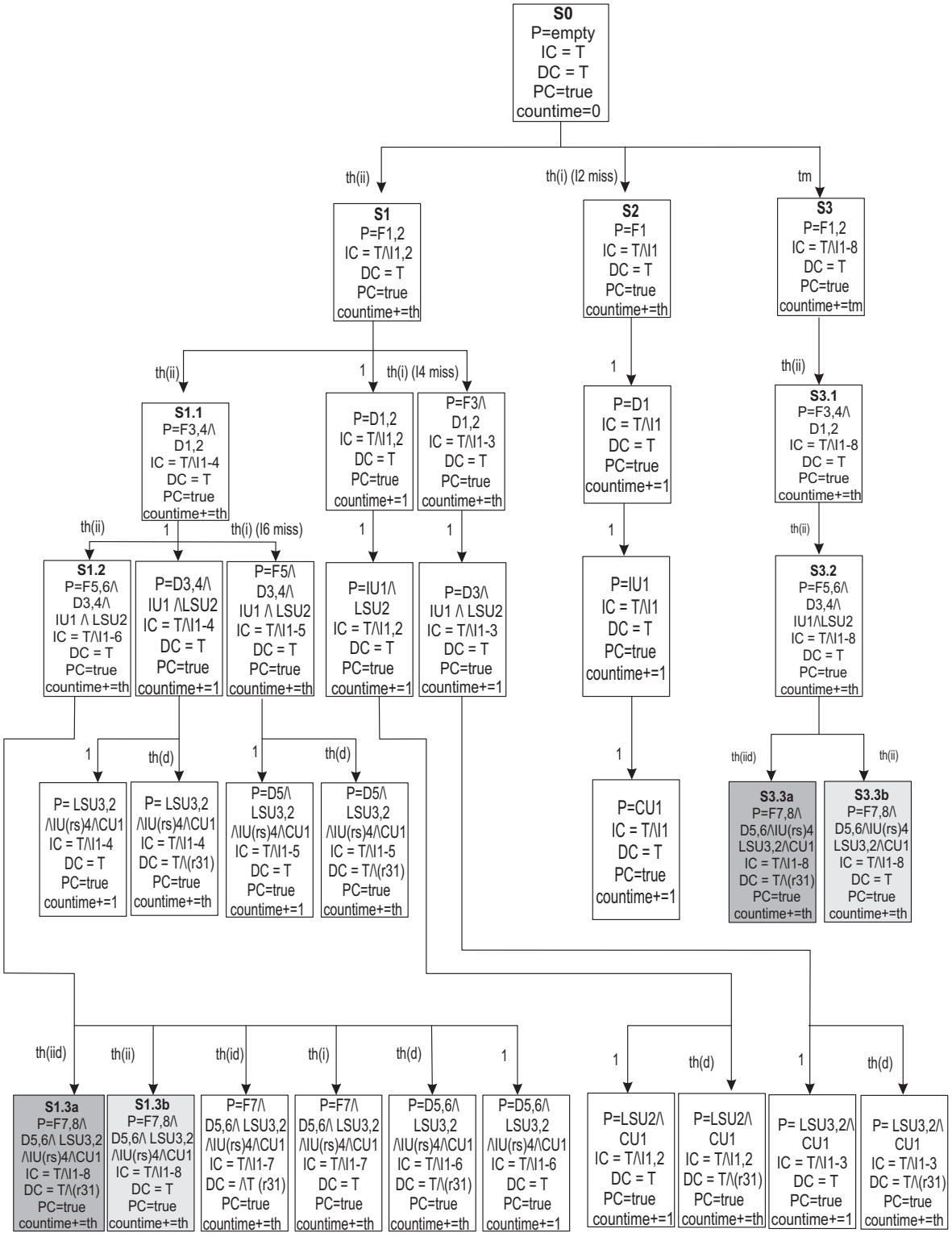


FIGURE 5.4: Identification : Etats fortement similaires

Phase de fusionnement

La méthode de fusionnement a donc formé deux classes d'équivalence. Le fusionnement consiste à remplacer tous les états appartenant à la même classe d'équivalence par un seul état. Prenons comme exemple la première classe d'équivalence (S1.3a et S3.3a). Après fusion de ces deux états (voir figure 5.5), nous obtenons un état $\mathbf{s}_{fs}$ paramétré comme suit :

- ($\mathbf{P}_{\mathbf{s}_{fs}}$= $\mathbf{P}_{S1.3a}$=$\mathbf{P}_{S3.3a}$), ($\mathbf{IC}_{fs}$ = $\mathbf{IC}_{S1.3a}$=$\mathbf{IC}_{S3.3a}$), ($\mathbf{DC}_{fs}$ = $\mathbf{DC}_{S1.3a}$=$\mathbf{DC}_{S3.3a}$)
- $\mathbf{PC}_{\mathbf{s}_{fs}}$ = $\mathbf{PC}_{S1.3a} \vee \mathbf{PC}_{S3.3a}$
- $countime_{\mathbf{s}_{fs}} = max\{countime_{S1.3a}, countime_{S3.3a}\}$
- Ses transitions (TR) entrantes sont toutes les transitions entrantes de S1.3a et de S3.3a.

Notons que le compteur interne de l'état représentant la fusion, est le maximum des compteurs internes des états appartenant à la classe d'équivalence ($countime_{\mathbf{s}_{fs}} = max(countime_{S1.3a}, countime_{S3.3a})$). Ainsi, les transitions entrantes, d'un état résultant d'une fusion, ne sont pas labélisées [5].

Cette fusion est très simple à mettre en œuvre et n'introduit aucune perte de précision sur les temps d'exécution (voir partie 5.6.6).

5. Ce compteur n'est pas incrémenté du temps (label) de sa transition entrante.

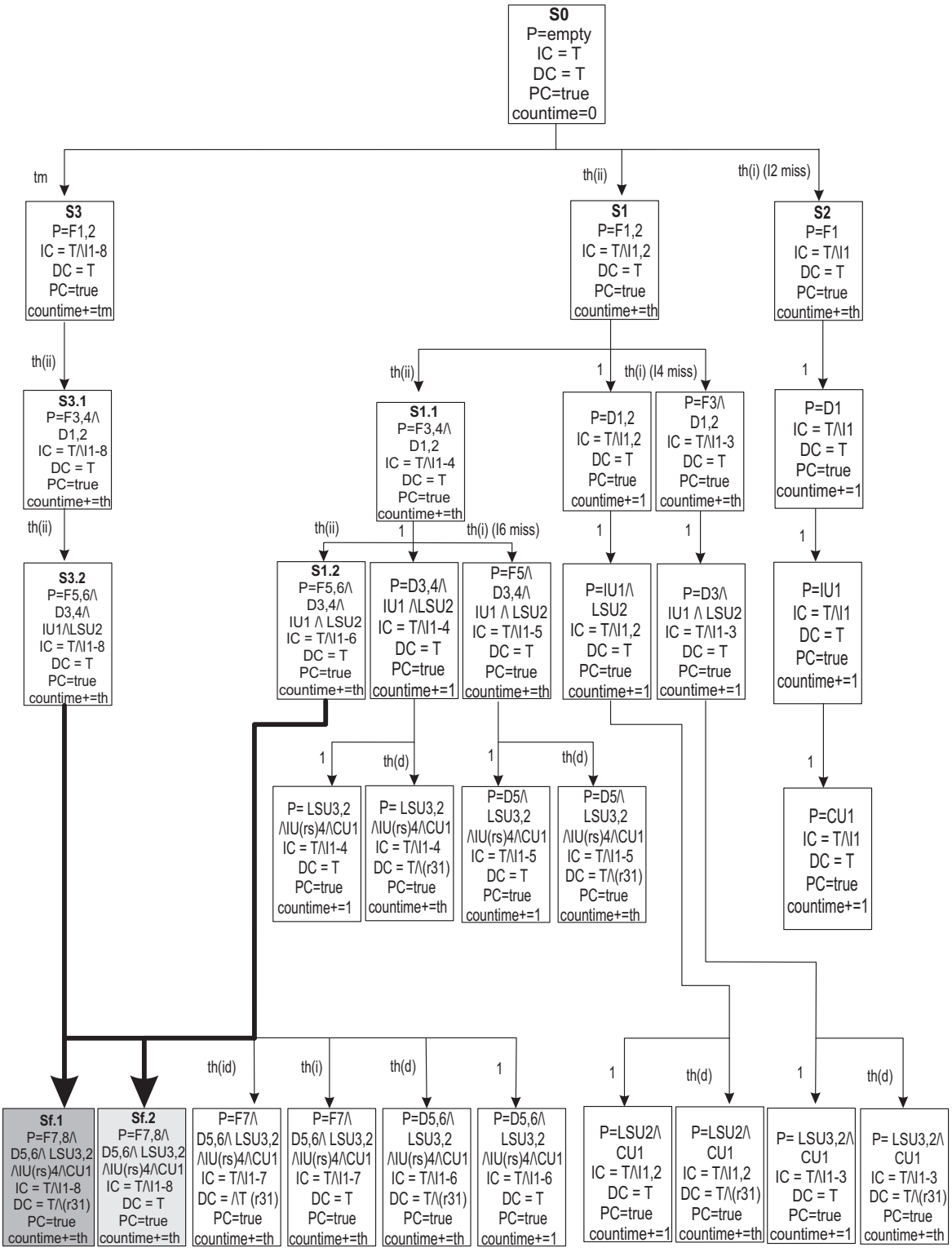


FIGURE 5.5: Fusionnement : Etats fortement similaires

5.6.5 Algorithme de fusionnement sans perte de précision

L'algorithme de fusionnement, ci-dessous, identifie et fusionne les états fortement similaires :

$Init\_state \leftarrow$ { (P = $\emptyset$, IC = DC = $\top$), PC = true, countime=0 } ;
$\mathbf{s}_0 \leftarrow \{Init\_state\}$, $current\_states \leftarrow \mathbf{s}_0$;
$\mathcal{G} \leftarrow \{Init\_state\}$;
while $current\_states \neq \emptyset$ **do** (Propagation)
  $\mathbf{s}_d \notin current\_states$ (retirer $\mathbf{s}_d$ de $current\_states$) ;
  $\mathcal{G}(\mathbf{s}_d)$ = succ$_\rho$ ($\mathbf{s}_d$) (construire l'arbre symbolique résultant après ρ pas symboliques);
  $\mathcal{G} \leftarrow \mathcal{G}(\mathbf{s}_d) \cup \mathcal{G}$;
  EQ = $\{Eq_1, ..., Eq_m\}$ (construire les classes d'équivalence (voir partie 5.6.8.1) avec m : le nombre de classe d'équivalence d'états fortement similaires);
  foreach $Eq_i = \{e_1, \ldots e_n\} \subset EQ$ **do** (Analyse avant)
    $S_{strong\_sim\_i} = \bigsqcup_{e_j \in Eq_i} e$ (fusionner les états fortement similaires) ;
    $S_{succ\_i} = \bigsqcup_{\mathtt{succ}(e)}$ (fusionner tous les successeurs des états fortement similaires) ;
    $e \notin Eq_i$ (retirer les états fusionnés des classes d'équivalence Eq_i) ;
    MAJ $\mathcal{G}$ (Mettre à jour $\mathcal{G}$) ;
  foreach $\mathbf{s}_d \in \mathcal{G}$ **do** (Fin de l'analyse ?)
    if ($\mathbf{s}_d \notin S_{fin\_analyse}$) **then**
      $current\_states \leftarrow \mathbf{s}_d$;

avec :
$\mathcal{G}$: arbre symbolique.
$\mathbf{s}_d$: état final de $\mathcal{G}$ vis-à-vis du point d'exécution courant et d $\in$ [0..h] indice de l'état final.
$S_{fin\_analyse}$: ensemble des états finaux de $\mathcal{G}$ vis-à-vis de l'analyse (états finaux générés par la dernière instruction du programme).

Principe de fonctionnement de l'algorithme de fusionnement

- Partant de l'exécution symbolique d'une instruction du programme, un arbre symbolique est construit ($\mathcal{G}(\mathbf{s})$). Chaque branche de cet arbre aura une longueur équivalente à un nombre de pas d'exécution symbolique égale à la profondeur du pipeline (ρ représente le nombre d'étages du pipeline).
- Identifier les états équivalents (fortement similaires) contenus dans l'arbre symbolique et fusionner ces états ($S_{strong\_sim\_i} = \bigsqcup_{e_j \in Eq_i} e$). Cette fusion n'introduit aucune perte de précision.
- Partir de ces états et fusionner tous leurs successeurs ($S_{succ\_i} = \bigsqcup_{\mathtt{succ}(e)}$).

5.6.6 Preuve de la précision des résultats

Fusionner deux états (S_1 ,S_2) fortement similaires consiste à
- Sauvegarder l'état du processeur partagé par ces deux états : $\mathcal{SC}_{merge}$:
  $\mathbf{P}_{merge} = \mathbf{P}_1 = \mathbf{P}_2$, $\mathbf{IC}_{merge} = \mathbf{IC}_1 = \mathbf{IC}_2$, $\mathbf{DC}_{merge} = \mathbf{DC}_1 = \mathbf{DC}_2$
- Conserver les historiques d'exécution des deux états :
  $\mathbf{PC}_{merge} = \mathbf{PC}_1 \vee \mathbf{PC}_2$
- Conserver la valeur maximale des temps d'exécution des deux traces :
  $countime_{merge} = max\{countime_1, countime_2\}$

Théorème

Fusionner des états fortement similaires n'a aucun impact sur la précision des temps obtenus en fin d'analyse (fusion sans perte de précision).

Preuve du théorème

L'état $\mathcal{SC}_{merge}$, du pipeline et des mémoires caches, est conservé $\mathcal{SC}_{merge} = \mathcal{SC}_1 = \mathcal{SC}_2$, ainsi la fusion n'a aucun impact sur l'état courant du processeur. De plus, l'historique d'exécution $\mathbf{PC}_{merge}$, de l'état généré par la fusion S_{merge}, est construite de façon à contenir toutes les traces faisables menant à cet état ($\mathbf{PC}_{merge} = \mathbf{PC}_1 \vee \mathbf{PC}_2$). Ainsi, après la fusion, l'évaluation des instructions de branchement se fera par l'évaluation disjointe des deux $\mathbf{PC}s$ (évaluation de $\mathbf{PC}_1$ indépendamment de $\mathbf{PC}_2$ et vis versa). De cet façon, l'influence de l'historique d'exécution sur les comportements futurs du processeur, avant et après la fusion, est la même (voir partie 5.6.7). Concernant le composant $countime$, nous conservons la valeur maximale des deux temps car notre but est de calculer des WCETs [6].

5.6.7 Impact de la fusion sur les traces d'exécution

La fusion implique d'effectuer une disjonction de comportements. Cependant si cette disjonction n'est pas maitrisée, des traces d'exécution infaisables sont susceptibles d'apparaitre. Prenons l'exemple de la fusion de deux états fortement similaires S_1 et S_2. Supposons que leurs **PC**s respectifs soient :
$\mathbf{PC}_1 = a_1 > 0 \wedge a_2 > 1$ et $\mathbf{PC}_2 = a_1 > 20 \wedge a_2 > 10$ (a_1 et a_2 sont des entrées symboliques).
Chaque **PC** représente un historique et donc une trace d'exécution précise. La fusion de S_1 et S_2 génère donc un état S_{merge} dont le **PC** est défini par :
$\mathbf{PC}_{merge} = \mathbf{PC}_1 \vee \mathbf{PC}_2 = (a_1 > 0 \wedge a_2 > 1) \vee (a_1 > 20 \wedge a_2 > 10)$

Cette expression mathématique implique que la fusion introduit de nouveaux historiques d'exécution. En effet, $(a_1 > 0 \wedge a_2 > 1) \vee (a_1 > 20 \wedge a_2 > 10)$ représente aussi d'autres historiques d'exécution :

- $(a_1 > 0 \wedge a_2 > 10)$ (historique d'exécution introduit par la disjonction).
- $(a_1 > 20 \wedge a_2 > 1)$ (historique d'exécution introduit par la disjonction).

Une fusion qui suit cette logique peut être vu, au même titre que l'interprétation abstraite, comme un opérateur d'abstraction α qui tend à représenter une ensemble de traces d'exécution par un sur-ensemble. En effet, l'expression $(a_1 > 0 \wedge a_1 > 20) \vee (a_2 > 1 \wedge a_2 > 10)$ définit un sur-ensemble contenant les traces faisables représentées par $(a_1 > 0 \wedge a_2 > 1)$ et $(a_1 > 20 \wedge a_2 > 10)$).

Ainsi, lors de futures évaluations d'instructions de branchement, des comportements infaisables sont susceptibles d'apparaitre (les comportement engendrés par les historiques d'exécution qu'a introduit la disjonction de $(a_1 > 0 \wedge a_2 > 10)$ et de $(a_1 > 20 \wedge a_2 > 1)$). Ce qui nuit considérablement à la précision des temps calculés.

C'est pourquoi, la disjonction que nous utilisons n'est pas le simple opérateur mathématique, mais plutôt une disjonction entre **PC**s. Autrement dit, chaque **PC** est considéré comme une entité indivisible par l'opérateur de disjonction. Ainsi, durant notre analyse le $\mathbf{PC}_{merge}$ est sauvegardé comme suit :
$\mathbf{PC}_1 \vee \mathbf{PC}_2 = (a_1 > 0 \wedge a_2 > 1) \vee (a_1 > 20 \wedge a_2 > 10)$

L'évaluation d'une instruction de branchement future se fera donc par l'évaluation de $\mathbf{PC}_1$. Ensuite, cette même opération sera réitérée avec $\mathbf{PC}_2$. Ceci garantit la continuation de l'analyse en respectant uniquement les historiques d'exécution possibles.

6. Notons que ce même raisonnement pourrait être appliqué au calcul des temps optimaux (BCET : best case execution time) à condition de conserver la valeur minimale des temps ($countime_{merge} = min\{countime_1, countime_2\}$).

Ainsi, la fusion de traces d'exécution fortement similaires est un moyen de supprimer les comportements redondants du processeur. De cette manière, la taille de l'arbre symbolique est réduite sans pour autant perturber l'évolution du modèle Time-accurate du processeur.

5.6.8 Extension de la notion de similarité entre les états

Il est possible que la fusion des traces fortement similaires ne permette pas d'assurer systématiquement la terminaison de l'analyse. Ainsi, lorsque les besoins de l'analyse en mémoire ne sont plus compatibles avec les ressources disponibles, la fusion doit être étendue à d'autre états. Cette nouvelle fusion utilise la notion de faible similarité. Elle est plus agressive en terme de diminution du nombre de traces d'exécution, cependant elle nous détourne d'un de nos objectifs premiers qui est de fournir des résultats exacts. Ainsi, de nombreux efforts ont été fournis de façon à ce que l'imprécision introduite par ce type de fusion soit la plus faible possible. Le principe de cette fusion consiste à exploiter au maximum la forte similitude entre les états identifiée grâce à l'algorithme présenté dans la partie 5.6.5.

Nous avons précédemment défini deux états fortement similaires comme étant des états qui, indépendamment de leurs historiques d'exécution respectifs, menaient à des traces d'exécution équivalentes. Ainsi, définir une notion de faible similarité entre les états consiste à identifier un ensemble de paramètres communs entre les états qui mènerait non pas à des traces d'exécution équivalentes mais néanmoins similaires.

Une façon intuitive de s'assurer de la similarité des traces d'exécution est que les paramètres choisis pour identifier les états faiblement similaires aient un impact sur le comportement global du modèle du processeur. C'est pourquoi, nous définissons la notion de faible similarité entre les états, comme suit :

Définition 17
Deux états (S_1 et S_2) sont faiblement similaires si :
- *Un de leurs paramètres principaux est différent.*

$$\begin{aligned} \mathcal{SC}_1 \simeq \mathcal{SC}_2 \quad : \quad & (\mathbf{P}_1 \neq \mathbf{P}_2 \wedge \mathbf{IC}_1 = \mathbf{IC}_2 \wedge \mathbf{DC}_1 = \mathbf{DC}_2) \\ & \vee (\mathbf{P}_1 = \mathbf{P}_2 \wedge \mathbf{IC}_1 \neq \mathbf{IC}_2 \wedge \mathbf{DC}_1 = \mathbf{DC}_2) \\ & \vee (\mathbf{P}_1 = \mathbf{P}_2 \wedge \mathbf{IC}_1 = \mathbf{IC}_2 \wedge \mathbf{DC}_1 \neq \mathbf{DC}_2) \end{aligned}$$

  *(un des paramètres (**P**, **IC**, **DC**) diffère d'un état à l'autre)*
- *Leurs* **PC***s peuvent être différents :* $\mathbf{PC}_1 \neq \mathbf{PC}_2$
- *Leurs compteurs internes peuvent être différents :* $countime_1 \neq countime_2$

5.6.8.1 Classes d'équivalence

Un algorithme qui fusionne les états faiblement (ou fortement) similaires compare, deux à deux, tous les états contenus dans l'arbre symbolique. Dans ce cas, la complexité de l'algorithme est égale à N^2 (N est le nombre d'états contenus dans l'arbre). Ceci est dû au fait que la notion d'ordre (total ou partiel) entre les états est impossible à déterminer. En effet, la comparaison est effectuée entre états concrets, ainsi un état inclut dans un autre état n'a aucune signification.

Cette complexité implique que la durée de l'analyse peut être très longue. C'est pourquoi, il est important de diviser l'ensemble des états contenus dans l'arbre symbolique en plusieurs sous-ensembles.

Cette division doit être faite dès la génération des états et en fonction de la valeur de certains de leurs paramètres. Autrement dit, tous les états contenus dans le même ensemble ont un ou plusieurs paramètres identiques. Les sous-ensembles construits sont appelés classes d'équivalences. Ainsi, la notion de classe d'équivalence a pour but de réduire la complexité de l'algorithme. En effet, cette complexité passe de N^2 à $N_1^2 + N_2^2 + ... + N_k^2$ avec $N = N_1 + N_2 + ... + N_k$ et k représente le nombre de classes d'équivalence. Le but est donc de construire un maximum de classes d'équivalence.

Définition 18 *Une classe d'équivalence est un ensemble d'états $\{S_1, S_2, ..., S_m\}$ non ordonnés susceptibles d'être fusionnés entre eux. Un état S_i ne peut être contenu dans deux différentes classes d'équivalence.*

5.6.8.2 Fusionnement de traces d'exécution similaires

Comme nous l'avons précisé, ces états faiblement similaires mènent à des traces d'exécution non pas équivalentes mais néanmoins similaires. Autrement dit, la fusion de ces traces implique de définir une trace implicite représentant ces deux traces. Evidemment cette trace implicite est susceptible de ne pas représenter uniquement ces deux traces. En effet, déterminer une trace implicite, comme nous l'avons démontré dans la partie 4.8, suit la logique de l'opérateur d'abstraction α. Il est donc possible, comme dans l'approche AbsInt, de définir une fonction d'abstraction α et une autre de concrétisation γ. La fonction α aurait pour objectif de déterminer la trace abstraite (trace implicite) qui correspondrait à la fusion des traces et la fonction γ déterminerait l'ensemble des traces concrètes associées à une trace abstraite. Cet ensemble contiendrait évidemment les traces explicites (faisables) plus un ensemble de traces concrètes infaisables.

Exemple, admettons que S_1 et S_2 soient deux états faiblement similaires. S_1 est défini par :

- $\mathcal{SC}_1$: $\mathbf{P}_1 = F_i$,$\mathbf{IC}_1 = T \wedge i$, $\mathbf{DC}_1 = T \wedge d$
- $\mathbf{PC}_1$=$true \wedge d > 10$
- $countime_1 = 70$

S_2 est défini par :

- $\mathcal{SC}_2$: $\mathbf{P}_2 = D_i$,$\mathbf{IC}_2 = T \wedge i$, $\mathbf{DC}_2 = T \wedge d, d_1$
- $\mathbf{PC}_2$=$true \wedge d < 20$
- $countime_2 = 120$

Définissons une fonction α qui transforme ces deux états faiblement similaires en un état S_{merge}. Admettons que cette transformation soit une simple disjonction entre les paramètres respectifs des deux états. Ainsi, la disjonction ne serait pas introduite uniquement au niveau des **PC**s (comme pour les états fortement similaires) mais aussi au niveau des composants du processeur. La fusion de ces deux états résulterait donc en un état S_{merge} dont les paramètres seraient les suivants :

- $\mathcal{SC}_{merge}$: $\mathbf{P}_{merge} = F_i \vee D_i$,$\mathbf{IC}_{merge} = T \wedge i$, $\mathbf{DC}_{merge} = (T \wedge d) \vee (T \wedge d, d_1)$
- $\mathbf{PC}_{merge}$=$(true \wedge d > 10) \vee (true \wedge d < 20)$
- $countime_{merge} = 120$

Ainsi, même si la disjonction considère les opérandes comme des entités indivisibles, de nouveaux comportements apparaissent. En effet, l'état abstrait S_{merge} ne représente pas uniquement les états S_1 et S_2 mais représente également l'ensemble des états inatteignables que nous pouvons construire en combinant les composants du processeur au **PC**s. Exemple, l'état abstrait S_{merge} représente aussi l'état concret $S_{inatteignable}$ dont les paramètres sont :

- $\mathcal{SC}_{inatteignable}$: $\mathbf{P}_1 = F_i$,$\mathbf{IC}_2 = T \wedge i$, $\mathbf{DC}_2 = (T \wedge d, d_1)$
- $\mathbf{PC}_{inatteignable}$=$\mathbf{PC}_2$=$(true \wedge d < 20)$
- $countime_{inatteignable} = 120$

Ce type d'état engendre un ensemble de comportements (de traces d'exécution) qui ne sont pas cohérents avec la sémantique imposée par le processeur. La fusion dans ce cas est destructive du fait que nous

n'avons aucune information concernant les méfaits des comportements infaisables (trace d'exécution infaisables) ainsi que leurs impacts sur les temps calculés. Ainsi, remplacer un ensemble d'états concrets par un état abstrait en se basant sur l'équivalence de certains de leurs paramètres (états du pipeline ou des mémoires caches) est une solution certes adéquate pour éviter l'explosion combinatoire, cependant elle conduit à des résultats catastrophiques vis-à-vis de la précision des résultats calculés.

La solution que nous avons choisi pour atténuer ces effets destructeurs consiste à associer chaque fusion à un arbre symbolique construit grâce à une analyse avant. Autrement dit, à chaque fois qu'une possible fusion est détectée :

- 1) nous l'effectuons.
- 2) nous construisons à partir de l'état abstrait engendré S_{merge}, un arbre symbolique.
- 3) nous comparons cet arbre aux arbres symboliques que nous avions obtenus avant d'effectuer cette fusion.

5.6.9 Fusion contrôlée par l'impact dans le futur

Afin d'effectuer des fusions ayant un impact minime sur la précision des temps d'exécution calculés, nous avons développé un module de prédiction que nous avons associé à l'algorithme de fusionnement présenté dans la partie 5.6.5. Pour un fonctionnement optimal, le module de prédiction doit calculer l'impact de la fusion sur les exécutions suivantes (voir figure 5.6).

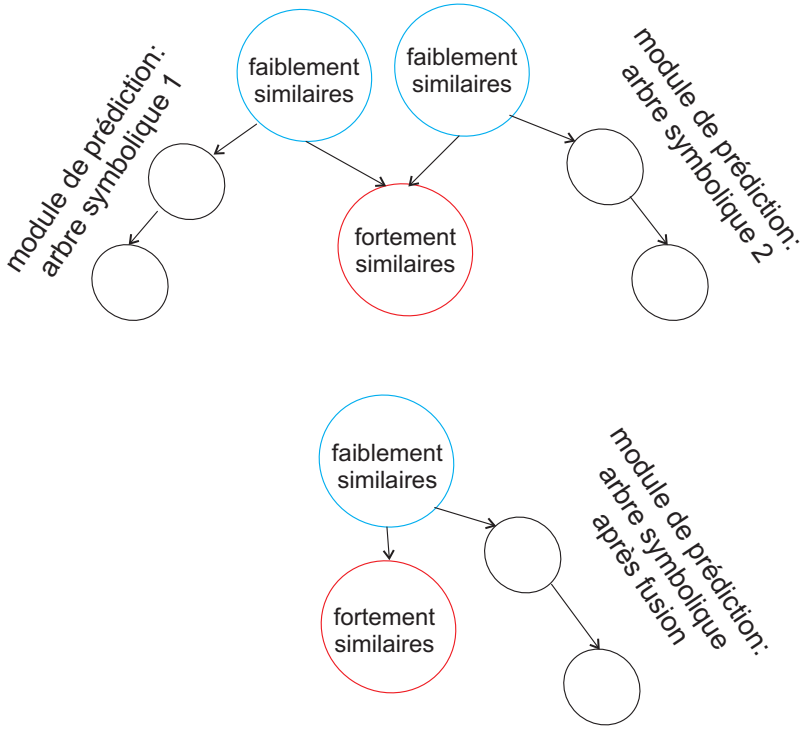


FIGURE 5.6: Fusion contrôlée par l'impact dans le futur

Ceci ne peut être déterminé que si nous pouvons prédire dynamiquement le comportement de l'exécution. Cette prédiction implique de générer d'autres états alors que la méthode de fusionnement a, par définition, la fonction inverse (de diminuer le nombre d'états). Ceci peut paraître contradictoire, néanmoins il est important de garder à l'esprit que les états générés durant la fusion ont une durée de vie

inférieure au temps que prend la méthode de fusionnement pour s'exécuter.

Fonctionnement de l'algorithme

- Partant de l'exécution symbolique d'une instruction du programme, un arbre symbolique est construit. Chaque branche de cet arbre aura une longueur équivalente à la profondeur du pipeline. Autrement dit, chaque branche est générée suite à un nombre de pas d'exécution symbolique égale au nombre d'étages du pipeline.
- Les états équivalents (fortement similaires) contenus dans l'arbre symbolique sont identifiés et fusionnés. Cette fusion n'introduit aucune perte de précision sur les temps calculés.
- Partant des états fusionnés, faire une analyse arrière. Remonter dans l'historique d'exécution de manière à identifier les états (faiblement similaires) susceptibles d'être fusionnés. Cette fusion introduit une perte de précision.
  A ce moment **le module de prédiction** est lancé :
  Son rôle est d'évaluer cette perte de précision dans un futur proche, ainsi il doit : (1) construire à partir de chacun des états candidats au fusionnement $e_1^b \ldots, e_n^b$ un arbre symbolique dont chaque branche aurait une longueur équivalente à λ pas d'exécution symbolique [7] (arbre symbolique 1 et arbre symbolique 2 de la figure 5.6). (2) Fusionner les états $e_\epsilon = e_1^b \sqcup \ldots \sqcup e_n^b$ et construire l'arbre d'exécution symbolique à partir de l'état résultant de la fusion précédente (e_ϵ). Cet arbre aura également des branches dont la longueur sera équivalente à λ pas d'exécution symbolique. (3) Calculer la différence entre les temps d'exécution estimés avant et après la fusion. Cette comparaison représente l'impact de la fusion sur les temps d'exécution.
- En fonction de la perte de précision tolérée, décider de valider ou d'invalider les fusions.

7. λ : représente le nombre de "pas d'exécution symbolique" devant être effectué pour mesurer l'impact d'une fusion d'états faiblement similaires.

5.6.10 Algorithme de fusionnement avec perte de précision maitrisée

L'algorithme de fusionnement, ci-dessous, identifie et fusionne les états faiblement similaires :

$Init\_state \leftarrow$ { (P = $\emptyset$, IC = DC = $\top$), PC = true, countime=0} ;
$\mathbf{s}_0 \leftarrow \{Init\_state\}$, $current\_states \leftarrow \mathbf{s}_0$;
$\mathcal{G} \leftarrow \{Init\_state\}$;
while $current\_states \neq \emptyset$ **do** (Propagation)
  $\mathbf{s}_d \notin current\_states$ (retirer $\mathbf{s}_d$ de $current\_states$) ;
  $\mathcal{G}(\mathbf{s}_d)$ = succ$_\rho$ ($\mathbf{s}_d$) (construire l'arbre symbolique résultant après ρ pas symboliques);
  $\mathcal{G} \leftarrow \mathcal{G}(\mathbf{s}_d) \cup \mathcal{G}$;
  $EQ = \{Eq_1, ..., Eq_m\}$ (construire les classes d'équivalence (voir partie 5.6.8.1) avec m : le nombre de classe d'équivalence d'états fortement similaires);
  foreach $Eq_i = \{e_1, \dots e_n\} \subset EQ$ **do** (Analyse avant)
    $S_{strong\_sim\_i} = \bigsqcup_{e_j \in Eq_i} e$ (fusionner les états fortement similaires) ;
    $S_{succ\_i} = \bigsqcup_{\mathtt{succ}(e)}$ (fusionner tous les successeurs des états fortement similaires) ;
    $e \notin Eq_i$ (retirer les états fusionnés de la classe d'équivalence Eq_i) ;
    MAJ $\mathcal{G}$ (Mettre à jour $\mathcal{G}$) ;
    foreach $s_v \in S_{strong\_sim\_i}$ **do** (Analyse arrière)
      if s_v *est valide* **then**
        $S_{weak\_sim\_i} = \{e_1^b, \dots, e_n^b\}$ (sélectionner les états précédents de s_v de telle façon que e_i^b soient faiblement similaires) ;
        Invalider s_v;
        if $S_{weak\_sim\_i} \neq \emptyset$ **then**
          $\mathcal{G}(e_1^b \cup ... \cup e_n^b)$ = succ$_\lambda(e_1^b) \cup$ ... $\cup$ succ$_\lambda(e_n^b)$(construire l'arbre symbolique partant des états candidats à la fusion e_k^b que nous obtenons suite à λ pas symboliques) ;
          $e_\epsilon = \bigsqcup_{k=1...n} e_k^b$ (fusionner tous les états faiblement similaires) ;
          $\mathcal{G}(e_\epsilon)$= succ$_\lambda(e_\epsilon)$ (construire l'arbre symbolique partant de e_ϵ que nous obtenons suite à λ pas symboliques) ;
          Comparer $\mathcal{G}(e_\epsilon)$ et $\mathcal{G}(e_1^b \cup ... \cup e_n^b)$ (estimer l'erreur entre les temps calculés avant la fusion des états et après la fusion de ces états- erreur introduite par la fusion);
          if $\mathcal{G}(e_\epsilon) \simeq \mathcal{G}(e_1^b \cup ... \cup e_n^b)$ **then**
            $e_{k=1...n}^b \notin S_{weak\_sim\_i} \wedge e_k^b \notin EQ$ (retirer les états fusionnés de $S_{weak\_sim\_i}$ et de toutes les classes d'équivalence dans EQ) ;
            MAJ $\mathcal{G}$ (Mettre à jour $\mathcal{G}$) ;
            else
              $e_\epsilon \neq \bigsqcup_{k=1...n} e_k^b$ (invalider la fusion des états faiblement similaires)
  foreach $\mathbf{s}_d \in \mathcal{G}$ **do** (Fin de l'analyse ?)
    if ($\mathbf{s}_d \notin S_{fin\_analyse}$) **then**
      $current\_states \leftarrow \mathbf{s}_d$;

avec :
$\mathcal{G}$: arbre symbolique.
$\mathbf{s}_d$: état final de $\mathcal{G}$ vis-à-vis du point d'exécution courant et d $\in$ [0..h] indice de l'état final.
$S_{fin\_analyse}$: ensemble des états finaux de $\mathcal{G}$ vis-à-vis de l'analyse (états finaux générés par la dernière instruction du programme).

Notons que la terminaison de cet algorithme est assurée par le nombre fini d'instructions du programme

analysé.

5.6.11 Intéret de l'analyse arrière

L'algorithme considère les états fortement similaires comme des points de départ. Cet algorithme effectue une analyse arrière pour identifier les états faiblement similaires et évalue l'impact de chaque fusion avant de l'effectuer grâce à une analyse avant (module de prédiction).

Notons que chaque état faiblement similaire est susceptible d'avoir plusieurs transitions sortantes. Durant l'analyse arrière nous identifions tous les états faiblement similaires. Ensuite à partir de ces états, le module de prédiction construit un arbre d'exécution symbolique qui contient tous les chemins d'exécution, à l'exception de ceux menant aux états fortement similaires. Cela assure que la fusion d'états faiblement similaires soit effectuée sans avoir d'impact sur les fusions faites en amont (fusions d'états fortement similaires). L'intérêt de l'analyse arrière est donc de mesurer l'impact d'une fusion sur toutes les traces d'exécution auquelles appartiennent ces états faiblement similaires. Ainsi, il est possible à partir d'un état fortement similaires de remonter dans l'historique d'exécution afin de fusionner toutes les traces d'exécution menant à cet état.

.

5.6.12 Illustration

Afin d'illustrer la méthode de fusionnement exposée précédemment, nous proposons de traiter une séquence d'instructions. La figure 5.7 montre le code C de la fonction acquisition ainsi que la traduction de ce code en instructions "assembleur PowerPC".

```
int N, MTable[N];

...

void acquisition(int measure)
{

  if(measure > MTable[N-1])
    MTable[N-1]=measure;



  else
    if(measure < MTable[0])
      MTable[0]=measure;

}
```

```
BEGIN
 1   lwz    %r1,off(@N)
 2   lwz    %r2,off(@measure)
 3   mulli  %r1,%r1,4
 4   addi   %r1,%r1,-4
 5   addi   %r1,%r1,off
 6   lwz    %r3,%r1(@MTable)
 7   cmp    %r2,%r3
 8   ble    $LN1
 9  lwz     %r4,off(@N)
 10  lwz    %r5,off(@measure)
 11  mulli  %r4,%r4,4
 12  addi   %r4,%r4,-4
 13  addi   %r4,%r4,off
 14  stw    %r5, %r4(@MTable)
 15  b      $EXIT
$LN1:
 16  lwz    %r6,off(@MTable)
 17  cmp    %r2,%r6
 18  bge    $EXIT
 19  stw    %r2,off(@MTable)
$EXIT:
  END
```

FIGURE 5.7: Un exemple de code en C et en assembleur PowerPC : la fonction `acquisition` est déclenchée periodiquement et rafraichie le premier ou le dernier élément du tableau `measure_table`

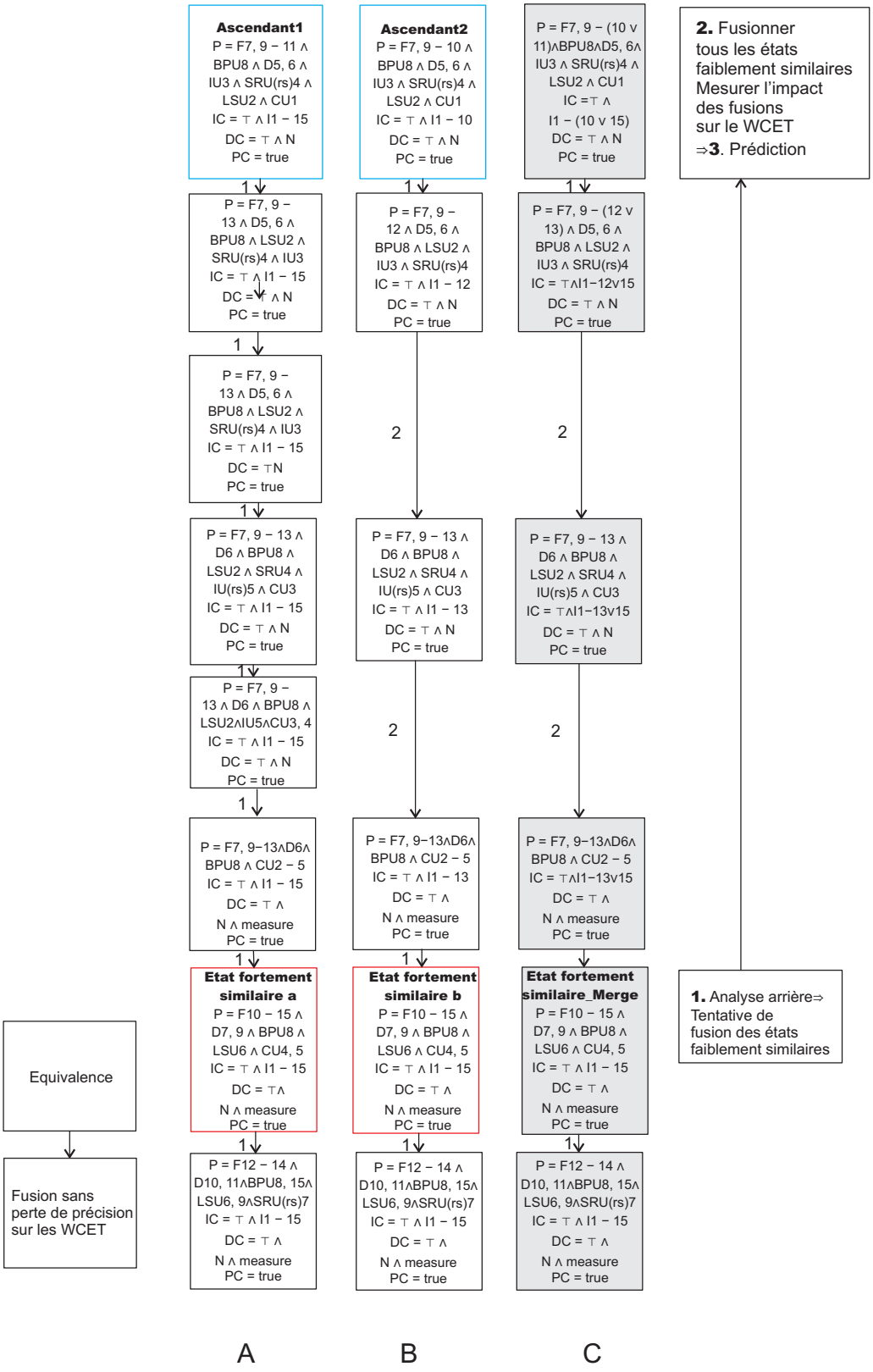


FIGURE 5.8: méthode de fusionnement

La figure 5.8 montre trois traces d'exécution (A,B,C). Les deux traces à gauche (A,B) sont générées durant l'exécution symbolique de la fonction `acquisition`. La trace (C), en gris, est la trace obtenue après la fusion des deux précédentes traces.

Durant l'exécution symbolique plusieurs traces convergent vers un état fortement similaire. Cet état est caractérisé par :

- (1) C'est un point de l'exécution qui rassemble plusieurs traces qui ont des historiques d'exécution différents (différent PCs).
- (2) Après cet état fortement similaire, toutes les traces d'exécution qui ont convergées vers lui, ont des comportements équivalents.

Dans la figure 5.8, l'état fortement similaire est caractérisé par : (1) un fetcher (F) qui contient les instructions de 10 à 15,(2) un dispatcher (D) qui dispatche les instructions 7 et 9, (3) une unité de Branchement (BPU) qui contient l'instruction 8, (4) une unité de lecture et d'écriture mémoire (LSU) qui exécute l'instruction 6, (5) une unité d'achèvement qui stocke les résultats des exécutions des instructions 4 and 5. (7) une mémoire cache d'instructions (IC) qui contient les instructions de 1 à 15, (8) une mémoire cache de données (DC) qui contient les données symboliques N et $measure$ et (9) un PC qui est à true. Lorsque, nous identifions des états fortement similaires nous les fusionnons. Cette fusion n'a aucun impact sur les temps d'exécution calculés à posteriori.

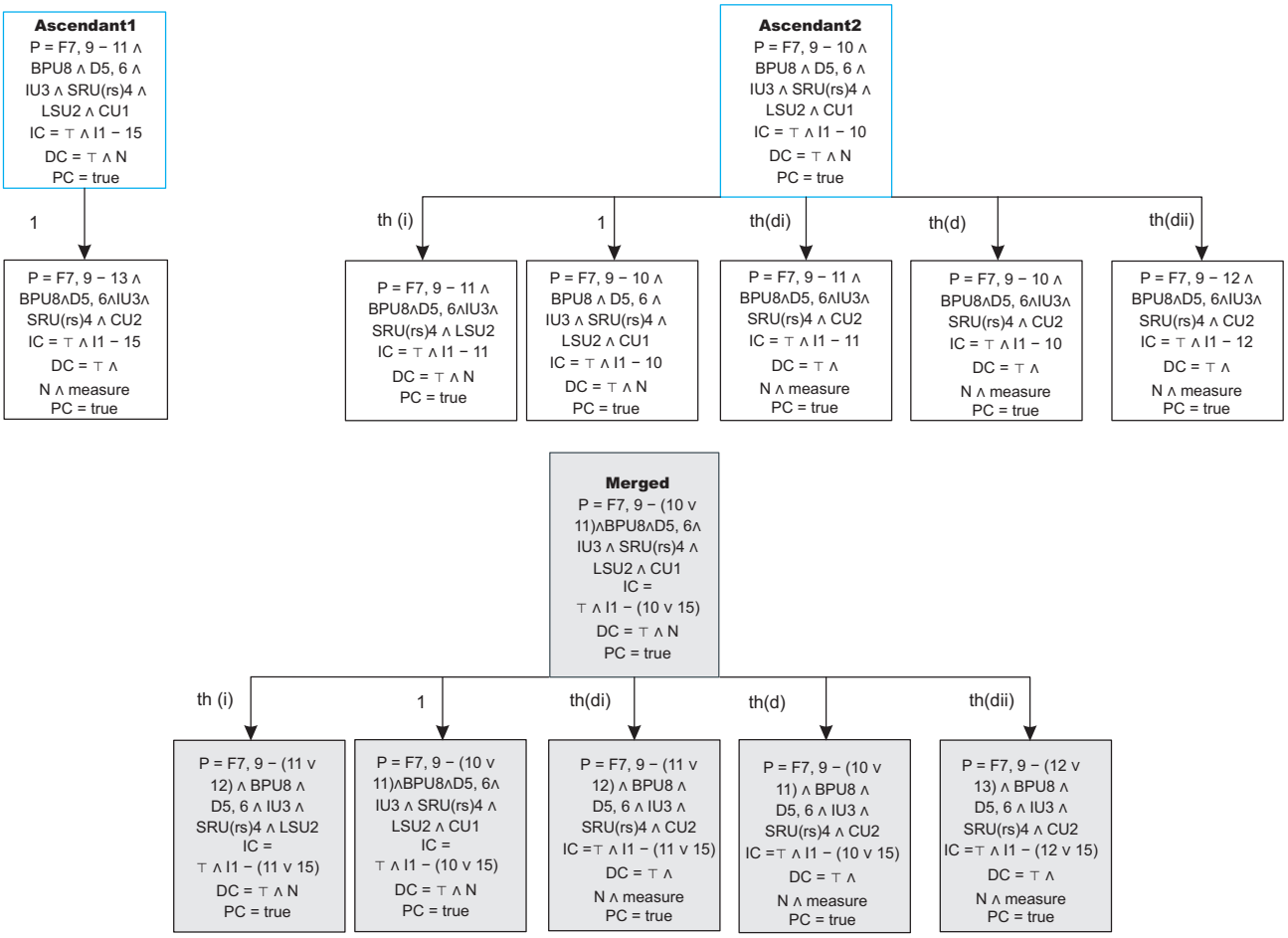


FIGURE 5.9: Module de prédiction

Dans le but de réduire le nombre des états générés plus efficacement tout en contrôlant la perte de précision sur les temps calculés, l'algorithme effectue également une analyse arrière.

Comme nous pouvons le constater sur la figure 5.8, l'analyse arrière commence par comparer les états qui précédent les états fortement similaires [8]. Lorsque les états faiblement similaires sont identifiés (la figure 5.9 montre la prédiction faite avant la fusion des états labélisés par $Ascendant$ dans la figure 5.8), nous construisons un arbre symbolique à partir de chacun d'entre eux (module de prédiction). Ensuite nous supposons que nous fusionnons ces états (les états faiblement similaires) et nous construisons un arbre symbolique de cet état résultant de la fusion ($Merged$). Cet arbre symbolique est ensuite comparé aux arbres précédents (les arbres construits par le module de prédiction) afin de mesurer la perte de précision que cette fusion introduit sur les temps calculés. En se basant sur le résultat de cette comparaison, une décision de fusionner ou de ne pas fusionner ces états est prise.

Notons que : (1) dans l'exemple le module de prédiction n'exécute qu'un seul pas d'exécution symbolique mais évidemment en pratique la prédiction est faite sur plusieurs pas d'exécution (approximativement égale à la profondeur du pipeline), (2) les arbres d'exécutions symboliques construits durant la phase de prédiction contiennent toutes les traces d'exécution à l'exception des traces qui mènent aux

8. Evidemment, nous tentons de fusionner les états ascendants qui n'appartiennent pas à la même trace d'exécution.

états fortement similaires.

5.6.13 Fusion assurant la terminaison de l'analyse

La fusion de traces d'exécution après l'identification des états fortement similaires permet de réduire fortement la taille de l'arbre symbolique. Ainsi, dans le cas général, cet algorithme est suffisant pour contenir le nombre d'états engendrés par l'analyse. Cependant, cet algorithme peut être confronté à certains cas "pathologiques" qui limiteraient ses performances, lorsque par exemple : nous disposons de peu de ressources mémoires (l'analyse est lancée sur une machine avec peu de mémoire vive) alors que le programme à analyser est de taille conséquente.

Dans ce type de contexte, la terminaison de l'analyse est assurée par un autre algorithme. Celui-ci identifie et fusionne directement les états faiblement similaires.

Cet algorithme est donc susceptible d'introduire beaucoup d'imprécision sur les temps calculés. En effet, il permet, au même titre que l'interprétation abstraite, de remplacer plusieurs états concrets par un état générique. Pour atténuer cet effet néfaste à l'analyse, la fusion des états faiblement similaires est faite par passes successives : nous commençons par fusionner les états faiblement similaires qui n'ont qu'un seul sous-composant différent. Si cette première passe permet de réduire le nombre d'états suffisamment pour poursuivre l'analyse, cet algorithme est stoppé. A présent, si cette première passe n'a pas suffisamment réduit le nombre d'états, alors une deuxième passe est lancée. Celle-ci fusionne les états faiblement similaires qui ont deux sous-composants différents. La fusion d'états se poursuit ainsi jusqu'à ce que la diminution du nombre d'états soit satisfaisante.

Cet algorithme évolue donc à chaque passe en relâchant une contrainte de similarité concernant un sous-composant particulier, jusqu'à ce que la consommation mémoire de l'analyse baisse en dessous d'un seuil imposé par les ressources mémoires disponibles.

Il est à noter que cet algorithme a été implémenté et testé pour garantir la terminaison de l'analyse quelque soit l'environnement dans lequel elle est effectuée. Cependant, durant ce travail de thèse cet algorithme n'a jamais été utilisé du fait que, concernant les programmes analysés, la fusion de trace était largement suffisante.

5.7 Conclusion

La précision des résultats est fortement corrélée à l'efficacité de la méthode de fusionnement. Celle-ci doit diminuer au maximum le nombre d'états générés sans pour autant nuire à la précision des résultats. Les rôles de la méthode de fusionnement sont donc, d'une part, d'éviter que le nombre des états engendrés par l'analyse ne devienne trop important et donc ingérable, et d'autre part, que les efforts fournis pour obtenir une estimation précise des temps d'exécution ne soient pas vain.

CHAPITRE 6

Résultats pratiques

6.1 Présentation des résultats

L'approche proposée dans ce rapport a été testée avec différentes structures de code.

Le but est de montrer que

- l'indéterminisme et les différents effets qu'introduisent les aléas de donnée, peuvent être pris en compte durant l'analyse. Cette prise en compte ne nécessite aucune étude au préalable.
- il est possible de calculer des résultats exacts ou très légèrement approximés de manière totalement formelle. C'est pourquoi, la phase de tests s'est limitée à des tests de faisabilité (tests effectués sur des petits programmes [1]).

Les tableaux suivant exposent les différents résultats obtenus suite à l'analyse de plusieurs structures de code. Ces analyses ont été effectuées sur un ordinateur dont les caractéristiques sont :

- processeur intel(R) Core 2.
- fréquence CPU : 2.83 GHz.
- capacité de la RAM : 2,00 GB.

Le premier tableau de résultats (voir tableau 6.1) est obtenu en débutant l'analyse avec des mémoires caches instructions et données vides (flashage des caches). Ainsi, toute récupération d'un nouveau bloc d'instructions ou de données résulte en un cache miss.

L'intérêt de ce premier test est de :

- Montrer que, dans ce cas précis, la durée de l'analyse est infime du fait que le nombre de traces d'exécution générés est petit.
- Prouver que notre approche s'adapte parfaitement à toutes les structures de code.
- Montrer que débuter une exécution avec des caches vides n'implique pas nécessairement que le pire scénario d'exécution soit généré (voir section 4.6.4).

Les programmes choisis pour effectuer nos tests sont de différents type : (1) des programmes linéaires sans instructions de saut (linéaire 1 et 2) , (2) des programmes contenant des instructions de branchement (indécidable et indécidable-pas-imbriqué), (3) des programmes contenant des instructions de branchement imbriquées (indécidable-imbriqué), (4) code contenant des boucles (linéaireloop et loop). Au début de chaque analyse toutes les entrées de ces programmes ont été initialisées à "inconnu", de façon à prendre en compte tous les comportements possibles de ces programmes.

1. Notons que notre méthode peut analyser des programmes plus grands (utilisés dans l'industrie), cependant compte tenu des ressources mémoires dont nous disposions, ceci aurait produit des résultats surapproximés. Ce qui n'est pas l'objet de ce travail de thèse.

TABLE 6.1: ” Mémoires caches vides”

Benchmarks	nb d’instruct.	nb scén. finaux	durée de l’analyse	conso. mém (KB)	BCET nb de cycles	WCET nb de cycles
linéaire 1	83	1	0.5s	12 684	136	136
linéaire 2	193	1	1s	21 000	303	303
indécidable	100	2	1s	17 260	159	160
indécidable-pas-imbriqué	113	4	2s	21 220	172	174
indécidable-imbriqué	34	3	0.5s	7 320	38	46
lineaireloop	101	1	0.5s	7 520	163	163
loop	106	1	0.5s	7560	180	180

Le tableau 6.2 présente, pour chaque séquence d’instructions, son pire (WCET) ainsi que son meilleur (BCET : best case execution time) temps d’exécution. La faible différence entre ces deux temps indique que les résultats obtenus sont très précis (voir figure 6.1). En effet, plus les résultats sont proches des valeurs réelles et plus l’intervalle [BCET,WCET] est petit.

Il est à noter que durant l’analyse la quantité de mémoire allouée oscille (augmente lors de la phase d’exécution symbolique et diminue durant la phase de fusionnement). Ainsi, dans les tableaux 6.2 et 6.1 la colonne "conso. mém (KB)" représente la quantité de mémoire allouée à un instant donné. Plus précisément, cette valeur indique le plus grand pic d’allocation mémoire constaté durant l’analyse.

TABLE 6.2: ”Mémoires caches initialisées a "inconnu"”

Benchmarks	nb d’instruct.	nb scén. finaux	durée de l’analyse	conso. mém (KB)	BCET nb de cycles	WCET nb de cycles
linéaire 1	83	124	2h 17min	993 000	115	218
linéaire 2	193	88	5h 23min	1 818 000	272	392
indécidable	100	106	5h 02min	1 623 000	131	241
indécidable-pas-imbriqué	113	119	6 h 30min	1 740 000	141	263
indécidable-imbriqué	34	40	14min	333 000	26	67
lineaireloop	101	45	2h 52min	362 384	138	182
loop	106	31	3min 39	95 000	150	181

La précision des résultats obtenus grâce à notre méthode s’explique comme suit :
Tant que la fusion d’états faiblement similaires n’est pas obligatoire, tous les résultats obtenus sont les valeurs exactes des temps d’exécution. Ainsi, nous obtenons non pas des estimations du pire et du meilleur temps d’exécution mais leurs valeurs respectives réelles.

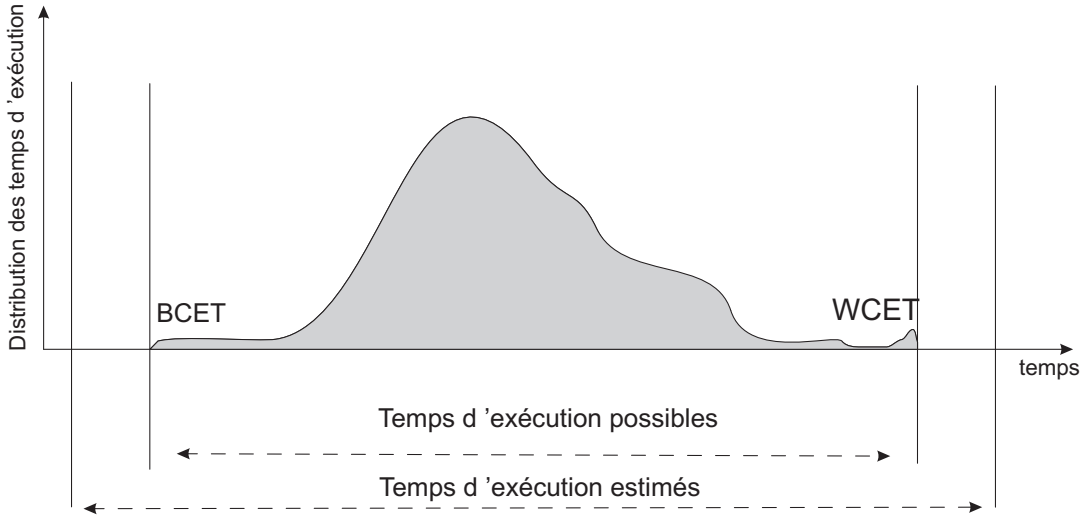


FIGURE 6.1: Différence entre temps d'exécution ([BCET,WCET]) possibles et estimés

La fusion d'états faiblement similaires, susceptible d'introduire de la perte de précision, n'est pas utilisée systématiquement. En effet, l'algorithme présenté dans la section 5.6.8, n'est lancé que lorsque le nombre d'état généré n'est plus compatible avec les ressources mémoires disponibles. Ainsi, lorsque le nombre d'états générés atteint une valeur seuil (imposée par les limites des ressources mémoires), les états faiblement similaires sont fusionnés de manière à diminuer plus efficacement la largeur de l'arbre symbolique.

6.2 Objectif de la phase de tests

Le principal objectif de la phase de test est de montrer qu'il est possible, de façon totalement formelle, d'obtenir des valeurs exactes de temps d'exécution. En effet, la précision des résultats fournis est dictée par la quantité de ressources mémoires disponibles. Ainsi, notre méthode adapte la précision des temps calculés à son environnement d'exécution. Autrement dit, si l'analyse est lancée sur une machine contenant peu de mémoire vive, la fusion d'état sera très agressive (fusion de traces et fusion assurant la terminaison de l'analyse) ce qui produira des résultats sur-approximés. A présent, si cette même analyse est lancée sur une machine contenant beaucoup de mémoire vive, la fusion d'états (fusion d'états fortement similaires uniquement) sera sans conséquence sur les temps d'exécution calculés. Etant donné que le coût des ressources mémoires (RAM) est de plus en plus faible, nous estimons que notre méthode est parfaitement en adéquation avec les demandes émergentes du monde industriel.

6.3 Evaluation des performances de la méthode

Les résultats obtenus indiquent que notre méthode s'adapte parfaitement à toutes les structures de code. Par ailleurs, l'écart assez faible entre le BCET et le WCET est une garantie de la précision des valeurs calculées. Ainsi, sur des séquences de code de l'ordre d'une centaine d'instructions aucune perte de précision n'est introduite. Au delà, si les ressources mémoires disponibles sont de l'ordre de 2GB, entre 100 et 200 instructions, la perte de précision reste raisonnable (entre 4 et 10 cycles). Pour des programmes plus grands, si une très grande précision des résultats est requise, la méthode nécessite plus de ressources mémoires.

Il est à noter que ces besoins en mémoire ainsi que la durée nécessaire à l'exécution de notre approche augmentent quasi-linéairement en fonction de la taille du programme analysé [2]. Cette durée peut donc être assez longue, néanmoins ceci n'est pas pénalisant du fait que cette analyse n'est effectuée qu'une seule fois pour chaque séquence d'instruction.

Notre méthode d'analyse nous a permis d'analyser les comportements de processeur de type PowerPC. Néanmoins, cette approche peut être étendue à d'autres architectures plus complexes. Le challenge est d'implémenter les comportements des processeurs sous forme de fonctions qui interagissent entre elles dans le but de simuler ces nouveaux comportements. Ainsi, cette méthode est extensible et peut donc être adaptée à différents types de processeurs. Par ailleurs, comme nous l'avons vu dans la section précédente, un des points forts de cette méthode réside aussi dans sa capacité à fournir des valeurs exactes. En effet, tant que les fusions ne concernent que les états fortement similaires aucune perte de précision n'est introduite et les valeurs obtenues sont donc les valeurs réelles des temps d'exécution.

2. Quasi-linéairement et non linéairement vu que concernant les instructions de branchement, leur première exécution fait doubler la largeur de l'arbre symbolique (les deux branches sont pris en compte)

CHAPITRE 7

Conclusion et perspectives

Nous avons présenté une méthode de détermination des pires temps d'exécution fiable et fournissant des résultats précis. Cette méthode a pour principal avantage d'être facile à mettre en œuvre en comparaison aux méthodes statiques. De plus, cette analyse qui se situe à mi-chemin entre les méthodes statiques et dynamiques fournit une couverture totale des traces d'exécution. Ce qui fait généralement défaut aux méthodes dynamiques.

Par ailleurs, cette approche est extensible, elle peut ainsi s'adapter à la sophistication croissante des processeurs. L'unique frein est de disposer d'un modèle exécutable reprenant tous les comportements de ces architectures complexes. Une fois ce modèle développé, plusieurs problèmes actuels peuvent être résolus, tel que : la prise en compte des effets propres aux processeurs modernes sans effectuer d'analyses supplémentaires. En effet, comme nous l'avons démontré dans la partie 4.6, notre méthode prend en compte systématiquement tous les aléas et anomalies temporelles susceptibles d'apparaitre durant une exécution. Ceci est dû au fait que ces effets sont considérés comme des comportements possibles du processeur (dépliage systématique de tous les scénarios atteignables). Ainsi, de façon naturelle nous obtenons une couverture totale des comportements prévus et imprévus du processeur. Cependant, l'énumération exhaustive des états accessibles est limitée par le phénomène d'explosion combinatoire. De multiples travaux sont en cours en vue de maîtriser cette explosion, exemple : (1) l'abstraction ou (2) la compression de graphes par des diagrammes de décision binaire (BDD). Le problème est que ces méthodes ne permettent pas de mettre en avant la notion de temporalité. C'est pourquoi, concernant notre approche, nous avons privilégié une méthode dynamique de fusion de trace qui puisse estimer la perte de précision introduite sur les temps d'exécution. Ceci, afin de garantir que les résultats produits sont au plus près de la réalité.

En résumé, le principal objectif de ce travail est de concevoir une méthode formelle et extensible qui produit une valeur exacte ou très légèrement approximée du pire temps d'exécution. Cependant, au stade de prototype, notre méthode nécessite quelques améliorations :

- De disposer d'un outil capable de valider la modélisation des comportements propres au processeur, dans le but d'obtenir rapidement des simulateurs respectant parfaitement les différents choix pris par l'architecture matérielle durant l'exécution. La partie exécution symbolique étant générique, nous pouvons imaginer développer un générateur de modèles de processeurs paramétrables. Ceci rendrait minime la durée d'adaptation de la méthode à une nouvelle architecture. Cette idée demeure parfaitement réalisable du fait que beaucoup de comportements ne diffèrent pas d'un processeur à un autre.

- Les méthodes de fusionnement peuvent aussi être améliorées, en raffinant les classes d'équivalence durant la construction des arbres symboliques. Ainsi, chacune de ces classes contiendraient moins de traces suceptibles d'avoir des comportements équivalents. Par conséquent, la méthode

s’exécuterait beaucoup plus rapidement.

Bibliographie

[1] Rapitime white paper. Technical Report, june 2008.

[2] ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. Bounding worst-case instruction cache performance. In *In IEEE Real-Time Systems Symposium* (1994), pp. 172–181.

[3] BENHAMAMOUCH, B., AND MONSUEZ, B. Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model. *IMECS 2009 . The International Multi-Conference of Engineers and Computer Scientists , Special Session : Design, Analysis and Tools for Integrated Circuits and Systems (DATICS), Hong Kong* (march 2009).

[4] BENHAMAMOUCH, B., AND MONSUEZ, B. Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model (extented version). *International Journal of Design, Analysis and Tools for Circuits and Systems, Vol. 1, No. 1*, (November 2009).

[5] BENHAMAMOUCH, B., MONSUEZ, B., AND VÉDRINE, F. Computing wcet using symbolic execution. *Second International Workshop on Verification and Evaluation of Computer and Communication Systems. Leeds GB* (july 2008).

[6] BERNAT, G., COLIN, A., AND PETTERS, S. pwcet : A tool for probabilistic worst-case execution time analysis of real-time systems. Tech. rep., 2003.

[7] BERNAT, G., COLIN, A., AND PETTERS, S. M. Wcet analysis of probabilistic hard real-time systems. In *In Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002* (2002), pp. 279–288.

[8] BLIEBERGER, J., FAHRINGER, T., AND SCHOLZ, B. Symbolic cache analysis for real-time systems. *Real-Time Systems 18* (1999), 181–215.

[9] BÖRGER, E., AND STÄRK, R. Abstract state machines. a method for high-level system design and analysis. Tech. rep., March 2003.

[10] BURGUIÈRE, C., AND ROCHANGE, C. History-based schemes and implicit path enumeration. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis* (Dagstuhl, Germany, 2006), F. Mueller, Ed., Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany.

[11] CASSÉ, H., AND SAINRAT, P. Otawa, a framework for experimenting wcet computations. *ERTS'06* (janvier 2006).

[12] CHRISTIAN FERDINAND, R. H., AND WILHELM, R. Analyzing the worst-case execution time by abstract interpretation of executable code. *Springer Berlin / Heidelberg. Automotive Software – Connected Services in Mobile Networks 4147* (2006), 1–14.

[13] COEN-PORISINI, A., DENARO, G., GHEZZI, C., AND PEZZÉ, M. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2001), ACM, pp. 142–151.

[14] COLIN, A., AND PUAUT, I. A modular and retargetable framework for tree-based wcet analysis. In *In Proc. of the 13th Euromicro Conference on Real-Time Systems* (2001), pp. 37–44.

[15] CORTI, M., BREGA, R., AND GROSS, T. Approximation of worst-case execution time for preemptive multitasking systems. In *LCTES '00 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (London, UK, 2001), Springer-Verlag, pp. 178–198.

[16] COUSOT, P. Semantic foundations of program analysis. In *Program Flow Analysis : Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, ch. 10, pp. 303–342.

[17] COUSOT, P., AND COUSOT, R. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[18] DARRINGER, J. A. The application of program verification techniques to hardware verification. *Annual ACM IEEE Design Automation Conference* (1988), 376–381.

[19] ENGBLOM, J. *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Uppsala University, April 2002.

[20] ENGBLOM, J., ERMEDAHL, A., SJÖDIN, M., GUSTAVSSON, J., AND HANSSON, H. Towards industry strength worst-case execution time analysis, 1999.

[21] ENGBLOM, J., ERMEDAHL, A., AND STAPPERT, F. Validating a worst-case execution time analysis method for an embedded processor. Tech. Rep. 2001-030, 2001.

[22] ENGBLOM, J., AND JONSSON, B. Processor pipelines and their properties for static wcet analysis. In *Proceedings of EMSOFT 02 : Second International Conference on Embedded Software, volume 2491 of Lecture Notes in Computer Science* (2002), Springer-Verlag, pp. 334–348.

[23] ERMEDAHL, A., AND GUSTAFSSON, J. Deriving annotations for tight calculation of execution time. In *Euro-Par '97 : Proceedings of the Third International Euro-Par Conference on Parallel Processing* (London, UK, 1997), Springer-Verlag, pp. 1298–1307.

[24] FERDINAND, C., KASTNER, D., LANGENBACH, M., MARTIN, F., SCHMIDT, M., SCHNEIDER, J., THEILING, H., THESING, S., AND WILHELM, R. Run-time guarantees for real-time systems – the USES approach. In *GI Jahrestagung* (1999), pp. 410–419.

[25] FERDINAND, C., AND WILHELM, R. On predicting data cache behavior for real-time systems. In *LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (London, UK, 1998), Springer-Verlag, pp. 16–30.

[26] HEALY, C. A., ARNOLD, R. D., MUELLER, F., HARMON, M. G., AND WALLEY, D. B. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput. 48*, 1 (1999), 53–70.

[27] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In *PLDI '88 : Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM, pp. 35–46.

[28] IAIN BATE, PHILIPPA CONMY, T. K. J. M. Use of modern processors in safety-critical applications. *The Computer Journal 44* (2001), 531–543.

[29] JEAN SOUYRIS, ERWAN LE PAVEC, G. H. V. J. G. B., AND HECKMANN, R. Computing the worst case execution time of an avionics program by abstract interpretation.

[30] KING, J. C. Symbolic execution and program testing. *Communications of the ACM (Association for Computing Machinery) 19*, 7 (July 1976).

[31] LI, X., ROYCHOUDHURY, A., AND MITRA, T. Modeling out-of-order processors for wcet analysis. *Real-Time Syst. 34*, 3 (2006), 195–227.

[32] LI, Y.-T. S., MALIK, S., AND WOLFE, A. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS '95 : Proceedings of the 16th IEEE Real-Time Systems Symposium* (Washington, DC, USA, 1995), IEEE Computer Society, p. 298.

[33] LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., PARK, K., MOON, S.-M., AND KIM, C. S. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng. 21*, 7 (1995), 593–604.

[34] LINDGREN, M., HANSSON, H., AND THANE, H. Using measurements to derive the worst-case execution time. In *RTCSA '00 : Proceedings of the Seventh International Conference on Real-Time Systems and Applications* (Washington, DC, USA, 2000), IEEE Computer Society, p. 15.

[35] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (1973), 46–61.

[36] LUNDQVIST, T. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Department of Computer Engineering CHALMERS UNIVERSITY OF TECHNOLOGY, Goteborg, Sweden, 2002.

[37] LUNDQVIST, T., AND STENSTRÖM, P. Integrating path and timing analysis using instruction-level simulation techniques. In *LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (London, UK, 1998), Springer-Verlag, pp. 1–15.

[38] LUNDQVIST, T., AND STENSTRÖM, P. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Syst. 17*, 2-3 (1999), 183–207.

[39] LUNDQVIST, T., AND STENSTRÖM, P. A method to improve the estimated worst-case performance of data caching. In *RTCSA '99 : Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications* (Washington, DC, USA, 1999), IEEE Computer Society, p. 255.

[40] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99 : Proceedings of the 20th IEEE Real-Time Systems Symposium* (Washington, DC, USA, 1999), IEEE Computer Society, p. 12.

[41] MAUBORGNE, L., AND RIVAL, X. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming (ESOP'05)* (2005), M. Sagiv, Ed., vol. 3444 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 5–20.

[42] MONSUEZ, B., AND VÉDRINE, F. Modélisation d'une plate-forme multi-processeurs pour calculer et contraindre des temps d'exécution. In *Contrat ANR Applications Parallèles pour l'Embarqué* (2007).

[43] MUELLER, F. *Static cache simulation and its applications*. PhD thesis, Departement of computer Computer Sciences, Florida State University, july 1994.

[44] OTTOSSON, G., AND SJODIN, M. Worst case execution time analysis for modern hardware architectures. In *In : Proc. of ACM SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems* (1997), pp. 47–55.

[45] PARK, C. Y. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst. 5*, 1 (1993), 31–62.

[46] PARK, C. Y., AND SHAW, A. C. Experiments with a program timing tool based on source-level timing schema. *Computer 24*, 5 (1991), 48–57.

[47] PAUL LOKUCIEJEWSKI, HEIKO FALK, M. S. P. M. H. T. Influence of procedure cloning on wcet prediction. *International Conference on Hardware Software Codesign archive. Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis. Salzburg, Austria SESSION : Embedded software* (2007), 137–142.

[48] PREANU, C. S., AND VISSER, W. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf. 11*, 4 (2009), 339–353.

[49] PUSCHNER, P., AND BURNS, A. A review of worst-case execution-time analysis. *Journal of Real-Time Systems 18*, 2/3 (May 2000), 115–128.

[50] PUSCHNER, P., AND KOZA, C. Calculating the maximum, execution time of real-time programs. *Real-Time Syst. 1*, 2 (1989), 159–176.

[51] PUSCHNER, P. P., AND SCHEDL, A. V. Computing maximum task execution times — a graph-basedapproach. *Real-Time Syst. 13*, 1 (1997), 67–91.

[52] Mpc603e ec603e risc microprocessors user's manual with supplement for powerpc 603^{TM} microprocessor. Technical Report, 1997.

[53] Powerpc 603 risc microprocessors technical summary. Technical Report, 1994.

[54] REINHOLD HECKMANN, C. F. Worst case execution time prediction by static program analysis. *Parallel and Distributed Processing Symposium. Proceedings. 18th International Volume* (April 2004), 125–134.

[55] RUF, J., HOFFMANN, D., GERLACH, J., KROPF, T., ROSENSTIEHL, W., AND MUELLER, W. The simulation semantics of systemc. In *DATE '01 : Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 64–70.

[56] SEBEK, F., AND GUSTAFSSON, J. Determining the worst case instruction cache miss-ratio. In *In Proceedings of Workshop On Embedded System Codesign (ESCODES'02* (2002), p. 2002.

[57] STEWART, D. B. Measuring execution time and real-time performance. *Embedded Systems Conference. Boston* (september 2006).

[58] THEILING, H., FERDINAND, C., SAARBRUCKEN, A. A. I. G., AND WILHELM, R. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems 18* (1999), 157–179.

[59] THESING, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, November 2004.

[60] TSUN STEVEN LI, Y., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *in Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1995), pp. 456–461.

[61] TSUN STEVEN LI, Y., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *in Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1995), pp. 456–461.

[62] WHITE, R. T., HEALY, C. A., WHALLEY, D. B., MUELLER, F., AND HARMON, M. G. Timing analysis for data caches and set-associative caches. In *RTAS '97 : Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)* (Washington, DC, USA, 1997), IEEE Computer Society, p. 192.

[63] WHITE, R. T., MUELLER, F., HEALY, C., WHALLEY, D., HARMON, M., AND HALANG, A. Timing analysis for data and wrap-around fill caches. *Real-Time Systems 17* (1999), 209–233.

[64] WILHELM, R. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In *In Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937* (2004).

[65] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. 7*, 3 (2008), 1–53.

[66] WILHELM, R., AND WACHTER, B. Abstract interpretation with applications to timing validation. In *CAV '08 : Proceedings of the 20th international conference on Computer Aided Verification* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 22–36.

[67] WILHELM, S. Efficient analysis of pipeline models for wcet computation. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis* (Dagstuhl, Germany, 2007), R. Wilhelm, Ed., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[68] ZHANG, N., BURNS, A., AND NICHOLSON, M. Pipelined processors and worst case execution times. *Real-Time Syst. 5*, 4 (1993), 319–343.

[69] ZHANG, Y. Evaluation of methods for dynamic time analysis for cc-systems ab. Technical Report, Mälardalen University, August 2005.