RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Calcul de temps d'exécution pire cas pour un processeur superscalaire à exécution non ordonnée

Jonathan Barre, Cédric Landet, Christine Rochange, Pascal Sainrat

Université Paul Sabatier,
Institut de Recherche en Informatique de Toulouse
31062 TOULOUSE cedex 9 - France
{barre,landet,rochange,sainrat}@irit.fr

Résumé
Les systèmes temps-réel intègrent de plus en plus fréquemment des processeurs haute-performance. Il en résulte que l'estimation du temps d'exécution pire cas ou WCET (*Worst-Case Execution Time*) d'applications temps-réel strict est de plus en plus difficile. Lorsque l'on considère une architecture moderne, estimer le temps d'exécution d'un simple bloc de base n'est pas trivial à cause des anomalies temporelles qui peuvent résulter de l'ordonnancement dynamique des instructions, et de l'influence sur l'état du pipeline des blocs de base exécutés avant et après. Récemment, un modèle de l'exécution d'un bloc de base dans un processeur à pipeline et à exécution dans le désordre s'appuyant sur des graphes a été proposé [10]. Dans cet article, nous étendons ce modèle pour exprimer le parallélisme d'instructions, de manière à pouvoir analyser des processeurs superscalaires disposant d'unités fonctionnelles multiples. Des résultats de simulation montrent que ce modèle étendu estime les temps d'exécution pire cas avec une précision acceptable, et ce en considérant un processeur réaliste. Ces résultats donnent aussi une idée de la complexité du modèle en termes de temps d'analyse.

Mots-clés : temps-réel, temps d'exécution pire cas, processeur.

1. Introduction

Ordonnancer des tâches temps-réel ayant des échéances strictes nécessite de connaître leurs temps d'exécution pire cas (WCETs). La recherche de stratégies pour estimer le WCET d'un programme a donné lieu à de nombreux travaux de recherche ces dix dernières années. Les méthodes basées sur des mesures ne sont pas suffisantes si l'on veut calculer une borne garantie supérieure du WCET. C'est pourquoi des techniques sûres, reposant sur une analyse statique du code, ont été développées. Parmi elles, la méthode IPET (*Implicit Path Enumeration Technique*) [9] travaille sur le code objet du programme et semble être la plus largement utilisée. Cette méthode exprime le temps d'exécution du programme comme la somme des temps d'exécution des blocs de base qui le composent pondérés par leurs nombres d'exécutions respectifs sur le chemin d'exécution. Le WCET est alors obtenu en maximisant cette expression sous un ensemble de contraintes qui portent sur les nombres d'exécutions des blocs (ce problème est résolu à l'aide d'algorithmes de programmation linéaire en nombres entiers). Certaines contraintes expriment la structure du graphe de contrôle de flot (CFG) et d'autres les résultats d'une analyse de flot préliminaire (bornes de boucles, chemins impossibles, ...). Les temps d'exécution individuels des blocs de base sont issus d'une analyse temporelle qui prend

en compte les caractéristiques du processeur cible. Le travail présenté dans cet article concerne plus particulièrement cette phase d'analyse temporelle.
Le modèle du processeur utilisé pour estimer le WCET doit être précis. Alors que les premiers processeurs à pipeline pouvaient être analysés à l'aide de simples tables de réservation [5][7], les architectures plus récentes nécessitent des modèles plus complexes. Il faut prendre en compte l'exécution en pipeline, superscalaire, dans le désordre, ainsi que la prédiction de branchements et les mémoires caches. Ces deux derniers éléments ont été traités dans plusieurs articles [2][3][5][6][11] mais la modélisation du cœur d'exécution nécessite encore des efforts de recherche, comme nous le verrons au paragraphe 2. Li *et al.* ont proposé une manière de représenter le comportement d'un processeur pipeliné à exécution non ordonnée [10]. Leur solution prend en compte les interférences temporelles entre blocs qui ont été identifiées comme des *effets temporels longs* par Engblom [4]. Cependant, leur modèle n'exprime pas le parallélisme d'instructions : ils ne considèrent qu'un pipeline *scalaire* dont chaque étage ne traite qu'une seule instruction à chaque cycle. Par ailleurs, le parallélisme potentiel des ressources (par exemple la présence de plusieurs unités fonctionnelles du même type) n'est pas représenté. Dans cet article, nous étendons leur modèle pour pouvoir exprimer le parallélisme et le prendre en compte lors de l'évaluation du WCET d'un bloc de base.
L'article est organisé de la manière suivante. Le pararaphe 2 décrit le modèle de Li et recense les architectures que ce modèle ne permet pas d'exprimer. Dans le paragraphe 3, nous proposons une extension de ce modèle qui permet de représenter des processeurs superscalaires (à exécution non ordonnée) réalistes. Des résultats sont donnés dans le paragraphe 4 et nous concluons dans le paragraphe 5.

2. Etat des lieux

2.1. Le modèle de Li : un processeur scalaire à exécution non ordonnée

Li *et al.*proposent de représenter l'exécution d'un bloc de base par un processeur scalaire à exécution non ordonnée par un *graphe d'exécution* qui exprime les dépendances entre instructions ainsi que les conflits possibles pour l'utilisation de certaines ressources [10]. Les instructions qui peuvent être exécutées avant et après le bloc de base, et leur impact possible sur l'exécution du bloc sont également représentés. Le graphe est ensuite analysé pour calculer des bornes inférieures et supérieures sur les dates auxquelles les différentes instructions sont traitées par les étages du pipeline. Le WCET du bloc de base est la date au plus tard de retrait du pipeline de sa dernière instruction. Dans ce paragraphe, nous donnons un aperçu du modèle et des algorithmes mis en œuvre pour déterminer les dates de traitement.

Graphe d'exécution. L'exécution d'un bloc de base dans le pipeline est modélisée par un ensemble de nœuds dont chacun représente le traitement d'une instruction par un étage du pipeline. Les nœuds sont reliés par des arcs qui expriment les dépendances entre nœuds. Les arcs *pleins* expriment : (i) l'ordre du programme (par exemple, les intructions sont chargées dans l'ordre du programme, elles sont insérées dans le tampon de réordonnancement – dont la capacité est limitée – dans l'ordre du programme) ; (ii) la structure du pipeline (par exemple, les instructions sont chargées avant d'être décodées) ; (iii) les dépendances de données (une instruction doit attendre que ses opérandes soient prêts avant de pouvoir être exécutée). A côté des arcs pleins, des arcs *pointillés* non orientés expriment une contention possible entre nœuds pour l'usage d'une ressource (par exemple, deux instructions nécessitent la même unité fonctionnelle et peuvent être exécutées dans le désordre). La figure 1 donne un exemple de graphe d'exécution pour un processeur avec un pipeline de 5 étages, une file d'instructions à 2 entrées

I_1 : `mult r6,r10,4`
I_2 : `mult r1,r10,r1`
I_3 : `sub r6,r6,r2`
I_4 : `mult r4,r8,r4`
I_5 : `add r1,r1,r4`

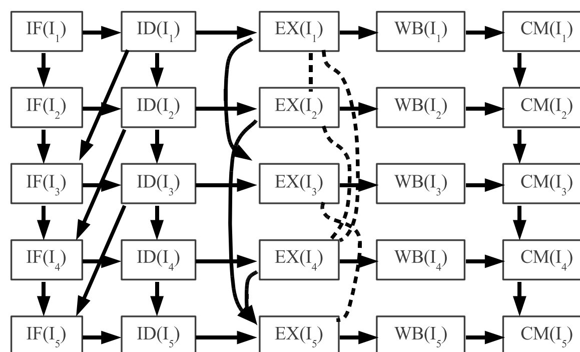

FIG. 1 – Exemple de graphe d'exécution.

et un tampon de réordonnancement à 4 entrées. Cet exemple est extrait de [8] mais le processeur modélisé est légèrement différent : nous supposons ici qu'il est muni d'un dispositif de renvoi des résultats.

Dates au plus tôt et au plus tard. Le modèle est destiné à prendre en compte des unités fonctionnelles à latence variable, exprimée sous forme d'intervalle. A partir des intervalles de latence et des dépendances entre nœuds exprimées dans le graphe d'exécution, on calcule une valeur au plus tôt et au plus tard pour les dates à laquelle un nœud est prêt (*ready time*), commence (*start time*) et se termine (*finish time*). La première instruction du bloc de base – I1 – est supposée être chargée à la date 0.
L'algorithme d'estimation des dates au plus tôt et au plus tard comporte trois étapes. Tout d'abord, les dates au plus tard pour chacun des nœuds sont calculées en considérant tous les concurrents possibles (un *concurrent* est un nœud susceptible d'entrer en compétition pour l'usage de la même ressource) et sont propagées aux nœuds sucesseurs. Ensuite, les dates au plus tôt sont estimées de la même manière. Finalement, les paires de nœuds qui ne peuvent pas être contemporains, c'est-à-dire avec des intervalles de dates disjoints, sont identifiées et les nœuds sont déclarés "non concurrents". Ces trois étapes sont répétées jusqu'à stabilisation ou à concurrence d'un nombre d'itérations prédéfini. Plus de détails sur l'algorithme peuvent être trouvés dans la thèse de Li [8].

Impact des blocs antérieurs et postérieurs. Un *prologue* (respectivement un *épilogue*) d'un bloc de base est une séquence d'instructions qui peut être exécutée juste avant (respectivement après) le bloc (cette séquence peut être extraite du graphe de contrôle de flot). Il contient autant d'instructions que la file d'instructions et le tampon de réordonnancement réunis (il s'agit du nombre d'instructions pouvant être actives dans le processeur à un instant donné). Des hypothèses conservatrices sont faites sur les instructions qui peuvent précéder un prologue. Bien entendu, un bloc de base peut avoir plusieurs paires $(\mathtt{prologue}, \mathtt{épilogue})$: son WCET est le maximum des temps calculés pour chacune des paires possibles.
Les nœuds du prologue et de l'épilogue sont inclus dans le graphe d'exécution. On traite en premier lieu les nœuds du prologue dont part un chemin (*via* des arcs pleins) menant à IF(I1). On calcule, pour ces nœuds, les bornes supérieures des dates auxquelles ils sont prêts, commencent et se terminent (en soustrayant de 0 – date de IF(I1) – la somme des latences maxi-

males sur le chemin). Les dates au plus tôt et au plus tard des autres nœuds du prologue, puis de ceux de l'épilogue, sont alors calculées comme expliqué ci-dessus. Enfin, le bloc lui-même est analysé pour prendre en compte les informations obtenues pour le prologue et pour l'épilogue. Encore une fois, on trouvera plus de détails dans [10] et [8].

Recouvrement de blocs. Deux blocs exécutés à la suite l'un de l'autre dans le pipeline se recouvrent partiellement. Aussi, le temps d'exécution d'une séquence de deux blocs est en général plus court que la somme de leurs temps d'exécution individuels. Li définit le *coût* d'un bloc de base comme le temps entre la fin du bloc (date à laquelle sa dernière instruction sort du pipeline) et la fin du bloc précédent. Il montre comment le coût pire-cas peut être évalué de manière sûre à partir des dates au plus tôt et au plus tard.

2.2. Parallélisme d'instructions et exécution superscalaire

Le modèle proposé par Li constitue une base solide pour l'analyse de programmes exécutés sur des processeurs haute-performance. Il permet de prendre en compte l'exécution pipelinée et dans le désordre, tout en tenant compte de possibles anomalies temporelles [13] quand les unités fonctionnelles ont des latences variables. Le modèle prend aussi en compte les effets de blocs antérieurs et postérieurs sur le chemin d'exécution. Enfin, il peut être combiné avec l'analyse d'un cache d'instructions et d'un prédicteur de branchements [10].

Toutefois, ce modèle ignore une caratéristique majeure des processeurs actuels : la possibilité de traiter plusieurs instructions en parallèle, connue sous le nom d'*exécution superscalaire*. En effet, les seules contraintes de précédence que peut exprimer le graphe d'exécution sont strictes : les deux noeuds doivent s'exécuter l'un après l'autre, que ce soit dans un ordre spécifié (ordre du programme, ordre du pipeline, dépendance de donnée) ou dans n'importe quel ordre (accès à une ressource commune), mais pas simultanément. Ceci implique qu'un étage du pipeline ne peut traiter qu'une seule instruction par cycle. Le modèle ne permet pas non plus de représenter des unités fonctionnelles multiples (plusieurs unités identiques) : les arcs pointillés empêchent des nœuds concurrents d'utiliser une ressource partagée simultanément. Or, les processeurs modernes ont souvent des unités en plusieurs exemplaires et plusieurs opérations identiques (par exemple plusieurs additions entières) peuvent être exécutées en parallèle.

Parce qu'il ne permet pas d'exprimer le parallélisme d'instructions, le modèle de Li ne peut pas être utilisé pour représenter des processeurs haute-performance réalistes. Dans cet article, nous proposons des extensions qui élargissent son champ d'application dans ce sens.

2.3. Travaux connexes

Comme nous l'avons mentionné à plusieurs reprises, évaluer le WCET des blocs de base d'un programme se heurte à deux difficultés bien connues : les anomalies temporelles liées aux unités fonctionnelles à latence variable (la latence la plus longue ne conduit pas forcément au WCET) [13] et les interférences entre blocs, également appelées *effets temporels longs* [4]. Ceci a conduit certains auteurs à recommander d'utiliser des processeurs scalaires à exécution ordonnée lorsqu'un WCET fiable doit être calculé. Plusieurs approches ont été proposées pour contourner la difficulté. Dans l'architecture VISA, le code est annoté avec des estimations de temps [1]. Le processeur commute dans un mode d'exécution dans l'ordre – permettant une estimation *sûre* du WCET – dès que le temps d'exécution réel dépasse l'estimation. Récemment, nous avons défini un pipeline qui inclut un mécanisme de régulation du chargement des instructions qui impose une certaine distance entre blocs de base dans le pipeline [14]. Ce mécanisme empêche les interférences entre blocs distants.

A notre connaissance, il y a eu très peu de tentatives de modéliser des processeurs superscalaires à exécution non ordonnée. A côté du travail de Li *et al.* qui a été décrit au paragraphe 2.1, seul l'outil commercial *aiT* développé par la société AbsInt prend en compte ce type d'architecture. Cet outil s'appuie sur des techniques d'interprétation asbtraite pour déterminer tous les états possibles du pipeline au début d'un bloc de base. Malheureusement, aucune description détaillée du modèle n'a été publiée et les algorithmes mis en œuvre ne peuvent pas être exploités par la communauté de recherche.

3. Modélisation du parallélisme d'instructions

Pour être applicable à des processeurs modernes, le modèle de Li doit être étendu pour permettre d'exprimer le parallélisme d'instructions et les deux situations suivantes : (i) deux nœuds doivent être exécutés dans un certain ordre ou éventuellement en parallèle ; (ii) plusieurs nœuds peuvent utiliser simultanément une ressource présente en plusieurs exemplaires. Dans cette partie, nous montrons comment ces caractéristiques peuvent être intégrées au modèle.

3.1. Exécution superscalaire

Un arc plein entre deux nœuds $E(I_a)$ et $E(I_b)$ signifie que les instructions I_a et I_b doivent être traitées par l'étage E dans l'ordre spécifié. En d'autres termes, la date de début de $E(I_b)$ est supérieure ou égale à la date de terminaison de $E(I_a)$, elle-même égale à la date de départ de $E(I_a)$ augmentée de la latence de l'étage E. Pour exprimer que $E(I_a)$ et $E(I_b)$ doivent être exécutés dans cet ordre mais *éventuellement en parallèle*, nous suggérons d'utiliser des arcs pleins *barrés*. Un arc barré entre deux nœuds $E(I_a)$ et $E(I_b)$ signifie alors que la date de départ de $E(I_b)$ est supérieure ou égale à la date de départ de $E(I_a)$. Le degré de parallélisme de l'étage E est exprimé par des arcs pleins, de la même manière qu'on représente la capacité finie d'une file.

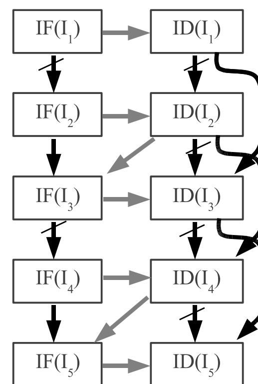

FIG. 2 – Un graphe d'exécution exprimant un traitement superscalaire.

La figure 2 montre un exemple de graphe d'exécution étendu qui modélise la partie frontale du processeur. Dans cet exemple, deux instructions, soit une ligne du cache d'instructions, peuvent être lues à chaque cycle : ceci est modélisé par les arcs barrés entre les nœuds $IF(I_1)$ et $IF(I_2)$, et entre $IF(I_3)$ et $IF(I_4)$. Les arcs non barrés entre $IF(I_2)$ et $IF(I_3)$, et entre $IF(I_4)$ et $IF(I_5)$, indiquent qu'une seule ligne de cache peut être lue à chaque cycle. Par ailleurs, compte tenu du fait que le processeur ne charge que des lignes entières (pas de chargement partiel), les arcs modélisant la capacité de la file de chargement sont différents de ceux présents sur la figure 1. Par exemple,

le décodage de I_1 ne suffit pas pour que I_3 puisse être chargée : il faut qu'il y ait également de la place pour I_4. C'est pourquoi il y a un arc entre $ID(I_2)$ et $IF(I_3)$. Par ailleurs, l'étage de décodage est supposé avoir une capacité de deux instructions par cycle. Cependant, il n'est pas possible de déterminer quelles paires d'instructions sont décodées ensemble : cela dépend du comportement du reste du pipeline (il peut arriver qu'une seule instruction soit décodée pendant certains cycles). C'est pourquoi les arcs barrés indiquent un traitement parallèle possible pour toutes les paires de nœuds ID adjacents. La capacité limitée de l'étage de décodage est exprimée par des arcs pleins supplémentaires qui lient chaque nœud $ID(I_x)$ au nœud $ID(I_{x+2})$.

Prendre en compte les arcs barrés dans le calcul des intervalles de dates est trivial : la seule modification concerne l'estimation de l'impact des dates d'un nœud sur ses successeurs. Dans le calcul des dates au plus tard, la propagation des dates du nœud v à son successeur w devient :

$$latest[t_w^{ready}] = \max(latest[t_w^{ready}], latest[t_v^{\mathbf{start}}])$$

La modification dans le calcul des dates au plus tôt est similaire :

$$earliest[t_w^{ready}] = \max(earliest[t_w^{ready}], earliest[t_v^{\mathbf{start}}])$$

3.2. Ressources multiples

Les processeurs modernes comportent souvent plusieurs unités fonctionnelles du même type. Le modèle de Li ne permet pas de décrire ce type de parallélisme et il suppose que les instructions qui utilisent les mêmes ressources sont systématiquement sérialisées. Les nœuds en compétition pour l'usage d'une ressource commune sont liés par un arc pointillé qui n'est pas orienté pour permettre une exécution dans le désordre.

Pour prendre en compte des ressources multiples, nous proposons d'*annoter* les arcs pointillés avec le nombre d'exemplaires de la ressource concernée. Les algorithmes de calcul des dates doivent alors être modifiés pour permettre à plusieurs nœuds d'utiliser simultanément une ressource parallèle. La figure 3 montre un graphe avec des arcs pointillés annotés. Dans cet exemple, les instructions I_1, I_2 et I_4 doivent être traitées par une unité de multiplication qui peut exécuter deux instructions par cycle (autrement dit, le processeur dispose de deux multiplieurs), tandis que les instructions I_3 et I_5 doivent être traitées l'une après l'autre (dans n'importe quel ordre) par l'unique unité d'addition.

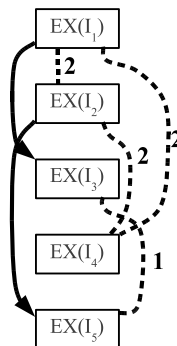

FIG. 3 – Un graphe d'exécution modélisant des ressources multiples.

Pour calculer les dates au plus tôt et au plus tard des nœuds, l'algorithme proposé par Li *et al.* [10] commence par déterminer les concurrents réels : les concurrents réels du nœud v sont ceux qui peuvent retarder son début. Il s'agit des nœuds qui ont besoin de la même ressource que v et : (i) concernent des instructions antérieures et sont prêts avant (ou en même temps que) v, ou (ii) concernent des instructions postérieures et commencent avant que v ne soit prêt. Les concurrents réels ne peuvent retarder le nœud v que si leur nombre excède la capacité de la ressource. Les figures 4 et 5 montrent comment les retards introduits par les nœuds concurrents peuvent être calculés en présence de ressources multiples. Dans ces algorithmes qui calculent les dates au plus tard et au plus tôt, par_v représente le degré de parallélisme de la ressource utilisée par le nœud v (en pratique, les ressources parallèles sont des unités fonctionnelles). On note $\max^{[n]}_{x \in \mathcal{E}}(f(x))$ la n-ième plus grande valeur de $f(x)$ pour toutes les valeurs $x \in \mathcal{E}$. La définition des ensembles de nœuds concurrents est inchangée par rapport aux algorithmes d'origine.

$latest[t^{ready}_{IF(I_1)}] = 0;$
pour chaque nœud v pris dans l'ordre topologique **faire**
  $latest[t^{start}_v] = latest[t^{ready}_v];$
  $S_{late} = late_contenders(v) \cap \{u | \neg separated[u, v] \wedge earliest[t^{start}_u] < latest[t^{ready}_v]\};$
  si $|S_{late}| \geq par_v$ **alors**
    $latest[t^{start}_v] = \min(\max^{[par_v]}_{u \in S_{late}}(latest[t^{finish}_u]), latest[t^{ready}_v] + max_lat_v - 1);$
  $S_{early} = early_contenders(v) \cap \{u | \neg separated[u, v]\};$
  si $|S_{early}| \geq par_v$ **alors**
    $tmp = \min(\max^{[par_v]}_{u \in S_{early}}(latest[t^{finish}_u]), latest[t^{start}_v] + (|S_{early}|/par_v) \times max_lat_v);$
    $latest[t^{start}_v] = \max(tmp, latest[t^{start}_v]);$
  finsi
  $latest[t^{finish}_v] = latest[t^{start}_v] + max_lat_v;$
  pour chaque successeur immédiat w de v **faire**
    si arc barré **alors**
      $latest[t^{ready}_w] = \max(latest[t^{ready}_w], latest[t^{start}_v]);$
    sinon
      $latest[t^{ready}_w] = \max(latest[t^{ready}_w], latest[t^{finish}_v]);$
finpour

FIG. 4 – Algorithme modifié de la fonction *LatestTimes()*.

Nous proposons, par ailleurs, de modéliser les unités fonctionnelles *pipelinées* en divisant les nœuds EX correspondants en autant de nœuds que d'étages dans l'unité et en liant ces nœuds par des arcs pleins, comme présenté sur la figure 6.

4. Evaluation de performances

Dans cette partie, nous donnons quelques indices de performances dans le but de montrer comment le modèle étendu permet d'obtenir des estimations de WCETs assez précises. Nous évaluons aussi la complexité du modèle en termes de temps de résolution.

$\mathtt{earliest}[t^{ready}_{\mathrm{IF}(I_1)}] = 0;$
pour chaque nœud v pris dans l'ordre topologique **faire**
  $\mathtt{earliest}[t^{start}_v] = \mathtt{earliest}[t^{ready}_v];$
  $S_{late} = \mathtt{late_contenders}(v)$
    $\cap \ \{u | \neg \mathtt{separated}[u,v] \wedge \mathtt{latest}[t^{start}_u] < \mathtt{earliest}[t^{ready}_v] < \mathtt{earliest}[t^{finish}_u]\};$
  $S_{early} = \mathtt{early_contenders}(v)$
    $\cap \ \{u | \neg \mathtt{separated}[u,v] \wedge \mathtt{latest}[t^{start}_u] \leq \mathtt{earliest}[t^{ready}_v] < \mathtt{earliest}[t^{finish}_u]\};$
  $S = S_{late} \cup S_{early};$
  si $|S| \geq \mathtt{par}_v$ **alors**
    $\mathtt{earliest}[t^{start}_v] = \max(\max^{[\mathtt{par}_v]}_{u \in S}(\mathtt{earliest}[t^{finish}_u]), \mathtt{earliest}[t^{ready}_v]);$
  $\mathtt{earliest}[t^{finish}_v] = \mathtt{earliest}[t^{start}_v] + \mathtt{min_lat}_v;$
  pour chaque successeur immédiat w de v **faire**
    si arc barré **alors**
      $\mathtt{earliest}[t^{ready}_w] = \max(\mathtt{earliest}[t^{ready}_w], \mathtt{earliest}[t^{start}_v]);$
    sinon
      $\mathtt{earliest}[t^{ready}_w] = \max(\mathtt{earliest}[t^{ready}_w], \mathtt{earliest}[t^{finish}_v]);$
finpour

FIG. 5 – Algorithme modifié de la fonction *EarliestTimes()*.

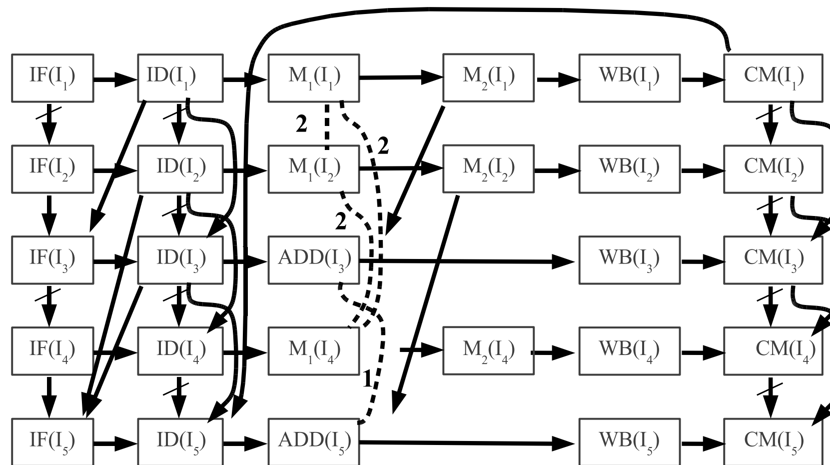

FIG. 6 – Exemple de graphe d'exécution pour un processeur avec des unités fonctionnelles pipelinées.

4.1. Méthodologie

Le modèle original a été implémenté dans l'outil Chronos développé à l'université de Singapour [12]. Cet outil est capable d'estimer le WCET d'un programme en prenant en compte les caractéristiques du processeur cible (pipeliné, à exécution dans le désordre, avec un cache d'instructions et un prédicteur de branchement). Il analyse les flots d'exécution possibles, estime les WCET individuels des blocs de base (en s'appuyant sur des graphes d'exécution) et calcule ensuite le WCET du programme en utilisant la méthode IPET. Nous avons implémenté l'extension du modèle décrite dans cet article dans l'outil Chronos pour obtenir les résultats présentés ici. Les benchmarks sont ceux distribués avec l'outil. Leurs WCETs ont été estimés en considérant un processeur superscalaire à 5 étages et à exécution non ordonnée, avec des unités fonctionnelles multiples. La table 1 donne des informations complémentaires sur les paramètres retenus pour l'évaluation.

largeur du pipeline	2 ou 4
taille de la file d'instructions	8
taille du tampon de réordonnancement	16
cache d'instructions	parfait (100% de succès)
prédicteur de branchement	parfait (pas d'erreur)
unités fonctionnelles	
addition entière	2 unités - latence de 1 cycle
mult./div. entière	1 unité - latence de 3 cycles
addition flottante	1 unité - latence de 2 cycles
mult. flottante	1 unité - latence de 4 cycles
div. flottante	1 unité - latence de 12 cycles
accès mémoire	1 unité - latence de 1 cycle

TAB. 1 – Caractéristiques du processeur

4.2. Complexité du modèle

La table 2 donne les temps de calcul mesurés pendant l'évaluation. La première colonne présente le temps nécessaire pour calculer les WCETs individuels de tous les blocs de base, tandis que la seconde indique le temps de calcul du WCET du programme complet avec la méthode IPET. Même si les benchmarks sont de taille limitée, ces résultats montrent que les besoins en puissance de calcul sont très raisonnables.

benchmark	temps d'analyse (sec.)	temps de résolution ILP (sec.)
matmul	0.059	0.007
matsum	0.052	0.007
isort	3.088	0.006
fdct	0.040	0.007

TAB. 2 – Temps d'analyse.

benchmark	DEGRÉ 2 mesuré	calculé	*rapport*	DEGRÉ 4 mesuré	calculé	*rapport*
matmul	11 207	14 657	1.31	9 752	13 657	1.40
matsum	70 306	80 512	1.15	60 305	80 511	1.34
isort	49 910	50 242	1.01	39 930	45 251	1.13
fdct	2 779	3 239	1.17	2 680	3 154	1.18
MOYENNE			**1.16**			**1.26**

TAB. 3 – WCETs mesurés et calculés.

4.3. Précision du WCET estimé

La table 3 indique les WCETs mesurés et calculés pour les quatre benchmarks. Le WCET mesuré a été obtenu en utilisant le simulateur au niveau cycle SimpleScalar (sim_outorder). Chaque benchmark a été exécuté avec un jeu d'entrées présumé correspondre au temps d'exécution le plus long (ce jeu de données est fourni dans la distribution de Chronos). Le WCET mesuré n'est donc pas garanti d'être le WCET réel mais il en est certainement très proche. Les WCETs calculés proviennent de l'analyse du modèle étendu.

Le rapport entre les WCETs mesurés et calculés montre que la surestimation est modérée : 16% en moyenne pour un processeur superscalaire de degré 2, et 26% pour un degré 4. Li a obtenu des résultats du même ordre [8](modèle sans cache ni prédicteur de branchement), ce qui montre que l'extension du modèle ne nuit pas à la précision.

5. Conclusion

Les besoins toujours plus grands en performance des systèmes temps-réel font que l'utilisation de processeurs avancés est inévitable. De nombreux auteurs ont souligné que modéliser un processeur superscalaire à exécution non ordonnée est complexe. Les pipelines ont été étudiés depuis plusieurs années mais le problème de l'exécution dans le désordre n'a été abordé que récemment par Li *et al.* [10]. Toutefois, leur modèle est limité à des processeurs scalaires. Dans cet article, nous l'avons étendu à des architectures superscalaires. Nous avons montré comment modéliser le parallélisme d'instructions dans les étages du pipeline et dans les unités fonctionnelles (quand il y a plusieurs unités du même type). Cela implique des modifications dans le graphe d'exécution (nouveaux types d'arcs) et dans les algorithmes de calcul des dates. Des résultats d'évaluation montrent que la surestimation du WCET reste modérée quand on prend en compte le parallélisme d'instructions. La précision des estimations est d'autant plus notable que les temps d'analyse sont réduits.

Bibliographie

1. A. Anantaraman, K. Seth, K. Patil, E. Rotenberg et F. Mueller – Virtual Simple Architecture (VISA) : exceeding the complexity limit in safe real-time systems – IEEE International Symposium on Computer Architecture, 2003.
2. I. Bate et R. Reutemann – Efficient integration of bimodal branch prediction and pipeline analysis – IEEE Conference on Embedded and Real-Time Computing Systems and Applications, 2005.
3. C. Burguière et C. Rochange – A contribution to branch prediction modeling in WCET analysis – Design, Automation and Test in Europe (DATE), 2005.
4. J. Engblom – Processor pipelines and static worst-case execution time analysis – PhD thesis, Uppsala University, 2002.

5. C. Healy, D. Whalley et M. Harmon – Integrating the timing analysis of pipelining and instruction caching – IEEE Real-Time Systems Symposium, 1995.
6. R. Heckmann, M. Langenbach, S. Thesing et R. Wilhelm – The influence of processor architecture on the design and the results of WCET tool – Proceedings of the IEEE, vol. 91(7), 2003.
7. S. Lim, Y.H. Bae, G.T. Jang, B. Rhee, S.L. Min, C.Y., Park, H. Shin, K. Park, S. Moon et C.S. Kim – An accurate Worst Case Timing Analysis for RISC processors – IEEE Transactions on Software Engineering, vol. 27(7), 1995.
8. X. Li – Microarchitecture modeling for timing analysis of embedded software – PhD thesis, National University of Singapore, 2005.
9. Y.-T. Li et S. Malik – Performance analysis of embedded software using implicit path enumeration – ACM SIGPLAN Notices, vol. 30, n°11, 1995.
10. X. Li, T. Mitra et A. Roychoudhury – Modeling out-of-order processors for software timing analysis – IEEE Real-Time Systems Symposium, 2004.
11. X. Li, T. Mitra et A. Roychoudhury – Modeling control speculation for timing analysis – Real-Time Systems Journal, 29(1), 2005.
12. X. Li, Y. Liang, T. Mitra et A. Roychoudhury – Chronos : a timing analyzer for embedded software – `http ://www.comp.nus.edu.sg/~rpembed/chronos`.
13. T. Lundqvist et P. Stenström – Timing anomalies in dynamically scheduled processors – IEEE Real-Time System Symposium, 1999.
14. C. Rochange et P. Sainrat – A time-predictable execution mode for superscalar pipelines with instruction prescheduling – ACM Conference on Computing Frontiers, 2005.