HyBrid-JIT : plateforme de compilation JIT assistée par du matériel spécialisé

Simon Rokicki

Stage du master de recherche

Rapport de stage

HyBrid-JIT : plateforme de compilation JIT assistée par du matériel spécialisé

Auteur :
Simon Rokicki

Superviseurs :
Steven Derrien, Cairn
Erven Rohou, Alf

Résumé

Le monde des architectures est très hétérogène : les architectures peuvent varier sur leur organisation et sur leur jeu d'instructions. Il est donc important d'offrir une portabilité entre ces différentes architectures.

L'ordonnancement dynamique des instructions est une technique très utilisée dans les processeurs modernes et dans les outils de compilation JIT (*Just in Time*). Si le problème est le même dans ces deux contexte, les contraintes sont très différentes. La différence majeure est que les processeurs modernes calculent cet ordonnancement de manière matérielle tandis que les outils de compilation JIT utilisent uniquement du logiciel.

Dans ce document, nous présentons une plateforme de compilation JIT utilisant du matériel spécialisé pour réaliser l'ordonnancement des instructions. Une étude expérimentale montre que l'utilisation de cette plateforme peut réduire le temps de compilation JIT jusqu'à 100 fois par rapport à une implémentation logicielle.

Table des matières

Introduction

Le réseau d'excellence HIPEAC (*High Performance and Embedded Architecture and Compilation*) a récemment dessiné sa perspective du domaine de l'embarqué pour 2020 [11]. D'après ce réseau, la consommation énergétique et la complexification des systèmes sont les deux enjeux majeurs des années à venir. Les systèmes embarqués actuels se caractérisent déjà par des contraintes très fortes en terme de consommation, de coût de production et de performance. Ces contraintes impactent fortement le choix des architectures matérielles utilisées. En effet, satisfaire toutes ces contraintes requiert l'utilisation d'architectures très spécialisées. Ainsi, les architectures présentes dans les systèmes embarqués sont très diversifiées (chaque système peut nécessiter une architecture spécifique optimisée pour certaines de ses applications) et difficiles à programmer efficacement. Cette difficulté croît avec le niveau de spécialisation.

Une illustration de ce lien entre spécialisation et difficulté de programmation est la présence ou non du parallélisme dans le jeu d'instructions : les processeurs à usage généraliste (*General Purpose Processor*) utilisent un jeu d'instructions séquentiel et vont réorganiser dynamiquement l'exécution des opérations pour faire apparaître du parallélisme d'instructions. Au contraire, les processeurs de type VLIW (*Very Long Instruction Word*) utilisent un jeu d'instructions dans lequel le parallélisme est explicite. Ainsi, pour produire du code exécutable, il est nécessaire d'affecter les différentes instructions aux unités fonctionnelles. C'est le problème d'ordonnancement sous contraintes de ressources qui est un problème NP complet. Cela rend le problème de génération de code pour VLIW difficile à résoudre.

Parallèlement, l'utilisation de *bytecode* indépendant de l'architecture s'est développée pour améliorer la portabilité du code. Si l'utilisation d'un *bytecode* est souvent associée à une exécution par interprétation, elle peut également conduire à la compilation JIT (*Just in Time compilation*) qui permet d'obtenir de meilleures performances.

La compilation JIT est une technique qui consiste à séparer la compilation en deux phases : la phase statique consiste à générer un *bytecode* indépendant de l'architecture cible et la phase dynamique va traduire ce *bytecode* pour obtenir un binaire compatible avec cette architecture. Ainsi, les passes de compilation machine dépendantes sont réalisées lors de l'exécution du code.

La compilation JIT offre donc les bénéfices suivants :

- Une portabilité efficace du code : le *bytecode* est indépendant de la machine et le fait qu'il soit compilé dynamiquement permet d'exploiter les spécificités de l'architecture.
- La possibilité de réagir dynamiquement à des évènements extérieurs (surchauffe de certaines zones du processeur, encombrement du cache de données, besoin de performances plus ou moins important) car le code peut être recompilé dynamiquement en prenant en compte ces paramètres.
- Une utilisation des paramètres spécifiques à une exécution pour la compilation. En effet, la compilation étant faite dynamiquement, il est possible de prendre en compte des informations qui n'étaient pas disponibles statiquement. Cela peut varier entre la prise en compte d'indicateurs de performance pour trouver les points chauds du programme, ou bien la considération des valeurs des arguments des différentes fonctions pour certaines optimisations.

Ces trois caractéristiques rendent la compilation JIT très attrayante pour les systèmes embarqués puisque qu'elle permet de gérer le problème de l'hétérogénéité des architectures tout en conservant une exécution relativement efficace (contrairement à l'interprétation). Cependant,

comme nous l'avons vu précédemment, les phases de génération de code machine pour les architectures utilisées dans l'embarqué sont plus complexes que pour les architectures généralistes, notamment parce qu'elles impliquent de résoudre un problème d'ordonnancement sous contrainte de ressources. L'utilisation de la compilation JIT pour les systèmes embarqués est de fait souvent écartée par les concepteurs car le coût de cette compilation est trop important pour être fait de manière dynamique.

Le sujet de ce stage est d'évaluer la faisabilité d'une plateforme de compilation JIT qui utilisera du matériel spécialisé pour réaliser l'étape d'ordonnancement dynamique des instructions. Une telle plateforme réduirait le temps de compilation pour les architectures de type VLIW et favoriserait une utilisation plus systématique de la compilation JIT pour de telles architectures.

Ce rapport commence en présentant dans la section 1 le contexte général dans lequel nous travaillons. Puis nous présenterons les différentes approches utilisées pour réaliser un ordonnancement dynamique des instructions dans la partie 2. La partie 3 sera consacrée à la présentation de notre approche à base d'accélération matérielle et la partie 4 à notre plateforme de compilation JIT. Enfin, une évaluation expérimentale des performances de notre outil de JIT sera présentée dans la section 5.

1 Contexte

1.1 Les architectures des systèmes embarqués

Le monde de l'embarqué est caractérisé par de fortes contraintes sur le coût matériel, sur la consommation énergétique et sur les performances des systèmes utilisés. A cause de ces contraintes, il est souvent nécessaire d'utiliser des architectures très différentes de celles utilisées pour les machines à usage généraliste.

En effet, un processeur à usage généraliste privilégie la performance, entraînant ainsi un surcoût en terme de consommation énergétique. L'exemple le plus représentatif d'architecture à usage généraliste est le processeur superscalaire à exécution dans le désordre (OoO : *Out of Order*). Ce processeur exécute un jeu d'instructions selon une sémantique séquentielle mais réorganise dynamiquement l'exécution des opérations en fonction de leurs dépendances pour trouver celles qui peuvent être exécutées en même temps : on parle de parallélisme d'instructions (ILP : *Instruction Level Parallelism*). Le matériel dédié à cette réorganisation permet d'exploiter facilement ce parallélisme d'instructions mais entraîne un surcoût énergétique. Le principe de l'architecture est illustré sur la figure 1 : les instructions sont décodées de façon séquentielle et sont ensuite réorganisées pour exploiter le parallélisme d'instructions.

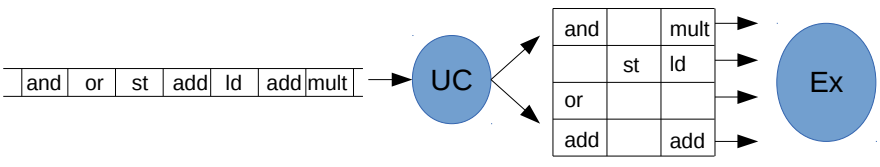


FIGURE 1 : Idée des processeurs superscalaires : l'unité de contrôle (UC) décode une fenêtre d'instructions et les réorganise pour les exécuter en parallèle.

Les processeurs utilisés dans l'embarqué sont en général des processeurs plus complexes à programmer mais qui consomment moins d'énergie que les processeurs superscalaires. Le pre-

mier exemple est celui des processeurs VLIW (*Very Long Instruction Word*). Le principe de ces processeurs est d'exécuter des paquets de plusieurs instructions en parallèle. Ainsi, le parallélisme est explicité dans le jeu d'instructions. La programmation efficace d'un tel processeur est plus délicate car le compilateur doit ordonnancer les instructions à exécuter sur les différentes unités d'exécution du processeur VLIW tout en respectant les dépendances. Ce type de processeur consomme moins d'énergie qu'un processeur superscalaire à exécution dans le désordre car il n'a plus besoin du mécanisme de réorganisation dynamique des instructions. Le principe de l'architecture est représenté sur la figure 2 : dans notre exemple, les instructions sont décodées et exécutées par paquets de quatre.

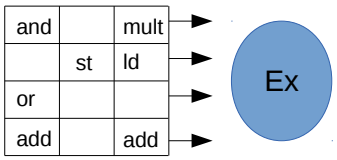


FIGURE 2 : Principe des processeurs VLIW : le processeur reçoit des paquets de plusieurs instructions qu'il exécute en parallèle.

Un deuxième type d'architecture utilisé dans le monde de l'embarqué est la famille des architectures de type CGRA (*Coarse Grain Reconfigurable Architecture*)[12]. Ces architectures sont composées d'une série d'unités d'exécution et de réseaux d'interconnexion. Ainsi, programmer l'architecture revient à modifier dynamiquement les chemins de données utilisés. Leur programmation est encore plus complexe que pour les VLIW et demande non seulement l'ordonnancement des instructions sur les unités d'exécution mais en plus le routage des opérandes entre les différentes unités d'exécution. L'avantage majeur de ce type d'architecture par rapport aux processeurs VLIW est l'absence de la file de registres partagée qui est l'élément le plus coûteux du VLIW (en terme de surface utilisée). Le principe des architectures CGRA est représenté sur la figure 3 : il n'y a pas d'instructions, le processeur va simplement lire une configuration de son chemin de données à chaque cycle. La configuration correspond à un ensemble de bits permettant de dire si les interconnexions doivent être passantes ou bloquantes.

Dans l'exemple de la figure 3, la configuration du chemin de données fait que l'unité d'exécution en haut à droite utilisera son propre résultat comme opérande au cycle suivant (car les deux points d'interconnexions correspondant à ses opérandes sont activés au niveau de son résultat).

Il y a donc un compromis entre la simplicité de programmation et la consommation en énergie. Les systèmes embarqués sacrifient cette simplicité au profit d'une faible consommation énergétique pour satisfaire ces contraintes très fortes.

1.2 Adaptation dynamique des performances

Comme nous l'avons vu dans la partie précédente, la consommation énergétique est l'un des soucis majeurs lors de la conception de systèmes embarqués. Dans cette partie, nous allons présenter différentes techniques permettant de modifier dynamiquement les performances et la consommation de l'architecture utilisée.

Ces techniques peuvent être utilisées dans un contexte où les tâches à exécuter ont des *deadlines* à respecter, et où il est possible d'adapter la performance du processeur pour s'approcher au maximum de cette limite. De même, dans un contexte où les tâches ont été ordonnancées selon leur pire temps d'exécution (WCET : *Worst case execution time*), quand une tâche se finit

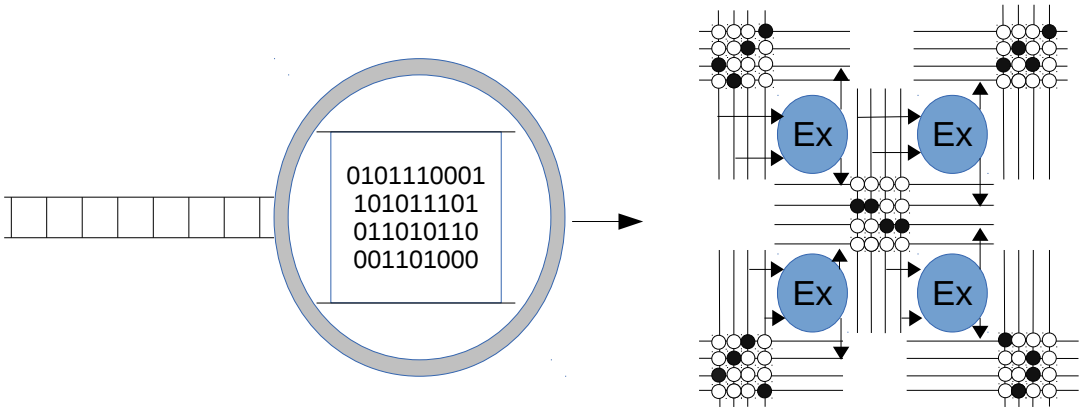


FIGURE 3 : Idée des architectures CGRA : l'architecture est constituée d'unités d'exécution (EX) séparées par des réseaux d'interconnexion. Sur la figure, ces réseaux sont représentés par des croisements qui peuvent être allumés (noirs) ou éteints (blancs). Une valeur calculée par une unité donnée peut être envoyée aux unités d'exécution voisines via ce réseau. Pour programmer l'architecture, il faut modifier dynamiquement les chemins de données en utilisant les réseaux d'interconnexion.

plus vite que son WCET, il est possible d'allouer le temps supplémentaire à la tâche suivante, en ralentissant son exécution. Cette idée correspond au DSR (*Dynamic Slack Reclamation*) [21].

Dans cette partie, nous commencerons par étudier les différentes composantes de la puissance dissipée par une architecture puis nous présenterons deux techniques permettant d'adapter dynamiquement la performance et la consommation de l'architecture.

La puissance consommée par l'architecture se découpe de la façon suivante :

$$P = P_{dyn} + P_{stat}$$

Dans l'équation précédente, P_{dyn} représente la puissance dynamique dissipée, c'est à dire la puissance dissipée lors des changements d'état des transistors. Au contraire, P_{stat} représente la puissance statique, qui correspond aux pertes dues aux courants qui circulent.

La puissance dynamique est modélisée par la formule suivante :

$$P_{dyn} = C.V_{dd}^2.f$$

Où C est un facteur spécifique à l'architecture et est proportionnel au nombre de transistors utilisés, f la fréquence et V_{dd} est la tension d'alimentation. La puissance statique peut être modélisée par la formule suivante [2] :

$$P_{stat} = V_{dd}.I_{leak}.N.k$$

où I_{leak} est le courant de fuite, N le nombre de transistors et k une constante empirique qui dépend du style de l'architecture.

Nous allons maintenant voir deux méthodes pour adapter dynamiquement les performances de l'architecture : l'adaptation dynamique de la tension et de la fréquence (DVFS : *Dynamic Voltage and Frequency Scaling*) et l'ajout ou la suppression d'unités d'exécution grâce au *power gating* qui permet de couper l'alimentation d'une partie du circuit.

Adaptation de la fréquence et de la tension La méthode la plus couramment utilisée pour adapter dynamiquement les performances d'un processeur est le DVFS [30]. Le principe de base du DVFS est de modifier la fréquence de fonctionnement quand il y a besoin. Cependant, lorsque la fréquence augmente, la tension nécessaire à un bon fonctionnement augmente également. Il est donc nécessaire de modifier les deux paramètres en même temps. Ces paramètres sont liés par la relation suivante :

$$T_{delay} = \gamma \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}}$$

où T_{delay} représente la durée du chemin critique (donc l'inverse de la fréquence) et V_{th} la tension de seuil (tension à partir de laquelle le transistor change d'état). γ et α sont des constantes qui dépendent de l'architecture et de la technologie utilisées.

Si nous considérons que la fréquence de seuil V_{th} est fixée, alors la puissance dynamique va augmenter de manière quadratique lors de l'augmentation de la fréquence. La puissance statique va augmenter de manière linéaire.

Adaptation du nombre d'unités fonctionnelles La deuxième solution proposée est de changer dynamiquement les ressources d'exécution. Par exemple, pour un processeur VLIW, l'idée est de changer le nombre d'instructions qui vont être exécutées en parallèle (nombre de voies du processeur). Cette modification est possible dynamiquement en utilisant le *power gating* : le principe est d'utiliser un transistor pour couper dynamiquement l'alimentation de l'unité fonctionnelle. Ainsi, ce transistor permettra d'allumer ou d'éteindre l'unité d'exécution.

L'ajout d'une unité d'exécution au processeur a un impact linéaire sur le facteur C de la puissance dynamique et un impact linéaire sur le nombre N de transistors pour la puissance statique. Ainsi, augmenter le nombre de voies du processeur VLIW va augmenter linéairement la puissance consommée.

Cependant, l'augmentation du nombre de voies est limitée : à cause de la file de registres partagée, il n'est pas raisonnable d'utiliser plus de 8 voies pour le processeur VLIW. Ainsi, pour adapter les performances, il serait possible de modifier conjointement la fréquence de fonctionnement et le nombre de voies du processeur pour obtenir le meilleur compromis en terme de puissance de calcul et de puissance dissipée. La figure 4 illustre le changement du nombre de voies dans un processeur VLIW.

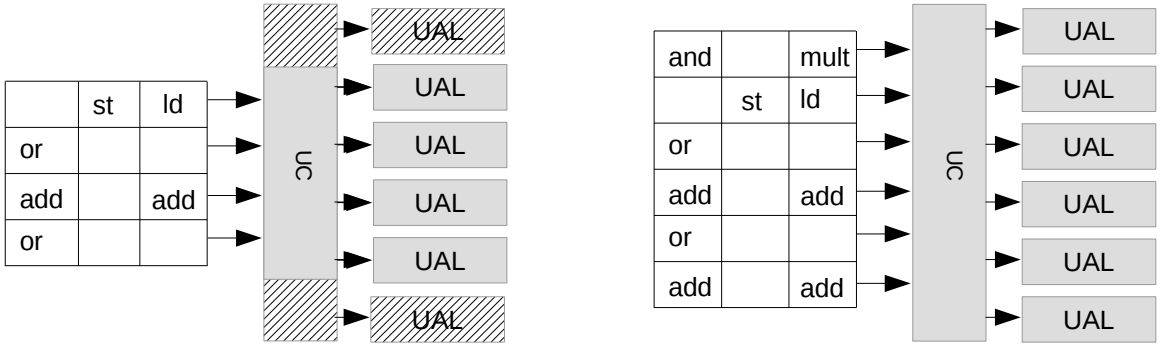


FIGURE 4 : Exemples de processeurs VLIW avec un nombre de voies différent. La partie de gauche représente un VLIW à 4 voies (les zones hachurées sont éteintes) et celle de droite représente un VLIW à 6 voies. Il est possible de passer d'une architecture à l'autre en éteignant des UAL (unité arithmétique et logique) et des zones de l'UC (unité de contrôle).

Nous avons tenté de modéliser ces variations. Pour cela, nous avons utilisé les valeures de la consommation du processeur Crusoe de Transmeta [16]. Le processeur Crusoe est un processeur VLIW à quatre voies, utilisant le DVFS. Selon ce document, la fréquence peut varier par sauts de 33MHz avec une variation de 25mV pour la tension d'alimentation. De plus, la puissance dynamique du processeur représente 90% de la consommation totale, à la fréquence et à la tension de fonctionnement classique. A partir de ces chiffres, il a été possible de modéliser la puissance dissipée par le VLIW à quatre voies pour différentes fréquences de fonctionnement. Ensuite, pour obtenir ces chiffres pour des processeurs à deux ou six voies, nous avons supposé que la consommation de base variait proportionnellement au nombre de voies. Il a ainsi été possible d'obtenir une approximation de la puissance dissipée pour chacune de ces architectures. La figure 5 (haut) représente la variation de la puissance dissipée en fonction de la fréquence pour les processeurs à deux, quatre et six voies.

Pour comparer les performances obtenues avec ces différents processeurs, nous avons calculé le nombre d'opérations effectuées par seconde. Cependant, afin de prendre en compte que le fait de doubler le nombre de voies d'un processeur ne doublera pas les performances, nous avons calculé la performance en utilisant le nombre moyen d'opérations par cycle sur ces architectures, après ordonnancement d'un algorithme de DCT (*Discrete Cosinus Transform*) sur l'architecture. Les résultats de cette étude sont présentés sur la figure 5 (bas), où est représenté la puissance dissipée en fonction du nombre d'opérations exécutées par cycle. Nous pouvons voir sur cette figure que varier uniquement la fréquence est efficace sur les basses fréquences mais qu'au delà d'un certain seuil, il est préférable de la réduire et d'augmenter le nombre de voies. Grâce à ce modèle, il est possible de définir une courbe de Pareto (c'est à dire une courbe permettant de visualiser les différents compromis optimaux) et des règles pour augmenter les performances.

Au delà de l'adaptation dynamique des performances, il est également possible d'utiliser le *power gating* des unités d'exécution du processeur lorsque le parallélisme d'instructions est insuffisant et sous-utilise les unités disponibles. Il serait ainsi possible d'introduire une mesure du parallélisme d'instructions disponible pour un programme donné et de décider l'adaptation des performances en fonction de cette mesure.

Cependant, le changement du nombre d'unités fonctionnelles du processeur demande l'utilisation d'un binaire différent puisque le parallélisme est directement présent dans le jeu d'instructions du VLIW. Nous avons donc deux possibilités pour rendre le changement du nombre de voies possible : conserver plusieurs versions du code compilé en fonction du nombre de voies sur le VLIW ou utiliser la compilation dynamique. Dans la partie suivante, nous allons présenter la compilation dynamique et ses enjeux pour les systèmes embarqués.

1.3 La compilation JIT

Le principe de la compilation JIT est de reporter la phase finale de la compilation (phase dépendante de l'architecture cible) au moment de l'exécution du code. Cela permet notamment d'apporter de la portabilité, d'adapter dynamiquement l'exécution du code à des évènements extérieurs ou de prendre en compte des paramètres spécifiques à une exécution pour les optimisations. Dans cette partie, nous étudierons plusieurs approches de la compilation JIT et des exemples des avantages apportés.

Portabilité du code En utilisant un flot de compilation classique, le déploiement d'un code demande de compiler pour chacune des architectures cibles. Ce point est particulièrement important dans le monde des systèmes embarqués car les architectures y sont très hétérogènes. Cependant, même dans le monde des processeurs généralistes où la plupart des architectures

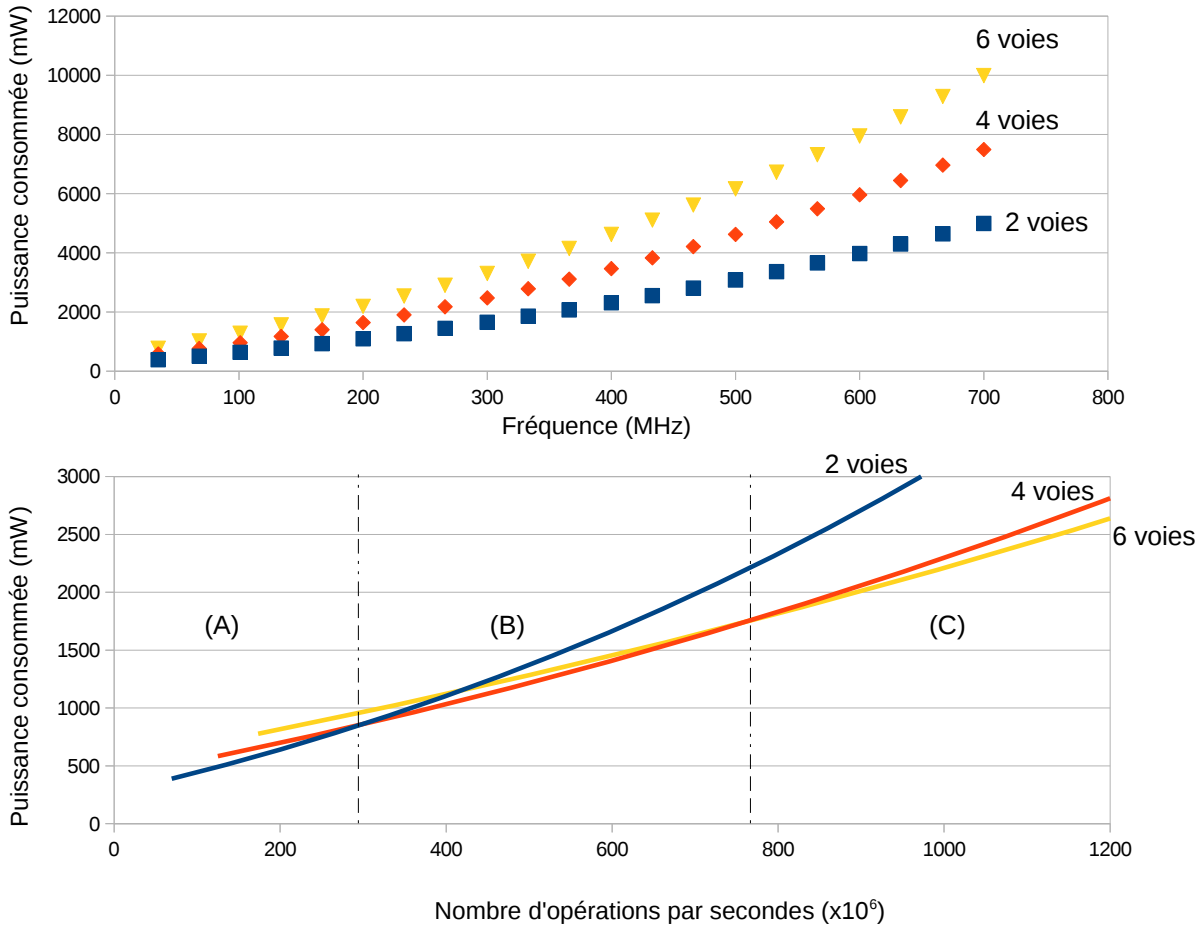


FIGURE 5 : Utilisation conjointe du *power gating* et du DVFS : la figure du haut représente la puissance dissipée en fonction de la fréquence pour des processeurs à 2, 4 ou 6 voies et la figure du bas représente la puissance dissipée en fonction du nombre d'opérations effectuées par seconde lors de l'exécution d'une DCT. La zone A correspond à la zone de performences où un processeur à deux voies est optimal, la zone (B) pour le processeur à quatre voies et la zone (C) pour le processeur à six voies.

sont compatibles entre elles (compatibilité due au x86), la portabilité apportée par un compilateur JIT est plus intéressante car elle permet de conserver les spécificités d'une architecture donnée. La figure 6 illustre cette portabilité : pour profiter des spécificités d'une machine (ici la présence ou non d'une unité arithmétique flottante), il faudra distribuer deux binaires, correspondant chacun à une architecture donnée. Cependant, grâce à la compilation JIT, un *bytecode* indépendant de la machine est généré statiquement et sera traduit dynamiquement en un binaire compatible avec l'architecture cible. Ainsi, la personne qui déploie le code n'a qu'une seule version à maintenir et pourra utiliser les spécificités de toutes les architectures cibles.

L'outil de JIT VaporSIMD [28] offre cette portabilité pour les différentes architectures offrant des instructions SIMD (*Single Instruction Multiple Data*). En effet, le principe de ces instructions est d'utiliser une instruction unique qui agira sur plusieurs données en parallèle. Il est ainsi possible d'exploiter du parallélisme d'instructions en gardant un binaire de petite taille. Cependant, les instructions SIMD et leurs caractéristiques varient selon les architectures. Ainsi, le *bytecode* généré par VaporSIMD sera indépendant de ces caractéristiques et sera facile à traduire

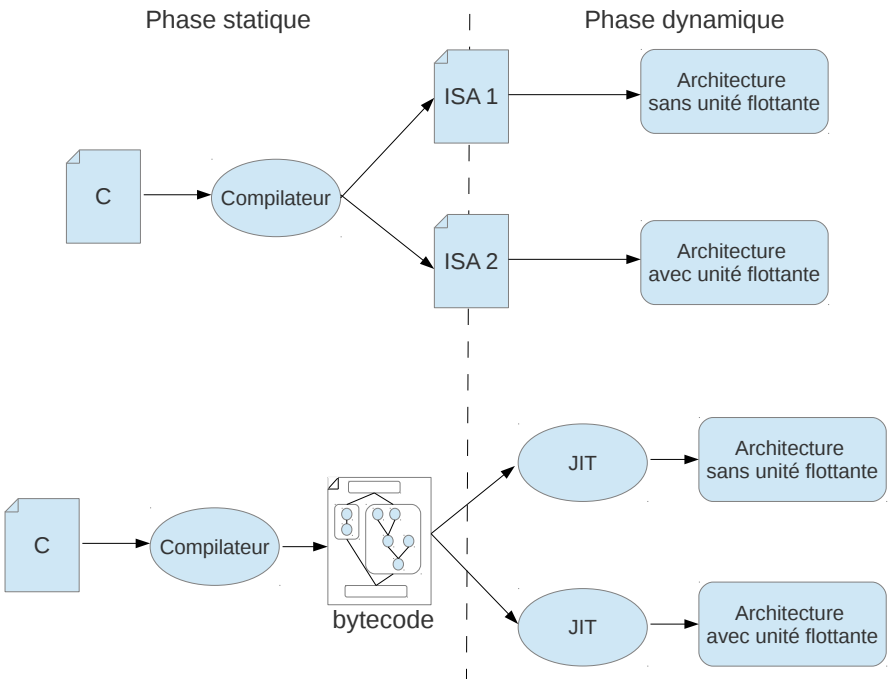


FIGURE 6 : La portabilité offerte par les outils de JIT : sans utiliser ces méthodes, il faut compiler le code pour chacun des jeux d'instructions (ISA) des différentes architectures cibles. Avec la compilation JIT, le programmeur n'a qu'à compiler vers un *bytecode* qui permettra ensuite de passer, de manière dynamique, vers le jeu d'instructions supporté par l'architecture

dynamiquement vers les différents jeux d'instructions.

Adaptation dynamique Un autre avantage de compilation dynamique est qu'il est possible de réagir à des évènements extérieurs lors de l'exécution du code. En effet, nous avons présenté dans la partie précédente une plateforme adaptative capable de modifier ses performances pour contrôler la consommation énergétique. Avec une telle plateforme, un outil de compilation JIT peut, en fonction des échéances des différentes tâches, choisir d'augmenter ou de réduire les performances.

L'outil de compilation JIT RobustSIMD [24] permet de réagir dynamiquement à des surcharges du cache de données. En effet, si l'indicateur de performances détecte que le cache de données est trop utilisé et ralentit l'exécution, le code sera recompilé pour prendre en compte ce paramètre (en changeant le niveau de parallélisme exploité pour réduire la pression sur le cache).

Les avantages apportés par la compilation dynamique sont donc la portabilité du code et une possibilité d'adaptation dynamique. Ces points peuvent être intéressants dans le domaine des systèmes embarqués.

Dans cette section, nous avons présenté les différentes architectures utilisées dans les systèmes embarqués et identifié l'une des contraintes majeures de ce domaine : la consommation énergétique. Nous avons également présenté une plateforme adaptative permettant de réguler cette consommation et basée sur la compilation JIT. Cependant, nous avons vu précédemment que les architectures utilisées dans ces systèmes sont souvent complexes à programmer (notamment en raison de la phase d'ordonnancement des instructions). Nous allons maintenant étudier la résolution de cet ordonnancement dynamique des instructions dans les processeurs superscalaires et

dans les outils de JIT existants.

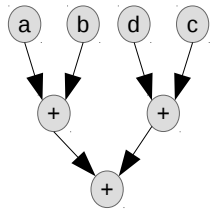
2 Ordonnancement dynamique des instructions

Dans cette partie nous allons étudier plusieurs approches d'ordonnancement dynamique des instructions. Dans un premier temps, nous allons voir l'approche présente dans les processeurs superscalaires, entièrement basée sur du matériel, puis nous étudierons des outils de JIT qui visent des architectures de type VLIW et qui exécutent donc un ordonnancement dynamique des instructions.

2.1 Ordonnancement dynamique matériel : l'approche super-scalaire

Le fonctionnement des processeurs superscalaires est basé sur le parallélisme d'instructions (ILP : *Instruction Level parallelism*) : même si le jeu d'instructions de la machine a une sémantique séquentielle, le processeur va tenter de réorganiser les différentes instructions pour exploiter cet ILP. Ce mécanisme de réorganisation travaille sur une fenêtre de n instructions du programme et va déterminer dynamiquement les dépendances entre ces instructions.

Considérons l'exemple figure 7 : la partie gauche de la figure représente le graphe de flot de données d'un bloc de base et la partie de droite correspond à son implémentation en assembleur (avec une sémantique séquentielle).




```
Load r1 a          // r1 = a
Load r2 b          // r2 = b
Add r3 r1 r2       // r3 = r1 + r2
Load r1 c          // r1 = c
Load r2 d          // r2 = d
Add r4 r1 r2       // r4 = r1 + r2
Add r1 r3 r4       // r1 = r3 + r4
```

FIGURE 7 : Exemple de graphe de flot de données d'un bloc de base et son implémentation en assembleur

Considérons l'exécution de ce code sur un processeur superscalaire pipeliné à deux voies où chaque instruction va s'exécuter en deux cycles et où il est possible de commencer l'exécution de deux nouvelles instructions à chaque cycle. Le mécanisme de réordonnancement utilise une fenêtre de huit instructions. Le résultat d'une telle exécution est représenté dans le tableau suivant :

Cycle	Slot 1	Slot 2
0	Load r1 a	Load r2 b
1	Load r1' c	Load r2' d
2	Add r3 r1 r2	-
3	Add r4 r1' r2'	-
4	-	-
5	Add r1" r3 r4	-

Il y a plusieurs points importants sur la trace d'exécution précédente :

- Les instructions indépendantes peuvent être exécutées en parallèle. Par exemple, lors du premier cycle d'exécution, les deux *load* sont exécutés en même temps car l'architecture a déterminé qu'il n'y avait aucune dépendance entre eux.
- L'exécution ne respecte pas obligatoirement l'ordre séquentiel du code assembleur. En effet, les deux derniers *load* sont exécutés avant la première addition, ce qui est contraire à la sémantique séquentielle de l'assembleur représenté figure 7.
- Le nom des registres a été modifié pour éviter les dépendances de nom : dans les *load* exécutés au deuxième cycle, les registres $r1$ et $r2$ ont été renommés $r1'$ et $r2'$ afin d'éviter toute dépendance avec les deux premiers *load* et la première addition. Ce processus de renommage de registre permet donc d'augmenter les possibilités de réorganisation des instructions.

Dans les processeurs superscalaires, ce processus d'ordonnancement dynamique des instructions est réalisé par du matériel dédié, selon la méthode de Tomasulo [32] proposée en 1967 et permettant d'ordonnancer les instructions sélectionnées sur une fenêtre donnée tout en renommant les registres pour éviter les dépendances de noms. Même si la méthode date des premiers processeurs superscalaires, elle est toujours d'actualité. En effet, le développement des processeurs superscalaires est surtout basé sur l'augmentation de la taille de la fenêtre d'instructions considérée et sur les mécanismes de prédiction de branchement qui permettent d'aller chercher des instructions plus loin dans le code. Le principe de l'ordonnancement dynamique des instructions est cependant toujours le même.

Algorithme de Tomasulo L'algorithme de Tomasulo est un algorithme glouton d'ordonnancement des instructions sous contraintes de ressources. Son fonctionnement peut se résumer simplement : lorsqu'une instruction est décodée, elle est stockée dans une table de réservation avec la valeur de ses opérandes déjà prêts et un identifiant virtuel des opérandes non calculés. Une fois dans cette table de réservation, un mécanisme observe l'exécution pour détecter le moment où un opérande est calculé afin d'ajouter sa valeur dans la table de réservation. Lorsque tous les opérandes de l'instruction sont disponibles, l'instruction peut être exécutée.

Nous retrouvons donc les caractéristiques de l'ordonnancement dynamique présenté pour l'exemple de la figure 7 : la taille de la fenêtre des instructions considérées correspond à la taille des tables de réservation et le fait d'exécuter les instructions dès que leurs dépendances sont résolues est l'heuristique gloutonne permettant l'exécution dans le désordre. Enfin, l'utilisation des valeurs des opérandes et des identifiants virtuels correspond à l'étape de renommage des registres utilisés.

Un des avantages principaux de la méthode de Tomasulo est qu'elle est particulièrement adaptée à une implémentation matérielle, comme nous pouvons le voir sur la figure 8 qui représente la première architecture qui utilise la méthode de Tomasulo.

La durée de vie d'une instruction à exécuter se découpe en plusieurs étapes :

- Tout d'abord, l'instruction à exécuter est stockée dans la file d'instructions.
- Les instructions de cette file sont décodées dans l'ordre. Le décodage de la fonction consiste à lire le nom des registres des opérandes et le nom du registre où le résultat sera écrit. Si la valeur de certains opérandes est disponible, elle est lue dans la file de registres. Sinon, l'identifiant virtuel de l'opérande est déterminé. Si l'instruction écrit un résultat, un identifiant virtuel est attribué à cette valeur. Ces noms virtuels sont uniques pour éviter

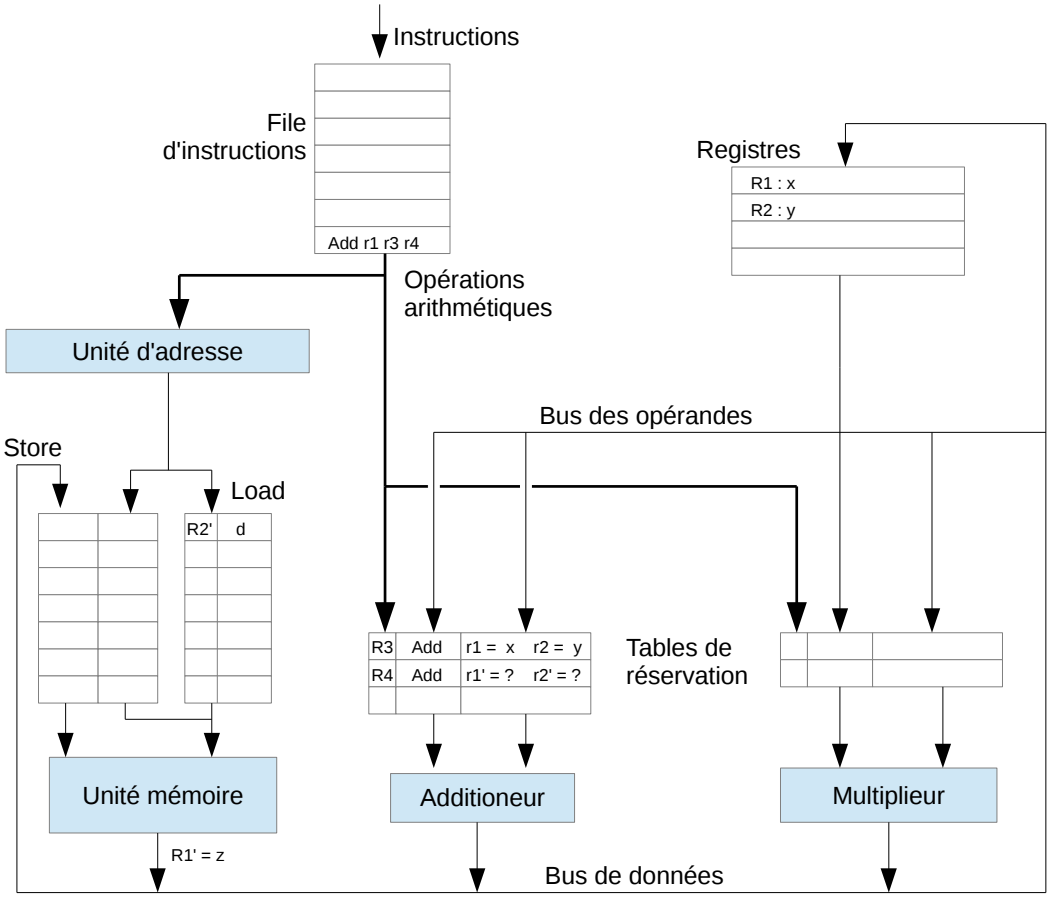


FIGURE 8 : Organisation d'un processeur superscalaire selon l'approche de Tomasulo [19]

les fausses dépendances. Il est important de remarquer que cette étape de renommage de registres est possible uniquement parce que les instructions sont décodées dans l'ordre.

- Une fois l'instruction décodée, elle est stockée dans une table de réservation en attendant que ses opérandes manquants soient calculés. Un mécanisme va surveiller le bus de données pour détecter le passage d'un opérande ayant le bon identifiant virtuel, et va ainsi copier les valeurs des opérandes dans la table de réservation lorsqu'ils sont calculés.

- Enfin, lorsque tous les opérandes de l'instruction sont prêts, elle est exécutée et le résultat est envoyé vers les registres via le bus de données, accompagné par son identifiant virtuel. Si cette valeur était attendue par une autre instruction, elle sera alors recopiée également dans la table de réservation correspondante.

Pour aider à comprendre le fonctionnement de la méthode de Tomasulo, la figure 8 illustre également une étape de l'exécution de l'exemple de la figure 7 : à l'instant représenté sur la figure, les six premières instructions ont été décodées et placées dans les tables de réservation correspondantes. Les deux premiers chargements mémoire sont déjà exécutés et leur résultat est stocké dans la file de registre avec le nom virtuel du registre correspondant. Dans la table de réservation de l'additionneur, la première addition connaît la valeur de ses deux opérandes et peut être exécutée. Le bus de données est surveillé dans l'attente de la valeur des opérandes de la deuxième addition. Pour ce qui est des chargements mémoire, la valeur de $r1'$ vient d'être lue en mémoire et la valeur de $r2'$ est prête à être lue.

Ainsi, à la prochaine étape de l'exécution, la première addition va être exécutée ainsi que le la lecture de d en mémoire. La valeur de $r1'$ va être copiée dans la file de registre ainsi que dans la table de réservation correspondant à la deuxième addition. Enfin, l'addition présente dans la file d'instructions va être décodée et placée dans la table de réservation.

Un défaut majeur de cette approche est le passage à l'échelle. En effet, une approximation empirique de la quantité de parallélisme d'instructions dans les programmes dit que dans une fenêtre de n instructions, il est possible de trouver $\sqrt{n}$ instructions à exécuter en parallèle [25]. Ainsi, pour augmenter le nombre d'instructions à exécuter en parallèle, il faut augmenter de façon quadratique la taille de la fenêtre. De plus, le coût en silicium de la fenêtre d'instructions évolue de façon quadratique en fonction du nombre d'instructions. Ainsi, le coût du mécanisme d'ordonnancement dynamique est de l'ordre de $O(n^4)$ où n correspond au nombre de voies du processeur.

Pour conclure cette partie, la méthode de Tomasulo permet de résoudre de façon matérielle le problème de l'ordonnancement dynamique des instructions dans les processeurs superscalaires. Nous allons maintenant voir comment ce problème est abordé dans les outils de compilation JIT qui visent une architecture VLIW.

2.2 Ordonnancement dynamique logiciel : les outils de JIT pour processeur VLIW

Comme nous l'avons vu dans la partie 1, un compilateur JIT qui vise des architectures de type VLIW devra ordonnancer dynamiquement les instructions du *bytecode*. Cette étape de la compilation est l'une des plus coûteuses en temps (avec l'allocation de registres). Les contraintes temporelles du JIT forcent à utiliser des heuristiques gloutonnes pour résoudre ce problème d'ordonnancement sous contraintes de ressources. Dans cette partie, nous allons étudier les différents outils de JIT de la littérature qui visent des architectures de type VLIW.

La première approche est celle de l'outil JIST [1] qui permet de compiler dynamiquement du *bytecode* java pour l'exécuter sur un processeur VLIW. Cet outil comporte donc une phase d'ordonnancement des instructions et une phase d'allocation des registres.

L'algorithme d'ordonnancement utilisé par JIST est l'algorithme de *list scheduling*. Cette approche gloutonne est centrée sur les cycles de l'exécution : pour chaque cycle, l'algorithme va tenter d'attribuer une instruction à chaque unité fonctionnelle. Si plusieurs instructions sont éligibles, une fonction de priorité permet de les départager.

Pour mieux comprendre le fonctionnement du *list scheduling*, nous allons l'appliquer au graphe de flot de données représenté figure 9. L'architecture ciblée a les caractéristiques suivantes :

- peut exécuter une addition et un accès mémoire en parallèle
- nécessite trois cycles pour exécuter une instruction
- peut commencer une instruction à chaque cycle (pipelinée).

L'ordonnancement obtenu est représenté sur la figure 10 : chaque colonne représente un cycle de l'ordonnancement et la partie inférieure de la figure donne, pour chaque cycle, la liste des instructions prêtes. Ainsi, à chaque cycle, l'algorithme va ordonnancer l'instruction prête compatible ayant la plus haute priorité. L'ordonnancement ainsi généré permet d'exécuter le bloc de base en 19 cycles.

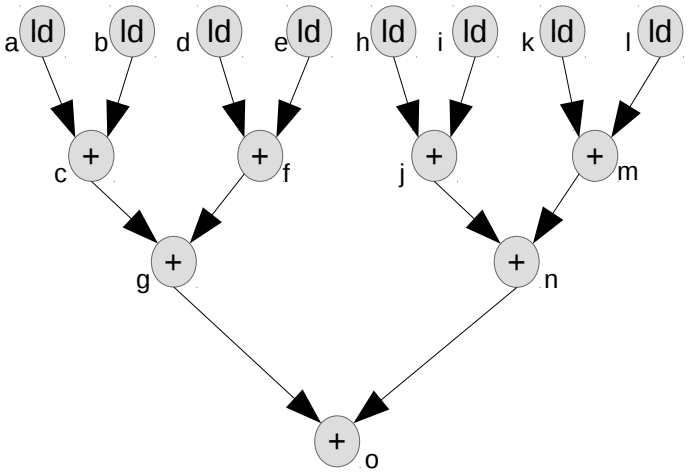


FIGURE 9 : Exemple de graphe de flot de données

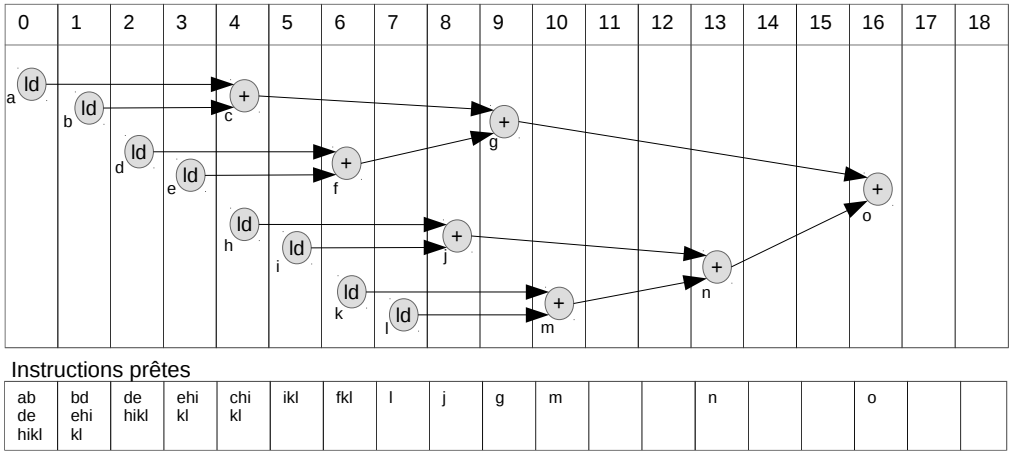


FIGURE 10 : Résultat d'un *list scheduling* sur l'exemple de la figure 9

La deuxième contribution que nous allons étudier est celle de Dupont de Dinechin [7] qui étudie plus en détail les différents algorithmes d'ordonnancement pour les outils de JIT visant des architectures VLIW. L'algorithme qu'il propose est une méthode basée sur le *scoreboard scheduling*, une heuristique gloutonne qui se base cette fois-ci sur les instructions à ordonnancer. Ces instructions sont placées dans l'ordre de leur priorité au premier cycle où elles peuvent être exécutées (c'est à dire au premier cycle où leurs opérandes sont prêts et où une unité d'exécution est disponible). Pour limiter la complexité, l'algorithme ne va considérer que les cycles présents sur une fenêtre de taille $window\_size$, commençant au cycle $window\_start$. Ainsi, pour ordonnancer une instruction, l'algorithme va commencer au cycle $window\_start$ et avancer jusqu'au premier cycle où l'exécution est possible. Si l'instruction est ordonnancée à un cycle c au delà de $window\_start+window\_size$ alors la fenêtre est déplacée selon la règle $window\_start = c - window\_size$ afin que c soit dans la fenêtre. Le *scoreboard scheduling* utilisant une fenêtre est en général plus efficace que le *list scheduling* (en terme de temps d'ordonnancement).

Pour mieux comprendre, nous avons appliqué l'algorithme sur le même graphe de flot de données de la figure 9, avec la même architecture cible que précédemment. Les instructions sont considérées dans l'ordre donné par leur nom et la fenêtre utilisée est de taille 4. L'ordonnancement

ainsi obtenu est représenté sur la figure 11. Sur cette figure, nous avons représenté la fenêtre considérée après l'ordonnancement de l'instruction g. En effet, après avoir placé l'instruction g, la fenêtre a été avancée de plusieurs cycles et l'instruction h est ordonnancée au cycle 6 alors qu'elle aurait pu être ordonnancé au cycle 4. Cet exemple permet donc de voir la limite de cette heuristique d'ordonnancement.

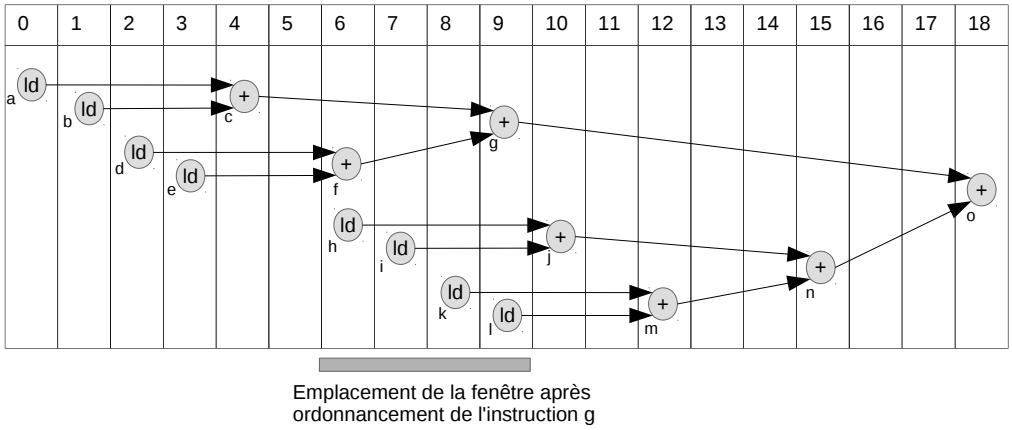


FIGURE 11 : Résultat d'un *scoreboard scheduling* sur l'exemple de la figure 9

Un élément de référence important dans le domaine de la compilation dynamique pour processeur VLIW est Transmeta [9]. Ce système, apparu dans les années 2000 proposait de traduire dynamiquement du code x86 pour l'exécuter sur un processeur VLIW afin de réduire grandement la consommation énergétique. La solution proposée est une forme de compilation JIT qui allie interprétation et compilation. Il y a très peu d'informations sur les mécanismes utilisés pour réaliser l'ordonnancement des instructions sur le VLIW. Cependantla cause majeure de l'échec de Transmeta était la lenteur du compilateur JIT.

Dans cette partie, nous avons donc étudié diverses approches logicielles pour l'ordonnancement dynamique des instructions. Tous les algorithmes utilisés sont des approches gloutonnes car le temps de compilation ne doit pas être trop important. L'intérêt majeur de la compilation dynamique sur l'ordonnancement matériel des superscalaires est le fait que l'ordonnancement ainsi obtenu est sauvegardé et réutilisé. Nous allons maintenant étudier des approches hybrides qui utilisent du matériel dédié pour ordonnancer les instructions mais qui comprennent également un processus de sauvegarde de cet ordonnancement.

2.3 Approches hybrides

Nous avons étudié précédemment l'ordonnancement dynamique des instructions dans les processeurs superscalaires et dans les compilateurs JIT pour VLIW. Un des inconvénients de la méthode utilisée dans les processeurs superscalaires est la nécessité de calculer l'ordonnancement à chaque exécution. Ainsi, lorsqu'une boucle est exécutée, l'ordonnancement sera calculé à chaque itération (et pourra de fait être différent à chacune de ces itérations). Ce mécanisme est donc très peu efficace en terme de consommation énergétique.

Plusieurs approches proposent des solution à ce problème en sauvegardant l'ordonnancement dans un cache pour le réutiliser plus tard. Nous allons étudier ici deux processeurs réalisant un ordonnancement dynamique du même type que celui des processeurs superscalaires mais qui sauvegardent le résultat de cet ordonnancement pour le réutiliser.

La première approche est le système Yoga [33] dont le principe est d'exécuter le code dans le désordre, avec le même mécanisme que les processeurs superscalaires. Pendant cette exécution, une trace est sauvegardée et est utilisée pour générer un binaire du bloc de base pour VLIW. Lorsqu'un bloc est répété un certain nombre de fois, le mécanisme d'exécution dans le désordre est désactivé et le code VLIW généré est exécuté. Cette approche permet donc de réduire le surcoût énergétique dû à l'ordonnancement des instructions.

La deuxième approche est celle du DTSVLIW [8]. Dans ce système, le code est exécuté une première fois sur processeur très simple pour obtenir une trace de l'exécution qui permettra de générer un ordonnancement des instructions. Chaque instruction qui vient d'être exécutée est placée au premier endroit où elle aurait pu être exécutée dans le bloc de base. Une fois le code entièrement généré, il est exécuté sur un processeur VLIW. Cette approche est donc très proche de celle de Yoga car le mécanisme utilisé pour ordonnancer est simplement une application du *scoreboard scheduling* utilisé dans les processeurs à exécution dans le désordre. Une fois encore, le problème de la surconsommation énergétique est résolu en sauvegardant un ordonnancement obtenu à partir d'une exécution.

Le problème des deux approches précédentes vient de la difficulté de généraliser une exécution pour générer un code réutilisable. Considérons l'exemple suivant (la partie de droite est le code assembleur correspondant au code C de gauche) :

```
int sum = 0;
for (int i=0; i<100; i++){
    sum = sum + 5;
}
```

```
  mv r1 0 // r1 = 0
  mv r2 0 // r2 = 0

loop:
  add r1 r1 5   // r1 = r1 + 5
  add r2 r2 1   // r2 = r2 + 1
  sub r3 r2 100 // r3 = r2 - 100
  bnez r3 loop  // if r3!=0 goto loop
```

Lors de l'exécution de la première itération, l'étape de renommage des registres va attribuer un registre pour la lecture de *r1* et un registre pour son écriture à la fin de la boucle. Le mécanisme d'exécution dans le désordre (tel qu'il est fait dans les processeurs superscalaires) n'a aucun moyen de savoir que l'exécution en cours est une boucle et n'a donc pas de raison d'utiliser le même registre pour ces deux attributions. Ainsi, si le code est réutilisé, l'exécution sera incorrecte. Pour réutiliser du code généré grâce à une exécution donnée, il faut donc complexifier le mécanisme d'exécution dans le désordre pour détecter ces problèmes et les résoudre. Ces mécanismes devront détecter les différents blocs de base et retrouver les registres qui doivent être conservés par le processus de renommage. Les articles précédents ne détaillent pas les mécanismes utilisés pour résoudre ces problèmes.

Dans cette partie, nous avons donc étudié les différentes approches pour résoudre le problème d'ordonnancement dynamique des instructions. Comme nous l'avons vu, la méthode de Tomasulo est la solution la plus efficace en raison de l'utilisation de matériel dédié. Cependant, l'idée même de l'approche oblige à réordonnancer le bloc de base à chaque exécution. L'approche JIT est un bon compromis car le résultat peut être sauvegardé mais le temps nécessaire à l'ordonnancement logiciel est beaucoup plus important. Nous allons maintenant présenter notre approche basée sur une compilation JIT assistée par du matériel spécialisé.

3 Ordonnancement dynamique matériel

Dans cette partie, nous allons présenter notre accélérateur matériel pour la compilation JIT ciblant un processeur VLIW. Comme nous l'avons expliqué précédemment, les parties critiques de la compilation dynamique pour VLIW sont les phases d'ordonnancement des instructions sur les différentes unités logiques du processeur VLIW et l'allocation des registres. Nous proposons donc un accélérateur matériel permettant d'ordonnancer les instructions d'un *bytecode* sur les unités d'exécution du VLIW. Dans cette partie, nous allons commencer par présenter le *bytecode* utilisé, puis l'ordonnanceur matériel réalisé. Nous parlerons également de la manière dont est géré le problème de l'allocation de registres.

3.1 Présentation du bytecode utilisé

Un des intérêts majeurs de la compilation JIT par rapport à une approche d'ordonnancement dynamique telle que celle de Tomasulo est la présence d'informations de haut niveau sur le programme en cours d'exécution. En effet, l'approche de Tomasulo travaille au niveau des instructions et n'a donc aucune vision globale du programme. Notre approche étant basée sur une compilation JIT, nous utilisons un *bytecode* qui nous permet d'avoir des informations de haut niveau supplémentaires (c'est à dire d'autres métadonnées).

Il existe plusieurs *bytecodes* basés sur des idées similaires :

- Le *bytecode* java qui est basé sur l'utilisation d'une machine à pile et permet d'offrir la portabilité du java.
- Le *bytecode* Dalvik [13], utilisé dans les téléphones Android, est une version du *bytecode* java basé sur un système de registres et non plus de pile. Ce système lui permet d'être plus léger car l'utilisation de registres nécessite moins d'instructions pour certaines opérations.
- Le *bytecode* CIL [26] qui est également basé sur l'utilisation d'une pile et est utilisé par Microsoft pour ses langages de haut niveau tel que le C# ou le VB.

Ces trois *bytecodes* ont été développés pour fonctionner sur des machines virtuelles, afin d'être interprétés. Pour cette raison, leur format est très proche de celui d'un assembleur de haut-niveau à sémantique séquentielle.

Dans le cadre de notre travail, nous souhaitons utiliser un *bytecode* plus adapté pour la *split compilation*. Le principe de la *split compilation* est de réaliser les optimisation coûteuses de façon statique et les optimisations dépendantes de l'architecture de manière dynamique. Pour cela, nous souhaitons utiliser un *bytecode* proche d'une représentation intermédiaire de compilateur et non d'un assembleur. Un tel *bytecode* posséderai toutes les informations de haut niveau permettant de simplifier la mise en oeuvre des optimisations lors de la phase dynamique. C'est pour cette raison que nous avons fait le choix de définir notre propre *bytecode* plutôt que d'utiliser une version existante. Ce *bytecode* est basé sur des registres virtuels et a la spécificité de ne pas avoir de sémantique séquentielle. En effet, il permettra d'encoder le graphe de flot de données de chaque bloc de base en spécifiant les successeurs des différents noeuds. Une telle représentation va donc simplifier l'ordonnancement des instructions puisque les dépendances sont immédiatement accessibles.

Le *bytecode* contient également d'autres informations de haut niveau que nous allons présenter dans la suite de cette partie.

Priorité d'une instruction Le *bytecode* contient la priorité de chaque instruction. Cette notion de priorité sera utilisée pour privilégier l'ordonnancement de certaines instructions par rapport à d'autres. Le choix de la mesure de la priorité d'une instruction dans un bloc de base est un paramètre important qui joue sur la qualité de l'ordonnancement calculé. Cependant, étant donné que notre algorithme de JIT ne fera qu'utiliser la valeur de priorité donnée dans le *bytecode*, il est facile de changer de mesure. Notre implémentation matérielle sera donc indépendante de la mesure de priorité choisie.

Dans la version actuelle du *bytecode*, nous avons choisi d'utiliser la mobilité comme mesure de priorité. Pour définir la mobilité d'une instruction, il faut se baser sur les ordonnancements ASAP et ALAP des instructions du bloc de base.

L'ordonnancement ASAP (*As Soon As Possible*) d'un bloc de base correspond à l'ordonnancement des instructions dans lequel chaque instruction est faite le plus tôt possible, sans aucune contrainte sur le nombre de ressources disponibles. Dans la suite, nous noterons ASAP(i) le cycle où l'instruction i est exécutée dans l'ordonnancement ASAP.

L'ordonnancement ALAP (*As Late As Possible*) d'un bloc de base correspond à l'ordonnancement des instructions dans lequel chaque instruction est ordonnancée aussi tard que possible, tout en conservant le même nombre de cycle total que l'ordonnancement ASAP. Dans la suite, nous noterons ALAP(i) le cycle où l'instruction i est exécutée dans l'ordonnancement ALAP.

Maintenant que nous avons défini les ordonnancements ASAP et ALAP d'un bloc de base, nous pouvons définir la mobilité d'une instruction :

La mobilité d'une instruction i, notée $m(i)$, est la différence entre son cycle dans l'ordonnancement ALAP et son cycle dans l'ordonnancement ASAP. Autrement dit, $m(i) = ALAP(i) - ASAP(i)$. Intuitivement, la mobilité d'une instruction correspond au nombre de cycles où il est possible de l'ordonnancer tout en conservant la solution optimale. Ainsi, une instruction sur le chemin critique aura une mobilité nulle.

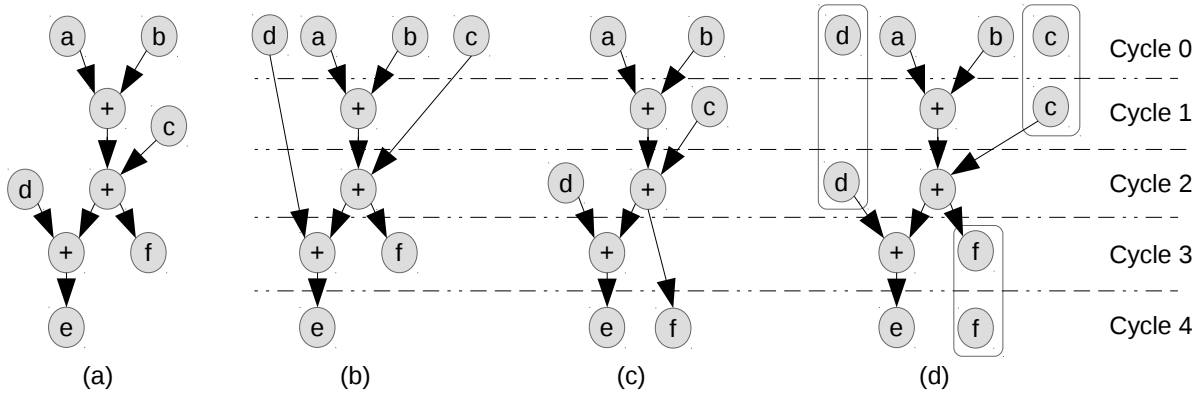


FIGURE 12 : Exemple de calcul de mobilité. La partie (a) représente le graphe d'un bloc de base, la partie (b) correspond à l'ordonnancement ASAP de ce graphe, la partie (c) correspond à l'ordonnancement ALAP et la partie (d) montre la différence entre ces deux ordonnancements.

La figure 12 illustre ce calcul de mobilité. Nous pouvons y voir le calcul des ordonnancements ASAP et ALAP d'un graphe donné, dans le cas où chaque instruction s'exécute en un cycle. La partie (d) du schéma montre ainsi la mobilité de ces instructions : le noeud d a une mobilité de 2, les noeuds c et f ont une mobilité de 1 et tous les autres ont une mobilité nulle et font donc partie du chemin critique du bloc de base.

Ainsi, pour donner une priorité à une instruction, nous nous basons sur la mobilité : plus une instruction est mobile, moins elle est prioritaire. Les priorités seront donc réparties selon cette valeur, lors de la génération du *bytecode*.

Notion de registre virtuel Le *bytecode* est indépendant du nombre de registres physiques présents sur le processeur, et utilise donc des registres virtuels. Ces registres virtuels ne pourront être écrits qu'une seule fois, suivant le même principe que la forme SSA (*Single static assignement*). Cette forme permet de faciliter l'allocation de registres et d'éviter les dépendances de noms. Lors de la compilation dynamique, la phase d'allocation de registre permettra de remplacer ces registres virtuels par des registres physiques du processeur. Une différence avec la forme SSA est que le *bytecode* n'utilise pas de ϕ. Ce symbole permet de symboliser qu'une variable peut avoir été modifiée dans deux blocs de base différents. Par exemple, une variable utilisée dans une boucle peut avoir été modifiée avant la boucle ou dans l'itération précédente. Cependant, comme nos registres virtuels ne sont que locaux (à l'intérieur d'un bloc de base), notre *bytecode* n'utilise pas de ϕ.

Nombre de dépendances Le *bytecode* contient également le nombre de prédécesseurs d'une instruction donnée, permettant de savoir rapidement si l'instruction est prête à être exécutée. En effet, le graphe de flot de données étant encodé grâce à la liste des successeurs de chaque instruction, il suffit de décrémenter le nombre de dépendances des successeurs lorsqu'une instruction est finie. Si ce nombre passe à zéro, l'instruction est prête à être exécutée. Ces dépendances sont à la fois les dépendances de données (besoin de la valeur des opérandes de l'instruction) et les dépendances mémoires (les écritures et lectures mémoires doivent être faites dans un ordre donné).

Le format binaire d'une instruction du *bytecode*, sur 128 bits, est le suivant :

0 ... 41	42 ... 49	50 ... 57	58 ... 60	61 ... 63	64 ... 127
Instruction	*#dep*	*Prior*	$\#succ_{total}$	$\#succ_{donnes}$	*successeurs*

Les 42 premiers bits servent à encoder l'instruction. Viennent ensuite le nombre de dépendances, la priorité de l'instruction, le nombre de successeurs total, le nombre de successeurs qui utilisent le résultat produit puis la liste de tous les successeurs.

L'exemple suivant est le *bytecode* généré pour le bloc de base décrit sur la figure 12, en supposant que les variables a, b, ... f sont stockées dans les registres 256, 257, ... 261. Nous pouvons y voir l'encodage du graphe de flot de données, le nombre de dépendances et la priorité de chaque instruction.

```
0 - ld %0 %256 %0 dependencies = 0 priority = 51  successors : 6
1 - ld %1 %257 %0 dependencies = 0 priority = 153 successors : 4
2 - ld %2 %258 %0 dependencies = 0 priority = 153 successors : 4
3 - ld %3 %259 %0 dependencies = 0 priority = 102 successors : 5
4 - add %4 %2 %1  dependencies = 2 priority = 255 successors : 5
5 - add %5 %3 %4  dependencies = 2 priority = 255 successors : 6 8
6 - add %6 %5 %0  dependencies = 2 priority = 255 successors : 7
7 - SET %6 %261   dependencies = 1 priority = 255 successors :
8 - SET %5 %260   dependencies = 1 priority = 255 successors :
```

Nous avons fait le choix d'utiliser un *bytecode* sur mesure et non pas une version existante afin d'avoir toutes les informations simples d'accès. Il serait cependant possible de nous baser

sur un *bytecode* existant au prix d'une phase de préparation dans laquelle l'outil calculerait les informations nécessaires à l'ordonnancement. Dans la suite, nous présenterons cette étape d'ordonnancement et la manière dont les informations présentes dans le *bytecode* sont utilisées.

3.2 Algorithme de *list scheduling*

L'algorithme qui sera utilisé pour réaliser l'ordonnancement sous contraintes de ressources est appelé *list scheduling* [20]. C'est une heuristique gloutonne au problème d'ordonnancement sous contraintes de ressources. Comme nous l'avons déjà vu dans la partie 2, le principe de cet algorithme est de trier les instructions prêtes selon une priorité et ensuite d'exécuter, à chaque cycle, l'instruction prête ayant la plus forte priorité. Plus concrètement, le fonctionnement du *list scheduling* est décrit sur l'algorithme 1. Nous pouvons y distinguer trois étapes clés :

- L'étape d'insertion des instructions dans la liste triée va parcourir les instructions qui n'ont plus de dépendances (instructions prêtes) et les ajouter dans la liste triée. Cette insertion dans une liste triée a un coût linéaire en fonction de la taille de la liste ($O(n)$) et se fait selon la valeur de priorité donnée dans le *bytecode*.
- L'étape d'assignation des instructions aux unités fonctionnelles consiste simplement à prendre les instructions les plus prioritaires et de les assigner aux unités d'exécution. Cette étape a un coût constant puisque la liste des instructions prêtes est triée (lecture de la tête de liste et écriture dans la table de réservation). A la fin de cette assignation, l'instruction correspondant à ce cycle est générée et ajoutée au binaire.
- Enfin, la dernière étape est de parcourir les instructions qui viennent d'être exécutées afin de réduire le nombre de dépendances de leurs successeurs. Étant donné que le *bytecode* utilisé comporte la liste des successeurs et le nombre de dépendances de l'instruction, cette étape consiste simplement à décrémenter la valeur du nombre de dépendances non résolues pour chaque successeur. Les instructions n'ayant plus de dépendances sont alors ajoutées à la liste des instructions prêtes.

L'algorithme d'ordonnancement par liste a été choisi car il offre un bon compromis entre le temps d'exécution et la qualité du résultat. Nous avons choisi un algorithme qui travaille uniquement au niveau d'un bloc de base car il pourra également être utilisé comme brique de base pour développer des algorithmes plus complexes. Par exemple, certain outils de JIT [31] comportent une étape d'ordonnancement pour réaliser du *software pipelining*, une transformation de compilation permettant d'exploiter très efficacement le parallélisme d'instructions dans les boucles. Dans cette application, le temps d'ordonnancement correspond à 40% du temps de compilation total. Ainsi, une version accélérée de l'ordonnancement d'un bloc de base peut également bénéficier à des optimisations de compilation très puissantes.

3.3 Implémentation matérielle de l'ordonnanceur

Dans cette partie, nous allons présenter la mise en oeuvre de l'implémentation matérielle de l'algorithme de *list scheduling*. Pour cela, nous avons utilisé la synthèse de haut niveau (HLS : *High Level Synthesis*).

En effet, pour réaliser à la main une architecture (c'est à dire pour la décrire en utilisant un langage de description matérielle), il est nécessaire d'écrire conjointement l'unité fonctionnelle et l'unité de contrôle de l'architecture. L'unité fonctionnelle correspond à la partie calculatoire de

```
while nbOrdonnancées ≠ nbInstructions do
    for instruction ∈ InstructionsPrêtes do
        ajouter(listeTriée, instruction)
    end
    for uf ∈ unitéesFonctionnelles do
        instruction := tête(listeTriée);
        tableDeReservation[cycle][uf] = instruction;
        nbOrdonnancées := nbOrdonnancées + 1
    end
    écrireBinaire();
    cycle++;
    for instruction ∈ instructionsFinies() do
        for successeur ∈ instruction.successeurs() do
            nbDépendences[successeur] := nbDépendences[successeur] - 1;
            if nbDépendences[successeur] = 0 then
                ajouter(instructionsPrêtes, successeur)
            end
        end
    end
end
```

Algorithm 1: Algorithme de *list scheduling*

l'architecture : il faut y gérer les différents chemins de données, les multiplexeurs et le placement des registres. L'unité de contrôle sert à orchestrer toute cette partie : elle est composée d'un automate d'état fini (FSM :*Finite State Machine*) qui contrôle les différents multiplexeurs.

De plus, le développement de ces parties se fait généralement en VHDL, qui est un langage particulièrement verbeux. Le fait d'écrire ce code est une tâche longue et fastidieuse (car le niveau d'abstraction est faible) qui est bien souvent source d'erreurs.

Prenons par exemple le code C suivant :

```
adx = 0
while(1) {
    X[adx]=read_fifo();
    y=0; i=0;
    adx=(adx++) % 8;
    while(i<8) {
      y=y+X[adx]*C[i];
      adx=(adx++)%8;
      i++;
    }
    write_fifo(y);
}
```

Pour réaliser une implémentation matérielle de cette fonction en VHDL, il faut décrire conjointement l'unité de contrôle et l'unité d'exécution correspondant au code C. La figure 13 donne un aperçu de ces deux entités : l'unité de contrôle est un automate d'état fini qui contrôle les différentes entités de l'unité d'exécution. L'unité d'exécution, elle, est constitué des différents opérateurs (unités de calcul et mémoires) qui interviendront dans le calcul. Cette description en VHDL prend environs 700 lignes.

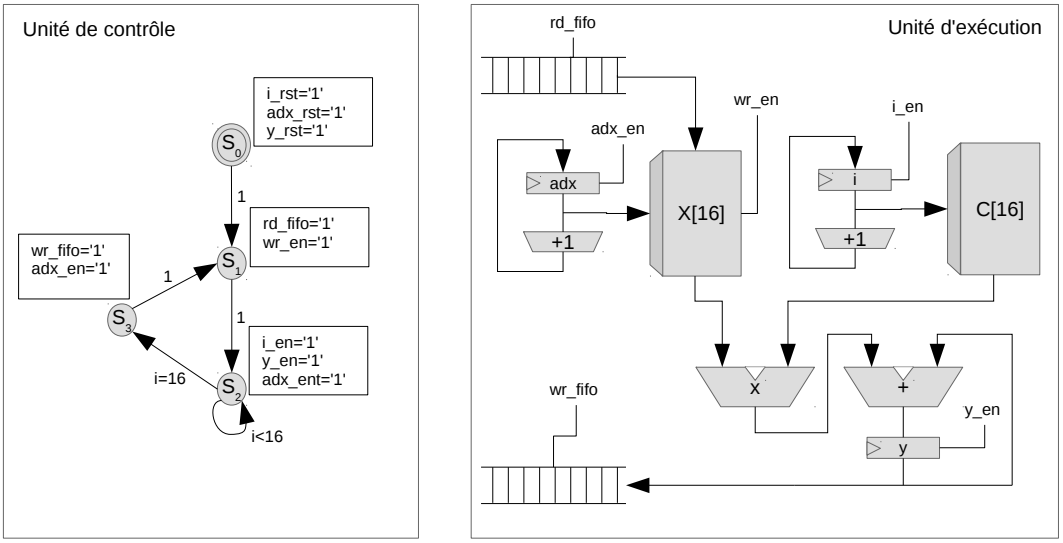


FIGURE 13 : Architecture à décrire pour implémenter l'exemple donné : la partie de gauche représente l'untié de contrôle (UC) et celle de droite l'unité d'exécution (UE).

Au contraire, la synthèse de haut niveau va permettre d'inférer le comportement de l'architecture grâce au code C et va générer automatiquement ces deux entités.

Ainsi, la synthèse de haut niveau permet de développer des architectures beaucoup plus rapidement et en évitant certaines erreurs dues à un code trop bas niveau [17]. De plus, même si la solution ainsi obtenue est de moindre qualité (c'est à dire moins performante ou utilisant une surface de silicium plus importante), le gain en temps de conception permet d'explorer un nombre important de solutions pour tenter de trouver la plus efficace. Ce principe est résumé sur la figure 14. Cet exemple montre la conception de matériel spécialisé pour calculer la somme suivante :

$$\sum_{i=0}^{i<N} A[i] * c[i]$$

Comme le montre la figure, il y a plusieurs solutions possibles, allant de la solution où chaque opération correspond à un opérateur, permettant ainsi de faire le calcul en un cycle (architecture de gauche), à celle ou il n'y a qu'un opérateur de chaque type qui sera réutilisé plusieurs fois (architecture de droite). La surface et la latence d'exécution de ces solutions sont calculées et ces valeurs sont placées sur une courbe de Pareto (partie basse de la figure). L'utilisateur peut ainsi choisir un compromis temps/surface qu'il juge acceptable. En règle général, l'utilisateur va contrôler la solution générée à travers des contraintes passées à l'outil de HLS. Dans notre exemple, le nombre de multiplieurs et d'additionneurs est une contrainte qui permettra de contrôler la solution générée.

Un point important à préciser est que le code C qui est pris en entrée des outils de synthèse de haut niveau doit être codé avec une vision architecturale du problème. En effet, la manière dont le code est écrit peut changer complètement l'architecture obtenue. Du code sous optimal en C peut générer une meilleure architecture [14].

Pour réaliser l'ordonnanceur, nous utiliserons l'outil de HLS Catapult [3].

Implémentation matérielle du *list scheduling* Comme nous pouvons le voir dans l'algorithme 1, le *list scheduling* est principalement composé d'une boucle critique. Il y aura donc très

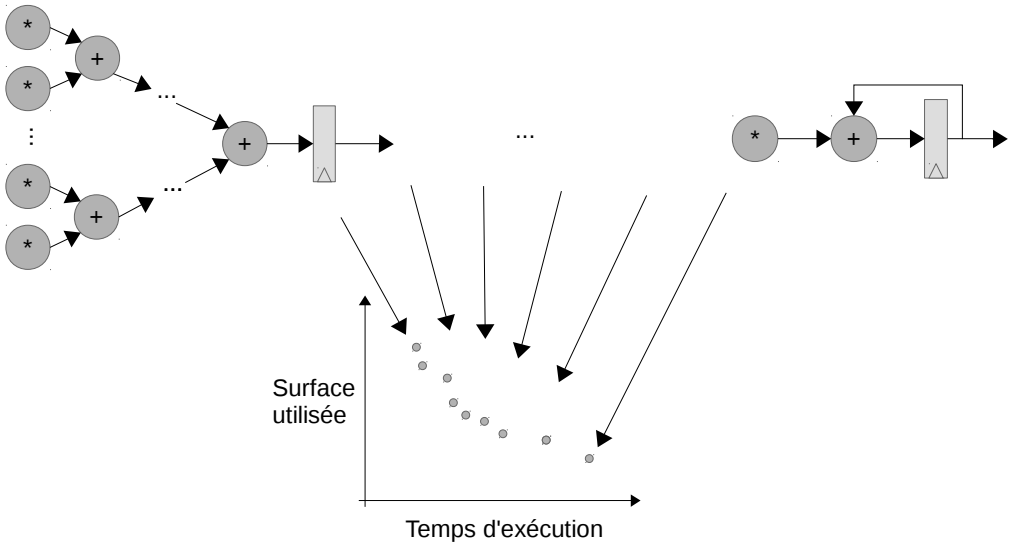


FIGURE 14 : Illustration des architectures possibles pour calculer une somme et de leur placement sur une courbe de Pareto.

peu de parallélisme de calcul à exploiter. Ainsi, pour rendre l'implémentation efficace, il faudra exploiter le parallélisme mémoire (paralléliser les accès aux différentes mémoires) et donc trouver des structures de données adaptées à notre problème. La figure 15 représente les accès mémoires réalisés par une version de l'algorithme où les priorités sont codées sur un bit. Ainsi, l'insertion dans la liste triée se fera soit en tête soit en queue de la liste. Les structures de données utilisées dans l'algorithme sont les suivantes :

- Les listes triées des instructions prêtes à s'exécuter (LT et LTs). Nous utilisons dans notre version une liste pour chaque catégorie d'instruction (branchement, accès mémoire, opération arithmétique simple et opération arithmétique complexe) afin de faciliter l'attribution des instructions aux unités fonctionnelles. Ces listes triées seront implémentées sous la forme d'une liste chainée où chaque élément connaît le nom de son successeur. De plus, le nom de la tête et de la queue des listes est stocké dans des registres. Ainsi, faire une insertion en tête ou en queue de liste est fait en temps constant.
- La table de réservation permet de connaître les instructions exécutées à un cycle donné. La taille de cette table correspond à la longueur maximale des pipelines utilisés. La table de réservation est implémentée comme une RAM à un port de lecture/écriture.
- Le bytecode et le binaire généré sont stockés dans des mémoires externes afin d'être lus par une autre entité du système (par exemple le processeur qui doit exécuter le binaire généré).
- Le nombre de dépendances de chaque instruction est stocké dans une mémoire RAM appelée *dep*.
- Les instructions qui doivent être insérées dans les listes triées sont stockée dans une mémoire FIFO (*First In First Out*).

L'algorithme de *list scheduling* peut se découper en quatre étapes, illustrées sur la figure 15, et détaillées ci-dessous :

- L'insertion des instructions dans les listes triées correspond à la boucle qui va parcourir chaque instruction de la FIFO et les insérer soit en tête, soit en queue des listes triées.

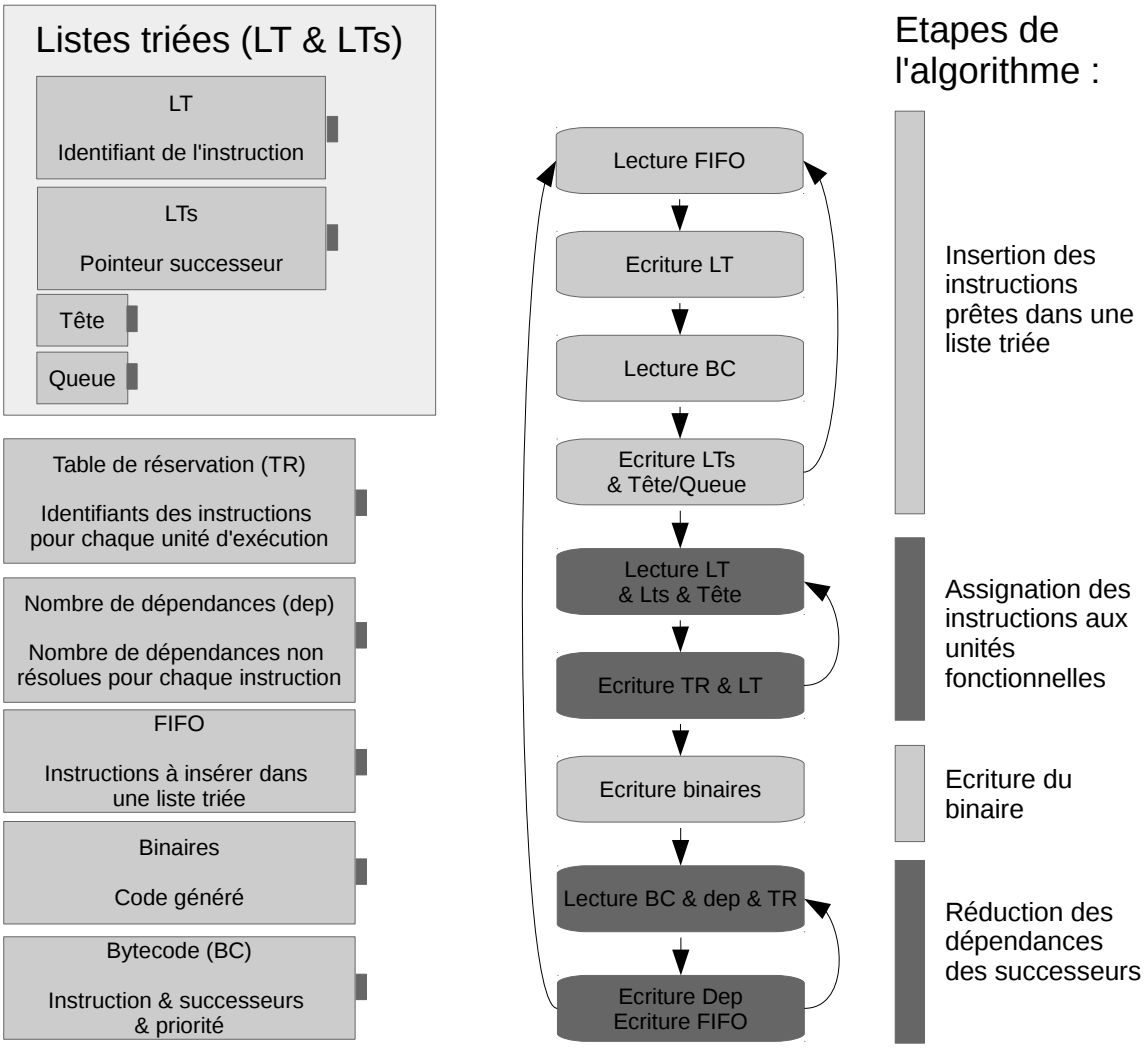


FIGURE 15 : Représentation des accès mémoires lors des différentes étapes du *list scheduling*. Les flèches correspondent au flot de contrôle, c'est à dire aux différents chemins d'exécution possibles.

- L'assignation des instructions de plus forte priorité aux unités fonctionnelles consiste à lire les têtes de listes, remplir la table de réservation et mettre à jour les listes triées.
- L'écriture du binaire correspond à la traduction des instructions vers le format binaire supporté par l'architecture cible.
- La réduction du nombre de dépendances des successeurs consiste à lire dans la table de réservation les instructions qui viennent d'être exécutées, trouver leurs successeurs dans le *bytecode* et réduire le nombre de dépendances. Si une instruction n'a plus de dépendances, elle est ajoutée dans la FIFO.

Sur la figure 15, nous pouvons remarquer que les listes triées sont construites avec deux blocs mémoires indépendants : un pour les identifiants d'instructions (LT) et un pour le nom des successeurs (LTs). Cela permet de pipeliner l'exécution de la boucle d'insertion des instructions prêtes c'est à dire de commencer l'itération suivante avant que l'itération courante ne soit terminée. Nous illustrons ce principe sur la figure 16 qui représente l'exécution de cette boucle de manière pipelinée. En effet, une nouvelle itération est commencée à chaque cycle. Cela est possible car il n'y a pas de dépendance de données entre les différentes itérations et parce que

le nombre de ressources est suffisant. Si les mémoires contenant les identifiants et le nom des successeurs n'avaient pas été séparées, il n'aurait pas été possible de faire mieux qu'une nouvelle itération tous les trois cycles car le deuxième bloc devrait être fait après le quatrième en raison des deux lectures sur LT (dépendance mémoire). L'exécution pipelinée de la boucle permet donc d'insérer n instructions en $n + 3$ cycles.

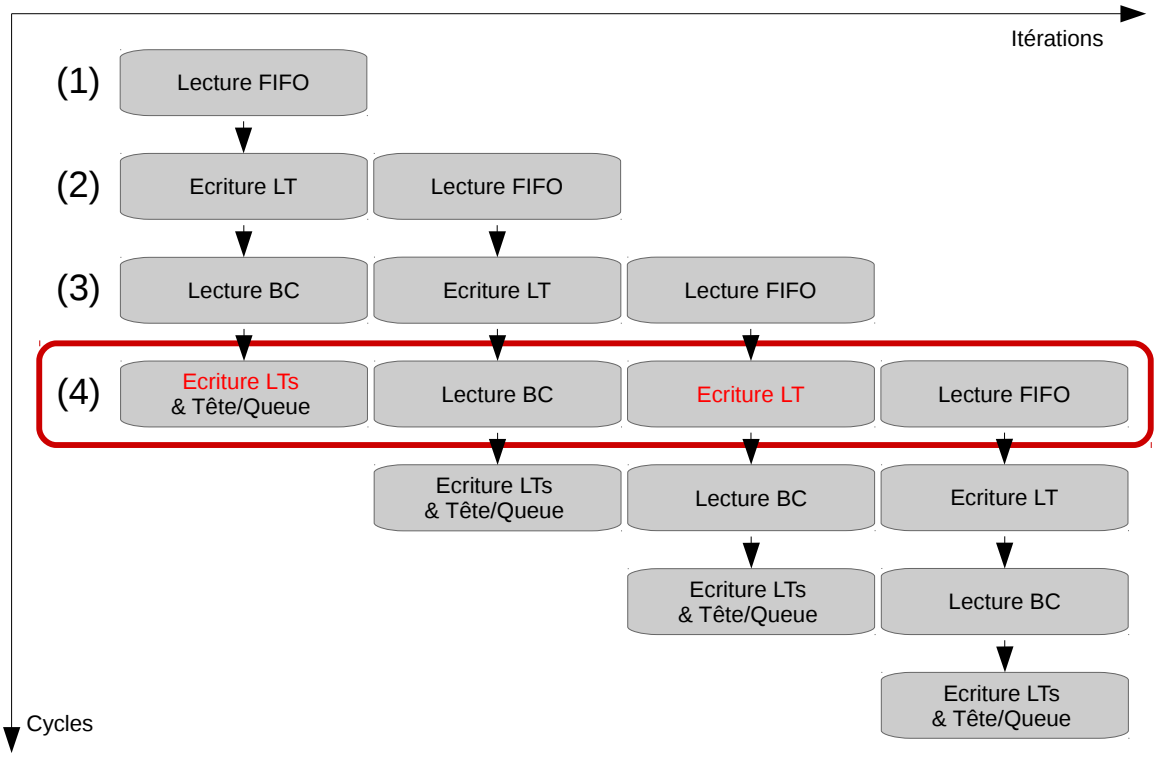


FIGURE 16 : Illustration de l'exécution pipelinée de la boucle d'insertion des instructions prêtes : parce qu'il n'y a pas de conflit entre l'étape (2) et l'étape (4) de la boucle, il est possible de commencer une nouvelle itération à chaque cycle de l'exécution.

Les dépendances mémoires empêchent d'exécuter les autres boucles de manière pipelinée. Cependant, l'architecture sera toujours capable d'exploiter le parallélisme d'instructions en effectuant plusieurs accès à différentes mémoires en même temps.

Notre accélérateur matériel pour l'ordonnancement a deux modes de fonctionnement : l'un utilise la valeur des priorités du *bytecode* encodées sur 8 bits tandis que l'autre n'utilise que le bit de poids fort pour décider si l'insertion se fait en tête ou en queue de liste (et peut être pipelinée comme vu précédemment). Ainsi, la deuxième version est un mode d'ordonnancement plus rapide, générant un ordonnancement moins performant pour les cas où le bloc de base ne fait pas partie d'un point chaud du programme, c'est à dire qu'il ne va pas être exécutée un grand nombre de fois.

Dans cette partie, nous avons présenté les principes majeurs de l'implémentation matérielle de notre accélérateur pour l'ordonnancement sous contraintes de ressources. Un point essentiel de notre approche est l'utilisation d'outils de synthèse de haut-niveau car réaliser un ordonnanceur matériel en VHDL serait particulièrement complexe.

3.4 Allocation de registres

Une autre étape de la compilation dynamique est critique et très dépendante de l'architecture cible : l'allocation de registres. Dans notre situation, ce problème consiste à allouer un registre physique à chaque registre virtuel. Ce problème est fortement lié au problème d'ordonnancement des instructions : si l'allocation est réalisée avant l'ordonnancement, cela peut introduire des dépendances (de nom) supplémentaires lorsqu'un même registre est utilisé pour deux instructions. Au contraire, si l'ordonnancement est fait avant l'allocation, le placement des instructions peut forcer l'utilisation d'un nombre important de registres. Ce lien entre les deux problèmes rend donc l'allocation de registres complexe dans notre contexte [18].

L'approche classique pour l'allocation de registres [4] consiste à générer un graphe d'interférences à partir des informations de durée de vie des différentes variables du programme. Ensuite, un algorithme de coloration de graphe permet de connaître le nombre de registre nécessaires pour faire cette allocation (nombre chromatique). Si le nombre de registres nécessaires est supérieur au nombre de registres disponibles sur la machine, il faut déterminer les variables qui seront stockées en mémoire (souvent sur la pile) pour réduire les contraintes sur les registres : c'est la notion de *spill*. Cette méthode est basée sur la durée de vie des variables et correspond donc à une allocation post-ordonnancement.

Dans notre outil, nous utilisons une méthode d'allocation cyclique des registres, similaire à celle de l'outil Jist [1] : les registres sont alloués au moment où l'instruction est ordonnancée. Nous nous plaçons donc dans la situation où l'allocation est faite après l'ordonnancement. Le nombre de registres utilisés est donc sous optimal car l'ordonnancement peut augmenter grandement la pression sur les registres. De plus, le *spill* n'est toujours pas géré par notre outil. Il faudrait ajouter une fonction permettant de déterminer quel registre stocker en mémoire lorsque le nombre de registres disponibles devient trop faible.

Une solution pour réduire le nombre de registre utilisés serait de modifier la fonction de priorité utilisée dans le *bytecode* pour qu'elle réduise cette pression.

Dans cette partie, nous avons donc présenté le *bytecode* utilisé et l'implémentation matérielle de notre outil d'ordonnancement. Cet accélérateur permet de réaliser l'ordonnancement d'un bloc de base du programme. Dans la partie suivante, nous allons voir comment il peut être intégré dans un système hybride logiciel/matériel pour faire de la compilation JIT.

4 La plateforme HyBrid-JIT

Dans cette section nous décrivons notre plateforme matérielle permettant une approche JIT utilisant à la fois du matériel et du logiciel. L'idée de cette plateforme est d'utiliser l'accélérateur pour l'ordonnancement dynamique qui a été décrit dans la partie 3 afin d'accélérer la compilation du *bytecode*. Un processeur dédié se chargera des diverses optimisations sur le *bytecode* et utilisera l'ordonnanceur pour générer le binaire final et un processeur VLIW exécutera les binaires ainsi générés. Dans cette partie, nous allons décrire cette plateforme et les différentes parties qui la composent.

4.1 Processeur pour la compilation

Comme expliqué précédemment, un processeur sera chargé d'optimiser le *bytecode* et d'appeler l'ordonnanceur matériel. Dans ce travail, nous avons fait le choix d'utiliser un processeur différent

pour la compilation et pour l'exécution afin de permettre l'exécution concurrente de ces deux tâches : pendant que le processeur VLIW exécute le binaire, le processeur chargé de la compilation pourra compiler les fonctions ayant une grande probabilité d'être appelées ensuite, ou optimiser les parties critiques du code en cours d'exécution.

Le processeur utilisé dans la plateforme HyBrid-JIT est le Nios II [27], développé par Altera. C'est un processeur 32 bits dont le jeu d'instructions est très proche de celui d'un MIPS [22]. L'intérêt de ce processeur réside dans sa simplicité d'utilisation : grâce aux outils fournis par Altera, l'utilisateur peut facilement ajouter des instructions spécialisées (ici une instruction qui appellera l'ordonnanceur) et de programmer le processeur en C. De plus, l'outil QSys de Altera [6] permet de créer facilement des systèmes embarqués entiers et nous a permis d'assembler les différents composants de notre architecture.

4.2 Le processeur VLIW

Le processeur utilisé pour l'exécution est du type VLIW [15]. Comme nous l'avons dit précédemment, les processeurs VLIW sont très utilisés dans le monde de l'embarqué en raison de leur faible consommation en énergie et de leur forte capacité à exploiter le parallélisme d'instructions. Fisher et al. présentent dans leur ouvrage [15] un modèle général de processeur VLIW appelé VEX (VLIW Example) et utilisé pour les processeurs ST200 (STMicroelectronics) et LX200 (HP). Ce modèle de VLIW a l'avantage d'être accompagné d'un compilateur performant facile à paramétrer. Cependant, il n'existe pas de format binaire définis pour le jeu d'instructions du VEX et pas d'implémentation matérielle. En effet, il est possible de modifier les paramètres suivants :

- Le nombre d'instructions exécutables en parallèle (nombre de voies du processeur VLIW)
- Le nombre d'instructions d'un certain type possible en parallèle
- La latence des différentes opérations (en nombre de cycles machine)
- La taille de la file de registres

Le processeur VEX est un candidat intéressant pour le processeur VLIW de la plateforme HyBrid-JIT. Dans un premier temps, nous avons cependant préféré utiliser notre propre version simplifiée de processeur VLIW, lui aussi généré avec Catapult. Cela nous a permis de simplifier le jeu d'instructions utilisé et donc de simplifier la génération de code lors de la compilation dynamique.

4.3 Organisation globale et compilation

La figure 17 représente l'architecture globale du système utilisé. En plus des éléments précédemment décrits, il comporte :

- Les mémoires qui permettent de passer des informations à l'outil d'ordonnancement matériel (*bytecode* du bloc de base, emplacement des variables globales et registres libres) et de récupérer le résultat de l'ordonnancement (binaires générés). Ces mémoires sont de petites tailles et ne sont pas directement accessibles ni par le Nios, ni par le VLIW,
- Le composant *memCpy* permet de transférer rapidement des données entre différentes mémoires connectées. Son utilisation évite de faire des boucles logicielles pour faire de la recopie de mémoire.

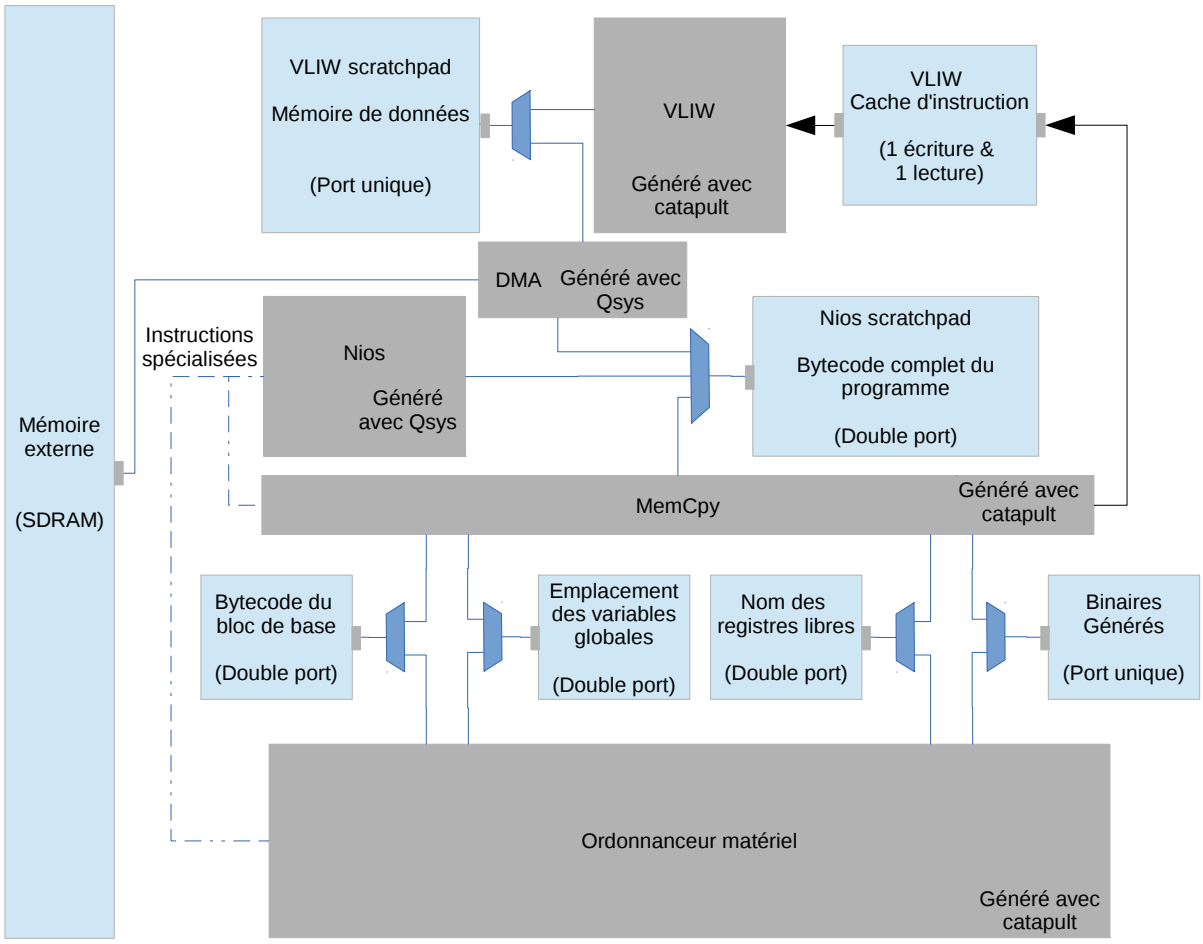


FIGURE 17 : Organisation de l'ensemble du système : représentation des blocs matériels principaux, des mémoires et de la façon dont ils sont connectés

- La mémoire *scratchpad* du Nios est un cache logiciel de la mémoire externe. Le Nios va gérer lui même les données qui doivent être écrites dans ce cache en utilisant le DMA, qui permet de copier rapidement des sections d'une mémoire à une autre.
- Le VLIW accède à ses deux mémoires : une mémoire pour les instructions et une mémoire pour les données. La mémoire d'instructions sera mise à jour par le composant *memCpy* tandis que le *scratchpad* sera contrôlé par le DMA.

Les composants *memCpy* et *Ordonnanceur matériel* sont mis en oeuvre sous la forme d'instructions personnalisées du processeur Nios. Il sera donc possible de les utiliser immédiatement en appelant ces instructions.

La figure 18 illustre le comportement classique d'une compilation JIT avec notre système : lorsque le VLIW souhaite exécuter une fonction, si celui ci n'est pas déjà compilé, le processeur Nios va lancer cette phase de compilation. Pour ce faire, il va charger les informations nécessaires à l'ordonnancement des instructions, bloc de base après bloc de base. Une fois que l'outil d'ordonnancement matériel a terminé son travail, le résultat est copié vers la mémoire d'instructions du VLIW. Lorsque tous les blocs de base ont été compilés, le processeur reprend son travail.

Dans cette partie, nous avons présenté l'organisation globale du système ainsi que son fonctionnement basique. Présenté comme tel, l'outil de JIT permet d'apporter la portabilité du code

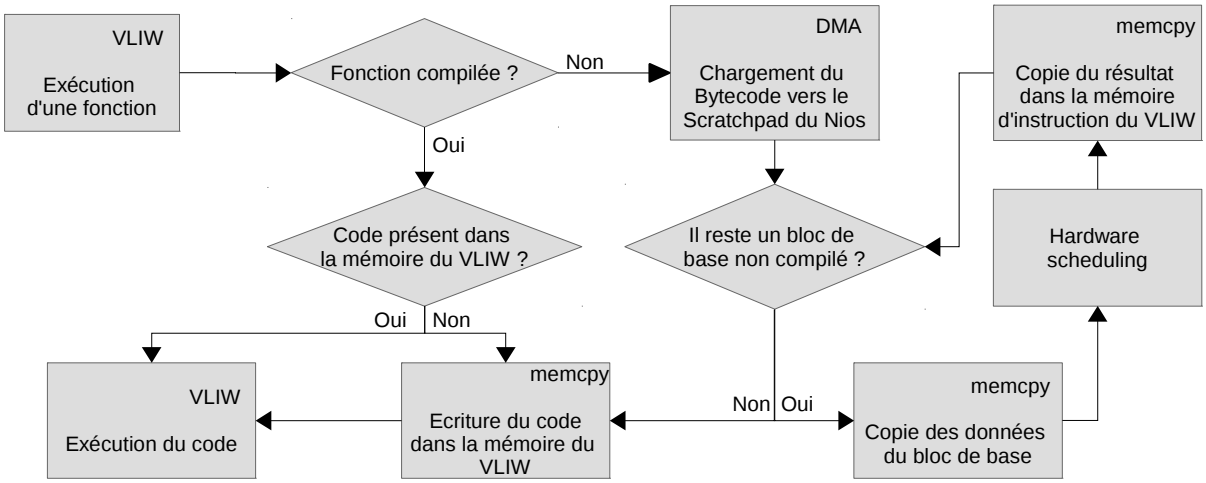


FIGURE 18 : Déroulement d'une étape de compilation JIT. Les cases indiquent à chaque fois qui fait l'action donnée.

en réduisant la pénalité de temps due à la compilation dynamique du code.

5 Étude expérimentale

Dans cette partie, nous allons présenter une étude expérimentale de notre outil de compilation JIT. Cette étude portera principalement sur le temps de calcul de l'ordonnancement sur notre plateforme et sur la qualité de l'ordonnancement ainsi obtenu. L'étude sera faite sur des programmes simples en raison de la faiblesse du compilateur statique. En effet, le compilateur statique, chargé de généré le *bytecode* est pour le moment limité et empêche certains tests. Nous commencerons par présenter les différents temps de compilation obtenus par notre approche et les comparer avec une approche logicielle puis la surface utilisée par notre plateforme de compilation.

5.1 Présentation des programmes étudiés

Dans cette partie, nous présentons les différents programmes utilisés pour évaluer la plateforme de compilation.

- Suite de Fibonacci : ce programme est composé de 5 blocs de base dont le plus important comporte 13 instructions.
- Multiplication de matrices : dans la version utilisée, la boucle interne est déroulée de 12 itérations. Le *bytecode* contient donc 11 blocs de base dont un composé de 193 instructions.
- Transformée en cosinus discret (DCT : *Discrete Cosine Transform*) : le code est composé d'un unique bloc de base de 143 instructions.

Les *bytecodes* précédents ont été générés en utilisant Gecos, une infrastructure de compilation source à source développé dans l'équipe Cairn [10]. Ainsi, le code généré n'est pas totalement optimisé car le compilateur n'est pas fait pour générer du code de VLIW. Utiliser un compilateur ciblant des processeurs VLIW améliorerai les performances grâce aux optimisations spécifiques permettant de faire apparaître du parallélisme d'instructions.

Ainsi, les benchmarks vont permettre d'évaluer les performances de l'ordonnanceur matériel pour des programmes plus ou moins longs et comportant plus ou moins de blocs de base. Dans la section suivante, nous allons présenter les résultats obtenus en mesurant le temps de compilation des différents programmes.

5.2 Temps de compilation

Dans cette section, nous allons présenter les temps de compilation des benchmarks. Nous comparons ici notre approche avec le même algorithme, réalisée de manière logicielle sur le processeur Nios (*Soft JIT*).

L'expérience consistera à ordonnancer les différents blocs de base des programmes et à écrire le binaire dans la mémoire du VLIW. Le processeur Nios ne fera pas d'optimisation sur le code, il se contentera d'appeler l'ordonnanceur avec les bonnes informations et de relier les différents blocs de base entre eux. Les deux versions de l'outil d'ordonnancement sont étudiées ici :

- La version non optimisée de l'ordonnanceur qui ne considère que le bit de poids fort de la priorité des instructions pour décider si l'instruction doit être insérée en tête ou en queue de la liste de priorité.
- La version optimisée qui opère avec une priorité encodée sur 8 bits et qui utilisera des listes triées.

Pour mesurer le temps nécessaire au calcul de l'ordonnancement, nous avons utilisé le composant *performance counter* d'Altera qui permet de connaître au cycle prêt le temps d'exécution d'un programme. L'expérience a été faite sur une FPGA Cyclone IV EP4CE115, cadencée à 50MHz.

Les résultats de cette étude sont représentés dans le tableau 1 : les colonnes *Total* et *Ordo* représentent respectivement le nombre de cycles pour la compilation totale et le nombre de cycles consacrés à l'ordonnancement des blocs de base.

	HyBrid-JIT				Soft JIT			
	Non opt		Opt		Non opt		Opt	
	Total	Ordo	Total	Ordo	Total	Ordo	Total	Ordo
Fibonacci	14 818	4 317	14 780	4 279	189 852	151 229	192 607	153 984
Matrix	34 812	14 337	36 341	15 954	2 293 069	1 947 582	2 730 005	2 353 586
DCT	12 813	4 739	14 764	6 734	1 352 058	1 162 248	1 511 838	1 326 208

TABLE 1 : Temps de compilation des différents programmes en nombre de cycles.

Nous pouvons voir dans ces résultats que la version non optimisée est dans la plupart des cas plus efficace que la version optimisée. Cependant, pour le cas du programme *Fibonacci*, le temps d'exécution de l'ordonnancement optimisé est supérieur au temps de calcul de l'ordonnancement non optimisé. Cela vient du fait que le temps de calcul de l'ordonnancement dépend principalement du nombre de cycle du résultat. Ainsi, la version non optimisée prenant plus de cycle, le temps de calcul sera plus important. D'autre part, nous pouvons voir que dans la version logicielle, l'ordonnancement représente la grande majorité du temps de compilation (entre 80 et 88%) alors que dans les résultats de la plateforme HyBrid-JIT, l'ordonnancement ne représente pas plus de 50% (entre 29 et 46%) du temps de compilation. Il faut cependant rappeler que les compilations qui sont faites ici ne comprennent pas d'optimisations du *bytecode*. Dans une application réelle, la proportion serait sans doute différente mais il est probable que l'ordonnancement reste la tâche principale pour la version logicielle.

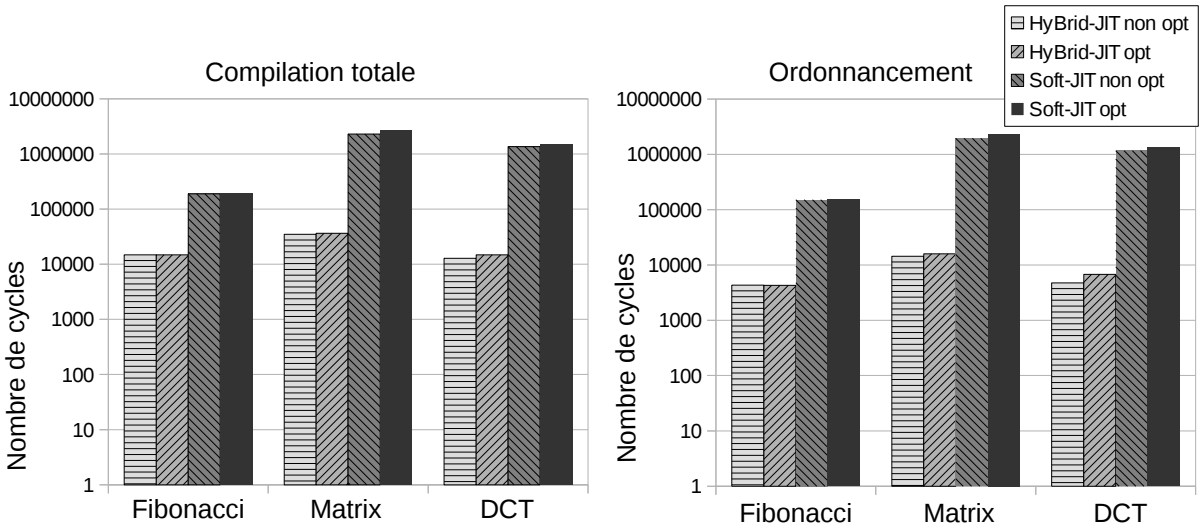


FIGURE 19 : Comparaison des temps d'exécution du tableau 1

Nous avons représenté dans le tableau 2 le gain en temps de compilation entre la version logicielle (*Soft JIT*) et notre version hybride (*HyBrid-JIT*) : nous pouvons voir que le temps de compilation total est réduit au minimum 13 fois et jusqu'à 100 fois pour la DCT. Le gain concernant le temps d'ordonnancement uniquement est évidemment bien plus important (jusqu'à 245 fois plus rapide pour l'ordonnancement du *bytecode* de la DCT). Les écarts de gains s'expliquent par la différence de taille des blocs de base. En effet, le programme *Fibonacci* est composé de 5 blocs de base de taille inférieure à 15 instructions alors que le programme DCT est constitué d'un unique bloc de base de 143 instructions.

	Speed up			
	Non opt		Opt	
	Total	Ordo	Total	Ordo
Fibonacci	12,81	35,03	13,03	35,99
Matrix	65,87	135,84	75,12	147,52
DCT	105,52	245,25	102,40	196,94

TABLE 2 : Accélération du temps de compilation de HyBrid-JIT par rapport à la version logicielle pour les différents programmes.

Nous allons étudier plus précisément l'impact de la taille des blocs de base sur le temps d'ordonnancement. Pour cela, nous allons utiliser un programme constitué uniquement d'une série d'additions en chaînes. Nous allons ici mesurer l'impact :

- de la taille de l'ordonnancement généré. Pour cela, nous allons faire varier la taille de la chaîne maximale de dépendances. Pour le cas extrême, toutes les additions sont présentes sur une unique chaîne d'opérations et doivent donc être ordonnancées les unes après les autres. Dans cette expérience, le nombre d'opérations est constant : seules les dépendances changent.
- du nombre d'instructions à générer. Pour cela, nous allons simplement ajouter des opérations qui n'augmentent pas la durée de l'ordonnancement généré : nous utilisons un programme composé d'une chaîne de 64 additions et nous ajoutons des additions qui ne dépendent pas de cette chaîne afin qu'elles n'impactent pas la taille du chemin critique.

Les mesures sont effectuées lors de l'exécution de la version optimisée de HyBrid-JIT sur les différents *bytecodes*. Le résultat de cette expérience est représenté sur la figure 20 : la figure de droite représente l'impact de la taille du chemin critique sur le temps d'ordonnancement (à nombre d'instructions constant) et la figure de droite représente l'impact du nombre d'instructions sur le temps d'ordonnancement (à taille de chemin critique constante). La mesure du temps d'ordonnancement a été normalisée par rapport au plus faible temps obtenu pour améliorer la lisibilité du schéma.

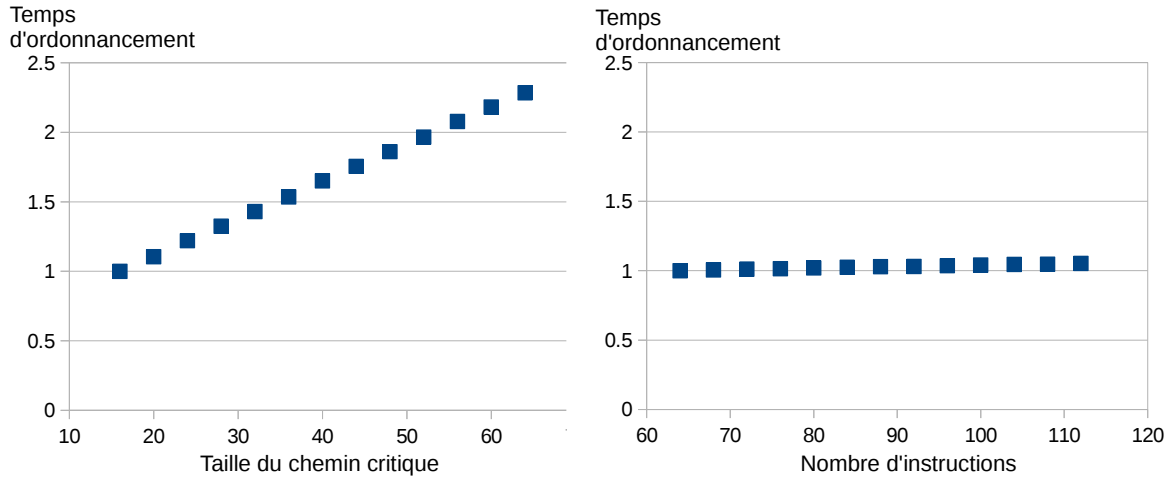


FIGURE 20 : Evolution du temps d'ordonnancement lorsque la taille du chemin critique ou le nombre d'instructions évoluent.

Nous pouvons voir sur ces figures que la taille du chemin critique impact fortement le temps d'ordonnancement allant jusqu'à doubler le temps de compilation lorsque la taille du chemin critique est doublée. Cependant, le nombre d'instructions n'a qu'un impact négligeable sur le temps de compilation : le fait de doubler le nombre d'instructions n'augmente que de 5% le temps d'ordonnancement du bloc de base. Si l'expérience avait été faite avec la version non optimisée, il est probable que l'impact du nombre d'instructions soit encore plus faible car la version non optimisée permet d'ajouter rapidement des instructions dans les listes triées, contrairement à la version optimisée qui doit faire une insertion triée selon la valeur de la priorité.

Le fait que le temps de compilation dépende principalement de la taille de l'ordonnancement généré est intéressant dans le cadre du changement du nombre de voies du processeur VLIW. En effet, plus le nombre de voies du processeur est grand, plus l'ordonnancement sera rapide à calculer. Ainsi, dans le contexte d'un VLIW à nombre de voies variable, plus le besoin en performances est élevé, plus le nombre de voies augmente et réduit ainsi le temps d'ordonnancement. Au contraire, pour les programme qui n'ont pas besoin de performances, le faible nombre de voies augmentera le temps d'ordonnancement.

Concernant les outils existants pour la compilation JIT visant des processeurs VLIW, la comparaison a été complexe car les auteurs donnent très peu d'informations sur le temps de compilation de leur outil. En effet, l'article d'Agosta et al. [1] n'indique aucune valeur des temps de compilation observés et se contentent des temps d'exécution (compilation comprise) sur certains exemples. Cependant, leur approche utilise le même algorithme d'ordonnancement (*list scheduling*) et d'allocation de registres, nous supposons donc que les résultats obtenus sont proches de ceux de notre version logicielle. Pour ce qui est de l'outil de Dupont de Dinechin [7], la seule information disponible est le temps d'ordonnancement de certains blocs de base en fonction de leur taille. Selon cette information, l'outil nécessite environ 100 000 cycles pour ordonnancer un

bloc de base de 200 instructions. Cependant, l'algorithme de *scoreboard scheduling* utilisé dans cet outil est très dépendant du nombre d'instructions et non pas de la taille de l'ordonnancement. Il est donc très difficile de prévoir la qualité de notre outil d'ordonnancement sur de tels blocs de base.

5.3 Surface utilisée

La plateforme telle que présentée dans la section 4 a été assemblée grâce à QSys et compilée sur Quartus II. La tableau 3 donne la surface utilisée par les différents composants. Nous pouvons voir que l'outil d'ordonnancement et le processeur Nios représentent une surface équivalente à celle du VLIW en terme de LUT (*Look up Table*). Concernant la mémoire utilisée, la majorité provient de la mémoire *scratchpad* du Nios ou du VLIW. La taille de ces mémoires peut être modifiée en fonction de l'utilisation. La valeur de la mémoire totale n'est donc pas fixée.

	Total	Nios	Ordonnanceur	VLIW
LUT	31 804	900	12 581	16 138
Registres	6 869	8 374	492	5 438
Memoire (kb)	337,5	10,24	22,656	0

TABLE 3 : Surface utilisée pour chacun des composants principaux de la plateforme.

Dans cette partie, nous avons présenté l'étude expérimentale de notre plateforme de compilation HyBrid-JIT. Cette plateforme permet d'accélerer la compilation jusqu'à 100 fois et le coût en surface est équivalent à celui d'un VLIW à quatre voies.

Conclusion

La consommation énergétique, la portabilité du code et l'adaptation dynamique sont trois facteurs très importants dans le domaine des systèmes embarqués. Comme nous l'avons montré dans la section 1, l'utilisation d'un processeur VLIW dynamiquement reconfigurable et utilisant la DVFS permet de ramener le problème de la consommation énergétique à un problème d'adaptation dynamique. La compilation dynamique permet d'améliorer la portabilité du code et l'adaptation dynamique, mais reste difficile à utiliser pour des architectures embarquées qui sont souvent complexes à programmer efficacement.

Dans ce rapport, nous avons proposé une plateforme de compilation JIT utilisant du matériel spécialisé dans le but d'accélérer la résolution du problème d'ordonnancement sous contraintes de ressources : la plateforme HyBrid-JIT. Ce matériel spécialisé a été réalisé grâce à la synthèse de haut niveau et permet d'ordonnancer un bloc de base en quelques milliers de cycles machine. Cette plateforme permet donc d'utiliser la compilation JIT pour un processeur VLIW.

Ce travail a démontré la faisabilité d'une plateforme de compilation JIT utilisant du matériel spécialisé pour l'ordonnancement. Les solutions qui ont été proposées peuvent cependant être améliorées, notamment sur les points suivants :

- Étude d'un algorithme d'ordonnancement plus puissant et peut-être mieux adapté à une implémentation matérielle. Par exemple, l'algorithme d'ordonnancement SDC [5] se distingue par la qualité de ses résultats et son faible temps de calcul. De plus les algorithmes utilisés (recherche de plus courts chemins dans un graphe) semblent adaptés à une implémentation matérielle.

- Amélioration de l'allocation de registre : en effet, cette partie est pour le moment une version fonctionnelle pouvant être améliorée. Il faudrait étudier des méthodes d'allocation de registres et d'ordonnancement des instructions conjoints.
- Ajout d'optimisations lors de la phase dynamique : le pipeline logiciel est l'une des optimisations de boucle les plus puissantes. Or, un outil de JIT utilise un simple algorithme d'ordonnancement pour réaliser cette optimisation [31]. L'exécution de cet ordonnancement représente 40% de son temps de compilation total. Ainsi, développer ce type d'optimisation pour le compilateur permettrait d'exploiter l'accélération de l'ordonnancement.
- Modification de la structure globale : il est possible de travailler sur l'intégration du matériel pour plusieurs processeurs. En effet, l'outil d'ordonnancement ne sera pas souvent utilisé et peut donc gérer la compilation dynamique pour plusieurs VLIW. Cela ajouterait des problématiques intéressantes de gestion du parallélisme. Le processeur Nios pourrait par exemple gérer le nombre de processeurs alimentés et répartir les différents processus en cours d'exécution.

La suite de ce travail consistera à développer un outil de JIT pour des architectures plus complexes, telles que les architectures CGRA. En effet, étant donné que la solution adoptée fonctionne pour des processeurs de type VLIW, il serait intéressant d'étudier des architectures plus complexes, pour lesquelles il n'y a pas d'outils de JIT fonctionnels.

Comme nous l'avons expliqué dans la section 1, les architectures CGRA permettent la reconfiguration dynamique du chemin de données. Ce type d'architecture est reconfigurable afin de s'adapter efficacement aux différentes applications. Ce degré de spécialisation permet d'obtenir de très bonnes performances et une faible consommation en énergie. Cependant, compiler du code pour le faire fonctionner sur une telle architecture demande de résoudre non seulement un problème d'ordonnancement sous contrainte de ressources mais également un problème de routage des opérandes qui dépend de la topologie du réseau d'interconnexion. Ainsi, le placement des instructions doit être fait en prenant en compte ces deux facteurs.

En plus des performances, l'utilisation d'architectures CGRA permet de multiplier les avantages offerts par la modification dynamique du nombre d'unités fonctionnelles. En effet, changer le nombre de voies d'un processeur VLIW est une solution limitée car l'utilisation d'une file de registres limite le nombre de voies (la surface utilisée évolue de façon quadratique en fonction du nombre de voies). Cependant, avec une architecture CGRA, la surface utilisée croît linéairement en fonction du nombre d'unités d'exécution.

Des algorithmes permettant de résoudre les problèmes d'ordonnancement et de routage sur CGRA existent [23] [29] mais aucune approche n'a été faite pour les résoudre de façon dynamique. Ainsi, il serait intéressant de développer une heuristique efficace et facile à implémenter de manière matérielle afin de réaliser une plateforme de compilation JIT ciblant des architectures CGRA. La transposition de notre approche à des architectures CGRA semble donc être une continuation logique de ce stage.

Références

[1] Giovanni Agosta, Stefano Crespi Reghizzi, Gerlando Falauto, and Martino Sykora. Jist : Just-in-time scheduling translation for parallel processors. *Scientific Programming*, 13(3) :239–253, 2005.

[2] J Adam Butts and Gurindar S Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201. ACM, 2000.

[3] Calypto. Catapult product family datasheet, 2012.

[4] Gregory J Chaitin. Register allocation & spilling via graph coloring. In *ACM Sigplan Notices*, volume 17, pages 98–105. ACM, 1982.

[5] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proceedings of the 43rd annual Design Automation Conference*, pages 433–438. ACM, 2006.

[6] Altera Coporation. Sopc builder user guide, 2003.

[7] Benoît Dupont De Dinechin. Inter-block scoreboard scheduling in a jit compiler for vliw processors. In *Euro-Par 2008–Parallel Processing*, pages 370–381. Springer, 2008.

[8] Alberto Ferreira de Souza and Peter Rounce. Dynamically trace scheduled vliw architectures. In *High-Performance Computing and Networking*, pages 993–995. Springer, 1998.

[9] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing$^{\text{TM}}$ software : using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, pages 15–24. IEEE Computer Society, 2003.

[10] Steven Derrien, Daniel Ménard, Kevin Martin, Antoine Floch, Antoine Morvan, Adeel Pasha, Patrice Quinton, Amit Kumar, and Loïc Cloatre. Gecos : Generic compiler suite.

[11] Marc Duranton, David Black-Schaffer, Koen De Bosschere, and Jonas Maebe. The hipeac vision for advanced computing in horizon 2020. 2013.

[12] Kenneth Eguro and Scott Hauck. Issues and approaches to coarse-grain reconfigurable architecture development. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 111–120. IEEE, 2003.

[13] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[14] Michael Fingeroff. *High-level synthesis blue book*. Xlibris Corporation, 2010.

[15] Joseph A Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing : a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[16] Marc Fleischmann. Longrun power management. *White Paper of Transmeta Corporation*, 2001.

[17] Daniel D Gajski, Nikil D Dutt, and Allen CH. *High-level synthesis*, volume 34. Kluwer Boston, 1992.

[18] James R Goodman and W-C Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd international conference on Supercomputing*, pages 442–452. ACM, 1988.

[19] John L Hennessy and David A Patterson. *Computer architecture : a quantitative approach*. Elsevier, 2012.

[20] Rajiv Jain, Ashutosh Mujumdar, Alok Sharma, and Hueymin Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 686–689. ACM, 1991.

[21] Ravindra Jejurikar and Rajesh Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42nd annual Design Automation Conference*, pages 111–116. ACM, 2005.

[22] Gerry Kane. *mips RISC Architecture*. Prentice-Hall, Inc., 1988.

[23] Wonsub Kim, Yoonseo Choi, and Haewoo Park. Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4) :58, 2013.

[24] Jiayuan Meng, Jeremy W Sheaffer, and Kevin Skadron. Robust simd : Dynamically adapted simd width and multi-threading depth. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 107–118. IEEE, 2012.

[25] Pierre Michaud, Andre Seznec, and Stephan Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 2–10. IEEE, 1999.

[26] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil : Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, pages 213–228. Springer, 2002.

[27] II Nios. Processor reference handbook, 2009.

[28] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor simd : Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 151–160. IEEE Computer Society, 2011.

[29] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[30] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(5) :695–708, 2013.

[31] Hongbo Rong, Hyunchul Park, Youfeng Wu, and Cheng Wang. Just-in-time software pipelining. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 11. ACM, 2014.

[32] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1) :25–33, 1967.

[33] Carlos Villavieja, José A. Joao, Rustam Miftakhutdinov, and Yale N. Patt. Yoga : A hybrid dynamic vliw/ooo processor. 2014.