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

JMQ, un Processeur Java de hautes performances

Maxime PALUS et François ANCEAU

Université Pierre et Marie Curie,
LIP6, 4, place Jussieu,
75252 Paris Cedex 05 - France
maxime.palus@lip6.fr

Résumé
Le langage Java est très utilisé par les concepteurs d'applications mobiles et sans fil. Son exécution sur les processeurs embarqués se heurte à la lenteur de son interpréteur (JVM). Un certain nombre de fabricants se sont donc tournés vers l'exécution directe du code intermédiaire Java (Bytecode) par des processeurs spécialisés. Ces processeurs sont basés sur des machines à pile et intègrent certains mécanismes destinés à accélérer l'exécution du Bytecode. Cet article présente une nouvelle architecture appelée JMQ (Java Machine on Queue) dont la pile d'exécution Java est remplacée par une file. Cette approche permet d'augmenter le parallélisme de l'exécution de Java. La JMQ est comparée à un modèle de machine à pile appelé JMS (Java Machine on Stack) basé sur le PicoJava-II développé par Sun Microsystems. Les premiers résultats montrent une accélération de l'exécution de l'ordre de 16% par rapport à la machine sur pile.

Mots-clés : Java ; Processeur Java.

---

1. Introduction

Java est un langage de programmation orienté objet très connu. Une des clefs de son succès est sa grande portabilité due au fait que les programmes Java sont compilés dans un code intermédiaire appelé Bytecode. Celui-ci est interprété ou exécuté sur un grand nombre de systèmes, il est compact (1,8 octets en moyenne par instruction [PIM96]) et il peut être sécurisé. Java est aujourd'hui l'un des langages les plus utilisés pour développer des applications destinées aux systèmes embarqués. La société Evans Data Corporation déclare que 56% des développeurs d'applications sans fils utilisent J2ME (Java 2 Mobile Edition) [O05]. Les performances de Java sont malheureusement limitées par son interpréteur (Java Virtual Machine : JVM), qui exécute le Bytecode sur un processeur cible. Cette technique est très mal adaptée à l'exécution de Java sur les systèmes embarqués car elle est très lente et accroît les besoins de bande passante avec la mémoire. Plusieurs techniques logicielles et matérielles permettent d'augmenter la vitesse d'exécution des programmes Java, tels que le JIT (Just In Time) [HS96]. Cette technique consiste en une traduction à la volée du Bytecode dans le langage de la machine cible pendant l'exécution. La taille du code est alors multipliée par un facteur 3, alors que les systèmes embarqués ne sont pas pourvus d'une grande quantité de mémoire. La compilation "Off-line" [PR97] et la compilation "Native" consistent respectivement en une recompilation du Bytecode et une compilation directe du programme Java pour une machine cible. Elles permettent d'exécuter les programmes Java aussi vite que les programmes C++. Néanmoins, des compilateurs doivent être développés pour chaque couple système d'exploitation/processeur. De plus, un interpréteur reste nécessaire si l'ont veut pouvoir continuer à exécuter des classes chargées dynamiquement. La philosophie de Java "Compilé une fois, exécuté partout" est alors complètement oubliée.
Le Bytecode peut être utilisé comme jeu d'instructions pour un processeur spécialisé. Les implémentations "classiques" de processeurs Java sont basées sur des machines à pile qui nécessitent un grand nombre de manipulation des opérandes (environs 30% de plus qu'une machine à registres [WW73]). Les processeurs simples nécéssitent plusieurs cycles pour exécuter chaque instruction Java [MS05]. Ces

processeurs sont de faible complexité mais également de faible performance, ils sont généralement destinés à l'implémentation sur FPGA. Pour les processeurs plus complexes, plusieurs techniques d'optimisations matérielles peuvent être utilisées afin d'augmenter la vitesse d'exécution jusqu'à atteindre plusieurs instructions Bytecode par cycle.
Le processeur PicoJava-II [CT97] [GC98] de Sun Microsystems utilise pour ce faire les mécanismes suivants :
– Un buffer d'instructions.
– Une unité de décodage multiple d'instructions.
– Une file circulaire de 64 entrées contenant le sommet de la pile.
Nous disposons de la description vérilog du Picojava_II, nous utiliserons donc le picojava_II comme base de comparaison.
La nature objet de Java provoque une grande fréquence de changement de contextes d'exécution, nous cherchons donc à accélérer l'exécution des instructions qui y sont liées en utilisant deux mécanismes distincts pour stocker la pile d'environnement Java et le mécanisme d'exécution. Nous cherchons également à augmenter le parallélisme des autres instructions en utilisant la technique d'exécution sur file.
Cet article rappelle la technique d'exécution sur file, initialement développée pour un processeur Pascal, puis présente l'architecture d'un processeur Java à exécution sur file appelé JMQ (Java Machine on Queue). Nous verrons que cette technique permet d'augmenter le parallélisme d'exécution des instructions du Bytecode Java.

2. L'exécution du Bytecode

Le Bytecode Java est un langage post-fixé et est, à ce titre, naturellement exécuté sur une pile (Figure 1). Les opérandes sont chargés au sommet de la pile. Lors d'un calcul, les données sont extraites du sommet de la pile puis le résultat est chargé à son sommet. Exemple :
L'expression "(16 - 11) + 12 * 2", dont la forme post-fixée est "16 11 - 12 2 * +", est exécutée comme suit :

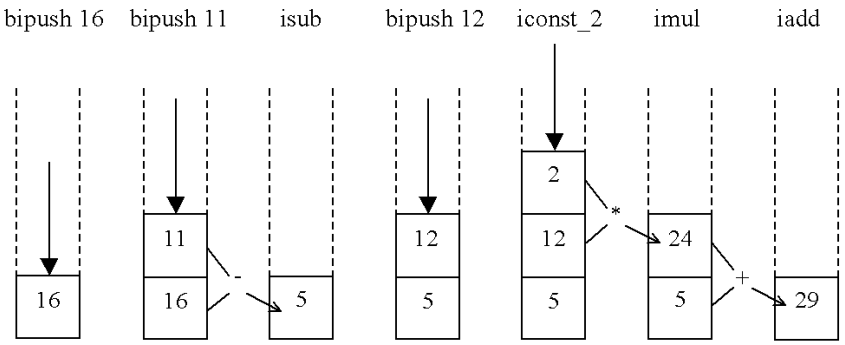

FIG. 1 – Exécution sur pile du Bytecode Java.

L'utilisation d'une pile ne permet pas de dégager du parallélisme dans l'exécution des instructions. La pile est un goulot d'étranglement qui force la séquentialité des instructions.

2.1. Histoire de l'exécution sur file des langages post-fixés

Cette technique d'exécution a été développée par l'équipe de recherche sur l'architecture des ordinateurs de l'IMAG en 1974 [ABS75] [BS75]. Elle fut d'abord utilisée pour un prototype de machine Pascal vers la fin des années 70 [JPS77]. Son application à un processeur Java constitue donc un nouveau projet.

2.2. Principe de l'exécution sur file

Un code post-fixé peut être exécuté dans une file grâce à l'utilisation de pointeurs [ABS75] [BS75] (Figure 2). Il ne s'agit pas de simuler une pile dans une file, mais d'une modification de la technique d'exécution

des expressions post-fixées. Les opérandes sont chargés d'un coté de la file, tandis que les calculs sont effectués de l'autre. Un pointeur P indique le sommet du mécanisme pour la partie calcul. Un pointeur P1 désigne le premier opérande (sommet de la pile virtuelle) et un second appelé P2 accède le second opérande (celui-ci sera effacé après l'opération). Le résultat engendré est replacé dans la file via P1 (ce qui diffère d'une pile) pour "dégager" l'extrémité de la file. Le pointeur P est déplacé avant chaque opération du nombre d'opérandes devant être chargés dans la pile depuis l'opération précédente. Les pointeurs P1 et P2 sont eux-même déplacés afin d'indiquer les prochains opérandes valides. Cette organisation permet de paralléliser le chargement des opérandes et les calculs associés.

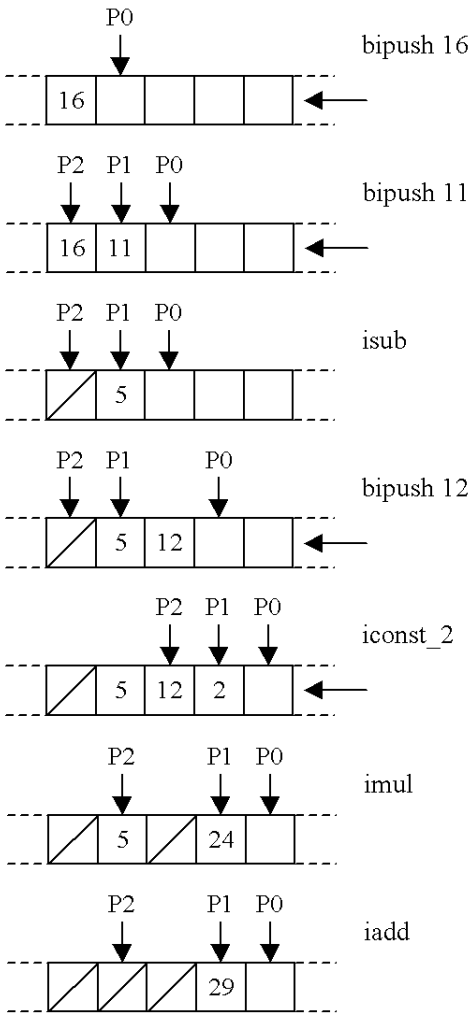

FIG. 2 – Exécution sur file du Bytecode Java.

3. Architecture

Pour évaluer les performances de ce principe d'exécution et du modèle qui l'implémente (JMQ), un modèle de processeur à exécution sur pile appelé JMS (Java Machine on Stack) a été développé. La JMS hérite des techniques d'accélération de l'exécution du Bytecode présentes dans le PicoJava-II de Sun Microsystems.

3.1. Les Pipelines des modèle JMS et JMQ

Le modèle JMS implémente le sommet de sa pile dans une file circulaire de 64 registres 32-bits ainsi qu'une unité de décodage multiple des instructions capable de décoder et de regrouper plusieurs instructions par cycle. Comme le PicoJava-II, la JMS possède un pipeline à 6 étages (Figure 3) :

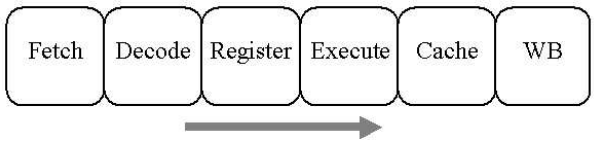

FIG. 3 – Pipe-line de la JMS.

- Fetch : Charge les instructions depuis le cache instructions.
- Decode : Décode jusqu'à 4 instructions comprises dans un maximum de 7 octets.
- Register : Charge les opérandes depuis la pile et les constantes dans le pipeline. Détermine les conditions de by-pass.
- Execute : Effectue les instructions de calcul et de modification de l'environnement. Calcule les conditions de branchement et les adresses de saut. Calcule les adresses d'accès au cache de donnée.
- Cache : Effectue les accès au cache de donnée.
- Writeback : Effectue les écritures dans la pile.

Les mêmes mécanismes sont également utilisés dans le modèle JMQ, mais l'utilisation de la technique d'exécution sur file requiert une nouvelle architecture. Cette architecture peut être vue comme un pipeline d'instructions (Figure 4) ou comme un pipeline de données (Figure 5), l'exécution sur file étant synchronisée par les données.

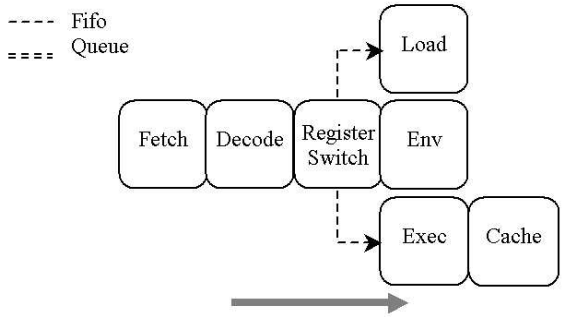

FIG. 4 – Pipe-line instruction de la JMQ.

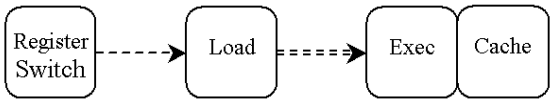

FIG. 5 – Pipe-line de données de la JMQ.

- Fetch : Charge les instructions depuis le cache instructions.
- Decode : Décode jusqu'à 4 instructions comprises dans un maximum de 7 octets.
- Register Switch :
  - Extrait les opérandes depuis la pile et détermine les conditions de by-pass.
  - Aiguille les instructions vers les différents étages du pipeline.
- Environment (Env) : Exécute les instructions liées à l'environnement.
- Load : Remplit la file d'exécution avec des constantes ou des variables extraites de la pile par l'étage Register.

– Execution (Exec) : Effectue les instructions de calcul. Vérifie les conditions de branchement. Calcule les adresses d'accès au cache de données.
– Cache : Effectue les accès dans le cache de données.

Remarque : L'organisation de la machine JMQ rend optionnel l'utilisation de l'étage Cache par les instructions. Le rangement des données, soit dans l'espace des variables locales, soit dans la file d'exécution pourra donc être effectué en fonction de l'instruction exécutée dans les étages EXEC ou CACHE du pipe-line. On considérera par la suite le mécanisme d'écriture des variables locales dans la pile d'environnement comme un étage de pipe-line concurrent des étages EXEC ou CACHE appelé WriteBack.

3.2. Les piles de la JMS et de la JMQ

Les machines Java à pile implémentent les piles d'environnement et d'exécution Java dans une seule et même pile. Le modèle JMS, comme le Picojava-II, stocke le sommet de cette pile dans une file circulaire de 64 registres 32-bits [CG98] (Figure 6) qui permet d'accéder à n'importe quelle donnée dans cette file. Dans le modèle JMQ, ces deux piles sont réalisées par deux mécanismes différents. La pile d'exécution est remplacée par une file d'exécution constituée de registres 64-bits et la pile d'environnement est implémentée de la même manière que dans la JMS (Figure 7).

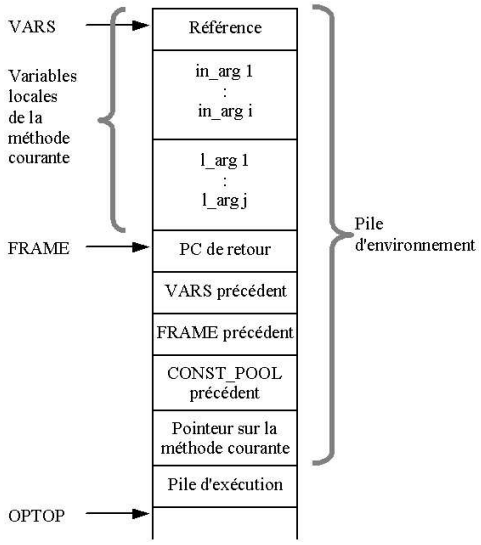

FIG. 6 – Pile de la JMS.

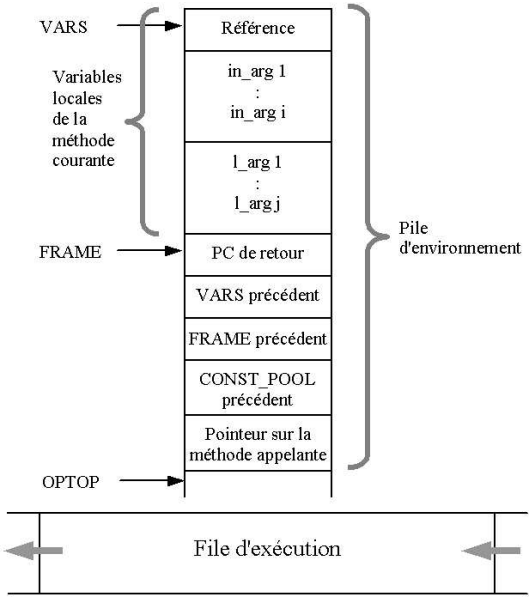

FIG. 7 – Pile et file de la JMQ.

La séparation des mécanismes d'exécution et de gestion de l'environnement permet de dégager du parallélisme entre ces deux types d'instructions qui peuvent désormais être exécutés simultanément. Ces deux mécanismes communiquent à travers deux FIFOs appelées files de dépendances, l'une contient les variables locales extraites de la pile d'environnement, en attente d'être chargées dans la file d'exécution. L'autre contient les adresses des écritures pendantes de variables locales. Ces FIFOs constituent également le mécanisme de by-pass entre les étages LOAD et EXEC_C du pipeline.

3.3. Regroupement d'instructions

Le regroupement d'instructions consiste à décoder et regrouper plusieurs instructions qui utilisent des parties distinctes du matériel afin de les exécuter en parallèle. Ce mécanisme est issu des processeurs PicoJava-I et PicoJava-II [PIM96]. Il permet de réduire le temps d'exécution des programmes et le nombre d'accès à la pile [PJIIM99] (Figure 8). Comme indiqué dans l'introduction, le regroupement

d'instructions, associé à l'utilisation d'une file, permet d'éliminer, en partie, les pertes de performance dues à l'utilisation d'une pile. Le mécanisme de regroupement de la JMS est le même que celui du PicoJava-II.

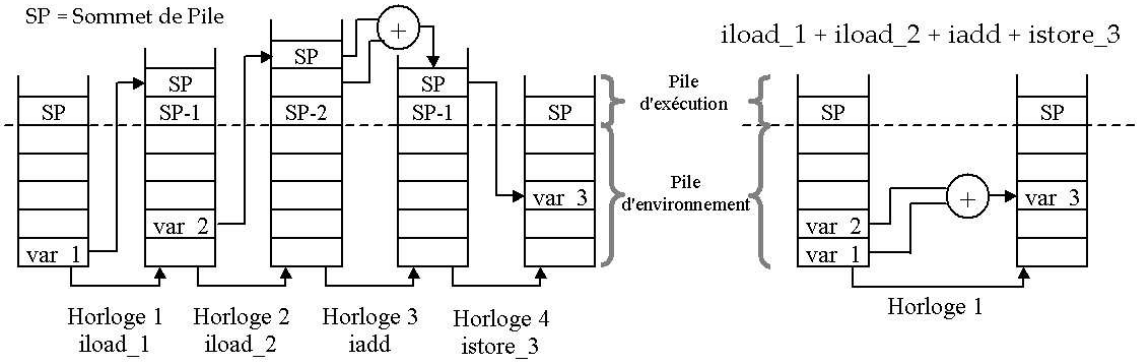

FIG. 8 – Exécution sans et avec regroupement d'instruction[GC98].

Certaines instructions du Bytecode ne peuvent pas être regroupées, notamment celles de changement de méthode [SA99]. L'architecture de la JMQ permet d'accroître les possibilités de regroupement des instructions de deux façons :

– La file permet de rendre les instructions de chargement et de calcul indépendantes les unes des autres.
– La séparation des mécanismes d'environnement et d'exécution permet de regrouper les instructions liées à l'environnement avec les autres types d'instructions.

Il devient donc possible d'accroître le nombre d'instructions décodées à chaque cycle. La Figure 9 montre un exemple de regroupement d'instructions effectué par les modèles JMS et JMQ. Le groupement effectué par la JMQ est plus agressif et réduit le nombre de cycles nécessaires au décodage et à l'exécution de la même suite d'instructions.

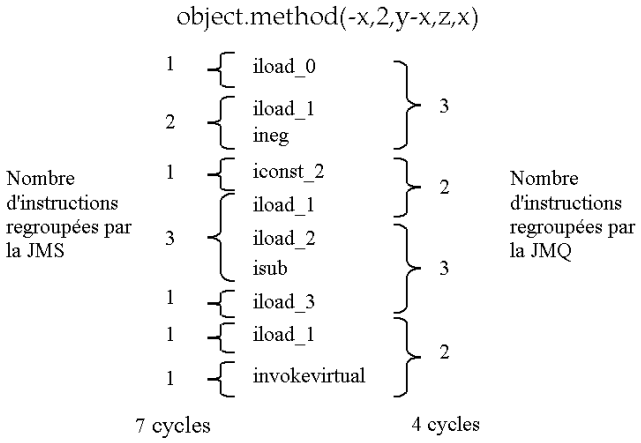

FIG. 9 – Comparaison des regroupements d'instructions JMS et JMQ.

3.4. Organisation matérielle de la JMQ

Le modèle JMQ est organisé autour de la pile d'environnement et de la file d'exécution. La Figure 10 montre cette organisation. Les flèches qui représentent le trajet des instructions permettent de retrouver

le pipeline instruction (Figure 4), tandis que celles qui représentent le trajet des données montrent le pipeline de données (Figure 5). La JMQ est composée des modules suivants :
– FETCH : Charge les instructions depuis la mémoire (étage FETCH).
– DEC : Décode et regroupe les instructions (étage DECODE).
– PREP : Contient la pile d'environnement matérielle et les files de dépendances. Il gère tous les accès à la pile d'environnement et effectue les instructions liées à l'environnement des méthodes Java (étages Switch-Register, Env et Writeback).
– CHARG : Charge les opérandes dans la file d'exécution (étage Load).
– File d'exécution.
– EXEC : Effectue les opérations sur les opérandes extraits de la file d'exécution (étages EXEC et CACHE).

Les deux files de dépendances sont incluses dans le module PREP. Elles chargent et rangent les variables locales depuis ou vers la pile d'environnement, vérifient les dépendances de données et constituent le mécanisme de by-pass entre les modules CHARG et EXEC. Elles permettent à ces modules de ne pas connaître l'environnement courant. La file de dépendance de lecture contient des données (variables locales) chargées depuis la pile d'environnement ou depuis le module EXEC. La file de dépendance d'écriture contient les adresses d'écriture des variables locales dans la pile d'environnement. Deux FIFOs acheminent les instructions depuis le module PREP vers les modules CHARG et EXEC. La présence de ces deux FIFOs permet de désynchroniser les flots d'instructions destinés aux modules LOAD et EXEC. Il devient donc possible d'anticiper le chargement des opérandes dans la file d'exécution.

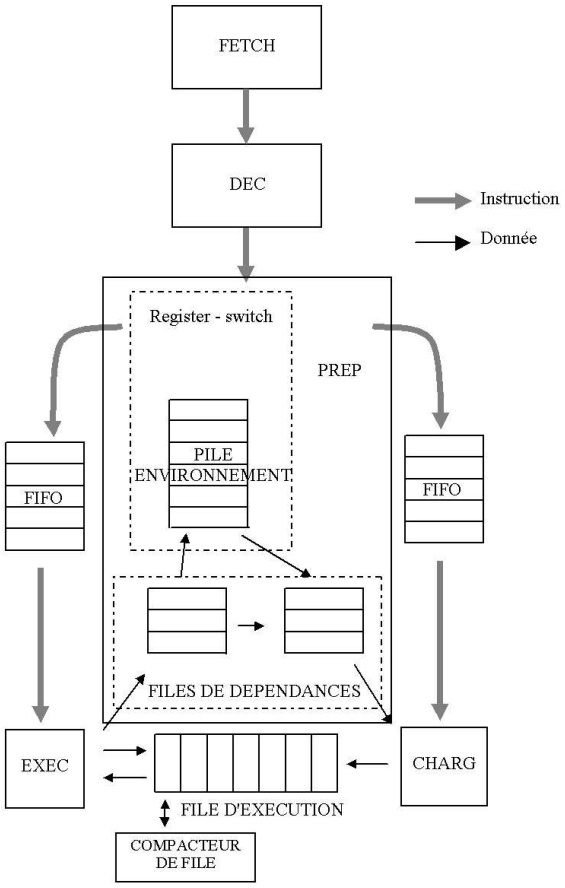

FIG. 10 – Diagramme de la JMQ.

Les modules Load et Exec sont synchronisés par la présence des données. Ils exécutent chacun leurs propres flots d'instructions en mode Data-flow. De plus, cette architecture permet au module PREP de changer d'environnement alors que les modules CHARG et EXEC continuent à travailler sur l'environnement courant. La présence de FIFOs entre le module PREP et la file d'exécution augmente la profondeur du pipeline et justifie donc l'ajout d'une unité de prédiction de branchement.

4. Exemple d'exécution

Le Bytecode est chargé, décodé et regroupé (Figure 11) par les modules FETCH et DECODE. Les groupes d'instructions obtenus sont traités par le module PREP comme le montre la Figure 12.

Pour chaque groupe, les instructions sont acheminées vers les modules appropriés.

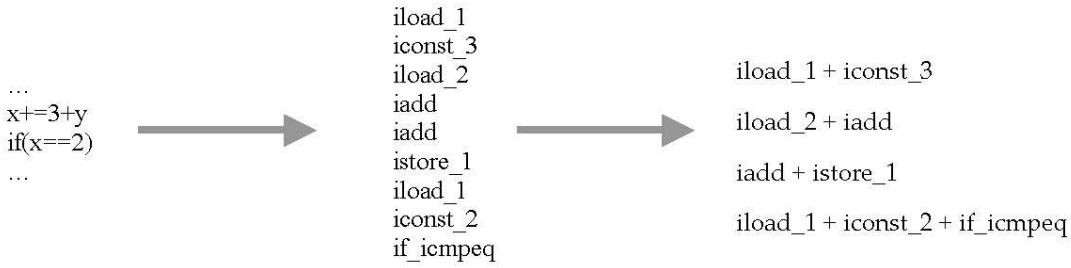

FIG. 11 – Programme Java et regroupement d'instructions associés.

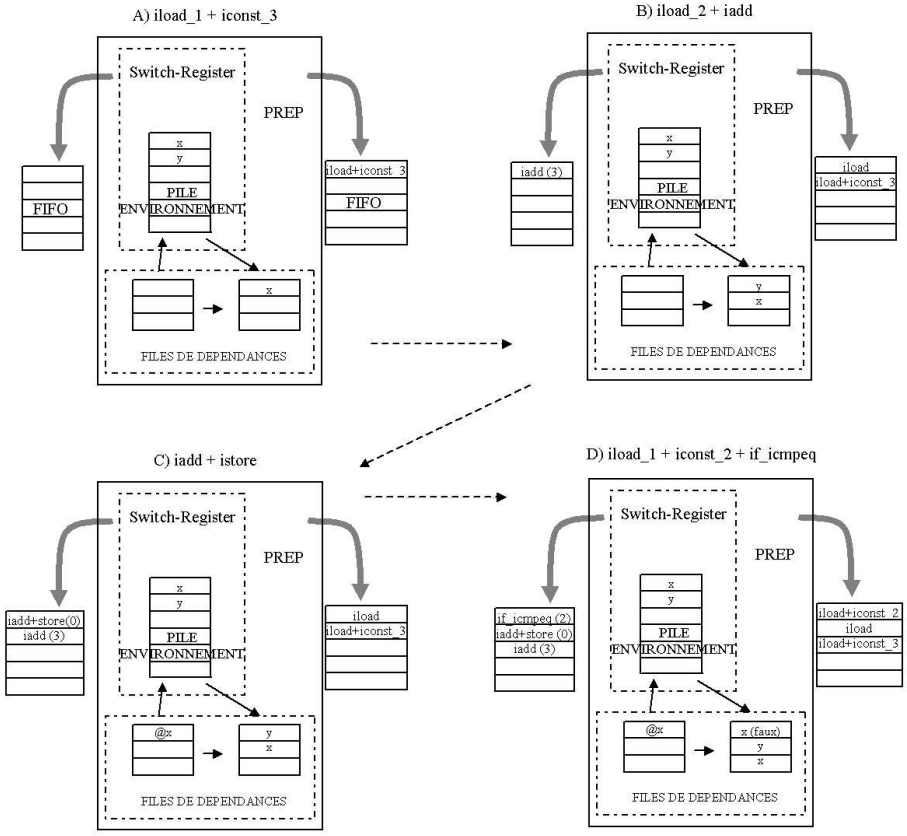

FIG. 12 – Exemple d'exécution par le module PREP.

- A) montre le chargement d'une variable locale dans la file de dépendance de lecture.
- B) montre comment PREP ajoute des informations aux instructions destinées au module EXEC (nombre d'opérandes entre deux opérations).
- C) montre l'ajout d'écritures "en attente" dans la file de dépendance d'écriture.
- D) montre une dépendance de données.

Les FIFOs entre le module PREP et les modules CHARG et EXEC, ainsi que les files de dépendances permettent le parcours des instructions et l'échange de données entre ceux-ci, elles rendent indépendantes l'exécution des flots d'instructions destinés à chaque module.

La Figure 13 illustre le chargement dans la file d'exécution de la variable x et de la constante 3 par le module CHARG dans les registres pointés par P0_a et P0_b (Figure 2, le module CHARG est capable d'insérer deux données consécutives dans la file d'exécution), ces pointeurs sont ensuite déplacés pour pointer sur de nouvelles cases libres. La variable x est lue dans la file de dépendance de lecture. Le pointeur de lecture de la file de lecture est déplacé. Le déplacement du pointeur P (sommet de la pile virtuelle) n'est pas effectué car seulement 2 nouvelles données sont insérées, alors qu'un déplacement de 3 est requis.

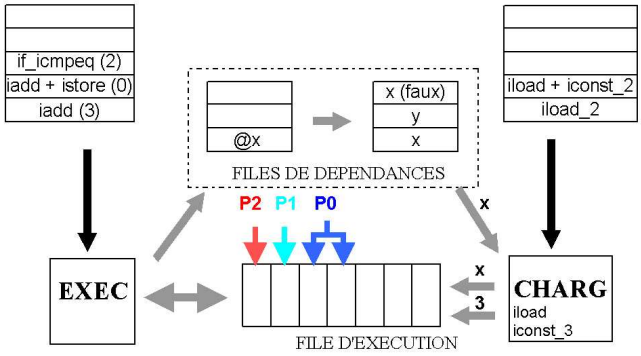

FIG. 13 – Exemple exécution sur file au temps T.

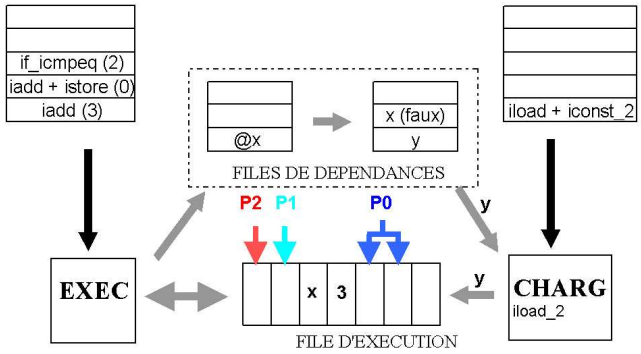

FIG. 14 – Exemple exécution sur file (T+1).

Au temps T+1, le module CHARG insère la variable y dans la file d'exécution, lue à partir de la file de dépendance de lecture (Figure 14). Les pointeurs P0 sont déplacés. Le pointeur P peut être déplacé car le nombre de nouvelles entrées dans la file est suffisant. Les nouvelles positions des pointeurs P1 et P2 sont ensuite calculées.

Au temps T+2 (Figure 15), Le module EXEC effectue l'addition sur les deux données extraites de la file d'exécution et envoie le résultat dans le registre pointé par P1. Le registre pointé par P2 est invalidé. Ce pointeur est ensuite déplacé vers la première entrée occupée en aval de la file d'exécution (nouveau sous-sommet de la pile virtuelle). La valeur de x contenue dans la file de dépendance de lecture étant toujours invalide, l'instruction iload ne peut pas être exécutée par le module CHARG et aucune donnée n'est chargée dans la file.

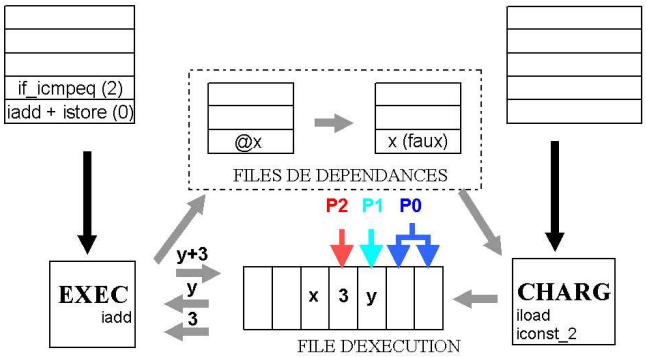

FIG. 15 – Exemple exécution sur file (T+2).

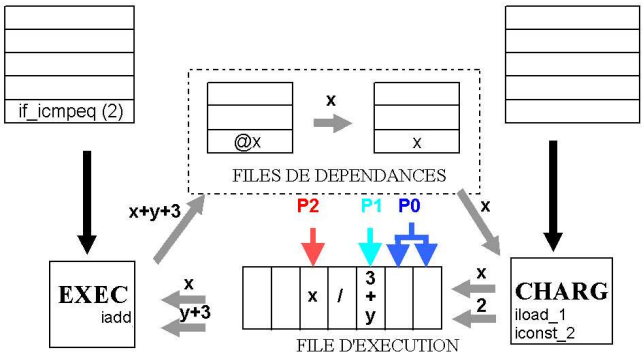

FIG. 16 – Exemple exécution sur file (T+3).

Au temps T+3 (Figure 16), Le module EXEC effectue l'addition sur les deux données extraites de la file d'exécution et envoie le résultat vers la file de dépendance d'écriture (module PREP). La donnée est écrite dans la pile d'environnement et mise à jour dans la file de dépendance de lecture. Elle est simultanément fournie par le mécanisme de by-pass au module CHARG qui l'insère dans la file d'exécution. La constante 2 est insérée dans la file d'exécution. Les pointeurs P0 sont déplacés. Le pointeur P est déplacé et les positions des pointeurs P1 et P2 en sont déduites.

Au temps T+4 (Figure 17), le module EXEC effectue la comparaison sur les données extraites de la file d'exécution. Le résultat de la comparaison est fourni au module PREDIC, si la résolution du branchement correspond à la prédiction effectuée, l'exécution continue. Si la prédiction est mauvaise, le pointeur P0_a prend la position P+1 (P0_b <= P+2), les files de dépendances et les FIFOs sont vidées, tous les étages du pipeline sont invalidés et l'exécution reprend à la bonne adresse de branchement.

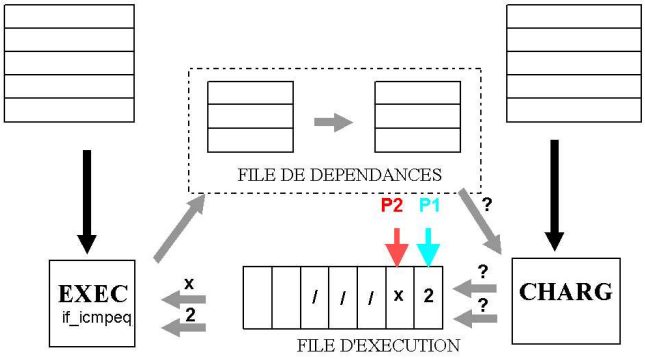

FIG. 17 – Exemple exécution sur file (T+4).

5. Comparaison

5.1. Complexité matérielle

Les processeurs JMS et JMQ utilisent plusieurs modules identiques. Les caches instructions (4Ko) et de données (émulé) ainsi que le module FETCH sont identiques dans les deux modèles. Le buffer d'instructions utilisé permet de contenir jusqu'a 24 octets d'instructions (16 pour le Picojava-II). Les modules DECODE des deux processeurs ne diffèrent que dans les types associés aux instructions et les regroupements effectués. Le modèle JMS a été amélioré par raport au Picojava_II qui n'était pas capable de regrouper les instructions sur les données 64 bits. La JMQ est plus complexe que la JMS car elle utilise en plus de nombreuses FIFOs et une unité de prédiction de branchement. Afin de pouvoir apprécier le gain apporté par l'utilisation d'une file d'exécution, un troisième modèle appelé JMS_PRED a été développé. La JMS_PRED est une JMS avec une unité de prédiction de branchement. Les modèles JMS_PRED et JMQ utilisent un prédicteur de branchement à compteur à saturation 2-bit [S81]. Ce prédicteur utilise un cache 2 voies de 128 entrées chacune correspondant à 128 compteurs 2-bit.

Les tailles des FIFOs utilisées sont les suivantes :

- FIFO CHARG : 16x48 bits
- FIFO EXEX : 32x34 bits
- File de dépendance de lecture : 16x69 bits
- File de dépendance d'écriture : 8x34 bits
- File d'exécution : 256x70 bits

La complexité des modèles systèmeC est difficile à évaluer compte tenu du niveau de description. Néanmoins, une bonne approximation peut être effectuée en utilisant la complexité du processeurs Picojava-II puis en utilisant les deux règles suivantes :
– Picojava-II : 442K portes dont 128k portes de logique et 314k portes de mémoire (16ko cache de données + 16ko cache instructions + ROM)
– un bit de registre = 5 portes
– un bit de RAM = 1,5 portes

	JMS (K portes)	JMS_PRED (K portes)	↗ (JMS/JMS_PRED)	JMQ (K portes)	↗ (JMS/JMQ)
logique	128	155	21,1%	191	49,2%
logique + mémoire	442	469	6,1%	505	14,2%

TAB. 1 – Comparaison de complexité JMS/JMS_PRED et JMS/JMQ

L'augmentation de la complexité est de l'ordre de 6,1% pour le modèle JMS_PRED et de 14,2% pour le modèle JMQ (Table 1).

5.2. Performance

Pour comparer les modèles JMS, JMS_PRED et JMQ on utilise le programme Raytracer qui fait partie des Benchmark JavaGrande.

	JMS (cycle)	JMS_PRED (cycle)	gain (JMS/JMS_PRED)	JMQ (cycle)	gain (JMS/JMQ)
Raytracer (init)	454 096	373 509	17,7%	327 263	27,9%
Raytracer (exec)	133 822 917	123 482 288	7,7%	112 002 151	16,3%
Raytracer (total)	134 284 180	123 862 553	7,8%	112 335 295	16,3%

TAB. 2 – Comparaison JMS/JMS_PRED et JMS/JMQ

Les premiers résultats montrent un accroissement de la vitesse d'exécution de l'ordre de 7,8% avec un prédicteur de branchement et de 16,3% avec l'architecture de la JMQ (à cycle constant)(Table 2). Ce gain semble cohérant avec l'augmentation de la complexité du modèle JMQ.

6. Travail à effectuer

Le prédicteur de branchement utilisé donne 83% de réussite aux branchements conditionnels. Ce mécanisme de prédiction peut être remplacé par un mécanisme plus efficace afin d'étudier son impact sur les performances des deux architectures, la JMQ devrait tirer avantage d'une prédiction plus précise. Les tailles des différentes FIFOs doivent être optimisées pour trouver un bon compromis entre le coût matériel et les performances de la JMQ. Une amélioration des performances reste également possible par une optimisation de certains détails de son fonctionnement (augmentation du parallèlisme d'exécution des instructions par l'utilisation de modules spécialisés et indépendants dans le module EXEC).

7. Conclusion

Les premiers résultats montrent que la JMQ apporte un accroissement de la vitesse d'exécution des programmes Java de l'ordre de 16,3% pour une augmentation de la complexité de 14,2%. Cela montre que l'exécution du Bytecode peut être améliorée par la séparation de l'exécution des instructions de calcul et de gestion de l'environnement mais également par l'utilisation d'une file d'exécution. Ce second mécanisme permet de fournir une indépendance entre les chargements de données et l'exécution

des opérations au sommet de la pile virtuelle. La conception d'un processeur étant un très gros travail, les modèles de simulation ont été décrits en SystemC et la plupart des mécanismes annexes ont été empruntés au PicoJava-II et à la JVM [LY99].

Bibliographie

ABS75 F. Anceau, G. Baille et J.P. Shoellkopf, "machine informatique électronique destinée à l'exécution parallèle d'un langage post-fixé", ANVAR brevet n°75 29 473, France, septembre 1975.

BS75 G. Baille et J.P. Schoellkopf, "Evaluation of polish-form expression on a FIFO queue : a new approach towards the realization of a high-level pipeline computer" Sagamore Computer Conference on Parallel Processing, août 1975.

CT97 M. O'Connor et M. Tremblay, "PicoJava-I : The Java Virtual Machine in Hardware", IEEE Micro, pp. 45-47, mars-avril 1997.

GC98 H. McGhan et M. O'Connor, "PicoJava : A Direct Execution Engine For Java Bytecode", IEEE Computer, pp. 22-30, octobre 1998.

HS96 C-H A. Hsieh, J. C. Gyllenhaal, W. W. Hwu, "Java Byte-code to Native Code (Translation : The Caffeine Prototype and Preliminary Results", in proceeding of the 29th Annual International Symposium on Microarchitecture (MICRO-29), pp 90-97, Los Alamos, CA, USA, 2-4 Décembre 1996, IEEE Computer Society Press.

JPS77 JP. Schoellkopf, "Machine PASC-HLL : Définition d'une architecture pipe-line pour une unite centrale adaptée au langage pascal", Thèse de 3ème cycle, INP6, Grenoble, 28 juin 1977.

LY99 T. Lindholm et F. Yellin, "The JavaTM Virtual Machine Specification", Second Edition, Addison-Wesley Pub Co, 1999.

MS05 M. Schöberl, "JOP : A Java Optimized Processor for Embedded Real-Time Systems", Technischen Universität Wien, Fakultät für Informatik, janvier 2005.

O05 E. Orgell, "Wireless Developers Lament Current State Of Important Development Tools New Evans Data Survey", Evans Data Corporation, Santa Cruz, CA, octobre 3, 2005.

PIM96 Sun Microsystems, "PicoJava™-I Microprocessor Core Architecture", transp. From Sun Microsystems, octobre 1996.

PJIIP99 Sun Microsystems, "PicoJava™-II Programmer's Reference Manual", transp. From Sun Microsystems, Part No. : 805-2800-06, mars 1999.

PJIIM99 Sun Microsystems, "PicoJava™-II Microarchitecture Guide", transp. From Sun Microsystems, Part No. : 960-1160-11, mars 1999.

PR97 T. A. Proebsting, G. T. Townsend, P. Bridges, J. H. Hartman, T. Newsham, S. A. Watterson, "Toba : Java For Applications : A Way Ahead of Time (WAT) Compiler", in Proceeding Third Conference on Object-Oriented Technologies and Systems (COOTS'97), 1997.

S81 J. E. Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8th International Symposium on Computer Architecture, Mineapolis, pp.135-148, May 1981.

SA99 K. SanKaralingam et K. Agaram, "Evaluating the Instruction Folding Mechanism of the PicoJava Processor", December 1999, http ://www.cs.utexas.edu/users/karu/docs/report.ps.

WW73 W. Wulf et al., "The Design of an Optiming Compiler", American Elsevier, New York, 1973.