Optimisation des performances et de la consommation de puissance électrique pour architecture Intel ltanium/EPIC

Jamel Tayeb

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre : 08/40

Optimisation des performances et de la consommation de puissance électrique pour architecture Intel Itanium/EPIC

THÈSE

Présentée et soutenue publiquement le : 25 novembre 2008

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Spécialité Automatique et Informatique des Systèmes Industriels et Humains

Discipline : Informatique

par

Jamel TAYEB

Rapporteurs :

Dr. Albert COHEN,	INRIA, École Polytechnique de Paris
Pr. William JALBY,	Université de Versailles-Saint-Quentin-en-Yvelines

Examinateurs :

Pr. Jean-Luc DEKEYSER,	Laboratoire d'Informatique Fondamentale à Lille
Pr. Pierre MANNEBACK	Faculté Polytechnique de Mons, Belgique

Invité :

M. Gwenolé BEAUCHESNE,	Ingénieur, Mandriva S.A. Paris

Directeur :

Pr. Smail NIAR,	LAMIH, Université de Valenciennes

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines — UMR 8530

Remerciements

Merci tout d'abord à *Smail Niar* qui il y a tous juste quatre ans m'a offert, en plus de son amitié, l'opportunité d'effectuer ma thèse au sein du laboratoire LAMIH. Je le remercie pour cette chance inespérée, de la confiance et de son soutien de tous les moments, les plus heureux comme les plus difficiles. Son aide et ses conseils m'ont aidé à garder le cap.

J'exprime ma profonde gratitude à *Albert Cohen* et *William Jalby* qui m'ont fait l'honneur d'être rapporteurs de cette thèse et pour avoir pris le temps de rédiger un rapport sur celle-ci. Je les remercie pour l'intérêt sincère qu'ils ont porté à mon travail. Je tiens à remercier ici les examinateurs : *Jean-Luc Dekeyser* et *Pierre Manneback* qui m'ont fait le plaisir d'examiner ce travail et de faire partie du Jury. Merci enfin à *Gwenolé Beauchesne* qui m'a fait le plaisir de sa présence et que je retrouve à cette occasion avec plaisir.

Je remercie *Lakshmi Talluru* et *Adarsh Sogal* qui chacun à leur façon ont consenti des efforts inattendus en me donnant des moyens afin que je puisse mener à bien mes recherches en parallèle de notre exigent travail de tous les jours.

Merci aussi à mes amis qui m'ont – plus que de coutume – supporté, écouté et motivé lorsqu'il m'arrivait de perdre courage. Merci à vous *Stanislas Odinot*, *Dan Zimmerman*, *Jean-Laurent Philippe* et *Christopher Meredith*. Merci à *Sandrine Charbonnier* pour tous cela et aussi pour son aide précieuse et indispensable lors de la rédaction de ce mémoire.

Je remercie de tous mon cœur mes parents pour leur amour inconditionnel et pour m'avoir toujours poussé à aller de l'avant. Merci à mon épouse, *Danika Hercha*, qui m'a toujours encouragé à poursuivre mes rêves. Son amour et son soutien m'ont été particulièrement importants au cours de cette difficile année.

Enfin, je remercie du fond du cœur mon père et mon grand-père maternel qui n'auront pas la chance de lire ces lignes, et pourtant sans lesquels je n'aurais jamais entrepris cette aventure. Merci à vous d'avoir su éveiller ma curiosité et mon amour pour la science et la découverte. Je dédie mon travail à votre mémoire.

Résumé

En fondant sa nouvelle génération de processeurs sur l'architecture *Very Long Instruction Word*, *Intel Corporation* a introduit une famille de processeurs à la fois novateurs et généralistes, en ce sens que nous les retrouvons aussi bien dans le segment des calculateurs scientifiques que dans celui des serveurs d'entreprise. Les choix architecturaux retenus et la mise à la disposition du programmeur, désormais seul maître à bord, d'importantes ressources matérielles, nous ont conduit à entreprendre l'étude d'utilisations alternatives de certaines d'entre-elles. Parmi les ressources que nous avons ainsi étudiées, nous trouvons les piles des registres que nous avons optimisés pour l'exécution de machines à piles. Notre choix c'est naturellement porté sur le système *FORTH*, archétype des machines à pile s'il en est. Après une implémentation logicielle d'un système *FORTH* pour *Itanium*, nous avons introduit un ensemble d'amendements matériels à l'architecture *Explicit Parallel Instruction Computer* afin de compenser certaines limitations de notre implémentation initiale. Nous avons ainsi mis en place une pile matérielle dont le contrôle échoit explicitement au logiciel.

Cette première approche, si elle est bien adaptée au système *FORTH* montre toutefois ses limites lorsqu'il s'agit d'implémenter des langages évolués ayant connu un succès commercial et d'estime plus important. Bien entendu, ces deux langages ont des différences techniques fondamentales, dont la plus importante dans le cadre de cette étude est la nature fortement typée de la pile d'évaluation de *.NET*, là où la pile *FORTH* n'a aucune forme de typage. Nous avons repensé en conséquence notre approche initiale et nous avons proposé une pile matérielle dont le contrôle est assuré cette fois-ci implicitement par le matériel. Il en résulte une plus grande facilité d'implémentation de langages tels que le *Microsoft Intermediate Language* de *.NET* ou *Java*, tout en limitant l'ampleur des modifications requises à son emploi. L'utilisation de cette pile permettant une traduction directe et à faible coût du *MSIL* en binaire. Le second avantage de la méthode est d'être « implémentable » quasi-directement dans les processeurs *Itanium*.

Dans une seconde partie, nous nous sommes attelés à l'étude et à l'optimisation de l'efficacité énergétique des applications serveurs destinées à cette nouvelle famille de processeurs. Nous avons développé dans un premier temps des outils de mesure et d'analyse de la consommation d'énergie. Nous avons ensuite dégagé un corpus de règles qui permettent aux développeurs d'applications de rendre leurs logiciels plus économes et d'adapter leur consommation énergétique en fonction d'un niveau de performance requis ou décidé dynamiquement.

Table des matières

Introduction

1. Introduction générale

L'industrie du semi-conducteur vit depuis plus de quarante ans au rythme de la loi de *Gordon Moore*. Enoncée en 1965 (Moore 1965) elle peut se résumer par l'assertion suivante : « *le nombre de transistors par circuit de même taille doublera tous les 18 à 24 mois* ». Cette loi s'applique au processeur précurseur – ou *lead microprocessor* – qui intègre pour la première fois une nouvelle micro architecture, par opposition aux améliorations induites par les nouvelles techniques de lithographie – ou *compaction microprocessor*. Une finesse de gravure accrue permet d'augmenter la densité des transistors gravés sur le silicium, mais, pour un microprocesseur donné, cela n'augmente pas pour autant leur nombre.

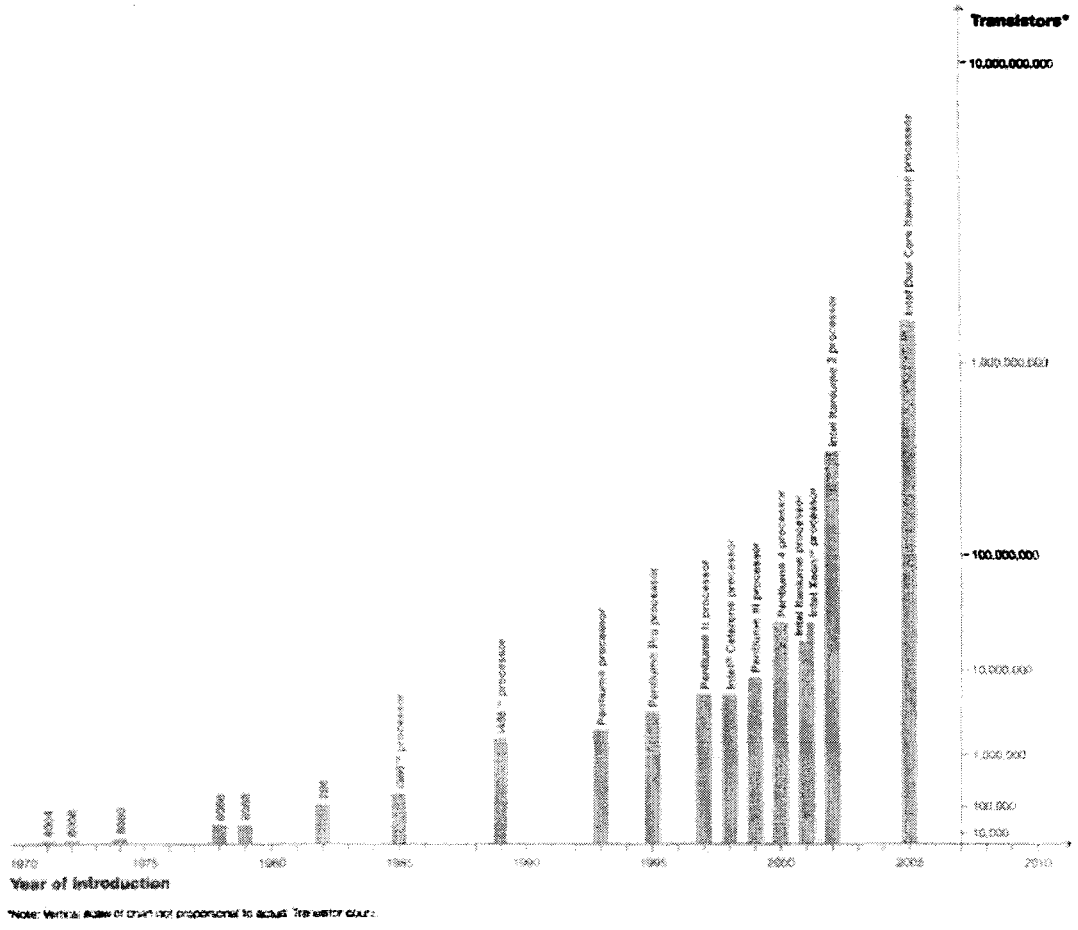

Figure 1 – Graphique montrant l'évolution du nombre des transistors pour chaque processeur précurseur introduit par Intel Corporation. Source : Intel Corporation.

Ainsi, la surface des processeurs précurseurs a été augmentée d'environ 14% tous les deux ans. En dix ans, la technologie de gravure est passée quant à elle de 1 µm à 0,18 µm (les industriels les plus en pointe produisent certaines de leurs puces en 45 nm et prévoient des circuits en 32 nm d'ici 2009 et 22 nm en 2011). La fréquence d'horloge a été multipliée pour sa part par cinquante (50x) dans le même temps. La performance des processeurs a subi un accroissement de ~75x, dont malheureusement seulement ~6x sont dus à des innovations micro architecturales. La question qui se pose est donc la suivante : « *quelle est la signification de la loi de Moore pour la performance ?* ». En effet, en dix ans, les avancées micro architecturales telles que les *pipelines*, le super scalaire, les cœurs à exécution désordonnée – ou *Out Of Order execution engine* (*OOO*) – ou encore l'exécution spéculative des instructions ont multiplié les performances d'un – petit – facteur six !

Pour étudier l'augmentation de la performance, nous devons recourir à la formulation de *Pollack. Fred J. Pollack – Intel Fellow* et Directeur du *Microprocessor Research Labs* – a observé que pour le même procédé de fabrication, le processeur précurseur offre en moyenne un gain de performance de ~1,4x pour un doublement de la surface par rapport au processeur de génération précédente (Ronen, et al. 1999). *Pollack* observe ainsi que le gain de performance décline avec les générations successives et évolue comme la racine carrée de l'accroissement de la surface de la *die* du processeur. En d'autres termes, bien que les physiciens permettent d'augmenter la densité de transistors et que les usines peuvent fabriquer ces mêmes transistors à grande échelle selon la prévision de *Moore*, il faut impérativement introduire des modifications radicales dans les microarchitectures pour assurer les gains de performance escomptés – ou du moins significatifs. Autrement dit encore, l'industrie du microprocesseur s'est trouvée à la croisée des chemins entre 2000 et 2003. Ceci a été illustré par les difficultés rencontrées par l'architecture *NetBurst* à offrir le niveau de performance attendu par les consommateurs. Il était dès lors clair qu'il n'était plus possible de s'en remettre à la seule recette du passé, à savoir augmenter la fréquence d'horloge. En outre, cette montée en fréquence pose clairement le problème de la consommation d'énergie. Les projections donnaient en effet des consommations de l'ordre de 10 000 Watts pour le processeur à l'horizon 2010-2015 si rien ne changeait (Borkar 1999). Cette projection a agi comme un coup de semonce et a forcé l'ensemble des acteurs de l'industrie informatique à reconsidérer sérieusement les solutions privilégiées jusqu'à lors et à amender leurs plans de recherche et de développement.

Malheureusement, la montée en fréquence a des arguments de poids. Premièrement, augmenter la fréquence se traduit par un gain de performance quasi-linéaire pour les processus légers – ou *threads* – des applications. En d'autres termes, ce qui rend ce modèle très apprécié par les fondeurs et les éditeurs de logiciels, c'est qu'il ne demande aucun effort particulier de la part de ces derniers, et qu'il est simple à expliquer aux clients des premiers. A présent que le modèle est brisé – du moins, si les coûts de développement et de production en volume sont pris en considération – il ne reste plus qu'une seule variable à explorer pour assurer les gains de performances promis et attendus. Et la solution retenue par l'industrie pour résoudre ce problème se résume en un seul mot : ***parallélisme***. Parallélisme à gros grain tout d'abord *via* la virtualisation, parallélisme de données ensuite avec le calcul vectoriel ou *SIMD* (*Single Instruction Multiple Data*), mais aussi parallélisme au sens premier du terme avec des technologies telles que le multi-flots (*SMT Simultaneous Multi Threading*), le multiprocesseurs sur puce ou multi-cores (*CMP Chip Multi Processor*), ou plus fondamentalement le *MIMD*

(*Multiple Instruction Multiple Data*) d'*EPIC* (*Explicit Parallel Instruction Computer*). Nous présenterons dans la suite de cette introduction comment chacune de ces technologies tente de répondre au problème de la performance – et à celui de la densité de puissance qui en est le corolaire. Nous souhaitons ainsi mettre notre travail en perspective, en le situant dans cette période propice au développement d'idées nouvelles que nous avons la chance de vivre.

2. La réponse *EPIC*

Développée conjointement par *Hewlett-Packard* et *Intel Corporation*, l'architecture *EPIC* se retrouve implémentée au cœur des processeurs de la famille *Itanium*. Elle offre plusieurs aménagements matériels inédits, qui utilisés conjointement avec des logiciels correctement conçus, permettent de passer outre certaines barrières de performances difficilement franchissables par les architectures traditionnelles (*Reduced Instruction Set Computer* et *Complex Instruction Set Computer* essentiellement). Comme nous le verrons tout au long de ce document, la composante logicielle de l'architecture *EPIC* est essentielle. Elle l'est à un tel point, que le compilateur fait intégralement partie de l'architecture. Ce n'est plus un simple outil destiné aux programmeurs d'applications, mais fait partie d'*EPIC*. Avec son support, *EPIC* autorise principalement l'augmentation du degré de parallélisme des instructions ou d'*ILP* (*Instruction Level Parallelism*). Rappelons que jusque-là, l'expression d'un *ILP* élevé était échue aux cœurs à exécution désordonnée – *Out of Order*. Ainsi, le duo formé par le compilateur et un processeur *EPIC* peut espérer absorber les latences d'accès aux données et d'exécution des instructions.

Pour utiliser pleinement les capacités de l'architecture *EPIC*, les logiciels doivent littéralement coopérer avec le processeur. Si cette interaction est souhaitable pour n'importe quelle architecture pour obtenir le meilleur niveau de performance possible, cela est encore plus vrai pour l'architecture *EPIC*. En effet, prendre le code source d'un logiciel – le porter pour gérer un adressage sur 64-bit si nécessaire – et le recompiler n'offre pas toujours le meilleur résultat. Cela est d'autant plus vrai que le compilateur utilisé n'intègre qu'un nombre limité de technologies d'optimisation ou est en retard sur l'état de l'art. En effet, il faut constamment garder à l'esprit que le compilateur est un composant fondamental de l'architecture *EPIC* et qu'il a pour vocation de permettre l'évolution de l'architecture tout en retardant le besoin de modifier sa composante matérielle.

Mais comme tout outil, aussi évolué soit-il, le compilateur n'est hélas pas en mesure de réaliser des miracles sur tous les codes, et particulièrement sur les plus mal conçus. En effet, puisque le processeur ne prend plus en charge la recherche de l'*ILP* et exécute les instructions dans l'ordre exact dans lequel le compilateur les a programmées (à l'exception des lectures en mémoire générées lors de la résolution des défauts de cache de second niveau qui sont, elles, réordonnées), les erreurs de conception algorithmiques et d'implémentation s'expriment de façon patente. Les techniques d'optimisation sont nombreuses et ne seront pas couvertes dans ce document (Niar et Tayeb 2005).

Comme nous le verrons plus loin, l'architecture *EPIC*, avec les dernières moutures du processeur *Itanium*, profite aussi des technologies *SMT* et *CMP*. Il nous semple intéressant de

présenter une technologie intermédiaire puisqu'elle permet aux applications de bénéficier de certains avantages du parallélisme sans pâtir de la difficulté de mise en œuvre. Des recherches menées par *Intel Corporation* utilisent le phénomène de pré-chargement – *prefetch* – induit par la technologie *SMT* (Wang, et al. 2004). L'idée repose sur la création d'un *thread* délinquant (*delinquent thread*) pour épauler un *thread* de calcul (*worker thread*) en induisant le pré-chargement des données dans les mémoires caches. Cela s'applique aux constructions logicielles qui mettent quasi-systématiquement en défaut les algorithmes des unités de pré-chargement matérielles. Il s'agit par exemple des indirections multiples, qui peuvent être isolées dans un *thread* délinquant par le compilateur, et dont l'exécution est programmée en amont de celui du *thread* de calcul. Les accès en mémoire ainsi isolés seront fautifs – ce qui est recherché ici – et initieront le pré-chargement des données. Pendant l'exécution du programme, le *thread* de calcul et son *thread* délinquant sont liés aux processeurs logiques d'un même processeur physique. Avec le processeur *Itanium* – la plateforme de cette étude –, la méthode est rendue possible grâce à la présence des *PAL – Processor Abstraction layer* – et *SAL – System Abstraction layer*. Le *PAL* et le *SAL* sont normalement associés aux mécanismes de détection, de confinement, de correction et de publication d'erreurs matérielles. Ils permettent également la configuration des processeurs – sans intervention du système d'exploitation (*SE*). Ainsi, un mécanisme ultra-rapide de changement de contexte a été implémenté dans le *PAL/SAL* du processeur *Itanium 2*. Là où le *SE* requiert environ 500 cycles d'horloge (variable en fonction de la latence de la mémoire centrale) pour effectuer un changement de contexte, l'implémentation expérimentale procède à l'entrée et à la sortie du nouveau contexte en 32 cycles d'horloge. Le prototype améliore ainsi les performances des applications testées (SGBDR essentiellement) de l'ordre de 8~20%. Cela est possible, rappelons-le, sans exiger la modification du *SE*. Le *PAL* et le *SAL* s'avèrent être des auxiliaires précieux pour l'ajout de nouvelles fonctionnalités, comme pour la fonction de *lock-step* des cœurs du processeur *Montecito* qui permet l'exécution simultanée du même flot d'instructions par deux cœurs (Figure 3).

L'ensemble des travaux que nous avons effectué dans le cadre de cette étude a été conduit sur des processeurs *Itanium 2* de première génération (9 Mo de mémoire cache de troisième niveau, cadencés à 1,8 GHz). Pour autant, et pour conclure ce rapide survol de l'architecture *EPIC*, nous allons donner un aperçu de la feuille de route de la famille *Itanium*. Ainsi, la prochaine mouture de l'*Itanium* (nom de code *Tukwila* – Figure 2) devrait normalement être disponible lorsque vous lirez ces lignes. En plus des fonctionnalités introduites avec *Montecito*, (la technologie *Pellston* pour invalider dynamiquement des lignes de cache défectueuses ; la technologie *Foxton* offrant des fonctions d'économie d'énergie avancées; le *Lock-stepping* des cœurs déjà citée), *Tukwila* bénéficiera de la technologie d'interconnexion *QPI* (*Quick-Path Interconnect*). Celle-ci permet le remplacement des bus frontaux partagés et d'offrir, en conjugaison de contrôleurs mémoire intégrés, la bande passante mémoire et inter-processeurs qui font aujourd'hui défaut à ce processeur. Nous avons mis en annexe une présentation détaillée de l'architecture *EPIC*. Le lecteur pourra s'y référer pour y trouver les explications que nous avons éliminées du corps du texte pour en faciliter la lecture.

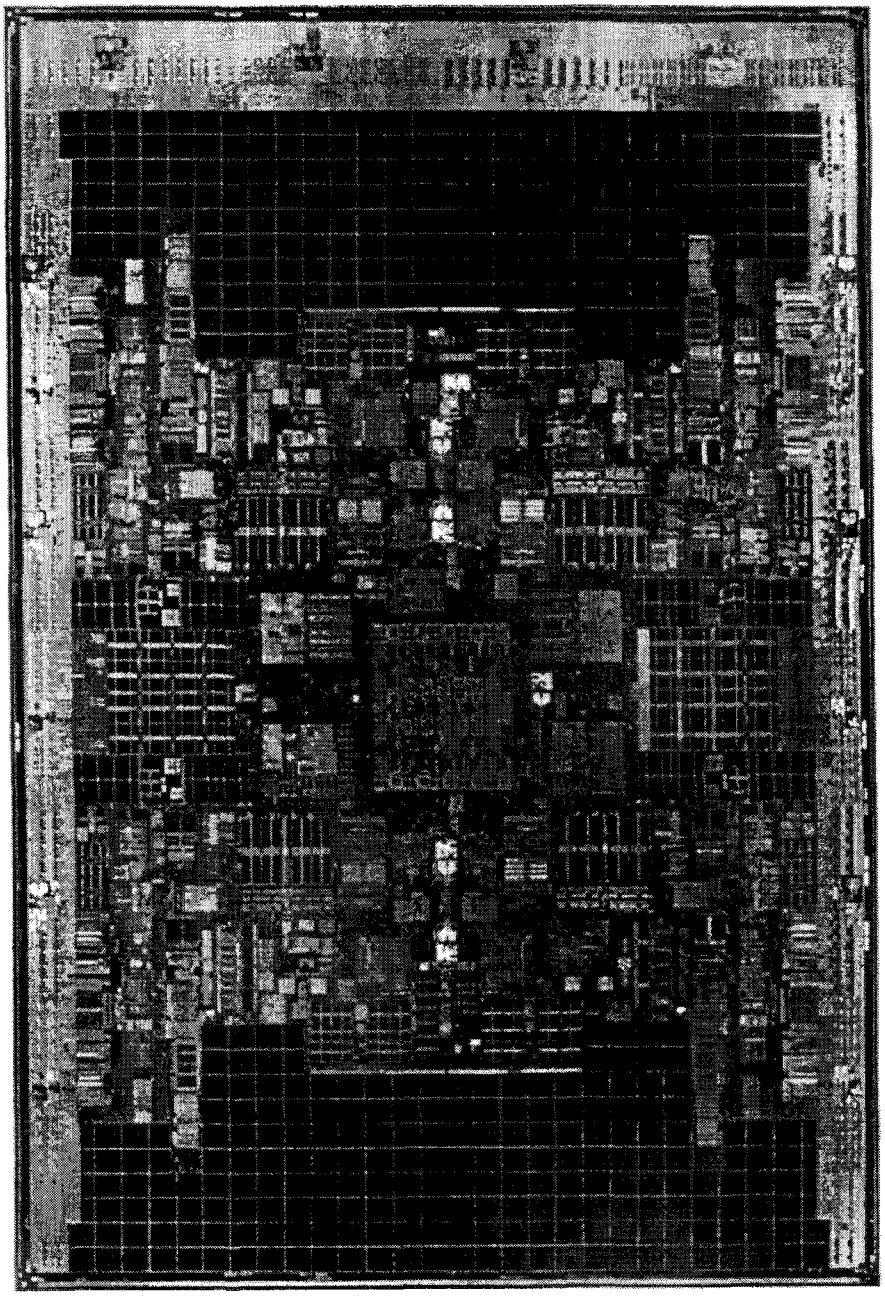

Figure 2 – Sur cette image de la die du processeur Tukwila, nous voyons clairement les quatre cœurs symétriques, ainsi que l'arbitre d'accès au bus en position centrale. Nous voyons nettement la prédominance des mémoires cache (30 Mo au total – 2 milliards de transistors). Source : Intel Corporation.

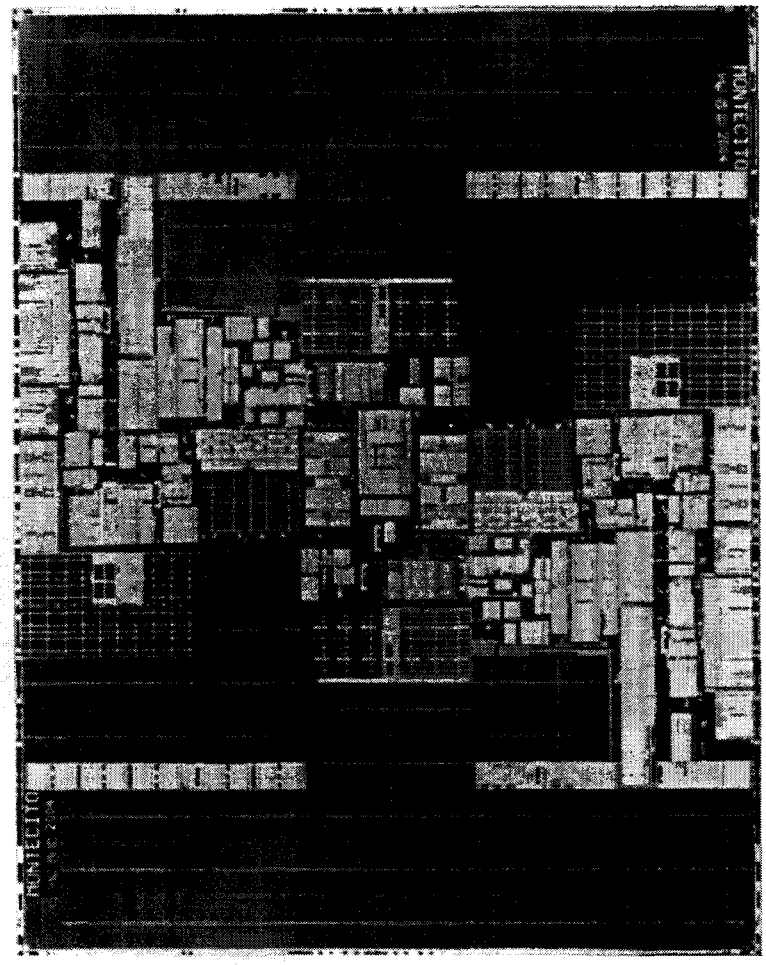

Figure 3 – En comparaison, le processeur Itanium 2 avec 9 Mo de mémoire cache de second niveau semble...petit. Il n'intègre pourtant pas moins de 410 millions de transistors, là où le Montecito embarque 1 720 000 000 transistors (gravé en 90 nm). Source : Intel Corporation.

3. La réponse *CMP*

Là où l'architecture *EPIC* n'a pas rencontré le succès commercial escompté en dehors de son marché de niche (remplacement des serveurs *RISC*), la technologie du *Chip Multi Processor* (*CMP* – aussi désigné par les noms *Multi-Core* et *Many-Core*) remporte un franc succès. En effet, *CMP* nous semble être **LA** réponse retenue par l'industrie du semi-conducteur pour résoudre le problème décrit par *Pollack*. Il s'agit d'utiliser le budget de transistors d'un projet de développement de microprocesseur, non plus pour la conception d'une puce monolithique, mais plutôt de le fragmenter pour graver plusieurs cœurs – plus ou moins simplifiés et spécialisés –

qui sont interconnectés par un *un-core*. Le tout étant fait de façon modulaire pour apporter un maximum de flexibilité et répondre ainsi rapidement et de façon économique aux besoins de performance et de fonctionnalité de chaque segment visé.

Ainsi, la prochaine génération de processeurs fondus par *Intel Corporation* – nom de code *Nehalem* – intègrera quatre cœurs et un *un-core* essentiellement composé d'interfaces d'interconnexion *QPI* et des contrôleurs de mémoire dédiés (Figure 7). En réutilisant certains de ces mêmes composants, les processeurs *Dunnington* seront composés de six cœurs tout en conservant la précédente technologie d'interconnexion de bus partagés (Figure 6). Comme nous l'avons évoqué plus tôt, la famille des processeurs *Itanium* bénéficie également de la technologie CMP, notamment depuis l'introduction du *Montecito* (Figure 2 et Figure 3). *Tukwila* offrira quatre cœurs physiques et huit cœurs logiques avec l'usage du *SMT*.

La technologie *CMP* va se développer considérablement dans les années à venir. Ainsi, le véhicule de développement *Tera-Scale* d'*Intel Corporation* (FIGURE 4) intègre quatre-vingt processeurs élémentaires (*PE*) fonctionnels dans la même puce (10 x 8). *Tera-Scale* n'a pas pour vocation de devenir un produit commercial, mais permet d'étudier et d'explorer les problèmes posés par l'interconnexion des cœurs. En effet, la topologie retenue pour assurer les communications entre les cœurs a une influence importante sur les performances. La topologie retenue utilise un réseau de routeurs. Chaque routeur a cinq ports : quatre ports pour assurer la connexion avec les routeurs voisons (N, S, E, O) et un cinquième port pour envoyer ou recevoir les données vers / ou du *PE* (Vangal, et al. 2007).

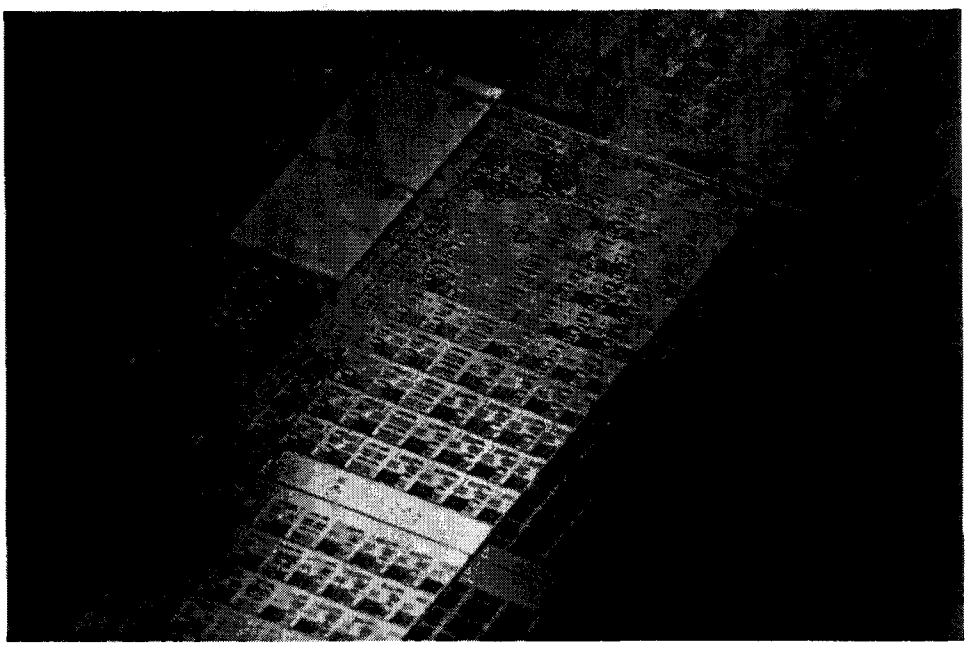

Figure 4 – Au centre de cette image nous voyons les quatre-vingt cœurs du véhicule de développement Tera-Scale. Source : Intel Corporation.

Le futur processeur graphique d'*Intel Corporation* (nom de code *Larrabee*) utilisera pour sa part un anneau d'interconnexion pour assurer l'échange des données entre ses cœurs (Abrash 2008). Ceux-ci, dont le nombre n'est pas officiellement arrêté – et sera très probablement variable –, sont des processeurs de la classe *Pentium* (*P6*) augmentés d'unités vectorielles spécialisées. Ils disposent également de la technologie *SMT* et sont privés de la logique d'exécution désordonnée (Figure 5). Ce dernier point est remarquable, car non seulement cela permet d'adapter l'architecture du processeur aux spécificités des traitements requis par le *pipeline* graphique, mais en plus, cela permet de réduire considérablement la quantité d'énergie consommée par le processeur de l'ordre de 25% – à l'instar du processeur *Atom* (Gerosa, et al. 2008). En comparaison des circuits graphiques spécialisés classiques, *Larrabee* innove en reprenant l'*ISA x86* et permet l'exécution des *threads* des programmes sur n'importe lequel de ses cœurs.

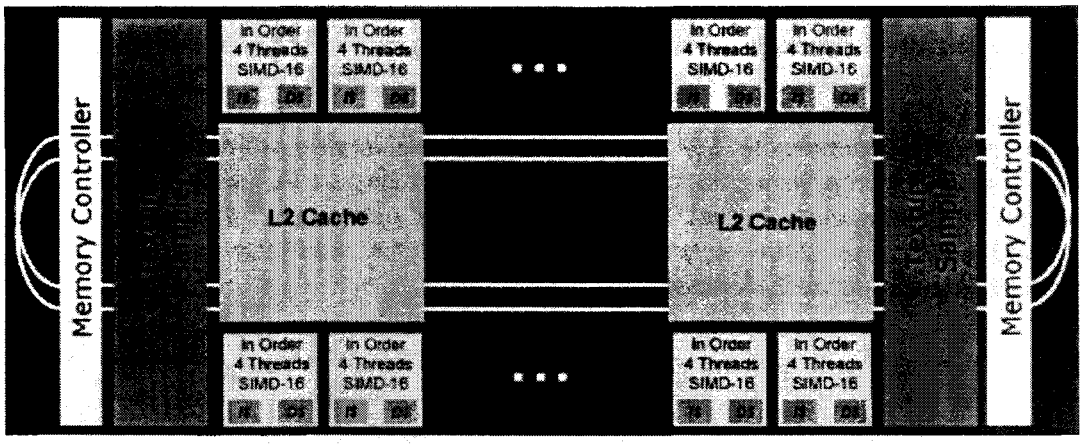

Figure 5 – Topologie en anneau pour l'interconnexion des cœurs du processeur Larrabee. Source : Intel Corporation.

Avec un nombre de cœurs croissants et des topologies d'interconnexions variables, les programmeurs vont faire face à de nouveaux problèmes. En plus des difficultés de parallélisassions évidents – que nous couvrirons plus loin – ils auront à faire face au problème de la qualité de service (*QoS*). En effet, à l'instar des réseaux, il sera nécessaire d'instaurer des priorités au sein du trafic de données inter-cœurs. Par exemple, pour assurer la fluidité de l'affichage de séquences vidéo, *Larrabee* lui réserve une fraction de la bande passante de l'anneau d'interconnexion. Si cette solution est statique, le problème se pose de façon plus aigüe avec les architectures plus généralistes. Ainsi, *Ravi Iyer* et *Al.* (Iyer, et al. 2007) explorent la mise en place au niveau de la logique des mémoires caches et du réseau d'interconnexion d'un mécanisme de *QoS* fondé sur un mécanisme complexe de définition de la priorité, des ressources requises et de la performance ciblée pour chaque tâche.

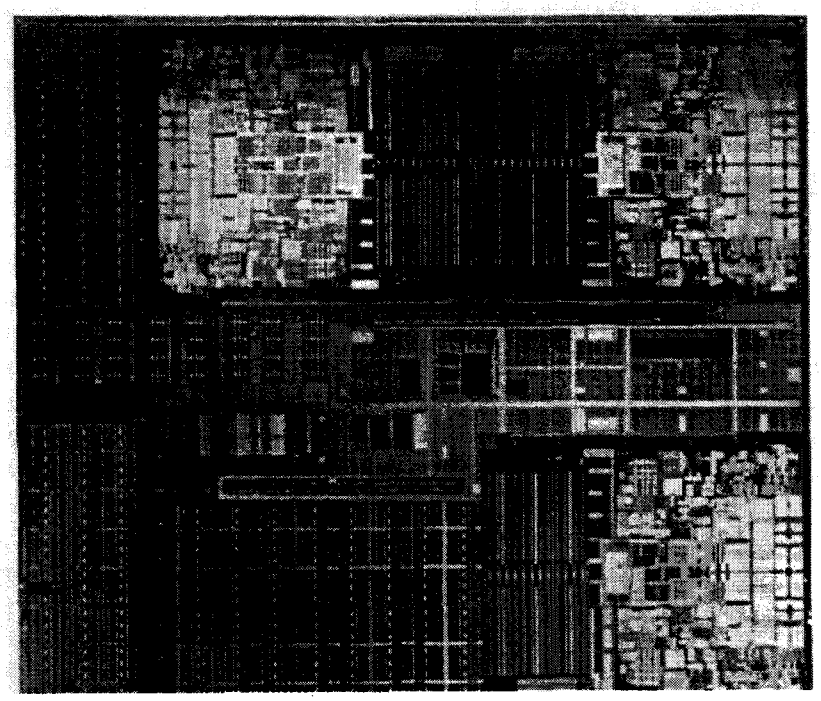

Figure 6 – Sur cette image de la die du processeur Dunnington, nous voyons les trois groupes de deux cœurs. Source : Intel Corporation.

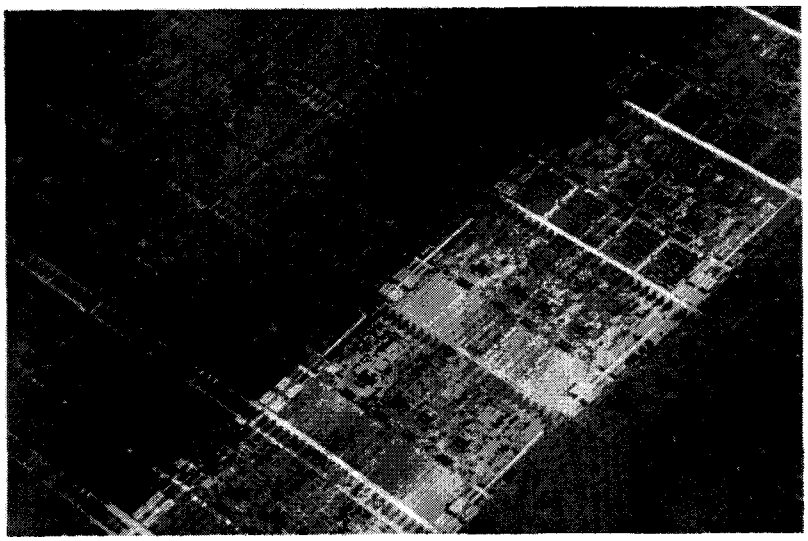

Figure 7 – Sur cette image de wafer de processeurs Nehalem, nous voyons les quatre cœurs de chaque die. Source : Intel Corporation.

4. La réponse *Virtualisation*

La *virtualisation* des processeurs et du matériel plus généralement a été introduite dès les années soixante par *IBM* pour ses *mainframes* de la série 360. L'objectif était de multiplexer l'utilisation du matériel qui était très coûteux. Cependant, la virtualisation est un terme générique qui désigne des concepts très variés. Si nous devions dégager un dénominateur commun à toutes ces technologies, ce serait l'*abstraction* (Tayeb 2008) Dès lors, il est aisé de comprendre pourquoi le mot *virtualisation* peut légitimement s'utiliser dans une vaste palette de concepts. Prenons quelques exemples de virtualisation couramment utilisés. Le premier niveau d'abstraction – et probablement le plus important – est celui du système d'exploitation (*SE*). En effet, l'une des définitions possibles pour le *SE* est: « Une couche logicielle qui permet l'abstraction du matériel et offre une interface standard aux logiciels applicatifs ». En effet, le programmeur n'a au final que peu d'intérêts pour les registres matériels à utiliser pour piloter un modèle spécifique de disque dur par exemple. Ce qui leur importe en revanche, c'est de pouvoir effectuer leurs entrées-sorties avec leur langage de programmation de prédilection et que le matériel se comporte comme cela est spécifié par les standards. Cela s'applique aussi aux librairies qui peuvent accéder aux routines de bas niveau proposées par le *SE*. Ce dernier dispose d'ailleurs de sa propre couche d'abstraction matérielle (*HAL – Hardware Abstraction Layer*).

Les développeurs sont d'ailleurs rompus au concept d'abstraction. Dès le début des années soixante-dix, le langage de programmation *Pascal* a démocratisé le concept de *P-Code*, un pseudo-code généré par le compilateur puis exécuté par un *runtime*. Celui-ci n'est autre qu'une machine virtuelle dont le *P-Code* est le langage machine – qui utilise une pile d'évaluation – et qui est traduit en binaire natif à la volée sur la plate-forme d'exécution. L'avantage de cette technique est d'isoler le frontal ainsi que le générateur de code du compilateur du matériel, rendant le portage de *Pascal* d'une plate-forme à une autre trivial. Seul le *runtime* devant être réécrit. C'est cette même technologie qui est à l'œuvre dans les environnements de développement *Java* de *Sun Microsystems* et *.NET* de *Microsoft Corporation*. Ainsi, un langage de programmation et un environnement de développement standardisé sont utilisés pour concevoir des applications qui sont interprétées ou compilées dans un langage intermédiaire. Ce code intermédiaire est alors exécuté par une machine virtuelle (la machine virtuelle de *Java – JVM –*, le *Common Language Runtime – CLR –* ou encore *Parrot*). *.NET* pousse l'abstraction un degré plus loin en autorisant les compilateurs de différents langages de programmation à générer du code pour la *CLR* en ciblant le langage intermédiaire de *Microsoft Corporation* (*MSIL*). L'écriture d'un compilateur devient par ce biais une tâche elle aussi abstraite.

Supposons à présent que nous disposions déjà d'un binaire natif pour une architecture donnée – l'*ISA* et le *SE*. Supposons aussi que nous voulions l'exécuter avec un couple *ISA/SE* différent de celui pour lequel il a été généré. Pour ce faire, nous devons normalement recompiler le code source de l'application – ce qui peut exiger des efforts considérables en termes de portage et d'optimisation des performances. Si cela s'avère être impossible, alors l'autre solution qui s'offre à nous consiste à utiliser un traducteur binaire. Un tel traducteur décodera le binaire et en traduira à la volée les instructions vers le jeu d'instructions du processeur cible. Il existe plusieurs traducteurs binaires, dont les plus connus sont l'*IA32EL*, l'*Aries Binary Translator*,

QEMU, *QuickTransit* ou *Wabi*. Signalons que l'*IA32EL*, conçu à l'origine par *Intel Corporation* et mis à la destination des vendeurs de *SE* pour exécuter le code *IA-32* sur les processeurs *Itanium* – ce qui a également permis de supprimer le décodeur d'instructions *IA-32* du processeur et de le simplifier par la même occasion.

En poussant le concept un peu plus loin il est possible de réduire encore la taille du problème. En effet, il est possible d'ajouter au couple abstrait *ISA/SE* le *BIOS* (*Basic Input-Output System*), la mémoire vive, les *ROMs* (*Read-Only Memory*), *etc.* Le traducteur binaire devient alors émulateur. Par exemple, l'*Ersatz-11* (DBIT 2008) ou le *REVIVER-11S* (Commware Technical Services 2008) sont des émulateurs de systèmes PDP-11 complet sous *Windows* et *Linux*. Les émulateurs sont disponibles pour pratiquement tous les systèmes – particulièrement pour les plus anciens et dont la production a cessé. Grâce à l'évolution des performances du matériel, celui des émulateurs dépasse largement celui des systèmes originaux. Bien entendu, il existe de nombreuses solutions d'abstractions qui ont en guise de point commun celui de vouloir offrir aux programmeurs d'atteindre l'objectif énoncé par *Sun Microsystems* lors du lancement de son système *Java* d'« *Écrivez une fois, exécutez n'importe où* ». Lorsque nous parlerons de virtualisation dans la suite de notre document, c'est à cette acceptation du terme que nous nous référerons.

Nous ne détaillerons pas ici les technologies de support matériel pour la *virtualisation*. Toutefois, il nous semble important d'en présenter rapidement le concept puisque ces technologies deviennent *de facto* synonymes de virtualisation pour le grand-public (Neiger, et al. 2006). En effet, ces technologies (chez *Intel Corporation VT-x* et *VT-i* pour la virtualisation du processeur et *VT-d* pour la virtualisation des entrées-sorties) introduisent de nouveaux modes de fonctionnement du processeur – pour résoudre le problème de l'aliasing de *ring* –, de nouvelles structures internes dont la plus importante est la *VMCS* (*Virtual Machine Control Structure*), ainsi que de nouvelles instructions pour faciliter la mise au point de machines virtuelles.

Il est important de considérer la virtualisation comme une piste d'avenir pour au moins deux raisons. La première de ces raisons est purement mécanique. La majorité des développeurs et ingénieurs en informatique sont aujourd'hui formés en utilisant des langages de la classe de *Java*. Ainsi, il est quasiment impossible de trouver de jeunes développeurs à même de concevoir des programmes en langage *C*, sans même parler de l'*assembleur* ou du langage *FORTRAN*. Il s'agit là d'un alignement naturel sur les attentes du marché, puisque les entreprises d'édition de logiciels ont rapidement vu les nombreux avantages qu'ils pouvaient en tirer à juste titre. Autre indice de cette omniprésence, le nombre croissant de projets de validation de machines virtuelles *Java* dans des segments jusque-là dominés par le langage *C*. Parmi ceux-ci, nous pouvons citer les systèmes embarqués haute performance, les télécommunications et les services financiers. En plus d'exiger des machines virtuelles et des compilateurs dynamiques de plus en plus performants – responsabilité qui échoit aux éditeurs de machines virtuelles et aux fondeurs de processeurs –, ces segments ont un autre point en commun : celui d'avoir des contraintes fortes sur les temps de latences. En plus d'être courts, ces temps de latences doivent être déterministes, une prouesse dont les machines virtuelles de première génération sont bien incapables. Nous voyons ainsi apparaitre par exemple des ramasse-miettes (*garbage collector*) déterministe chez les principaux vendeurs de machines virtuelles (*BEA Systems*, *IBM*, *etc.*) ainsi que le regains

d'intérêt et de projets pour l'optimisation des architectures de processeurs pour l'exécution efficace des machines virtuelles.

La seconde raison qui justifie, à notre avis, l'intérêt porté à la virtualisation est que les éditeurs de logiciels peinent à tirer profit du degré de parallélisme accru qui est offert par les plates-formes modernes. Et ce, d'autant plus que le parallélisme est devenu le choix premier pour les fondeurs afin d'augmenter les performances des processeurs tout en restant fidèles à la loi de *Moore*. Ceci est actuellement le problème majeur que l'industrie du logiciel doit impérativement résoudre. En effet, si jusqu'à présent un programmeur pouvait se contenter de ne littéralement *rien faire* pour optimiser la performance de son logiciel, avec des fréquences d'horloge stables, voire en régression, ils leur est impossible de répondre aux attentes des utilisateurs avec les recettes du passé. En effet, les clients ont étés habitués à constater l'augmentation automatique des performances avec celui la fréquence d'horloge. Les logiciels doivent donc être parallélisés. Si cela est souvent le cas pour des secteurs très particuliers comme ceux du calcul scientifique ou celui des serveurs de bases de données, ils n'en restent pas moins l'exception qui confirme la règle. Les programmeurs doivent êtres formés à cet effet, mais plus encore, de nouvelles technologies doivent êtres développées – comme les mémoires transactionnelles – et produites en masse pour permettre la transition entre le modèle de la performance d'un *thread* unique à celui assuré par une multitude de *threads*. En attendant que cela se produise, la virtualisation (au sens émulateur) reste la seule planche de salut vraiment disponible. Ainsi, en multipliant les machines virtuelles sur les plates-formes multi-cœurs et multiprocesseurs, il est possible de pallier au manque de parallélisme de la plupart des applications sans nécessité de modifications aux codes existants. En outre, en ayant recours à la consolidation de serveurs – l'un des modèles d'usage de la virtualisation qui connait le plus grand succès à ce jour – il est aussi possible d'améliorer considérablement l'efficacité énergétique des centres de calculs.

5. Structure du document

La suite du document est composée de quatre chapitres et d'une annexe. Le premier chapitre est consacré à la présentation de notre cache de pile pour l'architecture *EPIC*. Nous y introduisons notre première implémentation qui permet de conserver à même les registres des processeurs *Itanium* tout ou partie de la pile opérationnelle d'une machine à pile. Ainsi, pour notre démonstration, nous avons utilisé le langage *FORTH* qui représente selon nous le canon de la machine à pile. En plus de cette première implémentation du cache de pile qui utilise la translation binaire, nous introduisons notre première proposition de support matériel du cache de pile. Ce support a la spécificité de donner le contrôle du cache de pile matériel explicitement au seul logiciel, par exemple au gestionnaire de machine virtuelle ou au système *FORTH* comme dans notre exemple.

Dans le second chapitre, nous étendons le concept de cache de pile pour introduire notre pile matérielle et son implémentation pour l'architecture *EPIC*. Contrairement au cache de pile, le contrôle de la pile matérielle est implicite et échoit au seul matériel. En limitant certaines caractéristiques bien choisies de la pile matérielle, nous proposons une implémentation réaliste qui s'avère être très peut intrusive au niveau architectural. En offrant un contrôle implicite de la pile matérielle, nous pouvons l'utiliser telle-quelle avec des langages et des environnements de

programmation évolués tel que le *MSIL* (*Microsoft Intermediate Language*) de l'infrastructure *.NET*. En guise d'illustration, nous introduisons et évaluons notre traducteur binaire de *MSIL* vers du code machine *Itanium* en simple passe dans le but de simplifier l'implémentation d'une machine virtuelle *.NET*.

Le troisième chapitre concerne la consommation de puissance. Le travail réalisé dans ce chapitre est indirectement relié aux deux chapitres précédents. Il correspond à une demande de plus en plus grande chez les utilisateurs – dont les administrateurs de centres de calculs et d'hébergement – et les administrations nationales. En effet, les processeurs qui équipent les serveurs d'entreprises et en particulier les membres de la famille *Itanium* ont une *Thermal Design Power* (*TDP* – qui désigne la puissance maximale que chacun des composants du système est autorisé à dissiper) comprise entre 75 et 104 Watts. Ceci pour les modèles de dernière génération. Nos modèles de tests ont une TDP de 130 Watts. Avec des *TDP* de cet ordre, il nous a semblé important d'étudier l'impact du logiciel sur la consommation d'énergie de la plate-forme. Ainsi, dans le troisième chapitre, nous présentons l'état d'avancement de cette recherche dont l'objectif principal est d'étudier et de proposer des techniques de transformation de code pour réduire la consommation énergétique des applications. Pour mener à bien nos mesures, nous introduisons une interface standard – en cours d'étude pour adoption par le *Green Grid* (The Green Grid Consortium 2008) – pour instrumenter les logiciels d'entreprise. Enfin, différentes techniques de transformation de code sont évaluées. Nous menons actuellement cette tâche dans le cadre d'un projet de recherche interne à *Intel Corporation*.

Dans le dernier chapitre de ce mémoire, nous présentons un bilan général des différents travaux que nous avons réalisés dans le cadre de cette thèse. Ce chapitre est l'occasion de les mettre en perspective et de décrire les projets que nous souhaitons mener à bien au cours des prochaines années. En particulier, nous présenterons les derniers développements autours de notre participation au comité du *Green Grid*, une tâche que nous poursuivrons de concert avec *Intel Corporation*.

Notre annexe propose pour sa part deux sections. La première est destinée à la présentation détaillée de l'architecture *EPIC* et des processeurs *Itanium 2*. Le rôle de cette section est d'apporter les précisions nécessaires à la lecture de ce document sans pour autant en encombrer le corps du document. La seconde section de l'annexe présente pour sa part les fondamentaux de la machine – virtuelle – *FORTH* en détaillant le rôle et le mode de fonctionnement de ses interpréteurs et de son compilateur.

6. Références

Abrash, Michael. «Larrabee: A Many-Core x86 Architecture for Visual Computing.» *International Conference on Computer Graphics and Interactive Techniques archive.* Los Angeles, California : ACM SIGGRAPH , 2008. Article No. 18.

Borkar, Shekhar. «Design challenges of technology scaling.» *IEEE Micro*, 1999: 23-29.

Commware Technical Services. Commware Technical Services. 8 Septembre 2008. http://www.comwaretech.com/PDP-11/PDP-11-emulator.html (accès le Septembre 8, 2008).

DBIT. DBIT. 8 Septembre 2008. http://www.dbit.com/ (accès le Septembre 8, 2008).

Domeika, Max, et Jamel Tayeb. Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture . Elsevier, 2008.

Gerosa, Gianfranco, et al. «A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-κ Metal Gate CMOS.» *IEEE International Solid-State Circuits Conference.* 2008.

Iyer, Ravi, et al. «QoS policies and architecture for cache/memory in CMP platforms.» (ACM SIGMETRICS Performance Evaluation Review) 2007.

Moore, Gordon E. « Cramming more components onto integrated circuits. » (Electronics) 38, no. 8 (1965).

Neiger, Gil, Amy santoni, Felix Leung, Dion Rodgers, et Rich Uhlig. « Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. » (Intel technology Journal) 10, n° 3 (2006).

Niar, Smail, et Jamel Tayeb. *Les processeurs Itanium: Programmation et optimisation.* Paris: Eyrolles, 2005.

Ronen, Ronny, Avi Mendelson, Konrad Lai, Shih-Lien Lu, Fred J. Pollak, and John P. Shen. « Microarchitecture challenges in the coming generations of CMOS process technologies. » *32rd Annual International Symposium on Microarchitecture - MICRO-32.* 1999. 2.

Tayeb, Jamel. « Virtualization and Partitionning.» Dans Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture », de Domeika Max, Chapter 9. Elsevier, 2008.

The Green Grid Consortium. The Green Grid. 8 Septembre 2008. http://www.thegreengrid.org/home (accès le September 8, 2008).

Vangal, Sriram, et al. « An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS. » *IEEE International Solid-State Circuit Conference.* 2007.

Wang, Perry H., et al. « Helper Threads via Virtual Multithreading On An Experimental Itaniumr. » (ACM SIGOPS Operating Systems Review) 38, n° 5 (2004).

Cache de pile pour architecture *EPIC*

Résumé

Dans ce chapitre, nous étudions une utilisation novatrice des registres du processeur Itanium 2 *pour optimiser les performances des machines virtuelles. Ainsi, nous montrons qu'il est possible de cacher les piles d'évaluation des machines virtuelles à même les registres du processeur pour en améliorer sensiblement les performances. Nos expériences, menées sur notre implémentation du système* FORTH, *montrent des gains de performances moyens de l'ordre de 7x. Pour autant, notre implémentation purement logicielle souffre de certaines pénalités que nous proposons d'éliminer par l'ajout d'un support matériel pour accélérer l'accès indexé aux registres du processeur. Après l'étude d'une approche systématique du problème, mais difficilement implémentable de façon réaliste, nous présentons une version simplifiée mais très peu intrusive.*

1. Introduction

La virtualisation – au sens large – s'est avérée être un moyen efficace et peu couteux pour exploiter le budget de transistors toujours croissant depuis les années 70. Nous pouvons à ce titre considérer la virtualisation comme une parallélisation à très forte granularité. Ainsi, de nouvelles technologies, avec leur cohorte de nouvelles instructions, font leur apparition dans les architectures généralistes avec pour unique objectif d'améliorer les performances des machines virtuelles (R. Uhlig, et al. 2005). En multipliant le nombre des machines virtuelles, il est possible d'exploiter facilement les ressources massives des plates-formes les plus récentes. A titre d'exemple, un serveur multiprocesseurs *grand public* au goût du jour n'offre pas moins de 24 cœurs physiques et 48 cœurs virtuels ! Le problème essentiel est que les logiciels pouvant tirer profit d'un tel niveau de parallélisme sont extrêmement rares.

Mais les machines virtuelles amènent leur propre lot de problèmes, notamment en termes de performances. Au final, la performance d'une machine virtuelle, qui se résume à un moteur d'interprétation bâti autour d'une pile d'évaluation, dépend avant tout de son interaction avec l'architecture du processeur sous-jacent. La recherche et l'industrie explore plusieurs domaines d'investigation en ce sens. L'un d'entre eux est la mise au point de circuits spécialement conçu pour gérer directement certaines machines virtuelles (Michael O'Connor et Tremblay 1997) (Schoeberl 2005) (Imsys 2008).

Dans ce chapitre nous allons montrer qu'il est possible d'utiliser l'architecture généraliste *EPIC – Explicit Parallel Instruction Computer* – pour arriver à un résultat comparable et combler ainsi l'écart théorique qui sépare les circuits spécialisés et le processeur *Itanium 2* – un processeur généraliste – pour ce même exercice. Nous avons fait le choix d'utiliser le processeur *Itanium 2* car il représente selon nous un bon candidat potentiel pour exécuter le code d'une machine virtuelle efficacement. En plus des avantages intrinsèques de son architecture *VLIW – Very Long Instruction Word* – qui tend à relever le défi de l'exploitation optimale de l'*ILP* –

Instruction Level Parallelism – (Schlansker, et al. 1997), le processeur *Itanium 2* intègre un vaste champ de registres que nous utilisons de façon originale. La dernière raison de ce choix est notre expérience personnelle de plusieurs années au sein du groupe *Software & Solutions* d'*Intel Corporation*, de ce type de système.

Enfin, et en guise de machine virtuelle de référence pour cette étude, nous avons choisi le système *FORTH*, qui malgré son ancienneté, est l'archétype de la machine virtuelle. Le langage ou le système *FORTH* – nous reviendrons sur cette distinction plus tard – a été inventé par *Charles H. Moore* pour satisfaire ses propres besoins informatiques (Rather, Colburn et Moore 1993).

2. Travaux connexes

La communauté *FORTH* a exploré les gains en termes de performance que peuvent offrir les circuits spécialisés ou *ASIC – Application Specific Integrated Circuits*. Bien que chaque développement aie un cahier de charges spécifiques, nous pouvons toutefois dégager trois caractéristiques fondamentales pour l'ensemble des projets que nous avons répertoriés. Ces caractéristiques se retrouvent systématiquement dans les circuits dédiés les plus aboutis et les plus performants.

La première de ces caractéristiques fondamentales est l'intégration à même le processeur d'au moins deux mémoires dédiées pour conserver les deux piles essentielles de l'interpréteur *FORTH*, à savoir la pile d'évaluation et la pile des retours (P. Koopman 1987) (Hayes et Lee 1988) (Schelisiek 2004) [1]. Le nombre, le rôle et l'implémentation des piles intégrées au processeur est variable d'un circuit à un autre. Ainsi, le *Stack Frame Computer* (P. Koopman 1987), conçu pour exécuter essentiellement du code *FORTH* mais aussi du code généré par un compilateur C, intègre plusieurs piles. La première implémentation proposée dispose ainsi de cinq piles. Le circuit dispose de deux bus. Le premier accède à la mémoire centrale (*MBUS*) et le second est dédié à l'interconnexion des piles (*SBUS*). Ces bus sont multiplexés et ils offrent la possibilité de lire simultanément les instructions depuis la mémoire et les données depuis les piles. L'*ALU – Arithmetical and Logical Unit* – dispose d'un unique registre de sommet de la pile – *TOS* ou *Top Of the Stack* – qui agit sur la pile utilisée par l'instruction en cours d'exécution. C'est au programmeur de s'assurer de la gestion des piles puisque le composant n'offre pas de support particulier à cet effet. Le résultat de toutes les opérations effectuées par l'*ALU* est conservé dans le registre *TOS* qui agit comme un tampon. Le second registre de l'*ALU* (*ALUI – ALU Input*) peut recevoir sa donnée depuis l'un des deux bus (Figure 8). Parmi les huit types de piles qu'il est possible de connecter au *SBUS*, les piles *G* et *S* (*Global Stack* et *Stack*) correspondent à des piles d'évaluation. La pile *R* (*Return*) correspond à la pile de retour utilisée pour le contrôle de flux. Le rôle des autres piles est très spécifique (*L* pour le compteur de

[1] Nous présenterons dans la section suivante les mécanismes et les structures fondamentales du langage *FORTH*

boucles, *P* pour le compteur ordinal – utilisé pour sauvegarder les adresses de sous-programmes, *etc.*).

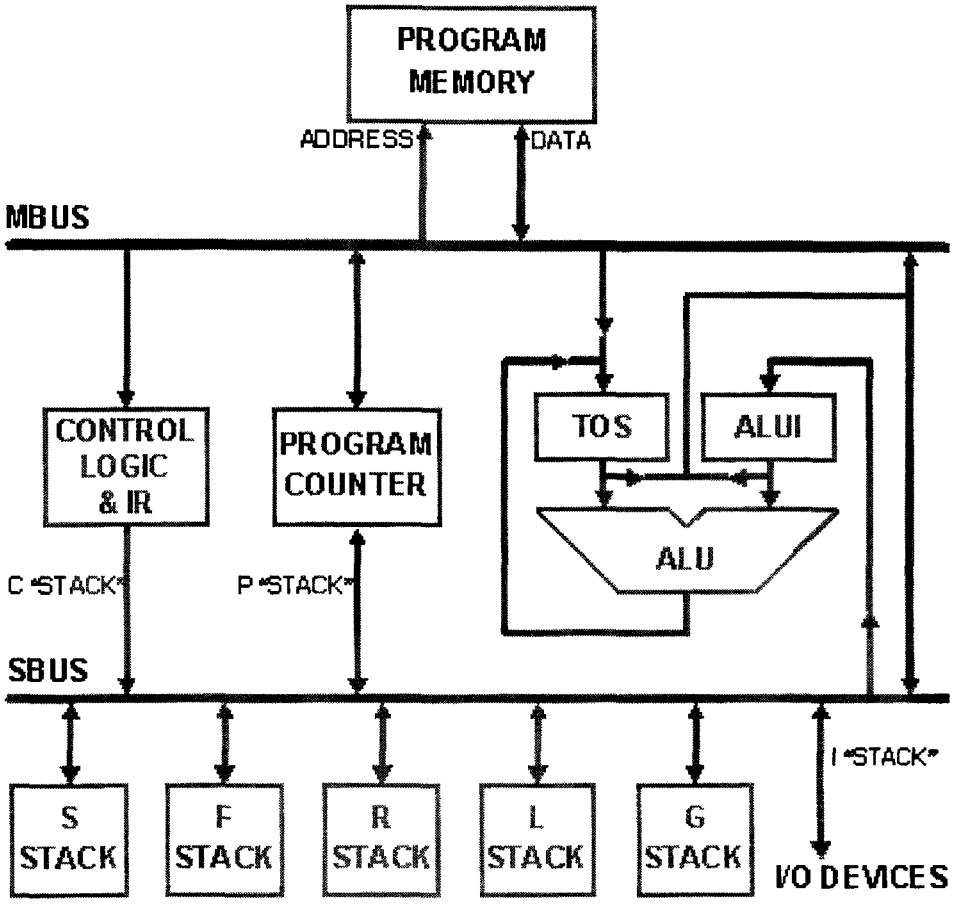

Figure 8 – Diagramme du Stack Frame Computer 1. Source : Stack Computers: the new wave© Copyright 1989, Philip Koopman.

La seconde caractéristique fondamentale est la présence de quelques registres dédiés à la gestion des piles. Le minimum étant un pointeur de sommet de la pile – *TOS* – pour chacune des piles (d'évaluation et de retours), intégrées ou pas au processeur. Pour permettre un accès rapide aux données enfouies dans une pile, des registres supplémentaires peuvent être proposés. Généralement, en y écrivant une valeur, le matériel peut générer l'adresse de n'importe quel élément de la pile. C'est l'approche retenue par les microcontrôleurs de la famille *HS-RTX* (Hand 1990) (Figure 9).

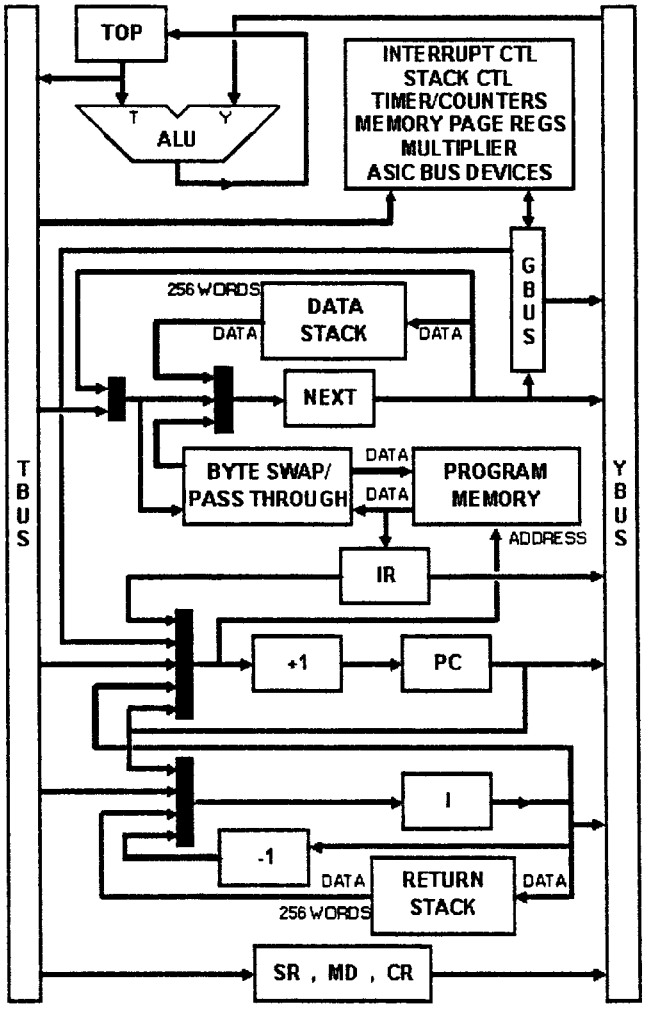

Figure 9 – Diagramme du HS-RTX-2000. Source : Stack Computers: the new wave© Copyright 1989, Philip Koopman.

La troisième et dernière des caractéristiques fondamentales est la faible latence d'exécution des instructions. Le plus souvent, celle-ci est d'un cycle d'horloge. Cette optimisation spécifique permet l'implémentation efficace des primitives du langage *FORTH*. Différentes approches sont possibles pour atteindre ce résultat. Par exemple, les premiers niveaux de la pile peuvent être cachés dans des registres directement reliés aux *ALU*. C'est l'approche retenue par le *Writable Instruction Set Computer* (P. Koopman 1987). Une autre technique consiste à combiner et superposer des cycles d'accès aux bus comme le font le *Minimum Instruction Set Computer* et le *FORTH Reduced Instruction Set Computer*. Ce dernier utilise des bus dédiés pour accéder dans le même cycle d'horloge au sommet de la pile et à n'importe lequel des quatre premiers éléments de la pile d'évaluation et de retour (Figure 10).

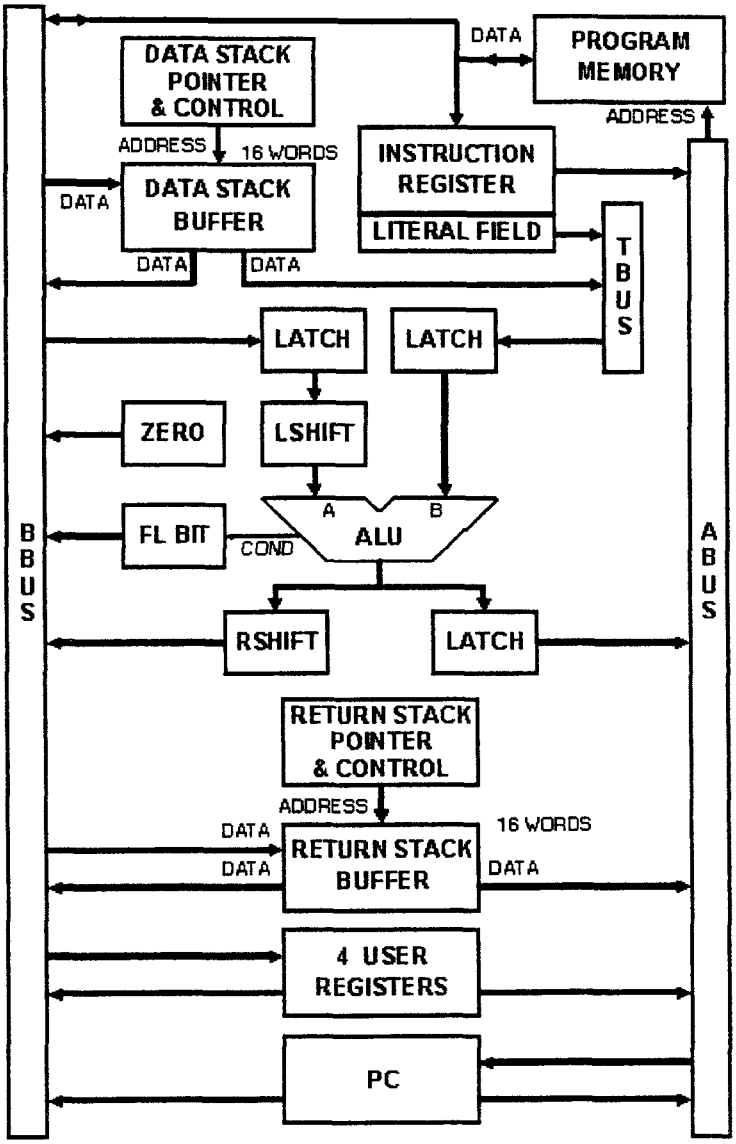

Figure 10 – Diagramme du FORTH Reduced Instruction Set Computer. Source : Stack Computers: the new wave© Copyright 1989, Philip Koopman.

Le projet *Open Source MicroCore* est une des implémentations les plus récentes d'un microcontrôleur dédié pour le langage *FORTH*. Signalons cependant que le *MicroCore* permet l'exécution de code écrit avec d'autres langages de programmation comme le C. Cependant, c'est avec le *FORTH* que ce microcontrôleur se distingue, puisqu'il utilise ce langage en guise d'assembleur (Figure 11). Il intègre une pile d'évaluation et de retour et implémente directement 25 mots *FORTH* dont la plupart s'exécutent en un cycle d'horloge (Schelisiek 2004).

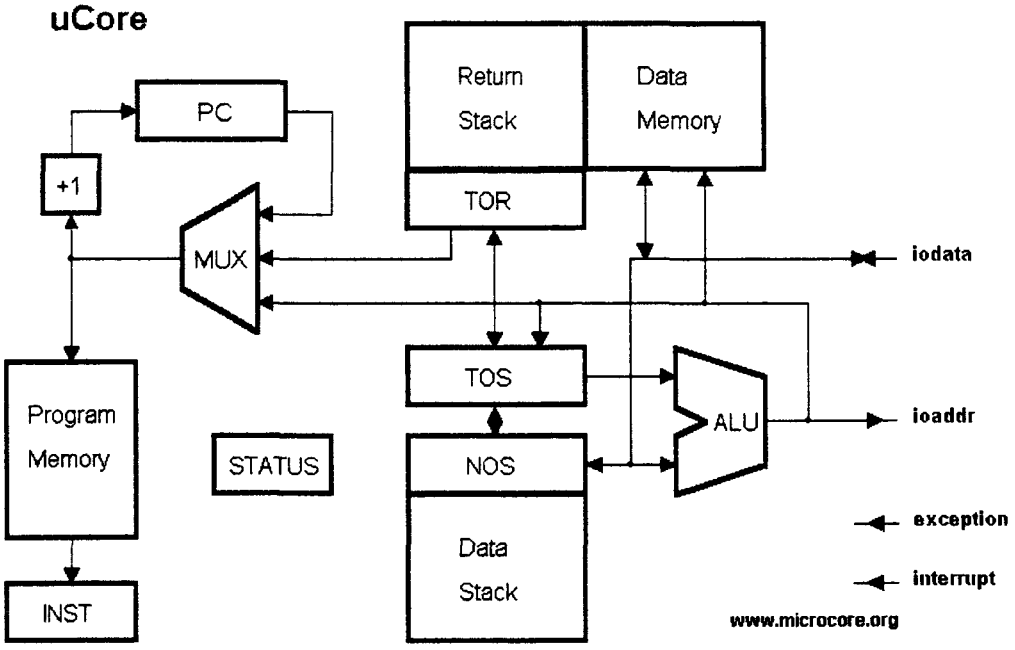

Figure 11 – Diagramme du MicroCore. Source : microcore.org.

Bien que le processeur *4Stack* ne soit pas implémenté (Paysan 2000), il est intéressant d'en présenter le concept dans le cadre de nos travaux. En effet, il s'agit d'une architecture *VLIW – Very Large Instruction Word* – qui intègre à même le processeur quatre piles d'évaluation. Il s'agit de 32 registres de 32 bits organisés en quatre groupes. Chaque groupe de registres est une pile *LIFO – Last-In, First-Out* (Figure 12). Chaque pile est connectée à une *ALU* et chaque *ALU* peut accéder, *via* un cross-bar, à n'importe quel sommet des autres piles. En revanche l'accès aux autres niveaux d'une pile n'est possible que depuis l'*ALU* qui lui est relié. Enfin, chaque pile dispose de son circuit de *spill* et de son circuit de *fill* dédiés pour gérer les débordements vers la mémoire centrale. Certaines opérations peuvent êtres exécutées en parallèle avec des opérations de gestion de pile, ce qui permet théoriquement au compilateur de les programmer pour une exécution simultanée pour absorber la latence de la gestion de la pile.

Contrairement aux autres processeurs spécifiques que nous avons présentés jusqu'ici, *4Stack* n'a pas pour vocation d'exécuter préférentiellement le langage *FORTH*. Cela est bien entendu possible, faille-t-il encore réécrire le système *FORTH*. Cela est essentiellement dû au fait que pour exploiter le parallélisme de l'architecture *VLIW*, il faut utiliser les quatre piles simultanément. D'une façon générale – et à l'instar d'autres architectures *VLIW* – la responsabilité d'extraire des instructions indépendantes depuis le code source des applications échoit ici aussi aux compilateurs. Cette tâche, bien qu'ardue, n'est pas insurmontable. Mais avant tout, il faudrait définir ce que peut apporter le parallélisme d'instruction – *ILP* (*Instruction Level Parallelism*) – au langage *FORTH*, sujet qui n'est pas couvert par les concepteurs de cette architecture.

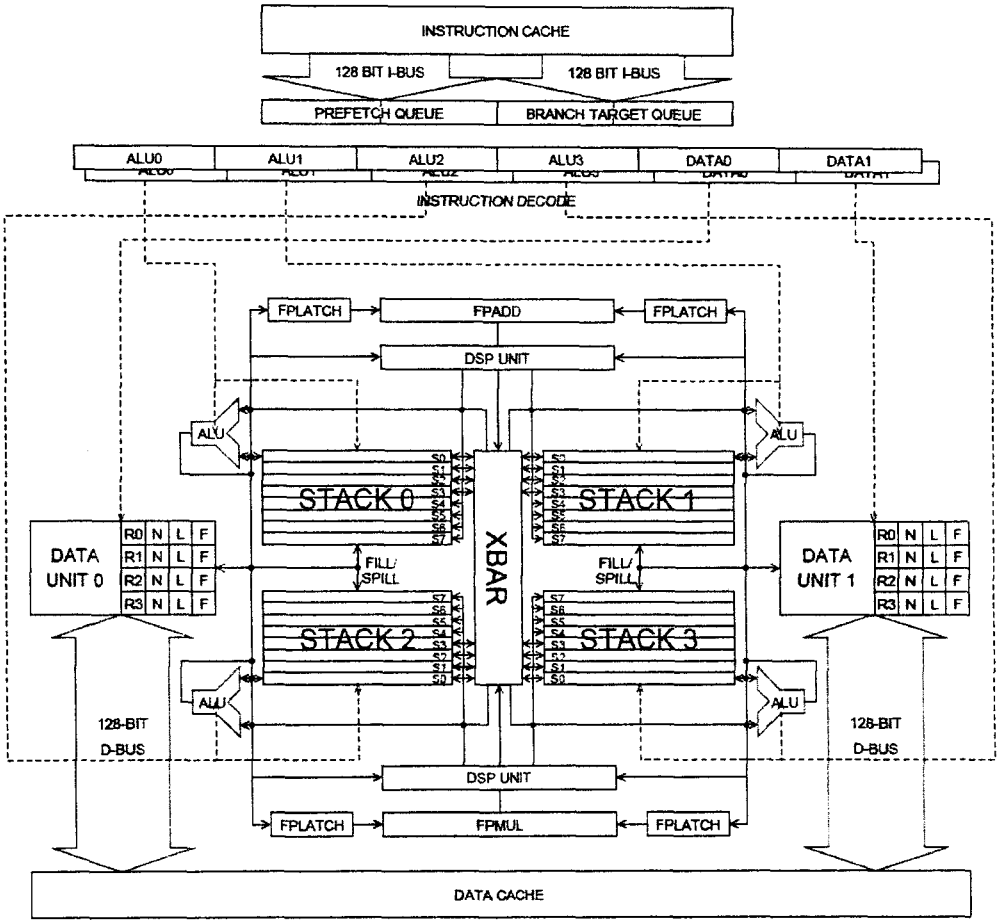

Figure 12 – Diagramme du processeur 4Stack.

Pour conclure notre présentons des circuits spécialisés, il nous faut nécessairement introduire la famille des microcontrôleurs *MuP21* d'*UltraTechnology*. Cette architecture a été conçue par *Moore* et *Chen-hanson Ting* (Ting et Moore 1995). Le microcontrôleur *F21*, la dernière génération de la famille *MuP21* intègre en plus du processeur *FORTH* un circuit de génération de signal vidéo et un circuit d'entrée-sortie série et parallèle (

Figure 13). La pile d'évaluation de la dernière mouture dispose de 18 niveaux. La pile de retour dispose pour sa part de 17 niveaux. Les codes d'opérations sont encodés sur 5 bits, ce qui permet de regrouper 4 instructions par mot et découpler ainsi l'exécution des instructions des accès en mémoire. Les instructions ne sont pas exécutées en parallèle mais bien de façon sérielle. Le registre *T* est le *TOS* dont le contenu est empilé dans *S*. L'*ALU* opère pour sa part sur les registres *T* et *S*. Le processeur dispose enfin d'un compteur ordinal et d'un registre d'adresse (les adresses étant codées sur 20 bits – plus 1 bit pour la retenue et / ou la sélection de page

d'adresses). La Table 1 résume les principaux avantages et les inconvénients de ces circuits spécialisés.

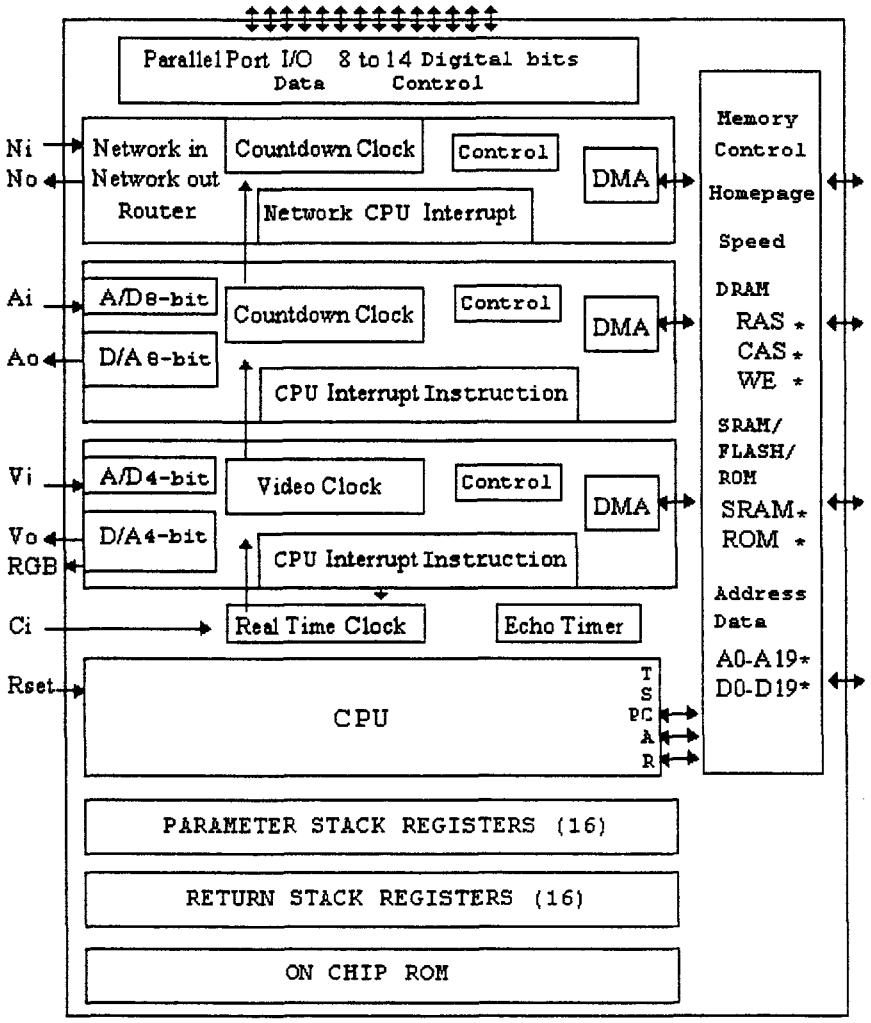

Figure 13 – Diagramme du processeur F21. Source : ultratechnology.com.

Parallèlement à la conception de circuits spécifiques, la communauté *FORTH* a également cherché à optimiser l'exécution du langage sur des processeurs généralistes. Bien qu'il s'agisse cette fois-ci d'optimisations logicielles, le principe fondamental reste identique. Il consiste à cacher un nombre plus ou moins important de niveaux de la pile d'évaluation et éventuellement de retour dans les registres du processeur. Cette mise en cache peut se faire de façon statique ou dynamique lors de l'exécution du code (Ertl et Gregg 2004). Bien que l'efficacité de ces techniques dépende de la nature des codes exécutés, les auteurs obtiennent des gains de performance de l'ordre de 4x en moyenne. En revanche, il est intéressant de constater

que ces mêmes travaux tendent à montrer que ces gains de performance diminuent avec le nombre des niveaux de pile cachés. Pis encore, la taille du code généré croit considérablement. En effet, les mécanismes proposés reposent sur la génération de différentes combinaisons de code qui permettent différentes combinaisons d'accès aux registres. En pratique, au-delà de trois niveaux de pile caché ; l'accroissement de la taille du code devient rédhibitoire. *Wu* et *Lueh* (Peng, Peng et Lueh 2004) montent toutefois qu'en cachant le haut de la pile et en utilisant conjointement des techniques de cache de trace d'exécution il est possible de limiter l'accroissement de la taille du code (Peng, Peng et Lueh 2004). Là où *Ertl* duplique le code de la machine virtuelle en fonction des différents états que la pile cachée peut avoir pour chaque instruction exécutée par l'interpréteur, *Wu* et Lueh réordonnent les instructions du code de gestion du cache de pile de façon à réutiliser au mieux le code de gestion des états entre eux dynamiquement. Ceci est possible puisque contrairement à *Ertl*, *Wu et Lueh* maintiennent une association constante entre les registres physiques du processeur et les niveaux de pile qu'ils cachent. Cela se traduit toutefois par une répétition de certaines instructions de manipulation de cache de pile –. Les expériences de *Wu* et Lueh montrent que, bien que redondantes, ces instructions et les copies de registre restent peux coûteuses sur les architectures modernes. Ainsi, la machine virtuelle de *Wu* utilisant son cache de pile hybride génère un gain de performance de 13,6% en moyenne par rapport à un interpréteur *threadé*. Du point de vue de la taille de code, Un accroissement de la taille de l'interpréteur de seulement 1% est rapporté. Signalons dès à présent que notre approche diffère ici par le fait que notre cache de pile est intégralement conservée dans les registres du processeur et que des registres spécifiques servent de registre d'index. Ainsi, nous n'avons pas à payer le coût supplémentaire de copies entre les registres dévolus au cache de pile. Si avec un cache de pile de trois entrées *Wu et Lueh* obtiennent les meilleurs gains de performance, ceux-ci disparaissent au-delà de cinq niveaux avec la plupart des benchmarks. Quelle que soit la taille du cache de pile (jusqu'à 64 entrée), nous ne payons le surcoût dû à la gestion du cache de pile qu'une seule fois.

Enfin, *Hoogerbrugge et Al* utilisent la technique de mise en cache partiel de la pile conjointement avec d'autres techniques pour améliorer le degré de compression du code généré par une machine virtuelle. Le code critique – du point de vue des performances – est compilé en code natif. Le reste du code est compressé et est exécuté par un interpréteur, qui est lui-même optimisé pour l'architecture *VLIW* (*pipeline* logiciel de trois étages, interpréteur *threadé* et cache de pile). En utilisant le média processeur *TriMedia TM1000* de *Philips Semiconductors*, les auteurs atteignent pour le code interprété une latence d'exécution de 4 cycles par instruction en régime de pointe – 6,27 cycles en moyenne. En pratique, les codes applicatifs testés sont compressés d'un facteur 5x et leur performance est dégradée d'un facteur 8x (Hoogerbrugge, Augusteijn et Van De Wiel 1999).

Circuits	*Avantages*	*Inconvénients*
Stack Frame Computer	Cinq piles ou plus de taille variable (dépend de l'implémentation). Deux bus multiplexés pour la lecture simultanée des	Gestion des piles sous le contrôle du logiciel. Architecture scalaire.

	instructions et des données. TOS câblé directement à l'ALU.	
HS-RTX-2000	Pile de donnée et de retour. Deux bus multiplexés pour la lecture simultanée des instructions et des données. TOS câblé directement à l'ALU.	Architecture scalaire.
FORTH Reduced Instruction Set Computer	Pile de donnée et de retour. Bus dédié pour accéder au TOS et à l'un des quatre premiers niveaux de pile des données dans le même cycle.Quatre registres utilisateur.	Architecture scalaire.
MicroCore	Pile de donnée et de retour. 25 mots FORTH forment l'ISA. La majorité s'exécute en un cycle. Open Source et générique.	Architecture scalaire.
4Stack	Quatre piles génériques. Quatre blocs de huit registres (32-bits). Une ALU par pile. CrossBar pour échange de données entre piles. Spill / Fill matériel en mémoire centrale. Architecture VLIW.	Non implémenté.
F21	Pile de donnée et de retour. Pseudo-VLIW. Conçu par l'inventeur du FORTH.	FORTH uniquement. Architecture scalaire et adressage otique.

Table 1 – Avantages et inconvénients des principaux circuits spécialisés dans l'exécution d'un système FORTH.

En guise de comparaison avec les solutions présentées dans cette section, le travail, que nous présentons dans le restant de ce chapitre, se situe entre les deux approches qui sont respectivement la conception d'un circuit spécialisé et une couche logicielle exécutée par une architecture généraliste. En effet, nous avons opté pour l'utilisation d'une architecture généraliste que nous avons étendu afin d'offrir un support matériel pour l'exécution de machines à piles. Cette approche bénéficie à la fois des avantages et des inconvénients des deux méthodes. Toutefois, nous pensons que les avantages sont plus importants que les inconvénients.

3. Code de référence et utilisation de la pile de registres

Notre implémentation de référence de la machine virtuelle *FORTH* exécute du code chaîné indirect (*indirect threaded code*) et fonctionne suivant les principes décrits dans l'annexe du rapport consacré à la présentation des interpréteurs et du compilateur *FORTH (Annexe 2)*. Le code de référence utilise une pile d'évaluation et de retour maintenue dans la mémoire centrale. Le passage des paramètres se faisant *via* la pile d'évaluation en mémoire. Le code de référence de notre implémentation de référence est écrit en quasi majorité en langage C pour des raisons de simplicité de mise au point mais aussi parce qu'il est recommandé de ne pas coder en assembleur pour les processeurs *Itanium*. La logique derrière cette dernière recommandation réside dans le fait que le compilateur a la charge d'optimiser le code des applications et d'y déceler le plus de parallélisme possible au niveau des instructions et d'utiliser au mieux l'architecture *VLIW* sous-jacente du processeur. En conséquence de quoi, le binaire du système de référence est généré avec un compilateur optimiseur pour l'architecture *EPIC* (*Microsoft Visual C++ 2005 for Itanium*). Nous avons ainsi implémenté la grande majorité des mots définis par le standard de *FORTH X3.215-1994 ANS* (X3J14 1994). Lorsque nous avons codé certaines sections du code de notre *FORTH* en assembleur, nous avons utilisé l'assembleur *EPIC ias* d'*Intel Corporation*. En effet, l'architecture 64 bit interdit l'utilisation de l'assembleur en ligne (`__asm`).

En guise d'introduction à notre proposition, nous allons commencer par examiner le code assembleur *EPIC* généré par le compilateur d'un exemple simple. Par cet exercice, nous souhaitons aider à la compréhension du document mais ne voulons pas être exhaustifs sur le sujet. Plus de détails sur l'architecture *EPIC* et son assembleur sont donnés en annexe de ce document. Précisons également que l'examen visuel du code assembleur généré par un compilateur *EPIC* est une technique quasi-incontournable pour s'assurer que le compilateur a généré du bon code et / ou n'a pas généré de code connu pour réduire sensiblement les performances. L'idée étant qu'après l'examen visuel du code assembleur, l'utilisateur peut aider le compilateur à améliorer le code généré au travers de directives de compilation, et le cas échéant par la réécriture du code source (Niar et Tayeb, Les processeurs Itanium: Programmation et optimisation 2005).

En guise d'exemple, étudions le code de branchement vers le *CFA* du prochain mot à exécuter et de mise à jour du pointeur de mot – encapsulé dans une structure de la machine virtuelle *FORTH* (`pf`) – et plus précisément la ligne code :

```
(pf->internals.ip->cfa)(pf);
```

Pour rester simple, nous dirons que le *CFA* (ou *Code Field Address*) correspond au pointeur sur le début de la définition d'une instruction (mot *FORTH*), qu'il soit défini en *FORTH* ou en langage machine. Le rôle et l'usage du *CFA* sont détaillés en annexe 2 de ce document. Le désassemblage est donné par le Listing 1. Nous ne reviendrons pas ici sur les détails d'implémentation ou de fonctionnement du code. L'annexe 1 fournit tous les détails si cela s'avère être nécessaire.

```
      { .mii
1 :       alloc r35 = 2, 3, 1, 0
2 :       mov r34 = b0
3 :       adds r31 = 528, r32
      } … { .mmb
4 :       mov r36 = gp
5 :       mov r37 = r32
6 :       nop.b 0 ;;
      } { .mmi
7 :       ld8 r30 = [r31] ;;
8 :       ld8 r29 = [r30]
9 :       nop.i 0 ;;
      } { .mmi
10 :      ld8 r28 = [r29], 8 ;;
11 :      ld8 gp = [r29]
12 :      mov b6 = r28
       } { .mmb
13 :      nop.m 0
14 :      nop.m 0
15 :      br.call.dptk.many b0 = b6 ;;
      }
```

Listing 1 – Traduction en assembleur EPIC de la ligne de code branchement vers le CFA

Dans notre exemple, le registre `r32` utilisée par l'instruction 3 correspond au pointeur vers la structure interne de notre *FORTH*, qui est notre argument unique :

```
((pf->internals.ip->cfa)(pf);).
```

Etudions à présent le cas du mot *FORTH* +. Ce mot fondamental (du *CORE word set*) effectue l'addition des deux valeurs présentes au niveau un et deux de la pile. La somme des valeurs y est ensuite empilée. Pour information, la notation adoptée par les programmeurs

FORTH pour spécifier l'action d'un mot sur la pile est donnée sous la forme d'un commentaire *FORTH*. Un commentaire en *FORTH* est une suite de caractères quelconques compris entre parenthèses. Par exemple, pour l'addition, nous noterons : `( n1|u1 n2|u2 -- n3|u3 )`. Le code source en langage C du mot d'addition est donné par le Listing 2.

```
void CORE_PLUS(PFORTH pf) {
            __int3264 n1 = 0;
            __int3264 n2 = 0;
            POP(n2);
            POP(n1);
            PUSH(n1 + n2);
}
```

Listing 2 – Implémentation du mot de l'addition dans notre FORTH de référence.

L'examen du code généré par le compilateur montre essentiellement que les opérations de manipulation de la pile d'évaluation sont implémentés en tant qu'appels de fonctions – le compilateur ayant échoué dans l'optimisation des appels de fonctions sous la forme de code en ligne – *inline* (même en précisant qu'il n'y a pas d'alias d'adresse sur les pointeurs *via* la directive `restrict`). Cela se traduit par un code sous-optimal en terme de performance.

4. Code optimisé et modification de la machine virtuelle

Nous avons donc recodé notre machine virtuelle *FORTH* de façon à ce que la pile d'évaluation soit maintenue à même les registres du processeur. Il s'agit de la technique de mise en cache de la pile telle que nous avons décrit dans la section dédiée à la présentation des circuits spécialisés et des optimisations pour les processeurs généralistes. Puisque le compilateur ne nous est pas d'une grande utilité pour cet exercice, nous avons recodé dans un premier temps notre *FORTH* de référence en assembleur *EPIC*. Le Listing 3 donne le code de la nouvelle implémentation de l'addition.

```
.global EPIC_CORE_PLUS
.type EPIC_CORE_PLUS, @function
.proc EPIC_CORE_PLUS
pfs = r34
EPIC_CORE_PLUS:
            alloc pfs = 2, 1, 1, 0 // allocation des registres pour les param
            add out0 = in0, in1 //addition entre in1 et in2 dans out1
            mov ar.pfs = pfs ;; //Fin premier groupe, …
```

```
                br.ret.sptk.many b0 //retour
.endp
```

Listing 3 – Implémentation en assembleur EPIC de mot FORTH +.

En examinant le code du Listing 3, nous voyons que l'addition est à présent programmée pour deux cycles d'horloge. Rappelons que le nombre de cycles d'horloges programmé par le compilateur et / ou le programmeur s'identifient en dénombrant le nombre de double point-virgule (;;) dans le listing en assembleur. Précisons deux points importants. Premièrement, il arrive que la matérialisation de la rupture de parallélisme dans une séquence d'instructions soit implicite (est qu'elle n'est donc pas marquée dans le listing – pour autant, elle existe bien et est imposée par l'unité de ventilation des instructions vers les ports d'exécution du processeur). Deuxièmement, le compilateur et / ou le développeur codent généralement les cycles suivant les latences idéales d'accès aux mémoires cache (2 cycles à la mémoire de cache de données de premier niveau pour l'*Itanium 2*). En revanche, lors de l'exécution, si les données ne sont pas présentes là où elles sont supposées être, alors des cycles de décrochage du pipeline (*stall*) sont introduits. En d'autres termes, l'exécution d'une opération programmée pour deux cycles peut prendre plus de temps si l'unité d'exécution doit attendre ses données.

La Table 2 compare les ressources requises par les deux implémentations du mot *FORTH* + dans les deux versions de notre *FORTH* pour *EPIC*.

Ressources utilisées	*Version de référence*	*Version optimisée*
Registres généraux	9	2
Registres flottants	0	0
Groupes d'instructions	14	2
Nops	5	3
Bits de stop	10	1
Branchements	6	1

Table 2 – Comparaisons des ressources requises pour le mot FORTH + dans nos deux implémentations du FORTH.

Lors de la réécriture de la machine virtuelle *FORTH*, nous avons initialement exploré l'utilisation du *RSE* et de son mécanisme de gestion des débordements de pile de registres pour simplifier notre code. En effet, le *RSE* permet la sauvegarde (*spill*) momentanée des cadres de registres les plus anciens dans une section de la mémoire vive du système d'exploitation – par exemple lorsqu'il n'y a plus de registres disponibles pour l'instruction `alloca` en cours d'exécution. Ce même mécanisme permet lors d'un retour de fonction dont les registres on étés sauvegardés par un *spill* de les recharger dans la pile de registres (*fill*). Malheureusement, le *RSE* ne gère pas les registres flottants (et notre *FORTH* intègre le vocabulaire adéquat et gère une pile dédiée – comme le permet la norme). En outre, il est quasiment impossible de contrôler précisément l'exécution du *RSE* depuis le code et ce quel que soit le mode dans lequel nous le

positionnons (*eager*, *enforced lazy*, *etc.* (Intel Corporation 2008). Enfin, et peut-être surtout, le coût d'activation du *RSE* en termes de cycles de décrochage du pipeline (*pipeline stall*) est élevé. Il existe d'ailleurs un compteur architectural de performance spécifique qui permet de déceler une forte activité du *RSE* (`BE_RSE_Bubble.All`). C'est ce qui se produit notamment avec les fonctions récursives et les fonctions ayant un nombre élevé de paramètres ou plus simplement qui allouent de nombreux registres locaux.

Suite à nos tentatives infructueuses, nous avons donc pris le parti de recoder notre machine virtuelle en supprimant les appels de fonctions vers les mots *FORTH* de base en proposant une implémentation conventionnelle en langage machine pur. Ainsi, pour effectuer la mise en cache de la pile d'évaluation, nous allouons en entrée de l'interpréteur interne 32 registres flottants et généraux contigus *via* l'instruction `alloca` comme nous le faisions jusqu'à présent, mais avec plus de registres locaux cette fois-ci. Nous nous réservons également huit registres généraux statiques. Nous utilisons ces registres comme registres d'adresse des quatre premiers niveaux des piles d'évaluation (pour les mots *FORTH* opérant sur les entiers et sur les flottants). Deux de ces registres sont les *TOS*, les six autres nous sont utilisés pour générer rapidement les adresses des niveaux de pile 2, 3 et 4 dans les deux piles lorsque nous empilons ou dépilons des éléments (Figure 14).

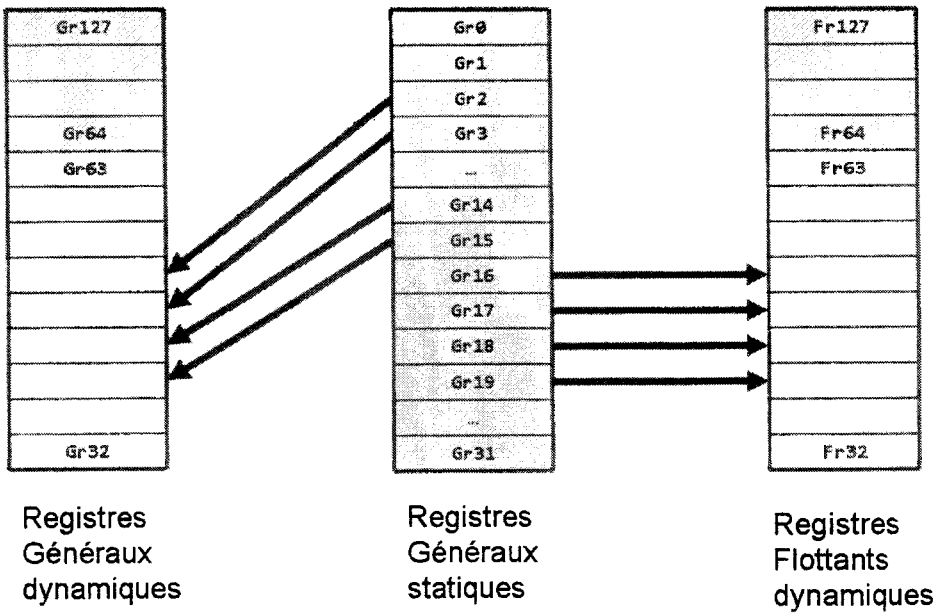

Figure 14 – Organisation logique des piles et utilisation de registres statiques en guise de pointeurs de pile. Les registres hachurés sont exclus de la pile. Les huit registres généraux sont gr2, gr3, gr14 à gr19 inclus.

Pour compléter le mécanisme, nous devons modifier notre machine virtuelle en ajoutant à l'interpréteur interne du *FORTH* une passe de régénération de code avant exécution. En effet, nous devons dynamiquement recoder les adresses physiques des registres qui sont encodées dans les codes d'opération des instructions. Ainsi, avant de pointer le compteur ordinal sur le premier

bundle pointé par le *CFA* du mot à exécuter, notre routine additionnelle – qui est une séquence de code dans l'interpréteur interne – est exécutée. Celle-ci traverse le code machine en une passe et traduit les adresses des registres des instructions en utilisant les valeurs stockées dans les registres d'index que nous avons introduit précédemment. Bien qu'il soit possible d'effectuer cette opération par avance, nous devons l'effectuer de façon dynamique, puisqu'il nous est impossible de prédire l'utilisation de la pile.

Il nous faut encore régler le problème de la cohérence de cache introduit par la modification dynamique du code. En effet, l'architecture *EPIC* suppose que le code applicatif assure cette cohérence et rien n'est mis en œuvre au niveau du matériel pour assurer la cohérence de cache d'instructions. Ainsi, si aucune action spécifique n'est prise en ce sens, alors rien ne nous garantit que le code modifié sera effectivement exécuté en lieu et place d'une copie éventuellement présente en cache de niveau 1. Ce problème est minimisé avec le processeur *Itanium 2* qui dispose d'un cache de premier niveau d'instructions – mais aussi de données – de 16 Ko seulement. Ce premier niveau est secondé par le cache unifié de second niveau (de 256 Ko). Il est ainsi peu probable que le code modifié réside dans le cache *L1I* lors de son exécution après modification puisque toute ancienne copie aura été évincée par le code de translation. En revanche, avec la génération suivante de processeurs *Itanium*, la situation est différente puisque la mémoire cache de second niveau d'instructions est de 1 Mo (le second niveau n'étant plus unifié). Dès lors, il est possible qu'une copie antérieure à la modification de code soit présente dans le cache. Pour palier à ce problème – qui rappelons-le n'apparaît pas sur notre plate-forme de développement –, nous devons synchroniser le cache de premier niveau. Cette mise en cohérence du cache d'instructions de premier niveau se fait en rajoutant les instructions du (Listing 4) juste avant d'effectuer le transfert du compteur ordinal vers le *CFA*.

```
fc.i
nop.i 0
nop.m 0 ;; // vide le cache d'instructions
sync.i ;; // force la synchronisation des caches des cpus distants
srlz.i ;; // vérifie la synchronisation du cache par le processeur local
```

Listing 4 – Code nécessaire au maintien de la cohérence du cache d'instruction pour notre code auto-modifiant.

Il nous est dès lors possible de recoder l'ensemble des primitives *FORTH* pour utiliser les piles cachées. Le Listing 5 donne la nouvelle forme du mot d'addition qui référence le *TOS* et le premier niveau de la pile directement via les registres statiques dédiés à cet effet. Le code de traduction recherche les adresses des registres pointeurs de pile, lit et met à jour leur contenu et ré-encode les nouvelles adresses physiques des registres dans le *bundle*.

```
EPIC_CORE_PLUS:
add r2 = r2, r3 ;;
```

Listing 5 – Implémentation optimisée du mot FORTH + utilisant la pile cachée.

Les Table 3 et Table 4 résument le résultat de nos mesures expérimentales. Pour chacun des treize mots *FORTH* que nous avons sélectionnés car très fréquemment utilisés en *FORTH* (P. J. Koopman 1989) et manipulent profondément la pile – onze sont exclusivement dédiés à la gestion des piles.

Nous indiquons le nombre de cycles d'horloge nécessaires à leur exécution dans la version de référence et la version optimisée utilisant la pile cachée et la ré-encodage des adresses de registre dynamique. Nous constatons des gains d'autant plus importants que les manipulations de piles sont complexes. Ainsi, l'addition que nous avons étudiée précédemment ne gagne qu'un facteur ~2x, `2OVER` atteint ~16x [`2OVER ( n1 n2 - n1 n2 n1 )`].

Mots FORTH	*Cycles d'horloge*	*Gains de performance*
CORE_PLUS (+)	*29.25*	-
EPIC_CORE_PLUS (+)	*15.00*	*1.95*
CORE_TWO_DUP	*48.00*	-
EPIC_CORE_TWO_DUP	*5.00*	*9.60*
CORE_TWO_OVER	*78.00*	-
EPIC_CORE_TWO_OVER	*5.00*	*15.60*
CORE_TWO_SWAP	*62.00*	-
EPIC_CORE_TWO_SWAP	*6.00*	*10.33*
CORE_DUP	*28.00*	-
EPIC_CORE_DUP	*5.00*	*5.60*
CORE_OVER	*41.00*	-
EPIC_CORE_OVER	*6.00*	*6.83*
CORE_ROT	*48.00*	-
EPIC_CORE_ROT	*5.00*	*9.60*
CORE_SWAP	*33.00*	-
EPIC_CORE_SWAP	*5.00*	*6.60*
FLOATING_F_PLUS (F+)	*44.00*	-
EPIC_FLOATING_F_PLUS (F+)	*13.25*	*3.32*
FLOATING_FDUP	*43.00*	-
EPIC_FLOATING_FDUP	*8.00*	*5.38*
FLOATING_FOVER	*64.00*	-
EPIC_FLOATING_FOVER	*14.00*	*4.57*
FLOATING_FROT	*66.00*	-
EPIC_FLOATING_FROT	*7.00*	*9.43*
FLOATING_FSWAP	*51.00*	-

EPIC_FLOATING_FSWAP	7.00	7.29

Table 3 – Comparaisons de la performance de treize mots FORTH essentiels entre la version de base et la version optimisée. Par définition, le code de référence n'a pas gain de performance, ce qui est matérialisé par un tiret (–) dans le tableau.

Mots FORTH	Vocabulaire d'appartenance	Opération effectuées
+	CORE	(n1|u1 n2|u2 -- n3|u3)
2DUP	CORE	(x1 x2 - x1 x2 x1 x2)
2OVER	CORE	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)
2SWAP	CORE	(x1 x2 x3 x4 -- x3 x4 x1 x2)
DUP	CORE	(x -- x x)
OVER	CORE	(x1 x2 - x1 x2 x1)
ROT	CORE	(x1 x2 x3 -- x2 x3 x1)
SWAP	CORE	(x1 x2 - x2 x1)
F+	FLOATING	(F: r1 r2 -- r3) ou (r1 r2 -- r3)
FDUP	FLOATING	(F: r -- r r) ou (r - r r)
FOVER	FLOATING	(F: r1 r2 -- r1 r2 r1) ou (r1 r2 -- r1 r2 r1)
FROT	FLOATING	(F: r1 r2 r3 -- r2 r3 r1) ou (r1 r2 r3 -- r2 r3 r1)
FSWAP	FLOATING	(F: r1 r2 -- r2 r1) ou (r1 r2 -- r2 r1)

Table 4 – Liste des mots FORTH testés avec leur vocabulaire d'appartenance et leur action sur la pile.

Pour autant, notre implémentation optimisée a son lot d'inconvénients. En effet, comme nous l'avons vu précédemment, l'exécution sur les générations récentes de processeurs *Itanium*, il nous faut assurer la cohérence du cache d'instructions. Or cette synchronisation a un coût qui est loin d'être négligeable puisqu'elle requiert le vidage et la synchronisation de la mémoire cache d'instructions (par taille de lignes de cache L3). En outre, l'instruction `fc.i` n'est que le synonyme de l'instruction `fc` qui effectue la même opération sur la mémoire cache de données. Ainsi, en pratique, bien que notre code de synchronisation demande explicitement la synchronisation de la mémoire cache d'instructions, le processeur synchronise également la mémoire cache des données. Rappelons que ce problème n'apparaît qu'avec les nouvelles générations de processeurs ayant une mémoire cache de second niveau non-unifiée et non-uniforme (1 Mo contre 256 Ko partagés entre les instructions et les données). Enfin, tant que le contrôle d'exécution s'effectue dans le code principal de la machine virtuelle, et que nous n'avons pas de débordement de pile (très rare avec du code *FORTH* standard il faut le souligner), les performances sont très bonnes avec le processeur *Itanium 2*. En revanche, dès que le *RSE* est activé – lors d'un appel système par exemple – alors le dorsal du *pipeline* est en décrochage

(d'autant plus longtemps que le nombre des registres à sauvegarder / charger est important). Ainsi, les transitions depuis et vers la machines virtuelle tendent à diluer les gains obtenus à l'aide de la mise en cache des piles dans les registres du processeur.

5. Accès indexés aux registres

Pour bénéficier des gains de performances offerts par la mise en cache de la pile dans les registres dynamiques, et pour ne pas avoir à payer les pénalités dues à une implémentation purement logicielle, nous proposons d'ajouter la possibilité d'accès indexé à un sous-ensemble des registres généraux et flottants dynamiques du processeur. L'indexation pouvant se faire soit *via* un registre général ; soit *via* une valeur immédiate. Nous proposons la notation suivante pour décrire l'indexation : `gr[gr]`, `gr[imm]`, `fr[fr]` ou `fr[imm]`. En guise d'exemple pour notre étude, nous allons reprendre l'organisation que nous avons retenue pour la machine virtuelle *FORTH* optimisée (Figure 15). Avec cette organisation et l'accès indexé aux registres, le code du mot *FORTH* + est donné par le (Listing 6). Notons que la fonction intègre cette fois-ci l'ensemble des contrôles de la pile et de gestion des registres d'index, une tâche qui échoit à la machine virtuelle dans notre solution uniquement logicielle. Notons également que le coût de ces opérations de maintenance est réduit puisque l'architecture *VLIW* permet de programmer ces instructions pour le même cycle d'horloge.

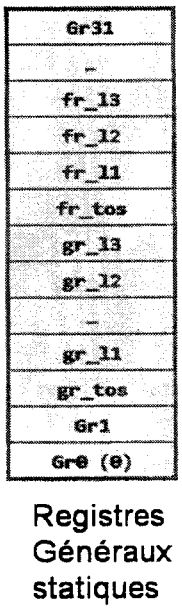

Figure 15 – Association entre les registres généraux et les pointeurs de pile utilisés en guise de registre d'index. Les accolades regroupent les registres utilisés pour la gestion de la pile entière et flottante. Les pointillés symbolisent les registres intermédiaires que nous n'avons pas représentés. Ainsi, entre gr_l1 et gr_l2, les pointillés représentent les registres physiques gr4 à gr13.

```
.global EPIC_CORE_PLUS
.type EPIC_CORE_PLUS, @function
.proc EPIC_CORE_PLUS
EPIC_CORE_PLUS:
            cmp4.lt.unc p6, p0 = r0, gr_tos
  (p6) br.cond.dptk.many @underflow_exception ;;
            add gr[gr_l1] = gr[gr_tos], gr[gr_l1] ;;
            mov gr_tos = gr_l1 ;;
            add gr_l1 = -1, gr_tos
            add gr_l2 = -2, gr_tos
            add gr_l3 = -3, gr_tos
            br.ret.sptk.many b0 ;;
.endp
```

Listing 6 – Code du mot FORTH + utilisant l'accès indexe aux registres.

Voyons à présent comment nous proposons d'implémenter l'accès indexé aux registres pour l'architecture *EPIC*. L'index, qu'il soit fourni sous la forme d'une valeur stockée dans un registre ou bien sous la forme d'une valeur immédiate, est codé sur sept bits, puisque la pile contient 128 registres. Dans le cas d'un index registre général, nous ne retenons que les sept bits de poids faible de la valeur stockée. Quelle que soit la nature de l'index, sa valeur sert à calculer l'adresse du registre indexé à référencer par l'instruction. Par convention, un index nul désigne les registres `gr0` et `fr0`. Un index égal à `0x7Fh` désigne les registres `gr127` et `fr127`. Signalons que l'accès indexé existe dès à présent dans les processeurs *Itanium*. Cependant il est limité à l'instruction `mov` entre les registres architecturaux (Intel Corporation 2008).

En guise d'exemple, nous allons utiliser les instructions `mov` sur les registres généraux et flottants. Comme nous l'indiquions avec les instructions `fc` et `fc.i`, le processeur *Itanium* utilise abondamment des synonymes d'instructions. Ainsi, l'instruction `mov` est en fait – et selon la nature de ses arguments – un synonyme des instructions `add` (`adds r1 = 0, r3` est synonyme de `mov r1 = r3`) et `fmerge` (`fmerge.s f1 = f3, f3` est synonyme de `mov f1 = f3`). Nous avons choisi d'étudier l'instruction `mov` car elle permet à elle seule l'implémentation de l'accès indexé aux registres. En effet, si seule cette instruction disposait de l'accès indexé (à l'instar de sa version agissant sur les registres architecturaux, qui elle gère l'accès indexé), alors nous serions toujours en mesure de coder notre machine virtuelle *FORTH* de façon efficace. Malgré le léger surcoût induit par la gestion des registres d'index de pile et celui des déplacements occasionnels entre registres, nous serions toujours gagnants en termes de performances. Il serait également possible d'optimiser les routines pour exécuter plusieurs de ces instructions de gestion de pile en parallèle.

```
(qp) adds gr[r1] = 0, gr[r3]
(qp) adds gr[imm7₁] = 0, gr[imm7₂]
(qp) adds gr[r1] = 0, gr[imm7₁]
```

```
(qp) adds gr[imm7₁] = 0, gr[r1]
(qp) fmerge.s fr[r1] = fr[r3], fr[r3]
(qp) fmerge.s fr[imm7₁] = fr[imm7₂], fr[imm7₂]
(qp) fmerge.s fr[r1] = fr[imm7₁], fr[imm7₁]
(qp) fmerge.s fr[imm7₁] = fr[r1], fr[r1]
```

Listing 7 – Différentes formes de l'instruction mov indexée.

Du point de vue du logiciel, la méthode la plus simple pour proposer l'accès indexé consiste à étendre le jeu d'instructions du processeur. Il existe plusieurs méthodes pour y parvenir. Nous avons retenu comme critère principal de limiter au minimum l'impact sur le jeu d'instructions existant. Lorsque cela s'avère nécessaire, nous avons opté de sacrifier la taille de code en faveur de la simplicité et la systématicité de l'implémentation. Nous devons en effet ajouter la capacité d'accès indexé aux registres à 89 instructions *EPIC* (sur les 118 du jeu d'instructions). Ces 89 instructions ayant au moins une forme avec un registre source ou destination. Nous nous sommes bien sûr interdit de modifier le format des instructions.

Ainsi, nous avons arbitrairement décidé de recycler le code d'opération majeur réservé pour un usage ultérieur numéro `0x8h` (Schlansker et Ramakrishna Rau 2000).

Nous réutilisons également le bit 36 de tous les formats de *bundle* (*template format*). Précisons que le bit 36 est toujours utilisé comme un bit individuel et qu'il encode différentes informations aussi variées que l'annonce d'une valeur immédiate, le slot marshaling, l'extension du code d'opération, un bit de signe, *etc.* Nous l'utilisons en guise d'indicateur d'utilisation de l'accès indexé. Ainsi, lorsque ce bit est à 1, ceci indique l'utilisation de l'accès indexé. Ce bit vaut 0 dans le cas ou l'accès n'est pas utilisé. Notre proposition d'encodage nécessite aussi l'utilisation d'un slot supplémentaire de 41 bits pour coder ces instructions. Rappelons que le processeur *Itanium* regroupe trois instructions dans un paquet (appelé *bundle)* encodé sur 128 bits et que chaque instruction occupe un slot (première instruction dans le slot 0, deuxième instruction dans le slot 1 et ainsi de suite). Chaque slot étant encodé sur 41 bits. Les cinq bits restants codent pour le patron de *bundle* (*template bits)*. Le recours à un slot supplémentaire est une méthode utilisée par exemple pour charger une constante de 64 bits dans un registre général. En effet, un slot de 41 bits ne suffit pas à encoder la valeur à charger. Un slot additionnel est alors utilisé. Le patron est de type `L+X` (annexe 1).

Avec cette nouvelle organisation, nous commençons par translater le bit 36 vers le bit 40 du nouveau slot. Le bit 36 indiquant l'utilisation ou non du mode indexé. Nous faisons de même pour le code d'opération majeur qui est translaté vers le champ de bits 7-0 du nouveau slot. Dans la mesure du possible, nous avons choisi de repositionner les bits déplacés à la même position dans le nouveau slot. Cela étant dit, les positions sont arbitraires et peuvent changer suivant les contraintes liées à l'implémentation. Le champ de bits 39-36 encode l'extension du code d'opération. Nous devons procéder ainsi pour gérer correctement les formes à quatre opérandes de certaines instructions *EPIC* – minoritaires il est vrai, puisque la majorité d'entre-elles est triadique. Ces quatre bits indiquent la nature de l'index. En effet, un seul bit associé à un index

suffit pour faire le distinguo entre un index sous forme de registre et un index sous forme d'une valeur immédiate codée sur sept bits (Table 5, Figure 16 et Figure 17).

Slot	*Bits*	*Description*
0	36	IDX - 0: non indexé, 1: indexé (recyclé)
1	40	Copie du bit 36 original (translaté)
1	39	x1 - 0: index imm7, 1: index register
1	38	x2 - 0: index imm7, 1: index register
1	37	x3 - 0: index imm7, 1: index register
1	36	x4 - 0: index imm7, 1: index register
1	35-29	imm71 / r1
1	28-22	imm72 / r2
1	21-15	imm73 / r3
1	14-8	imm74 / r4
1	7-0	Copy du code d'opération original

Table 5 – Nouvelle proposition d'encodage autorisant la gestion de l'accès indexé aux registres.

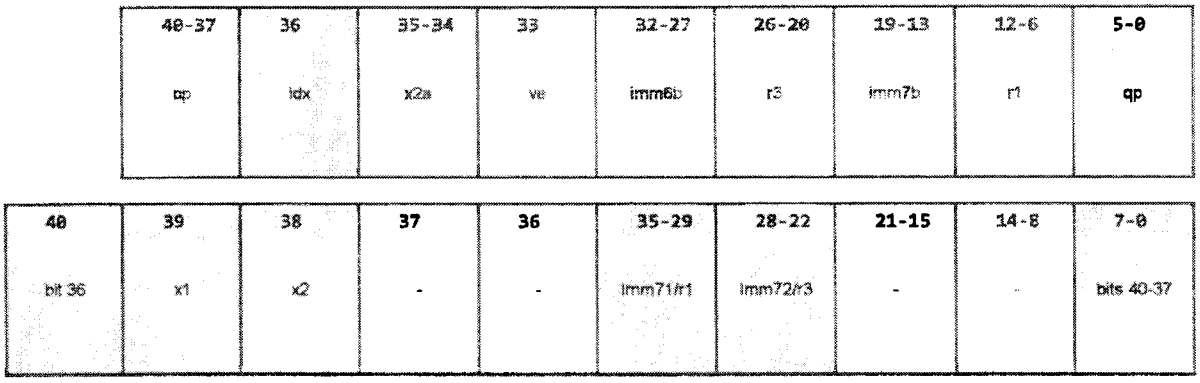

Figure 16 – Proposition de ré-encodage de l'instruction add pour gérer l'instruction mov indexé entre les registres généraux.

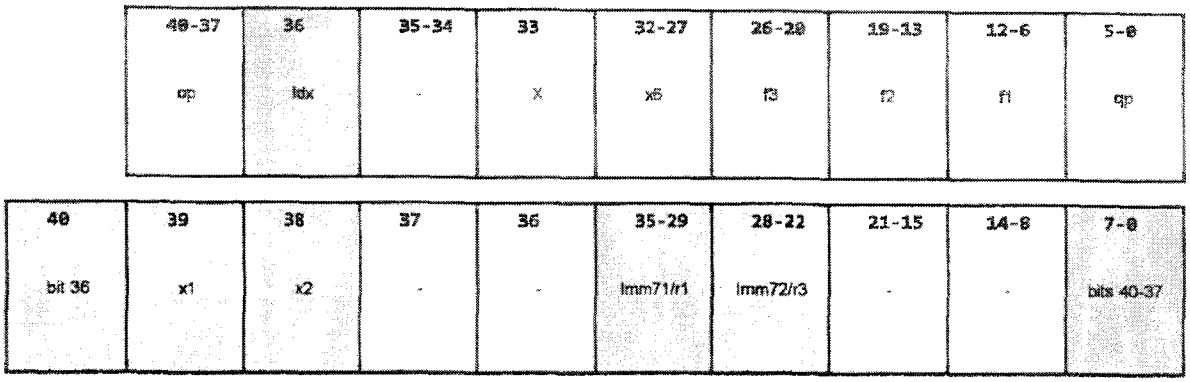

Figure 17 – Proposition de ré-encodage de l'instruction fmerge pour gérer l'instruction mov indexé entre les registres flottants. x1 et x2 les bits indiquant la nature de l'adresse physique de registre ou de la valeur immédiate sur 7 bits (1 = indexé, 0 = non-indexé).

Le décodage des bits `xn` doit prendre place tôt dans le pipeline, dans l'étage de renommage des registres (`EXP/REN`) et alimenter ainsi l'étage `REG` avec les adresses de registre définitives (Figure 18). Pour autant, le recours à l'accès indexé requiert une lecture de registre – si l'index est sous la forme registre – pour récupérer les sept bits de poids faible qu'il contient. Dans cette forme uniquement, et puisque le registre d'index peut être marqué pour modification par une autre instruction, une sérialisation prend nécessairement place et un cycle d'horloge est ajouté au pipeline principal.

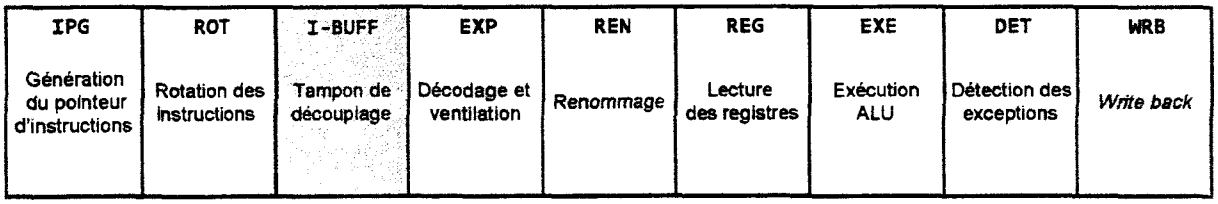

Figure 18 – Organisation originale simplifiée du pipeline du processeur Itanium 2. Le tampon de découplage du frontal est représenté en hachuré.

L'unité de ventilation des instructions (*dispersal unit*) doit également être amendé de façon à ce qu'il puisse reconnaître le code d'opération majeur que nous avons recyclé (`L+X`). Lors de la détection de ce code d'opération, il doit décoder correctement le champ de bits 40-37 qui a été translaté dans le champ 7-0 du slot additionnel.

La nature même de l'instruction `mov` autorise des simplifications importantes. Ainsi, la valeur 0 de l'addition est générée directement depuis le registre `gr0` (`adds r1 = 0, r3`). De façon similaire, l'instruction `fmerge` utilise la même source pour ses deux registres source. Malheureusement, le format de ces deux instructions ne permet pas l'encodage des trois bits `idx`, `x1` et `x2` requis. Les Figure 16 et Figure 17 donnent le nouveau format d'encodage proposé pour ces instructions.

Nous avons opté de sacrifier la taille de code en faveur de la relative simplicité et la systématicité de l'implémentation. Il nous a donc fallu consommer un slot d'instruction supplémentaire pour encoder le code d'opération majeur et les bits indiquant la nature des index pour nos instructions. Si théoriquement, cela peut conduire à une diminution de l'*ILP – Instruction Level Parallelism* – il n'en est rien en pratique. En effet, nous n'avons pas rencontré de nombreuses opportunités de coder plus de deux instructions en parallèle lorsque nous avons développé les mots du dictionnaire de notre système *FORTH*. En effet, la nature même des mots fondamentaux de *FORTH* exhibe généralement de courtes séquences d'instructions fortement dépendantes les unes-des-autres, ce qui rend la tâche d'utilisation optimale des *bundle* très difficile. Ainsi, même si deux instructions d'accès à la pile pourraient être programmées pour le même cycle, alors il serait toujours possible de le faire avec deux *bundles* de type L+X.

6. Implémentation simplifiée

Malgré nos efforts, l'implémentation de la méthode proposée dans la section précédente est peu réaliste vu le nombre important des modifications à apporter à l'encodage et au décodage des instructions, sans parler des ressources nécessaire à la validation. Enfin, le code source des applications doit être recompilé et du code spécifique doit être généré pour utiliser la nouvelle fonctionnalité. En revanche, et en limitant le nombre des registres dynamiques utilisés pour cacher la pile à seulement 64 – ce qui s'avère être largement suffisant pour une implémentation conformable du *FORTH* – nous pouvons proposer une implémentation matérielle très réaliste et qui ne requiert pas la modification du jeu d'instructions et reste compatible au niveau binaire avec les applications existantes.

Pour ce faire, nous conservons l'encodage de l'adresse des registres sur sept bits comme c'est le cas aujourd'hui (Figure 19). Dans cette nouvelle organisation, la responsabilité de ségréguer le contenu de la pile cachée et celui des registres dynamiques échoit au compilateur.

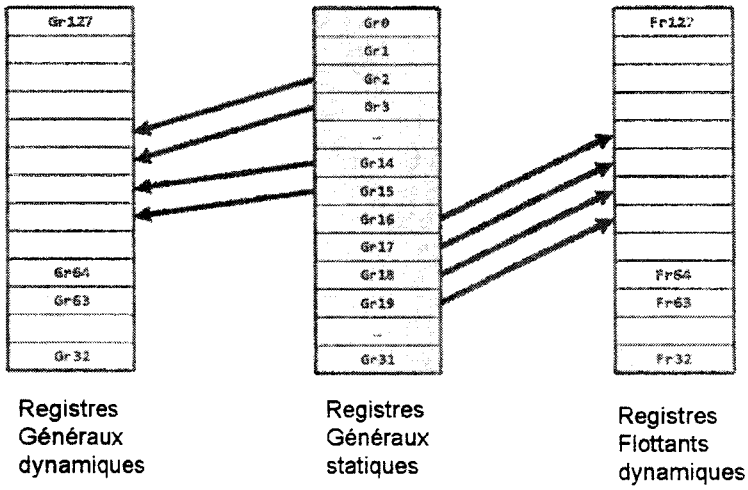

Figure 19 – Organisation logique simplifiée des piles et utilisation de registres statiques en guise de pointeurs de pile. Les registres hachurés sont exclus de la pile.

Nous définissons premièrement la fonctionnalité d'accès indexé aux 64 registres dynamiques généraux et flottants supérieurs (`gr64` – `gr127` et `fr64` – `fr127`). La présence de cette fonctionnalité est indiquée par un bit lu par l'instruction `CPUID`. Un bit supplémentaire est requis dans un *MSR – Model Specific Register* – du processeur. Si ce bit est désarmé, alors la *RAT – Register Allocation Table* que nous décrirons plus loin – est contournée. Un cycle additionnel est ajouté au *pipeline*. Signalons que le code existant peut s'exécuter tel-quel et qu'aucune recompilation n'est nécessaire. En revanche, si un code souhaite utiliser la nouvelle fonctionnalité, alors il devra vérifier la disponibilité de la fonctionnalité sur la plate-forme d'exécution et procéder à son activation *via* le bit de *MSR*.

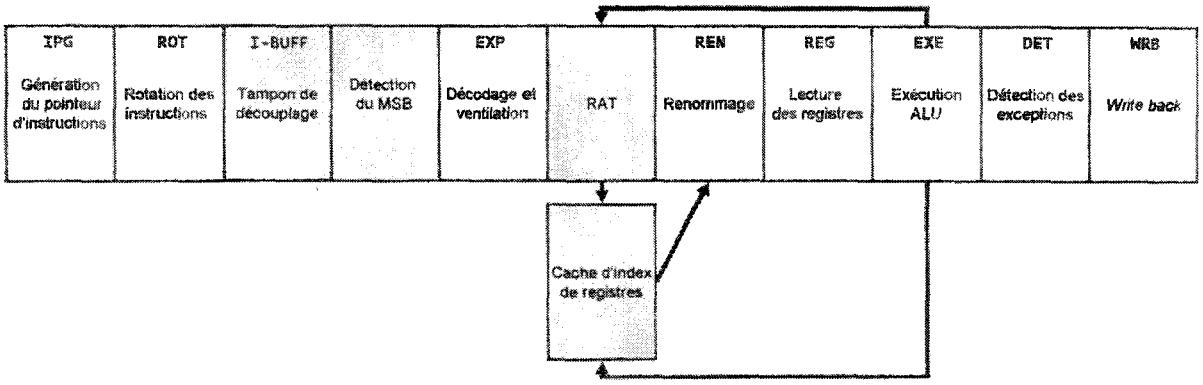

Figure 20 – Organisation simplifiée du pipeline du processeur Itanium 2 amendé pour gérer les accès indexés aux registres. Le tampon de découplage du frontal est représenté en hachuré. Les circuits ajoutés sont grisés.

Une fois que la pile cachée est activée, l'étage `EXP` (*Template decode, Expand and Disperse*) doit vérifier pour chaque instruction si le bit de poids fort (*MSB – Most Significant Bit*) de l'adresse de l'un des registres source est armé (numéro >= 64). Si ce n'est pas le cas, alors l'exécution normale de l'instruction se poursuit. Si un des *MSB* est armé, alors la table d'allocation de registre additionnelle (*RAT*) vérifie si le registre cible est modifié par une instruction en cour d'exécution (dépendance *Read-After-Write*). Pour suivre l'état des registres cibles (prêt / non-prêt) la *RAT* utilise un vecteur de 64 bits. Cette *RAT* peut-être implémentée comme une extension de la *RAT* déjà présente dans le processeur.

Si le bit associé au registre est armé alors la *RAT* injecte l'adresse du nouveau registre dans l'étage `REN`. Une entrée de la *RAT* correspond à six bits de l'adresse physique du registre dans sa pile. Si par contre le statut du registre dans la *RAT* n'est pas prêt, alors sa valeur est propagée dans la *RAT*, ce qui met à jour le bit de statut du registre. A cet instant l'exécution de l'instruction bloquée peut reprendre son cours (Figure 20). Sur la Figure 20, l'étage additionnel responsable du cycle supplémentaire est représenté en gris entre les étages `EXP` et `REN`. Les flèches depuis l'étage `EXE` vers la `RAT` et le cache d'index représentent la mise à jour de l'état des registres (prêt / non-prêt). La flèche reliant le cache d'index et l'étage `REN` représente l'injection de la nouvelle adresse de registre physique (après indexation) dans l'étage `REN`.

En comparaison de l'implémentation initiale et systématique de l'accès indexé aux registres dynamiques, l'implémentation matérielle partielle que nous avons introduit dans cette section présente de nombreux avantages. Mais avant de les aborder, rappelons- en les inconvénients. Nous limitons le nombre des registres à accès indexé à 64. Nous avons besoin d'une *RAT* de 64 entrées utilisant 64 bits chacune, ainsi qu'un vecteur de 64 bits et deux bits indicateurs (un pour l'instruction CPUID et un pour le *MSR*). Enfin, le fonctionnement même de l'accès indexé ajoute un cycle supplémentaire au pipeline principal, rendu nécessaire pour lire le contenu du registre d'index. Du point de vue des avantages de la méthode, nous pouvons signaler les points suivants : simplicité d'implémentation d'une mise en cache d'une pile d'évaluation entière et flottante. Cela permet d'implémenter des machines virtuelles de façon efficace et en limitant les nombre des accès à la mémoire. L'implémentation matérielle proposée est réaliste et applicable à l'architecture des processeurs *Itanium*. La solution proposée est totalement compatible avec les logiciels existants. Enfin, une fois implémentée, elle permet la suppression des instructions de lecture en mémoire associées aux opérations de gestion de la pile et contribue à la réduction de la consommation électrique du processeur.

7. Conclusions

Dans ce chapitre, nous avons présenté une utilisation novatrice de la pile des registres dynamiques des processeurs *Itanium*. Notre objectif étant de combler l'écart qui existe entre les circuits spécialisés conçus pour exécuter des machines à piles, et l'architecture généraliste *EPIC*. Nous avons démontré que la technique de la mise en cache de la pile d'évaluation dans les registres du processeur est une technique efficace pour optimiser les performances d'une machine à pile. Nos tests comparant les performances de notre implémentation de référence d'un système *FORTH* et de sa mouture optimisée montrent des gains de performance moyenne de 7x avec le processeur *Itanium 2*.

En nous fondant sur ces résultats prometteurs obtenus avec notre implémentation de cache de pile et pour simplifier l'écriture d'une machine virtuelle, nous avons proposé un mode d'accès indexé aux registres du processeur. Une première implémentation systématique est proposée. Cependant, son implémentation reste théorique car trop intrusive malgré les choix conservatifs que nous avons faits. En revanche, et en limitant le nombre des registres à accès indexés à 64, nous avons proposé une implémentation matérielle réaliste qui satisfait l'ensemble de nos critères.

8. Références

Bai, P. « A 65nm Logic Technology Featuring 35nm Gate Lengths, Enhanced Channel Strain, 8 Cu Interconnect Layers, Low-k ILD and 0.57μm² SRAM Cell. » *International Electron Device Meeting.* 2004.

Benini, L, et G De Micheli. « Networks on Chip: A New Paradigm for System on Chip Design. » *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition.* 2002.

Dixon, R. D., M. Calle, C. Longway, L. Peterson, et R. Siferd. « The SF1 Real Time Computer. » *Proceedings of the IEEE National Aerospace and Electronics Conference.* Dayton, OH, 1988.

Ertl, Anton, et David Gregg. « Combining stack caching with dynamic superinstructions. » *Virtual Machines and Emulators.* 2004.

Hand, T. « The Harris RTX 2000, Microcontroller. » (Journal of Forth Application and Research) 6, n° 1 (1990).

Hayes, J. R., et S. C. Lee. « The Architecture of FRISC 3: A Summary. » (Rochester Forth Conference Proceedings) 1988.

Hoogerbrugge, Jan, Lex Trum, Jeroen Augusteijn, et Rik Van De Wiel. « A Code Compression System Based on Pipelined Interpreters. » *Software - Practice and Experience.* JohnWiley & Sons, Ltd., 1999.

Imsys. *Imsys IM1101.* 1 Aout 2008. http://www.imsys.se/products/im1101.htm (accès le Aout 1, 2008).

Intel Corporation. Intel Itanium 2 Processors Reference Manual for Software Development and Optimization. Vol. 1, 2, 3. Intel Corporation, 2008.

Koopman, P. Jr. *Stack Computers: the new wave.* Ellis Horwood, 1989.

Koopman, P. « Writable Instruction Set Stack Oriented Computers: The WISC Concept. » (Journal of Forth Application and Research (Rochester Forth Conference Proceedings)) 5, n° 1 (1987).

Michael O'Connor, J., et M. Tremblay. « PicoJava-i: The Java Virtual Machine in Hardware. » 1997.

Moore, Gordon E. « Cramming more components onto integrated circuits. » (Electronics) 38, no. 8 (1965).

Niar, Smail, and Jamel Tayeb. « Adapting EPIC Architecture's Register Stack for Virtual Stack Machines. » *Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD'06).* IEEE, 2006.

Niar, Smail, et Jamel Tayeb. *Les processeurs Itanium: Programmation et optimisation.* Paris: Eyrolles, 2005.

Niar, Smail, et Tayeb Jamel. « Optimizing Intel EPIC/Itanium2 Architecture for Forth. » *22nd EuroForth Conference.* Cambridge, UK, 2006.

Paysan, Bernd. *A Four Stack Processor.* Technische Universität München, 2000.

Peng, Jinzhan, Jinzhan Peng, et Guei-Yuan Lueh. « Code sharing among states for stack-caching interpreter. » (Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators) 2004.

R. Uhlig, R, et al. « Intel Virtualization Technology. » *Computer*, 2005.

Rather, Elisabeth D, D Colburn, et Charles H. Moore. « The Evolution of Forth. » (ACM SIGPLAN) 28, n° 3 (1993).

Schelisiek, K. « MicroCore: an Open-Source, Scalable, Dual-Stack, Hardware Processor Synthesisable VHDL for FPGAs. » *EuroForth.* 2004.

Schlansker, M. S., et B. Ramakrishna Rau. « EPIC: Explicitly Parallel Instruction Computing. » (IEEE Computer Society Press) 33, n° 2 (2000).

Schlansker, M., et al. « Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. » (HPL Technical Report HPL-96-120, Hewlett-Packard Laboratories) 1997.

Schoeberl, Martin. *JOP – a Real-time Java Processor.* Vienna: Institute of Computer Engineering - Vienna University of Technology, Austria, 2005.

Shpeisman, T., G-Y. Lueh, et A-R. Adl-Tabatabai. « Just-In-Time Java Compilation for the Itanium Processor. » *11th International Conference on Parallel Architectures and Compilation Techniques.* 2002.

Smith, J., et R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Elsevier Science & Technology Books, 2005.

Ting, Chen-hanson, et Charles H. Moore. *MuP21--A High Performance MISC Processor.* 1995. http://www.ultratechnology.com/mup21.html (accès le August 2008).

Tullsen, D, S Eggers, et H Levy. « Simultaneous Multithreading: Maximizing On-Chip Parallelism. » *22nd Annual International Symposium on Computer Architecture.* 1995.

X3J14, Technical Committee. *X3.215-1994: Programming Languages – Forth.* American National Standard for Information System, 1994.

Une pile virtuelle pour l'architecture *EPIC*

Résumé

Les technologies .NET et Java connaissent un succès essentiellement fondé sur la richesse et la souplesse de leurs environnements de développement. En contre partie, les considérations de performances sont reléguées au second rang et sont la responsabilité des compilateurs et des équipes d'optimisation des machines virtuelles. Conjointement, de nombreuses recherches débouchent sur la mise au point de circuits spécialisés pour accroître les performances des machines virtuelles, notamment en intégrant une pile d'évaluation matérielle. L'approche que nous décrivons dans ce chapitre est une solution hybride qui, tout en se fondant sur l'architecture standard des processeurs Itanium, *offre un support matériel à la pile d'évaluation.*

1. Introduction

Au chapitre précédent, nous avons proposé d'utiliser les registres de l'architecture *Explicit Parallel Instruction Computer – EPIC* – pour optimiser les performances des machines à piles. Nous avions alors retenu en guise d'exemple le langage *FORTH*. En plus de raisons pratiques liées à sa relative facilité d'implémentation – mais aussi à notre intérêt personnel pour ce système –, nous avons opté en sa faveur car il s'agit de l'archétype de la machine à pile.

Nous avions ainsi pu montrer qu'il était possible de maintenir les piles de l'interpréteur *FORTH* dans les registres physiques du processeur, ce qui se traduit par des gains de performance considérables. Deux implémentations ont été proposées. La première, fondamentalement logicielle, requiert une translation binaire à la volée du code. Afin de réduire l'impact de cette technique sur les performances, nous avons proposé, en guise de seconde implémentation, un amendement matériel au gestionnaire de registres ainsi qu'au jeu d'instructions *EPIC*.

Ces implémentations, bien qu'étant de natures très différentes, peuvent se résumer à la mise à la disposition du logiciel d'une pile matérielle dont le contrôle lui est explicitement échu. Malgré les gains de performance mesurés, le coût de ces solutions reste élevé. En outre, l'approche purement logicielle, appliquée a des langages à piles plus évolués comme le *Microsoft Intermediate Language – MSIL* – ou le langage *Java*, requiert une modification en profondeur du code existant. Quant à l'amendement matériel, sa réalisation en l'état est peu probable car trop intrusive.

Nous avons donc décidé, tout en conservant notre objectif initial de mettre à la disposition du programme une pile matérielle maintenue à même les registres du processeur, d'aborder le problème sous un angle différent. En effet, notre nouvelle approche – que nous allons détailler au cours de ce chapitre – retire au logiciel le contrôle de la pile et le transfert au

matériel. En d'autres termes, la gestion explicite de la pile devient implicite du point de vue du programmeur.

Les avantages de cette approche sont nombreux comme nous le verrons. Parmi ceux-ci, nous citerons dès à présent une implémentation matérielle très peu intrusive – et donc plus réaliste –, une conservation de la quasi-totalité des codes existants et surtout l'ouverture sur des langages d'un usage beaucoup plus répandus que le *FORTH*, comme par exemple le *MSIL* ou *Java*. L'utilisation de ces langages connaît un succès certain auprès des développeurs et des entreprises. Pour s'en convaincre, il nous suffit de noter la décision prise par *Microsoft Corporation* de recentrer son offre aux développeurs autour de *.NET*. et de placer *.NET* au cœur même de sa gamme de systèmes d'exploitation. Malheureusement, il n'existe pas de données publiques donnant le taux d'adoption de *.NET* – ou de *Java* d'ailleurs – auprès des développeurs.

Nous avons ensuite cherché à lever la contrainte d'accès séquentiel à la pile pour pouvoir profiter de architectures *Very Long Instruction Word – VLIW* – sous-jacente. Nous proposons ainsi une méthode qui vise à augmenter le parallélisme des instructions ou *Instruction-Level Parallelism – ILP* – du code généré.

Nous montrerons également que l'utilisation de cette pile matérielle permet la simplification des compilateurs *Just in Time – JIT* – notamment en abrégeant la phase d'allocation des registres. Elle permet aussi la mise au point de traducteurs d'un binaire *MSIL* vers un binaire *Itanium* en une simple passe et donc à très faible coût. Nous présenterons à cette occasion quelques techniques d'optimisation rudimentaires qui améliorent la qualité du code généré pour les critères de qualité que nous avons retenus.

Dans la suite de ce chapitre, nous commencerons par présenter une vue d'ensemble du problème au travers de quelques travaux connexes à notre étude. Nous rappellerons aussi quelques notions fondamentales relatives à *.NET*. Alors seulement, nous présentons notre pile matérielle fondée sur les piles de registres du processeur *Itanium 2*. Nous proposerons dans cette section une implémentation matérielle très peu intrusive fondée sur l'architecture *EPIC*. Nous montrerons ensuite comment les instructions *EPIC* peuvent utiliser la pile matérielle. Nous conclurons enfin en présentant nos résultats expérimentaux et notre conclusion.

2. Travaux connexes

Le recours au matériel pour supporter l'exécution de machines virtuelles est une technique communément usitée. Elle est généralement choisie pour améliorer le niveau de performance par rapport à un interpréteur logiciel. D'une façon plus anecdotique, elle est retenue pour la flexibilité qu'elle apporte, notamment dans les environnements embarqués.

De ce point de vue, les environnements de développement *Java* et *.NET* font l'objet de nombreuses implémentations à la fois industrielles et académiques. De par son adoption précoce, *Java* dispose d'une bonne longueur d'avance sur son compétiteur qui n'a été introduit pour sa part que plus récemment (1995 contre 2002). Ainsi, parmi les nombreux exemples de processeurs dédiés à l'exécution de *Java*, nous pouvons citer le *PicoJava* de *Sun*

Microelectronics, l'*Imsys Cjips*, l'*aJile* ou bien le *JOP* (Sun Microsystems, 1997) (Imsys, 2008) (aJile Systems, 2008) (Schoeberl, 2005) (Figure 21). Ces processeurs sont de bons exemples de circuits spécialisés et intègrent tous un gestionnaire de pile d'évaluation dédié. Par exemple, le *DTC Lightfoot Java Processor Core* n'intègre pas moins à même la puce qu'une pile 32 bits et d'une profondeur de huit éléments. Elle dispose d'un pointeur de pile directement câblé à l'*ALU* et d'un circuit de gestion de débordement de pile (*spill / fill*) (DTC, 2001). La série des processeurs *Application Assist Processors* de la famille *zSeries* d'*IBM* (*zAAP*) propose un support asynchrone pour l'exécution de code intermédiaire *Java*. La sélection des méthodes qui bénéficient de l'accélération matérielle se fait par la machine virtuelle *Java* de *WebSphere* et par le processeur central (Rogers, et al., 2004). Rappelons enfin, comme nous l'avons vu au chapitre précédent, que le support matériel de la pile n'est pas l'apanage de *.NET* ou de *Java*, mais que des langages comme le *FORTH* ou même le C peuvent en bénéficier.

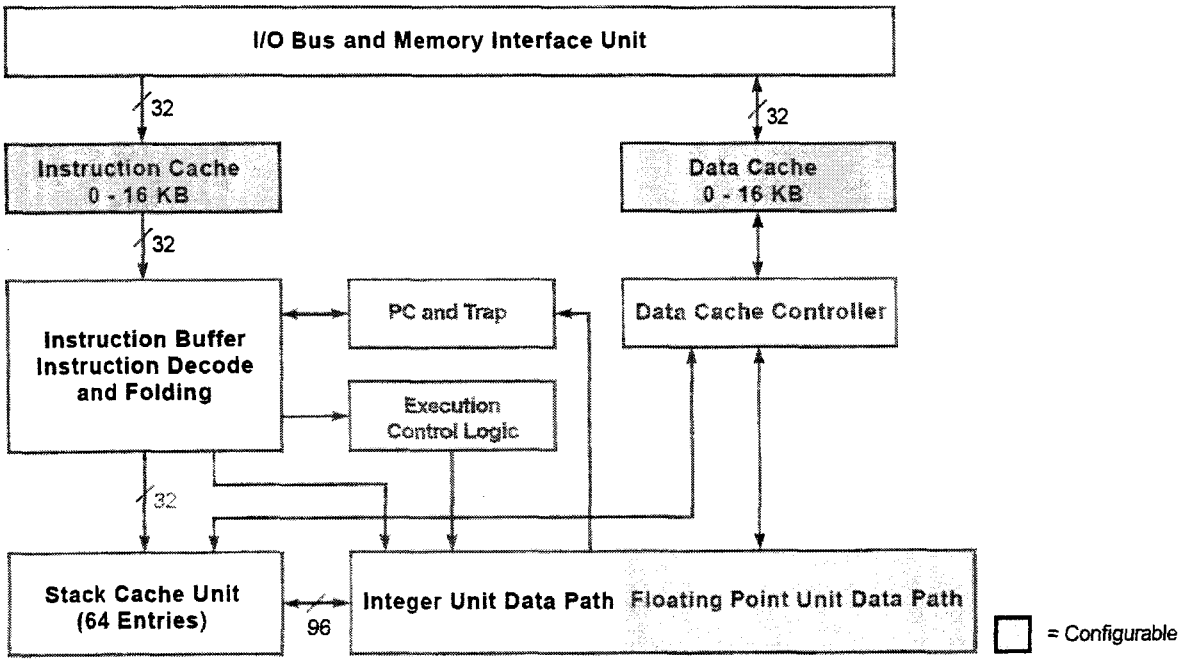

Figure 21 –Diagramme du processeur PicoJava I. La cache de pile dispose de 64 entrées directement accessibles aux unités de calcul. Souce : Sun Microelectronics.

Il est intéressant de signaler qu'il existe sur l'extrémité embarquée du spectre des solutions existantes, un fort intérêt naissant pour l'environnement de développement *.NET*. Cet engouement s'explique en particulier par la flexibilité et la facilité de développement qu'offre *.NET* aux chercheurs et aux ingénieurs ; tout en assurant des performances supérieures à l'interprétation logicielle pure. Ainsi, de nombreuses machines virtuelles *Java* et *.NET* ont été implémentées sous forme de *FPGA* pour ces raisons (O'Connor, et al., 1997) (Imsys, 2008) (aJile Systems, 2008) (Schoeberl, 2005) (Srinivasan.T, et al., 2005) (Kumar, 2002) (Ragozin, et al., 2006). Parmi ces implémentations, l'une s'intéresse à la version allégée de *.NET* (*.NET Micro Framework*) qui permet entre autre d'amorcer directement la machine virtuelle *.NET* et disposer

d'un environnement, certes limité, mais qui conserve l'essence même de l'environnement de *Microsoft*. Signalons toutefois que les cœurs *MicroBlase* de l'implémentation effectuent une interprétation du code *MSIL* et ne disposent pas de compilateur *JIT*. Cette tendance est générale dans ce segment, et se voit compensée par la possibilité de compilation croisée par anticipation (*AOT*).

Le *CIL Hardware Processeur – CILHP* – [(Ragozin, et al., 2006) – Figure 22] est une des première – et unique à notre connaissance – implémentation d'un processeur .NET couvrant l'ensemble des fonctionnalités le la *CLR .NET*. Il utilise le *FPGA LX25 Virtex-4* qui intègre également un *DSP*. Ce dernier est utilisé comme additionneur rapide d'adresses pour le décodeur *CIL* et les étages successifs du pipeline. Le *CILHP* conserve les quatre premiers niveaux de la pile d'évaluation dans les registres du *DSP* (A0-A7). Puisque les instructions *CIL* n'accèdent qu'à trois niveaux de piles au maximum, le contrôleur de bloc de piles et le générateur d'adresses (*Stack Block Transfer Controller and Address Generator* ou *SBACAG*) peuventsuperposer l'exécution d'une instruction CIL et charger / décharger (*spill / fill*) quatre registres de 32-bits. L'instruction en cours d'exécution accédant à une banque de registres différente. La mémoire interne du *FPGA*, qui est utilisée pour implémenter la cache de pile, dispose de deux ports pour l'écriture et la lecture simultanée. Cela permet au *SBCAG* d'effectuer des transferts en mémoire et les opérations sur la pile en parallèle. En outre, le *SBCAG* pré-charge les données dans la pile avant que celles-ci ne soient utilisées par l'instruction *CIL* suivante, ce qui permet selon les auteurs de limiter les conflits en lecture / écriture dans les registres du *DSP*. Enfin, signalons que les opérations utilisant trois registres sont micro-codées.

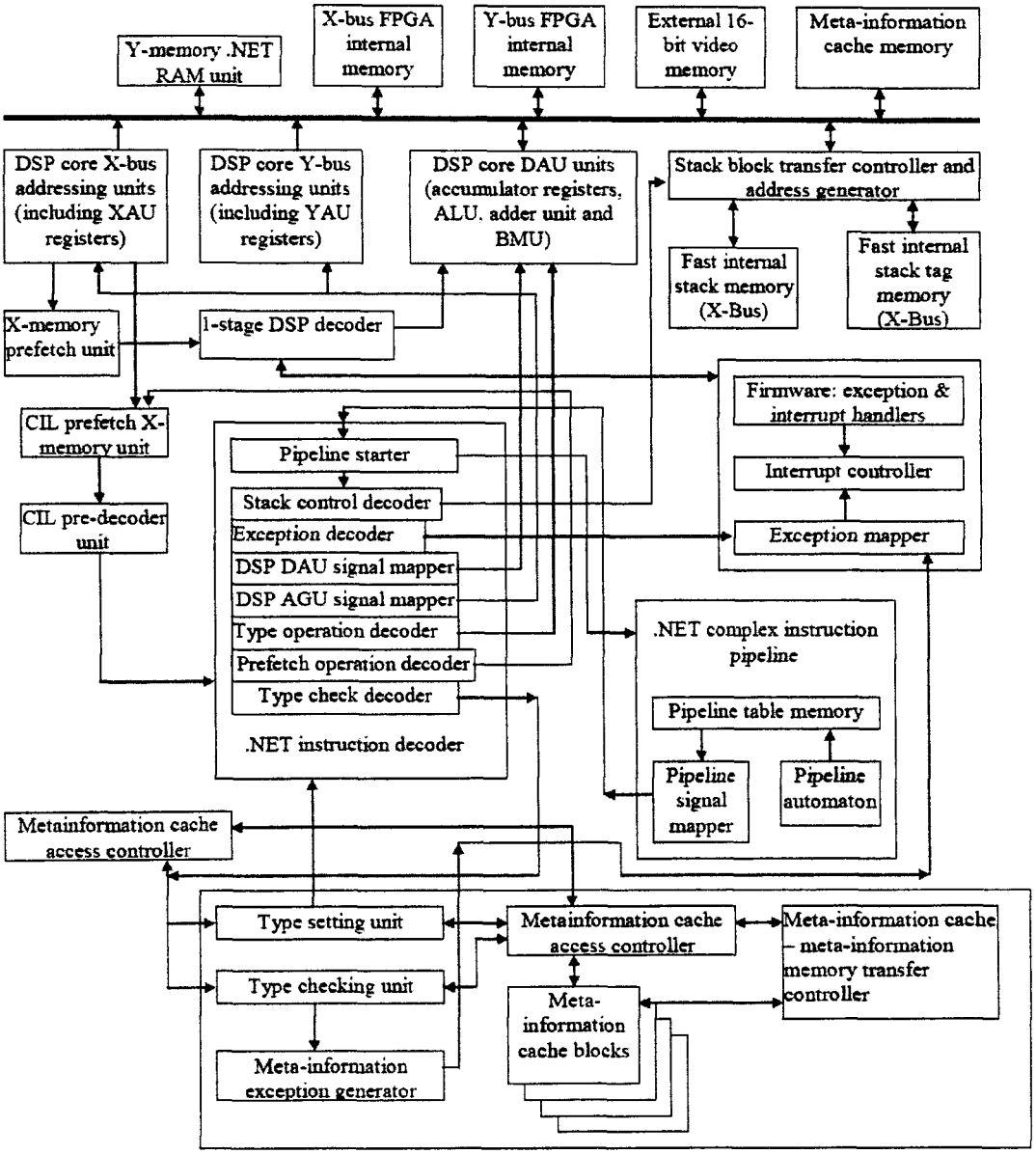

Figure 22 –Diagramme du CIL Hardware Processor. Cette implémentation à base de FPGA est à notre connaissance l'unique implémentation d'un processeur .NET couvrant l'ensemble des fonctionnalités de la CLR. Source : les auteurs.

En comparaison de ces axes de recherches et de développement qui donnent la primeur à la conception de circuits spécialisés, la solution que nous proposons dans ce chapitre présente une approche intermédiaire et originale en se fondant, premièrement, sur l'architecture d'un processeur généraliste et en proposant, deuxièmement, un support matériel pour la pile d'évaluation des machines virtuelles modernes. La Table 1 donne un résumé des avantages et inconvénients de chacune des méthodes.

Implémentation	*Avantages*	*Inconvénients*
Processeur généraliste	Faible coût d'achat, large diffusion.	Performance en fonction de l'implémentation de la machine virtuelle.
Processeur spécialisé	Excellente performance.	Coût d'achat élevé, faible diffusion. Développement d'une machine virtuelle spécifique.
Interpréteur	Performances limitées.	Excellente portabilité.
Interpréteur + JIT	Bonnes performances.	Portabilité limitée.
Cache de pile logiciel	Gains de performance.	Développement d'une machine virtuelle spécifique.
Cache de pile matériel	Gains de performance.	Développement d'une machine virtuelle spécifique.

Table 6 – Avantages et inconvénients des principaux supports et des techniques d'exécution d'optimisation de machines virtuelles.

3. Terminologie *.NET*

La technologie *.NET* a été introduite par *Microsoft Corporation* en guise d'environnement de développement de prochaine génération pour la famille des systèmes d'exploitation *Windows*. Pour le programmeur, l'environnement de développement, ou *Framework*, offre avant tout une librairie – voulue exhaustive – qui fournit les fonctionnalités élémentaires requises au développement rapide d'applications standards. En plus de cette *Base Class Library*, ou *BCL*, l'environnement dispose d'une machine virtuelle connue sous le nom de *Common Language Runtime – CLR*. Cette dernière assure l'exécution du code intermédiaire qui n'est pas directement exécutable par le processeur – ou *bytecode* – et qui est généré en amont par les compilateurs *.NET* à partir de codes source écrits en langages de programmation compatibles avec l'environnement. Parmi les plus connus, citons les langages *Visual Basic .NET* et *C#*.

Les compilateurs *.NET* génèrent donc du code intermédiaire ou *Common Intermediate Language – CIL* – également connu à son origine sous le nom de *MSIL – Microsoft Intermediate Language* (nous utiliserons d'ailleurs indifféremment les deux noms dans ce chapitre). Le *CIL* est le langage machine d'un processeur virtuel dont le jeu d'instructions opère sur une pile et manipule des métadonnées telles qu'elles sont définies par la *Common Type Specification*. La *Common Type Specification* étant elle-même un sous-ensemble du *Common Type System* qui défini intégralement la façon dont les types de données sont déclarés, utilisés et maintenus par le *CLR*. Le *Common Type System* est d'autant plus important pour *.NET* que l'environnement permet l'interopérabilité entre tous les langages *.NET* existants et à venir. Grâce à cette connaissance, le *CLR* peut dès lors offrir des services évolués qui font le succès de ce type d'environnements de développements – également décrits comme *managed*. Parmi les services

les plus en vogues, nous pouvons citer le ramasse-miettes ou *garbage collection*, la sécurité ou la réflexion.

Pour qu'un code applicatif puisse être exécuté par le processeur, le *CIL* doit être traduit en binaire natif pour la plate-forme d'exécution. Cette opération peut s'effectuer suivant différents procédés, et se pratique généralement à la volée. Les méthodes les plus courantes sont l'interprétation ou la compilation à la volée ou *JIT*. Le code exécutable peut aussi être généré par une compilation à l'avance – ou *Ahead of Time* –, par exemple avec le compilateur *NGen* de *Microsoft Corporation*. La Figure 23 schématise ce processus.

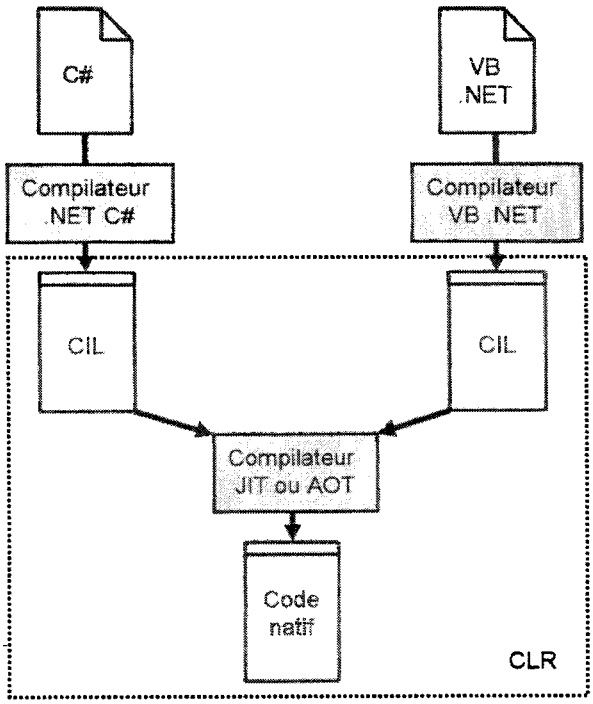

Figure 23 – Etapes nécessaires à la génération de code machine natif sous .NET depuis un code source rédigé dans différents langages (C# et VB .NET par exemple). L'interprétation n'est pas représentée ici. Le bloc en pointillés montre les opérations sous le contrôle du CLR à l'exécution (ou runtime). La compilation AOT prend place hors contrôle du CLR.

Pendant l'exécution des applications, le *CLR* emploit trois mémoires locales et une mémoire externe ((Gough, 2002) – Figure 24). Ces mémoires sont listées ci-après et peuvent contenir n'importe quel type de données défini par le *CTS*. C'est d'ailleurs en ce sens que le *CIL* diffère fondamentalement du *FORTH* ou bien du *P-Code* du langage de programmation *Pascal* par exemple. Les instructions du processeur cible virtuel manipulent des types de données évoluédont certains sont des objets :

- La pile d'évaluation (*Evaluation Stack*). La pile d'évaluation est l'équivalent de la pile de même nom que nous avons présenté au chapitre précédent (*FORTH*). Il s'agit de la pile utilisée par les instructions *CIL* pour y lire leurs arguments et y écrire leur résultat. Le nombre des entrées de la pile utilisée est donné par la directive *.maxstack* de la méthode.

- La table des arguments (*Argument Table*). Le nombre des entrées de la table des arguments est déterminé au point d'appel en utilisant la signature de la méthode. Cette détermination se fait lors de l'appel pour pouvoir prendre en compte les méthodes *vararg* (nombre d'augments variable).

- La table des variables locales (*Local Variables Table*). Le nombre des entrées de la table des variables locales est déterminé à partir de la signature de la méthode contenue dans l'entête (*method header*).

- Les champs de méthode (*Community of Fields*). Il s'agit d'une zone de la mémoire externe de la *CLR* qui est utilisée par la méthode.

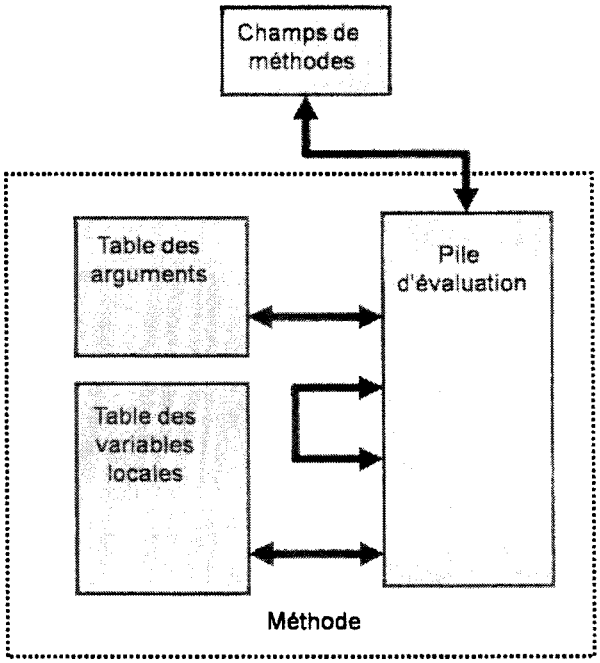

Figure 24 - Mémoires utilisés par les méthodes. Source Microsoft Corporation. Les flèches représentent les mouvements de données entre les différentes mémoires utilisées par les méthodes.

Notre étude menée au chapitre précédent avec le langage *FORTH* nous a montré qu'il est possible de cacher la pile d'évaluation d'une machine à pile à même les registres du processeur *Itanium 2* pour obtenir des gains de performance significatifs. En plus de la méthode logicielle que nous avons utilisée – et qui requiert d'apporter des modifications importantes à la machine virtuelle –, nous avons introduit un support matériel pour optimiser le cache de pile d'évaluation.

Cependant, comme dans le cas de la solution purement logicielle, ce support matériel reste sous le contrôle du logiciel. Ce qui implique que la machine virtuelle et / ou le code applicatif doit être modifié pour en tirer profit.

Dès lors, nous pouvons nous poser la question de savoir s'il est possible d'appliquer la même technique à *.NET*. En principe, le cache de pile doit profiter à la machine virtuelle de *.NET* puisque le langage intermédiaire est l'assembleur d'une machine à pile. Cependant, des différences fondamentales existent entre les deux machines à pile. En effet, et comme nous le détaillerons plus tard, la différence essentielle entre la pile d'évaluation *FORTH* et *.NET* est le typage dynamique fort de cette dernière. Enfin, là où la pile *FORTH* est directement visible des programmes, la pile d'évaluation de *.NET* est invisible et échappe au contrôle du programmeur. En conséquence de quoi, nous nous sommes directement focalisés dans ce chapitre sur la possibilité d'offrir un support matériel pour un cache de pile qui satisfasse les besoins spécifiques introduits par *.NET*. et qui ne dépende pas du logiciel.

4. La pile matérielle

Voyons à présent comment nous proposons d'implémenter une pile matérielle dans le but d'optimiser les performances du *CLR*. Le concept fondamental pouvant se résumer à conserver les données manipulées par le programme au plus près du cœur du processeur. Nous avons donc décidé de conserver la pile d'évaluation, la table des arguments ainsi que la table des variables locales dans les registres du processeur. La table des arguments est gérée par le mécanisme d'appel du *Register Stack Engine – RSE*. Les paramètres de l'instruction *EPIC* `alloca` utilisée pour invoquer le *RSE*, généralement au point d'entrée des fonctions, sont directement déduits de l'arité des méthodes, elle-même connue *via* la directive `.maxstack`. Le Listing 8 montre un exemple d'utilisation de cette directive sur une méthode désassemblée – partiellement représentée pour limiter la taille du listing.

```
.method assembly static void
modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
f8(uint8* data, int32 length, valuetype _kasumi_data* pk) cil managed {
      .vtentry 82 : 1
      // Code size       273 (0x111)
      .maxstack  4
      .locals (int32 V_0,  int32 V_1, … )
…
} // end of method 'Global Functions'::f8
```

Listing 8 – Exemple de méthode CIL et sa directive .maxstack définissant le nombre de niveaux de pile d'évaluation requis par la méthode.

Le concepteur de compilateur et de machine virtuelle a essentiellement le choix entre deux options pour maintenir la table des variables locales ainsi que celle des arguments. Soit il le fait en utilisant le *RSE* (96 registres entiers supérieurs entiers et sans le *RSE* pour les flottants),

soit il fait en maintenant ses tables dans les 32 registres bas des registres généraux (*general purpose – GP*) et flottants (*floating-point – FP*) du processeur. Il s'agit essentiellement d'un choix d'implémentation. Cependant, dans le cadre de notre étude, nous avons choisi la méthode de la gestion directe qui consiste à charger les valeurs depuis les tables dans les 32 registres bas pour maximiser les performances en limitant le recours au *RSE*. En effet, une sollicitation trop forte de ce composant peut entrainer un décrochage – ou *stall* – de la partie dorsale (ou *backend*) du processeur, ce qui se traduit alors irrémédiablement par une perte de performance. Il est impossible de prédire l'impact de l'activité du *RSE* sur les applications en général. Cependant, il est possible d'en évaluer l'impact durant l'exécution en comptant les cycles de décrochage du pipeline. Le comptage des cycles de décrochage dûs à l'activité du *RSE* se fait en utilisant les compteurs architecturaux `BE_RSE_Bubble` et ses sous-compteurs. La Table 7 donne la liste des sous événements et leurs causes. Le nombre de cycles de décrochage est variable, mais nous pouvons compter une moyenne de 7 cycles (avec des pics de 14+ cycles sur l'accès à certains registres architecturaux tels que `ar.rsc` ou `ar.bspstore`). Lors de l'étude de l'optimisation de code OLTP, *Hoflehner* montre que les optimisations *RSE* du compilateur atteignent 2% lorsqu'au moins 10 registres sont sauvegardés sur la pile avant un appel de fonction (Hoflehner, et al., 2004).

Extention	*PMC.umask*	*Cause*
ALL	*bx000*	*RSE (toutes causes confondues)*
BANK_SWITCH	*bx001*	*Bank Switching*
AR_DEP	*bx010*	*Dépendance sur un registre AR*
OVERFLOW	*bx011*	*Spill.*
UNDERFLOW	*bx100*	*Fill.*
LOADRS	*bx101*	*Calcul de Loadrs*

Table 7 – Liste et signification des sous-compteurs de BE_RSE_Bubble.

a. La pile typée dynamique

La spécificité de la pile d'évaluation de la machine virtuelle *CLR* est sa nature fortement typée. Qui plus est, le typage est dynamique. En d'autres termes, le *CLR* n'utilise pas par exemple deux piles disjointes pour les opérations sur les nombres entiers et flottants comme pourrait le faire un système *FORTH*. Ainsi, en fonction de l'étape de l'évaluation en cours et du type des opérandes déjà présents dans la pile, le type de son sommet change. Par exemple, l'addition d'un nombre flottant et d'un nombre entier situés aux deux premiers niveaux de la pile produit un nombre flottant au sommet de celle-ci. La promotion de type étant édictée par le *CTS*. Pour effectuer cette même opération en *FORTH*, le programmeur doit transférer l'argument entier vers la pile des nombres flottants – pour peu que celle-ci existe dans l'implémentation utilisée. Si tel n'est pas le cas, alors le programmeur / l'interpréteur doit convertir le nombre entier au sommet de la pile en sa représentation flottante, ce qui requiert d'ailleurs souvent plusieurs niveaux de pile. Enfin, l'addition se fait en invoquant `F+`. Ce dernier correspond au mot

FORTH pour effectuer l'addition de deux nombres flottants – cet exemple a été vu au chapitre. Par la suite, c'est au programmeur qu'échoit la tâche d'utiliser et d'interpréter correctement le résultat de l'addition stocké dans la pile. Bien sûr avec *CIL*, le programmeur n'a aucun contrôle, ni même d'ailleurs de visibilité sur la pile d'évaluation, et c'est donc au compilateur de la gérer de façon appropriée dans les registres du processeur cible. Le LISTING 9 et la Figure 25 montrent symboliquement l'usage et le fonctionnement de la pile d'évaluation. Notre solution devra donc impérativement intégrer ce niveau d'abstraction pour effectuer implicitement la sélection du bon type de registre.

```
01: IL_0000:  ldarg.0      // empiler le premier argument
02: IL_0001:  ldarg.1      // empiler le second argument
03: IL_0002:  add          // dépiler les deux arguments,
                           // les additionner puis empiler le résultat
04: IL_0003:  ldarg.2      // empiler le troisième argument
05: IL_0004:  add          // l'ajouter à la somme déjà dans la pile
06: IL_0005:  ldc.r8 0.5   // empiler la contante 0.5
07: IL_000e:  mul          // la multiplier par la somme déjà
                           // dans la pile
08: IL_000f:  stloc.0      // stocker le résultat
```

Listing 10 – Exemple de code CIL utilisant la pile d'évaluation.

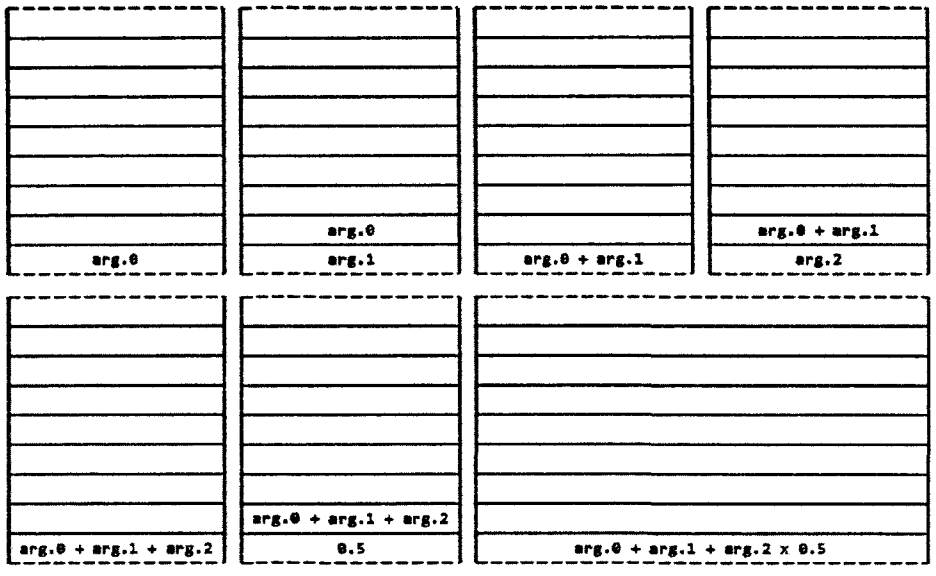

Figure 25 – représentation symbolique de la pile d'évaluation au cours de l'exécution du code du Listing 11. Le résultat de l'exécution des instructions 1 à 7 est figuré.

Remarquons que le compilateur, pour assurer la conformité avec le *CTS*, devra utiliser judicieusement les instructions de conversion de types.

Types de données MSIL	*Registres associés*
int32	GP (64-bit)
Format d'entier natif (native int)	GP (64-bit)
int64	GP (64-bit)
float32 *(80-bit floating-point)*	FP (82-bit)
float64 *(80-bit floating-point)*	FP (82-bit)
& *(managed pointer)*	GP (64-bit)
ObjectRef *(instance pointer to an object)*	GP (64-bit)

Table 8 – Liste des types manipulés par le CIL et leur association avec les registres physiques du processeur.

Notre implémentation matérielle de la pile d'évaluation du *CLR* doit être capable d'assurer automatiquement la manipulation des données en fonction de leurs types. Ce ne sera donc plus le rôle du compilateur d'assurer la bonne association entre les données et les registres, mais bien celui du matériel. Un compilateur passe en moyenne 6% du temps de compilation dans sa phase d'allocation des registres (avec des pics pouvant atteindre 20%).

Pour ce faire, nous commençons par introduire la notion de pile virtuelle qui est essentiellement une abstraction d'un sous-ensemble des registres physiques du processeur. Ce sous-ensemble est utilisé pour implémenter la pile d'évaluation matérielle. Rappelons une fois encore que l'usage de cette pile matérielle ne se limite pas au seul *CLR*, mais que la machine virtuelle *Java* ou le système *FORTH* peuvent aussi en profiter.

L'utilisation de la pile virtuelle par les instructions *EPIC* est sous le contrôle du compilateur ou traducteur binaire. Celui-ci utilise un encodage particulier du code d'opération de l'instruction. Cet encodage sera ensuite détecté par la logique de contrôle de la pile virtuelle lors de l'exécution. Si pour une instruction, le compilateur ne souhaite pas faire usage de la pile virtuelle, alors seul l'encodage du code opératoire change. Nous verrons ce mécanisme en détails prochainement (Figure 26).

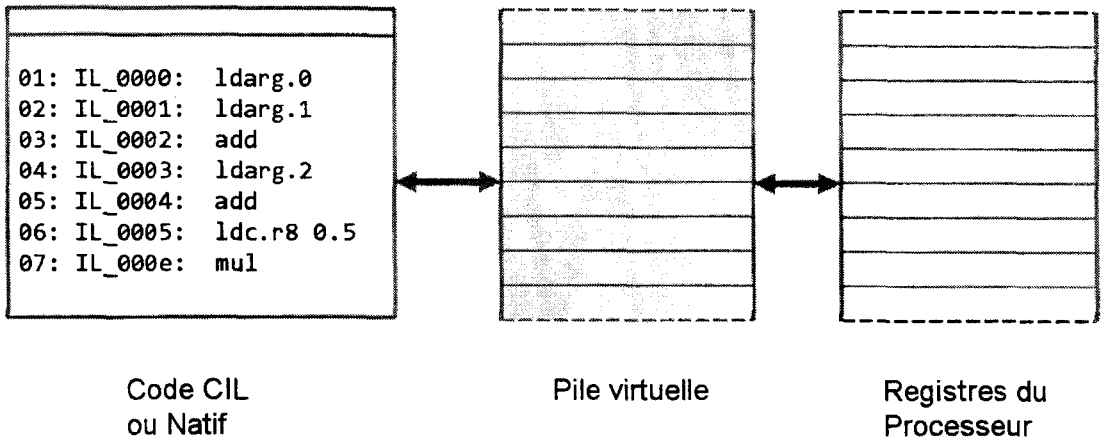

Figure 26 – La pile virtuelle se place entre le code CIL ou natif et les registres du processeur. Pour les instructions y accédant, seule cette pile virtuelle est visible.

Précisons aussi dès à présent que les niveaux de la pile virtuelle ne stockent pas les données qui y sont écrites par les instructions et que ce sont bien les registres physiques du processeur qui s'en chargent. En revanche, les niveaux de la pile virtuelle mémorisent, entres autres, les adresses des registres physiques auxquels ils sont associés à un instant donné. Cette association sera décrite en détails plus tard. Pour le moment, elle peut-être vue comme un index ou un déplacement (*offset*).

Pratiquement, nous réservons la partie haute des registres physiques généraux et flottants du processeur pour mémoriser les données stockées dans la pile virtuelle. Le nombre des registres physiques ainsi réservés est variable, mais il est limité à 64 (Figure 28). Ainsi, la pile virtuelle aura au maximum 64 niveaux, ce qui est plus que suffisant pour les codes *.NET* ou *Java* (Figure 27). Cette figure montre que pour les trois benchmarks considérés, le nombre de niveaux de pile consommé ne dépasse pas dix. Rappelons que chaque méthode .NET doit préciser dans son code le nombre de niveau de pile utilisé. C'est cette information que nous représentons ici. Pour le *FORTH*, 64 niveaux de pile est confortable.

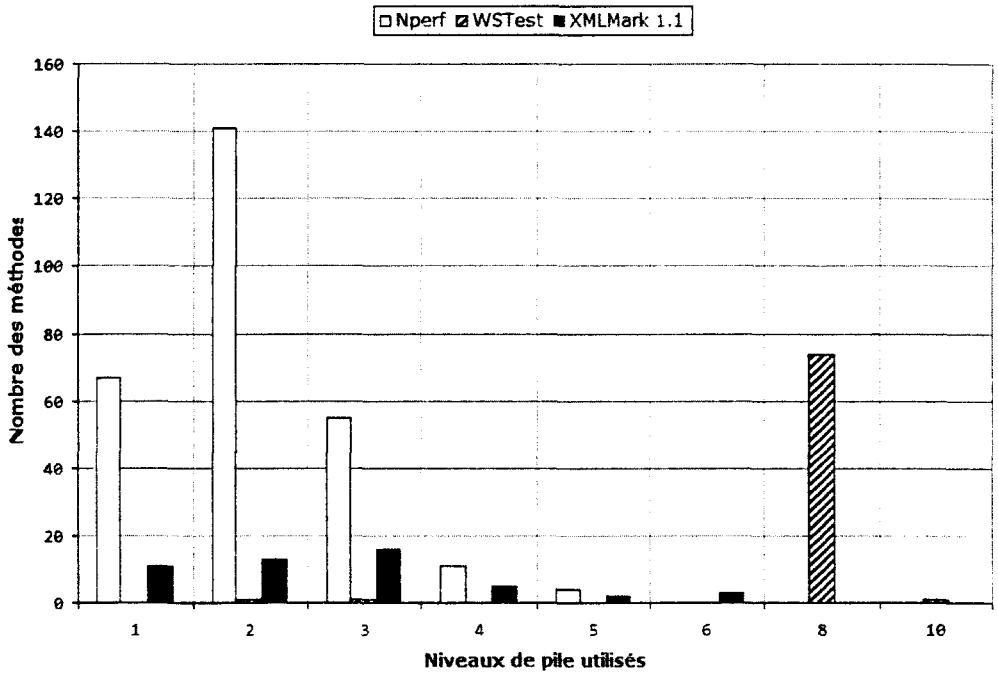

Figure 27 – Distribution du nombre des niveaux de pile d'évaluation utilisés par les méthodes CIL de trois benchmarks .NET (NPErf, WSTest et XMLMark 1.1).

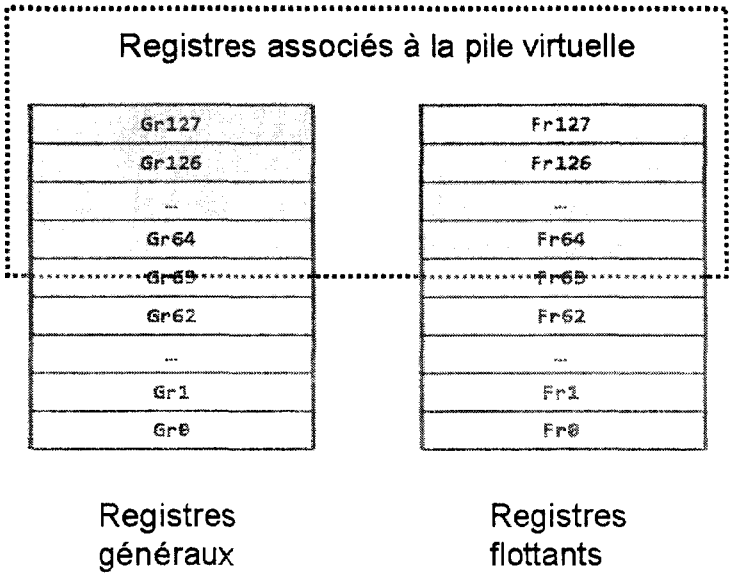

Figure 28 – Sous-ensemble des registres physiques réservés pour l'implémentation de la pile virtuelle.

La pile virtuelle intègre une logique qui traduit les index mémorisés dans ses niveaux en adresses de registres physiques du processeur. La Figure 29 montre de façon symbolique cette traduction des index que nous détaillerons ci-après.

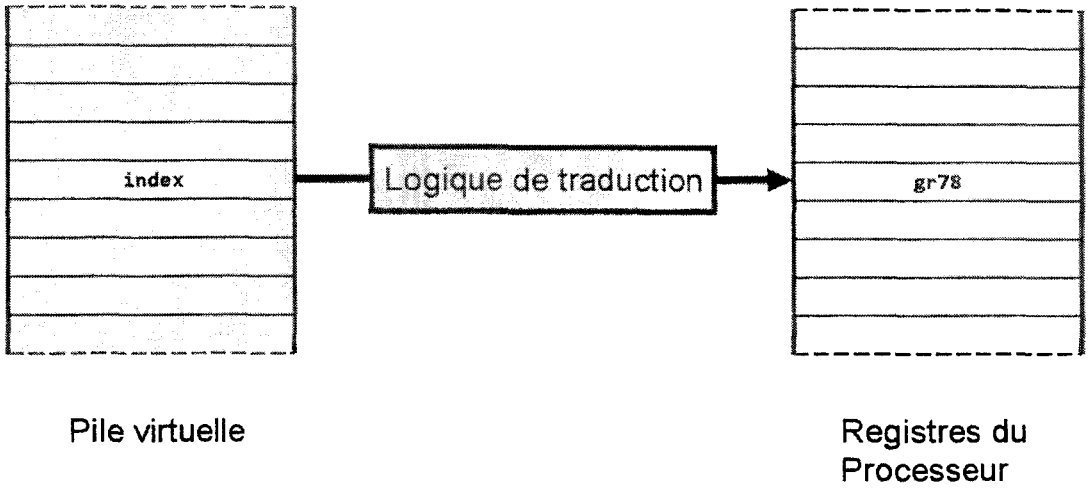

Figure 29 – Entre la pile virtuelle à gauche et les registres physiques associés à cette même pile à droite, la logique de traduction convertie des index en adresses physiques. Seuls les registres généraux sont représentés sur cette figure.

C'est cette même logique qui assure le typage dynamique voulu pour la pile virtuelle. Nous détaillerons comment cela est fait lorsque nous étudierons les primitives implicites d'accès à la pile virtuelle que sont `push` et `pop`. Par implicite, nous entendons qu'il n'y a pas d'instruction `push` et `pop`, mais que ces opérations sont réalisées automatiquement et de façon transparente lors de l'utilisation de la pile virtuelle.

Cette pile virtuelle – qui est intégrée à l'architecture – est constituée d'une table à 64 entrées et d'un registre de sommet de pile. Ce registre est de six bits et se nomme `vtos` – pour *virtual top of the stack*. Chaque entrée de la table est encodée sur huit bits. Le type de chaque entrée peut-être soit entier ou flottant. Il s'agit ici du type codant l'association avec un registre général ou flottant. Ce type est codé par le champ `t` – comme *type*. Ce champ `t` est encodé sur les deux bits de poids fort de l'entrée. La Table 9 donne le détail de l'encodage du type. Bien qu'un seul bit soit suffisant pour faire le distinguo entre les registres généraux et flottants, nous avons pris en compte la possibilité future pour la pile virtuelle de gérer des données stockées en mémoire centrale (code `10`). Le code `11` est réservé et génère quant-à lui une exception s'il est détecté.

T (2 bits)		*Types*
0	0	Entier
0	1	Flottant
1	0	En mémoire

1	1	Exception

Table 9 – Encodage du type des entrées par le champ t des entrées de la table décrivant la pile virtuelle.

Les six bits résiduels de l'entrée forment le champ a – comme adresse (l'index introduit précédemment). Les six bits de l'index codent un déplacement par rapport à l'adresse du premier registre physique dévolu à la pile virtuelle (Figure 30).

Figure 30 – Structure d'une entrée de la pile virtuelle.

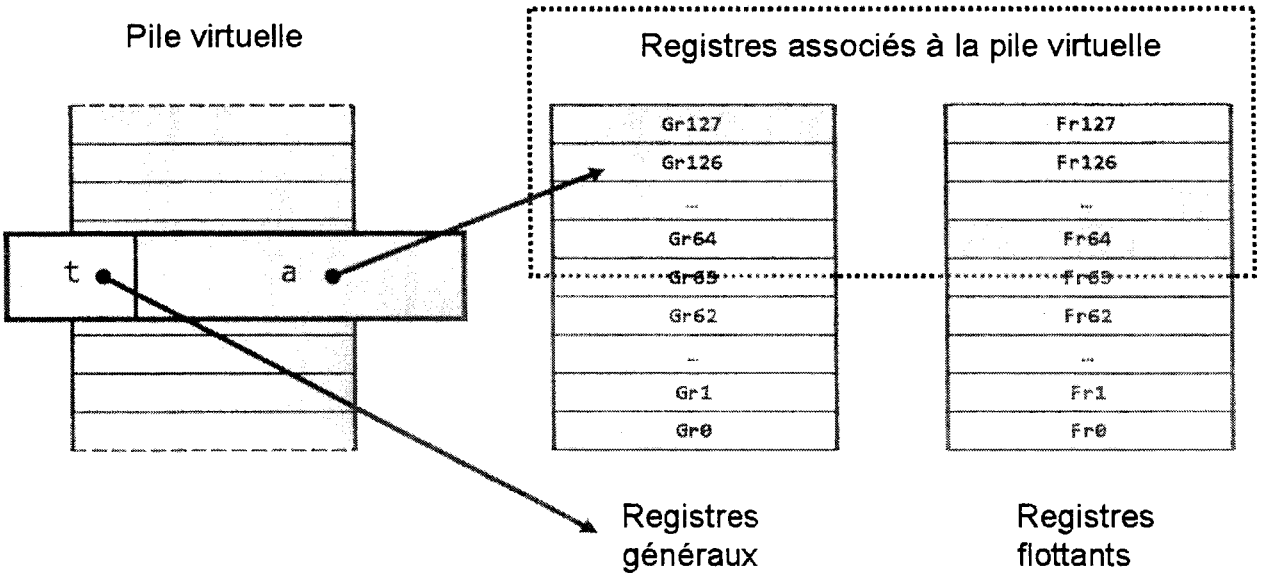

Figure 31 – Relations entre les champs a et t des entrées de la pile virtuelle et les registres physiques associés. La valeur du champ a correspond à un déplacement par rapport aux registres Gr64 et Fr64. Une des entrées a été agrandie sur la figure.

Pour compléter l'implémentation matérielle de la pile virtuelle, nous utilisons deux registres additionnels de six bits chacun. Ils pointent sur le sommet des piles de registres généraux et flottants. Nous entendons par ce terme les 64 registres contigus et hauts du processeur (`Gr64-127` et `Fr64-127`). Ces registres pointeurs de sommet sont nommés respectivement `gtos` et `ftos`. La Figure 32 donne une représentation symbolique de cette organisation. Sur la gauche de la figure se trouve la pile virtuelle avec son pointeur de pile `vtos`.

vtos pointe sur l'entrée où le prochain élément sera empilé. Sur la partie droite de la figure, les deux blocs côte-à-côte représentent les registres physiques du processeur associés à la pile virtuelle. Le bloc de gauche (au centre de la figure) correspond aux registres généraux. Celui de droite correspond à celui des registres flottants. Sous chaque bloc est représenté son pointeur de pile dédié, respectivement gtos et ftos. Ceux-ci fonctionnent à l'identique de vtos.

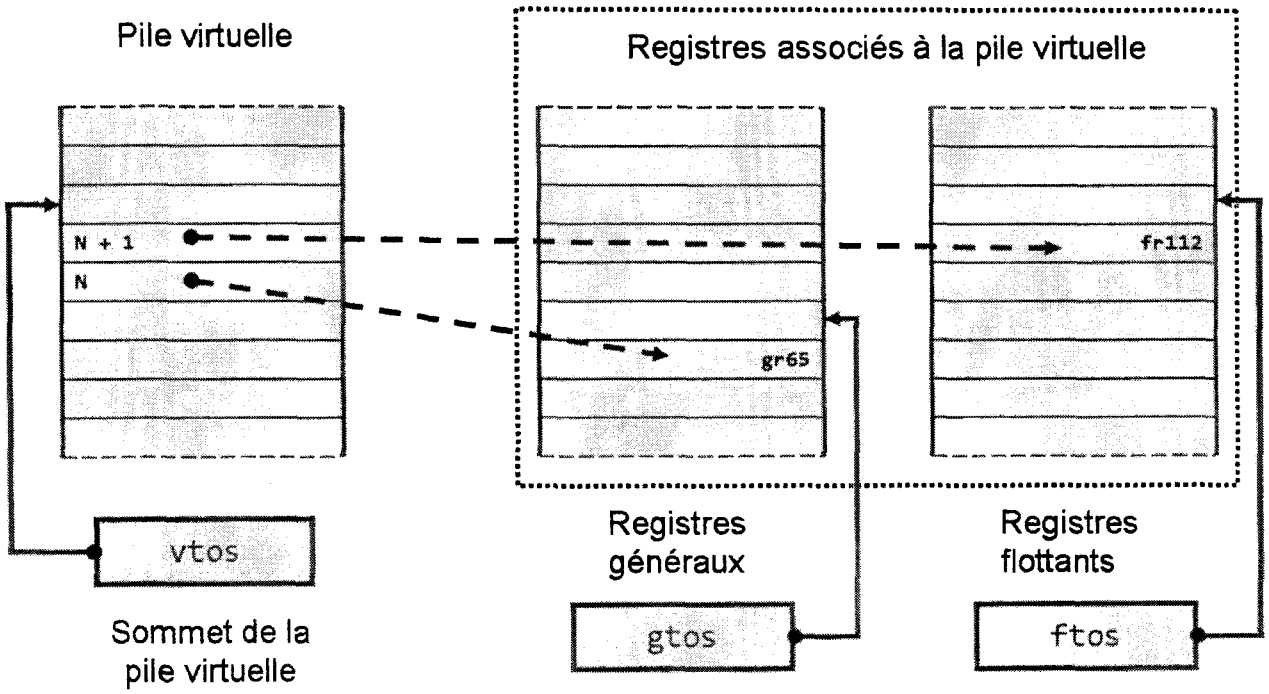

Figure 32 – Représentation logique de la pile virtuelle et de son usage.

Le rôle de la logique de traduction introduite sur la Figure 29 est symbolisée ici par les flèches en pointillés reliant les entrées de la pile virtuelle et les registres physiques. La première flèche en pointillés en partant du bas de la figure signifie que l'entrée N de la pile virtuelle contient dans son champ a le déplacement depuis le registre général Gr63 au registre Gr65 (soit la valeur 2). Pareillement, la seconde flèche en pointillés en partant du bas de la figure signifie que l'entrée N+1 de la pile virtuelle contient dans son champ a le déplacement depuis le registre flottant Fr63 au registre Fr112 (soit la valeur 49).

Pour parachever le mécanisme, deux registres supplémentaires sont nécessaires. Le premier, de deux bits, nommé rt (*type register*), encode un type de données comme cela est indiqué dans la Table 9. Il permet la sélection du pointeur de pile qui sera utilisé lors des opérations implicites de push et pop (entre gtos et ftos). La TABLE 10 précise ce mécanisme de sélection.

Valeur de rt	*Sommet de pile à sélectionner*
0	*Gtos*
1	*Ftos*
2	*Mémoire*
3	*Réservé*

Table 11 – Sélection du pointeur de pile en fonction du registre rt.

Le second registre additionnel requis est de sept bits et il est nommé `ra` (*address register*). Il encode une adresse de registre physique. Cette adresse est réelle et n'est pas un déplacement comme dans la valeur encodée par le champ a des entrées de la pile virtuelle. Ce qui explique le bit supplémentaire utilisé. Rappelons dès à présent que la forme registre des opérandes est encodée dans les code-opératoires par sept bits dans l'architecture *EPIC*. Nous reviendrons sur ce détail par la suite.

Dans sa configuration maximale – donc avec une pile virtuelle de 64 niveaux –, `ra = x + 0x40h`. Où `0x40h` est l'adresse de base des registres physiques utilisés par la pile virtuelle (64 dans notre cas) et x est l'*offset*. L'adresse de base est une variable et est ajustée en fonction du nombre des entrées de la pile virtuelle voulu.

La Figure 33 donne une représentation de l'ensemble des éléments nécessaires à l'implémentation matérielle de la pile virtuelle que nous avons décrite dans cette section. Il est à noter ici que le registre `ra`, en bas à gauche de la figure 13, ne doit pas être confondu avec le champ ra des entrées de la pile virtuelle, Figure 30, qui elle est encodée sur 6 bits seulement.

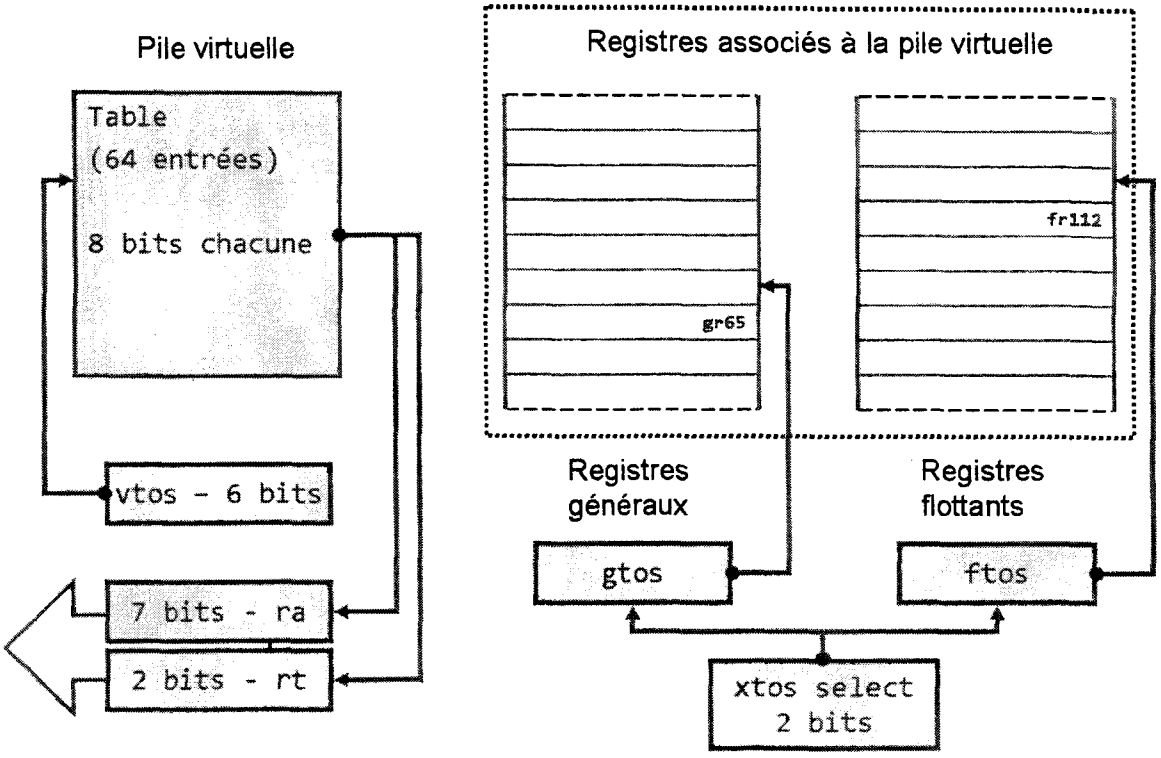

Figure 33 – Représentation des composants matériels nécessaires à l'implémentation de la pile virtuelle. La flèche vers la gauche sous les registres ra et rt symbolise le fait que le contenu valide de ces registres est propagé vers les étages du pipeline : ra pour encoder l'adresse de registre et rt pour le contrôle de type et la levée d'exception.

b. Opérations sur la pile virtuelle

Avant de détailler les opérations de manipulation de la pile, il nous semble important de préciser que notre méthode permet aux instructions *EPIC* d'accéder directement à notre pile d'évaluation. En effet, nous ne proposons pas d'interpréter les instructions *CIL*, mais bel et bien de le traduire en binaire *EPIC*, qui lui, lors de son exécution utilisera notre pile d'évaluation comme le feraient les instructions *CIL* équivalentes. Nous reviendrons plus en détails sur ce mécanisme de traduction binaire dans la suite du texte.

Deux opérations élémentaires sont implémentées : *empiler* (push) et *dépiler* (pop). Comme nous l'indiquions en préambule, ces opérations ne sont pas exposées aux programmes et ne disposent pas de code d'opération. Elles sont implicitement exécutées par la logique de la pile virtuelle lorsque les instructions accèdent en lecture et en écriture à la pile virtuelle. Une écriture se traduisant par un push implicite et une lecture par au moins un pop implicite.

Une opération d'empilement commence par la détection d'une éventuelle exception de débordement de pile sur `vtos`. S'il n'y a pas d'exception alors le registre pointé par `rt` est chargé avec le code du type associé à la donnée en passe d'être empilée. Ce type est déduit du code d'opération (*opcode*) principal de l'instruction (encodé par les bits 37 à 40 de l'*opcode*) et des bits codant le type des instructions du paquet d'instructions (ou *bundle*) – ou *template bits* (encodé par les bits 0 à 5 du *bundle*). La même technique, utilisant l'*opcode* et des *template bits* est en fait utilisée par le dispatcher du processeur pour disperser les instructions des *bundle* vers les ports d'exécution appropriés.

La valeur stockée dans `pv[vtos]` est calculée comme `((rt << 6) | xtos)`. `pv` représente la pile virtuelle dans les pseudo-codes que nous donnerons dans la suite du chapitre. Nous utilisons une notation équivalente à celle des vecteurs dans le langage de programmation C. `xtos` pour sa part représentant l'un ou l'autre de `gots` ou `ftos` (systématiquement noté `x = f | g` par la suite).

Le contenu du registre d'adresse `ra`, calculé par `0x40h + xtos`, est propagé aux étages consécutifs du *pipeline* (l'annexe du document présente le *pipeline* en détail). Cette opération est initiée lors du décodage des instructions (*decode*) et se termine avant l'étape de re-nommage des registres (*register remapping*). Le Listing 12 décrit cette opération sous la forme d'un pseudo-code. Nous le détaillerons par la suite, mais précisons que chaque traducteur d'adresse de la logique de la pile virtuelle opère sur une copie locale de `vtos` qui est mis-a-jour très tôt dans le *pipeline*.

```
push() {
   if(!overflow) { /* si pas de dépassement */
rt = decod_instruction_type(major_opcode, bundle_bits); /* extraction de rt*/
switch(rt) {
case 00: /* cas des registres genereaux gr*/
   pv[vtos] = ((rt << 6) | gtos);
   ra = 0x40h + gtos++;
   break;
case 01: /* cas des registres flottants fr*/
   pv[vtos] = ((rt << 6) | ftos);
   ra = 0x40h + ftos++;
   break;
case 10: /* mémoire */
   /* optionnel */
   break;
default:
   ; /* exception */
```

```
      }
      vtos++; /* mise à jour de vtos */
   } else {
      call_rse_spill(); /* activer le RSE pour gérer le débordement */
   }
}
```

Listing 12 – push dans la pile virtuelle.

Une opération de dépilement – pop – depuis la pile virtuelle commence par la détection d'une éventuelle exception de pile vide sur vtos. S'il n'y a pas d'exception alors le registre rt est chargé avec le code du type correspondant à celui attendu par l'instruction effectuant le pop. Ce type est déduit comme lors de l'empilement d'une donnée, à partir du code d'opération principal de l'instruction et des bits codant le type des instructions du paquet d'instructions. Le contenu du registre rt est alors comparé avec la valeur du champ t de la dernière entrée de la pile virtuelle. Si elles diffèrent alors une exception est générée. Cette exception signale que le type de la donnée dépilée ne correspond pas au type attendu par l'instruction. Dans le cas contraire, pv[vtos] est utilisé pour charger le registre ra, qui reçoit alors l'adresse physique du registre associé a cette entrée. Le contrôle de type ainsi assuré par la logique de la pile virtuelle aide à décharger une partie de la tâche de vérification normalement dévolue au programme de la machine virtuelle. Enfin vtos et xtos sont mis à jour et le contenu du registre ra est lui aussi propagé dans les étages consécutifs du *pipeline*. Le Listing 13 décrit cette opération sous la forme d'un pseudo-code.

Les exceptions de débordement de pile sur vtos sont gérées par le *RSE* qui rappelons-le conserve le contenu des registres temporairement sauvegardés dans la mémoire système réservée à cet effet (*backing store memory*). Ces accès, connus sous le nom de spill et fill s'effectuent rapidement depuis des pages mémoires bloquées par le système d'exploitation. Avant d'invoquer le *RSE*, une détection sur gtos permet de connaître le type des registres à sauvegarder / charger.

```
pop() {
   if(!underflow) { /* si la pile n'est pas vide */
      rt = decod_instruction_type(major_opcode, bundle_bits);
      if(rt != (((pv[vtos]) & (0xC0h)) >> 6)) { /* extraction de rt*/
         raise_exception();/* détection et levée d'exception */
      }
      ra = 0x40h + (pv[vtos] & 0x40h);
      switch(rt) {
      case 00: /* cas des registres genereaux gr */
         --gtos;
```

```
            break;
        case 01: /* cas des registres flottants fr */
            --ftos;
            break;
        case 10: /* mémoire */
            /* Optional */
            break;
        default:
            ; /* exception */
        }
        vtos--; /* mise à jour de vtos */
    } else {
        call_rse_fill();/* activer le RSE pour charger la pile */
    }
}
```

Listing 13 – pop depuis la pile virtuelle.

I. Utilisation de la pile virtuelle par les instructions *EPIC*

Dans cette section, nous présentons la méthode avec laquelle la pile virtuelle sera utilisée par les instructions *EPIC*. La plupart de celles-ci sont triadiques. Seule trois instructions opérant sur les registres flottants ont une forme à quatre registres. Il s'agit des instructions `fma`, `xma` et `fselect`. Nous allons donc utiliser l'instruction `fma` en guise d'exemple. En effet, elle représente l'une des formes les plus compliquées du jeu d'instructions *EPIC*. Elle est aussi l'une des instructions les plus emblématiques des processeurs *Itanium*. Pour mémoire, la forme générique de l'instruction `fma` est donnée par :

```
(qp) fma.pc.sf f1 = f3, f4, f2
```

L'opération effectuée est : `f1 = (f3 x f4) + f2`.

Chaque registre est encodé par sept bits dans le code d'opération de l'instruction. Les opérations triadiques travaillent sur le même principe que l'instruction `fma` mais avec moins d'arguments. Plus de détails sur le jeu d'instruction sont disponibles en annexe du document.

Lorsque la pile virtuelle est utilisée par une instruction `fma`, la logique de contrôle de celle-ci convertit les sept bits encodant les adresses des registres `f1`, `f2`, `f3` et `f4` (appelons-les `fx`) en sept bits encodant l'adresse des registres physiques qui contiennent les données stockées par la pile virtuelle (appelons-les `f'x`). Pour simplifier, nous considérons ici le cas où tous les arguments de l'instruction sont mémorisés dans la pile virtuelle. En effet, cette traduction pour un registre ne prend effet que lorsque le numéro de ce dernier est supérieur ou égal à 64, autrement dit le bit de poids fort de son adresse est égal à `1`. C'est d'ailleurs le seul bit qui doit

être positionné pour déclencher l'utilisation de la pile virtuelle et c'est donc par le positionnement de ce bit que le compilateur / traducteur binaire demande l'utilisation de la pile. Ainsi, si le bit de poids fort est nul, alors l'adresse du registre n'est pas traduite et est traitée comme à l'accoutumée par le *pipeline*. Il s'agit ici du principe général, et cette détection effectue en pratique une comparaison sur un sous-ensemble des bits de l'adresse source pour la détection de l'utilisation de la pile virtuelle. C'est ce qui permet la modification de la taille de la pile virtuelle. Toutefois, dans le schéma que nous décrivons ici, la pile virtuelle a toujours sa taille maximale de 64 entrées et seul le bit de poids fort est utilisé.

Supposons à présent que tous les registres de l'instruction `fma` ont leurs bits de poids fort égaux à `1`. C'est-à-dire que l'instruction `fma` prend tous ses arguments dans la pile virtuelle et y stocke son résultat. Dans ce cas de figure, `f2`, `f3` et `f4` sont traduits comme décrit dans l'opération `pop`, et `f1` est traduit comme décrit dans l'opération `push`. Comme nous l'indiquions en début du chapitre, nous voyons ici que les opérations d'empilement et de dépilement sont implicites et sont exécutées par la logique de la pile virtuelle. A aucun moment, le programme ne peut directement faire un `push` ou un `pop` sur la pile virtuelle. Il peut bien entendu le faire indirectement. Par exemple, pour supprimer le dernier élément empilé, il pourra exécuter :

```
add r2 = r64, r0 ;;
```

La valeur du registre `r2` n'étant pas utilisée mais recevra la valeur dépilée. Le Listing 14 décrit ce mécanisme en pseudo-code. Notez que le compilateur pourrait très bien ne pas stocker le résultat de l'instruction `fma` dans la pile virtuelle, mais dans l'un des registres non utilisés par celle-ci. Il s'agirait alors typiquement de l'un des 32 registres bas de la pile de registres ou de l'un des 32 registres statiques.

```
if(msb(f4 == 1)) { /* si numéro >= 64 */
   f4' = pop(); /* dépiler dans f4' */
}
if(msb(f3 == 1)) {
   f3' = pop();
}
if(msb(f2 == 1)) {
   f2' = pop();
}
if(msb(f1 == 1)) {
   f1' = push();
}
```

Listing 14 – Traduction des registres d'une instruction fma. Il s'agit ici de calculer les adresses physiques des registres à utiliser par le pipeline lors de l'exécution des instructions.

Du point de vue de l'implémentation, la pile virtuelle dispose de deux traducteurs : un pour les registres généraux et un pour les registres flottants. En option, un traducteur peut être ajouté pour les piles maintenues en mémoire. Chaque traducteur peut effectuer quatre traductions d'adresses simultanément.

Comme nous l'indiquions précédemment, chaque traducteur opère avec une copie locale de vtos. Cette copie est réalisée en amont du *pipeline* lors de la phase de décodage du paquet d'instructions (*bundle*). De cette façon nous pouvons absorber la latence de cette opération sérielle par les étages suivants du *pilepline* et la mémoire tampon de découplage du frontal du processeur. Le Listing 15 décrit cette opération en pseudo-code.

```
/* mise à jour sérielle et copie rapide de vtos */
For(i = 4 ; i ; i--) {
      if(msb(fi == 1)) {
      vtos_i = update(vtos);
      call register_address_translator_i();
      }
}

/* traductions parallèles des adresses */

register_address_translator_4() {
   f4' = pop(); /* vtos_4 */
}
register_address_translator_3() {
   f3' = pop(); /* vtos_3 */
}
register_address_translator_2() {
   f2' = pop(); /* vtos_2 */
}
register_address_translator_1() {
   f1' = push(); /* vtos_1 */
}
```

Listing 15 – Traduction simultanée de quatre adresses de registres avec mise à jour de vtos dans la phase de décodage du pipeline. Update met à jour le pointeur de pile virtuel en fonction de l'utilisation de la pile virtuelle des instructions du bundle.

Le mécanisme de traduction de l'adresse des registres sources en adresse de registres de destinations décrit pour l'instruction `fma` est appliqué à toutes les instructions *EPIC* ayant au moins un registre en guise d'opérandes source ou destination – et dont le bit de poids fort est positionné à `1` bien entendu.

Pour permettre un accès en parallèle à la table des entrées de la pile virtuelle, une mémoire multiport (trois ports de lecture et un d'écriture) est utilisée. Comme le nombre d'entrées dans la pile virtuelle est relativement réduit – par rapport au nombre d'entrées dans les mémoires caches par exemple –, la logique nécessaire pour implémenter l'accès parallèle n'est pas très coûteuse (Niar, et al., 2003).

II. Utilisation de la pile virtuelle par les groupes d'instructions

Après la description de l'utilisation de la pile virtuelle par les instructions *EPIC*, nous allons nous pencher à présent sur son utilisation par les groupes d'instructions ou *bundles*. Un processeur fondé sur l'architecture *EPIC* peut exécuter plusieurs groupes d'instructions en parallèle (cf. Annexe *EPIC*). Voyons dans un premier temps comment nous pouvons utiliser la pile virtuelle avec un unique groupe d'instructions. Cela sera l'occasion de présenter les deux modes de fonctionnement possibles pour la pile à savoir: le mode *bloquant* et le mode *non-bloquant*.

Nous expliquerons plus loin comment activer les différents modes de fonctionnement de la pile virtuelle. Quel que soit le mode – bloquant ou pas –, si aucune des trois instructions du groupe n'emploie la pile virtuelle, alors le groupe est traité comme à l'accoutumé, et il n'y a pas d'impact sur le déroulement des opérations.

Dans le mode *bloquant*, la logique de contrôle de la pile virtuelle injecte un bit d'arrêt (*stop bit*) dans le groupe d'instructions pour forcer l'accès séquentiel à la pile. C'est pour cette raison que nous appelons ce premier mode opératoire *bloquant*. Ce mode existe pour satisfaire la nature séquentielle des opérations `push` et `pop` implicitement introduites lors de l'utilisation de la pile virtuelle. Il est d'ailleurs recommandé au compilateur d'encoder par lui-même un bit d'arrêt explicite après les instructions utilisant la pile virtuelle.

En plus de faciliter la lecture et le contrôle du code obtenu par désassemblage, cela permet d'éviter une exécution erronée du code si le second mode opératoire (le mode *non-bloquant*) de la pile virtuelle est actif. La Figure 34 montre symboliquement le fonctionnement de la pile dans ce mode. Nous avons omis sur la figure les instructions nécessaires au chargement des arguments dans la pile (pour charger par exemple C, D, E après le premier stop).

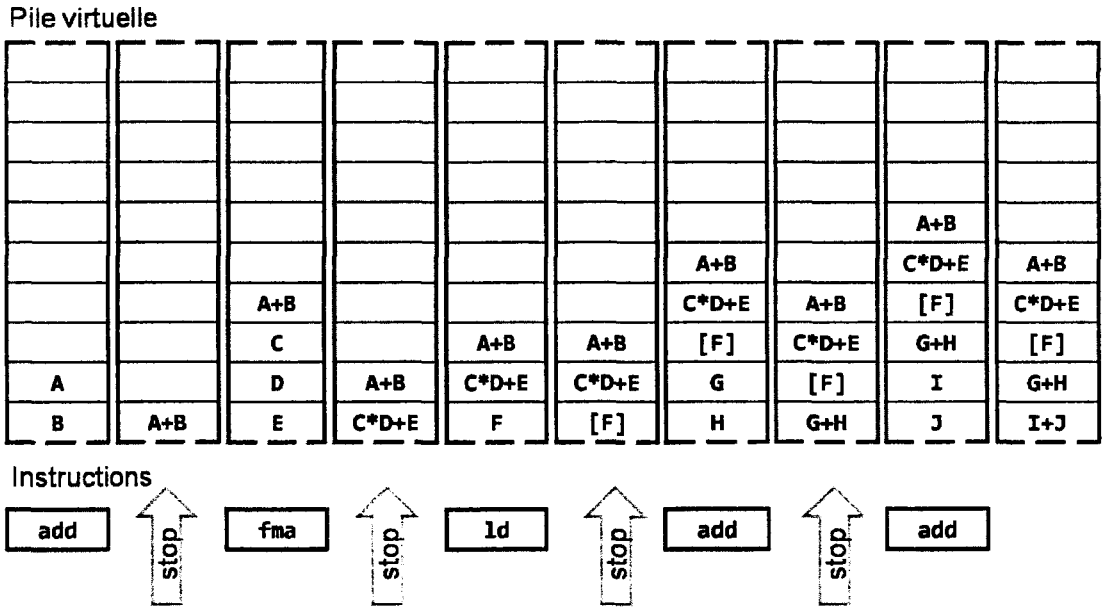

Figure 34 – Représentation du mode bloquant de la pile virtuelle. Le chargement des données dans la pile n'est pas représenté.

Le second mode, que nous appelons le mode *non-bloquant*, n'introduit pas de bit d'arrêt après les instructions du groupe qui utilisent la pile virtuelle. Dans ce mode *non-bloquant*, c'est le rang de l'instruction dans le groupe d'instructions qui détermine le niveau de la pile virtuelle où les écritures et les lectures s'effectuent. Ce mode est important car il nous permet de lever la séquentialité de l'accès à la pile. Ainsi, les instructions du groupe d'instructions, programmées pour le même cycle d'exécution par le compilateur puisqu'elles sont indépendantes, ne sont pas bloquées. L'indépendance des instructions du groupe étant par principe vérifiée.

Voyons à présent comment l'opération se déroule avec un exemple de deux groupes d'instructions – en utilisant la notation explicite des groupes d'instructions (Listing 16). Le premier groupe (lignes 2 à 4) renferme une instruction d'addition entière (`add`) et une instruction flottante de multiplication / addition fusionnée que nous avons déjà vues (`fma`). Le deuxième groupe (lignes 6 à 9) renferme deux additions entières et une lecture en mémoire (`(qp) ld4 r1 = [r2]`). Toutes les instructions sont indépendantes les unes des autres et toutes utilisent la pile virtuelle comme le signale la notation `r1xxxxxx`. Cette notation indique juste que le numéro est supérieur à 64.

```
1: { .mfi /* template bits */
2:     add r1xxxxxx = r1xxxxxx, r1xxxxxx            /* slot 0 */
3:     fma f1xxxxxx = f1xxxxxx, f1xxxxxx, f1xxxxxx  /* slot 1 */
```

```
4:    .nop.i                                          /* slot 2 */
5: } { .mii /* template bits */
6:     ld4 r1xxxxxx = [r1xxxxxx]                      /* slot 0 */
7:     add r1xxxxxx = r1xxxxxx, r1xxxxxx              /* slot 1 */
8:     add r1xxxxxx = r1xxxxxx, r1xxxxxx              /* slot 2 */
9: }
```

Listing 16 – Deux groupes d'instructions types utilisant la pile virtuelle.

Par convention – et arbitrairement –, la pile virtuelle effectue la traduction d'adresses des registres en commençant par l'instruction située dans le premier emplacement du groupe d'instructions (ou *slot 0*) et continue jusqu'à la fin de la section parallèle, qui correspond ici à la fin du deuxième groupe d'instructions mais qui pourrait aussi bien être le premier bit d'arrêt explicite ou implicite rencontré. Les Figure 35 et Figure 36 résument le résultat de l'exécution des deux groupes d'instructions sur la pile virtuelle.

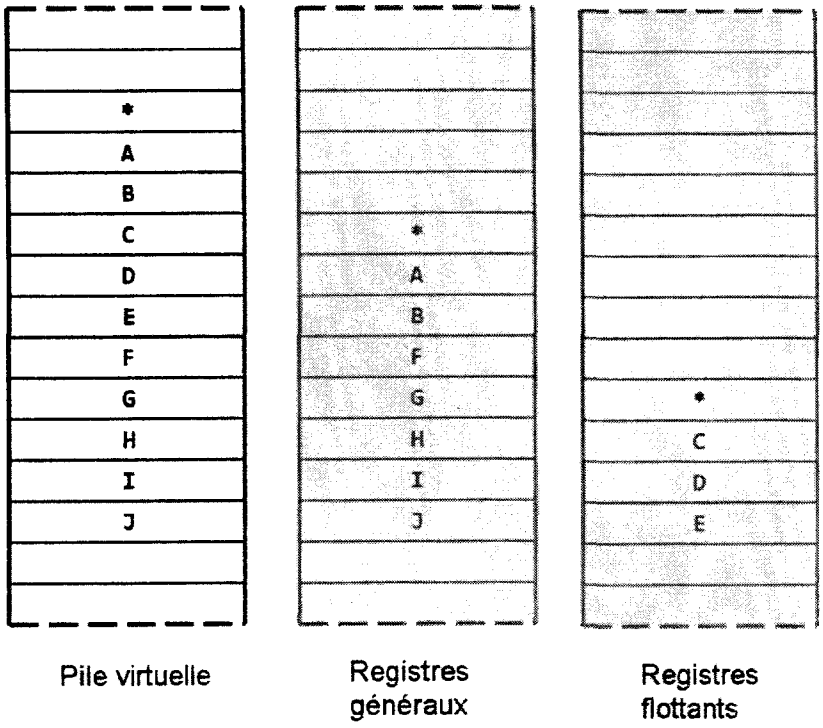

*Figure 35 – État initial de la pile virtuelle. * représente xtos.*

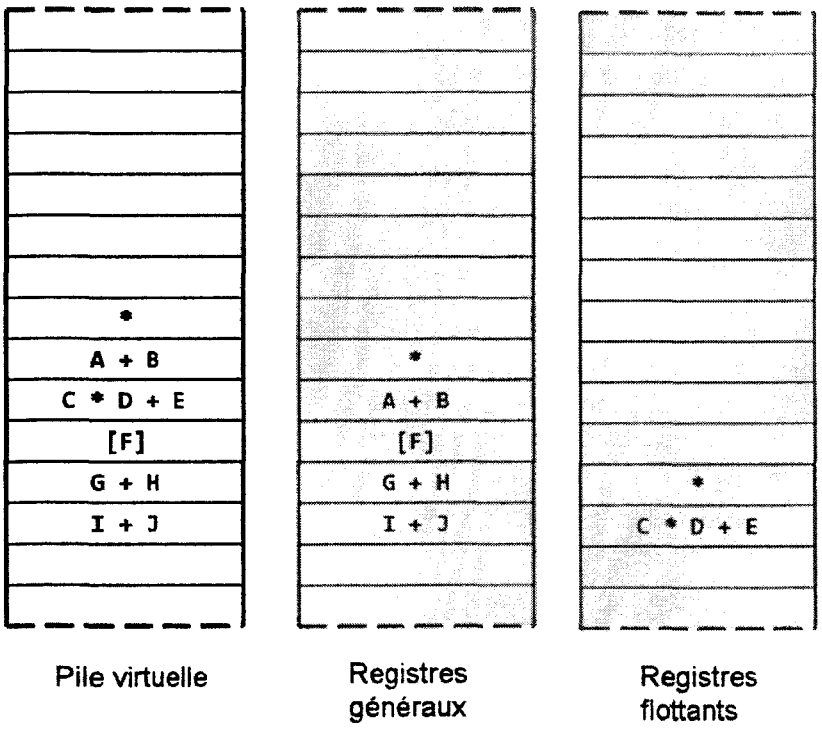

*Figure 36 – État final de la pile virtuelle. * représente xtos.*

Cet exemple montre comment un compilateur peut programmer l'exécution en parallèle de plusieurs instructions *EPIC* qui utiliseraient la pile (en utilisant le mode *non-bloquant* et en regroupant les instructions dans une section parallèle). Ceci est réalisé rappelons-le par le fait que le rang de l'instruction dans le groupe d'instructions détermine le niveau de la pile virtuelle où les écritures et les lectures s'effectuent. Ainsi, l'instruction `add` (ligne 8) lira les opérandes A et B, l'instruction `add` (ligne 7) lira C et D, et ainsi de suite. En plus de la recherche et de la détection des instructions indépendantes, le compilateur, en fonction des slots de destination des instructions devra charger la pile de façon adéquate, comme nous le montrons dans notre exemple.

Pour tirer pleinement profit de l'architecture *VLIW* des processeurs fondés sur l'architecture *EPIC*, et extraire ainsi le plus haut degré de parallélisme d'instructions (*ILP*) possible, c'est au compilateur qu'échoit la tâche d'utiliser le mode *non-bloquant* de la pile virtuelle. C'est également au compilateur de décider s'il souhaite ou pas utiliser la pile virtuelle. Pour ce faire, un premier bit dans le processeur signal si la pile virtuelle est implémentée ou pas. Ce bit doit être lu puis testé lors de l'exécution du code *via* l'instruction `cpuid`. Le compilateur doit donc générer ce code au point d'entrée du programme pour détecter la présence de la fonctionnalité. Ce test n'est pas nécessaire lors de la compilation, et une directive suffit pour autoriser le compilateur à utiliser la pile virtuelle. En effet, rien ne garantit que celle-ci soit disponible sur la plate-forme où le code sera exécuté. De part la nature très peu intrusive de la

pile virtuelle, si le retour de l'instruction `cpuid` n'est pas testé, alors rien n'indiquera au programme que la pile virtuelle n'est pas présente et le programme s'exécuterait probablement sans heurts, mais son résultat serait incontestablement erroné.

Sept bits, cette fois-ci situés dans un *Model Specific Register – MSR –* du processeur permettent au code généré spécialement à cet effet par le compilateur de configurer et d'activer ou de désactiver dynamiquement la pile virtuelle. Un premier bit, s'il est égal à `1`, active la pile virtuelle – qui rappelons-le doit être présente dans le processeur. La taille de la pile virtuelle est fixée pour sa part par les six bits restants des sept bits du *MSR*. Une taille nulle génère une exception lors du premier accès à la pile. Si le bit d'activation est remis à zéro, alors la pile virtuelle est désactivée et la logique de contrôle de la pile virtuelle se trouve contournée dans le *pipeline*. Notons que l'activation ou la désactivation de la pile virtuelle peut aussi être effectuée par les utilitaires systèmes capables de modifier les *MSR*.

5. Traduction de *CIL* vers *EPIC*

Le jeu d'instructions *CIL* s'apparie quasiment un-à-un avec le jeu d'instructions *EPIC*. Nous profitons de cette propriété pour proposer un traducteur 1:1 de binaire *CIL* en binaire *EPIC* utilisant la pile virtuelle. Cette traduction permet entre autre la suppression de la phase d'allocation des registres requis par un compilateur.

L'allocation des registres est un problème *NP* complet et requiert une heuristique évoluée pour trouver le juste équilibre entre le temps de compilation – qui doit être minimisé – et la qualité du code généré – qui doit être maximisée. De nombreux algorithmes d'allocation de registres sont disponibles. Si les auteurs de compilateurs statiques préfèrent la méthode de coloration de graphe (*graph coloring* (Garey, et al., 1974), les auteurs de compilateurs *JIT* implémentent généralement l'algorithme du *Linear Scan Register Allocation* (Poletto, et al., 1999) (Cooper, et al., 2006). La Figure 37 détaille la structure classique d'un compilateur *JIT* en mettant l'accent sur la phase d'allocation des registres située sur le chemin critique de la compilation.

Le temps utilisé par la phase d'allocation des registres est éminemment variable et dépend de la complexité du code source. Cette durée tend à croître avec la taille des blocs de base (*basic blocs*) et la nature des optimisations demandées (optimisations inter procédurales, *etc.*). Lors de nos expérimentations, nous avons mesuré une durée d'allocation représentant en moyenne 6% du temps total de la compilation.

Certains codes – minoritaires en nombre il est vrai – nécessitant jusqu'à 20% du temps de la compilation. Typiquement, les optimisations inter-procédurales sont à l'origine de cette augmentation significative. L'utilisation de la pile virtuelle par notre traducteur binaire 1:1 permet la suppression de la phase d'allocation des registres, puisqu'en utilisant celle-ci, l'allocation des registres s'effectue dynamiquement à l'exécution du code par la logique de contrôle de la pile virtuelle.

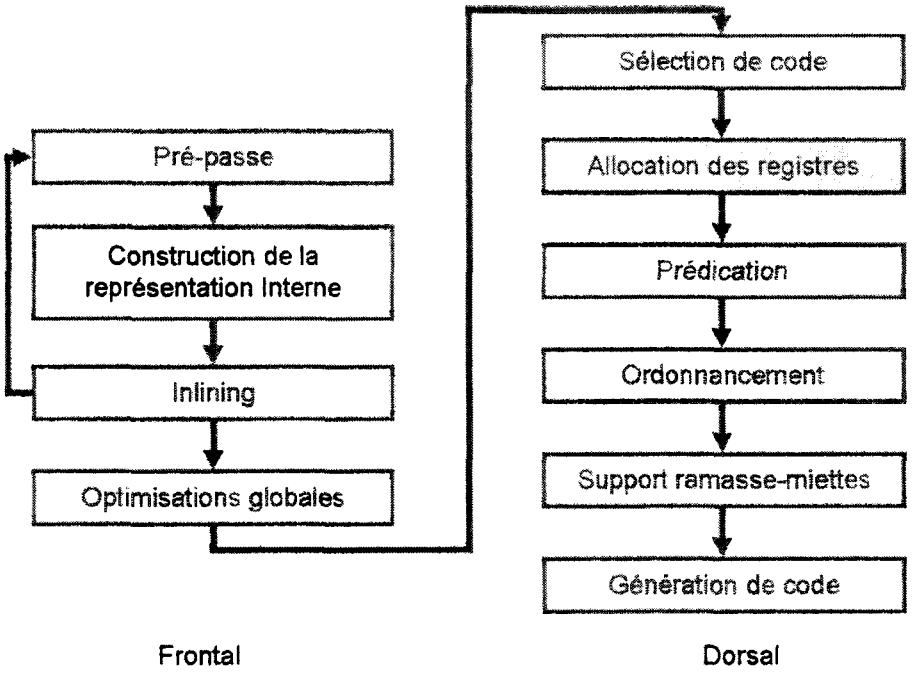

Figure 37 – Structure classique d'un compilateur JIT.

Pour décrire notre méthode de traduction, prenons en guise d'exemple une fonction simple qui calcule en double précision la surface d'un triangle irrégulier en utilisant la formule d'Héron. Le désassemblage partiel du code *CIL* de cette fonction est donné dans le Listing 10. Seules les instructions nécessaires au calcul y sont représentées. Nous retrouvons ici la structure classique d'un code *CIL* qui emploie la pile d'évaluation du *CLR* pour effectuer ses calculs. La Figure 25 montre symboliquement l'utilisation de la pile d'évaluation pour cette fonction. Habituellement, un compilateur *JIT*, *via* la phase d'allocation des registres transforme les références à la pile d'évaluation en références directes aux registres du processeur.

```
01: IL_0000:  ldarg.0  // a - clacul de s = ½(a+b+c)
02: IL_0001:  ldarg.1
03: IL_0002:  add
04: IL_0003:  ldarg.2
05: IL_0004:  add
06: IL_0005:  ldc.r8 0.5
07: IL_000e:  mul
08: IL_000f:  stloc.0
09: IL_0010:  ldloc.0 // clacul de s(s-a)(s-b)(s-c)
```

```
10: IL_0011:  ldarg.0
11: IL_0012:  sub
12: IL_0013:  ldloc.0
13: IL_0014:  mul
14: IL_0015:  ldloc.0
15: IL_0016:  ldarg.1
16: IL_0017:  sub
17: IL_0018:  mul
18: IL_0019:  ldloc.0
19: IL_001a:  ldarg.2
20: IL_001b:  sub
21: IL_001c:  mul
```

Listing 17 – Désassemblage partiel de la fonction Héron. La valeur hexadécimale qui suit IL_ indique le numéro de l'octet encodant l'instruction.

Au lieu de cela, notre traduction binaire 1:1 génère directement et en une simple passe le code présenté dans le Listing 18. Nous y avons conservé le code *CIL* originel sous forme de commentaires pour plus de clarté. Nous pouvons ainsi remarquer qu'à part le chargement de la constante flottante `0.5` en double précision dans la pile qui nécessite deux instructions avec les processeurs *Itanium* (instructions 6 et 7), chaque instruction *CIL* est traduite en une instruction *EPIC*. Ce code utilise bien entendu la pile virtuelle comme nous l'avons décrit lors de sa description. L'utilisation de la pile virtuelle dans le code est matérialisée par l'utilisation du registre `f64`. Les sept bits encodant ce registre sont dynamiquement traduits en adresses physiques des registres qui contiennent les valeurs référencées.

```
01: fadd.d    f64=f8,f0 ;;          // ldarg.0
02: fadd.d    f64=f9,f0 ;;          // ldarg.1
03: fadd.d    f64=f64,f64 ;;        // add
04: fadd.d    f64=f10,f0 ;;         // ldarg.2
05: fadd.d    f64=f64,f64           // add
06: movl      r2=0x3fe0000000000000 ;;
07: setf.d    f64=r2 ;;             // ldc.r8 0.5
08: fma.d     f64=f64,f1,f64 ;;     // mul
09: fadd.d    f32=f64,f0 ;;         // stloc.0
10: fadd.d    f64=f12,f0 ;;         // ldloc.0
11: fadd.d    f64=f8,f0 ;;          // ldarg.0
12: fsub.d    f64=f64,f64 ;;        // sub
```

```
13: fadd.d    f64=f32,f0 ;;       // ldloc.0
14: fmad.d    f64=f64,f1,f64 ;;   // mul
15: fadd.d    f64=f32,f0 ;;       // loadloc.0
16: fadd.d    f64=f9,f0 ;;        // loadarg.1
17: fsub.d    f64=f64,f64 ;;      // sub
18: fma.d     f64=f64,f1,f64 ;;   // mul
19: fadd.d    f64=f32,f0 ;;       // ldloc.0
20: fadd.d    f64=f10,f0 ;;       // ldarg.2
21: fsub.d    f64=f64,f64 ;;      // sub
22: fma.d     f64=f64,f1,f64 ;;   // mul
```

Listing 18 – Résultat partiel de la traduction binaire 1:1 de la fonction Héron.

Pour faciliter la compréhension du code généré, il nous faut préciser que la traduction binaire 1:1 utilise les conventions suivantes. Les arguments des fonctions (`arg.0`, `arg.1`, *etc.*) sont maintenus dans les registres [`f8-f15`] et [`r32-r63`] en fonction de leurs types. Les variables locales (`loc.0`, `loc.1`, *etc.*) sont maintenues selon leur type dans les registres [`f32-f63`] et les registres brouillons – ou *scratch registers* – [`r14-r27`]. En ce sens, nous ne faisons que respecter les *Application Binary Interfaces* (*ABIs*) définies pour les processeurs *Itanium*. La Figure 38 donne le plan d'utilisation des registres des processeurs *Itanium*.

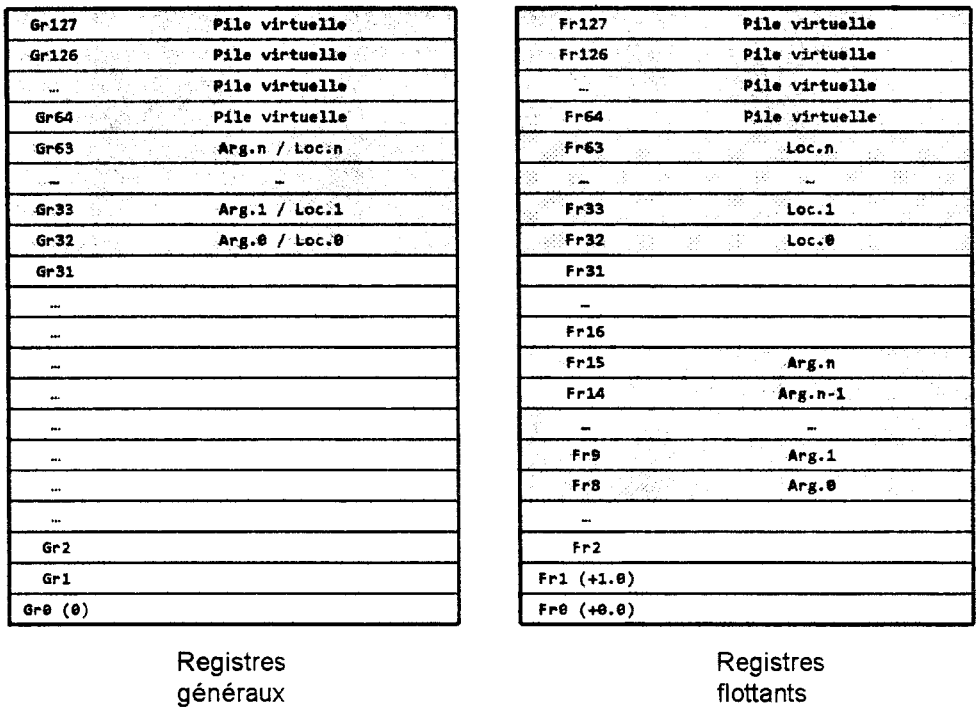

Figure 38 – Plan d'utilisation des registres des processeurs Itanium.

Il nous faut également rappeler que de nombreuses instructions *EPIC* ne sont en fait que des synonymes. Ainsi `fadd` est employé comme une instruction de transfert de registre à registre. `fadd` et `fsub` sont respectivement les synonymes de `fma` et `fms`. Ceci se comprend puisque le processeur *Itanium 2* dispose de onze ports d'exécution et unités d'exécution et que parmi celles-ci figurent deux unités de calculs flottants optimisées pour les additions et les multiplications fusionnées (5 cycles d'horloge par `fma` en moyenne). Enfin, rappelons que le registre `f0` est câblé à `+0.0` et que le registre `f1` est câblé à `+1.0`. Par exemple, la première instruction *CIL* du Listing 18 qui charge dans la pile d'évaluation le premier argument de la fonction est traduite en `fadd.d f64 = f8, f0 ;;`. `f64` active la logique de contrôle de la pile virtuelle et indique que le résultat de l'addition sera empilé au sommet de la pile virtuelle. Le décodage du type du résultat de l'instruction force la sélection de `ftos` et le résultat sera empilé dans les registres flottants comme escompté. L'opération consiste ici simplement à additionner le contenu du registre `f8` et de `+0.0`. Ce qui se traduit par le chargement dans la pile virtuelle du contenu de `f8`. Le double point-virgule (`;;`) représente le bit d'arrêt explicitement ajouté au groupe d'instructions.

Nous avons exploré certaines optimisations élémentaires (*peephole optimizations*) lors de la conception de notre traducteur binaire 1:1, tout en nous contraignant à ne pas rajouter de passe supplémentaire à la traduction. Bien entendu, il est possible – et même souhaité – comme nous le verrons par la suite d'ajouter des optimisations globales en employant des techniques de compilation traditionnelle. La propagation de constantes, le *Global Value Numbering*, l'optimisation des boucles – et notamment la mise en place de boucles *pipelines* lorsque cela est possible – la vectorisation ou l'*inlining* ne sont que quelques exemples. Cependant, une telle exploration ne faisait pas partie de notre objectif initial. Nous les réservons pour une étude ultérieure.

Parmi les optimisations applicables en une simple passe, nous avons retenu la fusion des opérations contigües sur la pile, le référencement direct aux arguments et aux variables locales lorsque cela est possible et la programmation pour le même cycle d'exécution de lectures et d'écritures indépendantes en mémoire. La recherche de ces optimisations s'effectue par une lecture en-avant sur un buffer de 128 instructions *CIL*. La taille de la fenêtre de lecture influe sur les optimisations trouvées par le traducteur. Au-delà de 128 instructions, nous ne voyons plus la différence et ne détectons pas de nouvelles possibilités. Avec une fenêtre plus étroite, il nous est difficile de détecter les lectures et les écritures en mémoire qui peuvent êtres programmées pour le même cycle. La recherche d'algorithmes plus efficaces devrait améliorer la détection d'instructions indépendantes.

La fusion des opérations de manipulation de pile contigües en revanche s'applique très bien et elle nous permet de supprimer la quasi-totalité des accès redondants à la pile virtuelle. Pour ce faire, nous mémorisons le long de notre fenêtre d'analyse les opérations *CIL* qui écrive dans la pile d'évaluation avec la source des données empilées. Pour chaque opération qui consomme au moins une donnée de la pile d'évaluation, nous recherchons dans l'historique des empilements la source de la donnée.

Lorsque nous générons l'*opcode* de l'instruction *EPIC*, au lieu d'encoder le registre source par la valeur `f64` ou `g64`, nous encodons directement la source trouvée. Ainsi, la source des deux premières instructions *CIL* sont mémorisées (respectivement `f8` et `f9`) (Listing 18). La

troisième instruction *CIL* (l'addition qui consomme deux arguments de la pile d'évaluation) est traduite en une instruction *EPIC*. Au lieu de coder `fadd.d f64 = f64, f64 ;;` nous générons `fadd.d f64 = f8, f9 ;;`. L'historique est mis à jour et nous supprimons les deux entrées consommées de l'historique. Suivant le même principe, les instructions *CIL* 4 et 5 sont fusionnées en `fadd.d f64 = f10, f64`. Puisque l'addition (instruction *CIL* 5) consomme deux entrées de la pile d'évaluation et puisque notre historique ne contient qu'une seule entrée à ce stade, nous encodons l'usage de la pile virtuelle pour lire le second argument et encodons directement la source du premier depuis l'historique. Une seconde structure conserve les instructions indépendantes. Avec le code *CIL*, il s'agit essentiellement des lectures et des écritures en mémoire (les références aux objets et aux méthodes). Ainsi, si une opération indépendante est détectée – ou est requise pour implémenter une opération *CIL* en langage machine *Itanium* – alors celle-ci est programmée pour une exécution dans le même cycle.

Ainsi l'instruction `movl r2 = 0x3fe0000000000000` du Listing 18, qui charge de la constante flottante `0.5` dans le registre `r2` en vue de sa conversion en flottant double précision dans un registre flottant, est programmée avec `fadd.d f64 = f10, f64`. Le Listing 19 donne le résultat de nos optimisations élémentaires sur la séquence listée dans le Listing 18.

```
01: fadd.d    f64=f8,f9 ;;       // ldarg.0, ldarg.1, add
02: fadd.d    f64=f10,f64        // ldarg.2, add
03: movl      r2=0x3fe0000000000000 ;;
04: setf.d    f64=r2 ;;          // ldc.r8 0.5
05: fma.d     f64=f64,f1,f64 ;;  // mul
06: fadd.d    f32=f64,f0 ;;      // stloc.0
07: fadd.d    f64=f12,f0 ;;      // ldloc.0
08: fsub.d    f64=f64,f8 ;;      // ldarg.0, sub
09: fma.d     f64=f64,f1,f32 ;;  // ldloc.0, mul
10: fsub.d    f64=f32,f9 ;;      // ldloc.0, ldarg.1, sub
11: fma.d     f64=f64,f1,f64 ;;  // mul
12: fsub.d    f64=f32,f10 ;;     // ldloc.0, ldarg.2, sub
13: fma.d     f64=f64,f1,f64 ;;  // mul
```

Listing 19 – Traduction du code CIL (Listing 10) en EPIC optimisé.

Il est intéressant de comparer le résultat de notre traduction binaire 1:1 optimisée au code généré par un compilateur statique optimiseur pour le même code source. Nous avons choisi pour cela le compilateur C/C++ d'*Intel Corporation* reconnu pour la qualité de son code généré. Nous avons activé le niveau d'optimisation O3 qui correspond au niveau d'optimisation le plus élevé.

Le Listing 20 donne le résultat de cette compilation. La différence essentielle entre les deux versions réside bien entendu dans l'utilisation de la pile virtuelle pour notre code, là où le compilateur statique utilise les références directes aux registres. Nous reviendrons plus tard en détails sur la qualité du code généré par notre traducteur dans la section suivante. Pour autant, nous pouvons constater que les deux codes sont très proches en termes de nombre d'instructions générées et de nombre de cycles programmés. Nous générons 13 instructions contre 11 pour le compilateur statique. Nous programmons 11 cycles contre 9 pour le compilateur statique. Nous sommes en déficit sur ce dernier point – et cela est vrai généralement. En revanche, notre traduction ne demande qu'une seule passe sur le binaire *CIL*. La génération en une seule passe s'avère être intéressante par exemple lors de la phase de mise au point d'une application. Ainsi, il serait possible de placer le compilateur *JIT* dans un mode mise-au-point, utilisant notre traduction binaire. L'avantage serait alors d'obtenir de meilleures performances que l'interprétation pure, tout en réduisant le temps nécessaire à la génération du binaire. Cet avantage s'applique également à la génération de binaire pour des méthodes qui ne seraient pas compilées par le *JIT* normalement, mais qui pourraient l'être à la vue du faible coût de la traduction binaire.

```
01: fma.d    f15=f8,f1,f9
02: movl     r2=0x3fe0000000000000 ;;
03: setf.d   f14=r2 ;;
04: fma.d    f13=f15,f1,f10 ;;
05: fma.d    f12=f14,f13,f0 ;;
06: fms.d    f11=f12,f1,f8
07: fms.d    f9=f12,f1,f9 ;;
08: fms.d    f8=f12,f1,f10 ;;
09: fma.d    f7=f12,f11,f0 ;;
10: fma.d    f6=f7,f9,f0 ;;
11: fma.d    f8=f6,f8,f0 ;;
```

Listing 20 – Code produit par un compilateur statique optimiseur.

6. Résultats expérimentaux

a. Méthodologie de tests

Cette section est consacrée à la description de notre méthodologie de tests et d'évaluation de l'impact que peut avoir la pile virtuelle sur un compilateur *JIT* simplifié. Dans un premier temps, nous allons travailler sur l'algorithme de *Kasumi* (Third Generation Partnership Project, 2001) pour éprouver notre traducteur sur un premier code complet. Cela nous permettra d'apporter des éclaircissements sur les causes de certains résultats. Dans un second temps, nous

présenterons le résultat de nos expériences sur un ensemble de benchmarks *.NET*. Nous avons retenu à cet effet les codes *XMLMark 1.1* (Sun Microsystems, 2006), *NPerf* (De Halleux, 2004) et *WSTest* (Mundlapudi, et al., 2007).

Kasumi est un algorithme de chiffrage itératif de blocs qui produit une séquence de 64 bits en sortie à partir d'une séquence de 64 bits en entrée sous le contrôle d'une clef de chiffrement de 128 bits. L'organigramme de gauche de la Figure 40 présente les étapes successives menant au chiffrement d'une séquence de 64 bits en entrée. Les organigrammes de FL et FO ne sont pas représentés sur la figure. Ces fonctions utilisent leurs propres sous-clés de chiffrement.

La Figure 39 donne pour sa part l'algorithme de la fonction de confidentialité *f8* utilisée en téléphonie mobile pour le chiffrement des communications. Cette fonction agit en tant que *driver* et utilise *Kasumi* pour chiffrer une séquence de bits indifféremment longue découpée en blocs de 64 bits.

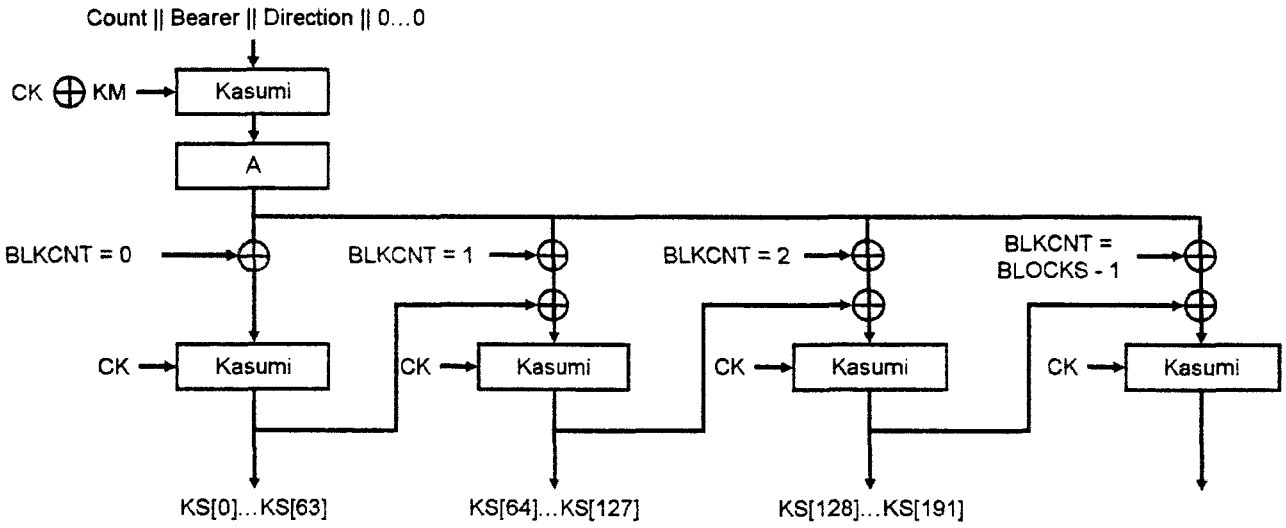

Figure 39 – Algorithme de la fonction de confidentialité f8 fondée sur Kasumi – Source : 3GPP.

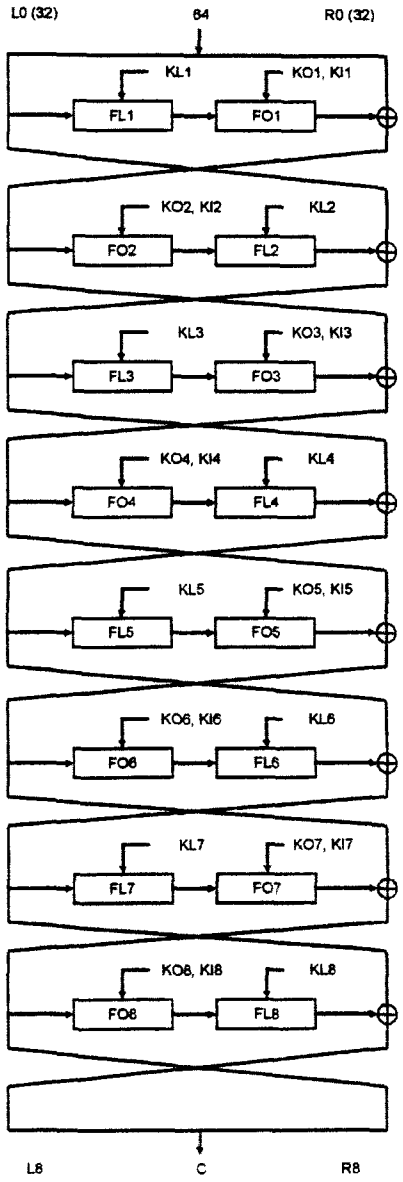

Figure 40 – Algorithme de Kasumi – Source : 3GPP. ⊕ représente un ou exclusif. FOx et FLx sont des sous-fonctions de Kasumi. Chaque sous-fonction utilise des clefs pour encoder les données.

Ce qu'il faut retenir de cette description, c'est que le code *Kasumi / f8* est dominé par : des calculs sur les nombres entiers, des manipulations logiques et des branchements. En ce sens, ce code est représentatif des applications d'entreprises destinées aux serveurs équipés des processeurs *Itanium*.

Nous utilisons comme point de départ le code de référence de *Kasumi* et de la fonction *f8* en langage C proposé par le *3GPP*[2] (Third Generation Partnership Project, 2001). Nous les implémentons en C++ natif et C++ *managé pour la CLR* (Microsoft Corporation) (Les deux codes sont strictement identiques du point de vue de leurs routines de calcul. Nous collectons ensuite le profil d'exécution du code en validant le résultat des calculs. La validation se fait en comparant le résultat de chiffrement obtenu avec les vecteurs de tests fournis par le *3GPP*. La Figure 41 donne la répartition du nombre des instructions non-annulées (*retired instructions*) pour le code *Kasumi*. Conformément à ce qu'indiquent les algorithmes de *Kasumi* et de *f8*, nous retrouvons les bonnes fonctions dans notre profil. La fonction `key_schedule` qui représente 33% des instructions exécutées génère les clefs intermédiaires et locales utilisées par les fonctions `FO`, `FL` et `FI`. D'ailleurs, si ce code devait être optimisé, il faudrait commencer par cette fonction, suivi de `FI` et `FO`.

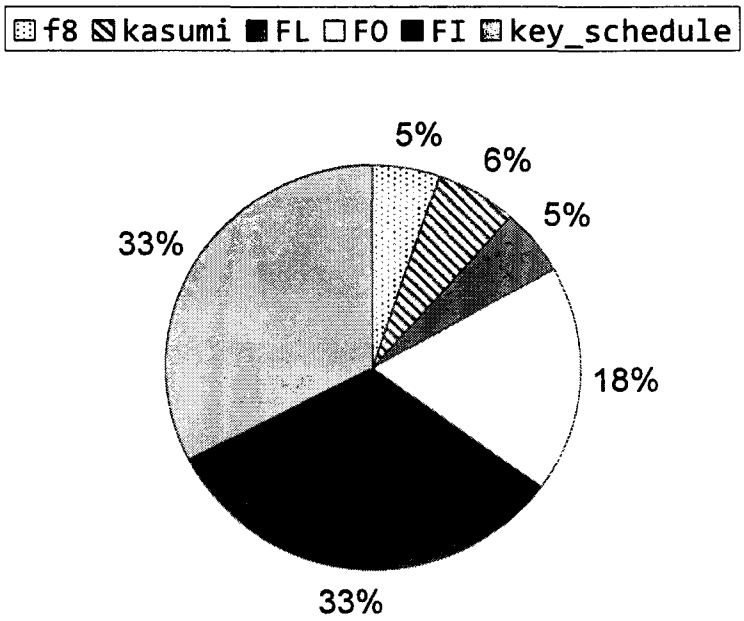

Figure 41 – Répartition des instructions non-annulées pour Kasumi / f8.

Pour collecter les données nécessaires à nos comparaisons, nous commençons par compiler le code source en binaire *CIL*. Il est intéressant de noter que durant nos premières tentatives, les compilateurs n'étaient pas en mesure de générer du *CIL* pour la fonction `key_schedule`, et généraient du code natif à la place du *CIL*. Cela était essentiellement dû au fait

[2] *3rd Generation Partnership Project* – il s'agit du consortium des organes de contrôle et de régulation de la téléphonie mobile en charge de la téléphonie mobile de troisième génération (3G).

que cette fonction utilisait des pointeurs pour accéder à ses structures de données. La suppression des pointeurs a fini par résoudre ce problème.

Le binaire *CIL* est ensuite traduit par notre traducteur 1:1 en binaire *EPIC* utilisant la pile virtuelle. Nous générons ensuite un deuxième binaire *EPIC* optimisé en utilisant la seconde mouture de notre traducteur 1:1 (la version optimisée). Une troisième version du binaire est générée en utilisant le compilateur *AOT NGen* de *Microsoft Corporation*. Ce binaire nous a servi à la vérification du code généré en utilisant *PIN* (Intel Corporation, 2008). Nous avons écrit à cet effet un ensemble de modules *pintool* pour simuler la pile virtuelle. Avec une pénalité d'un facteur dix mille sur le temps d'exécution, nous avons vérifié les seize premiers octets des vecteurs de test. Ces vecteurs de test – des séquences de bits et leurs versions cryptées– sont fournis par le *3GPP* pour valider les implémentations de *Kasumi*. Les clefs sont également fournies.

Pour récupérer le binaire généré par le compilateur *JIT* de *.NET*, nous utilisons l'analyseur de performances *VTune* et sa fonction d'échantillonnage (Intel Corporation). Pour nous assurer que nous obtenons la totalité du code généré et exécuté, nous fixons la valeur d'échantillonnage au centième de la fréquence d'horloge du processeur pour les instructions non-annulées et les cycles non-arrêtés. Nous laissons ensuite le programme s'exécuter plusieurs fois de suite. Le désassemblage du code est alors enregistré. En procédant ainsi, bien que *VTune* soit un outil statistique dans le mode d'échantillonnage sur les compteurs matériels du processeur, nous avons la garantie de capturer la trace d'exécution du code, et donc du code en lui-même.

Finalement, nous avons généré un binaire de référence avec l'aide du compilateur statique optimiseur C/C++ d'*Intel Corporation* (Intel Corporation) en invoquant le niveau d'optimisation `O3`, l'*Inter-Procedural Optipizer – IPO* – et le *Profile Guided Optimizer – PGO*. Ce binaire est considéré comme la référence en termes de performance et de qualité de code obtenu.

En guise de critères de comparaison des différents binaires obtenus, nous comptons le nombre d'instructions générées – à l'exception des instructions `nop` – ainsi que le nombre de cycles programmés. Pour ces deux critères, plus la valeur est basse, meilleure est la qualité du code. Notons qu'il nous a malheureusement été impossible de mesurer avec précision le temps de compilation utilisé par le compilateur *JIT*, n'ayant pas accès aux options – qui existent certainement – qui permettent de connaître le temps passé dans chacune des phases de la compilation. Bien que nous puissions affirmer que la traduction binaire est de loin plus rapide que la compilation à la volée, nous n'utiliserons pas cet argument pour notre comparaison, faute de preuve.

b. Analyse des données

Comme escompté, et d'une façon générale, le nombre d'instructions générées par le traducteur 1:1 est équivalent à celui des instructions du code *CIL*. Il est à noter que si en général, ce nombre est égal ou inférieur au nombre des instructions du code *CIL* (de 0,17% en moyenne), plusieurs opérations *CIL* nécessitent la génération d'instructions avec un ratio de 1:2 à 1:17.

Cela est requis pour satisfaire l'architecture *EPIC*. Ainsi, les branchements et les comparaisons utilisent des registres de prédicats qu'il faut calculer au préalable. Nous l'avons vu en introduction, le chargement de constantes en virgule flottantes nécessite deux instructions *EPIC*. Pis-aller, puisque la division ne dispose pas de son instruction *EPIC*, nous devons donc générer 17 instructions *EPIC* pour l'effectuer. En effet, le choix a été fait par les architectes de l'*Itanium* de laisser le programmeur et le compilateur de choisir – et de fournir au besoin – la routine de division la mieux adaptée au code. Il est ainsi possible de favoriser soit la précision du calcul, soit sa vitesse. Le processeur propose en revanche une instruction pour calculer la réciproque d'un nombre. Cette valeur peut par exemple service de valeur amorce pour un algorithme de calcul par convergence. Dans notre cas, nous avons choisi de limiter le nombre des opérations. Si cela n'est pas un problème en termes de performance, pour notre décompte des instructions, nous sommes pénalisés par rapport au *CIL* qui dispose de la division.

```
...                 ...
ldc.i4.8            add r64=8,r0 ;; // ldc.i4.8
div                 add r10=65501,r0 // div
stloc.0             sxt4 r9=r64 // div
...                 sxt4 r8=r64 ;; // div
                    setf.sig f6=r9 // div
                    setf.sig f15=r8 // div
                    cmp.ne.unc p0,p7=0,r9 ;; // div
                    (p7) break 1 // div - by 0
                    setf.exp f14=r10 ;; // div
                    fcvt.xf f12=f6 // div
                    fcvt.xf f13=f15 ;; // div
                    frcpa.s1 f11,p6=f13,f12 ;; // div
                    (p6) fma.s1 f9=f13,f11,f0 // div
                    (p6) fnma.s1 f10=f12,f11,f1 ;; // div
                    (p6) fma.s1 f7=f10,f10,f14 // div
                    (p6) fma.s1 f8=f10,f9,f9 ;; // div
                    (p6) fma.s1 f11=f7,f8,f8 ;; // div
                    fcvt.fx.trunc.s1 f6=f11 ;; // div
                    getf.sig r648=f6 ;; // div
                    add r14=r64,r0 ;; // stloc.0
                    ...
```

Listing 21 – Impact de la division sur le nombre d'instructions générées. A gauche le code CIL source, à droite le code généré par notre traducteur en langage machine EPIC.

Le traducteur 1:1 optimisé réduit sensiblement le nombre des instructions générées. En moyenne, celui-ci est réduit de 64,9% [entre -18,5% et -108,3%]. Sur les fonctions dont l'exécution compte pour près de 84% d'instructions non-arrêtées, nous générons moins d'instructions que le compilateur *JIT* et sommes proches du compte généré par le compilateur statique optimiseur (Figure 42). L'analyse des codes montre qu'au prix de passes additionnelles, notre traducteur pourrait encore améliorer son score par exemple en utilisant les instructions *SIMD*. Cela permettrait, notamment sur la fonction *key_schedule* de diviser le nombre des instructions par un facteur huit, ce qui nous rapprocherait alors de nouveau de la qualité de code du compilateur statique optimiseur.

En complément du nombre des instructions générées, nous évaluons également le nombre des cycles programmés pour l'exécution des instructions. Rappelons que ce nombre de cycle correspond à une programmation agressive des compilateurs et que lors de l'exécution, ce nombre peut être supérieur. En effet, les accès aux données en mémoire ou en mémoires cache peuvent introduire des cycles de décrochage – *stall cycles*. Nous constatons alors que le traducteur 1:1, de par sa nature, ne profite pas de l'architecture *VLIW* du processeur et programme une instruction par cycle d'horloge. Tout comme pour le nombre des instructions générées, nous constatons une variabilité en fonction du code traduit. Ainsi, lors de la génération de code additionnel – dans les mêmes cas que nous avons décrits précédemment –, nous pouvons souvent en programmer les instructions pour les mêmes cycles.

D'une façon générale, nous pouvons le faire dès que la pile virtuelle n'est pas référencée. Le traducteur 1:1 optimiseur effectue cette optimisation et obtient les meilleurs résultats lorsqu'il trouve des paires d'écriture et de lecture mémoires indépendantes qu'il peut programmer pour les mêmes cycles. Nous le voyons particulièrement bien avec la fonction *key_schedule* où nous pouvons systématiquement regrouper des séquences de deux lectures et de deux écritures dans deux groupes d'instructions indépendants. En moyenne, le nombre de cycles programmés est réduit de 81% [entre -43,9% et -108,9%]. Il est très certainement possible d'améliorer encore ce résultat – et gagner en *ILP* – en poussant l'analyse du code *CIL*, notamment en remplaçant l'algorithme de fusion des opérations contiguës par une version plus efficace.

Notre incapacité à repérer les candidats pour une exécution parallèle explique le déficit en termes d'*ILP* de notre traducteur 1:1 optimisé (Figure 44). En effet, l'*IPC* de notre code généré est de 1,43 à 2,97 fois inférieure à celui du code généré par le compilateur *JIT* et de 1,43 à 2,07 fois inférieur à celui du code généré par le compilateur statique optimiseur. En extrayant plus d'instructions indépendantes du code *CIL*, que nous pourrions programmer pour les mêmes cycles, nous devrions profiter pleinement du mode *non-bloquant* de la pile virtuelle. Nous n'avons toutefois pas entrepris cette recherche qui requiert l'implémentation de techniques de compilation avancées qui sortent du cadre de notre objectif initial.

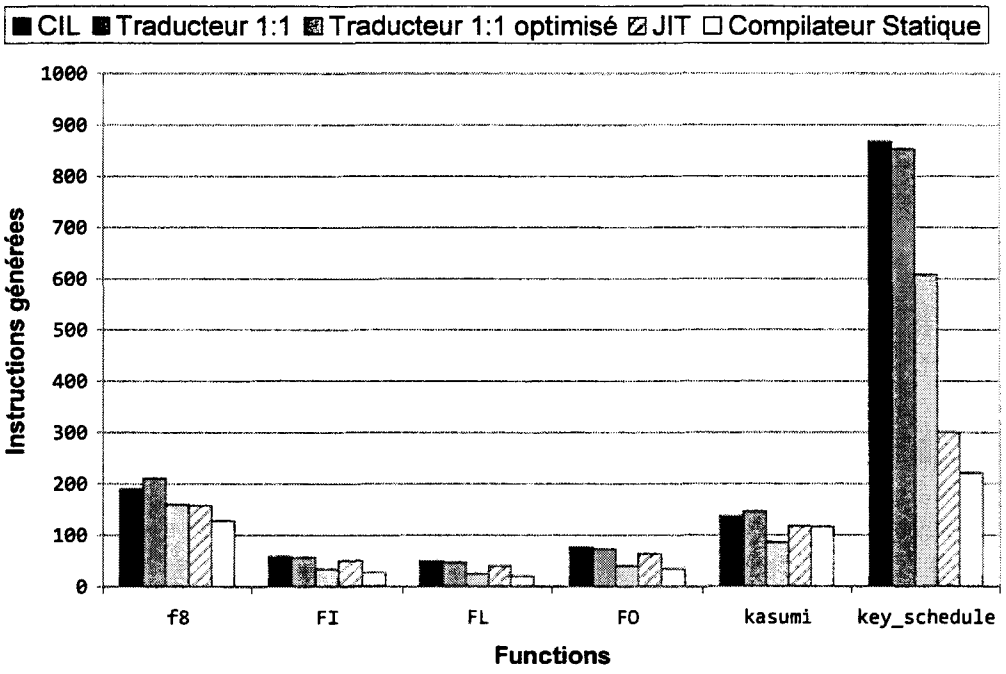

Figure 42 – Nombre d'instructions générées.

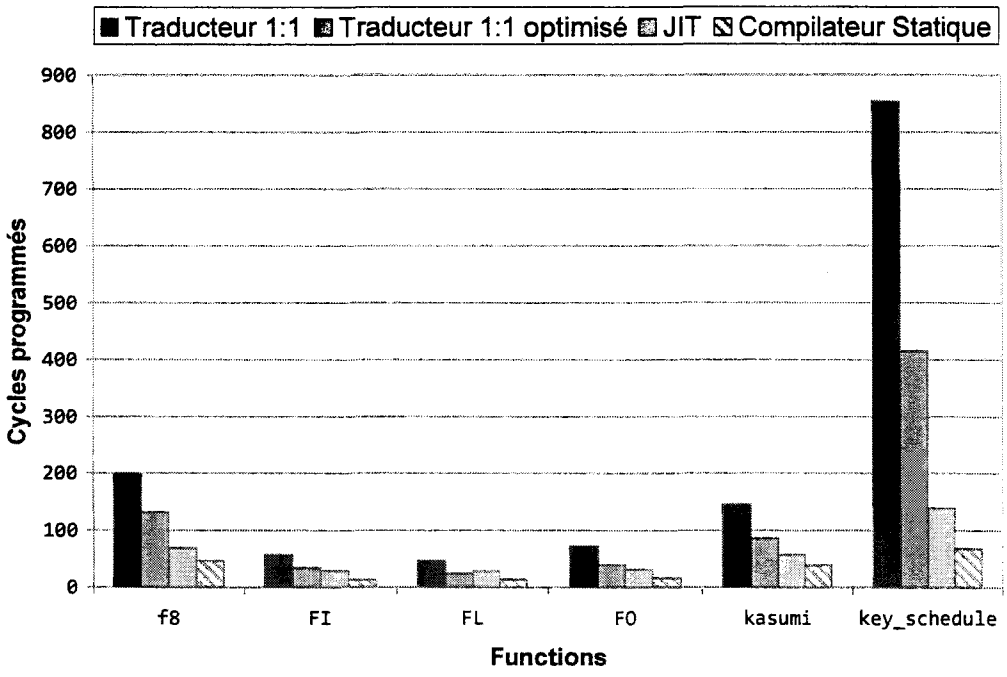

Figure 43 – Nombre de cycles programmés.

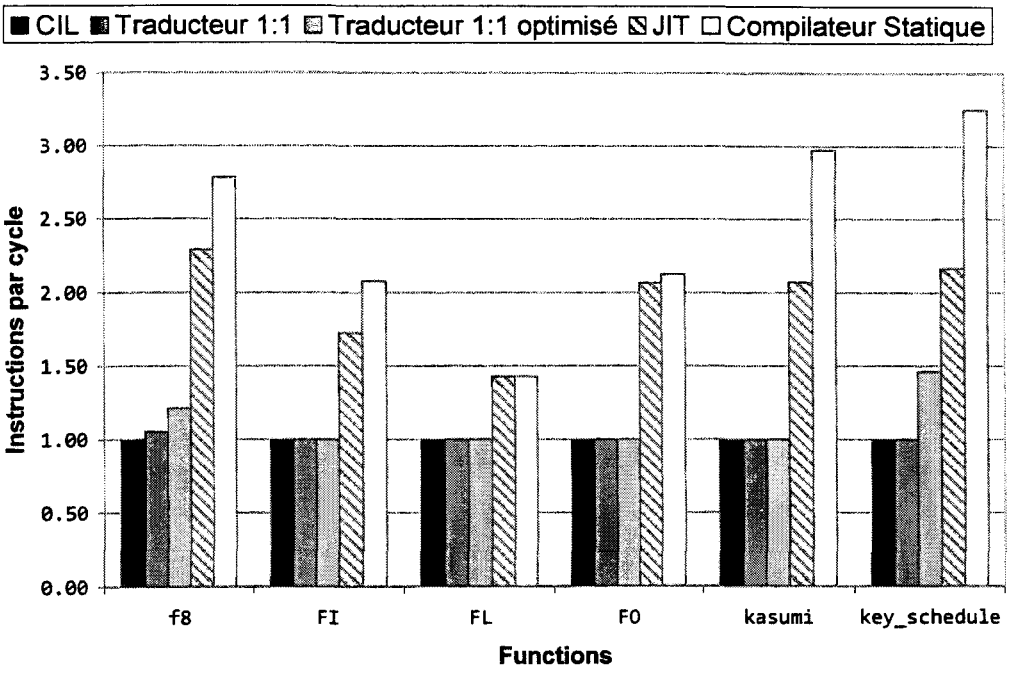

Figure 44 – IPC moyen – idéalement égal à 6.

c. Analyse de benchmarks

Après l'étude et l'analyse des résultats de l'utilisation de la pile virtuelle sur le code *Kasumi / f8*, nous avons entrepris d'étendre nos tests aux benchmarks *.NET XMLMark 1.1*, *NPerf* et *WSTest*. *XMLMark 1.1* est un benchmark de *parsing* XML créé par *Sun Microsystems* pour tester spécifiquement les performances de *Java versus .NET 1.1*. Depuis, ce benchmark est utilisé pour comparer différentes versions des machines virtuelles du marché. *NPerf* est un environnement de test créé pour faciliter la mise au point de benchmarks pour *.NET*. Nous avons utilisé *NPerf* avec le benchmark de démonstration qui effectue des recherches dans un dictionnaire. Enfin, *WSTest* a été initialement développé par *Sun Microsystems* et repris par *Microsoft Corporation*. Ce benchmark teste un ensemble de services Web en variant la taille d'objets *Simple Object Access Protocol* (*SOAP*). Nous avons utilisé la version *.NET* du benchmark (version 1.5).

Les résultats obtenus avec ces benchmarks confirment les résultats obtenus avec le code *Kasumi/ f8*. Nous constatons cependant certaines variations intéressantes à signaler. Là où *Kasumi* est un code monolithique, rédigé initialement en langage C et qui effectue un volume important d'opérations arithmétiques et logiques, *XMLMark 1.1*, *NPerf* et *WSTest* sont des applications orientées *Web* et utilisent massivement les technologies *objet*. Une des constantes avec ce type de code est qu'ils sont difficiles à optimiser. En effet, les compilateurs misent principalement sur leur capacité à *inliner* les méthodes qui ont une fâcheuse tendance à proliférer et n'exécutent qu'un faible nombre d'instructions. Souvent, il s'agit d'appels de fonctions. Ainsi,

en moyenne, *WSTest* exécute seulement 6 instructions par méthode. Les 286 méthodes de *NPerf* portent cette moyenne à 9,3 instructions. *XMLMark 1.1*, avec ses 51 méthodes, atteint en moyenne 34 instructions par méthode. En plus de l'*inlining*, la compilation guidée par des profils d'exécution ainsi que le regroupement des fonctions les plus souvent exécutées dans les mêmes pages de code par l'éditeur de liens donnent les meilleurs résultats. Ces techniques sont bien sur hors de portée de notre traducteur.

Notre traducteur se voit impacté fortement par ces mêmes difficultés. Les résultats obtenus sont donc moins bons qu'avec *Kasumi / f8*. Les Figure 45 à Figure 53 donnent nos résultats expérimentaux. Pour garder les figures lisibles, et puisque cela n'a d'ailleurs pas grand intérêt ici, nous avons supprimé le nom des méthodes. Ces dernières sont donc représentées par ordre alphabétique de leurs noms sur l'axe des abscisses.

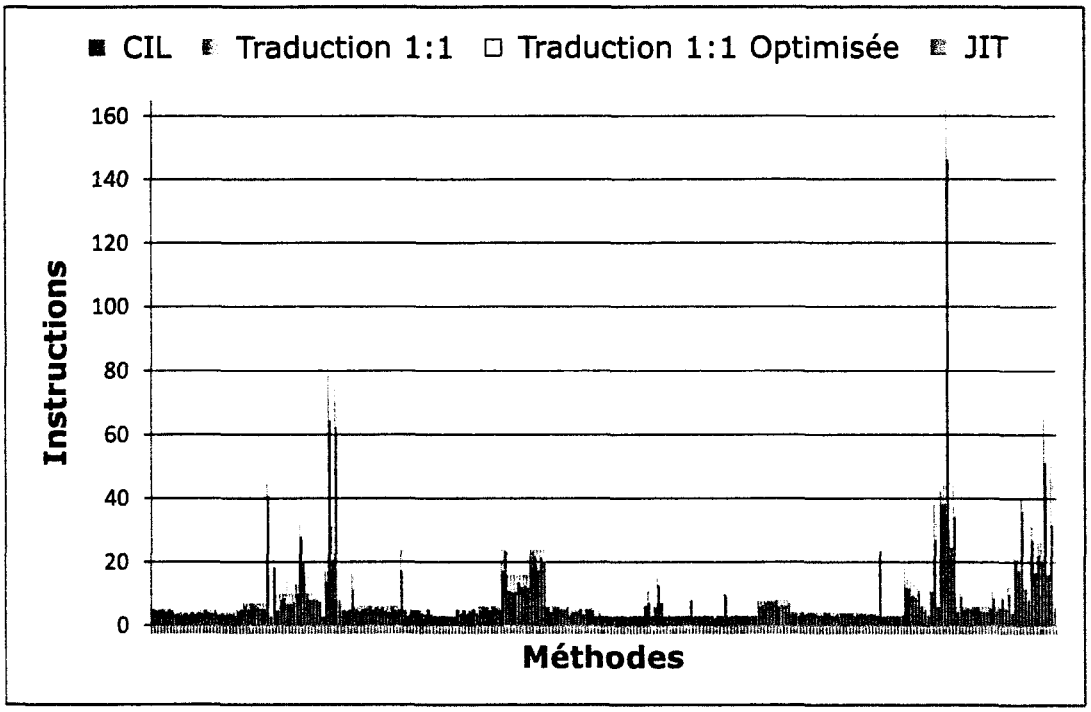

Figure 45 – Nombre d'instructions générées pour NPerf.

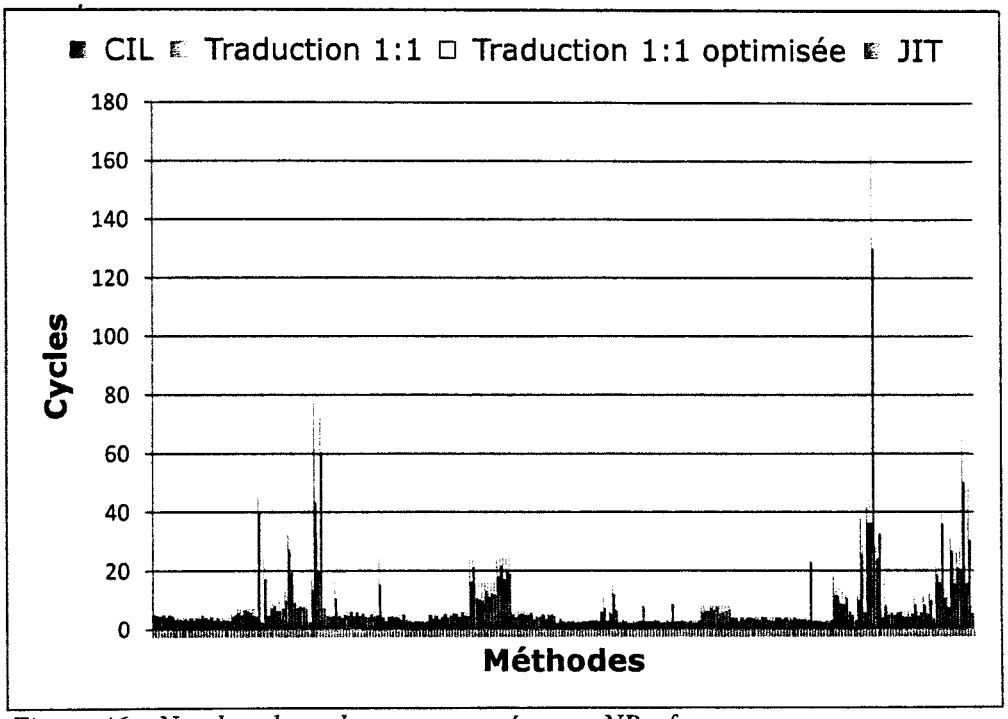

Figure 46 – Nombre de cycles programmés pour NPerf.

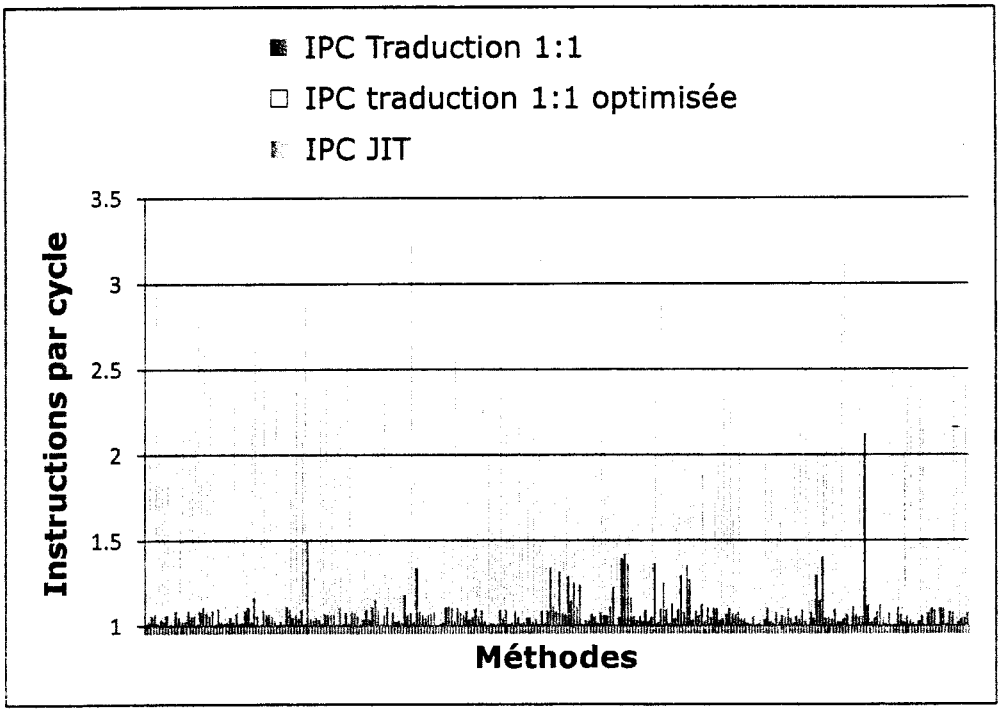

Figure 47 – IPC pour NPerf. L'IPC CIL est égal à 1.

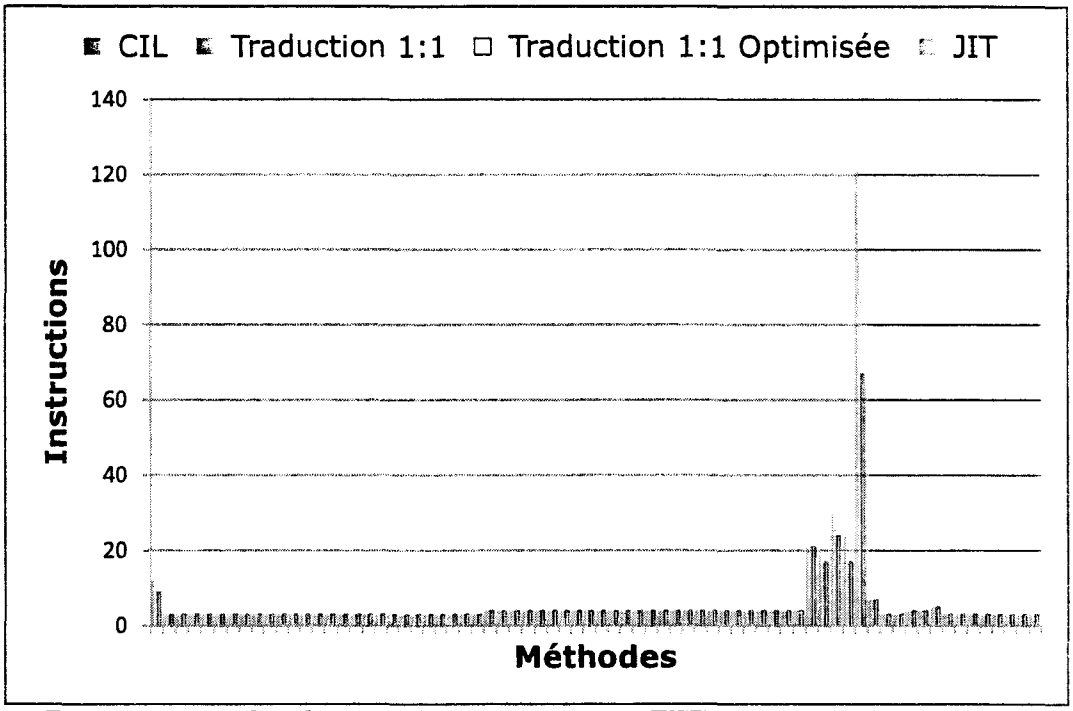

Figure 48 – Nombre d'instructions générées pour WSTest.

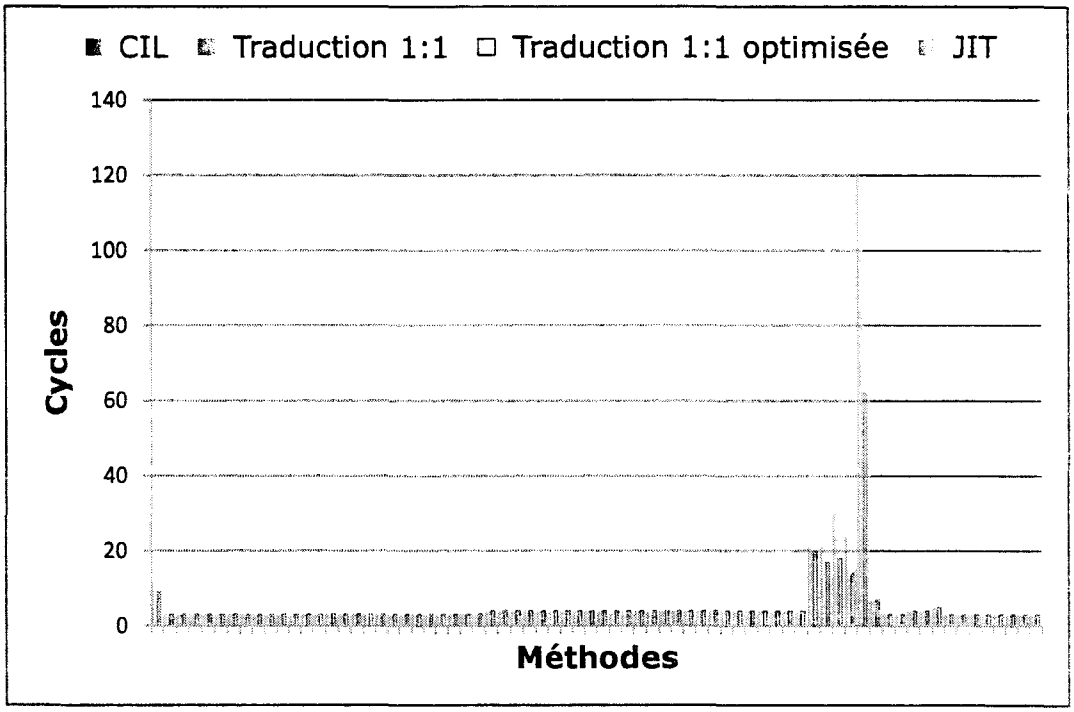

Figure 49 – Nombre de cycles programmés pour WSTest.

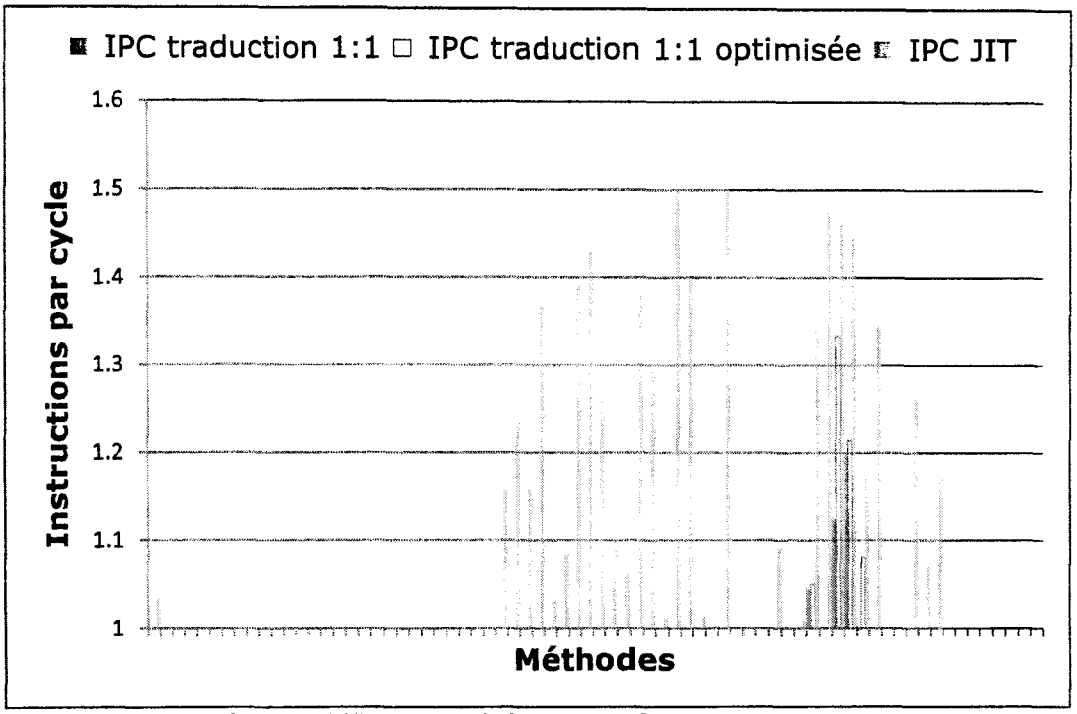

Figure 50 – IPC pour WSTest. L'IPC CIL est égal à 1.

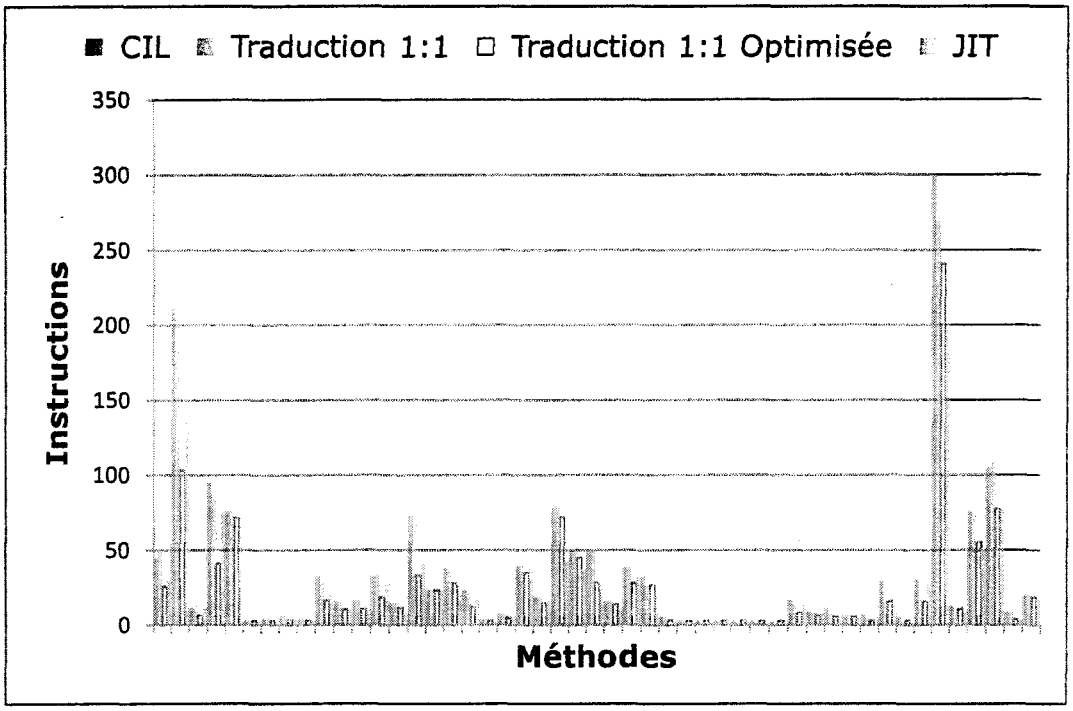

Figure 51 – Nombre d'instructions générées pour XMLMark 1.1.

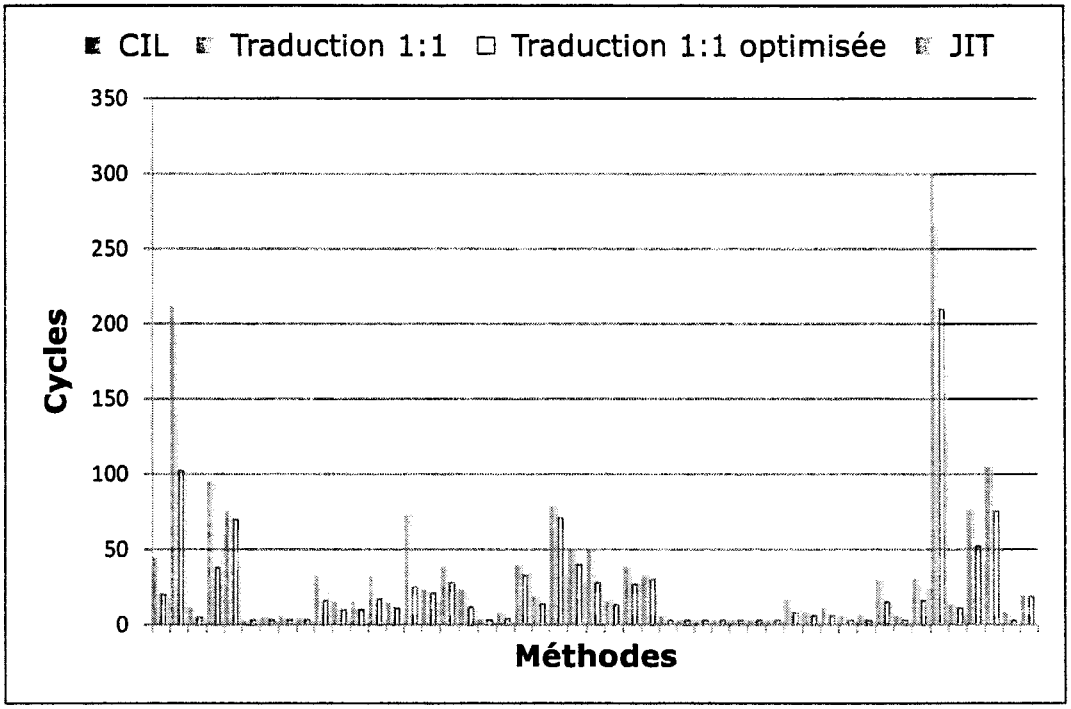

Figure 52 – Nombre de cycles programmés pour XMLMark 1.1.

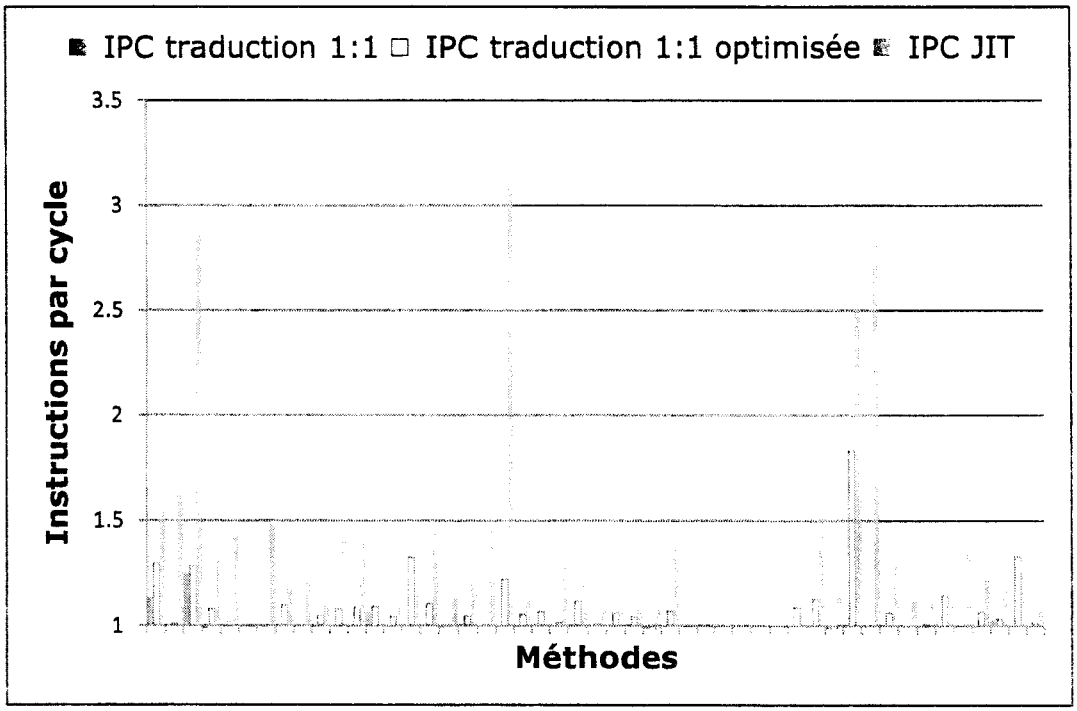

Figure 53 – IPC pour XMLMark 1.1. L'IPC CIL est égal à 1.

En guise de nombre d'instructions, nos performances sont inférieures à ce que nous avons constaté sur le code *Kasumi / f8*, avec un gain moyen de seulement 1% sur *WSTest*, 3% sur *NPerf*, de 12% sur *XMLMark 1.1 (% exprimé par rapport au binaire CIL)*. En revanche, si nous supprimons du compte les méthodes de moins de dix instructions, nous obtenons des gains de 14% sur *WSTest*, 12% sur *NPerf*, de 18% sur *XMLMark 1.1*. Ces résultats nous permettent d'affirmer que la nature même des langages *objets* rend l'optimisation du code difficile. En revanche, sur des codes dominés par les calculs et dont l'utilisation de la pile d'évaluation est plus soutenue, notre solution prend l'avantage.

Dans tous les cas, nous accusons un déficit du point de vue de l'*ILP*. Celui-ci est légèrement plus élevé avec ces benchmarks que sur le code *Kasumi / f8*. Nous expliquons cela par le fait que nous pouvons paralléliser les références aux objets et aux méthodes qui abondent dans *XMLMark 1.1*, *NPerf* et *WSTest*. Nous atteignons un *IPC* moyen de 1.07 sur *NPerf*, et 1.08 sur *XMLMark 1.1*. Les compilateurs atteignent pour leur part un *IPC* moyen de l'ordre de 1,5. Une analyse plus poussée du code *CIL* et l'utilisation plus systématique du mode *non-bloquant* de la pile virtuelle devrait cependant nous permettre de combler l'écart vis-à-vis des compilateurs *JIT* et statiques.

Pour ce faire, le traducteur binaire doit opérer en plusieurs passes. Lors de la première traversée du code *CIL*, les séquences d'instructions indépendantes doivent être marquées ainsi que leur action sur la pile mémorisée. Par exemple, nous pourrions appliquer la méthode développée *Wang* et *Yuen* (Wang, et al., 2006) pour leur processeur *Java TMSI* (*Tab-based Multi-Issue Semi In-Order*). En effet, pour alimenter l'architecture *VLIW* de leurs processeurs Java, les auteurs proposent un cache de registre mémoire conservant pour chaque instruction du *bytecode* des marqueurs (tags) qui décrivent leur action sur la pile. En recherchant les instructions indépendantes – en utilisant les marqueurs – ils parviennent à les programmer pour le même cycle. Les auteurs annoncent une capacité d'extraction d'*ILP* de 21 à 115%. Ce taux atteint 173% pour un encodeur audio, notamment grâce à la présence de nombreuses instructions dans les blocs de base (tout comme dans le cas de notre traducteur optimiseur). Nous proposons donc d'ajouter la génération de l'équivalent du cache de pile en mémoire et d'y conserver des marqueurs pour détecter les instructions indépendantes. Pendant la passe de génération de code, les instructions détectées comme indépendantes peuvent être programmées dans les mêmes groupes d'instructions. En plus de ce regroupement, le générateur devra également réordonner les instructions de chargement des données dans la pile de façon à ce qu'elles laissent celle-ci dans l'état attendu par les instructions indépendantes (et qui dépend comme nous l'avons vu de la position – ou *slot* – dans lequel l'instruction est encodée).

Enfin, une autre approche pour augmenter l'*ILP* consistera à rechercher le parallélisme de *thread* (*Thread Level Parallelism*) présent dans l'environnement *.NET* et de le transformer en *ILP*. C'est la solution retenue entre autre par le compilateur *JIT* et le processeur *MAJC* de *Sun Microelectronics* ou le processeur *JMTP* qui intègre un cœur généraliste secondé par de multiples *JTP* (*Java Thread Processors*). Dans les deux cas, le *JIT* est chargé de rechercher et d'isoler les *threads* à exécuter en parallèle. De tels *threads* indépendants pourraient être alors fusionnés et exécutés en parallèle en utilisant le mode non-bloquant de notre pile. Par exemple, nous pourrions fusionner deux threads dont les instructions seraient programmées séparément dans chacun des deux bundles que le processeur *Itanium 2* peut exécuter en parallèle.

7. Conclusions

Dans ce chapitre, nous avons montré qu'il était possible d'adapter la pile matérielle introduite au chapitre précédent pour une utilisation avec des environnements de développement modernes, tels que *.NET* ou *Java*. Nous avons démontré qu'en revoyant fondamentalement notre approche, nous pouvions proposer une pile virtuelle, maintenue à même les registres de l'architecture *EPIC*, et qui satisfait la contrainte de typage dynamique de la pile d'évaluation des machines virtuelles, telle que le *CLR* utilisé dans le cadre de notre étude. Nous proposons également une implémentation matérielle non intrusive de la pile virtuelle pour les processeurs de la famille *Itanium*. Enfin, nous avons montré que l'utilisation de la pile virtuelle simplifie considérablement le développement de compilateurs *JIT* ou *AOT* en éliminant notamment la phase d'allocation des registres, ce problème étant résolu dynamiquement lors de l'exécution du code. Nous avons ainsi introduit un traducteur de binaire *CIL* en binaire *EPIC* et montré qu'en appliquant un nombre limité d'optimisations élémentaires, nous pouvons raisonnablement améliorer la qualité du code généré lors d'un processus en simple passe. Pour autant, si nous obtenons des résultats similaires à ceux obtenus par les compilateurs sur les codes dominés par les calculs, nous n'arrivons pas combler l'écart sur les codes objets. Toutefois, l'amélioration et l'ajout de techniques d'optimisations avancées devrait nous permettre dans une phase ultérieure d'améliorer l'*ILP* du code généré, qui reste peut-être le point le plus faible de notre proposition.

8. Références

aJile Systems *Embedded Low-Power Direct Execution Java Processors* [En ligne]. - 1 Aout 2008. - 1 Aout 2008. - http://www.ajile.com.

Chaitin G. « Register Allocation and Spilling via Graph Coloring » [Conférence] // SIGPLAN82. - 1982.

Cooper K. D. et Dasgupta A. « Tailoring Graph-coloring Register Allocation For Runtime Compilation » [Conférence] // Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO'06). - New York, New York : [s.n.], 2006.

De Halleux Jonathan.« NPerf, A Performance Benchmark Framework for .NET » [En ligne] // The Code Project. - 1 Janvier 2004. - 1 Aout 2008. - http://www.codeproject.com/KB/architecture/nperf.aspx.

DTC Lightfoot 32-bit Java processor core. - 2001.

Garey M. R., Johnson D. S. et Stockmeye. L. « Some simplified NP-complete problems » [Conférence] // Proceedings of the sixth annual ACM symposium on Theory of computing. - 1974. - pp. 46-47.

Gough J. *Compiling for the .NET Common Language Runtime (CLR)* [Livre]. - [s.l.] : Prentice Hall, 2002.

Hoflehner Gerolf et al. « Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems » [Conférence] // 37th International Symposium on Microarchitecture (MICRO-37'04). - [s.l.] : IEEE, 2004.

Imsys. Imsys IM1101 [En ligne] // Imsys IM1101. - 1 Aout 2008. - 1 Aout 2008. - http://www.imsys.se/products/im1101.htm.

Intel Corporation. Intel® C++ Compiler for Itanium®-based applications. Version 9.1 Build 20061105. Package ID: W_CC_C_9.1.033 [Conférence].

Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual - Application Architecture* [Livre]. - [s.l.] : Intel Corporation, 2006. - Vol. 1.

Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual - Instruction Set Reference* [Livre]. - [s.l.] : Intel Corporation, 2006. - Vol. 3.

Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual - System Architecture* [Livre]. - [s.l.] : Intel Corporation, 2006. - Vol. 2.

Intel Corporation. Intel® VTune™ Performance Analyzer 9.0. Build:24052.

Intel Corporation. *Itanium® Software Convention and Runtime Architecture Guide* [Livre]. - [s.l.] : Intel Corporation, 2001.

Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool [En ligne] // PIN. - 1 Aout 2008. - 1 Aout 2008. - http://rogue.colorado.edu/pin/.

Intel Corporation. VTune Performance Analyzer. - 2008.

Kumar L.V. Nagendra. « JVM Implementation in FPGAs » [Rapport]. - Gachibowli, Hyderabad, India : International Institute of Information Technology, 2002.

Lidin S. *Inside Microsoft .NET IL Assembler* [Livre]. - [s.l.] : Microsoft Press, 2002.

Microsoft Corporation. Microsoft® .NET Framework 3.5.

Microsoft Corporation. Microsoft® C/C++ Optimizing Compiler Version 15.00.20706.01 for Itanium.

Mundlapudi Singh et Mundlapudi Bharath. « Performance Regression Testing Using Japex and WSTest, and Future Tips » [En ligne]. - 2007. - 1 Aout 2008. - http://java.sun.com/mailers/techtips/enterprise/2007/TechTips_June07.html.

Niar Smaïl, Lieven Eeckhout et De Bosschere Koenraad. « Comparing Multiported Cache Schemes » [Conférence] // International Conference on Parallel and Distributed Processing Techniques and Applications. - Las Vegas, NV : [s.n.], 2003. - Vol. 3.

O'Connor J. Michael et Tremblay Marc. « PicoJava-i: The Java Virtual Machine in Hardware » [Article] // Micro IEEE. - [s.l.] : IEEE, 1997. - 2 : Vol. 17.

Poletto Massimiliano et Sarkar Vivek. « Linear Scan Register Allocation » [Conférence] // ACM Transactions on Programming Languages and Systems. - 1999. - pp. 895-913.

Ragozin D. et al. « The CIL Hardware Processor: An Implementation of CIL Code Execution Engine », Technical report #5 for RFP2 Hardware CIL Processor project [Rapport]. - Nizhny Novgorod : Nizhny Novgorod State University, 2006.

Rogers Paul, Fadel Luiz et Ferreira Fernando. *zSeries Application Assist Processor (zAAP) Implementation* [Livre]. - [s.l.] : IBM RedBooks, 2004.

Schoeberl Martin. « JOP – a Real-time Java Processor » [Rapport]. - Vienna University of Technology, Austria : Institute of Computer Engineering, 2005.

Sneha M., Charanya D Sri et Bhuvaneswari R. Usha. « Hardware Support for .NET MicroFramework CLR Components » [Conférence] // hpic2006. - Anna University : Department of Computer Science, College of Engineering, Guindy, 2006.

Srinivasan.T Srinath S, Bhushan Vidhya.M et Parthasarathi Ranjani. « Implementation of .NET CLR on FPGAs » [Rapport]. - [s.l.] : Department of Computer Science, Anna University, 2005.

Sun Microsystems. Java Processor Core - Data Sheet. - 1997.

Sun Microsystems. XMLMark 1.1 . - 2006.

Tayeb Jamel, Niar Smail et Benameur Nasser. « An MSIL Hardware Evaluation Stack for EPIC » [Conférence] // 12th workshop on Interaction between Compilers and Computer Architectures (Interact-12), in conjunction with HPCA. - Salt-Lake City USA : [s.n.], 2008.

Third Generation Partnership Project. 3GPP TS 35.202 V3.1.1 // Technical Specification Group Services and System Aspects - 3G Security. - [s.l.] : 3rd Generation Partnership Project, 2001. - Vol. 2.

Wang Hai-Chen et Yuen Chung-Kwong. « Exploiting Dataflow to Extract Java Instruction Level Parallelism on a Tag-based Multi-Issue Semi In-Order (TMSI) Processor » [Conférence] // Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. - [s.l.] : IEEE, 2006.

Optimisation de l'efficacité énergétique

Résumé

Avec un coût de l'énergie sans cesse croissant, la réduction et le contrôle de la consommation électrique des systèmes informatiques haute performance et des serveurs deviennent une priorité pour l'industrie et les professionnels des technologies de l'information. L'objet de notre étude dans ce chapitre consiste à étudier/ démontrer comment améliorer l'efficacité énergétique des programmes en réduisant l'énergie électrique qu'elles ils consomment. Pour y parvenir, nous avons entrepris d'explorer des solutions logicielles, à utiliser de concert avec les mécanismes matériels, d'économie d'énergie développées par l'industrie, en commençant par les fondeurs de puces. Ce chapitre présente les premiers résultats de cette recherche que nous menons dans le cadre de notre activité chez Intel Corporation. *Attendues pour la fin 2009, les conclusions de ce travail devraient aboutir – si elles sont concluantes – à la mise en place d'un programme spécifique voué à l'optimisation de l'efficacité énergétique à l'attention des développeurs d'applications et des centres de calculs (2010).*

1. Introduction

Moduler la quantité d'énergie utilisée lors de l'exécution d'une application est et sera dans un futur proche une qualité souhaitée, sinon requise, pour les systèmes d'information (Congress, 2006). En complément de l'approche matérielle qui est privilégiée à ce jour par l'industrie afin de réduire la consommation électrique des serveurs, nous nous proposons d'aborder ce problème sous l'angle novateur du logiciel. Nous faisons ce choix car selon nous, le logiciel est en définitive un des éléments principaux du système informatique pouvant offrir aux utilisateurs un niveau de contrôle de l'énergie consommé. L'objectif étant d'associer *in fine* le matériel et le logiciel pour obtenir les meilleurs résultats possibles.

S'il est une catégorie d'utilisateurs qui est dès aujourd'hui fortement sensibilisée au coût croissant de l'énergie, c'est bien celle des administrateurs de fermes de serveurs. En effet, avec un coût annuel d'environ $2 par Watt et par an, le problème de l'efficacité énergétique des systèmes informatiques devient un problème tangible. Il est communément admis par les gestionnaires des centres de calcul et d'hébergement que le coût de l'électricité par an c est donné par l'Equation 1 ci-dessous, où w est le nombre de Watts consommés, h le nombre d'heures et p le prix du kilowatt par heure. Le prix annoncé de $2 est obtenu par le doublement de la valeur calculée par la formule (qui est d'environ $1). Ce doublement est dû à la prise en compte de l'énergie requise par la climatisation (*Heating, Ventilation and Air Conditioning – HVAC*) avec un ratio de 1 Watt de climatisation pour 1 Watt utilisé par les calculateurs. Les discussions menées avec nos partenaires qui disposent d'un centre de calcul, indiquent clairement que l'*HVAC* est le poste principal de leurs dépenses en énergie, juste avant celui de l'achat et la maintenance des serveurs. Le problème essentiel de l'*HVAC*, est que s'il est aisé de recycler des serveurs devenus obsolètes, il est difficile, voir impossible de refondre la

climatisation – et encore moins de « *pousser les murs* » (*Google, Pixar Animation Studios, etc.*). La valeur de $2 que nous donnons est valable en Oregon (USA) avec un coût au kilowatt par heure de $0,114 lors de la rédaction de ce rapport.

$$c = \frac{w.h}{1000}.p$$

Equation 1 – Equation communément utilisée pour évaluer le coût de l'électricité pour un centre informatique. La division par 1000 permet l'obtention du résultat en kWh, unité de facturation de l'électricité.

Un bon exemple de cet intérêt croissant auprès des entreprises pour minimiser le coût de l'énergie nécessaire à l'alimentation électrique et à la climatisation de leurs centres de calcul est la décision récente de *Google* d'installer son prochain centre d'hébergement sur les rives de la rivière *Columbia* en Oregon (Figure 54). En se positionnant stratégiquement à seulement quelques encablures des centrales hydro-électriques, le géant californien se garantit un coût de l'énergie avantageux. *Microsoft Corporation*, domicilié dans l'état de Washington (jouxtant l'Oregon), a fait ce même choix pour les mêmes raisons.

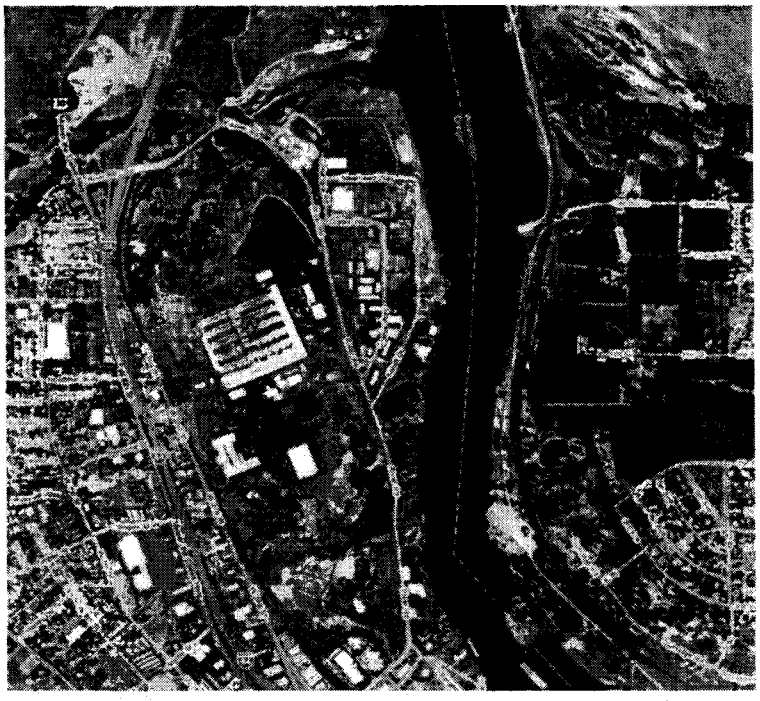

Figure 54 – Image satellite de l'emplacement du prochain centre de calcul en cours de construction que Google ouvrira en Oregon – USA – jouxtant la rivière Columbia séparant l'état de l'Oregon de Washington (en noir au centre de l'image). Cet emplacement stratégique permet au centre de calculs d'acheter son énergie électrique très bon marché car produite sur place. Source : Google Maps.

Parallèlement à cette réalité souvent douloureuse qui s'impose aux industriels, la sensibilisation de l'opinion publique à ce même problème est croissante. Ainsi, la notion de *Green Computing* gagne rapidement en visibilité et en notoriété. La première initiative du genre remonte à l'année 1992, lorsque l'*U.S. Environmental Protection Agency – EPA* – et l'*U.S. Department of Energy* ont initié le programme *Energy Star,* (EPA, 2008). Fondé sur le volontariat, ce programme récompense d'un label les équipements répondant à un cahier des charges édicté par l'*EPA*, dont le but est de réduire les émissions de gaz à effet de serre. Les premiers équipements à être ainsi estampillés ont été les ordinateurs personnels et leurs écrans. Le cahier des charges de ces derniers a d'ailleurs été amendé et renforcé en 2006. Depuis, plus d'une cinquantaine de classes d'équipements ménagers ont été définies dans le cadre du programme. Une classe *serveur d'entreprise* est actuellement en cours de définition sous l'*Enterprise Server and Data Center Energy Efficiency Initiatives* (Energy, 2008).

Ces programmes, dont le *Data Center Energy Efficiency Initiative*, couvrent l'intégralité de la chaîne de distribution de l'énergie, depuis la refonte du système de distribution du courant électrique vers les centres de calculs en lui-même (*DC Pro*), à l'amélioration du rendement des alimentations électriques (*Energy Start*, 80%+), en passant par des domaines essentiels que sont les télécommunications et le télétravail, la virtualisation, les clients légers, le stockage de l'information (disques *Solid-State, etc.*) et enfin la gestion des états de veille des composants électroniques (*Advanced Configuration and Power Interface – ACPI*).

Ces initiatives concernent surtout le matériel même si certaines technologies ont recours à une composante logicielle – essentiellement sous la forme de *firmware*. Ainsi, quasiment rien n'est proposé à ce jour du point de vue des logiciels applicatifs. Pis allé, les éditeurs de logiciels eux-mêmes ne sont pas ou peu sensibilisés à la question. En effet, pour eux, l'efficacité énergétique d'une application n'est pas un critère suffisamment différenciateur pour générer des ventes supplémentaire et pour y consacrer des ressources dédiées. Gageons cependant que cela va rapidement changer, notamment sous la pression exercée conjointement par les constructeurs d'équipements informatiques et les utilisateurs. Ainsi, si un module de rendu de contenu multimédia réduit de moitié l'autonomie d'un *Mobile Internet Devices – MID* de parson architecture logicielle male conçue, il est fort à parier que les utilisateurs réclameront sous peu ces heures d'autonomies perdues. Nous prévoyons que les premiers segments à être touchés seront ceux des équipements mobiles de télécommunication (du mobile au serveur Télécom) et les *Mobile Internet Devices – MID*. Les serveurs d'entreprises suivront très rapidement. Cette prise en étau des éditeurs par leurs clients a déjà fait preuve de son efficacité par le passé – notamment dans le secteur financier.

L'objectif de notre étude consiste à proposer une méthodologie d'optimisation des applications dans le but d'améliorer leur efficacité énergétique en réduisant et / ou en contrôlant leur budget énergétique. Pour ce faire, nous proposons dans un premier temps des outils pour mesurer la quantité d'énergie électrique consommée par un logiciel et pour la corréler à son activité. Dans un second temps, nous explorons et validons des transformations de code dans le but de réduire la quantité d'énergie utilisée. Nous avons classé ces transformations en deux groupes. Le premier groupe modifie le comportement de l'application, ce qui se traduit par une diminution contrôlée de la performance. Le second groupe n'altère pas le mode de fonctionnement du programme et s'accompagne en général par un gain de performance.

Notre méthodologie reprend celle de l'optimisation des performances. Cette approche itérative consiste dans un premier temps à mesurer l'énergie consommée par une application avec un ou plusieurs jeux de données significatifs. Cette mesure est la référence. En utilisant nos outils, nous recherchons les points chauds de l'application et appliquons les techniques de transformations que nous supposons être bénéfiques. Enfin, une fois les transformations de code effectuées, les gains énergétiques sont constatés – ou pas – par une seconde batterie de mesures à l'identique. Nous réitérons le processus jusqu'à l'obtention des résultats escomptés. Notons que l'optimisation de l'énergie consommée et celle des performances peuvent se mener en parallèle, ce qui selon nous devrait faciliter sa mise en œuvre auprès des programmeurs.

Précisions enfin que ce travail de recherche mené en interne par *Intel Corporation* nous a valu d'être invités à participer et à contribuer à l'initiative du *Green Grid* (The Green Grid Consortium 2008). En particulier, nos outils et méthodologies de mesure de l'énergie consommée et de sa corrélation avec le travail effectué par les applications, sont en cours d'évaluation par ce même comité. Nous pensons raisonnablement qu'ils seront retenus comme fondement du standard industriel que le comité est chargé d'édicter.

2. Mesure de l'énergie et de l'efficacité énergétique

« Vous ne pouvez pas gérer ce que vous ne pouvez pas mesurer. » Cet adage fondé sur le bon sens s'applique directement au sujet de notre étude. En effet, il n'existe aucune méthodologie industrielle standard pour mesurer la quantité d'énergie consommée par un système informatique pour effectuer un *travail utile*. Dans le cadre de notre étude, nous avons mis au point et promu auprès de nos partenaires industriels une méthodologie et les outils requis pour mesurer le *travail utile*, l'énergie consommée et pour corréler ces données au(x) logiciel(s) exécuté(s). Pour simplifier la lecture de ce chapitre, nous avons volontairement supprimé les détails techniques liés à l'implémentation des outils de mesures que nous y présentons. Ces détails sont regroupés dans l'annexe 3.

Nous allons à présent donner la définition des éléments essentiels à cette étude. Nous définissons la quantité d'énergie consommée par un logiciel comme étant l'énergie électrique – exprimée en Joules – nécessaire à l'ensemble des équipements informatiques impliqués dans son exécution. Elle représente l'intégration de la puissance consommée dans le temps. Par convention, nous n'intégrons pas la puissance lors des phases d'inactivité en début et en fin d'exécution des applications – ce qui explique le besoin d'instrumentaliser les logiciels de façon à ce qu'ils exposent leur état de fonctionnement (Annexe 3). Par analogie économique, nous appellerons indifféremment cette quantité le budget ou l'enveloppe énergétique. Nous adoptons ici une approche macroscopique en nous intéressant à l'ensemble du / des système(s) et en mesurant la puissance à la *prise électrique*. Cependant, il est possible d'affiner l'étude en ventilant le budget énergétique vers les composants les plus significatifs du système. Cette résolution accrue nous autorise alors à poser un diagnostique plus fin, ou du moins plus rapide, menant à l'identification des points énergétiques d'un logiciel.

Nous appelons point énergétique une plage d'adresses virtuelles dans le code qui peut être corrélé à une certaine quantité d'énergie – généralement significative. C'est l'équivalent du

point chaud (*hot spot*) qui pour sa part est corrélé à un nombre de cycles non arrêtés (*unhalted clocks*) et/ ou d'instructions non annulées (*retired instructions*). Les points énergétiques et les points chauds sont fortement corrélés (Annexe 3).

L'information qui nous intéresse dans cette étude est la quantité d'énergie nécessaire pour générer un résultat donné. Nous définissons cette donnée comme étant l'efficacité énergétique d'un couple (système, logiciel) et nous l'exprimons en joules par *résultat*. Par exemple, nous pouvons dire pour un système – et des conditions environnementales données –, qu'*il faut 20 kJ pour calculer une image de référence*. Nous voyons clairement la difficulté imposée par cette approche puisque l'espace des résultats est infini. En effet, il existe pour chaque application une infinité de résultats possibles. C'est d'ailleurs le problème essentiel qu'essaie de résoudre le *Green Grid* pour les centres de calculs en définissant l'*IT Productivity Metric*.

Nous avons ainsi convaincu ce même comité de ne pas céder à la facilité sur le long terme en utilisant des *benchmarks* en guise de *proxy* pour le *résultat* généré par les applications serveurs et de calcul, pour ensuite en dériver l'efficacité énergétique. Notre approche est résolument différente et s'attèle à mesurer – par instrumentation – le véritable travail effectué par les applications. Pour ce faire, nous avons défini une méthodologie et des outils logiciels pour standardiser l'export de compteurs architecturaux au sens large, et relatifs au travail effectué par les applications serveurs en particulier. En parallèle de ce travail de définition mené avec le *Green Grid*, nous testons en ce moment même notre méthodologie et notre *API* avec le câblo-opérateur *COMCAST* au sein de son centre *IPTV* de Denver dans le Colorado (*IPTV* désigne la diffusion de programmes télévisuels en utilisant le protocole réseau *Internet Protocol*).

Nous sommes conscients que cette approche de la mesure de l'efficacité énergétique peut surprendre certains acteurs de notre industrie qui sont habitués à étudier tous leurs problèmes sous le seul angle de la performance (les fondeurs de puces étant d'ailleurs les premiers d'entre eux). Ainsi, l'exemple précédent se transforme souvent en : *il faut 20 kJ pour calculer une image de référence avec une performance de 4000 pixels par seconde*. Cette approche est légitime, mais complique inutilement les comparaisons car elle ajoute une autre dimension (combinaisons du temps d'exécution et du travail utile déjà pris en compte) et peut rapidement mener au mieux à des interprétations involontairement erronées, au pire délibérément fallacieuses. Dans ce cas de figure, et puisque les performances sont un critère de comparaison, celles-ci ne doivent pas connaître de baisse non-voulue par l'utilisateur. Ainsi, baisser arbitrairement la fréquence d'horloge (qui réduit la consommation par un facteur cubique) n'est pas une solution retenue.

Enfin, nous imposons une règle essentielle, selon-nous, lorsqu'il s'agit d'étudier et de comparer l'efficacité énergétique de différents systèmes. Nous appelons cette règle, la **règle de l'invariance des résultats** et qui s'énonce ainsi : *les systèmes dont l'efficacité énergétique est comparée doivent impérativement générer à partir d'un même jeu de données un même résultat (égal bit-à-bit)*. La Figure 55 donne une interprétation imagée de cette règle.

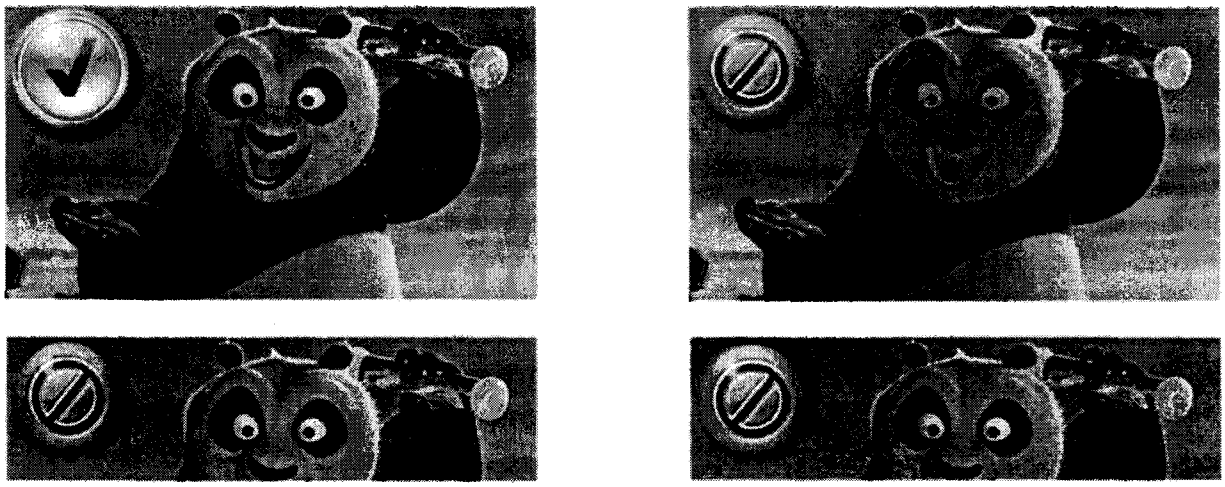

Figure 55 – Représentation en image de la règle de l'invariance des résultats qui édicte qu'un résultat partiel ou incorrect ne peut pas être utilisé pour effectuer des comparaisons d'efficacité énergétique. Ici, seule l'image calculée en haut à gauche peut-être acceptée. Source : DreamWorks Animation - DreamWorks Pictures SKG.

3. Les optimisations énergétiques

L'intersection majeure entre l'optimisation du code pour la performance et celle pour la consommation d'énergie est le temps oisif du système (*idle time*). En d'autres termes, en donnant l'opportunité au matériel de passer en mode oisif le plus vite et le plus longtemps possible, celui-ci peut réduire sa consommation électrique automatiquement et par son propre chef. Cela peut se faire par exemple en ayant recours aux mécanismes de positionnement de voltage et de fréquence des processeurs (technique largement utilisée dans les systèmes mobiles et connue sous le nom de *voltage scaling* ou *frequency scaling*). Dans la terminologie relative aux serveurs d'entreprise utilisée par *Intel Corporation*, cette technologie est connue sous le nom de *Demand-Based Switching – DBS*. Rappelons que nous ne prenons pas en compte les temps oisifs de début et surtout de fin d'exécution des applications lors de la mesure de l'énergie consommée. Lorsque nous parlons de temps oisif dans la suite de ce document, nous faisons référence à un temps oisif pendant l'exécution du code.

a. Les stratégies d'optimisation énergétiques existantes

Dans la suite du document, nous désignons par *système* le serveur utilisé pour exécuter l'application en cours d'analyse et d'optimisation, et par *composants* les éléments constitutifs de ce même système. Il s'agit par exemple de ses processeurs, sa mémoire vive, ses disques, ses interfaces réseau, *etc.*

L'*Advanced Configuration and Power Interface* (*ACPI*) (Hewlett-Packard Corporation - Intel Corporation - Microsoft Corporation - Phoenix Technologies Ltd. - Toshiba Corporation, 2006) définit pour un système informatique et ses composants des états de veille qui sont associés à un niveau de consommation électrique. Ainsi, G0 à G3 (Table 12) et S0 à S5 (Table 13) sont des états du système. Gi désigne un état global et Si désigne un sous-état d'un état global. Les états des périphériques sont notés D0 à D3 (*D* comme *Device*). Ceux des processeurs sont notés C0 à C3[3] (*C* comme *Central Processing Unit – CPU*). L'*ACPI* définit également des niveaux de performance du processeur P0 à Pn (*P* comme *Performance*). La Table 14 donne une description sommaire des niveaux Ci standards.

Etats	*Descriptions*
G0/S0	(Working) Le système d'exploitation exécute normalement les *threads* applicatifs. L'utilisateur peut activer des options de configuration de performance / économie d'énergie et les états énergétiques des périphériques sont maintenus par le système.
G1/(S1-S4)	(Sleeping) Les *threads* utilisateurs ne sont pas exécutés. Le système semble être hors tension et sa consommation est réduite par rapport à G0. Le système peut basculer en G0 sans nécessiter de *reboot*.
G2/S5	(Soft Off) Le système consomme au minimum et ni les *threads* utilisateurs. Ni les *threads* systèmes ne s'exécutent. L'état du système n'est pas maintenu par le matériel et un reboot est nécessaire pour sortir de ce état.
G3	(Mechanical Off) Le système est à l'arrêt. Seule l'horloge RT est active. La consommation électrique est nulle.

Table 12 – Description sommaire des états Gi de l'ACPI.

Etats	*Descriptions*
S1	état de veille à latence de réveil longue. L'état du matériel (processeurs, chipsets, etc.) est maintenu.
S2	Comme S1, à l'exception des caches des processeurs et du système qui ne sont pas maintenus. Le réveil de ce mode débute au vecteur *reset* du processeur.
S3	Tous les états systèmes sont perdus à l'exception de la mémoire centrale. Seuls quelques événements de configuration des processeurs et de la L2 sont assurés. Le réveil de ce mode débute au vecteur *reset* du processeur.
S4	Contexte de la plate-forme maintenu par le matériel, tous les périphériques sont éteints.

[3] Les architectures de processeur les plus récentes – comme le cœur *Nehalem* d'*Intel Corporation* – intègrent des niveaux plus profonds. Ainsi, les niveaux C4 à C7 sont disponibles. Les économies d'énergies sont rendues possibles en désactivant sélectivement des sous-ensembles du processeur, tels que tout ou partie des caches. Cependant, cela se traduit aussi par des latences de réveil plus élevées.

	La latence de réveil est la plus longue.
S5	Etat éteint (logiciel). Ce mode existe pour autoriser le reboot à faire la distinction entre un redémarrage depuis ce mode d'un redémarrage à froid (le BIOS ayant besoin de cette information pour ses initialisations).

Table 13 – Description sommaire des états Si de l'ACPI.

Etats	*Descriptions*
C0	Les instructions sont exécutées normalement, toutefois sous le contrôle du niveau de performance (Pi).
C1	Ce mode est obligatoire. Les instructions ne sont pas exécutées. Le processeur est mis dans ce mode *via* l'exécution d'une instruction dédiée (HLT pour l'architecture IA32). Le cache système est maintenu et le processeur peut quasi-immédiatement rebasculer dans le mode C0, généralement en réponse à une interruption.
C2	Ce mode est optionnel. Le processeur est dans un niveau de veille plus profond que le niveau C1. Le processeur assure la cohérence de ses caches et maintien son état architectural (exposé aux logiciels, tels que les registres, *etc.*). Cependant le processeur nécessite un temps plus long que le mode C1 pour sortir de ce mode.
C3	Ce mode est optionnel. Le processeur est dans un niveau de veille plus profond que le niveau C2. Le processeur maintient ses caches mais n'assure plus leurs cohérences. Le processeur nécessite beaucoup de temps pour sortir de ce mode. La sortie se fait généralement en réponse à une interruption ou un accès à la mémoire.

Table 14 – Description sommaire des états Ci des processeurs compatibles ACPI.

D'un point de vue macroscopique, l'optimisation énergétique vise idéalement à passer le plus rapidement possible et pour le plus longtemps possible du niveau système G0 (*Working*) à G1 (*Sleeping*). Nous excluons les niveaux G2 et G3 car les applications n'y sont pas exécutées et ne sont donc pas productives.

S'il est impossible d'agir directement sur le niveau de veille du système (Gi) depuis le logiciel – essentiellement depuis le mode utilisateur (*user land*) –, nous pouvons cependant positionner indépendamment celui de ses composants (Di, Ci), ce qui peut éventuellement se traduire par un changement de niveau de veille du système. Pour parvenir à un tel résultat, nous travaillons préférentiellement sur les processeurs.

Ces derniers participent à hauteur du quart du budget énergétique d'un serveur (Figure 56). Il s'agit également de la stratégie d'optimisation qui est la plus simple à mettre en œuvre tout en limitant la portée des modifications à apporter au code source. En revanche, comme nous le verrons par la suite, positionner Ci n'est pas toujours une stratégie gagnante.

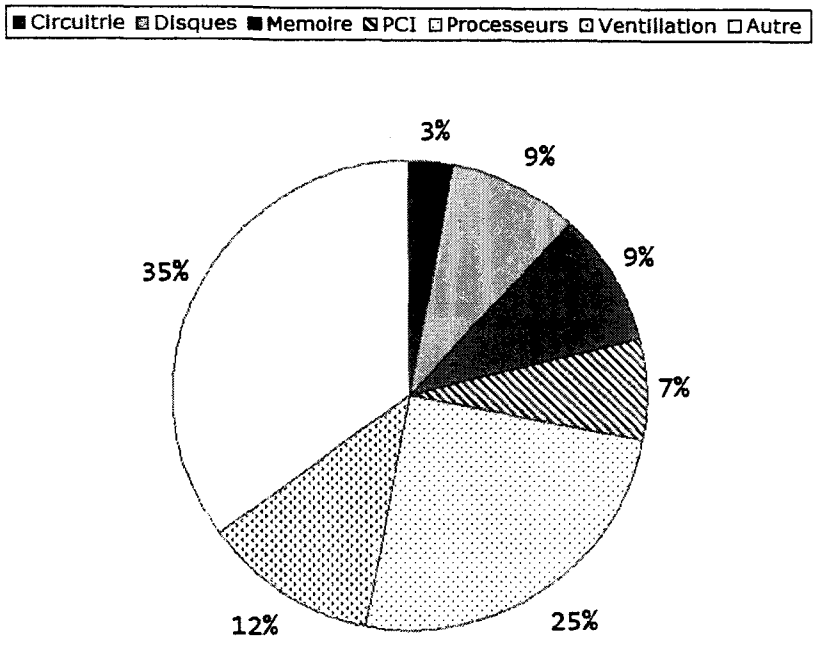

Figure 56 – Ventilation typique du budget énergétique pour un serveur biprocesseur (données obtenues sur un portfolio de plusieurs applications) Source : Intel Corporation.

Le mode `C0` – dans lequel les instructions sont exécutées par les processeurs du système – offre un niveau de contrôle de la consommation d'énergie additionnel *via* les niveaux `Pi` de l'*ACPI*. En développant un pilote spécifique, il nous est possible de programmer le circuit du contrôleur d'horloge de chaque processeur – ou de chaque cœur – de façon à définir l'intervalle de temps d'un cycle durant lequel l'horloge est active (*unhalted*) et celui pendant lequel elle est à l'arrêt (*halted*).

Ainsi, le niveau de performance `Pi` est déterminé par le nombre des intervalles de temps d'un cycle pendant lequel l'horloge est active (c). Le nombre maximum des intervalles pour un cycle d'horloge est donné par le nombre de bits alloué au champ `duty_width` (w) du registre `P_CNT` du *chipset*. Cette valeur est fixée par le fondeur du jeu de composants. Avec trois bits, huit intervalles sont possibles. Le niveau de performance est alors donné par l'Equation 2.

$$p = \frac{c}{2^w}$$

Equation 2 – Performance du processeur (p) en fonction du nombre d'intervalles d'activité (c) et de la résolution de duty_width (w).

Pour les périphériques, les disques principalement (qui représentent ~10% du budget énergétique d'un serveur – ou plus pour des applications utilisant intensivement les mémoires de masse), la stratégie d'optimisation est globalement plus passive qu'avec les processeurs. Elle consiste en effet à passer ces périphériques du mode D0 (*Fully-On*) à D1 puis D2, s'ils sont définis pour la classe de périphérique considérée.

b. Transformations de code

Voyons à présent comment les transformations de code permettent d'activer les mécanismes que nous avons introduits dans la section précédente. Nous avons ainsi réuni les transformations de code en deux groupes. Le premier groupe (groupe 1) est celui des transformations de code qui mènent à la réduction contrôlée des performances de l'application – en plus du contrôle de l'énergie consommée. Le second groupe (groupe 2) est celui des transformations qui se traduisent par une augmentation des performances du logiciel – en plus de la réduction de sa consommation électrique.

Pour chaque groupe, nous avons identifié un ensemble de transformations que nous souhaitons étudier dans le cadre de ce projet. Sans être exhaustive, cette liste offre une bonne base de recherche que nous prévoyons d'étudier d'ici à la fin de l'année 2009 (Table 15). Nous appliquerons chacune de ces techniques d'optimisation à une ou plusieurs applications de référence – de préférence des applications complètes et non pas des *micro-benchmarks*. Nous mesurerons les réductions – ou pas – de la quantité d'énergie consommée pour produire un même résultat. Si nous ne constatons pas de gains – pire, si nous mesurons une dégradation – alors nous proposerons des solutions pour soit éliminer les causes du problème, soit les contourner.

A partir de ces résultats, nous rédigerons un ensemble de recommandations que nous partagerons avec nos partenaires et les développeurs de logiciels. Cette phase de déploiement devrait prendre une forme proche de celle qu'*Intel Corporation* utilise pour promouvoir l'optimisation des performances.

Optimisations	*Groupe*	*Détails*
Autorégulation des performances	1	Offrir un mode de fonctionnement permettant la gradation des performances pour imiter le mode de gradation des performances introduit par le benchmark *SPEC Power* (en espérant reproduire les gains de puissance rapportés par les publications officielles).
Parallélisation	2	Réduire le temps d'exécution. S'assurer que le modèle de parallélisation est équilibré. S'assurer que le modèle de parallélisation permet aux threads exécutés par un même cœur d'être actifs / oisifs de façon synchrone.

Horloges	1 / 2	Limiter l'usage d'horloges ayant une résolution inférieure à 15 ms. S'assurer que les horloges inutilisées sont désactivées. Profiter des fonctions avancées des *API* systèmes pour gérer au mieux les horloges. Par exemple sous *Linux*, autoriser le *SE* à grouper les horloges (*round_jiffies*), *etc.*
Boucles	2	Limiter les petites boucles. Transformer les boucles d'attente active en boucle pilotées par événements / interruptions. Si cela est impossible, alors maximiser la durée entre itérations de la boucle d'attente active. Supprimer les boucles *spin-wait* - lorsque cela est possible.
Vectorisation	2	Utiliser les unités de calcul vectoriel du processeur. Utiliser des librairies optimisées et parallélisées.
Algorithmique	1 / 2	Implémenter des algorithmes plus efficaces, Implémenter et activer des algorithmes moins complexes / ayant une précision moindre dynamiquement. Eliminer les algorithmes récursifs.
Compilation	2	Optimiser le code pour l'architecture cible. Compiler le code par profil guidé pour compiler sélectivement le chemin chaud. Réaliser l'édition des liens de façon à optimiser l'emplacement des fonctions chaudes dans le binaire. Optimiser spécifiquement les fonctions qui intègrent des points énergétiques (Annexe 3).
Efficacité de données	1 / 2	Pré-charger les données, utiliser des tampons pour limiter l'accès aux supports de masse. Limiter ou fusionner les lectures et les écritures depuis les mémoires de masse. Minimiser les mouvements de données et rapprocher les données des processeurs. Réduire la fragmentation des supports de masse.

Table 15 – Liste – non exhaustive – des techniques d'optimisation que nous prévoyons d'étudier dans le cadre de notre projet d'ici la fin de l'année 2009.

I. Règles élémentaires de bonne conduite

Avant de nous intéresser aux transformations de code pour l'optimisation énergétique, il est important de commencer par s'assurer que l'application étudiée se comporte correctement d'un point de vue énergétique. Cela signifie qu'elle est à même de prendre en compte l'état énergétique de son environnement et de s'y adapter. Le minimum étant ici de répondre correctement aux messages de puissance du système d'exploitation (*SE*). Par exemple en répondant aux messages `WM_POWERBROADCAST` du système d'exploitation *Microsoft Windows*. Si les applications ne répondent pas à ces messages, alors elles empêcheront le *SE* d'activer ses modes d'économie d'énergie et ne pourra pas passer le matériel en mode `G1` (et plus). Toujours sous *Microsoft Windows*, Il est recommandé d'enregistrer l'application auprès du *SE* et de répondre à ses messages comme cela est édicté par la documentation (en utilisant par exemple la fonction `RegisterPowerSettingNotification` de l'*Application Programing Interface – API –* système). Ainsi, si l'application ne répond pas dans un délai de deux secondes au message `PBT_APMSUSPEND` – qui est diffusé par le *SE* juste avant de mettre l'ordinateur en veille –, alors l'application bloquera la mise en veille du système.

Pour conclure ce rappel de savoir vivre, il est conseillé à une application de signaler dynamiquement au *SE* ses besoins énergétiques – *via* le niveau de performance souhaité – et de ne pas omettre de réviser ses besoins à la baisse dès que cela est possible (fonction `SetThreadExecutionState` de l'*API* système de *Microsoft Windows*). Cette fonction permet aux *threads* d'une application d'indiquer qu'une ressource particulière lui est requise (par exemple l'écran pour un logiciel de visualisation *via* le paramètre `ES_DISPLAY_REQUIRED`). Ainsi cette ressource sera exclue de la mise en veille par le *SE* – même si par exemple les options choisies pour la mise en veille du moniteur l'exigent. Une fois la tâche du *thread* terminée, cette ressource n'est pas rendue automatiquement au contrôle du *SE*. Si le *thread* n'informe pas le *SE* qu'il a fini d'utiliser la ressource, alors celle-ci sera toujours active, augmentant inutilement la consommation d'énergie du système. Signalons enfin que la plupart des mécanismes de conservation de l'énergie sont de nature coopérative et que le *SE* ne dispose pas de moyens préemptifs dans ce domaine.

c. Transformations du premier groupe

Dans cette section, nous allons décrire deux transformations de code du premier groupe (qui rappelons-le correspond aux transformations qui peuvent se traduire par une diminution des performances de l'application optimisée) que nous avons étudiées à ce jour (Table 15).

- L'autorégulation des performances
- L'efficacité de données – Limiter ou fusionner les lectures et les écritures depuis les mémoires de masse

II. Contrôler la charge des processeurs

L'autorégulation de la performance s'applique bien aux processeurs qui, rappelons le, consomment d'après nos analyses entre 15% et 40% de la puissance électrique moyenne des serveurs (Figure 57). Les données de la Figure 57 ont été collectées en exécutant le *benchmark*

SPEC Power (Standard Performance Evaluation Corporation 2008) sur un serveur biprocesseurs à quatre cœurs (soit huit cœurs au total). Nous faisons ici souvent référence au *benchmark SPEC Power* pour deux raisons qu'il nous faut expliquer. La première de ces raisons est que *SPEC Power* est l'unique *benchmark* standard pour mesurer la puissance électrique moyenne consommée par un système à tester et pour mesurer la performance de l'application *Java* utilisée en guise d'application serveur générant la charge processeur. De ce fait, le *benchmark SPEC Power* est devenu *de-facto* la référence, et nous nous devions de le prendre en compte dans notre étude. La seconde raison, comme nous le verrons plus loin, est qu'il peut malheureusement introduire des attentes déraisonnables auprès des clients qui n'analyseraient pas suffisamment ce que fait le *benchmark*.

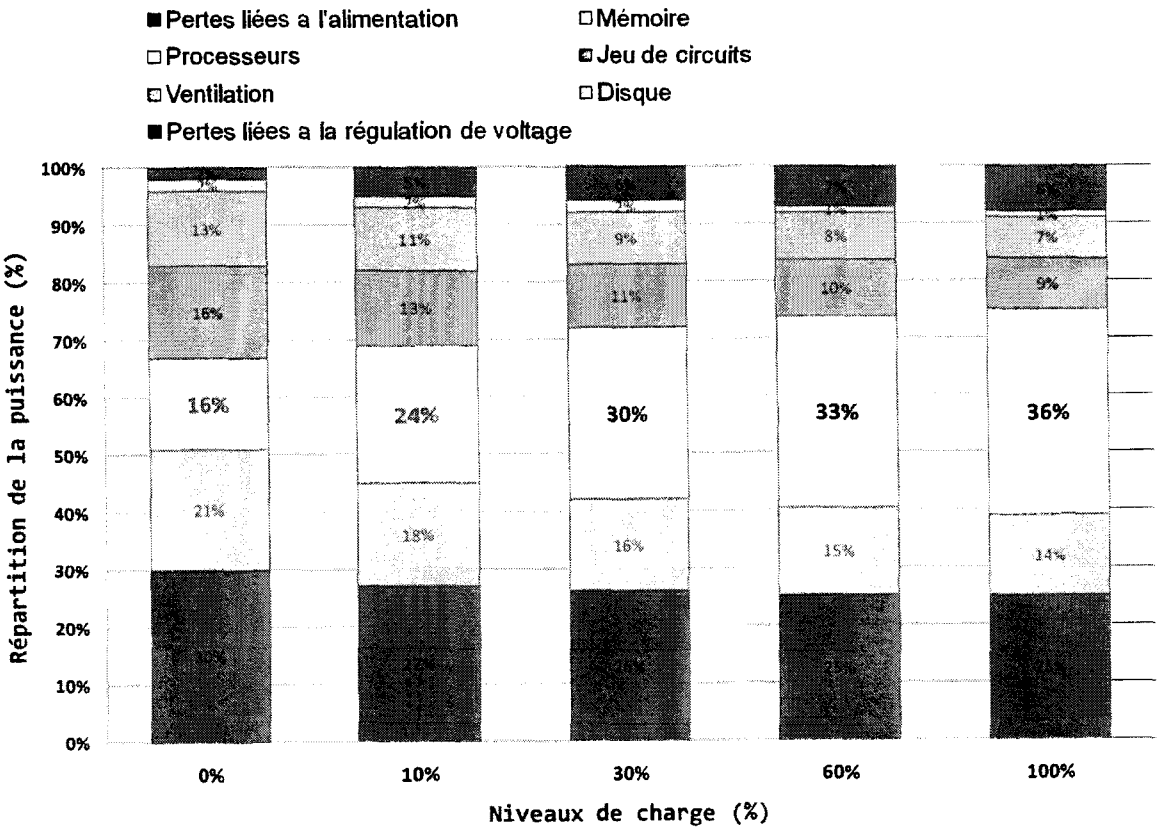

Figure 57 – Répartition de la puissance moyenne consommée entre les composants d'un serveur à huit cœurs exécutant le benchmark SPEC Power à différents niveaux de charge.

Ainsi, la transformation la plus remarquable – et remarquée – du premier groupe est la gradation des performances (*gradable workload*). Il s'agit de permettre à une application de fixer son niveau de performance – donné en pourcentage de sa performance maximale sur un système donné, exprimée entre 0% et 100%. Cette contrainte peut-être imposée par exemple par l'utilisateur ou bien par l'application elle-même en suivant une heuristique plus ou moins complexe pendant son exécution. La gradation des performances s'avère être la transformation la

plus intrusive que nous ayons testée jusqu'à présent – après la modification algorithmique bien entendu. Telle que plébiscitée par les clients, cette transformation vise à reproduire le comportement du *benchmark SPEC Power* (`SPECpower_ssj2008`) avec les applications dont on souhaite optimiser l'efficacité énergétique. En effet, ce *benchmark* définit une méthodologie standard implémentant onze niveaux de performance (de 100% à 10% par paliers de 10%, puis oisif actif (*active idle*) où l'application est initialisée et chargée en mémoire (Standard Performance Evaluation Corporation 2008). En outre, le *benchmark SPEC Power* exprime la performance du système testé en *Buisiness Operations per Second* (*BOPs*) telles qu'elles sont définies par le *benchmark SPEC Java Server* (`SPECjbb2005`) sur lequel il est fondé (SPEC, 2005). La Table 16 donne un exemple de résultats synthétiques générés par le *benchmark SPEC Power*.

Il est important de noter que le *benchmark SPEC Power* mesure la puissance moyenne consommée et non pas l'énergie consommée comme nous le proposons. De plus, le *benchmark* ne s'intéresse ni à la quantité de travail fournie pendant le test, ni à la reproductibilité des résultats. En ce sens, *SPEC Power* viole notre règle d'invariance des résultats. En effet, puisque l'exergue est mis ici sur les performances, le fait qu'à 10% de charge la quantité de *BOPs* effectuées pendant la durée fixe du test est de loin inférieure à la quantité de *BOPs* effectués à 100% (durant la même durée) passe le plus souvent inaperçue. C'est pour cette raison que nous disions en préambule que le *benchmark SPEC Power* introduit malheureusement une mauvaise approche du problème – et ce, d'autant plus qu'il est souvent décrit à tort comme un *benchmark* mesurant l'énergie !

Niveau de charge	*Moyenne des BOPs*	*Puissance moyenne (Watts)*
100%	185,456.00	115.00
90%	169,142.00	112.00
80%	149,666.00	108.00
70%	131,523.00	103.00
60%	112,618.00	97.10
50%	93,251.00	90.10
40%	76,105.00	83.20
30%	55,931.00	75.90
20%	37,346.00	69.90
10%	18,272.00	63.90
0%	0.00	57.50

Table 16 – Résultats SPECpower_ssj2008 pour l'IBM System x3200 M2.

Dès lors, les utilisateurs inattentifs exigent l'implémentation de la gradation des performances en croyant sincèrement que les gains de puissance moyenne publiés se traduiront sur leur facture électrique avec leurs applications en exploitation. En effet, puisque les serveurs d'entreprise sont généralement sous-exploités – avec des charges moyennes de l'ordre de 6% à 15% seulement (Energy, 2008) – les utilisateurs s'attendent à réduire par exemple leurs factures électriques de 65%. Nous avons démontré que cela est faux si la règle d'invariance des résultats est respectée (c'est-à-dire si le même résultat doit-être généré par les applications à différents

niveaux de charge). Ainsi, ce qui est effectivement constaté sur la puissance moyenne consommée ne se traduit pas sur le plan de l'énergie consommée.

Nous pouvons implémenter la gradation des performances de deux façons différentes. La première méthode consiste à utiliser un nombre variable de *threads* / cœurs de processeurs pour exécuter l'application. Ainsi, avec un système à huit cœurs, il nous est possible de placer le système aux niveaux de charge suivants : 0%, 12,5% ; 25% ; 37,5% ; 50%, 62,5% ; 75% ; 87,5% et 100%. Cette méthode est la plus simple à mettre en œuvre et ne nécessite pas la modification du code (à cet effet). La seconde méthode consiste à mettre périodiquement les *threads* de calcul en veille pour une durée déterminée lors d'une phase d'étalonnage (*calibration run*). Ces deux méthodes diffèrent par les mécanismes d'économie d'énergie du matériel qui s'active pendant l'exécution.

A ce titre, nous préférerons la seconde méthode qui conserve tous les *threads* de calcul et les cœurs actifs. Cette méthode repose essentiellement sur l'utilisation du *scheduler* du *SE* et de la résolution de son horloge. Elle est généralement assez simple à mettre en œuvre, mais nécessite en contrepartie la transformation conséquente du code source. Avec des applications d'entreprise (client / serveur par exemple) ayant des chemins de code de plusieurs dizaines de milliers d'instructions, une résolution d'horloge classique de 10 ms est suffisante pour reproduire les niveaux de performance du *benchmark SPEC Power* en utilisant la fonction `sleep`.

Cependant, dans le cas d'un *micro-benchmark*, ou si le développeur a opté pour une granularité fine pour le calcul de sa performance (à l'instar des pixels calculés par les logiciels de lancé de rayon tels que *PovRay – Persistence of Vision* – que nous étudierons plus tard en détail), une période de 10 ms s'avère être trop longue pour pouvoir atteindre tous les niveaux de charge voulus. Dans ce cas, il est toujours possible d'augmenter ponctuellement la résolution de l'horloge du *SE* pour l'application à 1 ms en théorie sous *Microsoft Windows*. En pratique, nous avons pu atteindre une résolution effective de 2 ms.

Nous avons appliqué la première méthode de gradation de la performance sur différentes applications. Tout en respectant la règle d'invariance des résultats, nous avons mesuré l'énergie consommée par les applications suivantes :

- *PovRay 3.7 beta 25* pour calculer *son bench*mark de référence ;
- *QlikView 8.1* de *QlikTech* pour calculer un tableau de bord décisionnel à partir d'une base de données en mémoire de 500 millions d'enregistrements (source *QlikTech*) ;
- *CATIA V5R17 SP4* de *Dassault Systèmes* pour calculer des contraintes mécaniques de deux modèles : *Cognac* (source *Dassault Systèmes*) et *Falcon* (source *Dassault Aviation*).

Notre système de test est une plate-forme *Tylesburg-EP* équipée de deux processeurs *Nehalem-EP* cadencés à 2.66 GHz (soit un total de seize cœurs en mode *Simultaneous Multi-Threading* activé – et *Non-Uniform Memory Access* désactivé). Les Table 18, Table 19 et Table 20 résument nos résultats expérimentaux pour nos trois benchmarks (et quatre jeux de données).

Dans tous les cas de figure, nos résultats montrent qu'en respectant la règle d'invariance du résultat, la gradation de performance est contreproductive, contrairement à ce que laisse imaginer le *benchmark SPEC Power*. Ainsi, nous constatons une augmentation de l'énergie consommée totale d'un facteur compris entre ~5 et ~8 entre une charge système de 100% et 12,5% (Table 17). Rappelons que le niveau de charge de 12.5% est obtenu ici en n'utilisant qu'un seul cœur.

	PovRay	QlikView	CATIA - Cognac	CATIA - Falcon
Energie 12,5% / 100%	6,56X	7,90X	6,31X	5,04x

Table 17 – Ratios entre l'énergie consommée à 100% et 12,5% de charge sur une plate-forme Tylesburg-EP de nos quatre applications de test.

Niveau de charge	Energie consommée (Joules)
6,25%	70.380,53
12,50%	56.207,40
18,75%	33.014,79
25,00%	29.915,22
31,25%	22.652,43
37,50%	21.555,09
43,75%	17.675,31
50,00%	17.216,06
56,25%	15.757,45
62,50%	15.405,36
68,75%	14.063,98
75,00%	13.621,88
81,25%	12.475,62
87,50%	12.320,57
93,75%	11.652,72
100,00%	11.145,66

Table 18 – Energie consommée par PovRay sur une plate-forme Tylesburg-EP pour le rendu du benchmark officiel.

Niveau de charge	Energie consommée (Joules)
6,25%	28.628,20
12,50%	28.894,58
18,75%	15.248,70
25,00%	15.287,74
31,25%	10.849,50
37,50%	11.015,91
43,75%	8.846,27
50,00%	8.679,25
56,25%	7.485,96
62,50%	7.638,45
68,75%	6.810,52
75,00%	6.831,18
81,25%	5.931,72
87,50%	5.977,80

93,75%	5.918,09
100,00%	5.834,53

Table 19 – Energie consommée par QlikView sur une plate-forme Tylesburg-EP pour l'analyse d'une base de données en mémoire de 500 millions d'enregistrements.

Niveau de charge	Energie consommée (Joules) - Cognac	Energie consommée (Joules) - Falcon
6,25%	70.380,53	270.173,87
12,50%	56.207,40	210.825,83
18,75%	33.014,79	126.476,62
25,00%	29.915,22	112.205,89
31,25%	22.652,43	85.907,52
37,50%	21.555,09	79.938,66
43,75%	17.675,31	67.013,84
50,00%	17.216,06	63.982,20
56,25%	15.757,45	57.759,18
62,50%	15.405,36	55.894,25
68,75%	14.063,98	51.050,69
75,00%	13.621,88	49.842,05
81,25%	12.475,62	45.794,34
87,50%	12.320,57	45.501,90
93,75%	11.652,72	42.295,14
100,00%	11.145,66	41.778,78

Table 20 – Energie consommée par CATIA V5R17 SP4 sur une plate-forme Tylesburg-EP pour le rendu des modèles Cognac et Falcon.

Nous avons ensuite appliqué la seconde méthode de gradation de la performance à deux applications : *Black-Scholes* (SunGard, 2008) (Black, et al., 1973) et *PovRay* que nous avons déjà utilisées avec la première méthode de gradation des performances. Il va de soi que nous ne pouvons appliquer ces transformations de code qu'aux logiciels dont nous disposons du code source. Ce n'est pas le cas par exemple de *CATIA V5R17 SP4* ou *QlikView*. C'est d'ailleurs pour cette raison que nous essayerons – dans la mesure du possible de travailler sur des logiciels libres. Les détails techniques de l'implémentation de la gradation sur le code de *Black-Scholes* (propriété de *SunGard*) sont donnés en annexe 3. Nous ne reproduirons ici que les résultats obtenus avec *PovRay*.

Avec *PovRay*, une résolution d'horloge de 2 ms ne nous permet malheureusement pas d'atteindre 80% des onze niveaux de performance édictés par le *benchmark SPEC Power*. Nous avons donc cherché des moyens alternatifs pour suspendre l'exécution des *threads* de calculs. La Figure 58 montre les deux options que nous avons, ainsi ajoutées à *PovRay*. La première option – image de gauche – consiste à utiliser `sleep` avec des durées inférieures à 10 ms [4]. La seconde

[4] Bien que l'horloge multimédia (*High Precision Event Timer – HPET*) offre une résolution de l'ordre de la µs, seuls les systèmes d'exploitation *temps réel* offrent des routines `nanosleep` ou `microsleep` fonctionnelles.

option – image de droite – consiste à introduire un nombre donné d'instructions `pause` au cœur de la boucle principale des *threads* de calcul. L'instruction `pause` qui suspend l'exécution de la prochaine instruction – pour une durée de temps spécifique à chaque processeur – est notre meilleur atout pour implémenter la gradation de noyaux de calculs très rapides.

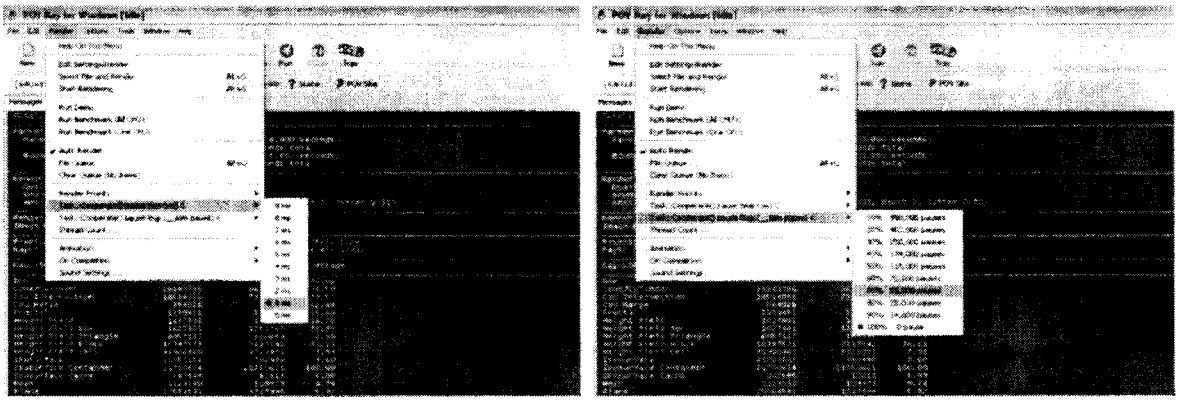

Figure 58 – Options de gradation de la performance que nous avons implémentées dans PovRay pour activer l'autorégulation des performances.

La Table 21 résume nos résultats expérimenteaux obtenus par l'injection d'instructions `pause` et par appel à `sleep`. Notre système de test pour ces mesures est une plate-forme *Stoakley* équipée de deux processeurs *Harpertown* cadencés à 3.20 GHz (soit un total de huit cœurs). Notons que nos tests ont été menés sur une période de près de six mois. En outre, ces travaux participent aussi à l'effort de promotion des nouvelles architectures d'*Intel Corporation*. Ainsi, nous privilégions pour nos tests les plates-formes les plus récentes. Or sur une période de six mois, nous avons souvent déjà changé de plate-forme. Il nous est ainsi très difficile de conserver et de tester de vieilles plates-formes. Ce qui explique que les systèmes utilisés entre les deux méthodes de gradation diffèrent ici. Nos résultats montrent une fois de plus qu'en respectant la règle d'invariance du résultat, la gradation de performance est contre-productive. Nous mesurons ainsi une augmentation de l'énergie consommée d'un facteur ~10 entre une charge système de 100% et 10%.

Niveau de charge	*Energie consommée (Joules) pause*	*Energie consommée (Joules) sleep*
10%	190.764,74	
20%	88.522,74	82.696,13
30%	62.441,80	
40%	49.664,10	
50%	39.103,86	
60%	31.668,03	
70%	27.954,26	

Dès lors l'usage de routines telles que `QueryPerformanceCounter` se traduit inévitablement par une attente active (*spin loop*) qui entraine irrémédiablement l'annulation de tous les mécanismes de mise en veille du matériel.

80%	24.104,11
90%	21.719,30
100%	19.678,34

Table 21 – Energie consommée par PovRay sur une plate-forme Stoakley pour le rendu du benchmark officiel. Les données manquantes dans la colonne de droite représentent les niveaux de charge que nous ne pouvons pas atteindre. Ainsi, seul le niveau de 20% est atteignable – en plus du niveau de performance nul. Cela est dû au problème de résolution d'horloge et de qualité des API du SE.

Ces mauvais résultats – en comparaison de la première méthode – sont dus au fait que l'*ES* ne dispose pas de fonctions de mise en veille des *threads* qui soient fonctionnelles avec des durées inférieures à 10 ms, et aussi au fait que l'instruction `pause` est inadaptée pour ce que nous souhaitons faire dans le cadre de cette expérience.

Plus fondamentalement, ces mauvais résultats dans leur ensemble sont imputables à la consommation électrique toujours trop élevée du système lorsqu'il est oisif (nous avons effectué des mesures sur différents systèmes et tous exhibent les mêmes problèmes. Nous avons pris la décision de ne sélectionner que ceux de la plate-forme la plus efficace). Ainsi, si le matériel pouvait réduire la consommation électrique de tous ses composants – idéalement à zéro Watt – et dans un temps idéalement nul, alors la gradation des performances pourrait avoir sa place dans la panoplie des outils pour l'optimisation énergétique – pour peu que l'utilisateur soit prêt à attendre plus longtemps pour d'obtenir le résultat des calculs, ce qui n'est pas une hypothèse déraisonnable.

Ce cas de figure montre bien que le logiciel dispose d'une longueur d'avance sur le matériel dans la mesure où il peut déterminer quand il souhaite placer le système en veille, là où le matériel est assujetti à une inertie inévitable. Ainsi, nous proposons l'ajout d'une nouvelle instruction à l'*ISA IA32* pour palier au problème. Cette instruction, que nous proposons d'appeler `sleep`, prendrait un argument *registre* ou *immédiat* qui indiquerait la durée de la mise en veille requise. Lorsqu'elle serait exécutée, elle viderait le pipeline et sauvegarderait l'état architectural du processeur avant de placer le cœur dans un état de veille profond (`C7` ou plus). Cela s'avère être suffisant pour peu que la latence de réveil (*resume*) peut être non-déterministe.

Enfin, l'exécution de l'instruction pourrait être utilisée au niveau de la plate-forme pour signaler à l'ensemble du système la possibilité de mise en veille (pour une durée connue). La faisabilité de limplémentation de l'instruction sleep est en cours de discussion avec les architectes. Passé le cap de l'acceptance, la première étape consistera à simuler l'instruction sur nos simulateurs pour vérifier le bien fondé de cette nouvelle instruction. Nous sommes en train de soumettre notre proposition au comité architectural d'*Intel Corporation*.

III. Contrôler l'utilisation des disques

Le taux d'utilisation des disques est très variable d'une application à une autre et d'un jeu de données à un autre. La stratégie d'optimisation énergétique que nous évaluons dans cette section consiste à limiter l'usage des disques – ou plus précisément à en concentrer et réordonner les accès dans le temps – lorsque cela est bien entendu possible. Au final, notre objectif est ici

encore de donner aux disques la possibilité d'activer leurs mécanismes de mise en veille pendant l'exécution du programme optimisé (autonomes ou commandés par le *SE*).

Deux caractéristiques physiques des disques justifient ce choix. Il s'agit de la durée de la mise en rotation (*spin start time*) et le temps moyen nécessaire au positionnement des têtes de lecture (*seek time*). Bien que ces caractéristiques soient propres à chaque type de disque, il est possible d'affirmer que la mise en rotation d'un disque est très coûteuse en temps et en énergie en comparaison d'une lecture de données séquentielle – 4000 ms pour la mise en rotation contre 12 ms pour le positionnement des têtes de lecture sur nos disques de test. En termes d'énergie consommée, la première opération requiert ~2,14 fois plus d'électricité (Krishnann et De Vega 2008). Nous proposons donc de limiter le nombre des mises en rotation des disques et de rationaliser celui des lectures et des écritures.

La première transformation consiste à lire les données et à les placer par anticipation dans des tampons en mémoire. Le cas extrême consistant à utiliser un disque virtuel, ce qui s'avère être raisonnable avec un adressage sur 64-bit démocratisé et des mémoires vives de plus en plus denses. Nous avons ainsi présenté au public lors de *l'Intel Developer Forum* (Août 2008 – San-Francisco) un serveur à base de processeurs *Nehalem* et de mémoire vive *MetaRAM* DDR3 offrant une mémoire vive de 144 Go et exécutant *QlikView* avec une base de données en mémoire de 100 Go. En effet, l'empreinte énergétique de la mémoire vive est inferieure à celle des disques durs fortement sollicités. Les technologies permettant de mettre sélectivement les composants de mémoires en veille en fonction de leur utilisation devraient encore creuser la différence.En complément de la lecture des données par anticipation, la seconde transformation vise à lire celles-ci par blocs de plus de 8 Ko. *Krishnann* et *De Vegas* montrent ainsi qu'il est environ quinze fois plus coûteux de lire un fichier par octet que par blocks de 64 Ko. En revanche, le gain énergétique s'estompe nettement avec des blocs de plus de 16 Ko.

Tailles de bloc	*Energie consommée par le disque dur (joules)*
1 Ko	49.54
4 Ko	48.82
8 Ko	47.63
16 Ko	46.73
32 Ko	46.26
64 Ko	46.66

Table 22 – Energie consommée par le disque dur test en fonction de la taille des blocs. Source : les auteurs.

La complexité de mise en œuvre d'une telle transformation s'échelonne du simple changement des paramètres des appels systèmes d'entrée-sortie, à la reconfiguration des disques et la gestion exclusive d'un large espace du disque par l'application (à l'instar des *blob* des gestionnaires de base de données). Cette dernière option offre également l'avantage de contrôler depuis l'application la fragmentation des disques. Si elle est faibles, alors cela diminue d'autant la consommation électrique. Un gain d'un facteur 2,5 a été démontré (Krishnann et De Vega

2008). De son côté, *Executive Software* – éditeur du défragmenteur de disques *Diskeeper* – rapporte une réduction d'énergie de 14% sur une batterie d'une dizaine d'applications (Executive Software 2008). Nous n'avons pas encore le résultat de cette étude sur nos plates-formes serveurs. En effet, nous sommes en train de développer une méthodologie et les outils pour fragmenter un disque dur de façon déterministique. Cela est impératif pour produire des images de disques ayant des taux de fragmentation connus (10%, 20%, etc. 100%). Ces images seront par la suite conservées et reproduite lors de chaque test. Une fois que notre environnement de tests sera en place, effectuer nos mesures ne pausera pas de difficultés particulières (cela s'applique aussi aux transformations que nous présenterons par la suite dans cette section). Nous souhaitons au final obtenir des résultats plus précis que ceux de *Krishnann et Al.* et *Executive Software.*

La troisième transformation que nous proposons d'appliquer consiste à différer les entrées-sorties. Et ce, plus particulièrement lorsque les disques sollicités sont dans un état de veille. Ainsi, sous *Microsoft Windows*, l'application peut utiliser la routine `DevicePowerState` pour connaître l'état de veille d'un disque, et ne le solliciter que lorsqu'il est dans l'état `D0`. Cette transformation peut-être avantageusement combinée avec la précédente en créant un service d'entrées-sorties géré par des *threads* dédiés aux opérations d'entrée et de sortie (*E/S*).

La quatrième transformation consiste à effectuer des entrées-sorties asynchrones plutôt que synchrones. En effet, il est possible de différer l'attente de l'achèvement d'une entrée-sortie à un *thread* spécialisé. Le *thread* ayant initié l'entrée-sortie peut dès lors soit entrer en veille – ce que nous recommandons – soit effectuer des tâches annexes, pour peu qu'il sache gérer l'asynchronisme. Le *thread* chargé de l'entrée-sortie sera lui mis *de-facto* en veille par le *SE dés que l'opération d'E/S est terminée*. Toujours à ce chapitre, lorsque la technologie sous-jacente le permet, nous conseillons l'utilisation des *Tagged Command Queuing – TCQ* (comme les technologies éponymes de *SCSI* ou bien le plus récent *Native Command Queuing – NCQ* de l'interface *SATA*). En effet, ces technologies laissent aux disques le loisir de réordonner certaines des requêtes reçues pour minimiser les déplacements des têtes de lecture. Cela est d'autant plus bénéfique que les accès aux disques sont majoritairement aléatoires.

Enfin, nous rappelons qu'il est important d'arbitrer les accès disques des *threads*. En effet, si ces derniers ne se concertent pas, alors le phénomène de *disk trashing* peut apparaître rapidement. Cette fois encore, un service de gestion centralisé des entrées-sorties intégré au logiciel à optimiser peut avantageusement consolider les opérations de lecture et d'écriture. C'est la solution la plus efficace, et aussi la plus simple à implémenter. En effet, si le modèle du service dédié n'est pas retenu, alors il faut identifier tous les sites d'entrées-sorties et rechercher ceux qui sont les mieux adaptés au modèle de parallélisation de l'application. A cela s'ajoute la difficulté d'implémentation d'une telle transformation dans le code de l'application.

Mises bout a bout, ces transformations permettent une réduction de l'énergie consommée – mesurée pour l'ensemble de la plate-forme – de l'ordre de 10%. Il s'agit là de nos premiers résultats obtenus avec un encodeur MPEG4 en temps en réel, mais dont l'implémentation souffre de plusieurs erreurs de programmation. Une réécriture du code devrait engendrer des gains supérieurs. Nous fondons cet optimisme sur les travaux de *Krishnann* et *De Vegas* qui rapportent

des réductions de consommation d'énergie pouvant atteindre 30% sur des applications multimédia.

d. Transformations du second groupe

Le second groupe de transformations est celui des optimisations de performances en général. En effet, plus nous réduisons le temps d'exécution d'une application, et plus nous donnons d'opportunités au matériel d'entrer en veille et pour une durée plus longue, pendant l'exécution du programme. Ainsi, bien que la puissance moyenne consommée pendant l'exécution du code optimisé soit plus élevée, le gain énergétique est rendu possible grâce à la diminution du temps d'exécution. L'énergie étant l'intégration de la puissance dans le temps.

Nous avons évalué dans cette section les deux optimisations capitales de cette catégorie. La première optimisation consiste à paralléliser l'application en utilisant un nombre plus important de *threads*. Cette solution a d'autant plus de sens que les processeurs multi-cœurs sont aujourd'hui une commodité disponible pour l'ensemble des plates-formes. La seconde optimisation est très classique puisqu'elle contribue à améliorer la performance d'un *thread*. Pour ce faire, nous appliquons la méthodologie itérative d'optimisation des performances utilisée par *Intel Corporation*.

Ces deux optimisations ne sont d'ailleurs pas mutuellement exclusives. Cependant, si la seconde permet au mieux un doublement de la performance, la première permet en théorie[5] des gains de performance linéaires avec celui du nombre des *threads* / processeurs.

IV. Paralléliser l'application

La littérature sur les méthodologies et les outils de parallélisassions d'applications abondent. Nous ne reviendrons donc pas dessus ici. En revanche, nous citerons les travaux de *Yen-Kuang Chen, Matthew Holliman* et *Al.* (Chen, et al. 2002) qui ont pour la première fois associé explicitement la parallélisassions d'un logiciel à la consommation d'énergie de la plate-forme.

---

[5] La loi d'*Amdahl* (Amdahl 1967) stipule qu'il est impossible de diviser le temps de traitement pour un algorithme donné au-delà du temps passé dans ses sections parallèles. L'équation ci-après donne l'accélération – ou *speedup* – théorique pour une application. En pratique, la parallélisassions des applications reste l'une des tâches les plus ardues qui soit pour les développeurs d'applications.

$$s = \frac{1}{(\frac{q}{p} + (1-q))}.$$

Où q est la fraction des opérations exécutables en parallèle et p le nombre des processeurs.

Ils montrent ainsi que la détection en temps réel de filigrane dans les images d'un flot vidéo peut avantageusement profiter de la technologie de *Simultaneous Multithreading Technology – SMT*. L'objectif de cette étude est d'optimiser la performance du détecteur de filigrane grâce à la technologie *SMT* (à deux voies) en utilisant les temps de décrochage (*stall*) d'un des *threads* pour laisser le second *thread* progresser dans ses calculs (et de mieux utiliser ainsi les ressources du processeur). Les gains de performance ainsi obtenus s'échelonnent de 8% à 18%. Les auteurs montrent enfin que cette optimisation se traduit également par une réduction de la puissance moyenne consommée de 12% (passant de 50 à 56 watts – Figure 60 et Figure 60).

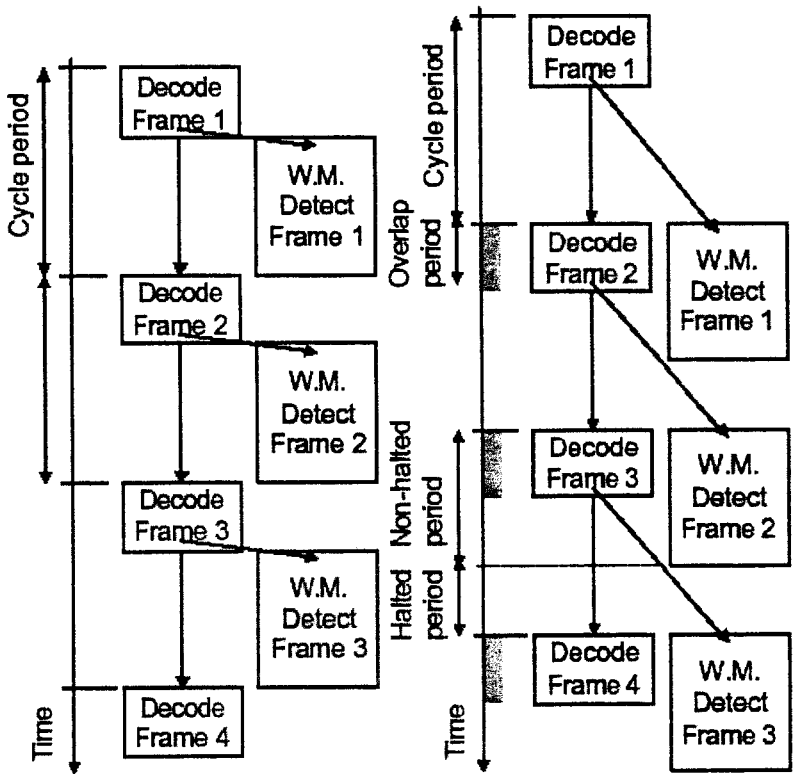

Figure 59 – Les cycles d'attente d'un thread sont exploités par un autre thread de façon à mieux utiliser les ressources des processeurs SMT pour la détection de filigrane. Source : (Chen, et al. 2002).

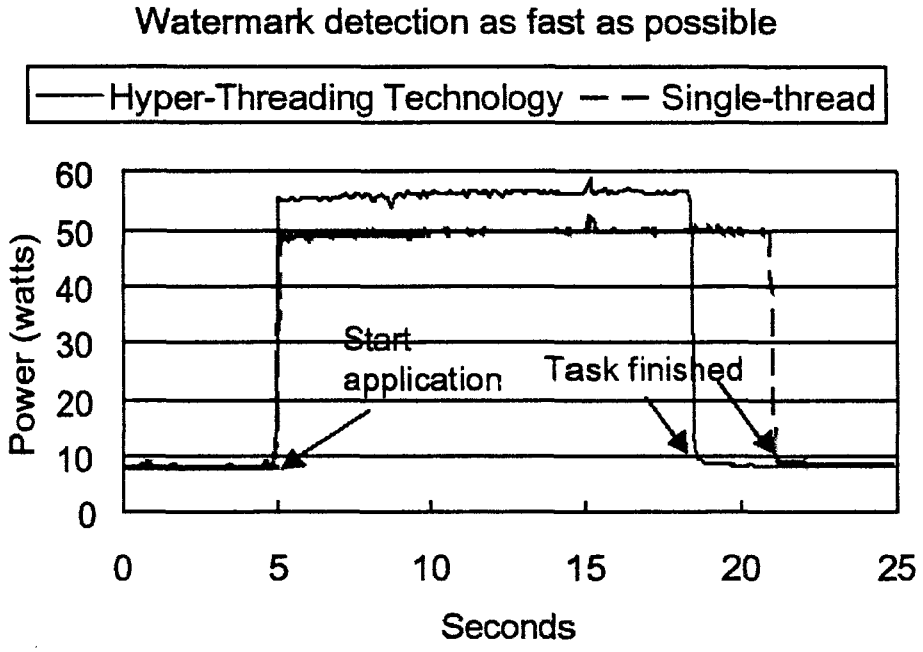

Figure 60 – Les cycles d'attente d'un thread sont exploités par un autre thread de façon à mieux utiliser les ressources des processeurs SMT pour la détection de filigrane. Source : (Chen, et al. 2002).

Nous avons décidé de reproduire ces résultats en appliquant la méthode au code *Black-Scholes*. Les calculs d'intégrales sont effectués par un *pool* de *threads*. Cette parallélisation est très efficace puisqu'elle permet une réduction quasi-linéaire des temps de calcul avec le nombre de processeurs. Nous utilisons l'*API* de *threading poxis* sous *Linux* et *Win32* sous *Microsoft Windows*. Tous les *threads* du *pool* sont actifs lors du calcul et sont synchronisés par des évènements. La Table 23 et la Figure 61 montrent nos résultats. Ainsi, en divisant le temps de calcul par un facteur de 2,82, nous divisions la quantité d'énergie consommée par un facteur de 1,62.

	1 thread	*2 threads*
Durée du calcul	96,00 s	34,00 s
Puissance moyenne	39,71 watts	22,81 watts
Energie	2.190,08 joules	1.350,32 joules

Table 23 – Impact de la parallélisation sur la consommation d'énergie.

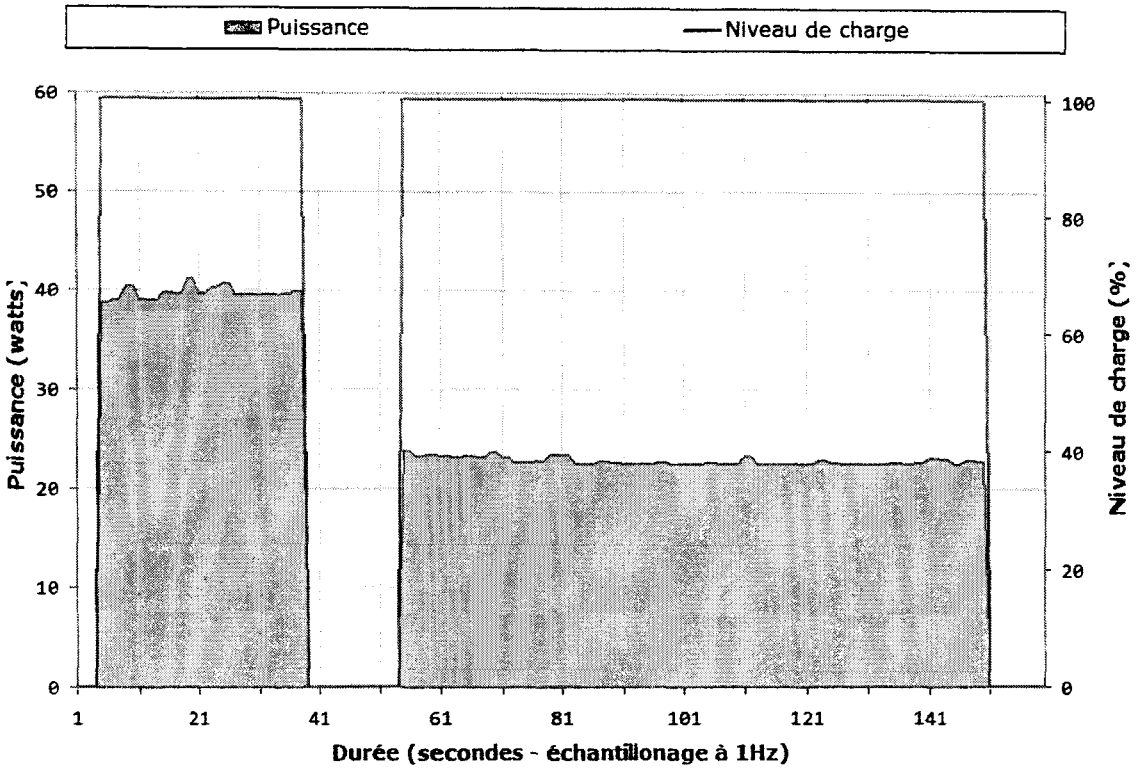

Figure 61 – Energie consommée par Black-Scholes avec un puis deux threads. La puissance dans la phase d'inactivité n'est pas représentée ici.

V. Accélérer le *thread*

A l'instar de la parallélisation, l'optimisation des performances pour un *thread* est largement couverte dans la littérature. Nous ne reviendrons donc pas dessus ici. Nous nous contenterons de démontrer la validité de la proposition en l'appliquant au code de *PovRay 3.7 beta 25*. La Table 24 résume nos résultats. Ceux-ci montrent une réduction moyenne de la quantité d'énergie consommée – en fonction de la charge du système – de 33%. Nous avons obtenu ces résultats en identifiant avec *VTune* les routines les plus utilisées lors de rendus représentatifs. Ainsi, la librairie *BOOST* et les routines du noyau de *PovRay* ont été identifiées comme participant le plus à la performance de l'application. Nous avons ensuite recompilé ces deux composants du programme avec le compilateutr *Intel* en forçant le compilateur a générer un binaire exclusif pour la plate-forme cible (par exemple avec l'option `-QxH` pour le processeur *Nehalem*). Nous avons ensuite utilisé l'option `-Qvec_report3` pour identifier les boucles significatives (aussi détectées avec *VTune*) qui n'ont pas été vectorisées par le compilateur. Nous avons dans un premier temps utilisé le mot clef `restrict` pour indiquer que les arguments de type pointeur des fonctions comprenant certaines de ces boucles ne sont pas *aliasées* ; et utilisé l'option `-Qrestric`. Lorsque cela a été possible, nous avons restructuré les boucles non-

vectorisées pour supprimer les dépendances inter-itération assumées par le compilateur. Enfin, nous avons effectué des compilations guidées par profils pour affiner les optimisations inter-procédurales. Le reste du code a été compilé au niveau -O2 avec le compilateur de *Microsoft*.

Niveau de charge	Energie consommée (Joules) - Code originel	Energie consommée (Joules) - Code optimisé
12,50%	165.400,18	122.900,03
25,00%	83.204.53	61.841,80
37,50%	58.214,77	43.580,10
50,00%	44.688.52	33.477.17
62,50%	37.603,45	28.026,72
75,00%	33.811,72	23.893,26
87,50%	28.867,58	22.176,86
100,00%	27.262,98	22.107,78

Table 24 – Energie consommée par différentes versions de PovRay sur une plate-forme Stoakley pour le rendu du benchmark officiel.

4. Conclusions

Nous avons montré dans ce chapitre qu'il est possible, en appliquant des transformations de code bien choisies, de contrôler et de réduire la consommation énergétique d'une application. Nous proposons de travailler sur deux axes majeurs. Le premier axe est celui de l'optimisation de la performance. Nous avons démontré la validité de la méthode en parallélisant le code et en réduisant le temps de calcul au niveau du *thread* par optimisation. Cette stratégie est gagnante si et seulement si la parallélisation se transforme par une réduction du temps de calcul. Si cela n'est pas vérifié, alors l'impact de parallélisation sur la consommation d'énergie n'est pas garanti et peut même être négatif. En d'autres termes, la parallélisation doit être efficace. Le second axe nous a amené à explorer deux techniques en particulier : l'optimisation et la rationalisation des accès aux disques ainsi que la gradation des performances. Si la première transformation a prouvé son efficacité, la seconde s'est montrée inefficace dans l'état actuel des technologies matérielles de réduction de la consommation électrique. Nous proposons à cet effet l'ajout d'une instruction spécifique qui pourrait rendre la gradation des performances neutre énergétiquement tout en offrant au logiciel – applicatif ou système – un moyen efficace d'informer le matériel sur des opportunités de mise en veille qu'il lui serait impossible à anticiper autrement. L'acceptation de cette proposition doit encore être validée par le comité architectural. Les huit transformations de code que nous avons présentées et tenté d'évaluer dans ce chapitre ne représentent qu'une partie limitée de la batterie de techniques que nous souhaitons étudier dans le cadre de ce projet de recherche mené au sein d'*Intel Corporation*. Ainsi, nous souhaitons terminer cette étude au plus tard à la fin 2009 et prévoyons de débuter la mise en œuvre des techniques les plus efficaces avec les développeurs de logiciels courant 2010. Nous comptons aussi mettre notre méthodologie et les outils logiciels de mesure de l'énergie consommée et du travail utile effectué à la disposition du *Green Grid*.

5. Références

Amdahl, Gene. « Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. » *AFIPS Conference Proceedings.* 1967.

Black, Fischer, et Myron Scholes. « The Pricing of Options and Corporate Liabilities. » (Journal of Political Economy) 81, n° 3 (1973).

Chen, Yen-Kuang, et al. « Media Applications on Hyper-Threading Technology. » (Intel Technology Journal) 6, n° 1 (2002).

Congress, 109th. « PUBLIC LAW 109–431 - An Act To study and promote the use of energy efficient computer servers in the United States. » *LEGISLATIVE HISTORY.* Vol. 152. Washington D.C.: CONGRESSIONAL RECORD, 2006.

Energy Star. 1 Aout 2008. http://www.energystar.gov (accès le Aout 1, 2008).

Energy, U.S. Department of. 1 Aout 2008. http://www.eere.energy.gov/ (accès le Aout 1, 2008).

EPA. *EPA.* 1 Aout 2008. http://www.epa.gov (accès le Aout 1, 2008).

Executive Software. *Reducing Computer System Power Consumption through Disk Defragmentation.* Executive Software, 2008.

Hewlett-Packard Corporation - Intel Corporation - Microsoft Corporation - Phoenix Technologies Ltd. - Toshiba Corporation. *Advanced Configuration and Power Interface Specification.* 2006.

Krishnann, Karthik, et Jun De Vega. « Disk I/O Power/Performance Analysis. » (Intel Corporation) 2008.

Standard Performance Evaluation Corporation. « SPEC – Power and Performance - User Guide. » *SPECpower_ssj2008 V1.00.* 2008.

Standard Performance Evaluation Corporation. « *SPEC JBB2005* ». 1 Aout 2005. http://www.spec.org/jbb2005/ (accès le Aout 1, 2008).

SunGard. Black-Scholes. - *SunGard.* Intel Corporation, 2008.

The Green Grid Consortium. The Green Grid. 8 Septembre 2008. http://www.thegreengrid.org/home (accès le September 8, 2008).

Conclusions et perspectives

1. Conclusions

L'augmentation de la densité d'intégration des processeurs suivant la loi de *Gordon Moore* d'une part, et celle de leur consommation électrique – corolaire de la montée en fréquence des horloges – d'autre part, on rendu les solutions usuellement appliquées pour assurer l'augmentation des performances de génération en génération caduques. Ainsi, et depuis une décennie environ, l'industrie du semi-conducteur à misé l'ensemble de ses espoirs sur le parallélisme. Deux illustrations de cette tendance sont les architectures multi-cœurs et l'architecture *EPIC*.

Mais ce nouvel ordre de marche, qui offre des avantages et des résultats indéniables, a aussi des inconvénients. En particulier, par ce choix, les fondeurs de processeurs on transféré sur les épaules des développeurs une partie considérable de la responsabilité de l'augmentation des performances attendues par les utilisateurs de leurs logiciels. Non seulement cela est un changement important pour la grande majorité des éditeurs de logiciels, mais en plus, cette tâche s'avère être très difficile. En effet, paralléliser correctement une application est loin d'être triviale. Même s'il est raisonnable de penser que les ressources seront allouées à la recherche pour rendre la parallélisation des applications à la portée des développeurs *lambda*, cela prendra plusieurs décennies.

En attendant que ce vœu pieux ne se réalise, la virtualisation – avec ses différentes incarnations que nous avons passées en revue dans notre premier chapitre – se présente comme une bonne alternative pour le court et le moyen terme en permettant l'exploitation des ressources matérielles sans cesse croissantes des plates-formes, notamment serveurs, par les machines virtuelles au sens d'acceptation le plus large, exécutant des logiciels dont la parallélisation pose souvent problème.

Une des incarnations de la virtualisation – un ensemble de technologies introduites depuis les années soixante – qui nous a intéressée dans le cadre de notre étude sont les machines virtuelles, à l'instar de la *Java Virtual Machine* de *Sun Microsystèmes* et de la *Common Language Runtime* de *Microsoft Corporation*. Le franc succès de ces machines virtuelles trouve incontestablement sa source dans les nombreux avantages techniques et financiers qu'ils procurent aux éditeurs de logiciels. Parmi ces avantages techniques, nous mettons l'accent sur le fait que l'optimisation des performances des machines virtuelles échoit à leurs seuls concepteurs.

Dans cette thèse, nous avons voulu faire le lien entre l'architecture *EPIC* et sa promesse d'*ILP* fort d'un coté et les mécanismes d'exécution des machines virtuelles de l'autre coté. Nous avons ainsi présenté une nouvelle approche pour utiliser les ressources matérielles de la famille des processeurs *Itanium* – fondée sur l'architecture *EPIC* – pour optimiser les performances des machines virtuelles. En particulier, nous avons introduit une pile matérielle pour simplifier et optimiser l'exécution du langage intermédiaire des machines virtuelles que nous avons étudiées.

Par ce biais, nous avons voulu offrir à une architecture généraliste – par opposition aux circuits spécialisés développés pour accélérer l'exécution de ces langages – le support matériel pour fermer l'écart de performance existant entre les deux approches. Nous avons voulu ces amendements architecturaux les moins intrusifs possibles et utilisant au mieux les ressources existantes.

Dans le chapitre 2, nous avons ainsi présenté une première implémentation de notre pile matérielle pour l'architecture *EPIC*. Cette première approche fait la part belle au logiciel en laissant celui-ci gérer la pile explicitement. Pour mener nos recherches et vérifier la validité de nos hypothèses, nous avons utilisé le langage et la machine virtuelle *FORTH*. Cette machine, très simple, nous a considérablement simplifié la tâche en remisant à plus tard l'implémentation de concepts plus complexes.

Nous avons ainsi pu montrer que la mise en place d'un cache de pile – logiciel en premier lieu *via* la traduction à la volée du binaire, matériel en second lieu *via* une implémention conservative et par simulation – permettait des gains de performances considérables avec les processeurs *Itanium 2*. Cela s'applique à la machine virtuelle simplifiée de *FORTH*, mais elle ne se prête pas facilement selon-nous à une utilisation avec des machines virtuelles plus complexes.

Dans le chapitre 3, forts de nos résultats et leçons apprises avec la machine virtuelle *FORTH*, nous avons proposé une seconde implémentation de notre pile matérielle pour l'architecture *EPIC*. La différence essentielle entre ces deux piles est que la gestion de la nouvelle est assurée par le matériel. Nous disons pour cela que sa gestion est implicite par opposition à notre première solution qui était explicite.

En particulier, la machine virtuelle peut utiliser directement la pile matérielle en guise de pile d'évaluation – en perdant toutefois en souplesse pour exécuter certaines manipulations de pile usitées par le langage *FORTH*. Notre implémentation permet ainsi aux instructions de l'*ISA* d'*EPIC* de lire et d'écrire leurs arguments directement depuis et dans le sous-ensemble des registres du processeur dédié à la pile matérielle, et ce sans aucune modification. La pile matérielle étant d'ailleurs configurable et même désactivable, ce qui répond à notre volonté d'être le moins intrusif possible et de ne pas impacter l'exécution des binaires des logiciels existants.

Notre nouvelle approche nous permet également de fournir des services avancés en comparaison de l'implémentation du chapitre 2. Ainsi, notre pile assure automatiquement le respect du typage des données mémorisées dans la pile matérielle. Nous pouvons ainsi gérer la pile d'évaluation des langages intermédiaires fortement typés comme le *CIL* de la *CLR* de *.NET*. Dès lors, nous avons montré que l'exécution du *CIL* pouvait se faire par une simple traduction binaire « simple passe ».

Nous avons également proposé un mode de fonctionnement alternatif à notre pile matérielle qui permet d'utiliser plus largement, en théorie du moins, le parallélisme des instructions que permet l'architecture *VLIW*. Nous avons introduit ce mode – que nous appelons *non-bloquant* – pour palier au faible *IPC* accessible par la traduction binaire du code intermédiaire. Nos explorations, malheureusement limitées à ce jour, nous montrent qu'il est

possible, en ayant recours à la compilation *JIT*, d'augmenter l'*ILP* du code généré pour la pile virtuelle.

Enfin, dans le chapitre 4, nous avons entrepris d'étudier l'efficacité énergétique des applications. Pour ce faire, nous avons introduit une méthodologie et les outils logiciels requis pour mesurer précisément l'énergie consommée par les logiciels, mais aussi, plus largement, pour déterminer le *travail utile* effectué par les programmes. Ces deux données sont essentielles à l'étude objective de l'efficacité énergétique des applications. Nos travaux ont d'ailleurs suscité un vif intérêt de la part du comité *Green Grid.*

Nous avons ensuite identifié une liste préliminaire de transformations de code que nous nous sommes proposés d'étudier dans le cadre de cette recherche. Après les avoir classées en deux catégories – l'une limitant / contrôlant la performance des applications transformées, l'autre l'augmentant –, nous avons entrepris de les évaluer en les appliquant à des applications majeures, plutôt que des *benchmarks* peu représentatifs. Dans ce chapitre, nous présentons les premiers résultats de notre étude sur un nombre limité de transformations de code.

Lorsque la réduction de l'énergie consommée par les applications transformées n'a pas pu être vérifiée expérimentalement, nous proposons, dans la mesure du possible, les moyens de corriger ou de contourner les causes des goulets d'étranglement. Nous rappelons enfin que les résultats présentés dans ce quatrième chapitre sont le fruit d'un travail de recherche en cours d'exécution.

2. Perspectives

Tout au long de cette thèse, nous avons été amenés à aborder de nombreux domaines connexes à nos deux sujets de recherches principaux. Nombreux sont ceux qui méritent une étude approfondie, étude que nous n'avons malheureusement pu mener à bien au cours de ces quatre dernières années.

Ainsi, le principal champ de recherche que nous proposons de mener pour compléter notre travail est celui de l'utilisation optimale de notre pile matérielle par un compilateur *JIT*. Ainsi, nous nous proposons d'étudier puis d'appliquer les techniques de compilation pour les architectures *VLIW* dont le but principal est d'augmenter l'*ILP*. En guise d'objectif, nous proposons d'atteindre un niveau d'*ILP* proche sinon équivalent à celui obtenu par les compilateurs *JIT* les plus performants du moment (soit un *ILP* d'environ 2.8). Le champ d'exploration secondaire que nous souhaitons explorer est celui de l'*inlining* guidée par profil d'exécution pour les codes conçus avec des langages de programmation orientés objets. Ce sont en effet ces derniers qui ont mis en échec nos tentatives d'optimisations élémentaires appliquées lors de notre phase de traduction binaire. A la vue de nos résultats expérimentaux nous avons en effet montré que si avec des codes dominés par les calculs, il nous est possible de d'améliorer l'*ILP* notamment en programmant des entrées-sorties en parallèle avec les instructions du programme, les codes orientés objets génèrent des centaines de méthodes ayant moins d'une dizaine d'instructions chacune. Cela ne laisse malheureusement que peu d'espace pour

rechercher du parallélisme d'instruction. En *inlinant* ces méthodes dynamiquement, nous pensons pouvoir améliorer l'*ILP* du code généré.

En ce moment, nous poursuivons activement nos recherches sur l'étude et l'optimisation de l'efficacité énergétique au sein d'*Intel Corporation*. Ce travail, que nous comptons mener à son terme au plus tard à la fin 2009 devrait nous permettre de définir un ensemble de recommandations d'écriture et / ou de réécriture du code source des applications pour en réduire et / ou contrôler la consommation énergétique.

Forts de ces recommandations étayées par nos premiers résultats obtenus lors de tests à grande échelle menés avec nos partenaires privilégiés – en cours d'exécution –, nous comptons diffuser nos recommandations et outils à l'ensemble des programmeurs dans le courant de l'année 2010.

En plus de cette perspective, nous comptons poursuivre et renforcer notre engagement auprès du *Green Grid* pour lui apporter notre soutien dans la définition de la mesure de l'efficacité énergétique des centres de calculs (*IT Productivity Metric*) que le comité est chargé d'édicter en 2009.

Annexes

1. Aperçu de l'architecture EPIC

L'architecture *EPIC* a été développée conjointement par *Hewlett-Packard* et *Intel Corporation*. Les processeurs *Itanium* en sont les premières implémentations. Dans l'architecture *EPIC*, une partie importante du travail habituellement faite par le processeur pendant l'exécution du programme est transférée au compilateur. Ce choix s'explique par le fait que le compilateur a une vue plus large sur le code et dispose d'une quantité de ressources beaucoup plus grande que le processeur. En outre, le compilateur dispose d'un temps quasiment infini pour arriver à ses fins : dégager le plus grand degré de parallélisme d'instructions possible. Cela est vrai pour un compilateur statique, mais ne l'est plus dans le cas d'un compilateur dynamique, qui lui, justement ne dispose pas de cette liberté. Quoi qu'il en soit, cette approche présente des avantages considérables. En premier, cela permet la simplification de l'architecture et une augmentation – théorique – du degré de parallélisme. Elle requiert néanmoins de la part du compilateur et / ou du développeur des connaissances précises sur la façon dont le code sera exécuté par le matériel.

Nous présenterons l'architecture de l'*Itanium* de façon générale. Nous le ferons du point de vue du programmeur. Et comme de coutume, nous le feront à travers les deux aspects fondamentaux qui sont le jeu d'instructions et les ressources matérielles mises à la disposition des programmes. Pour ce dernier aspect, nous nous concentrerons sur les ressources utilisables directement par le programmeur ou *via* une ou plusieurs couches logicielles. Précisons que notre objectif n'est bien évidemment pas de faire ici une présentation exhaustive ni du jeu d'instructions, ni des détails de l'architecture du processeur *Itanium 2*. Nous nous sommes limités aux éléments qui nous semblent importants pour faciliter la lecture de ce document.

a. Parallélisme d'instructions

Le parallélisme d'instructions consiste à sélectionner et à exécuter plusieurs instructions du programme en même temps. Notons que le terme exécution ici ne signifie pas uniquement la tâche qui consiste à évaluer le résultat d'une opération (tâche réalisée souvent dans l'unité arithmétique et logique – UAL) mais aussi toutes les étapes relatives à la lecture, au décodage de l'instruction, à la lecture des opérandes, *etc.* Pour pouvoir exploiter ce parallélisme (appelée en anglais *Instruction Level Parallelism* ou *ILP*), le processeur doit être doté de moyens spécifiques. C'est grâce à des techniques de plus en plus performantes d'*ILP* que les processeurs offrent des performances plus grandes que celles offertes uniquement par la technologie électronique *VLSI* (*Very Large Scale Integration*). Signalons que l'apparition des premiers ordinateurs exploitant l'*ILP* remonte aux années 60 avec le *CDC660* de la société *Control Data*. Une présentation détaillée des différentes formes de l'*ILP* est faite dans [HennPatte03].

b. Pipeline

La première forme d'*ILP* consiste à travailler sur plusieurs instructions en parallèle en découplant les étapes de l'exécution d'une instruction. L'idée du *pipeline* est de ne pas attendre la fin de l'exécution de l'instruction précédente pour démarrer celle d'une nouvelle instruction. L'exploitation efficace du pipeline nécessite une connaissance à priori de l'adresse de la prochaine instruction à exécuter avant la fin de l'exécution de l'instruction en cours. Ceci permet ainsi de commencer la lecture de la nouvelle instruction pendant le décodage de la précédente.

Dans 80% (ou plus) des cas, après l'instruction de numéro *i*, le processeur doit exécuter l'instruction numérotée *i+1*. Les 20% restants des cas concernent les instructions de branchements conditionnels. Ces dernières posent un problème pour le *pipeline*. Ne connaissant pas le résultat de la condition qui dictera le branchement à prendre, le processeur a la possibilité soit d'arrêter la lecture de toute nouvelle instruction tant que la condition n'a pas été évaluée ou bien de prédire le résultat de la condition. Cette prédiction amènera le processeur à choisir l'une deux branches. C'est souvent, cette deuxième approche qui est réalisée. En cas d'erreur sur la prédiction, il est nécessaire de revenir au branchement pour suivre l'autre branche.

Le choix du nombre d'étages et de leur rôle est à la charge du concepteur. Notons qu'en général, un nombre réduit d'étages simplifie la conception du processeur et permet un ré-aiguillage rapide du programme en cas d'erreur sur la prédiction des branchements. A l'opposé, un nombre plus important d'étages réduit le travail réalisé dans chacun des étages et offre ainsi la possibilité d'avoir des fréquences d'horloge plus élevées. Dans le cas des processeurs embarqués par exemple, où le coût du processeur et sa consommation électrique sont des facteurs importants, ce nombre d'étages est souvent réduit (trois dans l'*ARM7*).

Dans des processeurs dédiés aux hautes performances, ce nombre peut atteindre 30, comme dans le cas des dernières versions des processeurs *Pentium 4* d'*Intel Corporation*. Le processeur *Itanium 2* dispose d'un pipeline de huit étages comme le montre la Figure 62 (Intel Corporation 2008).

IPG	ROT	I-BUFF	EXP	REN	REG	EXE	DET	WRB
Génération du pointeur d'instructions	Rotation des instructions	Tampon de découplage	Décodage et ventilation	Renommage	Lecture des registres	Exécution ALU	Détection des exceptions	*Write back*

Figure 62 – Etages du pipeline du processeur Itanium 2.

Les deux premiers étages IPG et ROT composent la partie avant ou *front-end* du *pipeline*. Ces étages sont responsables de la lecture d'un groupe de six instructions par cycle d'horloge. Dans cette étape, il est possible de réaliser une rotation sur le paquet d'instructions lu afin de le présenter à l'étage exécution (EXE) de façon à ce qu'il soit directement exécutable (ou aligné). Le

résultat de cette étape est envoyé vers une suite de quatre *buffers* pouvant contenir chacun deux paquets (soit un total de 4*2*3=24) d'instructions. L'utilisation de ce buffer permet de découpler le *front-end* et *back-end* qui contient les six étages suivants. Ce découplage permet à la partie *back-end* de travailler à une grande vitesse. C'est dans l'étage IPG qu'est réalisée la prédiction de branchement.

Dans les deux phases suivantes (EXP et REN), le processeur décode les motifs associés aux paquets et envoie jusqu'à six instructions par cycle aux différentes unités fonctionnelles. C'est dans cet étage que sont renommés les registres pour supprimer des dépendances entre instructions pouvant ralentir le processeur.

Dans la phase REG, le contenu des registres sources des instructions sélectionnées pour l'exécution sont lus.Enfin les trois dernières phases : EXE, DET et WRB, concernent l'exécution de l'opération dans les unités fonctionnelles. La détection des exceptions et la résolution des branchements sont réalisées dans la phase DET. Enfin dans la phase WRB, il y a mise à jour du registre destination de l'instruction exécutée.

c. EPIC à la croisée du super scalaire et du VLIW

Afin d'augmenter le nombre d'instructions pouvant être exécutées par cycle d'horloge, une autre approche que le pipeline consiste à doter le microprocesseur de ressources lui permettant de traiter, à la même étape, plusieurs instructions à la fois. Avant l'architecture *EPIC*, les microprocesseurs exploitant ce type d'*ILP* étaient répartis en deux familles : les *super scalaires* (comme la famille des processeurs *Pentium* et autres *PowerPC* et *Alpha*) et les processeurs à mots d'instructions très longs aussi nommés *VLIW* pour *Very Large Instruction Word.*

Des microprocesseurs *super scalaires* sont capables d'exécuter plusieurs instructions par cycle (moins de dix en général). Dans le domaine des microprocesseurs pour la haute performance ou pour les serveurs de base de données, il existe très peu de processeurs *VLIW* commercialisés. Une grande partie des ces processeurs sont restés au stade du prototype.

La principale raison qui a freiné leur adoption est la non compatibilité du code entre les différentes architectures. A l'opposé, dans le domaine des systèmes embarqués et plus particulièrement dans les processeurs dédiés au traitement du signal, les processeurs *VLIW* sont largement utilisés. Les DSP *TMS320* de *Texas Instruments*, le *Trimedia* de la société *Phillips*, ou le *Crusoe* de la société *Transmeta* sont de parfaites illustrations de ce succès. En effet, pour ce type de plates-formes, le logiciel est intégré une fois pour toute dans la plate-forme embarquée. Puisque l'utilisateur ne dispose pas de moyen pour changer ou rajouter des applications, le problème de la compatibilité du code ne se pose pas.

EPIC est la seule implémentation du *VLIW* dans une plate-forme commerciale. En effet, *EPIC* exploite les informations statiques fournies par le compilateur afin de réaliser des optimisations et structurer le code sous forme d'instructions à mots très longs (approche *VLIW*). En plus de sa nature *VLIW*, *EPIC* intègre un certain nombre de caractéristiques que l'on trouve

habituellement dans les processeurs *super scalaires*, comme le *renommage* des registres, la prédiction dynamique des branchements, l'utilisation d'un tableau de marques pour la résolution de la dépendance entre registres, le prè-chargement des données dans la mémoire cache, *etc.*

d. Micro architecture du processeur *Itanium2*

La Figure 63 donne un aperçu simplifié de la micro architecture du processeur *Itanium 2*. Nous allons décrire brièvement les unités fonctionnelles essentielles. Commençons par la lecture des instructions. Cette partie contient la logique nécessaire pour réaliser le chargement et le pré-chargement des instructions de la mémoire cache des instructions de niveau 1 (IL1) ainsi que le mécanisme de prédiction des branchements. IL1 à une taille de 16 Ko structurée en ligne de 4 blocs (associativité de 4) contenant 64 octets chacun. Le mécanisme de pré-chargement, contrôlé par le compilateur (pré-chargement logiciel), a pour but de d'éviter des défauts de lecture dans le cache IL1. Ceci est réalisé par l'insertion au moment de la compilation d'indications permettant de charger les prochaines instructions à exécuter de la mémoire cache de niveau 2 (L2) vers IL1.

Les deux paquets d'instructions (6 instructions) lus par cycle sont envoyés vers le tampon de découplage ou buffer des instructions qui est directement connecté aux ports des unités fonctionnelles.

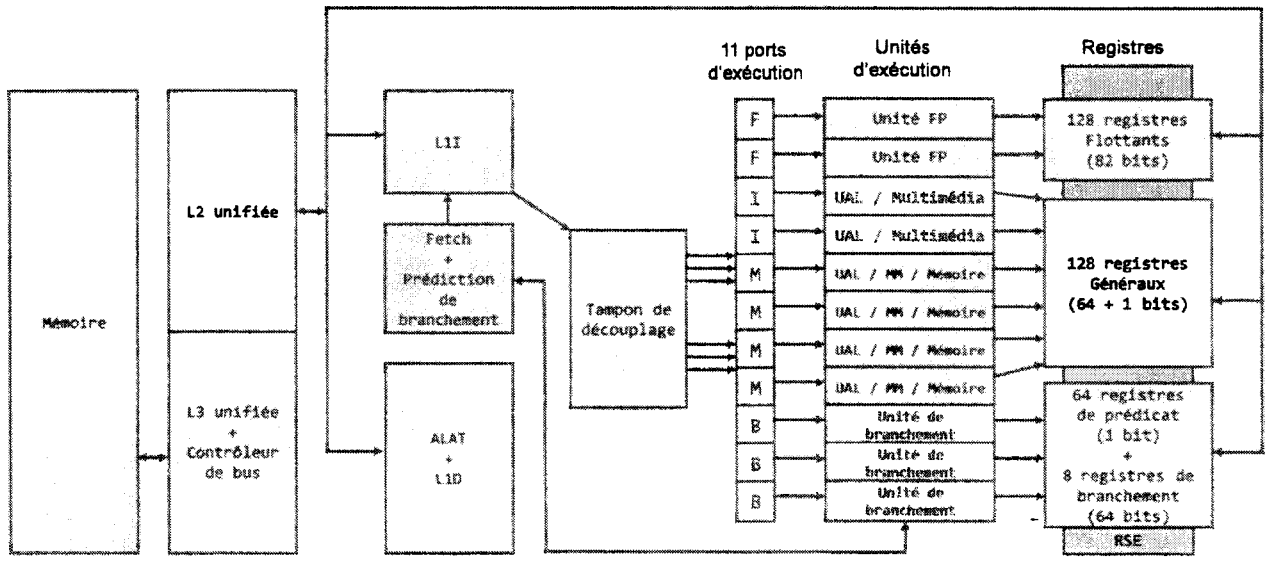

Figure 63 – Diagramme simplifié du processeur Itanium 2. MM est l'abréviation de Multimédia.

Viens ensuite l'exécution des instructions. Cette partie contient l'ensemble des unités fonctionnelles chargées de réaliser les tâches demandées par les instructions du groupe d'instructions sélectionné. Ainsi la sélection des instructions ne se fait pas instruction par instruction mais par la sélection de tout un groupe d'instructions. Plus de détails seront donnés sur la façon avec laquelle est construit un groupe dans les paragraphes suivants. Notons

simplement que la structure de paquet permet de véhiculer les instructions parallèles. Pour qu'un groupe s'exécute, il est nécessaire que toutes les instructions qui le composent soient prêtes à l'exécution. Les unités fonctionnelles de cette partie de la micro architecture sont regroupées comme suit.

Les unités entières notées `UAL0` à `5`. Deux unités entières notées `I0` et `I1`, et une unité pour les décalages notée `ISHFT`. Au total, six instructions entières peuvent être exécutées à chaque cycle.

Les unités d'accès à la mémoire. Quatre ports sont connectés à l'unité de mémoire cache, dont deux doivent êtres des instructions de chargement et deux doivent êtres des instructions de stockage. Notons que les requêtes relatives aux instructions de chargement sur les entiers sont envoyées vers la mémoire cache de données de niveau 1 (L1D) et, en parallèle de façon spéculative, elles sont aussi envoyées vers la mémoire cache de niveau 2 (L2). Pour le reste des requêtes, c'est à dire pour le stockage de données entières et de chargement / stockage des autres types de données, elles sont aiguillées directement vers la mémoire cache de niveau 2. La Figure 64 donne la structure des trois niveaux de mémoire caches, la largeur des transferts entre les niveaux ainsi que la bande passante du bus externe pour les transferts de données. On peut ainsi voir que le cache L1I et L1D ont été optimisés pour permettre des accès (en cas de sucée en seul cycle. S'il n'a y pas de défauts, alors le temps d'accès à L2 est de 5 cycles. En cas de défaut, il faut rajouter le temps de chargement du bloc de données qui dépend de la bande passante du bus.

Les unités multimédia. Il y six unités multimédia (notées `PALU0` à `5`), `PALU` correspond ici à *Parallel Arithmétique and Logic Unit*. Il y aussi deux unités de décalage pour les données multimédia (notées `PSMU0` et `1` pour *Parallel Shift Multimedia Unit*, une unité de multiplication multimédia `PMUL` pour *Parallel MULtiply* et une unité de comptage de bits à 1 `popcnt` pour *population count*. Au maximum deux instructions multimédias peuvent être exécutées par cycle horloge. Dans l'architecture du processeur *Itanium 2*, les données multimédia sont codées sur 64 bits et peuvent être considérées comme différents vecteurs. Par exemple comme deux données sur 32 bits, 4 données sur 16 bits ou 8 données sur 8 bits. Pour profiter de cette représentation plusieurs instructions du type *SIMD – Single Instruction Multiple Data* – sont disponibles. Il est donc possible de réaliser par exemple huit additions sur huit jeux de données sur huit bits chacun en seule instruction. Différentes opérations sont possibles : opérations logiques, décalages, *etc.*

Les unités en virgule flottante. Il y en a deux, capables d'effectuer des multiplications et addition (`FMAC 0` et `FMAC1` pour *Floating Multiply And Cumulate* et deux unités générales pour les données en virgules flottantes notées `FMISC0` et `FMISC1`. Au total, deux instructions flottantes peuvent être initiées par cycle.

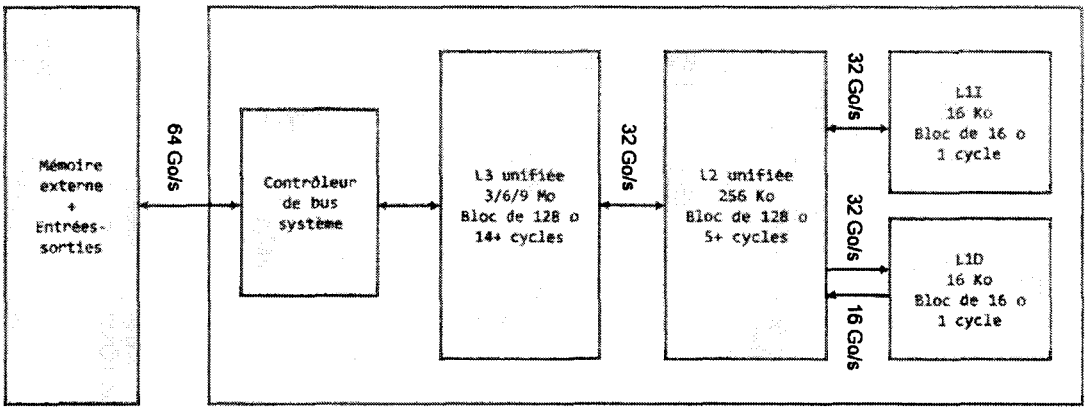

Figure 64 – Structure et performances de la hiérarchie des mémoires cache du processeur Itanium 2.

e. Registres

Les processeurs *Itanium* intègrent quatre types de registres. Les registres généraux sont au nombre de 128 registres et ont une largeur de 64 bits chacun (noté `gr0` à `gr127`). Le registre `gr0` contient toujours la valeur `0`. Ces registres servent à stocker les valeurs entières utilisées comme source et / ou destination des instructions entières. Associé à chacun des registres nous trouvons un bit noté `NaT` (*Not a Thing*). Ce bit est utilisé lors de la spéculation sur les instructions de chargement pour des données de type entier. Comme on le verra de façon plus détaillée dans le paragraphe dédié aux différentes techniques de spéculation utilisées dans l'*Itanium*, le bit `Nat` est utilisé pour signaler une exception suite à l'exécution d'une instruction de chargement spéculative (Figure 65).

Les registres `gr0` à `gr31` sont toujours accessibles au compilateur. A titre d'exemple, par convention, le registre `gr12` est utilisé comme pointeur de pile en mémoire, *memory stack pointer* ou *sp*. Les autres registres `gr32` à `gr127` sont utilisés comme une pile de registres pour faciliter le passage des paramètres entre les appels de fonctions imbriquées. Ceci permet de diminuer ou d'éliminer les accès mémoire habituellement réalisés pour passer les paramètres et / ou sauvegarder le contenu des registres de la fonction appelante. A chaque fois qu'une nouvelle fonction est appelée, une trame appelée *register stack frame* contenant les données utilisées en entrée par la fonction appelée est définie. Cette trame est composée de deux parties. La première contient les arguments d'entrée (notée *input arguments*) et les données locales (notée *local*) et la deuxième contient les arguments de sortie (notée *outputs*). La taille de ces deux zones est définie dynamiquement en cours d'exécution du programme. La Figure 66 illustre cette utilisation pour deux fonctions `A` et `B`.

La fonction `A` dispose d'un ensemble de 14 registres (de 32 à 45) pour la zone *inputs* et *local* et de 7 registres (de 46 à 52) pour la partie *outputs*. Lorsque `B` est appelée par la fonction `A`, la zone *outputs* de `A` est passée à `B` comme la zone contenant les paramètres d'entrée de `B`. La taille maximale de la zone *outputs* est de 8 registres. Si des éléments supplémentaires doivent

être passés à la fonction appelée, alors il est nécessaire de les placer en mémoire (dans la pile en mémoire par exemple). Si la fonction appelée a besoin d'écrire des valeurs dans la trame, la fonction doit exécuter l'instruction `alloc`. Dans ce cas, la fonction appelée (`B` dans l'exemple) peut contenir, en plus de la zone *inputs*, les zones *locals* et *outputs*. Les tailles respectives de ces zones sont données comme opérandes de l'instruction `alloc`. Notons au passage que les registres de `B` sont renumérotés à partir de 32 après l'appel de `B`.

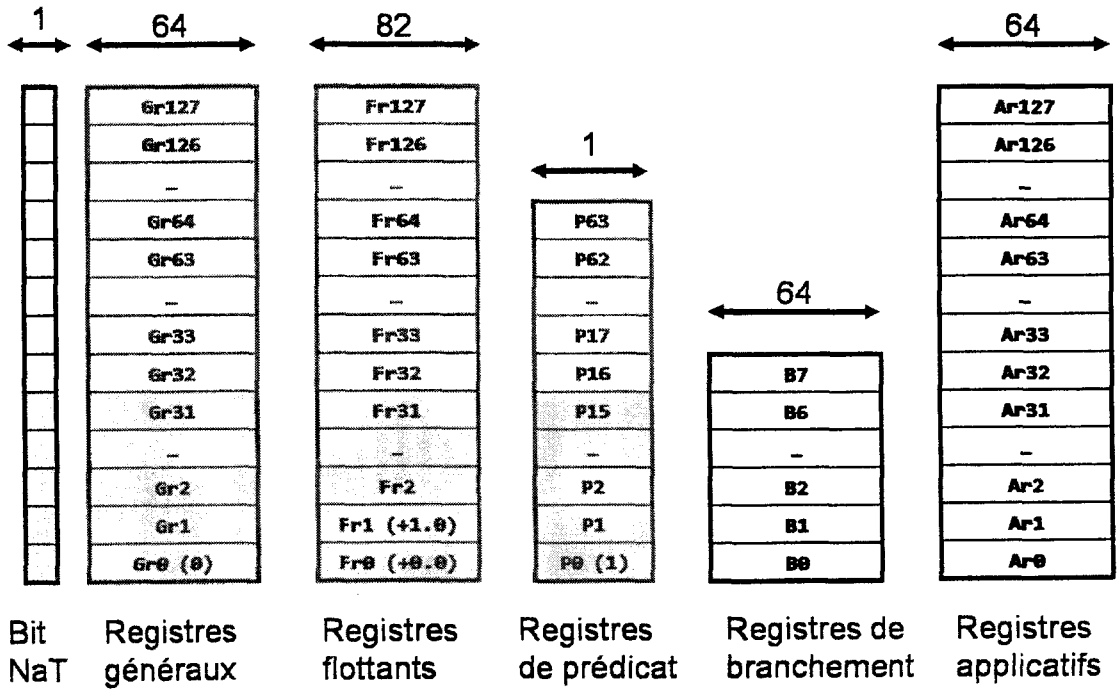

Figure 65 – Les registres des processeurs Itanium.

Lorsque la fonction `B` se termine, elle exécute l'instruction de retour vers `A`. La pile des registres retrouve alors sa forme initiale et permet ainsi de préserver le contenu des registres d'*inputs* et *locals* de `A`. Comme le montre la Figure 66, une fois que la fonction `B` est terminée, la fonction `A` retrouve sa trame à partir du registre 32.

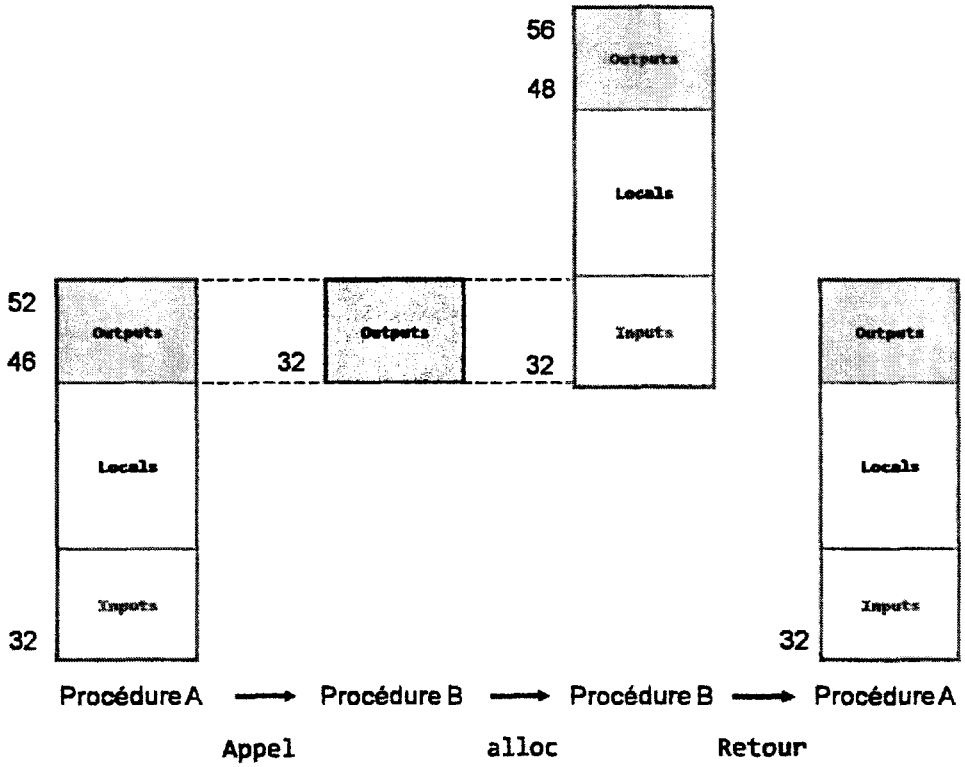

Figure 66 – Principe d'utilisation des registres des processeurs Itanium pour le passage des paramètres entre fonctions imbriquées.

Les processeurs *Itanium* contiennent 128 registres flottants de 82 bits chacun (`fr0` à `fr127`). Comme pour `gr0`, le registre `fr0` contient toujours la valeur `+0.0`. Le registre `fr1` contient lui aussi une valeur fixe : `+1.0`. Les registres généraux 0 à 31 sont statiques. Les registres flottants n'ont pas de bits `NaT` associés. Si une exception se produit dans un chargement vers un registre flottant, alors une valeur réservée spécialement dédiée à cet effet est chargée dans le registre destination. Cette valeur est notée `NaTVal` (`Not a Thing Value`) qui a `0` comme mantisse et une valeur hors des limites de la norme *IEEE* comme exposant. Le reste des registres (de 32 à 127) est utilisé pour réaliser le pipeline logiciel. Sans entrer dans les détails, l'utilisation d'un pipeline logiciel est l'une des meilleures stratégies d'optimisation dont dispose le compilateur pour les codes scientifiques qui calculent sur les nombres flottants.

En plus des registres généraux et flottants, les processeurs *Itanium* intègrent aussi 64 registres de prédicat notés `p0` à `p63`. Ces registres d'un bit servent à éliminer les instructions de branchement. Ils sont positionnés à `1` ou à `0` suivant le résultat (vrai ou faux) d'une instruction de comparaison. Le registre `p0` est toujours à `1` (vrai). Comme pour les registres généraux et les registres flottants, les registres `p0` à `p15` sont statiques alors que les registres `p16` à `p63` sont utilisés pour réaliser le pipeline logiciel. En général l'utilisation des registres de prédicat permet de transformer des sections contenues dans des branchements (structure *If-Then-Else*) en une suite d'instructions avec prédicat.

Le processeur *Itanium 2* intègre aussi huit registres de 64 bits destinés à contenir les adresses des cibles pour les instructions de branchement indirect. Dans ces instructions de branchement, l'adresse vers laquelle sera éventuellement aiguillée l'exécution est indiquée par un registre de branchement (`br0` à `br7`). Ce dernier est lu et son contenu est transféré vers le pointeur d'instructions noté *IP* pour *Instruction Pointer*. L'adresse cible utilisée pour mettre à jour *IP* doit être un multiple de seize pour pointer sur le début d'un paquet. En effet seize octets – ou 128 bits – correspondent à un paquet de trois instructions codées sur 41 bits chacune, plus le gabarit de 5 bits.

Enfin, le processeur *Itanium 2* dispose de 128 registres de 64 bits chacun nommés registres d'application `ar0` à `ar127`. Chacun des ces registres a une fonction particulière et peut être accédé par l'application. A titre d'exemple, les registres `ar65` (`ar.lc`) et `ar66` (`ar.ec`) sont utilisés pour implémenter le pipeline logiciel. Enfin d'autres registres sont utilisés pour permettre une compatibilité avec l'architecture *IA-32*. L'instruction `alloc` que nous avons vue dans le paragraphe réservé aux registres généraux, se sert aussi des registres d'applications. Voici son format : `alloc reg = ar.pfs, inputs, locals, outputs, rots`.

Cette instruction réalise deux taches. Premièrement, elle sauvegarde le contenu du registre `ar.pfs` (*previous frame state*) dans le registre `reg`. Le registre `ar.pfs` correspond au registre d'application `ar64` et contient trois informations sur la procédure appelante (son niveau de priorité, la valeur de `ar.lc`, et enfin une copie du registre *CFM – Current Frame Marker* – de la procédure appelante). Ce registre spécialisé contient les informations sur la trame courante de registres. En particulier sa position dans le *RSE – Register Stack Engine* – et sa taille. Le nombre de registres *inputs*, *locals* et *output* est stocké sous la forme de trois champs. *sor* (*size of rotating portion in stack frame*) qui vaut *rots / 8*. *sof* (*size of stack frame*) qui vaut *inputs* + *locals* + *outputs*. Et *sol* (*size of locals portion in stack frame*) qui vaut *inputs* + *locals*. Avant de rendre la main à la procédure appelante, toute fonction qui a exécuté une instruction `alloc` doit se terminer par une instruction `mov ar.pfs = reg` qui remet `ar.pfs` à son contenu initial. `reg` est le numéro du registre utilisé pour sauvegarder `ar.pfs` par l'instruction `alloc`. Dans un second temps, `alloc` déclare la zone *outputs* de la fonction appelante comme la zone *inputs* pour la fonction appelée et alloue les registres *locals*, *outputs* et *rots* (respectivement) pour la taille des zone registres pour les données locales, registres de sortie et registres pour les données de rotations (respectivement). En utilisant ces valeurs, les champs *sof*, *sor* et *sol* seront mis à jour dans le registre *CFM*.

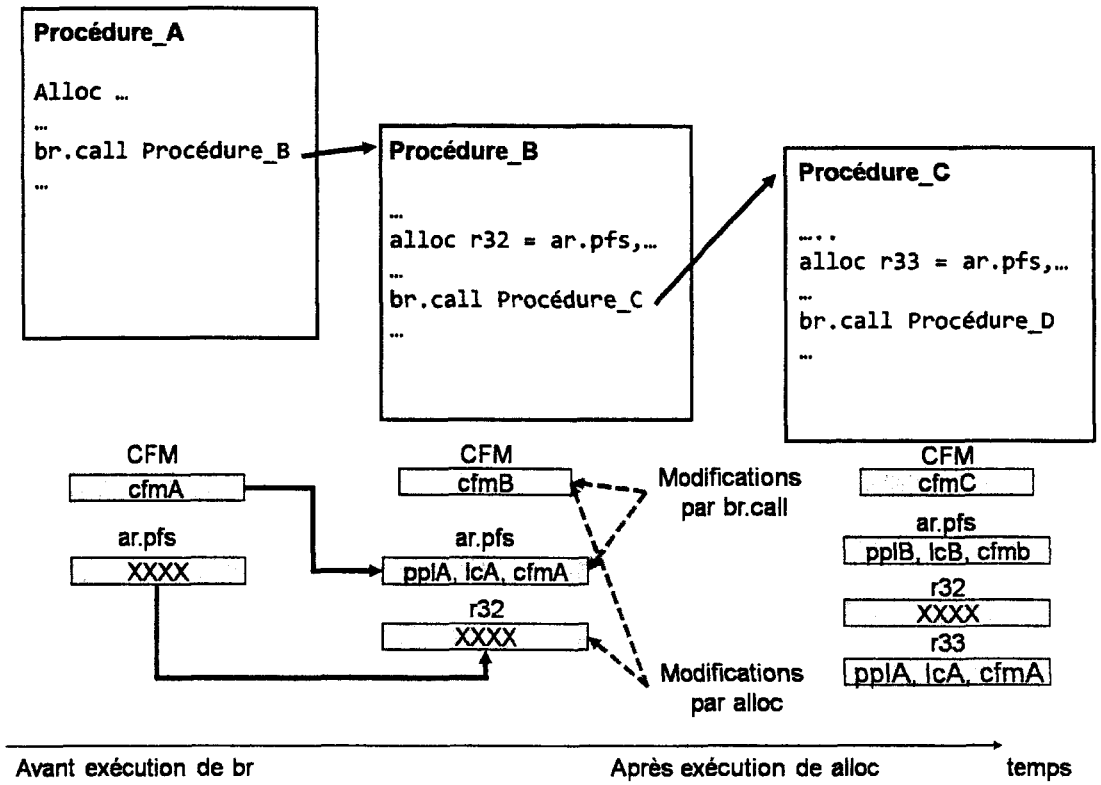

Figure 67 – Principe d'utilisation des registres CFM et PFS des processeurs Itanium.

f. Aperçu du jeu d'instructions du processeur *Itanium 2*

Dans cette section nous allons présenter l'essentiel du jeu d'instructions des processeurs *Itanium*. Nous allons passer en revue les instructions les plus courantes du jeu d'instructions. Une présentation détaillée se trouve dans le vivre de Trimper et S. Evans 2003. Le jeu d'instructions des processeurs *Itanium* est divisé en sept groupes qui gèrent respectivement : les transferts entre registres ; les calculs sur les entiers et les flottants ; les opérations logiques et de manipulation de bits ; les chaînes de caractères ; les comparaisons et les branchements ; les accès à la mémoire et enfin les opérations multimédia (vectorielles essentiellement). D'un point de vue syntaxique, une instruction Itanium a la forme suivante : `[rp] mnémonique[.comp] dest = src`. Où `rp` est un registre prédicat (`p0` à `p63`), `mnémonique` est le nom de l'instruction. `comp` est un suffixe permettant de compléter le nom de l'instruction en précisant son action. `dest` est l'opérande destination. Enfin, `srcs` représente un ou plusieurs opérandes sources. Par défaut le registre prédicat utilisé est le registre `p0` qui est toujours à un et le résultat est donc écrit dans le registre destination. Si `rp` vaut zéro, alors le résultat de l'instruction n'est pas validé. Les champs `dest` et `srcs` font références soit à des registres, soit à des opérandes immédiats. Le Listing 1 donne deux exemples d'instructions commentées.

```
(p1) add r10 = r12, r13 // r10 = r12 + r13 si p1 = 1
(p0) ld8.s r31 = [r3] //charge dans r31 la donnée (8 octets) à l'adresse r3
```

Listing 22 – Exemples d'instructions commentées.

La Table 25 donne les utilisations les plus courantes des instructions arithmétiques entières. Où `extSigne` signifie extension de signe de la valeur immédiate exprimée sur 8 bit (entre `-128` et `+ 127`) vers une valeur sur 64 bits signée.

Instructions	***Tâches réalisées***
(rp) add r1 = r2, r3	*Si rp = 1, r1 = r2 + r3*
(rp) add r1 = imm, r3	*Si rp = 1, r1 = (valeur immédiate) + r3*
(rp) sub r1 = r2, r3	*Si rp = 1, r1 = r2 - r3*
(rp) sub r1 = imm8, r3	*Si rp = 1, r1 = extSigne(imm8) - r3*

Table 25 – Exemples d'instructions sur les entiers.

Les instructions logiques réalisent un traitement bit à bit des données. A titre d'exemple, le compilateur transforme une instruction en langage C/C++ utilisant l'opérateur bit à bit *ET*, notée `&`, en instruction `and`. Il existe deux autres relations : *OU* et *OU* exclusif, notées `or` et `xor`. La Table 26 donne les formes les plus fréquentes pour les instructions logiques (illustrées avec l'instruction `and`).

Instruction	***Tâches réalisées***
(rp) and r1 = r2, r3	*Si rp = 1, r1 = r2 ET r3*
(rp) and r1 = imm8, r3	*Si p = 1, r1 = imm8 ET r3*
(rp) andcm r1 = r2, r3	*Si rp = 1, r1 = r2 ET (NON r3)*

Table 26 – Exemples d'instructions logiques.

Les instructions de décalage de données sont utilisées pour réaliser des opérations de multiplication et de division par des puissances entières de 2. A titre d'exemple un moyen simple pour vérifier la parité d'une donnée est de réaliser deux décalages d'une position à droite et d'une position à gauche : `(num_int >>1) <<1`. La Table 27 liste les formes les plus fréquentes pour les opérations de décalage à droite et à gauche.

Instructions	***Tâches réalisées***
(rp) shl r1 = r2, r3	*Décaler r2 de (r3) positions à gauche, résultats dans*

	r1. Compléter par des zéros (décalage logique).
(rp) shl r1 = r2, compteur	*Décaler r2 de « compteur » positions à gauche, résultats dans r1. Compléter par des zéros (décalage logique).*
(rp) shr r1 = r2, r3	*Décaler r2 de (r3) positions à droite, résultats dans r1. Compléter par le bit de signe (décalage arithmétique).*
(rp) shr r1 = r2, compteur	*Décaler r2 de « compteur » positions à droite, résultats dans r1. Compléter par des zéros (décalage logique).*
(rp) shrp r1 = r2, r3, compteur	*Décalage à droite d'une paire de registres (r2, r3) de compteur positions, résultats dans r1. Compléter par le bit de signe (décalage arithmétique).*
(rp) extr r1= r3, pos6, Long6	*Les bits < pos6 + Long6 - 1, pos6> de r3 sont copiés dans r1 de <Long6-1,0>. Les autres bits de r1 sont positionnés par le bit < Long6 - 1 > de r1.*
(rp) dep r1 = r2, r3, pos6, Long6	*r1 est positionné par r2 sauf aux positions < Long6 + pos6 - 1, pos6> où ce sont les bits < Long - 1, 0> de r3 qui sont écrit dans r1.*

Table 27 – Exemples d'instructions de décalage.

La structure générale d'une instruction de comparaison est la suivante : `(rp) cmp.relCom.typeComp p1, p2 = r2 (ou imm8), r3`. Les suffixes `relComp` et `typeComp` complètent le code opération `cmp`. Le premier champ `relComp`, pour relation de comparaison, est obligatoire et indique la relation sur laquelle porte l'opération de comparaison. Celle-ci peut être égale à : `eq` (égalité), `ne` (différence), `lt` (infériorité), `le` (infériorité et égalité), *etc.* Le deuxième champ `typeComp`, pour type de comparaison, n'est pas obligatoire. Lorsqu'il est présent, il indique de quelle manière les registres prédicat (`p1` et `p2` dans l'exemple) sont positionnés. Si `typeComp` n'est pas présent, `p1` et `p2` ne sont positionnés que si `rp = 1`. Dans le cas contraire, l'instruction `cmp` n'est pas exécutée. A l'opposé si `typeComp` est présent et vaut `unc` (pour *unconditionnal*) l'instruction est toujours exécutée et les registres prédicat sont toujours positionnés. Ainsi dans le cas où `rp = 0`, `p1` et `p2` dans notre exemple sont mis tous les deux à `0`. Le listing 2 donne un exemple de l'utilisation de cette instruction sur trois tests imbriqués (source (Trimper et S. Evans 2003)).

```
if(crel1) { /* code C */

      if(crel2) {

            /* Block A */

      } else {

            /* Block B */

      } else {

            if(crel3) {
```

```
                    /* Block C */
                } else {
                    /* Block D */
                }
            }
        }
        // code en assembleur Itanium
        cmp.crel1 pt, pf = r… // Premier test
        (pt) cmp.crel2.unc pa, pb = r… //deuxième test
        (pa) Block A
        (pb) Block B
        (pf) cmp.crel1.unc pc, pd = r…
        (pc) Block C
        (pf) Block D
```

Listing 23 – Exemples de code C et son expression en assembleur Itanium. r... équivaut à une expression du type r1, r2 ou r1, immédiat.

L'utilisation du complément `unc` simplifie grandement le code. En effet, dans le cas où le premier test est faux ,`pt` est mis à zéro. Du coup les deux branches du deuxième test avec `pa` et `pb` sont tous les deux mis à zéro. Ceci interdit l'exécution de ces deux instructions. L'utilisation d'une instruction `cmp` sans `unc` comme complément, aurait donné un code erroné, puisque les instructions avec les prédicats `pa` et `pb` seraient toujours en attente de l'écriture d'une valeur dans ces deux registres. Ceci peut se produire plus tard et aurait donné ainsi un code erroné.

Lors de la conception de l'architecture *EPIC*, les concepteurs ont porté une attention particulière aux instructions de branchement (`br`) dont il existe un grand nombre de variantes. La Table 28 donne les formes les plus fréquentes de l'instruction de branchement. Le compilateur transmet ses indications via `bwh` (*branch whether hint*). Cette indication peut prendre quatre formes :

- `spnt` (*static not-taken*) prédiction statique (sans utilisation de ressources). Le branchement est prédit avec une grande probabilité *non pris*. Dans ce cas, Le compilateur est quasi sûr que le branchement sera pris.
- `sptk` (*static taken*) prédiction statique (sans utilisation de ressources). Le branchement est prédit avec une grande probabilité *pris*.

- `dpnt` (*dynamic not taken*) prédiction dynamique (avec utilisation de ressources). Le branchement est prédit avec une faible probabilité *non pris*.
- `dptk` (*dynamic not taken*) prédiction dynamique (avec utilisation de ressources). Le branchement est prédit avec une faible probabilité *pris*.

`ph` (*prefetch hint*) représente l'indication donnée par le compilateur pour le pré-chargement des instructions dans le cache IL1. Deux valeurs sont possibles: `few` et `many`. Le compilateur utilise `few`, lorsqu'il estime que peu d'instructions seront exécutées à partir du point de branchement. A l'opposé, le compilateur utilisera `many`, lorsqu'il estime que plusieurs blocs d'instructions seront exécutés à partir du point de branchement. Dans l'implémentation *Itanium 2* [ItarefMan03], un seul bloc cache sera pré-chargé avec `few` et deux blocs seront pré-chargés avec `many`.

`dh` (*branch cache deallocation hint*) représente l'indication donnée par le compilateur sur la probabilité de ré-exécution ou non de l'instruction de branchement. Dans certaines implémentations, le processeur contient un buffer pour stocker les adresses cibles des branchements. Le processeur garde, dans ce buffer, les adresses cibles des dernières instructions de branchement (par défaut). Lorsqu'une instruction de branchement ne sera pas ré-exécutée dans l'immédiat, il est intéressant d'utiliser son entrée dans le buffer pour garder l'adresse cible d'une autre instruction pouvant être, elle, plus utile. Pour cela, le champ `dh` sera mis à `clr`. Dans le cas ou le champ `dh` est absent l'adresse sera écrite dans le buffer.

Enfin, la dernière instruction du tableau `br.cloop` est utilisée pour coder des boucles du type `for` en C. Elle fait implicitement référence au registre d'application `ar.lc` (noté aussi `ar65`). En général, `br.cloop` est utilisée en fin de boucle et le registre `ar.lc` est alors initialisé avec le nombre d'itérations moins une. A chaque exécution de `br.cloop`, le registre `ar.lc` est décrémenté. Tant que ce registre est non nul, le branchement est réalisé vers l'étiquette.

Instructions	***Tâches réalisées***
(rp) br.cond.bwh.ph.dh étiquette	*Branchement conditionnel relatif à la position courante (IP-relatif)*
(rp) br.cond.bwh.ph.dh b2	*Branchement conditionnel indirect par utilisation de registre de branchement b2*
(rp) br.call.bwh.ph.dh b1 = étiquette	*Branchement (ou appel) vers une fonction repérée par l'étiquette « étiquette ». b1 contient l'adresse retour.*
(rp) br.call.bwh.ph.dh b1 = b2	*Idem que précédemment sauf que la fonction est repérée le registre de branchement b2.*
(rp) br.ret.bwh.ph.dh b1	*Retour vers l'appelant, dont l'adresse à été sauvegardée dans b1 (fin de fonction).*
(rp) br.cloop.bwh.ph.dh étiquette	*Tant que le registre d'application ar.lc est non nul, aller à étiquette.*

Table 28 – Exemples d'instructions de branchement. Etiquette est une étiquette sur 25 bits. Les 4 derniers bits de l'adresse d'un bundle sont toujours à zéro. Pour économiser de la place, le champ Etiquette est codé sur 21 bits et complété par 4 zéros à l'exécution.

Concernant les accès à la mémoire, l'architecture *EPIC* reprend le principe des architectures *RISC*. Ainsi, les accès à la mémoire se font exclusivement au travers de deux instructions : chargement ou *load* et stockage ou *store*. La Table 29 donne les formes les plus courantes de ces instructions.

Instructions	Tâches réalisées
`(rp) ldsz.ldtype.ldhint r1 = [r3]`	Chargement d'une donnée dans le registre r1 par la donnée pointée par r3 (adressage indirect).
`(rp) ldsz.ldtype.ldhint r1 = [r3], r2`	Comme précédemment. Le registre r3 est par la suite incrémenté par le contenu de r2 (adressage indirect post incrémenté).
`(rp) ldsz.ldtype.ldhint r1 = [r3], Imm9`	Comme précédemment. Le registre r3 est par la suite incrémenté par la valeur immédiate Imm9 (adressage indirect post incrémenté).
`(rp) stsz.sttype.sthint [r3] = r2`	Stockage de la donnée contenue dans le registre r2 dans la position mémoire pointée par r3.
`(rp) stsz.ype.sthint [r3] = r2, imm9`	Comme précédemment. Le registre r2 est par la suite incrémenté par la valeur immédiate Imm9 (adressage indirect post incrémenté).

Table 29 – Exemples d'instructions d'accès à la mémoire.

Le champ `sz` (*size*) de l'instruction indique la taille de la donnée manipulée par l'instruction. Ce champ accepte quatre valeurs possibles (1, 2, 4, et 8), correspondant à 1, 2 (un demi-mot), 4 (un mot) ou 8 octets (un double mot). Pour les chargements avec une taille inférieure à 8, il y a extension de zéro vers la partie de poids fort pour obtenir une valeur sur 64 bits. Les champs `ldtype` et `sttype` indiquent le type de l'instruction mémoire. Plusieurs valeurs sont possibles. Par défaut (si aucune indication n'est donnée), il s'agit d'accès mémoire normale. Parmi les autres possibilités existantes pour `ldtype`, nous avons `s` (chargement spéculatif), `a` (chargement avancé), `sa` (chargement spéculatif avancé), `c.nc` (chargement avec vérification et non suppression de l'adresse de la table *ALAT*) et `c.clr` (chargement avec vérification et suppression de l'adresse de la table *ALAT*). Notons que les instructions de stockage (`st`) ne peuvent pas être spéculatives.

Le champ `ldhint` et `sthint` donne une indication sur le degré de localité temporelle du bloc cache contenant la donnée référencée. Le but étant d'éviter de charger dans les différents niveaux de mémoire cache de données (L1, L2 et L3) un bloc de données lorsque celui-ci est considéré par le compilateur comme ayant une faible localité temporelle. Autrement dit, lorsque le compilateur estime qu'il ne sera pas accédé prochainement. Cela évite d'évincer un bloc plus

fréquemment accédé. Pour le processeur *Itanium 2*, quatre options existent pour ce champ. Lorsque le complément n'est pas spécifié, alors le bloc de données est chargé (s'il n'est pas déjà présent) dans tous les niveaux de caches (sauf dans L1 pour les instructions de stockage). `nt1` existe uniquement pour les chargements. Le bloc de données ne sera pas chargé dans L1 s'il n'y est pas déjà. Dans le cas contraire, l'information *LRU* (*Least Recently Used*) utilisée pour remplacer les blocs les moins fréquemment utilisés n'est pas mise à jour afin d'augmenter la probabilité du bloc d'être évincé. `nt2` agit comme `nt1` mais s'applique aux instructions de chargement et de stockage et dans L2. Enfin, avec `nta` aucune localité temporelle n'est requise. Le bloc de données sera chargé uniquement dans le niveau 2. S'il n'est pas présent dans L3, alors il ne sera pas chargé. Les informations *LRU* de tous les niveaux ne seront pas mises à jour.

g. Format des instructions

Le compilateur et l'assembleur *Itanium* regroupent les instructions par paquets de trois instructions (ou *bundel*). Un paquet d'instructions est exécuté entièrement et le processeur *Itanium 2* est capable d'exécuter simultanément deux groupes d'instructions. Chaque groupe est codé sur seize octets ou 128 bits. La Figure 68 donne le format d'un groupe d'instructions.

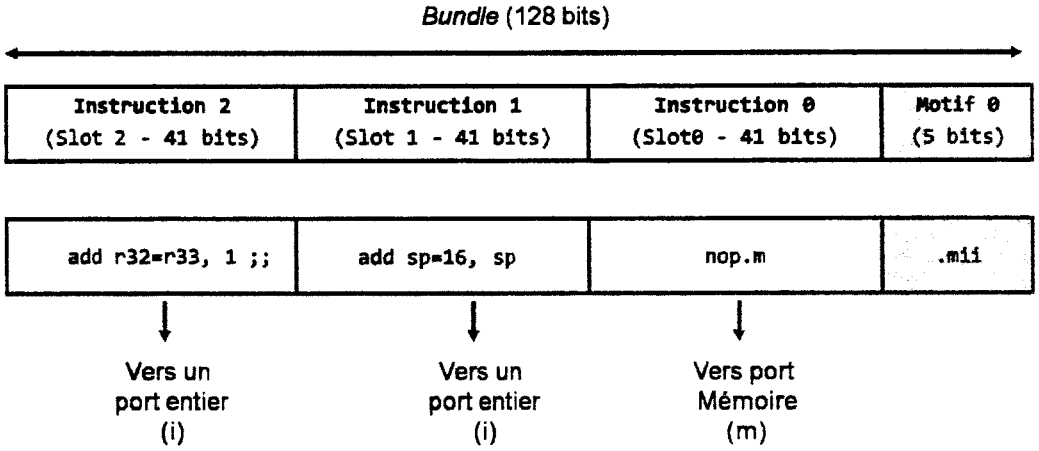

Figure 68 – Format d'une instruction de processeur Itanium.

La Figure 68 montre les trois instructions codées sur 41 bits chacune. Le champ *template* ou motif est utilisé lors de la phase de décodage des instructions. Dans l'exemple de la Figure 68, le groupe est composé de deux instructions entières (I) et d'une instruction mémoire (M). Pour cette raison, le champ *template* est positionné à MII. De cette façon, les deux dernières instructions sont envoyées directement vers l'unité de calcul sur les entiers alors que la première instruction est envoyée directement vers l'une des deux unités mémoire. Avec cinq bits pour le champ *template*, il est possible de coder 128 combinaisons différentes. Seul 24 combinaisons sont utilisées pour le moment. En plus du type d'instruction, M pour mémoire, I pour entier, les types F, B et X sont respectivement utilisés pour les flottants, les branchements et les opérations multimédia. Le champ *template* indique aussi les dépendances entre les instructions à l'intérieur d'un même paquet ou de deux paquets voisins. La dépendance est matérialisée à l'aide de la

marque stop (|| dans le tableau suivant). Ce bit lorsqu'il est présent, indique que les instructions après la marque stop dépendent (dépendance de données) des instructions avant la marque stop. A titre d'exemple, comme le montre le tableau suivant, une valeur `00` pour le champ *template* indique que les instructions qui composent le paquet ne contiennent aucune dépendance (ni entre elles ni avec les instructions du paquet suivant). Ainsi si le paquet suivant ne contient pas de bits stop, les six instructions des deux paquets peuvent s'exécuter simultanément. L'utilisation du bit stop simplifie la micro architecture. Une valeur `02` pour le champ *template*, place la marque stop après l'instruction 1 (instruction du type I), signifiant ainsi qu'une dépendance existe entre l'instruction 2 et les instructions avant (0 et 1).

Un ensemble d'instructions consécutives sans marque de stop entre elles constituent un groupe. Un groupe d'instructions est localisé entre deux marques stop et peut utiliser plusieurs paquets. La Figure 69 donne un exemple de programme composé de cinq groupes utilisant cinq paquets. Le groupe 2 utilise deux paquets. Il n'y a aucune relation entre la notion de groupe et de paquet. En effet, un groupe peut être encodé dans un ou plusieurs paquets. De même un paquet (*bundel*) peut faire partie d'un groupe, ou bien contenir tout seul un ou plusieurs groupes. Un champ *template* égal à `03` indique la présence de 2 groupes : le premier qui se termine après l'instruction 1 et le deuxième qui ne contient qu'une seule instruction (instruction 2).

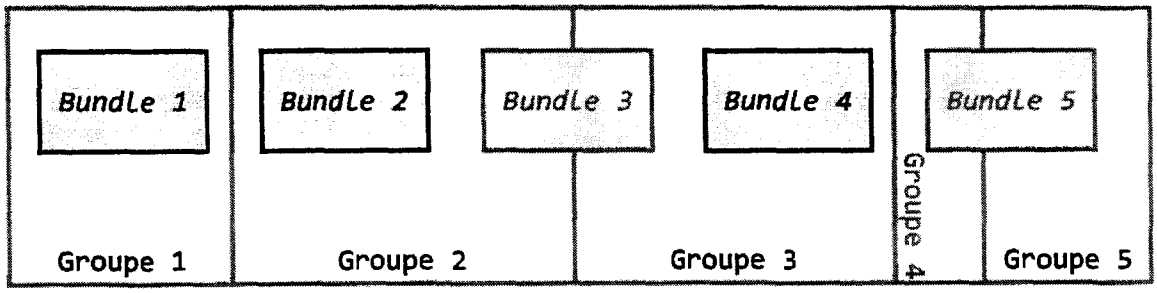

Figure 69 – Représentation d'une séquence d'instructions sous forme de groupes et de paquets (bundels).

Si le compilateur ou l'assembleur ne trouvent pas de motif correspondant à la séquence d'instructions, il devra alors remplir les instructions par des instructions vides (sans fonction). Ces instructions sont notées `nop` (pour *no operation*). Le code assembleur suivant donne l'exemple de deux paquets contenant six instructions et donnant de façon explicite le *template* des *bundles*.

```
{ .mib
      ld4 r37 = [r36] //chargement dans r37 par le contenu de r36
      add r34 = r0, r33
      br.call.sptk.many b0 = sousProg# ;; // appel et fin de groupe
} { .mii
```

```
        nop.m //nop pour l'unité mémoire

        add sp=16, sp

        add r32=r33, 1 ;;

}
```

Listing 24 – Exemple de deux bundles utilisant la notation explicite des bundles et des templates.

2. Structures et mécanismes fondamentaux de *FORTH*

Dans cette section, nous allons présenter les structures et les mécanismes fondamentaux du langage *FORTH*. Notre objectif n'est pas de proposer une description détaillée du langage et de sa syntaxe ni de son implémentation. Nous souhaitons avant tout fournir au lecteur une vision suffisamment précise du rôle du noyau *FORTH* et de son mode de fonctionnement. Cette description facilitera la lecture du chapitre.

FORTH est un langage de programmation procédural et extensible. L'utilisateur a ainsi le loisir de définir de nouvelles procédures – des *mots* – qui, une fois compilés, sont immédiatement utilisables *via* l'invite de commandes ou *via* d'autres mots, eux-mêmes rédigés en *FORTH*. Ces mots sont organisés en dictionnaires, avec au minimum un dictionnaire principal. L'aspect le plus important de *FORTH* pour notre propos, c'est que les mots *FORTH* prennent leurs arguments et stockent leurs résultats dans une pile dédiée, aussi connue sous le nom de pile des arguments ; pile des paramètres ou bien encore pile d'évaluation. Il s'agit d'une pile *LIFO* (*Last-In, First-Out*) généralement conservée en mémoire centrale. La Figure 70 montre un exemple d'utilisation de la pile d'évaluation lors d'une session *FORTH* hypothétique. + et * sont des mots *FORTH* et chacun d'eux dépile deux arguments depuis la pile, applique une opération aux données et empile le résultat. A l'instar de ces mots très simples, les mots *FORTH*, même les plus complexes, utilisent toujours la pile. Cette pile ne doit pas être confondue avec la pile dédiée au contrôle des flux – dénommée pile de retour. C'est l'utilisation de la pile – d'évaluation, nous ne répèterons plus cet adjectif par la suite et préciserons la nature de la pile que s'il ne s'agit pas de la pile d'évaluation – qui fait de *FORTH* le canon des machines à piles. C'est pour cette raison – en plus de notre affection pour ce langage – que nous l'avons retenu comme objet d'étude de ce chapitre.

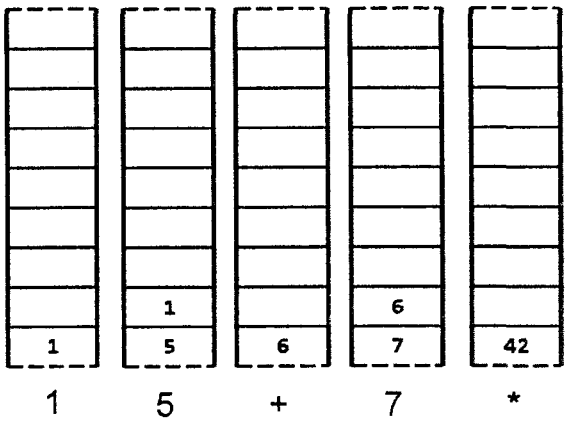

Figure 70 – Exemple d'utilisation de la pile en FORTH. Nous noterons la forte ressemblance avec la notation polonaise inversée, notamment utilisée par le langage de description de page PostScript ou par les calculateurs signés Hewlett-Packard.

Nous annoncions en introduction de ce chapitre que *FORTH* est plus qu'un langage de programmation et qu'il est possible de parler de système *FORTH* – si ce n'est de philosophie. En effet, *FORTH* est un langage de programmation interprété. C'est-à-dire qu'il n'utilise pas le modèle classique des langages compilés où par exemple une phase d'édition de liens – *linker* – est requise après la compilation et avant de pouvoir demander l'exécution du programme par le système d'exploitation. Cela procure une grande souplesse d'utilisation et simplifie grandement la mise au point des logiciels. De plus, *FORTH* peut se passer de système d'exploitation. *FCode*, une variante de *FORTH* est par exemple utilisée par certains systèmes, – notamment par ceux de *Sun Microsystèmes* – pour rédiger et exécuter le code du *Firmeware* pendant la phase de pré amorçage (*pre-boot*). Une autre qualité essentielle de *FORTH* explique aussi ce choix : il est possible de concevoir un système *FORTH* minimal mais fonctionnel qui tienne en moins de 10 Ko.

a. Les interpréteurs *FORTH*

L'interpréteur externe – nous verrons par la suite la raison de ce qualificatif – fournit au minimum la possibilité de saisir du texte et celle d'afficher un caractère. Si le concepteur d'un système *FORTH* ne souhaite pas fournir les pilotes nécessaires au contrôle du matériel (clavier, mémoire vidéo, *etc.*), alors il peut utiliser les services d'un système d'exploitation hôte. C'est l'approche que nous avons retenue dans le cadre de nos travaux. Signalons également qu'une grande partie du système *FORTH*, ses interpréteurs, son compilateur, *etc.* comme nous le verrons par la suite – est lui-même écrit en *FORTH*.

L'interpréteur externe lit le tampon de saisie et y détecte une suite de caractères séparés par au moins un espace. Cette chaîne de caractères est ensuite recherchée dans le dictionnaire du *FORTH*, en partant depuis la dernière entrée créée – donc la plus récente, ce qui permet d'ailleurs de modifier dans le temps la fonction des mots pour les nouvelles définitions. En résumé, les mots du dictionnaire sont stockés en mémoire sous la forme d'une liste chainée de définitions. Cette organisation est assurée lors de la compilation des mots, un mécanisme fondamental que nous détaillerons par la suite.

Si le mot n'est pas trouvé lors de la recherche dans le dictionnaire, alors l'interpréteur essaye de convertir la chaîne en un nombre – en utilisant la base numérique en cours et qui est choisie par l'utilisateur. S'il y parvient, alors la valeur est empilée (cela est représenté sur la Figure 70 suite à la saisie des nombres `1`, `5` et `7`). Dans le cas contraire, un message d'erreur est affiché – *mot inconnu* ou équivalent. Si la chaîne est trouvée dans le dictionnaire, alors le mot correspondant du dictionnaire est exécuté. L'exécution est assurée par l'interpréteur interne de *FORTH*. C'est l'existence de ce second interpréteur qui nous force à affubler le premier du qualificatif externe, sans lequel il y aurait confusion.

Un mot *FORTH* peut aussi être vu comme une suite d'appels de fonctions – ou d'autres mots. Définissons par exemple deux mots, respectivement `CARRE` et `PUISSANCE4`. Le premier va dépiler le sommet de la pile, élever cette valeur au carré puis empiler le résultat. Le second fait de même mais élève la valeur à la puissance quatre au lieu du carré. En *FORTH*, `CARRE` s'écrit : `CARRE DUP * ;` Pour information, le mot `DUP` en *FORTH* duplique le sommet de la pile et `*`

multiplie le contenu des deux premiers niveaux de la pile. Le deux-points ou *colon* (:) ainsi que et le point-virgule – *semicolon* (;) – sont aussi des mots *FORTH*. Nous reviendrons sur leur signification plus tard, mais notons juste que le *colon* débute une définition de mot et que le point-virgule marque la fin de la définition en cours. La Figure 71 montre l'état de la pile à la fin de l'exécution de chaque mot *FORTH* – situé en dessous de la représentation symbolique de la pile. La figure se lit de la gauche vers la droite et représente une chronologie.

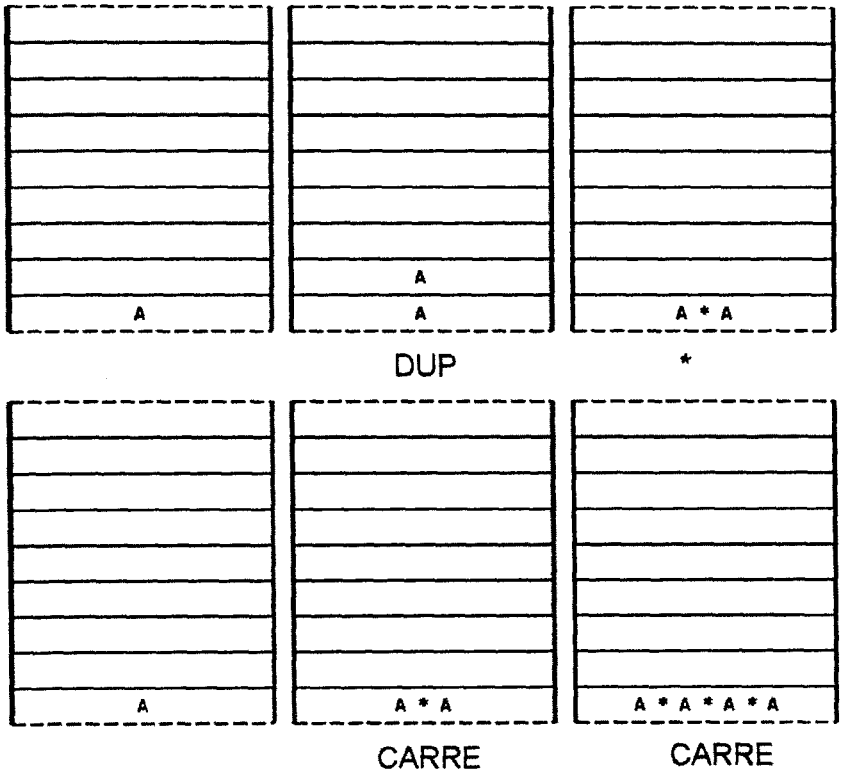

Figure 71 – Résultat sur la pile de l'exécution des mots CARRE (haut de la figure) et PUISSANCE4 (bas de la figure).

Une fois que nous avons saisi la définition de CARRE au clavier – ou bien qu'elle a été lue depuis un périphérique de stockage –, le mot CARRE est compilé et mémorisé dans le dictionnaire. Il est dès lors possible de l'utiliser pour effectuer des calculs en direct en interagissant avec l'interpréteur externe ou de l'intégrer dans d'autres programmes *FORTH*. Ainsi, nous pouvons définir PUISSANCE4 comme : PUISSANCE4 CARRE CARRE ; Ce dernier mot peut être vu comme l'appel successif de la fonction CARRE puis CARRE. Chaque appel de CARRE est lui-même une suite d'appels de DUP et * comme nous l'avons décrit précédemment. La Figure 72 donne une représentation symbolique de cette organisation.

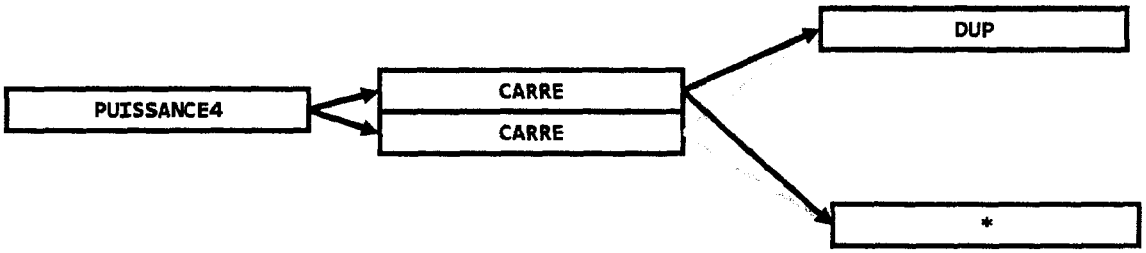

Figure 72 – Relations logiques entre les mots FORTH. Les flèches représentent l'équivalent des appels de fonctions des langages de programmation fonctionnels comme le C. Certaines flèches ont été grisées pour rendre le graphique plus lisible, sans autre signification particulière.

Donc, l'exécution de PUISSANCE4 peut être vue comme les appels successifs à CARRE puis CARRE. Chaque CARRE étant l'appel de DUP puis de la multiplication. Prenons le cas de CARRE pour décrire le mécanisme utilisé par l'interpréteur interne du *FORTH* pour exécuter les mots de son dictionnaire. Historiquement, et essentiellement pour des raisons de compression de code, l'utilisation de l'instruction `call` à l'adresse du code de chaque routine a été abandonnée. Signalons que l'adresse de début de code de chaque mot est conservée dans son entrée dans le dictionnaire. Puisque `call` (et à *fortiori* `ret`) n'est pas utilisé, il faut au *FORTH* un mécanisme d'appel qui lui soit propre. Il s'agit de l'interpréteur interne, qui peut aussi être vu comme une machine virtuelle qui exécute un code intermédiaire. En effet, sans le recours à `call`, le code généré par le compilateur ne peut pas s'exécuter.

L'interpréteur interne conserve l'équivalent du compteur ordinal du processeur pour les mots *FORTH*. Un registre y est généralement dédié et il contient l'adresse du prochain mot à exécuter. Tout aussi généralement, un autre registre du processeur contient l'adresse du début de la routine du mot en cours d'exécution, mais il ne s'agit là que d'une question de choix d'implémentation. Pour parachever le mécanisme, le retour d'appel nécessite aussi la définition d'un mot spécial en *FORTH*. Il s'agit de NEXT. Ce dernier charge l'adresse de la routine du prochain mot à exécuter dans le registre dédié à cet effet. Il incrémente le registre contenant le pointeur du mot suivant, et effectue un saut inconditionnel à l'adresse de la routine à exécuter. NEXT est ajouté par le compilateur à la fin de chaque mot *FORTH*. C'est ainsi que l'interpréteur enchaîne l'exécution des mots. La Figure 73 montre cette organisation, également connue sous les noms de chaînage direct ou *direct threaded code*.

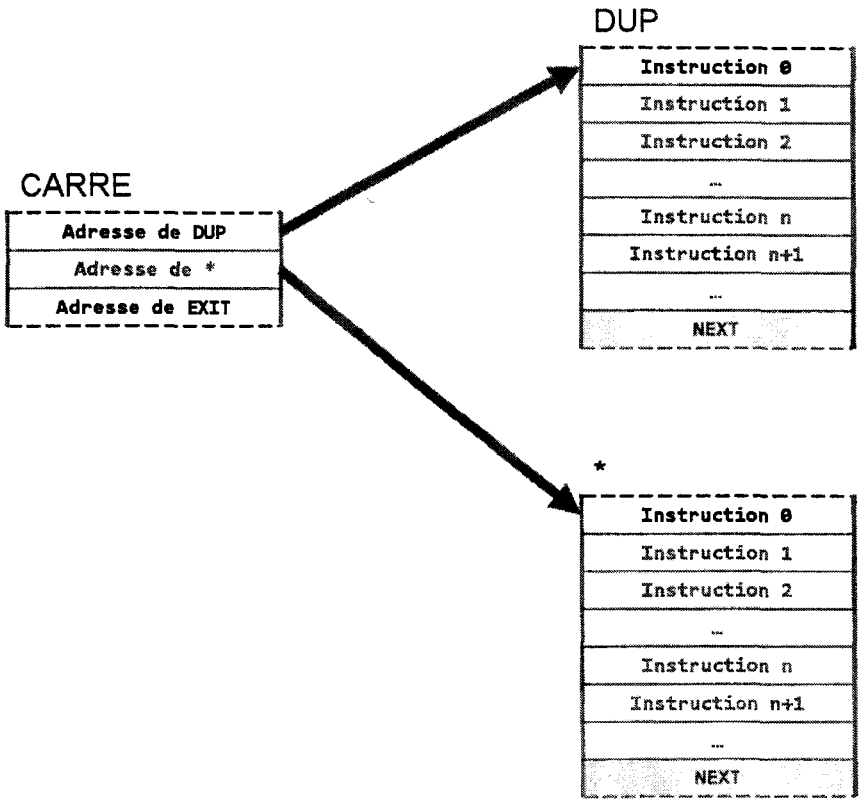

Figure 73 – Organisation des mots FORTH mise en place pour assurer le mécanisme d'exécution de l'interpréteur interne pour du code chaîné direct.

Jusqu'à présent, les mots exécutés par l'interpréteur *FORTH* sont implémentés en langage machine. C'est le cas des mots fondamentaux du système *FORTH*. D'ailleurs, en fonction de l'implémentation considérée, une part plus ou moins importante du système est écrite en *FORTH* ou en langage machine. C'est un choix d'implémentation, de degrés de portabilité voulu, et de niveau de performance requis. Signalons toutefois, que le *FORTH* est extrêmement rapide et compact en comparaison de codes générés par des compilateurs traditionnels. Enfin, notons l'existence d'un vocabulaire ASSEMBLEUR qui permet l'écriture de mots en langage machine (l'assembleur utilisant simplement la notation polonaise inversée).

Si nous nous penchons à présent sur le cas du mot PUISSANCE4, alors nous avons une difficulté supplémentaire à résoudre, puisque CARRE est un mot défini en *FORTH* et n'est donc pas codé en langage machine. Il s'agit ici bien sur d'un simple choix d'implémentation car pour des raisons de performance par exemple, PUISSANCE4 pourrait aussi bien être écrit en assembleur (mais nous perdrions alors la factorisation de DUP). Pour palier au problème, nous utilisons l'organisation connue sous le nom de chaînage indirect ou *indirect threaded code*. Comme son nom le laisse supposer, une *indirection* est requise par l'interpréteur interne dans ce cas. Elle prend la forme d'un pointeur en début de définition de mot, qu'il soit défini en *FORTH* ou en langage machine. Ce pointeur est généralement désigné par nom *CFA* ou *Code Field Address*.

Certains *FORTH*, dont le notre, conservent juste après le *CFA* un pointeur retour vers la définition du mot dans le dictionnaire, ce qui permet une décompilation rapide des mots (ce pointeur n'est pas représenté sur les figures). Si le mot est défini en langage machine, alors le *CFA* pointe sur la première instruction – qui se trouve juste après le pointeur (notée `instruction 0` sur les figures). En revanche, s'il s'agit d'un mot défini en *FORTH*, alors le pointeur d'*indirection* pointe sur la fonction d'interprétation de *FORTH*. Celle-ci est désignée par le nom DOCOL (qui signifie *DO COLON*, *colon* étant le mot de définition de *FORTH* comme nous l'avons signalé précédemment – Figure 74).

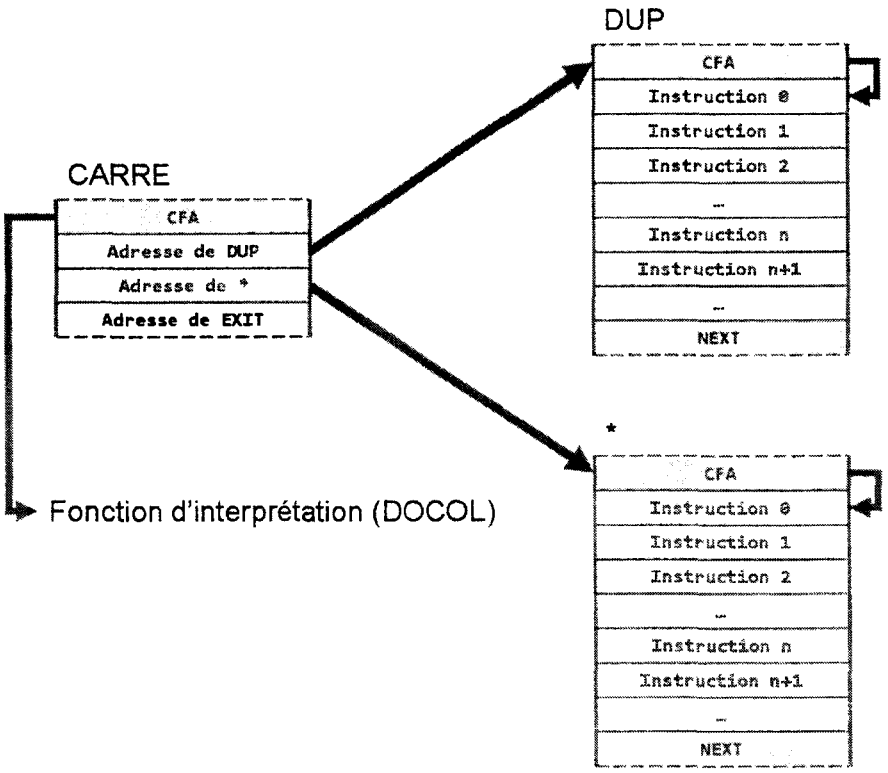

Figure 74 – Organisation des mots FORTH mise en place pour assurer le mécanisme d'exécution de l'interpréteur interne pour du code chaîné indirect.

Le rôle de DOCOL est simple – il ne s'agit en fait que de quelques instructions assembleur – et peut se résumer de la façon suivante : empiler sur la pile de retour de *FORTH* le contenu du pointeur de mot (l'adresse du prochain mot à exécuter), le pointeur de code (l'adresse du début de la routine) est utilisé pour calculer le nouveau pointeur de mot (une adition suffit). Lorsque NEXT est exécuté pour DOCOL (il s'agit d'un mot *FORTH* comme un autre, il se termine donc par NEXT), l'exécution du prochain mot débute. Et ainsi de suite. (Figure 75 et Figure 76).

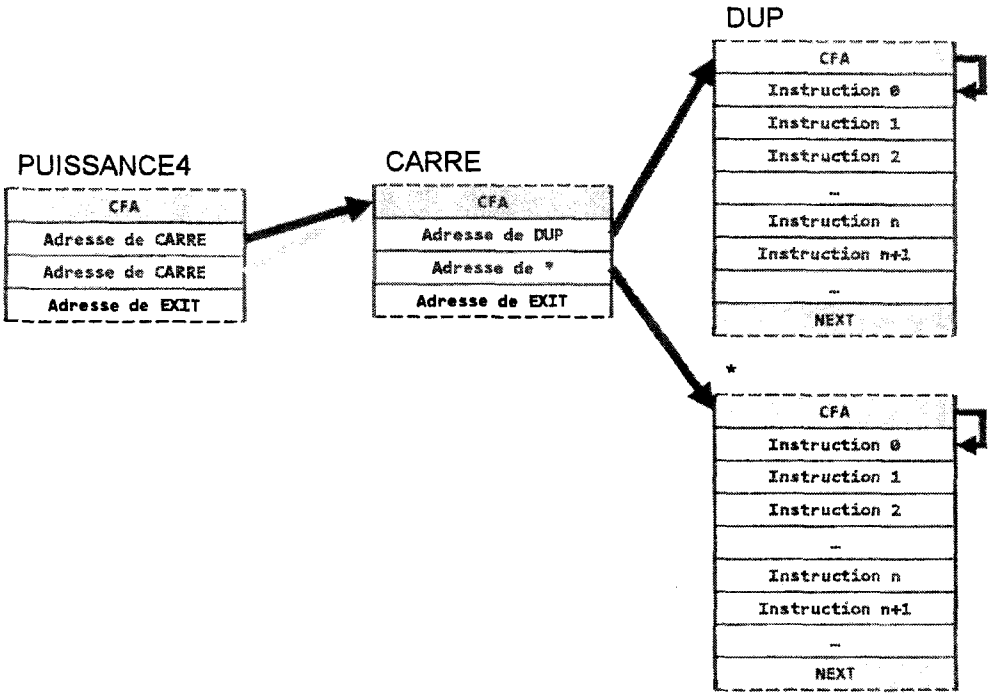

Figure 75 – Organisation des mots FORTH mise en place pour assurer le mécanisme d'exécution de l'interpréteur interne pour du code chaîné indirect. Nous montrons ici l'utilisation d'un mot rédigé en FORTH par un autre.

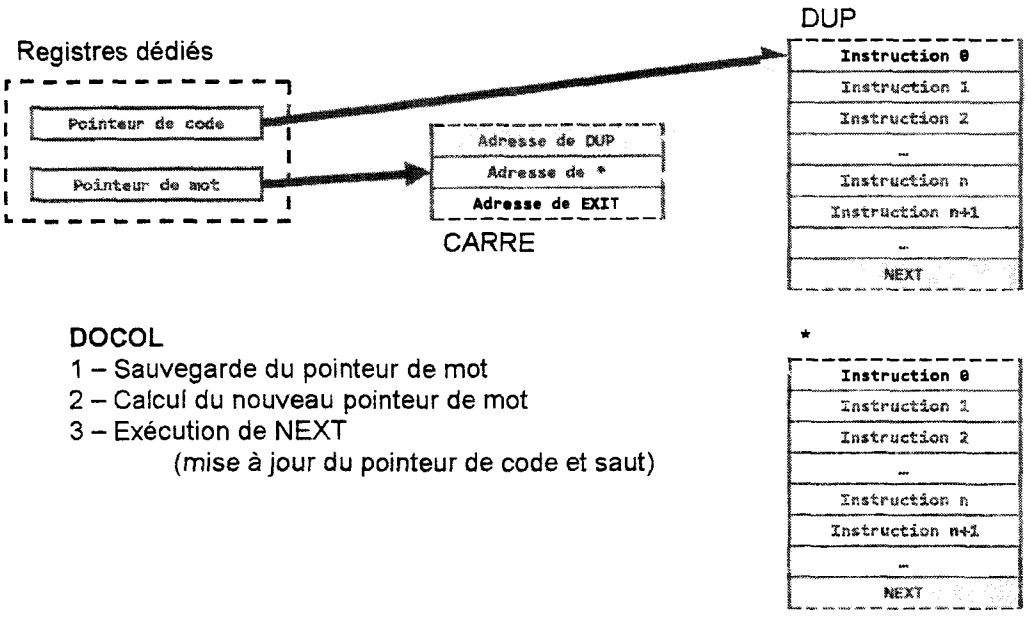

Figure 76 – Organisation des mots FORTH mise en place pour assurer le mécanisme d'exécution de l'interpréteur interne pour du code chaîné indirect et matérialisation des

registres dédiés, utilisés notamment par DOCOL. L'Exécution de NEXT de la figure fait référence au NEXT de DOCOL lui-même.

Sur la représentation des mots définis en *FORTH* dans nos figures, nous avons fait apparaître un nouveau mot. Il s'agir d'`EXIT` (ou `;S` selon le standard et l'implémentation). Ce mot parachève le mécanisme de l'interpréteur interne. En effet, c'est lui qui récupère l'ancienne valeur du pointeur de mot depuis la pile de retour de *FORTH* (où `DOCOL` l'a sauvegardé). Lorsque `NEXT` est exécuté pour `EXIT`, l'exécution continue là où elle a été interrompue dans le mot précédent. Nous nous en tiendrons à cette description sommaire du fonctionnement des interpréteurs de *FORTH*.

b. Le compilateur *FORTH*

Avant de clore cette introduction des fondamentaux de *FORTH*, nous devons encore présenter le rôle et le mécanisme de base du compilateur *FORTH*. L'organisation décrite lors de la présentation des interpréteurs de *FORTH* le laisse deviner, le compilateur *FORTH* est très simple. Son rôle et de générer pour une nouvelle définition son entrée dans le dictionnaire de *FORTH*. La Figure 77 donne une représentation de la structure simplifiée des entrées des mots `DUP` et `CARRE` dans le dictionnaire. Le champ `lien` est propre au dictionnaire et permet la recherche des chaînes de caractères isolées par l'interpréteur externe. Dans certaines implémentations, le chaînage peut être double ou plus complexe pour de meilleures performances. Le champ `longueur + cfg` encode la longueur du nom du mot en caractères ainsi que certains drapeaux dont nous ne détaillerons pas le rôle ici. Sachez simplement que ces bits sont utilisés notamment pour basculer dynamiquement entre le mode de compilation et d'interprétation interne, ce qui permet de définir par exemple, et ce pour un même mot, deux comportements différents. L'un est exprimé lors de la compilation et l'autre est exprimé pendant l'interprétation par exemple. Suivent ensuite les caractères qui composent le nom du mot. Enfin, le `CFA` du mot est suivit – s'il s'agit d'un mot défini en *FORTH* – des adresses des `CFA` des mots participant à la définition du mot (donc les adresses de début des mots). C'est pour cette raison que le compilateur *FORTH* est aussi décrit comme un compilateur d'adresses.

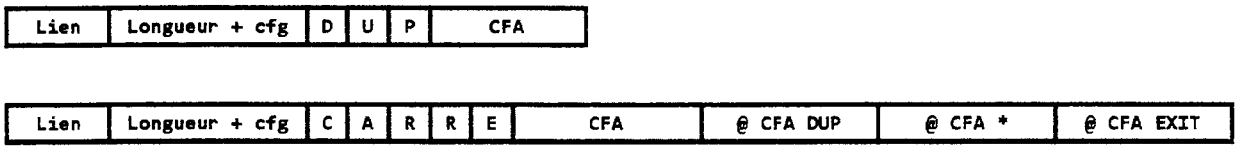

Figure 77 – Représentation simplifiée des entrées des mots DUP et CARRE tels qu'ils apparaissent dans le dictionnaire FORTH une fois compilés par le compilateur. Les alignements, etc. ne sont pas montrés

Nous allons à présent présenter les étapes essentielles qui mènent de la saisie de la définition d'un mot à la construction de son entrée dans le dictionnaire. En d'autres termes, à la compilation du nouveau mot *FORTH*. Comme nous l'avons présenté, l'interpréteur externe lit les mots dans le tampon de saisie. Il le fait *via* le mot WORD. Il les recherche ensuite dans le dictionnaire *via* le mot FIND et construit la liste des adresses de *CFA via* le mot >CFA. La destinée des adresses de *CFA* produites par >CFA dépend du mode de fonctionnement de l'interpréteur qui est défini par la variable d'état STATE. Si celle-ci est nulle, alors l'interpréteur exécute les mots à la volée – comme nous l'avons décrit au paragraphe précédent. Si la variable d'état est non-nulle, alors l'interpréteur est en mode compilation. Nous voyons donc que le compilateur *FORTH* n'est en fait que son interpréteur interne dans un mode particulier. Donc, le compilateur stocke les adresses de *CFA* dans la mémoire du système (*FORTH* gérant la variable HERE, un pointeur sur le premier octet libre de la mémoire du système).

Le mot de définition COLON (:) commence par utiliser WORD pour récupérer le nom du mot à compiler. Il construit ensuite une entrée de dictionnaire vide (l'entête uniquement – *via* le mot CREATE), à l'adresse HERE. Le champ link de l'entrée pointe sur la dernière entrée du dictionnaire (LATEST) et tous les champs, *CFA* inclus, sont initialisés. HERE est mis-à-jour et pointe à la fin du champ *CFA*. LATEST est également mis-à-jour pour pointer sur la nouvelle entrée. COLON positionne STATE à 1 (une valeur non-nulle) pour basculer l'interpréteur interne en compilateur. Celui-ci commence alors à ajouter les adresses de *CFA* des mots entrant dans la définition du mot en cours de définition à partir de l'adresse HERE (qui pointe rappelons-le sur la fin de la nouvelle entrée juste avant de quitter le mode d'interprétation).

Concrètement, avec le cas de CARRE (Figure 78), une fois que l'interpréteur est en mode de compilation, il lit la chaîne DUP et la recherche dans le dictionnaire, récupère son adresse *CFA* et l'ajoute après le *CFA* de la nouvelle entrée. Il fait de même avec le +. Pour éviter que SEMICOLON (;) ne soit compilé dans la nouvelle entrée, les bits que nous avons évoqués dans la description du champ Longueur + cfg prennent toute leur importance. L'un deux est IMMEDIATE. S'il est armé (égal à 1) pour un mot du dictionnaire, alors ce mot est immédiatement exécuté, même si l'interpréteur est en mode de compilation. C'est ainsi que SEMICOLON a son bit IMMEDIATE armé. SEMICOLON ajoute l'adresse *CFA* de EXIT et désarme la variable d'état STATE (l'interpréteur ne compile plus). Enfin, précisons l'existence d'un autre drapeau de configuration utilisé lors de la compilation. Il s'agit de HIDDEN qui, armé, empêche le mot FIND de trouver une entrée du dictionnaire. Pour éviter de compiler le mot en cours de compilation, le bit HIDDEN est armé en début de processus et désarmé en fin de compilation.

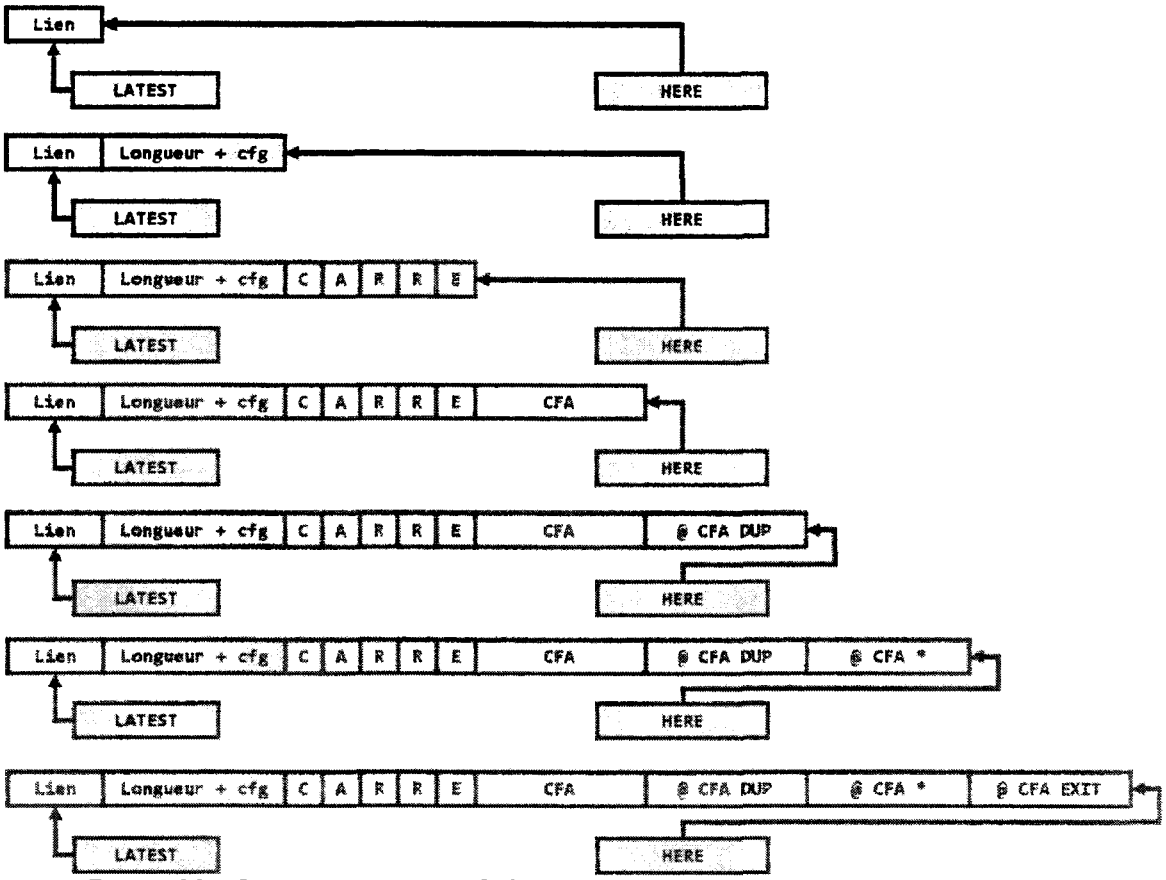

Figure 78 – Etapes successives de la compilation du mot CARRE.

3. Mesure de l'énergie consommée

La mesure de l'énergie consommée – exprimée en Joules – par un système (matériel et logiciel(s) confondu(s)) s'effectue en intégrant la puissance électrique consommée par ce système sur l'intervalle de temps de l'analyse. La puissance électrique consommée – exprimée en Watts – peut se mesurer à partir de différentes sources. Pour masquer la complexité engendrée par cette multiplicité des sources, nous avons développé un logiciel *serveur de puissance* (`power_server`) dont le rôle est de mesurer la puissance – et aussi de calculer l'énergie consommée – de façon transparente. Le *serveur de puissance* expose par la suite ces informations au système d'exploitation (*SE*) et aux applications. Ces derniers peuvent dès-lors consommer ces informations pour leurs besoins quels qu'ils soient (par exemple pour les enregistrer sous forme de traces, pour calculer l'énergie requise pour effectuer une tâche donnée, *etc.*).

Pour faciliter et standardiser l'échange de ces données, nous avons créé une interface logicielle (*Application Programing Interface – API*). Comme nous le verrons plus loin, les données échangées ne se limitent pas à la puissance et l'énergie, mais peuvent représenter / mesurer n'importe quelle métrique significative pour un problème donné. Cette *API* existe en deux versions. La première mouture est destinée aux développeurs souhaitant optimiser l'efficacité énergétique de leurs logiciels. Il s'agit de l'*API Power Link* qui s'utilise généralement sur un poste de travail unique. La seconde version de l'*API* est destinée aux professionnels des technologies de l'information qui cherchent à optimiser l'efficacité énergétique de leurs centres de calculs et d'hébergement. Il s'agit de l'*API Productivity Link* qui a été conçue pour être utilisée sur des milliers de serveurs. Sur le plan technique, les deux versions de l'*API* diffèrent principalement par la méthode de différenciation des sources de données (qui est unique avec *Power Link* et qui est illimitée avec *Productivity Link*), ainsi que par la méthode de stockage de l'information. Nous utiliserons dans cette annexe exclusivement la version *Power Link*. *Power Link* expose ses données *via* la base de données des performances du système d'exploitation (*Registry* dans le cas de *Microsoft Windows* ou bien le pseudo-système de fichiers `/proc` dans le cas de *Linux*).

Pour mesurer la puissance, notre *serveur de puissance* utilise soit un analyseur de puissance externe, soit les capteurs de puissance enfouis des alimentations électriques des serveurs et des stations de travail. La communication avec l'analyseur externe s'effectue préférentiellement par une interface `IEEE 488.2` (l'interface de commande standard pour les appareils de mesure, initialement conçue par *Hewlett-Packard*). La gestion de l'interface sérielle – et *USB* – est offerte pour les analyseurs d'entrée de gamme. Les capteurs enfouis sont lus quant-à eux *via* le contrôleur de carte mère (*Base Board Management Controller – BMC*) du serveur suivant les protocoles *IPMI* (*Intelligent Platform Management Interface*) et *PMBus* (*Power Management Bus*) (Figure 79). Le rôle du serveur de puissance est de mesurer et exposer les compteurs matériels. Le logiciel d'analyse à l'opposé, liera – ou importera – la valeur de ces compteurs pour, par exemple, l'enregistrer dans un fichier de *log* ou pour utiliser cette valeur à des fins algorithmiques. Enfin, le logiciel instrumentalisé exposera ses propres compteurs (compteurs logiciels tels que la performance, le nombre des opérations effectuées, *etc.*) et pourra

si cela est nécessaire importer et utiliser les compteurs matériels exposés par le serveur de puissance.

Cette seconde solution a l'avantage d'être autonome, d'être gratuite et de pouvoir accéder à la donnée même lorsque les processeurs sont hors service. En revanche, la précision de la mesure est généralement inférieure à celle obtenue par l'analyseur dédié (de l'ordre de ±5%). Enfin, certaines alimentations de serveurs d'entrée de gamme ne sont pas instrumentalisées.

Signalons enfin que pour une mesure plus fine de la puissance consommée, par exemple pour cibler un composant en particulier, la carte mère du système doit être instrumentalisée. Nous ne prévoyons pas de promouvoir cette solution car elle est trop difficile à déployer à moyenne et grande échelle. Pour nos mesures, nous avons instrumentalisé le logiciel d'interface du module *iPARK (Intel Power Acquisition Recording Kit)*.

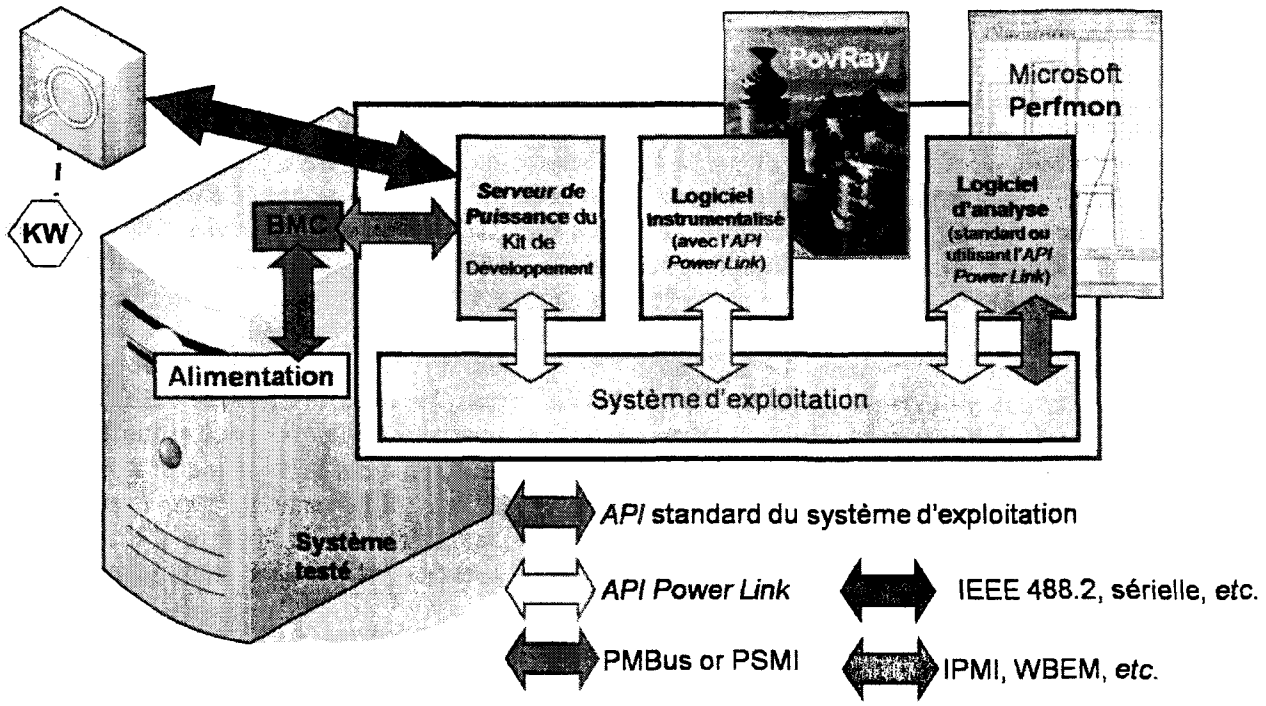

Figure 79 – Vue simplifiée de l'architecture de notre solution de mesure de l'énergie consommée en particulier – mais de n'importe quelle métrique significative pour un problème donné. BMC représente le microcontrôleur de la carte mère (Baseboard Management Chip). Les API du SE sont spécifiques à chaque OS (si ils existent), là où nous avons implémenté notre librairie sur les principaux ES du moment (Windowss, Linux, Solaris et MAC OSX).

a. Compteurs matériels et logiciels

Les mesures de la puissance et de l'énergie consommées sont accompagnées d'un ensemble de mesures environnementales qui sont également exposées au système d'exploitation

via Power Link. Nous donnons dans la Table 30 la liste de ces compteurs. Cette liste est fixe et prédéfinie avec *Power Link*. Ces données sont collectées – si les capteurs sont disponibles et à même de communiquer avec le *serveur de puissance* – essentiellement pour s'assurer que des mesures sont comparables. Par exemple, une même application peut-être exécutée et mesurée sur un même serveur mais à des altitudes très différentes. Avec la diminution de la densité de l'air, les ventilateurs seront plus sollicités à haute altitude et la consommation électrique augmentera. La température est la donnée environnementale la plus importante pour les mêmes raisons. Ainsi, certains *OEMs* (*Original Equipment Manufacturer*) n'ont pas hésité lors de la phase de mise au point du *benchmark SPEC Power* (Standard Performance Evaluation Corporation 2008) à tester leurs serveurs dans des chambres froides.

Compteur	Usage
PL_ENERGY	Mesure l'énergie consommée par le système (en joules)
PL_POWER	Mesure la puissance consommée par le système (en watts)
PL_VOLTAGE	Mesure le voltage (en volts)
PL_CURRENT	Mesure l'ampérage (en ampères)
PL_TEMPERATURE	Mesure la température ambiante (en degrés Celsius)
PL_PRESSURE	Mesure la pression atmosphérique (en Pascal)
PL_ALTITUDE	Mesure l'altitude (en mètres)
PL_REL_HUMIDITY	Mesure l'humidité relative ambiante (en %)

Table 30 – Liste et usages des compteurs relatifs au matériel et à l'environnement.

Nous avons également défini un ensemble de compteurs relatifs au logiciel (Table 31) pour exposer les informations de performance et de *travail utile* effectué. Cette liste est également prédéfinie avec *Power Link*. Nous les utilisons conjointement dans le cadre de l'analyse et de l'optimisation énergétique à des fins informatives et de marquage comme l'illustre la Figure 80. Précisons que lorsque nous parlons dans cet exemple d'*opérations*, c'est au sens macroscopique – par exemple un calcul d'intégrale, le traitement d'une requête d'un client, le rendu d'une image, *etc.* – et non pas au sens d'opérations arithmétiques.

Compteur	Usage
PL_PERFORMANCE	Mesure la performance de l'application telle que définie par le développeur
PL_LOAD_LEVEL	Mesure le niveau de charge de l'application (%)
PL_SW_PHASE	Mesure la phase logicielle de l'application telle que définie par le développeur
PL_RAW0 PL_RAW1	Compteurs génériques (PL_RAW0-PL_RAW4) mis à la disposition du développeur pour mesurer des paramètres tels que la performance par

PL_RAW2 PL_RAW3	Watt, le nombre total des opérations effectuées, l'énergie consommée par une opération, etc.

Table 31 – Liste et usages des compteurs relatifs au logiciel.

La Figure 80 – dont le rôle est illustratif – reproduit l'enregistrement dans le temps de certains compteurs matériels et logiciels que nous venons d'introduire. Elle mérite que nous nous y attardions pour en décrire la lecture – puisqu'elle n'utilise pas la forme habituelle des graphiques. Il s'agit ici de l'enregistrement dans le temps (axe des abscisses, gradué en secondes dans cet exemple) des compteurs significatifs pour cette mesure en particulier (nous restons volontairement imprécis car l'expérience en soit n'a pas d'importance pour nos propos). Nous préciserons les compteurs utilisés et la façon avec laquelle il faut interpréter les valeurs sur le graphe plus tard dans le texte. Signalons simplement que les compteurs matériels sont mesurés et exposés par le serveur de puissance précédemment introduit, et que les compteurs logiciels sont exposés par le logiciel en cours d'exécution.

Ces compteurs ont été enregistrés ici sur onze intervalles de vingt secondes, espacés par des intervalles d'inactivité de cinq secondes chacun. Le système d'alimentation ayant une inertie non négligeable – de une à deux secondes sur notre système de test – ce temps de pause est nécessaire pour ne pas fausser les mesures de la plage d'activité suivante. Le compteur PL_SW_PHASE délimite ces phases par sa courbe en pointillés légendés *Phase logicielle*. La phase logicielle est une valeur arbitraire sans unité et expose ici seulement deux états : *calcul* et *oisif* (à lire sur l'axe des ordonnées de gauche, bien que cela n'aie pas d'importance véritable avec notre choix binaire).

Pour les onze plages de mesures, nous avons fixé le niveau de charge que le logiciel doit atteindre et maintenir. Il s'agit ici d'une contrainte imposée à l'application en cours d'exécution. Le compteur PL_LOAD_LEVEL révèle la valeur de cette contrainte exprimée en pourcentages (légendée *Niveau de charge* et à lire sur l'axe des ordonnées de gauche). Ainsi, et dans l'ordre chronologique, nous forçons l'application à fonctionner à 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% puis 100% de sa capacité maximale. En pratique, 100% correspond à la performance maximale que l'application peut atteindre sur une plate-forme de tests donnée. Dans notre exemple, l'application répond à des requêtes de clients et la performance est exprimée en requêtes par secondes (exposée par le compteur PL_PERFORMANCE, légendée *Opérations par seconde* et à lire sur l'axe des ordonnées de droite). Ces opérations représentent des transactions et non pas des opérations logiques ou arithmétiques. La courbe pleine et noire représente cette donnée sur la figure. A 100%, l'application atteint ici une performance de ~260 requêtes par seconde. A un niveau de charge de 30%, l'application s'astreint à ne fournir qu'une performance de ~80 requêtes par seconde (visible dans le quatrième intervalle d'activité).

Les autres données sont calculées et exposées par l'application en utilisant les compteurs génériques PL_RAWi. Ainsi, en plus de la performance, nous collectons et représentons sur le graphique le nombre total des requêtes traitées par le logiciel (exprimé en requêtes, légendées *Opérations* et à lire sur l'axe des ordonnées de droite). Signalons que plus que la performance, nous préférons mesurer le *travail utile* effectué par un logiciel. La nature de ce

travail utile étant par nature infiniment variable, les compteurs génériques ont été créés à cet effet avec *Power Link*. C'est typiquement cette information que nous planifions de mesurer et d'utiliser avec le *Green Grid* (The Green Grid Consortium 2008). Il s'agit sur la figure des triangles qui représentent une fonction croissante monotone dans le temps. La pente de celle-ci est, comme attendu, proportionnelle à la performance. Signalons aussi que les décrochages constatés sur l'hypoténuse sont dus à des synchronisations entre les *threads* de travail. Enfin, le nombre des requêtes par Watts est représenté par la courbe pleine et en gris foncé.

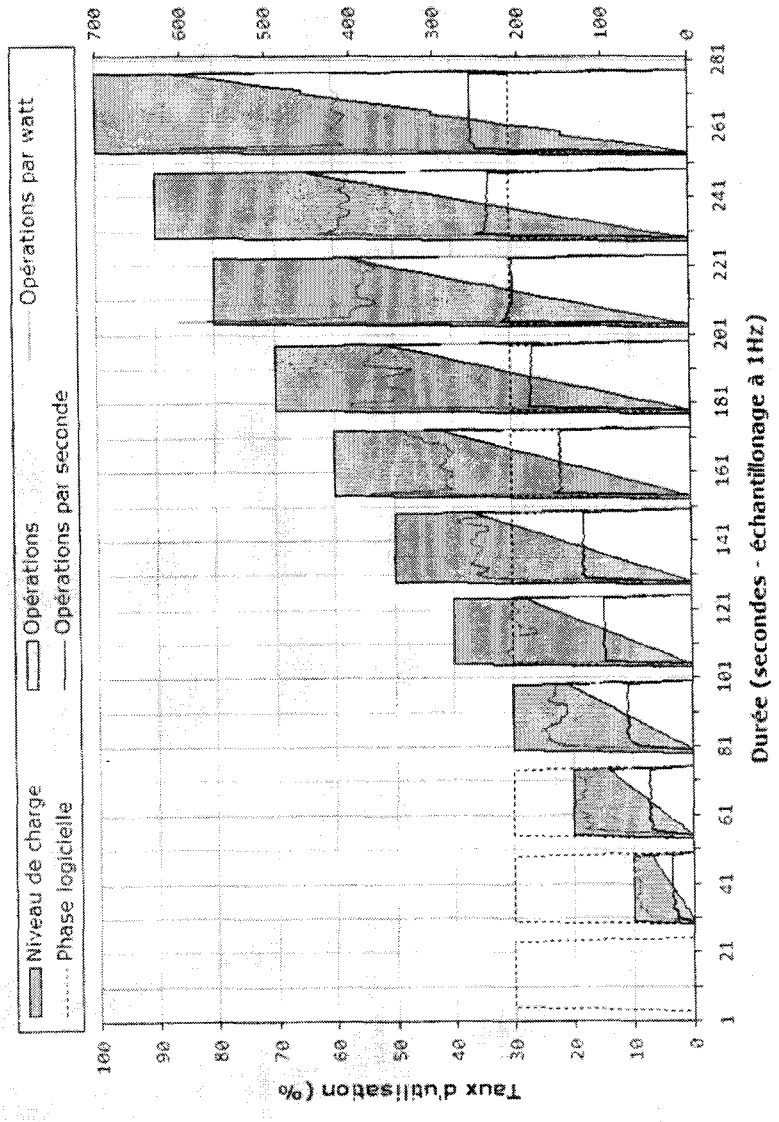

Figure 80 – Utilisation conjointe des compteurs matériels et logiciels pour corréler performance et énergie. Les compteurs logiciels informent sur l'activité de l'application, ce qui simplifie l'analyse des mesures d'énergie.

b. Recherche des points énergétiques

La recherche des points énergétiques s'effectue par un échantillonnage des compteurs ordinaux des processeurs ou des cœurs pendant la durée de l'analyse. L'unité de mesure de performances (*Performance Monitoring Unit – PMU*) est programmée pour générer périodiquement une interruption. La fréquence est fixée par défaut à 1 kHz. Le vecteur associé à cette interruption active alors le pilote logiciel qui mémorise la valeur du compteur ordinal, l'identifiant du processus léger (*thread*) et son niveau de privilège. L'ensemble des compteurs relatifs au matériel et au logiciel du *serveur de puissance* et de l'application sont aussi pris en compte. Il s'agit là de l'échantillon fondamental. Ces échantillons sont ultérieurement analysés et corrélés en utilisant des outils standards tels que l'analyseur de performances *VTune* d'*Intel Corporation* ou suivis en direct, avec par exemple l'analyseur de performances *Perfmon* de *Microsoft Corporation*. L'utilisation de *VTune* est rendue possible puisque les données sont enregistrées post-mortem dans un fichier au format *TB5* (format *VTune*).

La Figure 81 montre un exemple de suivi en direct des compteurs avec le logiciel de rendu d'image par lancé de rayons (*PovRay*).

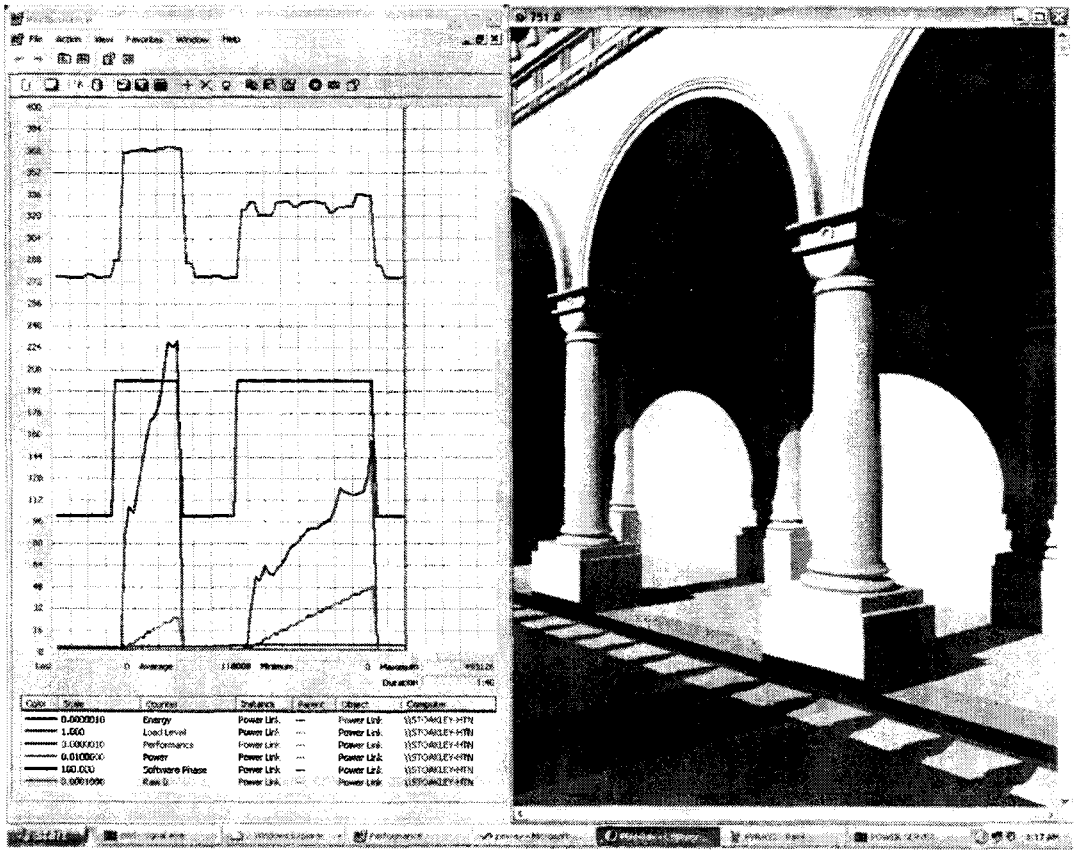

Figure 81 – Exemple de suivi en direct des compteurs pour le rendu d'image par le logiciel PovRay.

Sur cette figure, nous voyons à droite le rendu final et à gauche les traces collectées avec *Perfmon*. La légende de *Perfmon* donne la liste des compteurs collectés. Le compteur générique `PL_RAW0` est utilisé par notre version instrumentalisée de *PovRay* pour exposer la quantité d'énergie utilisée pour le calcul d'une image. Les données issues de deux exécutions consécutives du même calcul – menant au même résultat – sont représentées sur la figure et délimitées par la trace du compteur `PL_SW_PHASE` (courbe noire). La première exécution utilise deux *threads* de calculs, là où la seconde exécution n'en emploie d'un. Nous constatons un comportement conforme à ce que nous présentons dans le quatrième chapitre de ce document (lire la section *Optimisations de l'efficacité énergétique du chapitre 4*).

Pour illustrer le mécanisme de corrélation entre les compteurs architecturaux du processeur et la mesure de l'énergie, la Table 32 donne pour un intervalle de temps – sélectionné dans *VTune* (Figure 82) – la corrélation entre l'énergie consommée sur cet intervalle et les fonctions du programme qui ont été exécutées durant ce même intervalle.

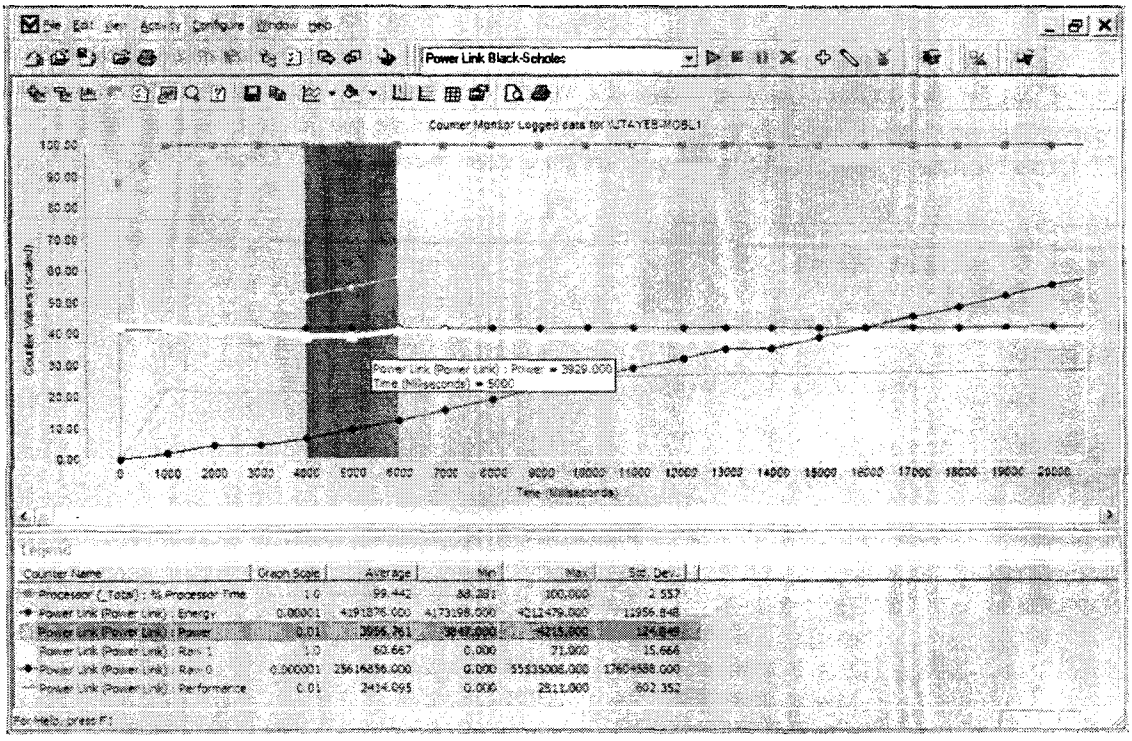

Figure 82 – Exemple de corrélation des compteurs matériels et logiciels et les compteurs architecturaux des processeurs. Il s'agit là de la première étape qui consiste à sélectionner l'intervalle de temps de l'analyse. Les compteurs matériels (puissance, énergie, etc.) sont utilisés pour sélectionner les intervalles pertinents.

Fonctions	*Energie*	*% Cycles*	*% Instructions*
`compute_funtion`	`1.125,45`	`74.00`	`81.48`
`<unresolved addresses>`	`Joules`	`12.00`	`11.11`

thread_function	10.00	7.41
RTC_CheckEsp	4.00	0.00

Table 32 – Plages d'adresses virtuelles associées à une quantité d'énergie consommée mesurée sur un intervalle de temps donné.

Les fonctions sont représentées sous la forme de plages d'adresses virtuelles, elles-mêmes exprimées comme un déplacement par rapport à l'adresse de base du module dans lequel le compteur ordinal a été capturé pour un échantillon. Ces adresses sont calculées en soustrayant du compteur ordinal l'adresse de début du module. Par exemple, la fonction N (dont le code est donné dans la Table 33) attire notre attention car elle consomme près de 65% des cycles non-arrêtés pour l'exécution du programme pendant un pic d'énergie consommée de 1.125,45 joules. La Table 33 donne l'association entre les lignes de code de la fonction N et les adresses virtuelles, calculées comme indiqué précédemment. Les informations symboliques sont extraites de la base de données de mise au point du programme (*debug database*).

La Table 34 donne le niveau d'analyse supérieur en indiquant les adresses virtuelles des points d'entrée de toutes les fonctions du module d'intérêt. Les adresses non résolues correspondent en général au code généré à la volée par exemple par le compilateur *Just In Time – JIT*. Cette limitation devrait être levée avec la prochaine mouture de *VTune*.

Adresses virtuelles	*Code source*
	`double N(double X)`
0x385A0	`{`
	`  double Y, Z, Result ;`
0x385BE	`  Result = 0 ;`
0x385C3	`  Z = (X>0) ? X:-X ;`
0x385EE	`  if (Z <= 7) {`
0x38602	`  Y = (Z * Z) * 0.5 ;`
0x38611	`  if (Z > 1.28) {`
	`  Result = 0.398942280385 * exp(-Y) /`
	`  (Z - 0.000000038052 + 1.00000615302 /`
	`  (Z + 0.000398064794 + 1.98615381364 /`
	`  (Z - 0.151679116635 + 5.29330324926 /`
	`  (Z + 4.8385912808 - 15.1508972451 /`
	`  (Z + 0.742380924027 + 30.789933034 /`
0x38621	`  (Z + 3.99019417011)))))) ;`
	`  }`
0x3869D	`  else {`
	`  Result = 0.5 - Z * (0.398942280444 - 0.399903438504 * Y /`
	`  (Y + 5.75885480458 - 29.8213557808 /`
	`  (Y + 2.62433121679 + 48.6959930692 /`
0x3869F	`  (Y + 5.92885724438)))) ;`
	`  }`

```
                }

0x386E7       if (X > 0) {
0x386F3         Result = (1 - Result) ;
                }

0x386FB         return(Result) ;
0x386FE       }
```

Table 33 – Adresses virtuelles relatives associées aux lignes de code de la fonction N.

Nom	*Déplacement*	*Adresse virtuelle*
N	0x205A0	0x385A0
BlackScholes	0x20770	0x38770
<unresolved addresses>	0x0	0x0
FullIntegral	0x203C0	0x383C0
RTC_CheckEsp	0x2AC90	0x42C90
exp	0x2B350	0x43350
sqrt	0x2B356	0x43356

Table 34 – Adresses virtuelles relatives et les déplacements par rapport au début du module des points d'entrée des fonctions du programme en cours d'analyse.

Une fois la corrélation établie entre les fonctions et l'énergie consommée, nous pouvons remonter la chaine et enrichir notre compréhension de l'application par différents niveaux d'analyse. Cela est d'une grande utilité afin d'optmiser le code pour les performances et l'énergie. Ainsi, la Table 35 nous donne pour les fonctions identifiées plus tôt, le nombre de cycles non-arrêtés et le nombre d'instructions non-annulées – exprimés en % du total de ces valeurs pour le module. Nous avons ainsi le nombre de cycles par instruction (*CPI*) qui est en général un bon indicateur de performances sous-optimales s'il est élevé. Les Table 36, Table 37 et Table 38 fournissent ces informations à des niveaux d'analyse croissants. En plus de la valeur illustrative, cela montre la façon dont nous recommandons d'exploiter ces données.

Nom de fonction	*Cycles par instruction*	*% Cycles*	*% Instructions*
N	5,32	64,97	44,61
BlackScholes	2,26	31,48	50,84
<unresolved addresses>	3,12	2,07	2,42
FullIntegral	2,75	8,80	1,17
RTC_CheckEsp	1,10	2,20	0,73
exp	3,33	0,20	0,22
sqrt	0,00	0,18	0,00

Table 35 – Informations relatives à l'exécution des fonctions.

Modules	*Processus*	*Cycles par instruction*	*% Cycles*	*% Instructions*
BlackScholes.exe	BlackScholes.exe	3,65	70,34	55,41
msvcr80d.dll	BlackScholes.exe	1,83	28,37	44,55
hal.dll	BlackScholes.exe	0,00	1,07	0,00
ntoskrnl.exe	BlackScholes.exe	0,00	0,17	0,00
NETw4x32.sys	BlackScholes.exe	3,00	0,04	0,04

Table 36 – Informations relatives aux modules exécutés.

Threads	*Processus*	*Cycles par instruction*	*% Cycles*	*% Instructions*
thread70	BlackScholes.exe	2,89	54,12	54,00
thread71	BlackScholes.exe	2,86	45,86	46,00
thread76	BlackScholes.exe	2,00	0,03	0,00

Table 37 – Informations relatives aux threads exécutés.

Processus	*Cycles par instruction*	*% Cycles*	*% Instructions*
BlackScholes.exe	2,88	89,17	80,03
FrameworkService.exe	1,19	4,87	10,57
svchost.exe	1,97	1,71	2,24
S24EvMon.exe	1,02	0,69	1,76
services.exe	1,42	0,72	1,30
ifrmewrk.exe	0,70	0,33	1,20
BESClient.exe	2,47	1,12	1,17
vmware-authd.exe	2,36	0,33	0,36
IEXPLORE.EXE	1,10	0,14	0,33

Table 38 – Informations relatives aux processus exécutés.

D'un point de vue pratique, la lecture des données se fait dans l'ordre inverse. Nous commençons toujours par la sélection d'un intervalle d'intérêt – qui correspond à un pic de consommation d'énergie dans le temps. La Table 38 nous indique que le processus `BlackScholes` consomme ~90% des cycles non-arrêtés et ~80% des instructions non-annulées. En outre, un *CPI* de ~2.9 nous indique clairement qu'il y a un fort potentiel d'optimisation pour ce programme. En revanche, nous ne pouvons pas ventiler objectivement l'énergie consommée au prorata des instructions non-annulées. La Table 37 nous donne pour le processus `BlackScholes` le *CPI* – et donc les données nécessaires à son calcul, nous ne les répèterons plus par la suite – pour les *threads* du programme. Nous apprenons ici que deux *threads* de travail prédominent. Le troisième *thread* semble être un *thread* de contrôle. En se concentrant sur les deux *threads* de travail, la Table 36 liste les modules exécutés par les *threads* de travail avec le *CPI* associé. Ici, nous voyons que c'est le code du programme en lui-même qui est le plus exécuté. Si le processus `BlackScholes` avait été dominé par exemple par des entrées-sorties, alors nous aurions vu apparaître les routines du noyau associées dans la liste. En nous concentrant sur le code de

`BlackScholes`, les Table 35 et Table 33 nous donnent enfin le détail des fonctions et des lignes de code très probablement responsables de l'énergie consommée pendant l'intervalle de temps d'intérêt sélectionné en début d'exercice. Si une optimisation énergétique doit être effectuée, alors elle devra prendre place dans ces fonctions.

c. Instrumentaliser les applications

L'instrumentalisation du code est nécessaire pour permettre à une application d'exposer les informations relatives à son exécution comme nous l'avons défini au début de ce chapitre. C'est aussi l'opportunité pour l'application de connaître dynamiquement son environnement énergétique, et y réagir suivant des règles établies par le développeur ou l'utilisateur de l'application. Le résultat d'une telle instrumentalisation a été présenté sur la Figure 80. La courbe `PL_RAW0` de la Figure 81 – qui représente rappelons-le l'énergie consommée par le calcul d'une image – est obtenu par ce biais. Ainsi, en début de calcul, le programme lit la valeur de `PL_ENERGY` exposée par le *serveur de puissance*. Pendant le calcul, et suivant la fréquence d'actualisation des compteurs – 1 Hz ici – la valeur de `PL_ENERGY` est lue, la différence avec la valeur initiale est calculée et `PL_RAW0` est exposé. Lorsque le programme ne calcule pas, `PL_RAW0` est mis à zéro.

L'utilisation de la librairie `power_link` est très simple. Nous avons choisi à cet effet une interface analogue à celle de la gestion des fichiers qui est maîtrisée par la vaste majorité des développeurs. Le schéma d'utilisation se résume à ouvrir un `power link` – l'équivalent d'un fichier – en écriture ou en lecture – suivant que l'on souhaite exposer une donnée ou l'importer (`pl_open`). L'écriture et la lecture se font *via* les appels à `pl_read` et `pl_write`. Enfin, le(s) `power link` ouvert(s) sont fermé(s) (`pl_close`). La Table 39 présente le code typique d'un client et d'un serveur utilisant l'*API Power Link*. Notez qu'un code peut à la fois être client et serveur.

code typique d'un serveur
(par exemple celui du *serveur de puissance*)

```
void main(void) {
        double p, e;
        int pld = pl_open(NULL, PL_WRITE);
        while(running) {
                p = get_power();
                e = get_energy();
                pl_write(pld, &p, PL_POWER);
                pl_write(pld, &e, PL_ENERGY);
        }
        pl_close(pld);
}
```

code typique d'un client
(par exemple celui de *PovRay*)

```
void main(void) {
        double p, l, w, ppw;
        int pld_r = pl_open(NULL, PL_READ);
        int pld_w = pl_open(NULL, PL_WRITE);
        while(running) {
                p = get_performance();
                l = get_load();
                pl_write(pld_w, &p, PL_PERFORMANCE);
                pl_write(pld_w, &l, PL_LOAD_LEVEL);
                pl_read(pld_r, &w, PL_POWER);
                ppw = p / w;
                pl_write(pld_w, &ppw, PL_RAW0);
        }
        pl_close(pld_r);
```

```
                                        pl_close(pld_w);
                                    }
```

Table 39 – Structure générale d'un code serveur et d'un code client utilisant l'API Power Link pour échanger leurs données. Les données transitent via le SE.

d. Autorégulation des performances

Nous avons appliqué cette transformation au code *Black-Scholes* (SunGard 2008). Ce code propriété de *SunGard* nous a été exceptionnellement ouvert à des fins de tests. Il effectue un ensemble de simulations financières en utilisant l'algorithme de *Fischer Black* and *Myron Scholes* (Black et Scholes 1973). Nous avons choisi ce code car il appartient à la classe des applications les plus difficiles à transformer. En effet, les applications clients-serveurs classiques sont d'excellentes candidates à la transformation puisqu'elles sont *multithreadées* et répondent quasi-linéairement à la montée en charge. Cette charge étant exercée par les clients. Cette précision a son importance puisqu'elle rappelle que l'application ne contrôle pas sa charge, mais que c'est son environnement qui la lui impose. Il est toutefois relativement aisé de réintégrer ce contrôle – et avec lui celui de la consommation d'énergie – au sein de ces programmes. C'est d'ailleurs ce qui a dicté le choix de *SPEC Java Server* (Standard Performance Evaluation Corporation 2005) pour la réalisation de *SPEC Power*.

Dans le cas de *Black-Scholes*, rien de tel n'est possible avec le code initial. Pour y parvenir, nous devons en premier lieu définir la métrique de performance qui servira à l'étalonnage des niveaux de charges. Nous utilisons une solution classique – mais pas exclusive – qui consiste à compter le nombre des opérations effectuées. Ici, opération désigne le nombre d'intégrales calculées par seconde. Une fois définie et calculée dans le programme, cette métrique est exposée suivant la méthode décrite précédemment en utilisant l'*API Power Link*.

En utilisant la même méthode, nous exposons les compteurs relatifs à l'activité du logiciel (`PL_SW_PHASE`). Nous y adjoignons l'énergie consommée en fin d'exécution. Le Listing 25 montre l'ajout du code additionnel nécessaire pour y parvenir. Le code originel étant grisé dans l'ensemble de nos listings. A titre d'information, les noms des fonctions préfixées par `plh_` signifient qu'il s'agit d'appels à la librairie `power_link_helper`. Il s'agit d'un ensemble de routines destinées à simplifier la programmation de tâches classiques. Elles n'appartiennent pas à proprement parler à l'*API Power Link*. Le prefix `pl_` signal un appel à la librairie `power_link` et `_pl_` à la librairie interne `_power_link` (normalement non-exposée).

```
sTime = GetTickCount(); // # milliseconds écoulées depuis le boot
plh_signal_starting(pl_w);
plh_start_energy_measurement(pl_r);
result = BenchmarkFunction(nSteps, numThreads);
eTime = GetTickCount() ;
```

```
plh_stop_energy_measurement(pl_r, &consumed_energy);
```

Listing 25 – Ajout du code nécessaire au calcul de l'énergie consommée par le benchmark en fin d'exécution.

L'instrumentation du code devant être la plus légère possible, nous avons choisi le site d'appel de la fonction d'intégration pour incrémenter le compteur des opérations. Cette addition est protégée par une section critique. Il s'agit d'un coût raisonnable en comparaison du nombre des instructions nécessaires au calcul d'une intégrale. Si le coût de la synchronisation est trop élevé, alors chaque *thread* peut disposer d'un compteur local qui sera sommé par le *thread* responsable de l'exposition des compteurs logiciels. Nous préférons cette dernière approche – en recommandant de contrôler l'alignement des données en mémoire pour éviter le *false sharing* entre les *threads*. Notre librairie de support `power_link_helper` propose un ensemble de fonctions pour simplifier la gestion d'un tel *thread*. Le Listing 26 montre son utilisation dans notre cas d'étude.

```
...
PLH_THREAD_CONTROL plh_thread_control;
BLACK_SCHOLES ops;
...
        plh_start_thread(
        &plh_thread_control,
        &ops,
        black_scholes_performance_function
);
...
        plh_stop_thread(&plh_thread_control);
...
```

Listing 26 – Seule la rédaction de la fonction black_scholes_performance_function est à la charge du développeur.

Le *thread* de support (*helper thread*) est automatiquement appelé à intervalle de temps régulier. Dans la pratique, une fréquence d'appel d'un hertz s'avère largement suffisante et n'impacte pas les performances du code initial. Nous recommandons en général de limiter l'étendue des calculs effectués dans cette fonction comme une règle de précaution. Toutefois, la priorité est de minimiser l'impact sur les *threads* de calculs, quitte à charger le *thread* de support et à diminuer sa fréquence d'activation. Le Listing 27 illustre cette fonction qui calcule et expose le nombre d'intégrales calculées, le nombre d'intégrales calculées par seconde et ce même nombre par Watt. Il serait par exemple tout aussi simple de maintenir le nombre de joules par intégrale (comme nous l'avons montré avec *PovRay*). Le Listing 28 présente le point de collection du nombre des intégrales calculées.

```
void.black_scholes_performance_function(PPLH_THREAD_CONTROL p) {

      pl_write(p->plh_thread_pl, &p->operations, PL_RAW0);
      eTime = GetTickCount();
      performance = (double)p->operations / (eTime- sTime);
      ull_performance = data_scale_value(performance, 1.0, 0);
      pl_write(p->plh_thread_pl, &ull_performance, PL_PERFORMANCE);
      plh_get_power_measurement(p->plh_thread_pl_r, &power);
      ull_performance_per_watt = data_scale_value(
      performance / power, 1.0, 0
);
      pl_write(p->plh_thread_pl, &ull_performance_per_watt, PL_RAW1);
}
```

Listing 27 – Code de la fonction de performance de Black-Scholes.

```
DWORD WINAPI FullIntegral(LPVOID pArg) {
…
      static PBLACK_SCHOLES p =
(PBLACK_SCHOLES)(plh_thread_control.plh_thread_custom_data);
…
#ifdef INSTRUMENTATION
            EnterCriticalSection(&guard);
            p->operations++;
      LeaveCriticalSection(&guard);
#endif // INSTRUMENTATION
            Local_Result = Local_Result + BlackScholes(100, 110, x, 0.05,
0.05, 0.1) * dx;
      } // while
…
}
```

Listing 28 – Site de calcul du nombre des intégrales calculées. La section critique peut-être supprimée si chaque thread dispose de son compteur local – ce que nous recommandons en général.

A présent que nous pouvons exprimer le niveau de performance du code (PL_PERFORMANCE), nous pouvons pareillement imposer un niveau de performance à maintenir

suivant une approche similaire à celle usitée par *SPEC Power* (`PL_LOAD_LEVEL`). En appliquant notre stratégie de transformation énoncée dans le quatrième chapitre, nous allons placer les processeurs en veille périodiquement – ce code ne faisant que peu d'entrées-sorties par ailleurs, tout en conservant tous les *threads* de calcul actifs.

D'un point de vue pratique, la mise en sommeil des *threads* de calcul s'effectue en ajoutant un appel à la fonction système `sleep` dans la boucle d'appel des fonctions de calculs. La durée du sommeil sera proportionnelle au niveau de charge souhaité.

```
...
        if(p->plh_pause > 0) {
                SleepEx(p->plh_calibration_pause, FALSE);
        }
...
```

Listing 29 – code typique de mise en sommeil des threads de calculs. La richesse des options – en particulier de réactivation des threads ou la durée de sommeil – dépend de la sophistication du système d'exploitation hôte. Ici Microsoft Windows.

C'est en contrôlant la valeur de la durée de sommeil que l'application peut fixer son niveau de charge tout en laissant les mécanismes de conservation d'énergie du matériel s'activer et remplir leur œuvre.

La logique de contrôle doit aussi pouvoir calibrer automatiquement les temps de pause. En effet, une calibration manuelle et statique est toujours possible, mais cela s'avère inexploitable dans le cadre d'un déploiement à grande échelle. Nous avons intégré à cet effet la fonction `plh_calibrate` à notre librairie de soutien. Suivant un principe similaire à celui du *thread* de support pour le calcul et l'exposition de la performance, nous créons deux *threads* pendant la phase de calibration. Le premier effectue les appels aux fonctions de calculs du programme alors que le second contrôle l'activation et l'arrêt de ce même *thread*. En effet, nous sommes ici dans un schéma de calcul en temps constant et non plus en nombre d'opérations constant (notre règle d'invariance des résultats énoncée au quatrième chapitre n'a pas raison d'être appliquée lors de la phase de calibration automatique). Une fois que le laps de temps imparti à une itération de calibration est écoulé, la performance est calculée – elle peut d'ailleurs simplement être lue depuis le système d'exploitation puisque le *thread* de support l'expose déjà (le serveur est alors son propre client) – et est comparée à la performance cible. Par une simple recherche dichotomique, `plh_calibrate` construit la table des durées de mise en sommeil pour le couple (application, plateforme de test). Le Listing 30 montre l'appel d'une telle calibration.

```
...
        int ret = PL_ERROR;
        int i = 0;
        PLH_THREAD_CONTROL control;
```

```
            ret = plh_calibrate(&control, black_scholes_compute_function,
black_scholes_compute_performance_function);
            assert(ret == PL_SUCCESS);
      ...
      }
```

Listing 30 – Code de calibration en phase d'initialisation de l'application.

La Figure 83 montre la trace de calibration du programme *Black-Scholes* effectuée avec la fonction de support `plh_calibrate` juste à la fin de l'initialisation du logiciel. La table des durées de pause des *threads* de calculs ainsi générée est ensuite utilisée par le code d'appel des intégrations pour positionner la puissance électrique consommée par le logiciel en fonction de la charge logicielle imposée au programme (généralement par l'utilisateur).

La surface sous la courbe noire continue (`PL_POWER` – *puissance*, lue sur l'ordonnée de gauche) montre les variations de l'énergie consommée par *Black-Scholes* entre les différents niveaux de charge. Rappelons que le programme se comporte ici comme le *benchmark SPEC Power*, et qu'il ne respecte donc pas la règle d'invariance des résultats pendant la calibration. Donc, si sur la trace de calibration, nous pouvons supposer qu'entre les niveaux de charge de 20% et 80% par exemple (`PL_LOAD_LEVEL`, *Niveau de charge*, sur l'ordonnée de droite) le programme consomme en moyenne environ 5000 Joules de moins (à 20% par rapport à 80%), cela est en fait faux si l'utilisateur entend estimer la consommation d'énergie pour calculer un nombre d'intégrales fixe. En revanche, si ce n'est pas le cas, alors l'assomption est bien entendu vraie.

La courbe noire en pointillés (`PL_SW_PHASE`, *Phase logicielle*, sur l'ordonnée de droite), permet d'identifier visuellement les mesures successives effectuées par les *threads* de calibration (21 mesures au total). La première mesure – à l'extrême gauche du graphe – représente le niveau de charge (courbe mauve surface bleue ciel – *Performance cible*, lue sur l'ordonnée de droite) et la performance (courbe verte – *Performance cible*, lue sur l'ordonnée de droite) maximale de l'application. Ces données servent de référence au calcul des performances à atteindre pour les niveaux de charge intermédiaires (10%, 20%, *etc.*). Une fois la référence à 100% mesurée, les *threads* de calibration commencent par rechercher la durée de mise en pause requise pour atteindre une charge de 10%. Sur la trace de la Figure 83, il s'agit des mesures successives de 1 à 10. C'est le temps de pause le plus long à trouver pour *Black-Scholes* car la valeur d'amorce de la recherche dichotomique est un peu trop haute pour ce code. Par la suite, le niveau de charge de 20% est calibré par les mesures 11 à 13. 30% est calibré entre les mesures 14 à 16, *etc.* Pour suivre visuellement le niveau de charge fixé par le programme pendant la calibration, repérez la courbe verte (*Performance cible*, lue sur l'ordonnée de droite) ainsi que la courbe bleue clair (*Performance*, lue sur l'ordonnée de droite). Enfin, Il est aussi intéressant de noter que le taux d'utilisation des processeurs – courbe grise (*% Processeur*, lue sur l'ordonnée de droite) est un très mauvais indicateur de performance. Après cette transformation, le code *Black-Scholes* peut être invoqué avec un niveau de charge cible depuis la ligne de commande.

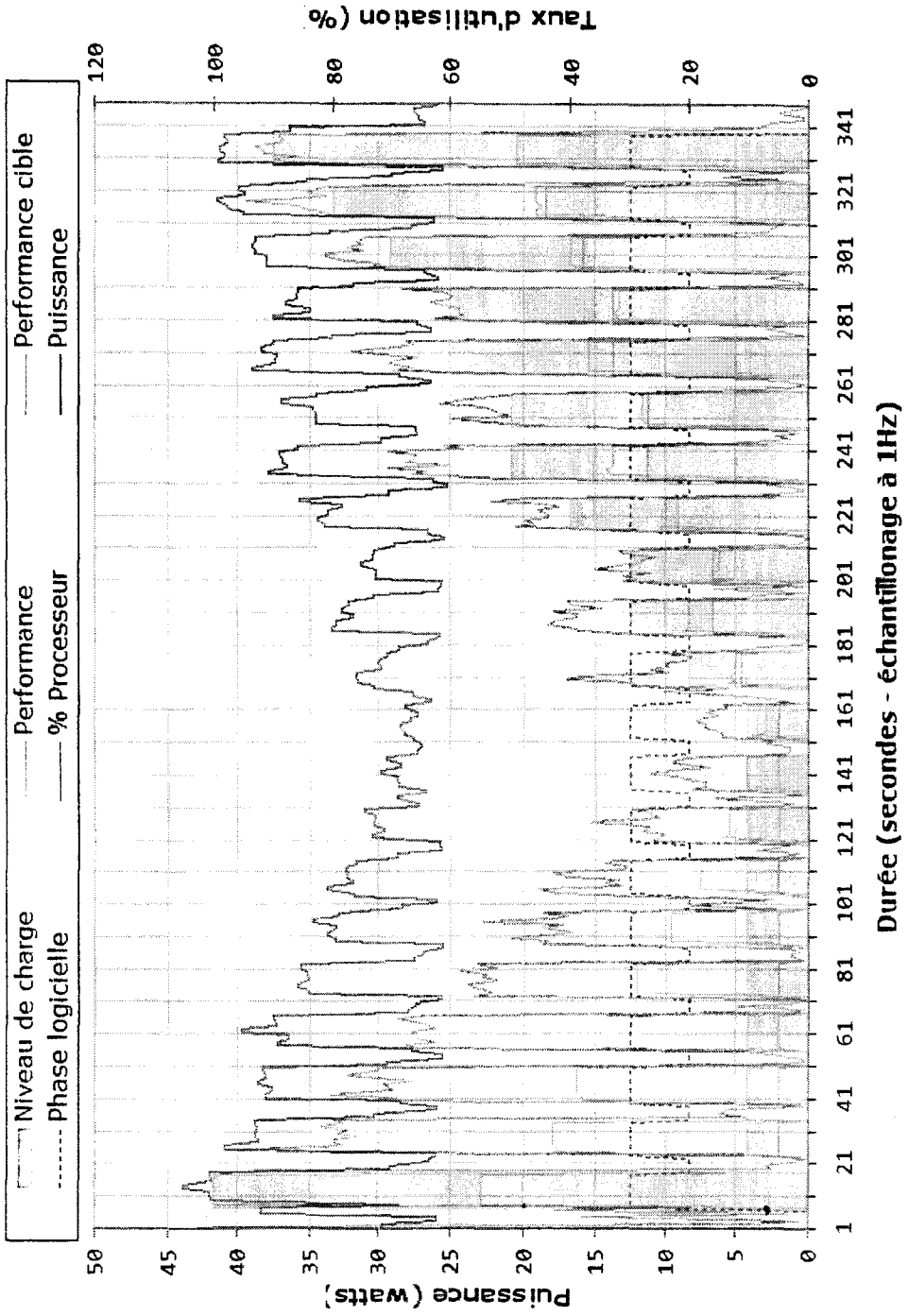

Figure 83 – Trace de calibration automatique du code Black-Scholes.

La différence entre les Figure 80 et Figure 83 réside dans l'analyse effectuée par le logiciel. La Figure 80 représente une application instrumentalisée dont le niveau de charge est imposé par l'utilisateur (dans ce cas précis *via* un signal émis par une interface graphique développée à cet effet). La Figure 83 quant à elle, représente une application qui s'auto-calibre pour déterminer les durées de pause de ses threads de calculs sur la plate-forme de test.

4. Références

Black, Fischer, et Myron Scholes. « The Pricing of Options and Corporate Liabilities. » (Journal of Political Economy) 81, n° 3 (1973).

Intel Corporation. *Intel Itanium 2 Processors Reference Manual for Software Development and Optimization.* Vol. 1, 2, 3. Intel Corporation, 2008.

Standard Performance Evaluation Corporation. « SPEC – Power and Performance - User Guide. » *SPECpower_ssj2008 V1.00.* 2008.

Standard Performance Evaluation Corporation. « *SPEC JBB2005* ». SPEC. 1 Aout 2005. http://www.spec.org/jbb2005/ (accès le Aout 1, 2008).

SunGard. « Black-Scholes. » *SunGard.* Intel Corporation, 2008.

The Green Grid Consortium. The Green Grid. 8 Septembre 2008. http://www.thegreengrid.org/home (accès le September 8, 2008).

Trimper, Gregory L., et James S. Evans. *Itanium Architecture for Programmers: Understanding 64-Bit Processors and EPIC Principles* . HP Professional Series, 2003.

Optimisation des performances pour les machine virtuelles et de l'énergie dans les architectures Intel/EPIC

Résumé

Cette thèse propose, dans sa première partie, d'étendre l'architecture *EPIC* des processeurs de la famille *Itanium* par l'ajout d'une pile matérielle. L'idée principale est qu'il est possible de combler l'écart de performance entre une architecture généraliste et les circuits spécialisés pour exécuter des machines virtuelles (*FORTH, .NET, Java, etc.*). Pour ce faire nous proposons de réassigner dynamiquement un sous-ensemble des ressources existantes d'*EPIC* pour offrir une pile d'évaluation matérielle. Deux implémentations, non-intrusives et respectant la compatibilité binaire des applications existantes, sont proposées. La principale différence entre ces piles réside dans leur gestionnaire: logiciel ou matériel. La pile d'évaluation sous le contrôle du matériel présente des fonctions avancées comme le support des piles d'évaluation typées promues par la *CIL* de *.NET*. Ainsi, nous proposons un traducteur simple-passe de binaire *CIL* en binaire *EPIC*, utilisant la pile d'évaluation matérielle. Dans la seconde partie de cette thèse, nous avons étudié l'efficacité énergétique des applications sur les architectures *Intel*. Nous avons ainsi défini dans un premier temps une méthodologie et des outils de mesure de l'énergie consommée et la quantité de travail utile fournie par les logiciels. Dans un second temps, nous avons entamé l'étude de transformations de code source afin de réduire / contrôler la quantité d'énergie consommée par les logiciels.

Mots clefs : machine virtuelle, pile d'évaluation, *EPIC*, *VLIW*, efficacité énergétique

Virtual Machines' Performance and Energy Optimization for Intel Itanium/EPIC architecture

Abstract

This thesis proposes, in its first part, to extend the EPIC architecture of the *Itanium* processor family by providing a hardware stack. The principal idea defended here is that it is possible to close the existing performance gap between generic architectures and application specific designs to run virtual machines (*FORTH, .NET, Java, etc*). With this intention, we propose to reallocate dynamically a subset of the EPIC architecture's resources to implement a hardware evaluation stack. Two implementations are proposed, both non-intrusive and compatible with existing binary codes. The fundamental difference between these stacks lies in their manager: software or hardware. The hardware controlled evaluation stack offers support for advanced functions such as the support of strongly typed evaluation stacks required by *.NET*'s *CIL*. Thus, we propose a single pass *CIL* binary translator into *EPIC* binary, using the hardware evaluation stack. In the second part of this thesis, we studied the energy efficiency of software applications. First, we defined a methodology and developed tools to measure the energy consumption and the useful work provided by the software. In a second time, we engaged the study of source code transformation rules in order to reduce/control the quantity of consumed energy by the software.

Keywords: virtual machine, evaluation stack, EPIC, VLIW, energy efficiency