Migration Alpha - Itanium sous OpenVMS

Auteurs :

ABOUNADA Sami
BOUKADIDA Mohamed J.
KSOURI Sonia

Encadrants :

M. BEDIN Christophe
M. LECARME Olivier

Version 1.6 (09/06/2005)

REMERCIEMENTS

Nous tenons à remercier tous ceux qui nous ont aidés, d'une manière ou d'une autre, pendant ce travail d'étude et de recherche.
Tout particulièrement monsieur Olivier Lecarme pour ses précieux conseils, la rectification ainsi que la validation de notre plan et pour sa disponibilité. Nous remercions tout autant monsieur Christophe Bedin d'avoir bien voulu accepter la charge de nous former et diriger tout le long de ce travail.
Nous remercions également monsieur Didier Pitisi de la Française des jeux.

Leur aide a été déterminante lors de la formalisation du projet où ils ont apporté de nombreuses idées clés.

Table des matières

Chapitre 1

Présentation générale du travail effectué

1. Introduction :

Le sujet de ce travail d'étude et de recherche étant la migration d'applications sous OpenVMS d'un environnement Alpha vers un environnement Itanium.

Ce sujet est proposé par Hewlett-Packard, concepteur du processeur Itanium en collaboration avec Intel, successeur de l'ancien constructeur Digital et donc détenteur actuel des droits sur le système d'exploitation OpenVMS.

Le travail réalisé a consisté en l'étude minutieuse des caractéristiques de chacun des deux processeurs Alpha et son successeur Itanium ainsi que du système OpenVMS. Cette étape a débouché sur la détermination de la possibilité de la migration d'une plateforme à l'autre et a permis l'élaboration d'une méthodologie théorique qui en donne les règles. Par la suite, un assez vaste travail pratique a permis de corroborer les résultats théoriques et de les mettre en corrélation avec ceux obtenus à partir des tests. Une série de benchmarks bien choisis et très ciblés a réaffirmé la validité de nos aboutissements notamment du côté des performances.
Ceci a pour but de guider les clients de HP qui seraient intéressé par les nouvelles machines à base du microprocesseur Itanium.

2. Environnement :

Deux machines mises en cluster mixtes à deux nœuds avec « OpenVMS (TM) Alpha Operating System, Version V8.2 » sur la première dont l'adresse sur le réseau est rondo.unice.fr et « HP OpenVMS Industry Standard 64 Operating System, Version V8.2 » sur la deuxième qui est biprocesseur et dont l'adresse est presto.unice.fr.
Deux translateurs binaires, divers compilateurs et trans-compilateurs ainsi que certaines librairies de tests de performances ont été installés sur les deux machines.

Voici les spécifications des deux processeurs :

Nom	Alpha	Itanium
modèle	Compaq AlphaServer DS10	HP rx2600
Fréquence	466 MHz	1,3 GHz
Bus principal	77 MHz, 128 bits	400 MHz, 128 bits
Temps par cycle	$2.1\ 10^{-9}$ s	$0.77\ 10^{-9}$ s
Niveau du cache	Niveau 2 (2 MB)	Niveau 3 (3 MB)

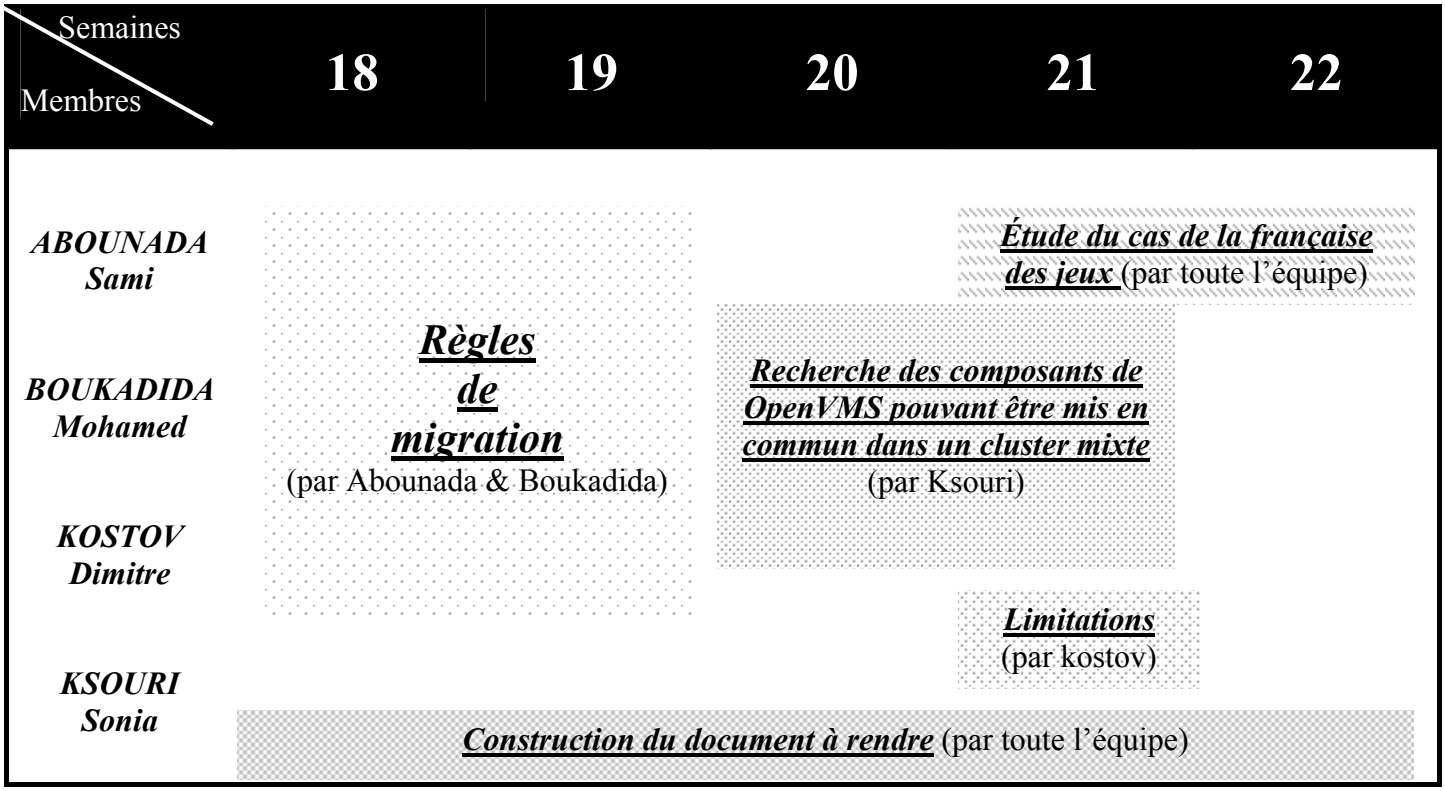
3. Déroulement :

Ce projet s'est déroulé en plusieurs étapes. Au tout début c'était une phase de compréhension et d'étude du sujet suivie d'une phase de documentation ayant eu lieu en parallèle avec l'élaboration du cahier des charges. Par la suite une micro formation de démarrage avec OpenVMS donnée par monsieur Bedin, complétée par une auto formation avec les commandes avancées et l'écriture de scripts DCL (Digital Command Language), a débouché sur une maîtrise partielle de ce système. À partir de ce point, nous avons pu installer un certain nombre d'utilitaires, bibliothèques et logiciels sur les deux machines. Nous avons commencé à écrire nos propres files de travaux batch (série de programmes séquentiels de traitement par lot), à automatiser nos tests et à mettre en place des méthodes permettant la récupération et l'archivage des résultats. Une phase d'élaboration de corollaires et conjectures théoriques notamment dans l'établissement de la méthodologie de la migration s'est effectuée en parallèle avec ce qui précédait. En dernier lieu, une comparaison des résultats théoriques avec les interprétations des tests pratiques a été faite et des conclusions s'en sont ressorties.

4. Retour sur le cahier des charges :

Le cahier des charges détaillé est disponible sur le site du suivi de notre TER à l'adresse : http://deptinfo.unice.fr/twiki/bin/view/Minfo/Cluster_OpenVMS.

Voici les principales dates de notre calendrier tel qu'il a été soumis au jury de pré soutenance :



Malheureusement quelques difficultés se sont présentées en cours de route et nous avons dû opérer des changements mineurs sur la répartition des tâches sur les personnes et sur le temps pour y palier. La réduction de l'effectif de l'équipe à trois étudiants a fait qu'on a pris un petit retard sur la partie limitations liées au fait d'être en cluster. Aussi avons nous négligé l'application de la Française des jeux faute de fourniture nécessaire de leur part (pour des raisons de confidentialité il leur a été impossible de divulguer leur code source. Nous n'avons même pas des démos de binaires !). Nous nous étions, donc, contenté d'une étude théorique qui a opposé les deux architectures, qui a bien répondu aux besoins et qui a exhibé clairement les règles de migration dans le cas général suivie d'un grand nombre de tests que nous avons écrits ou choisis nous même pour vérifier l'exactitude de nos résultats. Nous avons même chargé cette partie plus que prévue pour compenser l'absence d'application mettant ainsi l'accent sur la pratique comme nous l'a suggéré monsieur Lecarme.
Ces modifications n'ont aucunement impacté notre travail vu que tout était, heureusement, prévue dans notre cahier des charges et on s'y était bien préparé.

Chapitre 2

OpenVMS, cluster et modèles architecturaux

1. OpenVMS :

Initialement conçu en 1976 par Digital pour tourner sur les VAX (ordinateurs 32 bits). OpenVMS, autrefois VMS (Virtual Memory System), a été porté sur les architectures 64 bits ; Alpha en 1992 et plus tard sur Itanium en 2003.

Il est réputé être le système d'exploitation le plus sécurisé au monde et est utilisé sur d'énormes ordinateurs, par des clients tels que Airbus, Peugeot-Citroen, la Française des jeux, France Telecom, le CEA, les autoroutes du sud de la France, Canal Provence, les hôpitaux et bien d'autres de même taille dans le monde entier.

Il joue un rôle important dans les environnements informatiques stratégiques où les performances, la disponibilité et la sécurité sont des priorités absolues.

Les principaux traits qui le caractérisent :

- système d'exploitation 32 bits sur machines VAX, 64 bits sur Alpha et Itanium
- SMP, multi-tâches, multiprocesseur, multi utilisateur
- systèmes Alpha, VAX, Intel Itanium (à partir d'OpenVMS 8.0)
- POSIX
- contient un grand nombre de langages de programmation comme Pascal, DEC C, Fortran, DEC C++, et beaucoup d'autres
- 32 processeurs maxi. par système, jusqu'à 96 systèmes par cluster
- DCL Shell als CLI, X-11 + MOTIF GUI
- TCP/IP
- système de fichiers: ODS-2, ISO 9660 (Read), FAT (R/W), NFS et SMB
- le système des fichiers supporte Record Management Services (RMS)
- utilisation: mainframes, ordinateurs de bureau, serveurs, clusters
- Java Development Kit
- serveur Web Netscape Fasttrack

2. Cluster OpenVMS :

2.1. Définition :

Un cluster, en français grappe, est une concentration de deux ou plusieurs machines, qui accèdent à des données communes.

2.2. Généralités et architecture:

Il existe différentes architectures de cluster qui se distinguent par leurs manières d'accéder au domaine de stockage.

2.2.1. Principe de partage :

Un cluster peut adopter l'un ou l'autre des stratégies de partage suivantes :
Soit on permet à tous les nœuds d'accéder au même domaine de stockage. Des problèmes de concurrence se posent alors. Il faut gérer les accès parallèles en écriture (OpenVMS contrôle très bien ce point).
Soit on exclu l'accès commun aux mêmes données.

2.2.2. Principe du mirroring :

Avec le principe de mirroring, chaque noeud est disponible sur un domaine de stockage, reflété sur une connexion de réseau dédiée.
Le terme exact en OpenVMS est le shadowing.

2.2.3. La technologie du RAID :

Elle permet de constituer une unité de stockage à partir de plusieurs disques durs. L'unité ainsi crée a donc une grande tolérance aux pannes (haute disponibilité), ou bien une plus grande capacité/vitesse d'écriture (suivant le niveau du RAID choisi).

La répartition des données sur plusieurs disques durs permet donc d'en augmenter la sécurité, de contrôler le coût et de fiabiliser les services associés.

2.3. Grappe OpenVMS :

C'est un environnement informatique OpenVMS hautement intégré distribué sur de nombreux systèmes Alpha et IA64 ou bien (Alpha et Vax). Les systèmes faisant partie d'une grappe OpenVMS Cluster peuvent partager le traitement, le stockage de masse et d'autres ressources sous un même domaine de sécurité et de gestion OpenVMS. À l'intérieur de cet environnement hautement intégré, les systèmes conservent leur autonomie, car ils utilisent des copies locales résidantes en mémoire du système d'exploitation OpenVMS. Les systèmes OpenVMS Cluster peuvent ainsi démarrer et arrêter de façon indépendante tout en bénéficiant des mêmes ressources.

2.3.1. Caractéristiques :

- **Haute disponibilité :**
  C'est l'une des caractéristiques du cluster OpenVMS la plus importante. Les services et les applications qui fonctionnent sur un cluster sont installés de manière redondante pour un mécanisme haute disponibilité (Hight Avaibility), afin que le système reste disponible en cas de panne du matériel informatique .Le cluster offre également la possibilité de passer sans interruption d'un serveur défaillant à un autre serveur pouvant le remplacer. C'est ainsi que les clusters assurent une grande disponibilité des services serveurs, même en cas de panne.

- **Adaptabilité (scalability) et flexibilité :**
  Le cluster offre la possibilité d'ajouter ou de supprimer d'autres nœuds sans interrompre le fonctionnement . Cette souplesse nous permet de mettre en place un cluster répondant à toutes les attentes.

- **Partage des charges :**
  La répartition de la charge entre chacun des serveurs clusters se produit automatiquement. Les clusters OpenVMS comportent une fonction de répartition automatique de la charge.

2.3.2. Composants logiciels :

C'est ce qui assure les fonctions de communication et le partage des ressources. Ils s'exécutent sur tous les nœuds du cluster.
Ce sont :

- Le gestionnaire de connections : c'est le responsable du contrôle de l'adhésion des nœuds du cluster. Il assure que les nœuds du cluster sont admis directement en utilisant le mécanisme du vote et quorum et que les ressources sont réallouées au cluster quand les nœuds sont inactifs.

- Le gestionnaire de verrouillage distribué (Distributed Lock manager) : c'est le cœur d'un cluster OpenVms. Il est utilisé intensivement par le système d'exploitation.
  Ses services permettent un accès fiable et coordonné à toutes les ressources et fournissent des mécanismes de signalisation de problèmes de concurrence au niveau du système et des processus dans toute la grappe OpenVMS

- Le gestionnaire de files d'attente : OpenVMS gère les lots et les files d'attente d'impression à l'échelle de la grappe, lesquels deviennent dès lors accessibles depuis n'importe quel système. Les traitements par lots soumis dans des files d'attente à l'échelle de la grappe sont acheminés vers les systèmes disponibles afin d'assurer le partage de la charge.

- Le serveur de bande : Il utilise le serveur OpenVMS TMSCP (Tape Mass Storage Control Protocole) afin de rendre accessibles aux autres systèmes de la grappe OpenVMS Cluster les unités de bande auxquelles il a un accès direct.

- Le serveur de disque : Il utilise le serveur OpenVMS MSCP (Mass Storage Control Protocol) afin de rendre accessibles aux autres systèmes de la grappe OpenVMS Cluster les unités de disque auxquelles il a un accès direct.

3. Architectures 64 bits :

Les architectures 64 bits sont une avancée décisive pour les applications chargées de la gestion de jeux de données volumineux et complexes. Dans les anciennes architectures qui reposent sur l'adressage de mémoire 32 bits, l'UC a accès à une zone de données qui peut occuper jusqu'à 4 Go de mémoire virtuelle et est, en général, divisée en deux partitions : 2 Go sont alloués au système d'exploitation et les 2 autres Go sont affectés à l'application active. Dans les nouvelles architectures 64 bits, il est possible de spécifier les adresses de mémoire virtuelle en utilisant complètement les 64 bits. Cela augmente la plage de mémoire adressable bien au-delà des restrictions imposées par les modèles 32 bits.

Outre la capacité d'utiliser 64 bits pour définir une adresse en mémoire, les processeurs 64 bits ont en fait une autre fonction importante à savoir la manipulation de 64 bits de données simultanément. Etant donné que la capacité de manipuler 64 bits de données à la fois dépend plus de la structure des bus que des processeurs, la tendance vers le traitement 64 bits va de pair avec d'importants progrès en matière de technologie des bus système.

Par ailleurs, la réduction des interconnexions entre circuits entraîne la diminution des latences système globales. De plus, la combinaison de deux circuits engendre un libère de l'espace pour les cartes en vue de mises à niveau matérielles ultérieures.

Il ne faut pas oublier que ces architectures comptent un plus grand nombre de registre d'une plus grande taille.

Nous allons de plus près deux microprocesseurs 64 bits (l'Alpha et l'Itanium) qui sont l'implémentation de deux architectures différentes (RISC et EPIC).

4. Le modèle RISC :

L'architecture RISC (Reduced Set Instruction Computer) ou machine à jeu d'instruction réduit a été inventé pour simplifier l'enrichissement excessif du jeu d'instructions CISC. Les instructions étant plus simples, la gestion de *simultanéités* s'en trouve grandement simplifiée.

Les caractéristiques communes aux architectures RISC sont les suivants :

- Lancement de plusieurs instructions par cycle.

- des opérations registre-registre dans une architecture de type chargement-rangement.
- Allocation dynamique des ressources par le processeur.
- Analyse des dépendances lors de l'exécution.
- Renommage dynamique de registres.
- Exécution désordonnées des instructions.
- Prédiction dynamique des branchements.
- des modes d'adressage simples et peu nombreux, les modes d'adressages complexes pouvant être réalisés à partir des modes d'adressages simples ;
- des formats d'instructions simples et peu nombreux, taille d'instruction fixe, possédant des champs fixes.

Le tout résulte en des instructions et une unité de contrôle simples.

Nous allons voir la machine Alpha qui est une implémentation de cette architecture.

5. Le modèle EPIC :

À l'aube des années 90, les laboratoires HP commençaient à explorer un modèle pour une future architecture 64 bits qui augmenterait significativement le niveau de performance au dessus de ce qui pourrait être fait avec du simple RISC (Reduced Instruction Set Computer). L'idée de base était de tirer profit des leçons des modèles antérieurs et de construire une architecture à mot d'instruction plus long qui éviterait les problèmes traditionnels du modèle VLIW (Very Long Instruction Word).
Le but final était d'aller au delà du niveau de performance du modèle RISC. Plus tard, le paradigme résultant fut appelé EPIC (Explicit Parallel Instruction Computer) : Comme VLIW, ça exprime explicitement un parallélisme d'instructions mais qui le fait dans un mode qui évite le problème de comptabilité binaire associé à VLIW. Il fait d'une façon logicielle, reposant sur la robustesse des compilateurs, ce que fait le RISC d'une façon matérielle au niveau du processeur. Du coup ça donne au compilateur, voire au développeur, la possibilité d'identifier les tâches parallèles du code qui peuvent exploiter pleinement le matériel et améliorer les performances des logiciels.

HP a invité Intel à participer à l'effort de modélisation et de construction de cette architecture. Ceci a conduit à l'alliance HP/Intel et à la définition d'une architecture révolutionnaire, capable d'offrir à faible coût des performances de calcul inégalées, baptisée IA64.

Les caractéristiques communes aux architectures EPIC sont les suivants :

- Ordonnancement statique des instructions.
- Opérations codées dans des instructions composites à plusieurs champs.
- Analyse des dépendances faite par le compilateur.
- Allocation explicite des ressources faite statiquement par le compilateur.

Nous allons voir l'Itanium qui est une implémentation de cette architecture.

Chapitre 3

Alpha versus Itanium (étude théorique)

1. Le microprocesseur Alpha :

Les processeurs Alpha sont apparus vers 1992 sur le marché. Ils ont été produits pour succéder à la lignée des VAX. Leur production sera stoppée au profit des de l'Itanium. Ils ont des caractéristiques techniques très en avance sur leur temps: les performances élevées et les traitements 64 bits ont été une priorité. C'est une technologie RISC à hautes fréquences de fonctionnement.

Leurs principaux traits caractéristiques (on ne va pas beaucoup s'attarder sur les détails architecturaux ici) :

1.1. L'exécution dans le désordre du jeu d'instructions :

Dans le modèle classique d'exécution, le processeur va chercher une instruction dans le cache, l'exécute, met à jour les registres et la mémoire et passe à la suivante. Les systèmes à pipeline vont au-delà de ce mécanisme séquentiel. L'exécution dans le désordre repose sur le fait de charger plusieurs instructions à la fois, à les mettre dans une queue (dite fenêtre d'instructions) et à les exécuter suivant une optimisation matérielle (plusieurs instructions par cycle).

Cette optimisation repose sur la combinaison de quatre techniques :

1.1.1. Ordonnancement dynamique :

Le processeur peut réordonner dynamiquement (c'est-à-dire lors de l'exécution) les instructions afin de limiter sa latence.

L'exemple suivant présente un bloc linéaire de 4 instructions écrites dans un pseudo code. Considérons l'exécution de ce block sur une machine qui traite 4 instructions par cycle et qui a une latence mémoire de 3 cycles lors du chargement (LOAD) de la donnée si celle-ci est présente dans le cache (cache hit). Dans un pipeline ordonné, la séquence prendra 7 cycles, supposant que les données se trouvent déjà dans le cache. Dans un pipeline à exécution désordonnée, le processeur va chercher les quatre instructions à la foi et notera que le second LOAD est indépendant du premier ADD. Il va le déplacer et

l'exécuter avec le premier LOAD. Ainsi la séquence n'aura coûté que 4 cycles !

	Dans l'ordre	*Dans le désordre*
t1 = load a0 *t2 = add t1, 1* *t3 = load a1* *t4 = add t3, 1*	*0 : t1 = load a0* *1 :* *2 :* *3 : t2 = add t1, 1* *t3 = load a1* *4 :* *5 :* *6 : t4 = add t3, 1*	*0 : t1 = load a0* *t3 = load a1* *1 :* *2 :* *3 : t2 = add t1, 1* *t4 = t3, 1*

Évidemment si ce n'était pas un bloc linéaire, un compilateur intelligent aurait opéré lui-même cette optimisation, épargnant ainsi le processeur de faire ceci d'une manière matérielle. Cependant, le compilateur pourrait ne pas être assez efficace. (On va voir ce qu'il en est pour l'Itanium plus loin).

1.1.2. Renommage des registres :

Toujours d'une façon dynamique. C'est l'opération magique qui permet de transformer une instruction utilisant des registres architecturaux en une instruction utilisant des registres physiques. Ce renommage vise à éviter qu'une valeur encore nécessaire ne soit effacée.

L'exemple suivant présente un bloc linéaire similaire de 4 instructions. Notons que le deuxième chargement réutilise le même registre que le premier. Un compilateur n'aurait pas pu faire un ordonnancement statique, pour un processeur à exécution désordonnée d'instructions, en déplaçant le deuxième LOAD avant le ADD qui le précède parce que ceci aurait écrasé l'entrée du ADD. Dans un processeur à exécution désordonnée d'instructions, les registres architecturaux sont reliés à des registres physiques et le résultat de chaque instruction est écrit dans un registre physique distinct. Ceci évite le problème d'effacement de données et permet au processeur de déplacer le second LOAD devant le ADD !

	Dans l'ordre	*Dans le désordre*
t1 = load a0 *t2 = add t1, 1* *t1 = load a1* *t1 = add t1, 1*	*0 : t1 = load a0* *1 :* *2 :* *3 : t2 = add T1, 1* *t1 = load a1* *4 :* *5 :* *6 : t1 = add t1,1*	*0 : p1 = load a0* *p3 = load a1* *1 :* *2 :* *3 : p2 = add p1, 1* *p4 = p3, 1*

Même remarque que précédemment, si le bloc d'instructions n'était pas linéaire, le compilateur aurait intervenu statiquement. Là aussi des limites pourraient empêcher ce dernier de bien faire ses optimisations. On les abordera plus loin.

1.1.3. Prédiction de branchements :

Elle a lieu à l'exécution, au moment du chargement des instructions. Le processeur essaye de prévoir ce qui va leur être demandé. C'est ce que l'on appelle la prédiction de branchements : procédé permettant de diriger les instructions à traiter vers le bon pipeline.

L'exemple suivant présente une séquence deux blocs de base de code où on fait un LOAD suivi d'un ADD d'une valeur, puis, on branche sur L1. Supposons que le branchement est prédit correctement. Sur une machine à exécution ordonnée d'instructions, cette séquence prendra 9 cycles. Même si le branchement est prédit correctement, le processeur ne pourra pas terminer l'instruction qui suit le branchement avant que ce dernier ne soit terminé à son tour. Sur une machine à exécution désordonnée d'instructions, la prédiction exacte du branchement permettra au processeur d'examiner les 5 instructions à la foi et de réordonner dynamiquement le second LOAD et ADD avant le branchement.

	Dans l'ordre	*Dans le désordre*
t1 = load a0 *t1 = add t1, 1* *BEQ t1, L1* *L1 :* *t1 = load a 1* *t1 = add t1,1*	*0 : t1 = load a0* *1 :* *2 :* *3 : t1=add t1, 1* *4 : BEQ t1, L1* *5 : t1=load a1* *6 :* *7 :* *8 : add t3, 1*	*0 : p1 = load a0* *p3 = load a1* *1 :* *2 :* *3 : p2 = add p1, 1* *P4 = add p3,1* *4 : BEQ p2, L1*

Bien sûre une technique de compilation appropriée pourrait aider à faire ceci d'une façon logicielle à la compilation. Mais là aussi ça se discute !

1.1.4. Prédiction à travers un appel de fonction :

L'exemple qui suit présente une séquence de trois blocs de base de code, où les deux premiers blocs génèrent un appel de fonction. Supposons que le branchement est prédit correctement. Sur une machine à exécution ordonnée d'instructions, cette séquence prendra 11 cycles. Dans une machine à exécution désordonnée d'instructions, la prédiction exacte du branchement permettra au processeur de voir les instructions des deux côtés de l'appel de fonction. Il pourra, donc,

achever l'exécution du corps de la fonction avant que l'instruction BSR ne soit exécutée !

	Dans l'ordre	**Dans le désordre**
T1 = LOAD a0	***0: t1 = LOAD a0***	***0 : p1= LOAD a0***
T1 = ADD t1, 1	***1:***	***p3=LOAD a1***
BEQ t1, L1	***2:***	***1 :***
	3: t1=ADD t1, 1	***2 :***
L1 :	***4: BEQ t1, L1***	***3 : p2= ADD p1, 1***
BSR foo	***5: BSR foo***	***p4 = ADD p3,1***
	6: t1= LOAD a1	**4 : BEQ p2,L1**
***foo*:**	**7:**	**5 : *BSR foo***
t1 = LOAD a1	**8:**	**6: ret**
t1= ADD t1,1	**9: ADD t3,1**	
ret	**10: ret**	

De même, un compilateur intelligent pourrait ici incorporer (in-line) le corps de la fonction à la place de l'appel et ferait, ainsi, en sorte que les performance soit les même pour un processeur à exécution ordonnée (IA64) et un autre à exécution désordonnée (Alpha). Mais comme on av le voir plus loin, ce n'est pas si simple que ça!

1.2. Multi threading simultané :

Les systèmes informatiques ont deux formes de parallélisme : l'un au niveau des instructions, dit ILP (Instruction Level Parallelism), et l'autre au niveau des threads, dit TLP (Thread Level Parallelism). Le dernier permet à un système multiprocesseur d'exécuter plusieurs threads d'une application ou plusieurs programmes indépendants en même temps.
Le multi threading simultané, dit SMT (Simultaneous MiltiThreading), est une autre technologie qui permet à un processeur d'exploiter les deux types de parallélisme (ILP et TLP).
Plusieurs threads peuvent s'exécuter sur un processeur SMT et ce dernier va allouer dynamiquement les ressources entre les threads.
L'unique avantage de la technologie SMT est que le processeur peut utiliser le parallélisme d'instruction et le parallélisme de thread l 'un pour l'autre : Plusieurs threads peuvent tourner en parallèle dans une portion parallèle d'une application. Dans les portions séquentielles, toutes les ressources du processeur peuvent être appliquer à un seul thread.
La loi de « Amdhl » dit que les performances d'une application parallèle sont limitées par le temps passé dans les portions séquentielles ; améliorer les performances de ce genre d'application, nécessite, alors, une augmentation de la vitesse des portions parallèles et séquentielles. Un processeur SMT peut gérer cette vitesse efficacement.

Voici un exemple qui illustre l'exécution parallèles de plusieurs instructions par cycle :

```
for( i =o ; i<n ;i++)
        work on  a[i]
```

deux instructions par cycle

ALU0	ALU1
0 : load a[I+1]	.
1 : work a[i]	work a[i]
2 : work a[i]	work a[i]
3 : load a[i+2]	.
4 : work a[i+1]	work a[i+1]
5 : work a[i+2]	work a[i+1]

quatre instructions par cycle

ALU0	ALU1	ALU2	ALU4
0 : load a[I+2]	.	load a[I+3]	.
1 : work a[i]	work a[i]	work a[i+1]	work a[i+1]
2 : work a[i]	work a[i]	work a[i+1]	work a[i+1]
3 : load a[i+4]	.	load a[i+5]	.
4 : work a[i+2]	work a[i+2]	work a[i+1]	work a[i+3]
5 : work a[i+2]	work a[i+2]	work a[i+1]	work a[i+3]

Supposons qu'il faut 4 instructions « work » pour chaque élément du tableau « a », que tout le tableau est dans le cache et que le LOAD prend 3 cycles pour retourner la valeur (bien sûre quand celle ci est dans le cache, comme supposé).
Sur une machine qui traite deux instructions par cycle, on peut charger l'élément suivant du tableau alors qu'on travaillait sur l'élément en cours. Ceci permet d'utiliser les unités fonctionnelles d'une façon efficace.
Sur une machine traitant quatre instructions par cycle, on peut charger les deux prochains éléments alors qu'on travaillait déjà sur les deux qui les précédaient et continuer à utiliser les unités fonctionnelles d'une façon beaucoup plus efficace.

Voici un autre exemple illustrant l'appel au parallélisme de thread lorsque le parallélisme d'instruction se voit limité :

```
While ( a !=0)
        Work on  node  a
        a = a->next
```

deux instructions par cycle

ALU0	ALU1
0 : an= load a->next	.
1 : work a	work a
2 : work a	work a
3 : a=load an->next	.
4 : work an	work an
5 : work an	work an

quatre instructions par cycle

ALU0	ALU1	ALU2	ALU4
0: an = a->next	.	.	.
1: work a	work a	work a	work a
2: .	.	.	.
3: a= load an->next	.	.	.
4: work an	work an	work an	work an
5: .	.	.	.

On a changé la structure de données utilisant ainsi une liste chaînée à la place du tableau. Sur une machine exécutant deux instructions par cycle, on peut charger le prochain élément de liste alors qu'on travaillait encore sur l'élément courant utilisant, de la sorte, les unités fonctionnelles d'une façon efficace. Par contre, sur une machine à 4 instructions par cycles, il n'est pas possible de trouver davantage du parallélisme. On peut simplement traiter un élément de la liste tous les trois cycles même si on parallélise le travail sur un élément ! On ne peut pas charger le prochain élément de la liste à moins qu'on ait déjà charger le précédent. Cette limitation est inévitable à moins de réécrire le code.
Ici on va voir l'importance du parallélisme de thread : En effet on peut très bien utiliser le multi threading simultanée en exécutant un autre programme ou une autre thread du même programme sur d'autres unités fonctionnelles déjà initialisées. Mais comment est ce possible ?

En fait le mécanisme de multi threading simultanées transforme le parallélisme au niveau des threads en un parallélisme au niveau des instructions.
Un processeur Alpha à exécution désordonnée d'instructions à un mécanisme de chargement d'instructions (FETCH) ordonné et un mécanisme d'exécution désordonné. Les instructions sont prises dans l'ordre de leur apparition dans le programme, puis, exécutées dans un ordre déterminé par les dépendances existantes entre elles. Entre le FETCH et la portion de la machine où sera exécutée la donnée, il y a une queue. Les instructions y attendent d'être traitées.

C'est à peu près ceci :

[FETCH ordonné] [queue] [exécution désordonnées] [sortie ordonné]

SMT est implémenté de façon à permettre un FETCH d'un thread différent à chaque cycle. Les instructions cherchées (FETCH) voient leurs registres renommés (voir 3.1.2) et sont placées dans une queue. Grâce au renommage de registres, des instructions issues de différents threads, sont mélangées dans la queue et peuvent être exécutées dans un même cycle.

Voyons maintenant qu'en est-il d'une autre architecture qui elle aussi en est une à 64 bits mais qui est tout aux antipodes de la première

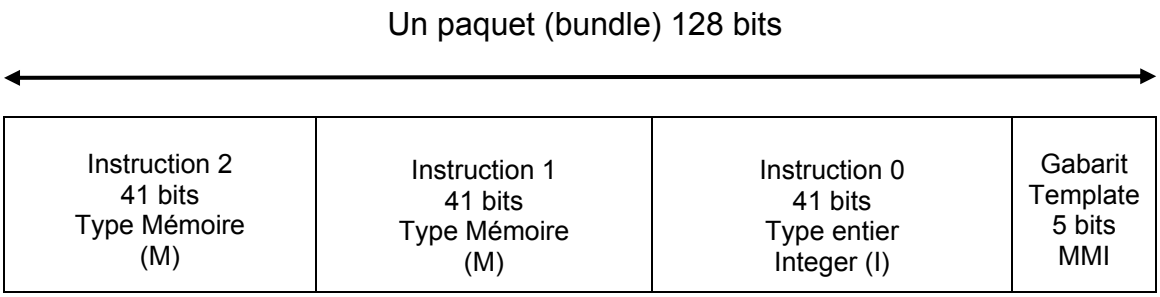
2. Le microprocesseur Itanium :

Dans cette partie, nous présentons les fondements de l'architecture des processeurs Itanium. Comme ceci a été dit dans le paragraphe précédent, l'Itanium tire une bonne partie de ses performances du fait qu'il expose au compilateur le parallélisme employé par le processeur.
Dans l'Itanium, une partie importante du travail, habituellement faite par le processeur pendant l'exécution du programme, se trouve transférée au compilateur. Ce dernier a une vue plus large sur le code à exécuter et dispose d'une quantité de ressource beaucoup plus grande que le processeur. Par conséquent, cette approche offre plusieurs avantages, notamment une simplification de l'architecture et une augmentation du degré de parallélisme. Elle requiert néanmoins de la part du compilateur et du développeur des connaissances élémentaires sur la façon dont le code est exécuté par le processeur. Une connaissance du jeu d'instructions et des ressources matérielles de l'architecture Itanium est donc utile pour exploiter pleinement ces avantages.

Voyons de plus près les aspects architecturaux de ce processeur (Nous allons bien détailler cette partie car la compréhension de la suite de notre étude en dépend) :

2.1. Le format des instructions :

Le compilateur et l'assembleur Itaium regroupent les instructions par paquets (bundles) de 3 instructions. Ces derniers sont exécutés entièrement et Itanium 2 (la machine sur laquelle nous travaillons en est un) est capable d'exécuter simultanément deux groupes. Chaque groupe est codé sur 16 octets ou 128 bits.
La figure suivante donne le format d'un groupe d'instructions :



Cette figure contient trois instructions sur 41 bits chacune. Le champ *Template* (gabarit) facilite la phase de décodage. Dans cet exemple, le groupe est composé de deux instructions mémoire (M) et d'une instruction entière (I). Pour cett raison, le champ *Template* est positionné à MMI. De cette façon, la première instruction est envoyée directement vers l'unité de calcul sur les entiers alors que les deux suivantes sont envoyées directement vers les deux unités mémoire. Avec 5 bits pour ce champ, il est possible de coder 128 combinaisons différentes. Seules 24 combinaisons sont utilisées. En plus du type d'instruction, M pour mémoire, I pour entier, F pour flottant, B pour branchement et X pour multimédia, le champ *Template*, indique les dépendances entres les instructions à l'intérieur d'un même paquet ou dans deux paquets voisins. Ceci se fait à l'aide de la marque stop. Lorsque ce bit est présent, il indique que les instructions après la marque stop dépendent des instructions avant la marque stop (dépendance des données). À titre d'exemple, 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 ce dernier ne contient pas de bit stop au milieu, les six instructions des deux paquets peuvent s'exécuter simultanément. L'utilisation du bit stop augmente le parallélisme et simplifie la micro architecture. Un ensemble d'instructions consécutives sans marque de stop entre elles constitue un groupe. Un groupe d'instructions est localisé entre deux marque stop et peut utiliser plusieurs paquets.

2.2. Lecture des instructions :

On passe d'abord par un mécanisme de préchargement, contrôlé par le compilateur ou par le logiciel. Il a pour but d'éviter des défauts dans le cache. Ceci se réalise, au moment de la compilation, par l'insertion d'indications permettant de charger les prochaines instructions à exécuter de la mémoire externe vers l'un des trois niveaux de mémoire cache (l'Itanium a trois niveaux de cache).
Deux paquets d'instructions (6 instructions) sont lus par cycle et sont envoyés vers le tampon des instructions qui est directement connecté aux ports des unités fonctionnelles.

2.3. Exécution des instructions :

L'exécution d'instructions se fait par la sélection d'un groupe de plusieurs instructions. Un paquet en est construit (nous en verrons dans le paragraphe suivant la méthode de construction). La structure de paquet permet de

véhiculer les instructions parallèles. Pour qu'un groupe s'exécute, il est nécessaire que les instructions qui le composent soient prêtes à l'exécution.
Les unités fonctionnelles de la partie de la micro-architecture chargée de l'exécution sont groupées comme suit :

- Unités entières : Six au total. Un maximum de six instructions entières peut être exécuter à chaque cycle.
- Unités mémoires : 4 ports sont connectés à l'unité mémoire cache, dont deux doivent être des instructions de chargement et deux doivent correspondre à des instructions de stockage (STORE) en mémoire.
- Unités multimédias : Il en existe dix. Au maximum deux instructions multimédias peuvent être exécutées par cycle d'horloge. Les données multimédias sont codés sur 64 bits et peuvent être considérés comme deux données sur 32 bits, 4données sur 16 bits ou 8 données sur 8 bits. Pour profiter de cette représentation, plusieurs instructions du type « une instructions sur plusieurs données » sont disponibles. De cette façon, il est possible de réaliser, en une seule instruction, par exemple 8 additions sur 8 jeux de données sur 8 bits chacun. On peut réaliser de la façon des opérations logiques, des décalages, etc.
- Unités fonctionnelles pour les données à virgules flottantes : Il existe deux unités de multiplication et addition et deux unités pour les autres opérations. Au maximum deux instructions flottantes peuvent être lancées par cycle.

L'Itanium contient 4 types de registres :

- Les registres généraux : au nombre de 128 registres de 64 bits chacun. Ils servent à stocker les valeurs entières utilisées comme source et/ou destination des instructions entières. Nous trouvons un bit noté NaT (Not a Thing) associé à chacun des registres. Ce bit est utilisé lors de la spéculation sur les instructions de chargement pour des données de type entier. Il sert à signaler une exception suite à l'exécution d'une instruction de chargement spéculative. Nous le détaillons dans le paragraphe dédié aux différentes techniques de spéculation utilisées dans l'Itanium.
- Les registres flottants : au nombre de 128 registres de 82 bits chacun. Les registres de 0 à 31 sont statiques et peuvent être utilisés à n'importe quel moment. Les registres flottants n'ont pas de bits NaT associés. Si une exception se produit dans un chargement vers un registre flottant, une valeur réservée spécialement dédiée à cet effet est chargée dans le registre destination. Cette valeur est notée NaTVal. Le reste des registres (de 32 à 127) est utilisé pour réaliser le pipeline logiciel. Ce mécanisme sera détaillé par la suite.
- Les registres de prédicats : l'Itanium contient aussi 64 registres de prédicats. Ces registres de 1 bit servent à éliminer les instructions de branchement. Ils sont positionnés à 1 ou à 0 (vrai ou faux) suivant le résultat d'une comparaison. Comme pour les registres généraux et les registres flottants, une partie (de p0 à p15) est statique alors que le reste est utilisé pour faire le pipeline logiciel. En général l'utilisation des registres de prédicats permet de transformer des sections contenues dans des branchements en une suite d'instructions avec prédicats. La

section dédiée aux techniques de spéculation nous donnera plus de détails.

- Les registres de branchement : L'Itanium 2 (la machine sur laquelle nous travaillons) contient 8 registres de 64 bits destinés à contenir les adresses des cibles pour les instructions de branchement indirect.
- Les registres d'application : l'Itanium contient enfin 128 registre de 64 bits chacun nommés registres d'application. Chacun de ces registres a une fonction particulière et peut être accédé par l'application. A titre d'exemple les registres ar65 et ar66 sont utilisés pour implémenter le pipeline logiciel.

2.4. Le jeu d'instruction de l'Itanium :

Une instruction Itanium a la forme suivante :

[rp] mnémonique[.comp] dest = srcs

Où :

- rp : registre de prédicats (p0 à p63) ;
- mnémonique : nom de l'instruction ;
- comp : symbole permettant de compléter le nom de l'instruction ;
- dest : opérande destination ;
- srcs : un ou plusieurs opérandes sources :

Par défaut, le registre de prédicats utilisé est le registre p0 qui est toujours à un, et le résultat est donc écrit dans le registre destination.
Mnémonique et comp permettent de spécifier l'opération désirée. Certaines instructions n'utilisent pas le champ comp.
Les champs dest et srcs font référence soit à des registres soit à des opérandes immédiats.
Voici deux exemples d'instructions commentées :

```
(p1)  add r10 = r12, r13
        //addition de r12 et r13dans r0 si p1=1
(p0)  ld8.s r31 = [r3]
        //chargement dans r31 de la donnée (8 octets)
        // pointée par r3 sans condition sur les prédicats
        //car p0 est toujours à 1
```

Le jeu d'instruction de l'Itanium est divisé en 8 groupes. Nous allons passé en revue les instructions les plus courantes :

2.4.1. Les instructions arithmétiques entières :

Le tableau suivant donne les utilisations les plus courantes des instructions arithmétiques entières :

Instructions	Tâche réalisée
(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*

Où extSigne signifie extension de signe de la valeur immédiate exprimée sur 8 bits (entre -128 et +127) vers une valeur sur 64 bites signée.

2.4.2. Les instructions logiques :

Les instructions logiques réalisant un traitement bit à bit des données. À titre d'exemple, le compilateur transforme une instruction en langage C/C++ utilisant l'opérateur bit à bit & en instruction « *and* ». Le tableau suivant donne les exemples d'instructions logiques les plus fréquentes :

Instructions	Tâche réalisée
(rp) and r1 = r2, r3	*Si rp=1, r1=r2 ET r3*
(rp) and r1 = imm8, r3	*Si rp=1, r1=imm8 ET r3*
(rp) andcm r1 = r2, r3	*Si rp=1, r1=r2 ET (NON r3)*

2.4.3. Les instructions de décalage :

Les instructions de décalage de données sont utilisées pour réaliser de opérations de multiplication et de division par des puissances entières de 2. Le tableau suivant énumére les instructions de décalage les plus fréqunetes :

Instructions	Tâche réalisée
(rp) shl r1 = r2, r3	Décaler r2 de r3 positions à gauche, résultat dans r1. Compléter par des zéros (décalage logique).
(rp) shl r1 = r2, compteur	Décaler r2 de «compteur» positions à gauche, résultat dans r1. Compléter par des zéros (décalage logique).
(rp) shr r1 = r2, r3	Décaler r2 de r3 positions à droite, résultat dans r1. Compléter par des bits de signe (décalage arithmétique).
(rp) shr.u r1 = r2, r3	Décaler r2 de r3 positions à droite, résultat dans r1. Compléter par des zéros (décalage logique).

2.4.4. Les instructions de comparaison :

La structure générale d'une instruction de comparaison est la suivante :

(rp) cmp.relComp.typeCom p1, p2=r2 (ou imm8), r3

Les parties *relcomp* et *typeComp* complètent le code opération *cmp*. Le premier champ *relComp* (relation de comparaison) est obligatoire et indique la relation sur laquelle porte l'opération de comparaison. Le deuxième champ *typeComp* (type de comparaison n'et pas obligatoire). Lorsqu'il est présent il indique de quelle manière les registres de prédicats sont positionnés.

2.4.5. Les instructions de branchement :

Dans l'Itanium, une attention particulière a été réservée aux instructions de branchement. Il existe un grand nombre de variantes d'instruction de branchement qui ont toutes comme mnémonique *br*. Le tableau suivant donne les instructions *br* les plus courantes et les plus importantes :

Instructions	Tâche réalisée
*(rp) br.cond.**bwh.ph.dh** étiquette25*	Branchement conditionnel relatif à la position courante (IP-relatif)
*(rp) br.cond.**bwh.ph.dh** b2*	Branchement conditionnel indirect par l'utilisation de registre de branchement b2
*(rp) br.call.**bwh.ph.dh** b1=étiquette25*	Branchement (ou appel) vers une fonction repérée par l'étiquette étiquette25. b1 contient l'adresse retour
*(rp) br.cloop.**bwh.ph.dh** étiquette25*	Tant que le registre d'application ar.lc est non nul, aller à étiquette25

bwh représente l'indication donnée par le compilateur pour la prédiction de branchement
ph (Prefech Hint) représente l'indication donnée par le compilateur pour le préchargement des instructions dans le cache.
Dh (branch cache Deallocation Hint) représente l'indication donée par le compilateur concernant la (probabilité de) ré-exécution ou non de l'instruction de branchement.
étiquette25 indique une étiquette sur 25 bits.
La dernière instruction de tableau *br.cloop* est utilisée pour exécuter des boucles du type *for* en C ou *DO* en FORTRAN.

Il existe une autre version de *br* largement utilisée pour réaliser le pipeline logiciel.

2.4.6. Les instructions d'accès mémoire :

L'Itanium reprends le principe de instructions RISC pour les instructions mémoires. Ce principe consiste à ne réaliser les accès mémoire qu'à travers les deux instructions de chargement (LOAD) et de stockage (STORE). Il existe des instructions dédiée aux accès mémoire pour charger u stocker les registres flottants.

2.5. Les techniques avancées pour la spéculation dans l'Itanium :

2.5.1. La spéculation de contrôle :

Dans la section 2.2 (dédiée au fichier de registres), nous avons déjà mentionné l'utilisation des bits NaT et NaTValue pour l'exécution des instructions de chargement spéculatif. Nous allons détaillé maintenant ce mécanisme :
Comme le montre l'exemple qui suit, l'Itanium utilise la technique des chargements spéculatifs avancés pour cacher la latence des instructions mémoires. Ainsi, au lieu de réaliser l'instruction de chargement juste avant l'instruction utilisant la donnée chargée, le chargement est déplacé suffisamment à l'avance pour éviter de faire attendre les instructions suivantes, et augmenter de ce fait les performances. Néanmoins, si le principe paraît simple, les instructions de branchement peuvent limiter cette utilisation. En effet, sans utiliser de spéculation, il est impossible de faire remonter une instruction de chargement plus haut que la première instruction de branchement conditionnelle. Cela peut altérer le contenu du registre destination et peut provoquer la génération d'exceptions suite à l'accès mémoire (adresse illégale ou défaut de page par exemple).
Pour cette raison, l'Itanium utilise une technique originale permettant de séparer d'une part le lancement de l'instruction de chargement et d'autre part la détection et la prise en charge de l'exception (si celle-ci se produit). Comme le montre l'exemple suivant, le mécanisme utilise deux instructions *ld.s* et *chk.s*. Dans notre exemple, l'instruction *ld8.s r1=adress* lance l'opération de chargement de la donnée depuis la mémoire. Mais elle ne réalise ni la modification du registre destination (*r1* ici), ni la génération d'exception s'il arrive que celle-ci se serait produite. Si cette instruction génère une exception, le bit NaT du registre destination *r1* est positionné à 1. Ceci permet de spécifier qu'une exception a été créée lors de l'exécution du chargement spéculatif, mais qu'elle ne sera réellement signalée que lors de l'instruction.

chk.s r1, routine_Exep

Cette instruction a pour rôle de vérifier qu'une donnée valide a été placée dans le registre destination *r1*. Autrement dit, elle sert a vérifié que le chargement est terminé et qu'il n'y a pas eu d'exception (NaT=0). Dans le cas contraire, un branchement vers la routine de gestion de l'exception est effectué. Notons enfin que cette technique est réalisée uniquement lorsque le compilateur juge que le branchement a une faible chance d'être pris. Dans le cas contraire le chargement n'est pas spéculé et il garde sa position après l'instruction de branchement.

Version sans chargement spéculatif	Version avec chargement spéculatif
add… *sub…* ….. *br.eq…* //barrière pour remonter ld *ld8 r1=[r2]* *add r3, r1, r10* …	***ld8.s r1=[r2]*** **//chargement spéculatif** add… sub… … br.eq… //Il n'y a plus de barrière pour ld ***chk.s r1, routine_Exception*** *add r3, r1, r10* …

2.5.2. Les branchements par le mécanisme de prédicats :

Prenons l'exemple suivant

if (r1) { r2=r3+r4 ;}
else { r7=r6-r5 ;}

Habituellement, cette séquence est traduite en langage machine en suivant le schéma suivant :
(1) Tester égalité de r1 à 0
(2) Si résultat faux aller à (5)
(3) r2 = r3 + r4
(4) aller à (6)
(5) r7 = r6 - r5
(6) fin : …

Le problème posé avec ce schéma concerne l'instruction (1). Elle doit être terminée avant de pouvoir commencer le reste des instructions afin de choisir l'une des deux voies (soit celle qui commence en 2 soit celle qui commence en 5). De ce fait, si le calcul de r1 dépend d'une autre valeur nécessitant beaucoup de temps, l'instruction (1) peut être retardée d'un grand nombre de cycles. Cette situation peut ralentir la vitesse d'exécution du programme. La solution la plus souvent adaptée consiste à faire une prédiction du résultat du test. Cette prédiction consiste en général à mémoriser le résultat du dernier test et à l'utiliser de nouveau comme nouveau résultat. Itanium 2 fait mieux que ça et il propose deux nouveaux mécanismes de prédictions de branchement. Le premier consiste à contrôler le mécanisme de prédiction par des indications fournies par le compilateur. Ces indications sont codées dans l'instruction de branchement (c'est une prédiction statique). De ce fait dès que l'instruction est lue, le processeur connaît le conseil donné par le compilateur pour réaliser la prédiction. Ce mécanisme est utilisé pour les instructions de branchement dont le résultat peut être prédit avec une grande confiance de façon statique, c'est-à-dire au moment de la compilation.
Voici un exemple :

(p1) br.cond.spntdebut

Dans cet exemple, nous avons un branchement conditionné par *p1*. Il est prédit statiquement non pris (spnt : static predicted not taken) vers l'instruction étiquetée par *debut*. Le processeur ne va pas alors utiliser les ressources matérielles pour traiter le branchement puisque le compilateur a jugé qu'il a suffisamment d'informations pour garantir avec une très forte probabilité que le branchement ne sera pas pris.
De la même manière, il existe trois autres types d'indications *sptk* (static predicted taken), *dpnt* (dynamic predicted not taken) et *dptk* (dynamic predicted taken). Pour les deux dernières indications *dptn* et *dptk,* le processeur doit utiliser le mécanisme matériel pour réaliser la prédiction de branchement.
Les branchements dont le résultat est difficile à prédire par les deux mécanismes précédents peuvent utiliser le troisième mécanisme. Cette solution repose sur l'utilisation de prédicats et permet d'éliminer les instructions de saut.

2.5.3. Dépendances des données et spéculation :

Les dépendances de données sont une autre cause pouvant limiter les performances des processeurs, tout particulièrement lorsqu'il s'agit d'instructions d'accès mémoires en lecture. Comme ceci a été expliqué dans le paragraphe précédent, il intéressant de lancer l'instruction de chargement au plutôt afin de réduire la latence de ces accès en lecture.
Un problème peut se poser lorsqu'une instruction de stockage (STOTRE) à la même adresse est exécutée entre le chargement avancé et l'instruction qui utilise la donnée chargée. Comme bien souvent le mode d'adressage utilisé par les instructions mémoire est l'adressage indirect, il devient impossible de détecter cette situation de dépendance entre les instructions mémoires *ld* et *st*. Le code suivant explique le problème à travers un exemple :

```
ld8 r7 = [r14]          //cycle -4, initialiser le registre r7 …
...                      //d'autres instructions
st4 [r1] = r3            //cycle 0 ; barrière de store
ld4 r6 = [r7] ;;         //cycle 0 ; load à monter
add r10 = r6, r1 ;;      //cycle 2 ; utilisation de la donnée chargée
```

Dans cet exemple, il est intéressant de remonter l'instruction *ld4* au plus tôt afin d'assurer la présence de la donnée dans *r6* avant le début de l'exécution de l'instruction : *add r10 = r6, r1*
La présence du *st4 [r1] = r3* empêche cela car les contenus de *r1* et *r7* peuvent être identiques. Autrement dit *r1* et *r7* sont deux alias de la même adresse. En effet, dans ce cas la donné chargée par *ld4* sera modifié par le *st4.* De ce fait, l'instruction *add* ne va pas additionner la valeur après le *st* mais avant cette instruction. Puisque le contenu de *r7* vient de la mémoire par l'instruction *ld8 r7 = [r14]*, nous ne pouvons pas contrôler l'égalité à l'avance.

Pour contourner ce problème d'alias et éviter la perte de cycles , l'Itanium propose d'utiliser une nouvelle version de l'instruction *ld.* Il s'agit de la version avancée *ld.a.* En utilisant cette instruction, le chargement spéculatif de la donnée dans le registre est réalisé bien avant le point d'utilisation *add r10 = r6, r1.*

```
ld8 r7 = [r14] ;;          //cycle -4 et initialiser le registre r7 …
...                        //d'autres instructions
ld4.a r6 = [r7] ;;         //cycle -2 ; chargement spéculatif avancé
...                        //d'autres instructions
st4 [r1] = r3              //cycle 0 ; barrière de store levée
ld4.c r6 = [r7] ;;         //cycle 0 ; vérification du load avancé
add r10 = r6, r1           //cycle 2 ; utilisation de la donnée chargée
```

Comme on peut le voir dans cette nouvelle version, l'instruction de chargement a été avancée à un tel point que la première instruction qui l'utilise s'exécute sans retard. Pour vérifier qu'aucune instruction *st* n'a accédé à la même adresse mémoire, on trouve à sa place une instruction : *ld4.c r6 = [r7].* Son rôle est de vérifié qu'il n'y apas eu d'alias entre *ld* et *st.* Si c'est le cas il faudra relancer l'instruction *ld.* C'est le compilateur qui est derrière tout ceci. Notons que pour les chargements avancés, le compilateur peut non seulement faire avancer l'instruction de chargement mais aussi les instructions qui utilisent la donnée chargée. Ainsi dans notre exemple même l'instruction *add* peut être avancée avant le *st* si le compilateur le juge intéressant, par exemple pour remplir des paquets.

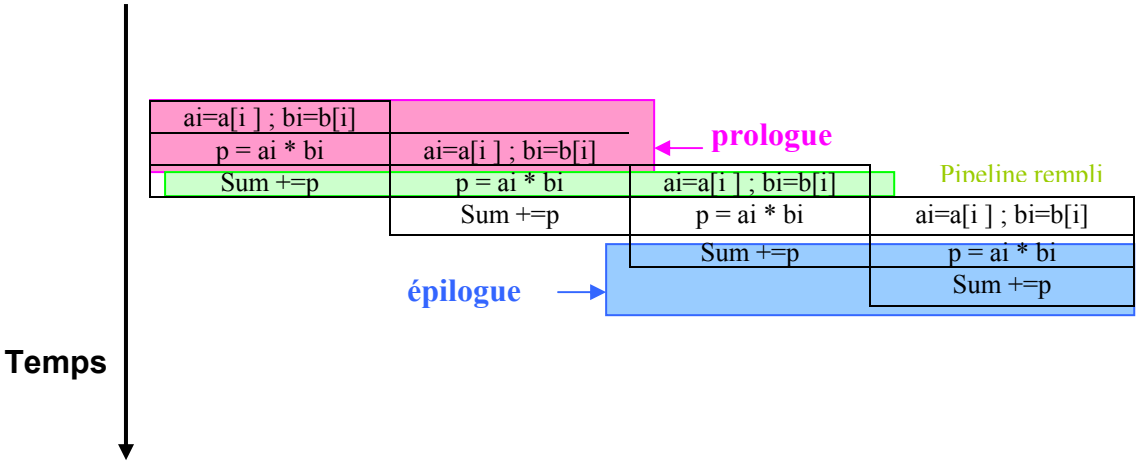
2.5.4. Le pipeline logiciel :

Le pipeline logiciel est l'une des techniques les plus utilisées pour augmenter le degré du parallélisme dans les boucles. L'idée générale est de réaliser un recouvrement entre les différentes itérations d'une boucle. Pour ^présenter le principe prenons l'exemple suivant (en langage C) :

```
for (int i=0 ; i < N ; i++)
    sum = sum + a[i] * b[i] ;
```

Dans chaque itération de la boucle de cet exemple, il y a trois étapes : un élément du tableau *a* et un élément du tableau *b* sont lus, ensuite traités (multiplication et addition) et enfin le résultat *sum* est écrit en mémoire. Il est clair qu'il existe une dépendance entre ces opérations. En conséquent si la boucle est exécutée un grand nombre de fois, les ressources matérielles seront mal utilisées. Le pipeline logiciel consiste à transformer le code pour manipuler des données indépendantes. Après les deux premières itérations (prologue) dans le nouveau code (voir schéma ci-dessous), chaque itération consiste à additionner le résultat de l'itération $i-2$, à multiplier les éléments de l'itération $i-1$ et à lire les éléments $a[i]$ et $b[i]$ de l'itération i. La fin de la boucle se

termine par deux boucles (épilogue) réalisant le traitement des deux derniers éléments qui ont été déjà chargés mais non encore traités.



Voici notre exemple après transformation :

```
//prologue
ai=a[0] ; bi=b[0] ;
p=ai*bi ; ai=a[1]
p=ai*bi ; ai=a[1] ; bi=b[1]

//pipeline logiciel
for (i=2 ; i < N ; i++ {
sum += p ;
p = ai * bi ;
ai = a[i] ; bi=b[i] ; }

épilogue
sum += p ; p=ai*bi
sum +=p ;
```

Il est clair que l'un des inconvénients majeurs de cette technique est l'augmentation de la taille du code. Par conséquent, le compilateur ne devra réaliser cette transformation que le lorsque le gain de performances apporté par cette technique est important. Ceci revient à ne considérer que les boucles avec un grand nombre d'itérations. Dans le cas de l'Itanium cette contrainte est levée car le processeur est doté d'outils (logiciels et matériel) permettant une utilisation efficace du pipeline logiciel, sans avoir à ajouter du code pour les parties prologue et épilogue de la boucle.

Chapitre 4

De Alpha vers Itanium (étude expérimentale)

Tests et Benchmarks :

Pour comparer les performances des deux processeurs ALPHA et ITANIUM dont les caractéristiques concessionnaire ont été données au tout début de ce document, nous avons choisi de faire tourner des benchmarks standardisés sur les deux machines.

- Les sources de ces benchmarks sont écrites en C. Pour les compiler sur l'une et l'autre des deux machines, on a utilisé les versions suivantes du compilateur (***CC***) :

Itanium	Alpha
HP C V7.1-011	Compaq C V6.5-001

- Un traducteur binaires(***AEST V1.0***) a été installé sur les 2 machines.Ce traducteur transforme les images binaires de l'Alpha vers des binaires équivalents aux formats IA64.

Les binaires qu'on a exécuté et dont on a récupéré les résultats d'exécution ont été obtenus de quatre manières :

- B1 : Binaires obtenus en compilant les codes sources sur la machine Alpha.
- B2 : Binaires obtenus en compilant les codes sources sur la machine Itanium.
- B3 : Binaires traduits en appliquant « AEST » depuis la machine Alpha sur B1.
- B4 : Binaires traduits en appliquant « AEST » depuis la machine Itanium sur B1.

Ainsi, un B1 est destiné a la machines Alpha, tandis que les binaires {B2,B3,B4} sont destinés à l'IA64.

- Les options d'optimisation du compilateur(loop unrolling ,function inlining ..) ont aussi été utilisées sur quelques benchmarks pour permettre d'atteindre les performances crêtes. Ceci vise aussi à solliciter les compilateurs en vue de voir leur implication dans l'optimisation (statique) du code.

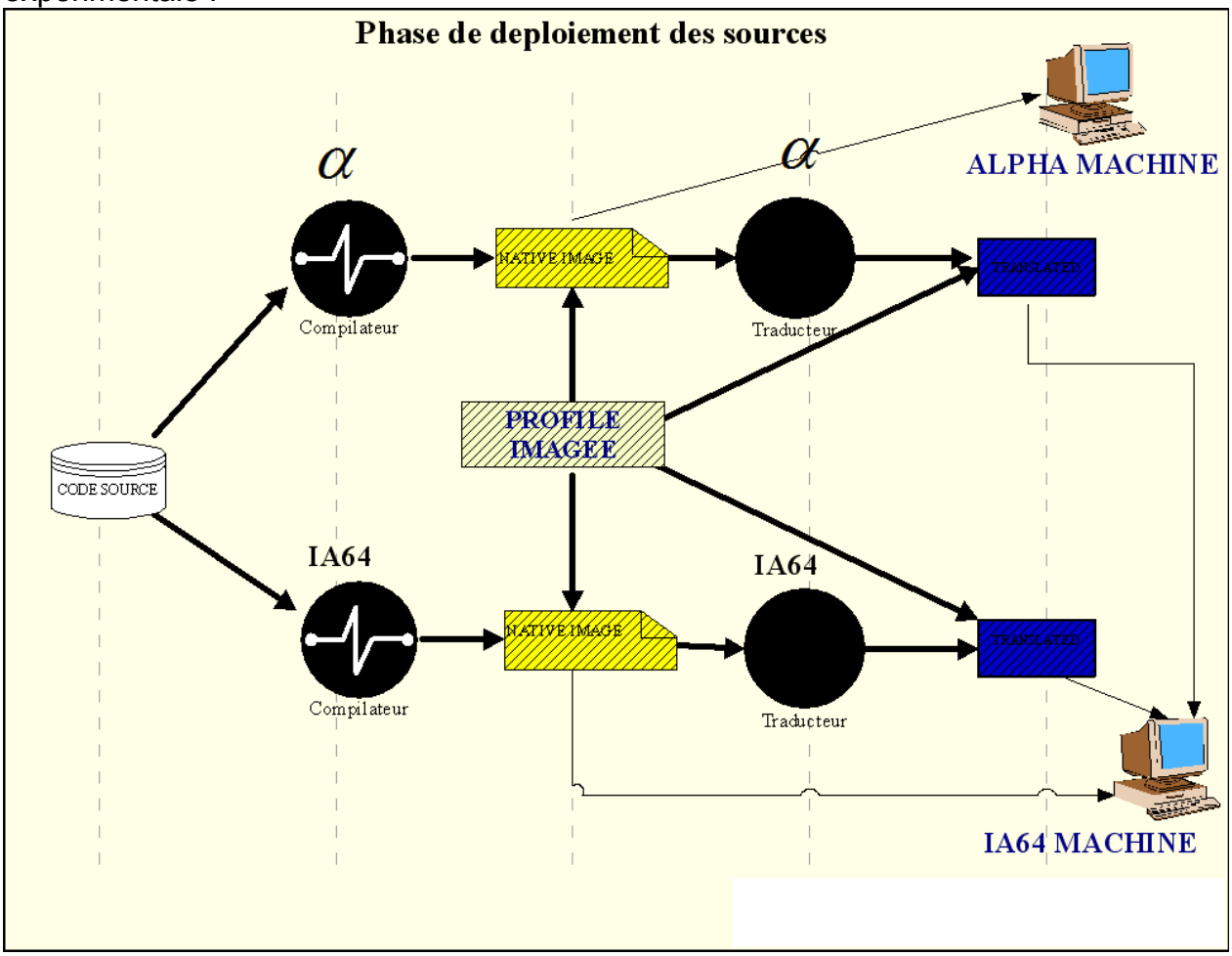
Pour permettre d'effectuer une mesure du nombre de cycles CPU nécessaires à l 'exécution des benchmarks, le nombre total d'instructions exécutées ainsi que le nombre de cycles CPU passés en décrochage, des bibliothèques de debogage spécialisées, fournies par le kit DECSet, ont été liées à nos binaires. L'avantage de ces bibliothèques est que non seulement elles permettent de mesurer le performances mais aussi la sauvegarde de certains événements pendant l 'exécution des processus (défauts de cache, temps d 'exécution, appels systèmes,..) dans des fichiers de trace nous permettant, ainsi

- Deux métriques principales ont été utilisées pour la mesure de performances :

  – le nombre maximal d'instructions pouvant être exécutées par un processeur, exprimé en MIPS (millions ou milliards d'instructions par seconde).

  – pour les calculs flottants intensifs, les processeurs peuvent exécuter jusqu'à des millions ou des milliards d'opérations flottantes par seconde. Ces capacités sont exprimées en termes de MFLOPS ou GFLOPS.

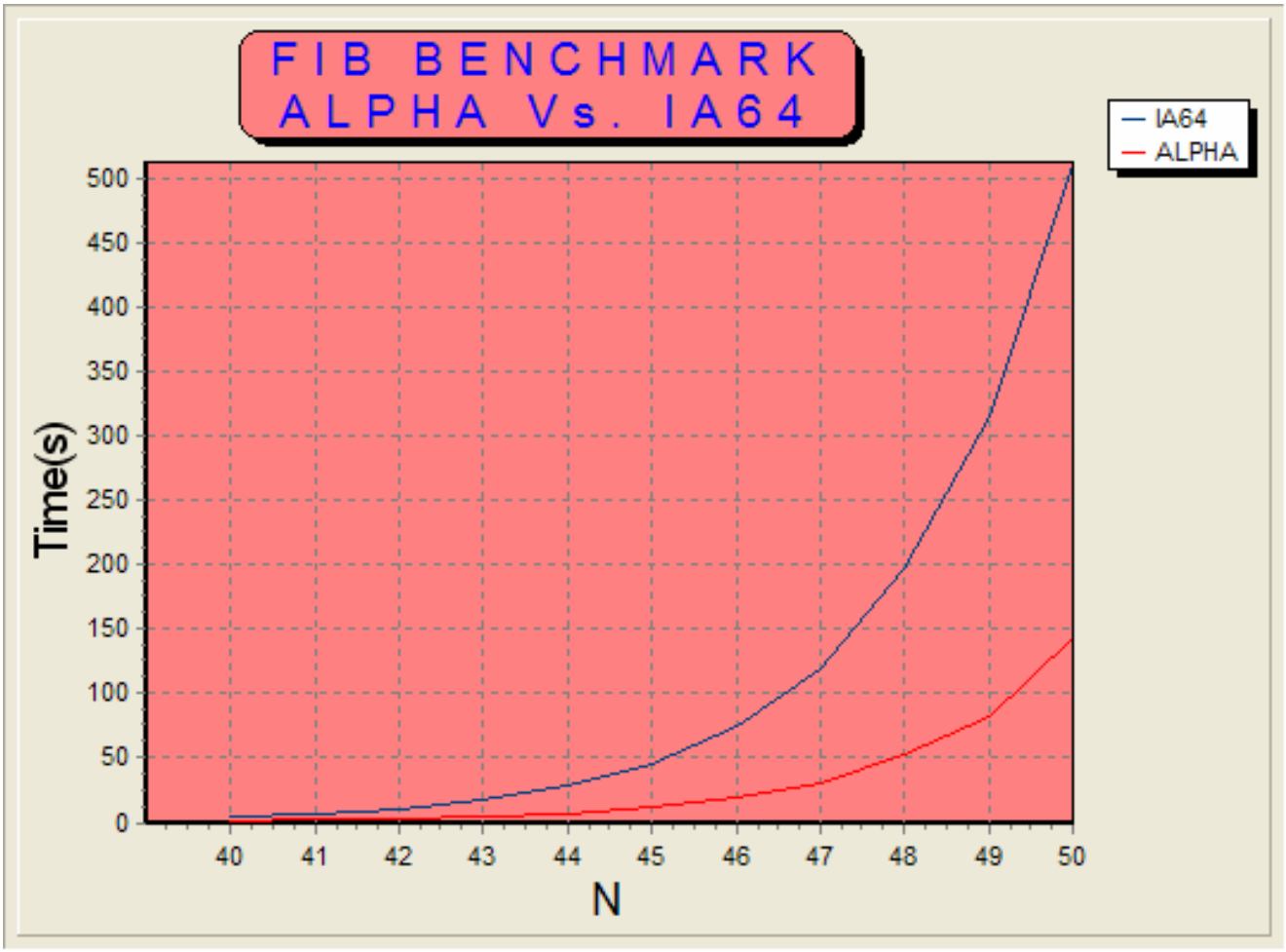
Le schéma suivant, montre comment nous nous étions pris pour faire notre étude expérimentale :



Les benchmarks qui ont été testé sur nos plates-formes et leurs résultats sont les suivants :

FIB BENCHMARKS :

Ce bench calcule la suite récursive de fibonnacci.
Son temps d'exécution T(n) , dépends du temps d'exécution d'un appel récursif (k) ; selon la relation :
T(n) = k * n^2
Le temps de calcul sur l' IA64 est inférieur a celui de Alpha, comme le montre la figure suivante suivant :



On remarque des performances écrasantes de l'Itanium face à l'Alpha. Ceci montre l'efficacité du mécanisme de spéculation de l'Itanium (voir plus haut) et témoigne du niveau de robustesse du compilateur CC disponible sur la version 8.2 de OpenVMS. On rappelle que les optimisation sont faites de manière logicielle au niveau des compilateur pour ce qui est de l'architecture EPIC.

MFLOPS BENCHMARK:

Ce benchmark vise à déterminer la puissance de calcul flottant d'un système en estimant le nombre maximum d'instructions flottantes qui peuvent avoir lieu en une unité de temps.

Le programme sollicite intensivement le FPU pour exécuter des instructions FADD, FSUB, FMUL, et FDIV.
Le code est écrit de telle façon à favoriser l'utilisation des registres et à minimiser l'interaction avec la mémoire principale, ce qui permet d avoir une estimation de temps d'exécution des instruction flottantes et diminue la latence du mémoire.
Les itérations sont de petites tailles et peuvent tenir dans la cache de telle sorte à minimiser les "Cache miss".
Flops.c est divisé en 8 modules (M) indépendants qui calculent l'intégrale de diverses fonctions mathématiques complexes, et 4 unités de tests (U) qui ne sont que des combinaisons des modules, déjà évoqués, de la façon suivante :

*U1 = 5*M2 + M3*
*U2 = M1 + M3 + M4 + M5 + M6 + (4*M7)*
U3 = M1 + M3 + M4 + M5 +M6 + M7 + M8
U4 = M3 + M4 + M6 + M8

Le tableau suivant décrit pour chaque module le nombre d'instructions à virgule flottante par itération.

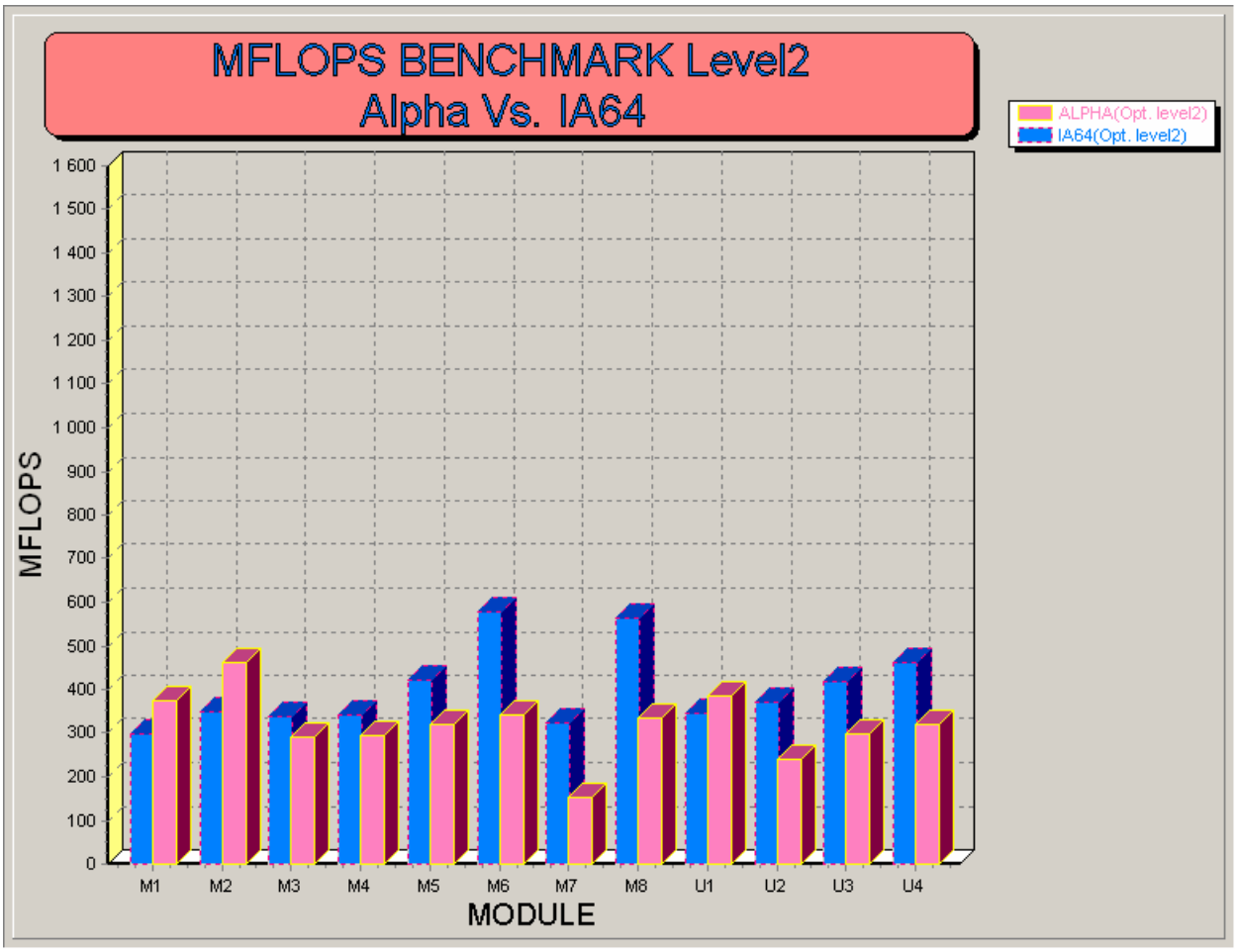
Module	FADD	FSUB	FMUL	FDIV	TOTAL
M1	7	0	6	1	14
M2	3	2	1	1	7
M3	6	2	9	0	17
M4	7	0	8	0	15
M5	13	0	15	1	29
M6	13	0	16	0	29
M7	3	3	3	3	12
M8	13	0	17	0	30
U1	21	12	14	5	52
U2	58	14	66	14	152
U3	62	5	74	5	146
U4	39	2	50	0	91

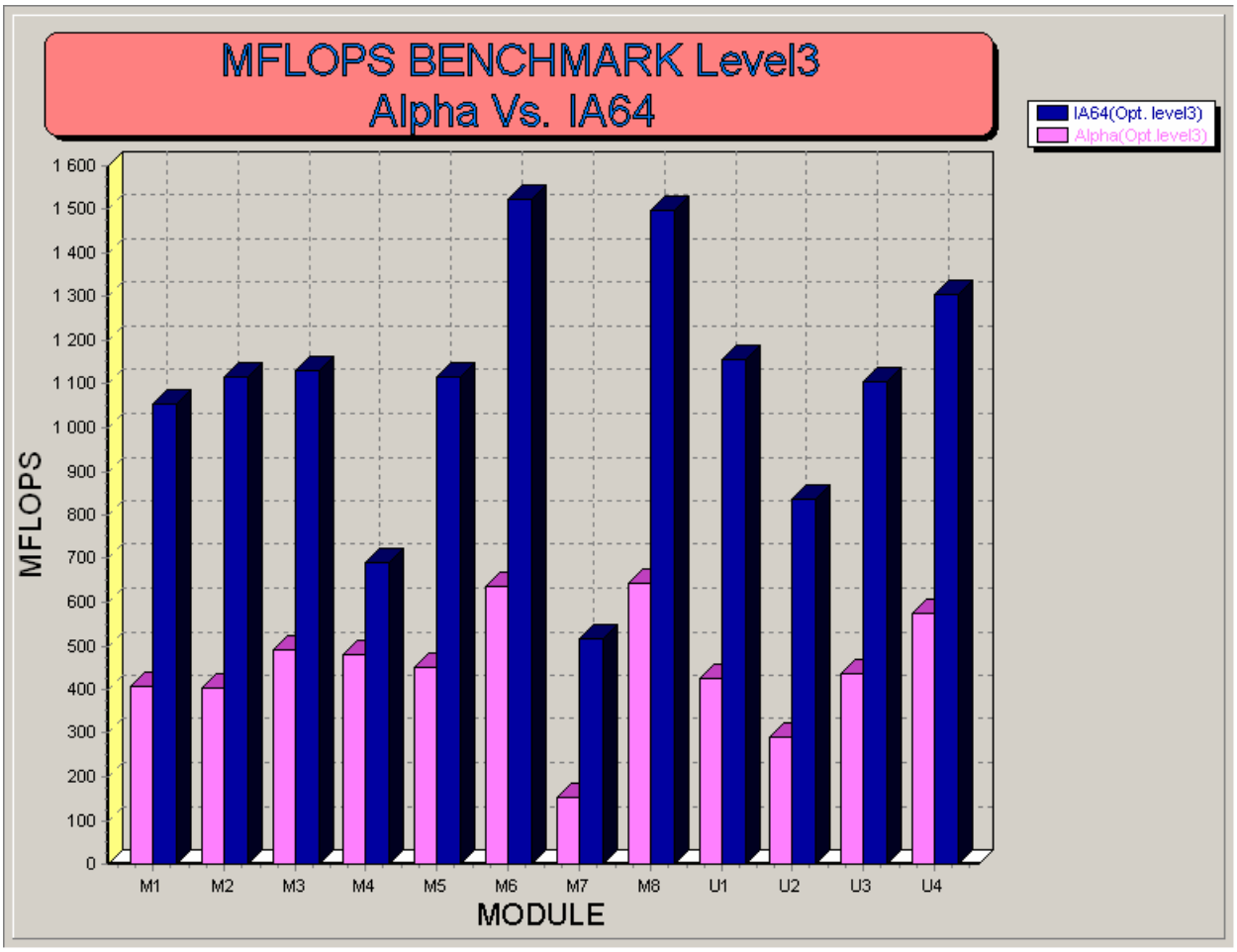
Parmi les choses qui nous ont marquées dans ce bench, le fait que M8 exécute 2.5 plus d'instructions que son prédécesseur M7 en 2 fois moins de temps.

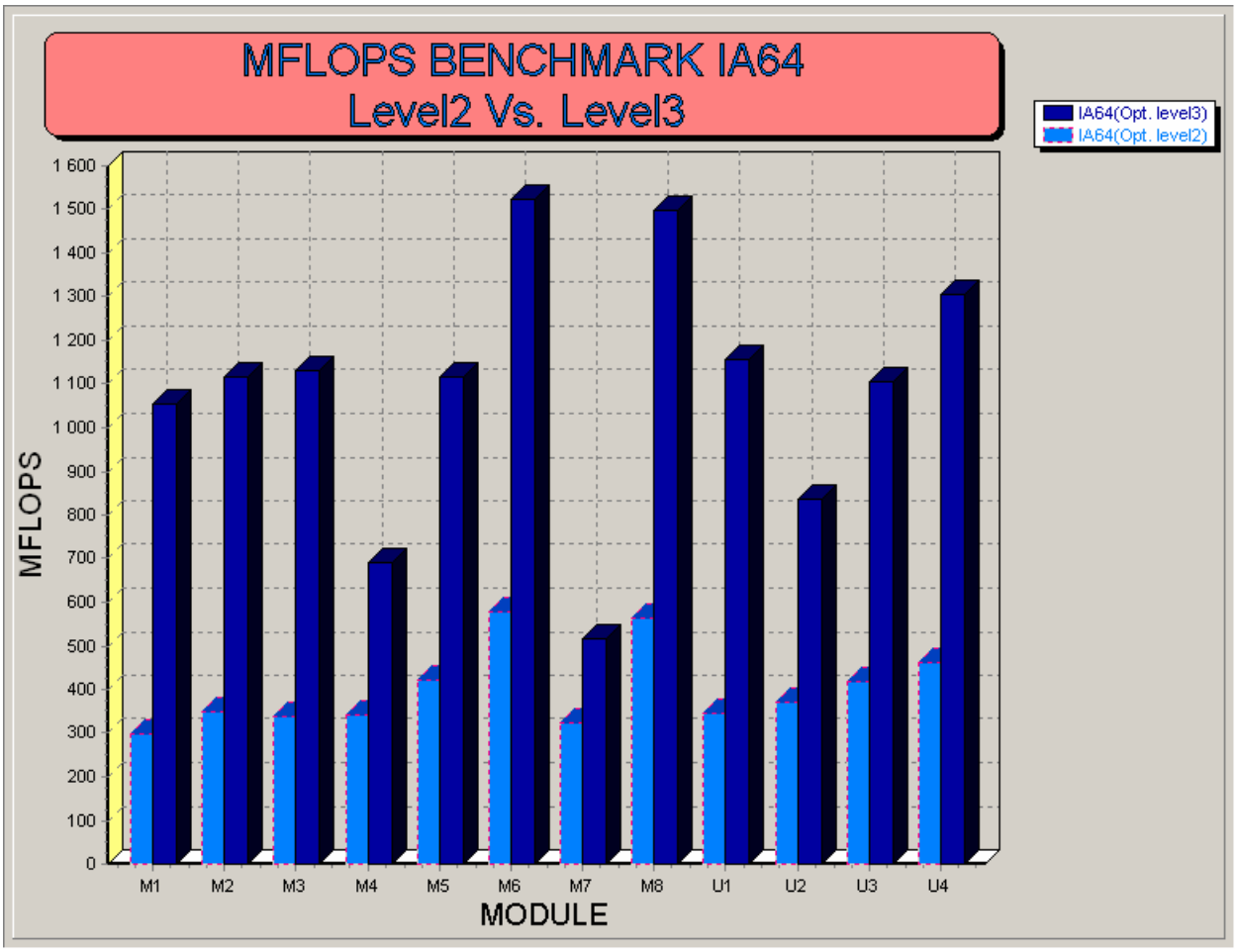
Ceci vient du fait que l'opération "Fdiv" très coûteuse en nombre de cycles se voit moins souvent du coté du M8.

Enfin, comme le montrent les figures suivantes, les opérations sur les flottants s'exécutent très efficacement sur un IA64 si on fait la comparaison avec un Alpha.

À noter que l'on sollicite, de nouveau, la capacité du compilateur à optimiser le code statiquement) et qu'on l'oppose aux optimisations faites par le processeur Alpha (dynamiquement) en compilant avec différents niveaux (Level) d'optimisation.

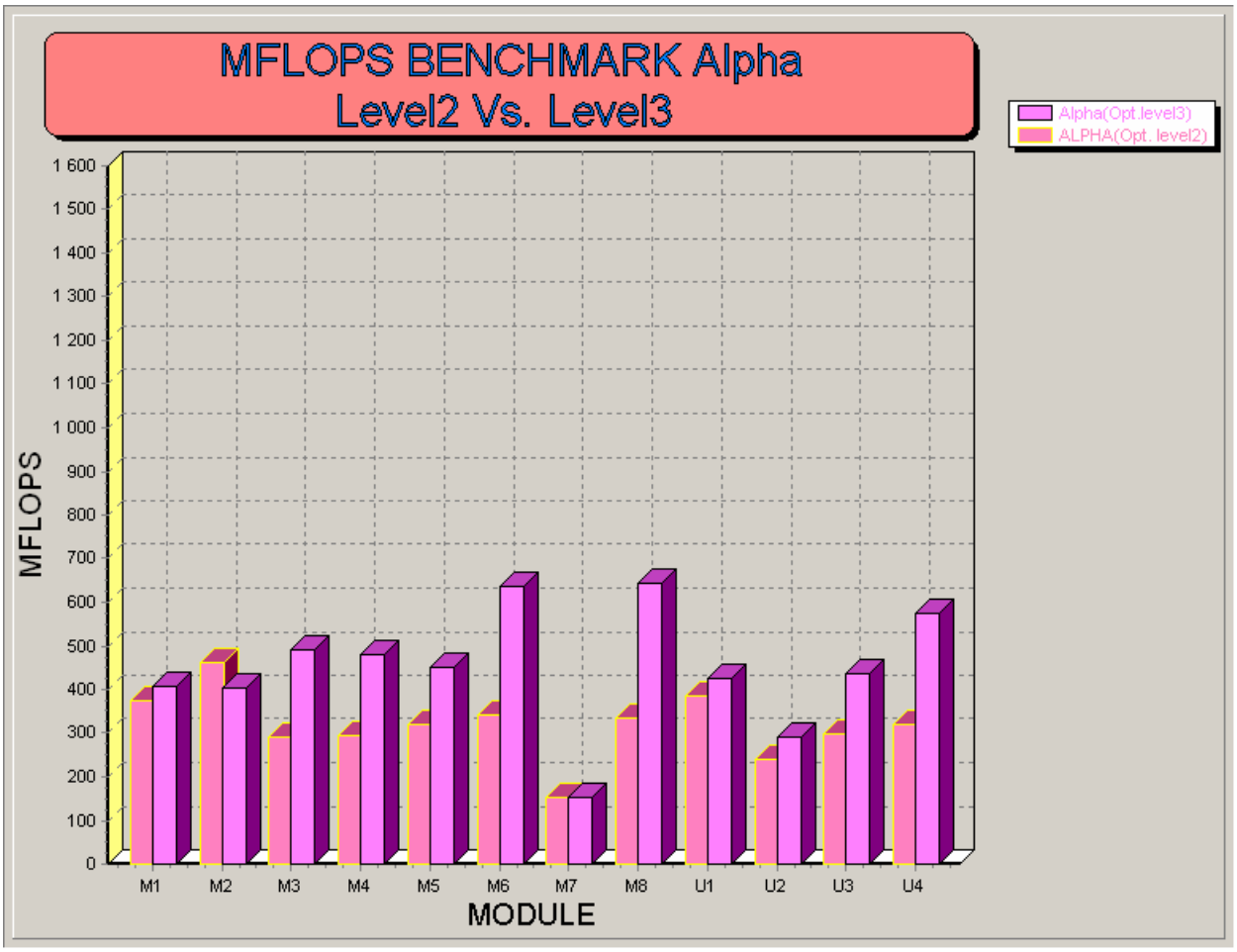



MFLOPS BENCHMARK Level3
Alpha Vs. IA64
IA64(Opt. level3)
Alpha(Opt.level3)
MFLOPS
1 600
1 500
1 400
1 300
1 200
1 100
1 000
900
800
700
600
500
400
300
200
100
0
M1
M2
M3
M4
M5
M6
M7
M8
U1
U2
U3
U4
MODULE



Le fait de passer à un niveau plus élevé d'optimisation faite par le compilateur de la machine Itanium a fait que les performances ont triplé !

Ceci corrobore le fait que ce sont les compilateurs qui ont la responsabilité de produire du bon code bien optimisé pour le processeur et qui sont maintenant à la charge d'exploiter au mieux les unités fonctionnelles de ce dernier.



QUEENS BENCHMARK :

Le problème de « queens » consiste à calculer le nombre de configurations possibles de positionner N dames sur un échiquier (N x N) sans qu'elles soient prises.
Le temps nécessaire de calculer ces configurations en fonction du nombre N de reines est donné dans le tableau suivant :

N	13	14	15	16
IA64	0.59s	3.58s	23.18s	160s
Alpha	2.1s	12.47s	79. 88s	545.750s

Nsieve BENCHMARK :

Ce bench se base sur l'algorithme du Crible d'Eratosthenes pour calculer les N premiers nombres premiers.Il est utilisé pour calculer le nombre maximal d'instructions entières que le CPU peut traiter par seconde (MIPS).
Les teste ont montré que sur IA64 , on peut atteindre jusqu'à d1450 MIPS contre seulement 458 MIPS du coté Alpha.

Hanoi BENCHMARK

C'est un benchmark qui résout le problème de la Tour de Hanoi en utilisant des appels récursifs. Le tableau ci contre montre les résultats d'exécution de ce bench sur les 2 architectures.

IA64 :

Disques	Déplacements	Temps(s)	Depl/25.usec
18	262143	0.01000	655.3575
20	1048575	0.01000	2621.4375
21	2097151	0.02000	2621.4388
23	8388607	0.09000	2330.1686
24	16777215	0.17000	2467.2375
25	33554431	0.33000	2542.0023
26	67108863	0.67000	2504.0621
27	134217727	1.33000	2522.8896
28	268435455	2.67000	2513.4406
29	536870911	5.33000	2518.1562
30	1073741823	10.67000	2515.7962

Moyenne de nombre de déplacements/ 25 usec = 2511.9407

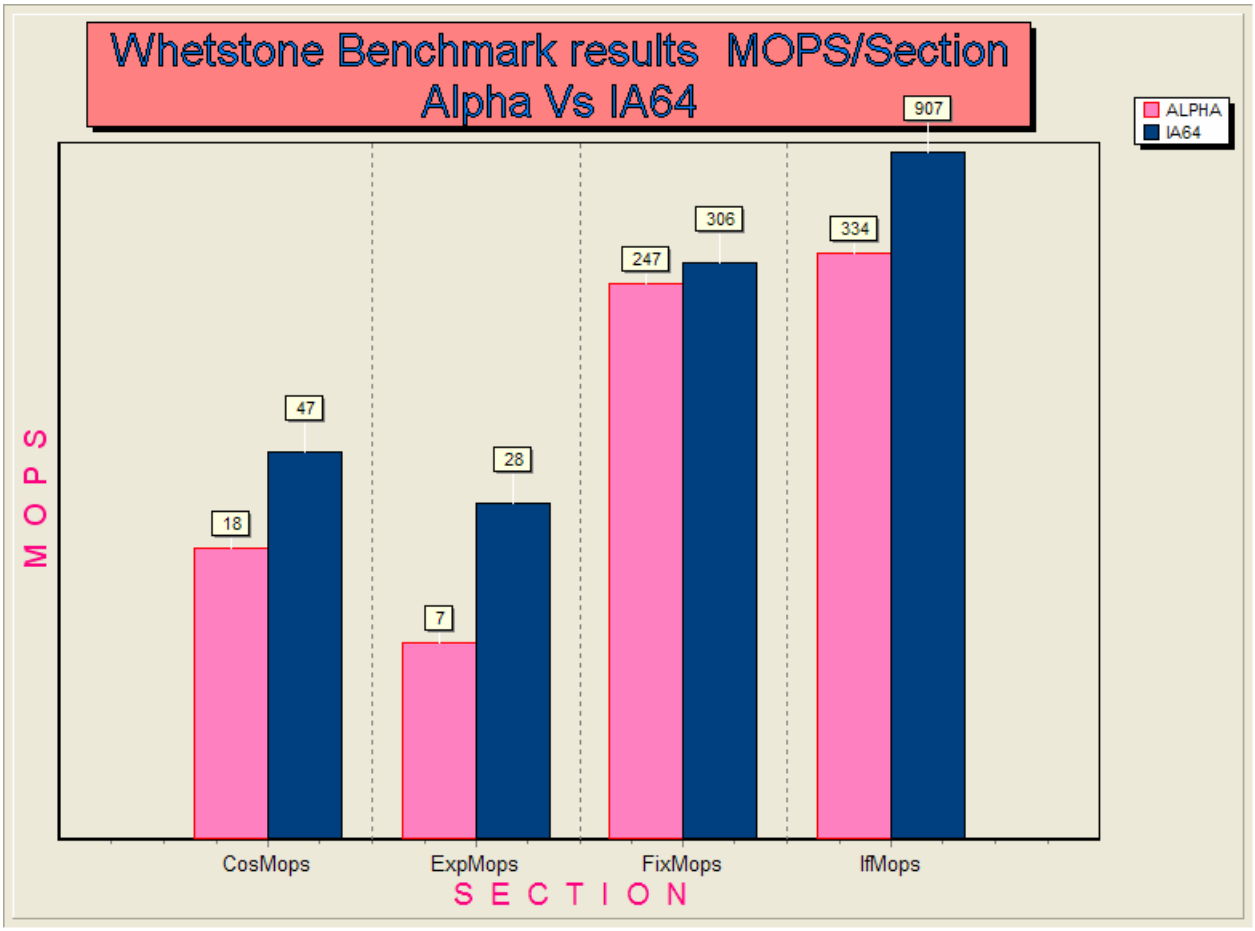
ALPHA:

Disques	Déplacements	Temps(s)	Depl/25.usec
17	131071	0.01000	327.6775
18	262143	0.01000	655.3575
19	524287	0.03000	436.9058
20	1048575	0.05000	524.2875
21	2097151	0.11000	476.6252
22	4194303	0.20000	524.2879
23	8388607	0.41000	511.5004
24	16777215	0.82000	511.5005
25	33554431	1.66000	505.3378
26	67108863	3.32000	505.3378
27	134217727	6.63000	506.1000
28	268435455	13.25000	506.4820
30	1073741823	53.02000	506.2909

WHETSTONE BENCHMARK:

Ce benchmark a été programmé par Harold Curnow en 1972 en FOTRAN .
Il mesure les performances du CPU à traiter les opérations sur les entiers et les flottants.
Il est mesuré en KWIPS = mille Instruction whetsone / secondes.
Le programme contient 8 boucles principales, reparties comme suit :
* 3 boucles qui opèrent sur les flottants.
* 2 boucles qui opèrent sur les appels de fonctions
* 1 boucle qui contient des affectations.
* 1 boucle qui contient des branchements (if-then-else).
* 1 boucle qui effectue du calcul arithmétique sur les entiers.

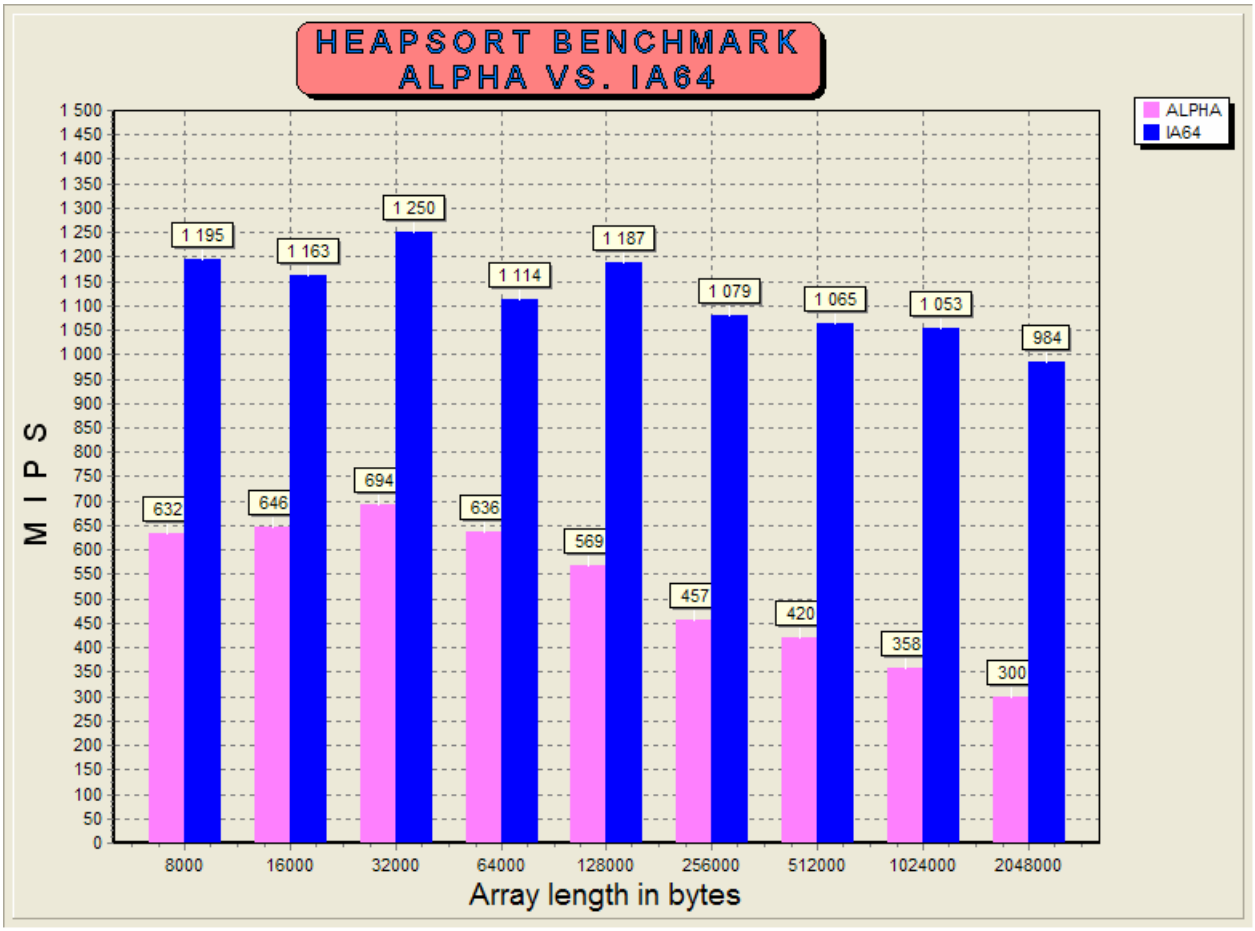
Les 3 premières boucles se mesurent en MFLOPS et le reste en MOPS(million d'opérations/secondes)
Lors de la phase de compilation, l'unique optimisation espérée par le compilateur, est le « in-lining » (le fait d'incorporer le code de la fonction appelée à la place de l'appel), car les restes d'optimisations n'ont pas d'impact fort sur les performances. Donc le pipeline logiciel, ne joue pas de rôle ici (voir chapitre précédent).



Lors de la recompilation sur Itanium, on a remarqué une augmentation légère de la taille de l'exécutable (c'est du à l'effet de « in-line ») par rapport à celui obtenu par compilation sur Alpha.
On voit de nouveau que l'alpha perd au profit de l'Itanium. Donc l'optimisation logicielle, même si elle est un peu pénalisante au niveau de la taille de l'exécutable, est meilleure que l'optimisation matérielle faite par le processeur Alpha, qui ici exploite son mécanisme d'exécution désordonnée des instructions via la prédiction à travers un appel de fonction (voir le chapitre précédent).

HEAPSORT BENCHMARK :

Ce bench se sert de l'algorithme de tri par tas, pour trier un tableau de « long ».
Il donne une estimation du nombre de MIPS pour les différentes architectures.



LINPACK :

Ce benchmark permet de mesurer les performances du calcul flottant des machines. Il est vu comme un benchmark sérieux, car il manipule de gros volumes de données en se contentant de résoudre un système d'équations linéaires en utilisant des matrices de taille 100x100.
Deux versions ont été testées et chaque version a été optimisée avec le « loop unrolling ».
La première version exécute ses instructions sur des flottants en simple précision (SP_ROLL , SP_UNROLL) et la seconde en double précision (DP_ROLL,DP_UNROLL). On remarque rapidement qu'en double précision, les performances ont chuté (ce qui semble normal, car les calculs en doubles précisions sont plus couteaux en nombre de cycle que ceux en précision simple).

Chapitre 5

Méthodologie de migration d'applications OpenVMS Alpha à OpenVMS IA64

1. Introduction :

L architecture OpenVMS I64 est basé sur le modèle 64-bit et sur des fonctionnalités similaires à celles de l'architecture Alpha, ainsi la plus part d'applications qui tournaient sur l'OpenVMS alpha peuvent aujourd'hui être portées sur la nouvelle architecture I64 avec un effort minime.

OpenVMS Alpha et OpenVMS I64 sont deux variantes de OpenVMS qui sont produites a partir du même code source et par la suite tous aspects non dépendant du matériel a été incorporé dans les deux version sans changements aux sources, minimisant ainsi le temps d'effectuer des testes de qualifications et assurant la disponibilité des applications critiques sur les deux plate-forme d 'OpenVMS.

"HP envisage de maintenir une politique stricte pour assurer que les applications qui tournaient sur un OpenVMS Alpha tourneront avec succès sur l'OpenVMS I64.

Les applications qui bénéficient des librairies et des services système doivent pouvoir sans modifications être portées sur la deniere version de OpenVms I64."

Des changements minimaux ont été apportés au nouvel environnement, mais la structure de base et les fonctionnalités d'OpenVms restent les mêmes.

2. Terminologie :

Il y a une différence majeure entre le vocabulaire migration et le vocabulaire transport dans le cadre de notre méthodologie. En effet, une migration consiste en le déplacement d'une application d'un environnement en un autre sans pour autant avoir à toucher à son code source (dans le cas où se dernier serait disponible). Toute réécriture de l'application ou toute modification aussi infinitésimale qu'elle soit est une transportation.

3. Différence entre OpenVMS Alpha et OpenVMS I64 :

Le protocole d'appel inter-procédural :

L'implémentation du protocole d'appel entre les procédures diffère légèrement sur la famille des processeurs Intel Itanium des conventions choisies par OpenVMS VAX et Alpha. Ceci a été établi dans le but de minimiser le coût et la difficulté de porter les

applications et le système d'exploitation lui même vers l'architecture Itanium.

La section suivante contient une description des différences des conventions d'appel entre les procédures sur les deux plate-formes.

Le Modèle des données:

OpenVms sur l'Alpha reste ambiguë vis à vis du modèle des données qu'il utilise: beaucoup de programmes semblent suivre le modèle ILP32, alors que le système, en grosse partie, opère en utilisant le modèle P64 ou LP64. Une règle de signe pour les entiers a permis de rendre cette ambiguïté plus ou moins transparente. OpenVMS I64 préserve cette caractéristique, alors que l'architecture de IItanium définie un modèle purement LP64.

Usage de registres généraux:

Les registres généraux sont utilisés pour les opérations arithmétiques. OpenVms I64 utilise les mêmes conventions pour ces registres sauf dans les cas suivants:

- R8 et R9 (seuls) sont utilisés pour la valeur de retour.
- R25 est utilisé pour passer des informations sur un argument.

Usage de registres flottants:

Les registres flottants sont utilisés pour le calcul sur les nombres à virgule flottante, ainsi que les émulation des flottants de VAX, et certains calcul sur les entiers.

OpenVms I64 utilise les mêmes conventions pour ces registres sauf dans le cas suivant:

- F8 and F9 (only) sont utilisés pour la valeur de retour.

Passage de paramètres :

Le passage des paramètres sous OpenVMS ressemble aux conventions de l'Itanium. Voici une description des différences principales :

- Le standard OpenVms ajoute un "registre d'information" pour permettre de connaître le nombre d'arguments lors d'un appel .
- Lors du passage des paramètres en registres, le premier paramètres est passé soit dans R32 soit dans F16, le second dans R33 ou F17 et ainsi de suite. On ne peut pas passer plus de 8 paramètres en registres.
- Lors du passage d'un paramètre à 32 bits dans un registre général, le bit 32 est assigné au signe du paramètre (n° 31 ca r on commence à R0).
- OpenVMS supporte seulement la représentation « I ittle-endian ».
- OpenVMS supporte les trois types de flottants VAX F_float (32 bits), D_float (64 bits) et G_floating (64 bits). Les valeurs de ces derniers sont passées dans les registres généraux.

Les valuers de retour :

Les valeurs de retour jusqu'à 16 octets de taille peuvent être retournées en registre. Les valeurs qui ont des tailles plus grandes sont placées dans le premier ou le deuxième emplacement des arguments.

Les Flottants:

L'architecture Alpha supporte les formats IEEE et VAX au niveau matériel. Les compilateurs d'OpenVms génèrent du code en utilisant le format de VAX par défaut, avec la possibilité sur l'Alpha de d'utiliser le format IEEE.

L'Itanium quand à lui , implémente seulement le format IEEE au niveau matériel, et supporte les IEEE simple et double précision.

Si une application a été écrite pour OpenVMS VAX ou OpenVms Alpha en utilisant les flottants par défaut, elle peut être migrer sur OpenVMS I64 de deux manières:

- Continuer a utiliser le formats VAX.Les compilateurs détectent ces formats et gênèrent du code qui converti ces formats en IEEE , et reconverti les résultats de nouveau en format VAX . (ceci peut engendrer une chute des performances, car les données sont converties à l'exécution)
- Convertir les données VAX et IEEE avec un convertisseur fourni par HP, et laisser l'application tourner avec ces données.

4. Évaluation d'applications en préparation à la migration :

L'évaluation de l'application détermine les étapes nécessaires a sa migration au nouvel environnement OpenVMS I64.

Le résultat de l'évaluation doit être un plan de migration qui réponds aux questions suivantes:

- Comment migrer l'application
- Combien de temps et d'effort la migration nécessite .
- Avez vous besoin du support des services de HP ?

L'évaluation doit inclure les étapes suivantes:

- Etablissement d'un cahier des charges de l'application pour identifier ses dépendances des autres logiciels et librairies.
- Sélection de la méthode de migration (recompilation du code source ou traduction du binaire).

La première étape dans le processus d'évaluation est de déterminer exactement ce qui doit migrer. Ceci n'inclut pas l'application seule mais aussi tout ce dont lelle dépend et qui assure son exécution correcte.

Pour commencer à évaluer l'application, on localise les éléments suivants:

- **Parties de L'applications :**

- Les codes sources pour le programme principal.
- Les bibliothèques et les images partagées.
- Les fichiers de configuration, (scripts, fichier de base de données, fichiers lues ou écrits par l'application)
- Les fichiers de documentation.

- **Autres composants binaires, dont dépends notre application :**
  - Bibliothèques de chargement dynamique.
  - Caractéristiques nécessaires de l'environnement dans lequel tourne l'application, par ex: Quantité de mémoires requise, espaces disque, etc.
  - Les outils de test, d'analyse et de vérification qui permettent de migrer, tester l application et d'évaluer ses performances.

Beaucoup de ces éléments ont déjà été portés vers le nouvel environnement, comme par ex les bibliothèques, les run-time images, les compilateurs et leur RTLs ..

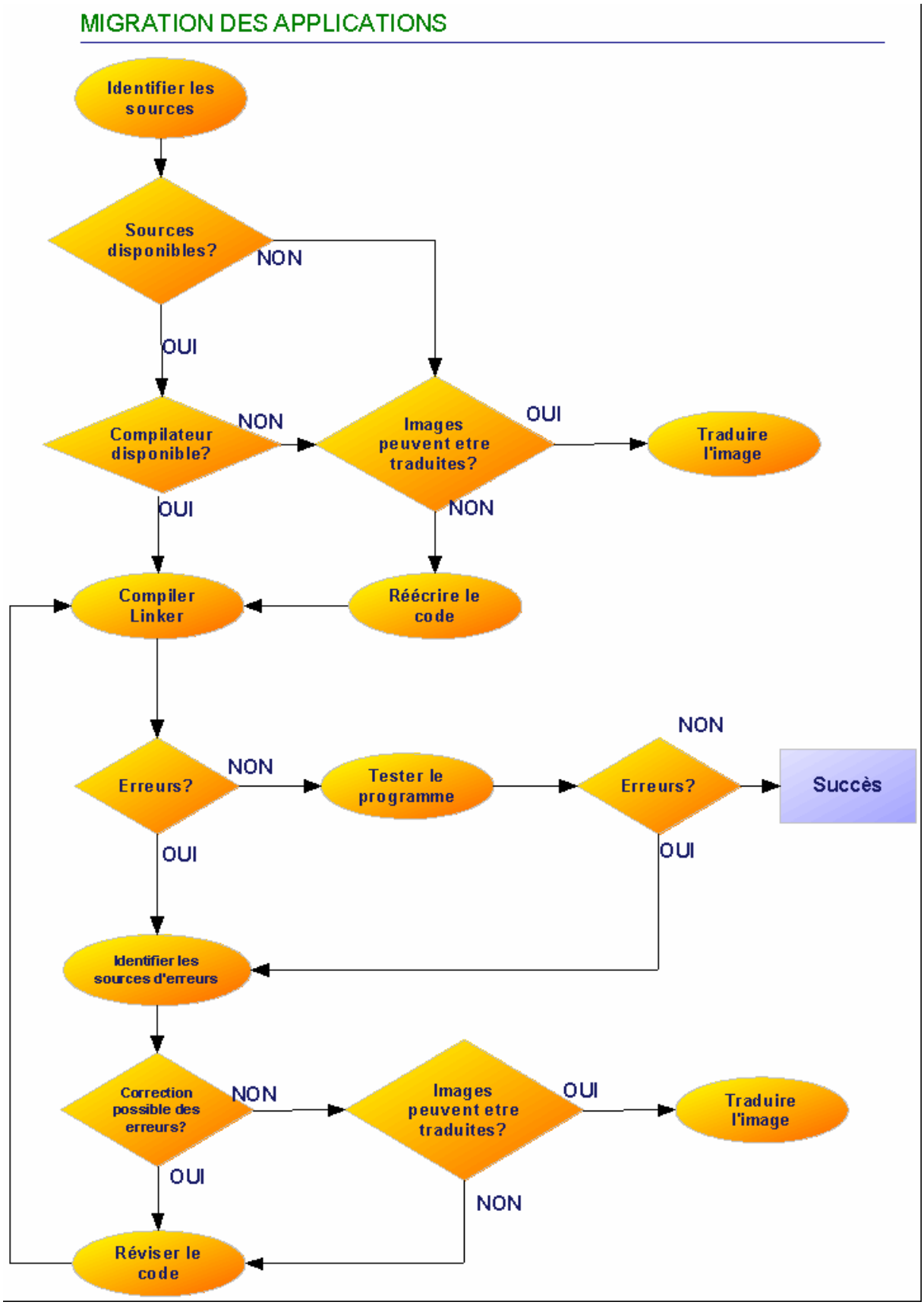
5. Choix de la méthode et processus de la migration :

Une fois le plan d'évaluation établi, l'application doit être migrée vers le nouvel environnement.

Deux méthodes d'action sont possibles :

- la compilation .
- la traduction binaire.

Le choix entre ces deux méthodes est le fruit de la réponse aux questions et de l'accomplissement des tâches de la figure suivante:


MIGRATION DES APPLICATIONS
Identifier les sources
Sources disponibles?
NON
OUI
Compilateur disponible?
NON
Images peuvent etre traduites?
OUI
Traduire l'image
OUI
NON
Compiler Linker
Réécrire le code
Erreurs?
NON
Tester le programme
Erreurs?
NON
Succès
OUI
OUI
Identifier les sources d'erreurs
Correction possible des erreurs?
NON
Images peuvent etre traduites?
OUI
Traduire l'image
OUI
NON
Réviser le code

PHASE DE MIGRATION:

Cette phase inclut les étapes suivantes:

1) Préparation de l'environnement de migration :

Pour préparer un environnement efficace de migration, il faut vérifier les éléments suivant:

- les outils de migration : (compilateurs, traducteurs binaire, TIE, RTLs, les libraires du système, ainsi que les fichiers d'inclusion des programmes C).
- Les noms logiques.
- Les outils de test et d'analyse.

2) Compilation de vos sources sur la dernière version d'OpenVMS Alpha :

Ceci permet de couvrir les erreurs qui peuvent avoir lieu sur l'OpenVMS I64. Les nouvelles version de compilateurs sont plus expressives et introduisent des standards plus strictes d'interprétation.

3) Tester l application sur l'Alpha :

Cette phase de test permet d'établir des statistiques de départ sur les performances de l'application. On pourrait par la suite, comparer ces résultats et les opposer à celles des tests **similaires** sur le système IA64(voir étape 5).

5) Recompiler et linker votre application sur le système I64:

En général , la migration de l'application nécessite des cycles minutieux de révision, de compilation, du link et de debogage du code.

Durant ce processus, on résout les erreurs syntaxiques et logiques par les outils de développement.

Les erreurs de syntaxe sont faciles à corriger, alors que les erreurs logiques nécessitent plus d'effort et de modification du code.

5) Tester l application migrée:

Tester l'application permet de comparer les fonctionnalités de la version migrée contre celle de l Alpha, établies à l 'étape 3.

La comparaison reste valides si on a activé les mêmes chemins d'exécution du code avec les même fréquences et en manipulant les mêmes volumes sur les deux machines.

6. Limites :

On va exposer les limites sans pour autant les détailler car comme dit dans la section « terminologie » ceci rentre dans le cadre du transport d'applications et on ne parle plus d'une simple migration.

- Code écrit en Macro 64 d'Alpha : Ce code doit être ré-écrit dans un autre langage.
- Code qui a été conditionné pour être exécuter sur VAX ou Alpha. Ce code doit être réviser pour prendre en compte l'IA64.
- Code qui utilise les services systèmes dépendant de l architecture Alpha.
- Code qui utilise les threads.
- Code privilégié.

Notons que ces limites sont pour la plupart issues des vieilles habitudes de programmations ou d'applications trop liées aux architectures pour lesquelles elles ont été crées et que la probabilité de les croiser est infime.

Une autre limite reliée au mécanisme de traduction du binaire a été constatée. En effet, parfois il y a des chutes de performances de l'application traduite par rapport à sa recompilation. Ceci est dû au fait que la traduction ne cible pas que l'application elle même mais, malheureusement dans certains cas, elle est obligée de prendre toutes bibliothèques et librairies de l'ancien environnement auxquelles celle-ci est liée et de les traduire aussi avec elle.

Chapitre 5

Conclusion

1. Bilan technique :

Malgré son architecture sophistiquée et en dépit du fait qu'il jette toute la responsabilité sur les compilateurs, l'Itanium a montré son efficacité à travers la quasi-totalité des benchmarks qu'on a fait tourner. OpenVMS V8.2 a prouvé qu'il peut très bien assumer la responsabilité et bien joué le rôle que lui a confié le processeur en offrant une panoplie de compilateurs assez robustes qui réussissent à faire d'une manière statique plus efficace ce que faisait les processeurs traditionnels dynamiquement d'une manière matérielle.

Aucun impact sur la sécurité et les performances des applications migrée n'a été remarqué.

2. Perspectives résultantes de ce travail :

Ce travail a été l'occasion pour nous de voir comment se passent les choses en pratique et nous a permis de mettre en œuvre les connaissances théoriques qu'on a acquises tout au long de notre cursus universitaire.
C'était aussi une chance de voir de plus près le monde industriel, vu que ce travail s'est déroulé en collaboration avec HP, qui a ses propres contraintes et qui ne ressemble pas toujours au monde universitaire.
Aussi avons nous appris de nouvelles connaissances et nous sommes nous familiarisés avec un nouveau système d'exploitation et des nouvelles plates-formes.

3. Extensions possibles de ce travail :

Ce travail pourrait être repris et poursuivi en vue de couvrir la partie traduction du binaire alpha. En effet, le mécanisme de traduction dans ses détails pourrait, à lui seul, faire l'objet d'un travail d'étude et de recherche.

Les équipes futures peuvent compléter un point resté inachevé dans ce projet à savoir l'étude d'une application pratique de l'ordre de celle de la Française des jeux, qu'on a malheureusement pas eu la chance d'aborder, faute de fourniture.

Un autre point peut être traité en extension de ce travail qui est la migration ou plutôt le transport d'applications Vax (la génération antérieure à Alpha) vers la plate forme Itanium.

Bibliographie

Internet :

1) Essentiellement le site OpenVMS de HP:
http://h71000.www7.hp.com/

2) Le manuel d'utilisateur OpenVMS :
http://tigger.stcloudstate.edu/~tigger/openvms/72final/6489/6489pro.html

3) Cours d'architecture des ordinateurs IUP GEII de l'université de Tours :
http://www.blois.univ-tours.fr/~marcel/archi/node195.html

4) document sur les clusters:
http://www.transtec.de/D/F/IT-Kompendium/ITKnowHow/Clusters/HighAvailabilityCluster.html

Livres :

1) The VMS User's Guide (Digital Press)
James F. Petrs, III
Patrick J. Holmay

2) IA-64 linux kernel design and implementation (hp invent)
David Mosberger
Stéphane eranian

3) Assembleur (Micro Application)
Pierre Maurette

4) The Theory of Parsing, Translation, and Compiling Volume I : Parsing
Alfred V. Aho
Jeffry D. Ullman

5) Les processeurs Itanium (eyrolles)
Smail Nar

6) Compilateurs Principes, techniques et ou outils (InterEditions)
Alfred aho
Ravi Sethi
Jeffrey Ullman

Autres supports :

http://www.raytheon-computers.com/ref_docs/alpha_ia64.pdf

http://www-lsr.imag.fr/Les.Personnes/Martin.Heusse/Appr/eaPres/Architectures64bits.pdf

www.decus.de/sig/vms/TUD_2004/VMS_Update.PDF

Annexe A

Exemple d'un fichier de commande DCL

```
$!*********************************************************
$! TER / MAITRISE INFORMATIQUE 2005
$!! ABOUNADA SAMI
$!!  BOUKADIDA MOHAMED J.
$!!  KSOURI SONIA
$! script qui lance l'exécution des benchmarks et sauvgarde leurs resultats.
$! La sortie standard des benchs est redirigée vers des fichiers de sauvgarde.
$! Par ex : Si le bench s 'appele toto.exe , sa sortie standard est redirigé vers le fichier tot.res
$! Ce fichier est sauvgarde dans un repertoire qui possede un nom logique sys$ter$res
$!*********************************************************
$ mess "NOEUD: " + THIS_NODE + "/" + ARCH_NAME
$ mess "####################"
$
$ TYPE = "NATIVE"
$ cd sys$ter$res
$ IF ARCH_NAME .EQS. "Alpha"
$ THEN
$    DIR_EXE = "sys$TER$alpha"
$ ELSE
$    DIR_EXE = "sys$TER$ia64"
$    IF ""P1"" .EQS. "T"
$      THEN
$        DIR_EXE = "sys$TER$trans"
$        TYPE = "TRANSLATED"
$        GOTO OK
$    ENDIF
$
$    REPONSE := "N"
$    cd [.ia64]
$    read/time_out=5 sys$command -
     /prompt="VEUILLEZ CHOISIR LE TYPE E'EXECUTION (Native/Translated) [N]? " -
     /end_of_file=OK -
     REPONSE
$    IF  ""REPONSE"" .EQS. "T"
$      THEN
$        DIR_EXE = "sys$TER$trans"
$        TYPE = "TRANSLATED"
$        cd [-.translated]
$    ENDIF
$ ENDIF
$
$ OK :
$ FILENAME = F$SEARCH(DIR_EXE+":*.EXE")
$ IF F$LENGTH(FILENAME) .EQ. 0
$ THEN
$   mess " %ERR: LE REPERTOIRE "+  f$trnlnm(DIR_EXE) + " EST VIDE"
$   IF TYPE .EQS. "NATIVE"
$     THEN
$       mess "       COMMENCER PAR COMPILER LES SOURCES AVEC LE BATCH"
```

```
$      mess "      CO-c MPILE.COM"
$    ELSE
$      mess "      COMMENCER PAR TRADUIRE LES BINAIRE AVEC LE BATCH"
$      mess "      TRANSLATE.COM"
$  ENDIF
$  GOTO END
$ ENDIF
$ mess "PHASE D EXECUTION [",TYPE,"]
$ LIMIT_COUNTER = 5
$
$
$    TOP:
$        BASENAME = f$element(0,".",f$element(1,"]",FILENAME))
$        BASENAME = f$edit(BASENAME,"upcase")
$        mess ""
$        mess " => EXECUTION DU "+ BASENAME
$        T := $'FILENAME'
$        DEFINE /process sys$output 'BASENAME'.res
$        COUNT = 0
$        LOOP :
$            mess "TEST NUMBER [" , 'COUNT' , "]"
$            if BASENAME .EQS. "QUEENS"
$              THEN
$                T -c ""'COUNT'+10"
$              ELSE
$                T
$            ENDIF
$            COUNT = COUNT + 1
$            if COUNT .NE. LIMIT_COUNTER
$             THEN
$               GOTO LOOP
$            ENDIF
$        FILENAME = F$SEARCH(DIR_EXE+":*.EXE")
$        IF F$LENGTH(FILENAME) .EQ. 0
$            THEN
$             GOTO END
$        ENDIF
$
$    GOTO TOP
$
$ END:
$ CD sys$ter$com
$ purge
$ exit
```

Annexe B

Logiciels installés pour l'étude expérimentale

```
############################################################
Softwares installés sur l'alpha et l itanium pour le TER:
############################################################

#######
Alpha :
#######
 |
 |
 |
 |=== Alpha Environment Software Translator (AEST V1.0) for Alpha systems
 |    (http://h71000.www7.hp.com/openvms/products/omsva/omsais_license.html)
 |
 |=== HP Digital Continuous Profiling Infrastructure (DCPI T2.0)
 |    (http://h71000.www7.hp.com/openvms/products/dcpi/)
 |
 |=== GNV V1.6-2 *
     (http://gnv.sourceforge.net/)


#########
Itanium :
#########
 |
 |
 |=== Alpha Environment Software Translator (AEST V1.0) for Integrity servers
 |    (http://h71000.www7.hp.com/openvms/products/omsva/omsais_license.html)
 |
 |=== Translated Image Environment (TIE V1.0) software component V1.0
 |    (http://h71000.www7.hp.com/openvms/products/omsva/omsais_license.html)
 |
 |=== GNV V1.6-2 *
 |    (http://gnv.sourceforge.net/)
 |
 |=== DECset |
         |_
           PCA :: DIGITAL Performance and Coverage Analyzer for OpenVMS
          (CD d'installation de HP)


###############
Les benchmarks
###############
Les benchmarks sont accessibles depuis: ftp://ftp.nosc.mil/pub/aburto/
```

* GNV is a GNU based project, delivering a Unix-like environment for
OpenVMS. It is intended to provide the important subset of Unix/Linux/POSIX
necessary to port UNIX OpenSource software to OpenVMS.
GNV consists of a Unix like shell environment and many of the tools and utilities common to
Unix shell environments.