Génération de trace avec *pixie* sous IRIX 5.3, et Analyse de l'impact des dépendances de données à partir des SPEC92

Stéphan Jourdan et Pascal Sainrat
Equipe Architectures PARAllèles
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier, Toulouse III
118 route de Narbonne, 31062 Toulouse cedex, FRANCE

Rapport Interne IRIT/95-10-R
30 Mars 1995, Version 1.0

Table des matières

1 Introduction

Pixie est un outil d'instrumentation pour processeur MIPS qui permet de générer des traces ou des statistiques statiques et dynamiques pour l'analyse de performance. Il modifie le code afin de recueillir, lors de l'exécution, des informations dynamiques. Le programme est directement exécuté sur la machine sans aucune émulation. L'exécution n'est donc que très légèrement ralentie.

Ce document se base sur [1] qui décrivait l'utilisation de *pixie* sous IRIX 4.x. Le passage de la version 4 à la version 5 s'est accompagné du changement du format des fichiers binaires (passage de COFF à ELF). *Pixie* s'en est trouvé grandement modifié. De plus, l'option de génération de traces ne marche plus sous IRIX 5.1 ni IRIX 5.2, mais de nouveau sous IRIX 5.3. Ce document relate donc les nouveautés dans l'utilisation de *pixie*.

Ce document décrit dans la section 2 comment l'utiliser de façon simple. La section 3 présente son utilisation en tant que générateur de traces d'instructions ou de données. La section 4 donne un exemple d'une telle génération sur les SPEC92. La méthode avec laquelle nous calculons des résultats moyens est démontrée dans la section 5. Enfin, un exemple d'analyse des traces est présenté dans la section 6 : l'analyse porte sur l'impact des dépendances de données.

2 Utilisation simple de pixie

La méthode d'utilisation de *Pixie* est identique quel que soit l'usage. La commande *pixie* prend en entrée un programme exécutable, le partitionne en blocs de base, et produit un nouveau programme exécutable contenant du code additionnel qui compte le nombre de parcours de chaque bloc ainsi que diverses informations suivant les options précisées. Un bloc de base est défini comme une région du programme où l'on ne peut entrer que par la première instruction et sortir que par la dernière.

Par exemple, supposons que nous ayons un programme exécutable pgnom. Si nous appelons la commande pixie comme suit :

```
pixie pgnom
```

Pixie crée alors un fichier binaire pgnom.pixie. C'est la nouvelle version du fichier exécutable. Il est lancé de manière identique à l'original. Si pgnom prenait deux arguments sur la ligne de commande et avait une redirection de l'entrée comme suit :

```
pgnom arg1 arg2 < pgnom.input
```

Alors, pgnom.pixie serait lancé de manière identique :

```
pgnom.pixie arg1 arg2 < pgnom.input
```

Exécuter pgnom.pixie produit la même sortie que l'original pgnom; néanmoins, un fichier de données supplémentaire nommé pgnom.Counts est créé.

Ce dernier fichier enregistre le nombre de parcours de chaque bloc de base lors de l'exécution. Le format de ce fichier n'est pas décrit dans ce document car nous ne nous en servons pas pour la génération de traces, il l'est par contre dans [1]. Les analyseurs *pixstats* et *prof* se basent entre autres sur ce fichier pour produire leurs résultats.

Le fichier pgnom.Addrs n'a pas survécu au passage d'IRIX 4.x à IRIX 5.x. Ce fichier, contenant les déplacements vis à vis de l'entrée de chaque bloc de base, n'est plus créé lorsqu'on *pixifie* un programme.

2.1 Pixie

Nous avons décrit ci-dessus l'utilisation la plus simple de *pixie*, nécessaire pour l'emploi de *pixstats* ou *prof* 2.2. Toutefois, d'autres buts peuvent être poursuivis. Les modifications ne portent alors que sur la commande de *pixification* du programme original, l'exécution ne créant que les informations spécifiées.

Nous ne passerons pas en revue les options standards comme le changement de nom des divers fichiers produits, les *man pages* étant assez explicites.

Par contre, comme on peut le remarquer lors d'une exécution de *pixie*, le logiciel parcourt aussi les librairies partagées comme la *libc* pour les programmes C. Il effectue donc un découpage en bloc de base des librairies appelées, créant ainsi les fichiers correspondants, par exemple :

libc.so.1.pixie (librairie C standard)   libm.so.1.pixie (librairie mathématique)

Toutefois, ces fichiers ne sont créés que s'ils n'existaient pas auparavant. Pour permettre à *pixie* de les retrouver, il est nécessaire de mettre à jour la variable d'environnement LD_LIBRARY_PATH pour qu'elle pointe vers le répertoire contenant les librairies *pixifiées*. Ceci est particulièrement important lorsqu'on utilise les analyseurs *pixstats* et *prof* afin qu'ils puissent retrouver le code des librairies partagées. Par exemple, changer cette variable vers le répertoire $HOME/repertoire_librairies se fait comme suit :

```
setenv LD_LIBRARY_PATH $HOME/repertoire_librairies
```

Néanmoins, les exécutables peuvent ne pas faire appel à ces librairies partagées si les versions non partagées de ces librairies sont disponibles (généralement sous /usr/lib/nonshared). Il suffit de l'indiquer à la compilation avec l'option -non_shared qui est commune à tous les compilateurs pour machine MIPS. L'inconvenient est la taille de l'exécutable créé qui est plus importante.

Une option de *pixie* permet de choisir le type du jeu d'instructions produit comme lors des compilations : -mips1 choisit le jeu d'instructions du R2000/R3000, alors que -mips2 concerne les microprocesseurs plus récents (à partir du R4000). La caractéristique principale de ce dernier jeu d'instructions est la gestion de données 64 bits. Le choix par défaut est le jeu *mips1*.

L'option -branchcounts sert à annoter le programme afin de connaître dynamiquement le nombre de branchements qui ont été pris ou non pris. Le problème des anciennes versions de *pixie* concernant le dépassement du déplacement des branchements dans le cas de code rajouté, semble être réglé sans nécessiter cette option. L'instruction est alors changée en un saut. En effet, l'adjonction de code peut rendre le déplacement d'un branchement trop grand pour rentrer dans le champ supposé (16 bits).

Enfin, la génération de traces concerne les traces d'instructions (option -itrace), les traces de données (option -dtrace), ou les traces d'instructions et de données (option -idtrace). Le format des traces est explicité à la section 3. Elles sont générées lors de l'exécution du programme *pixifié* vers le descripteur de fichier 19 par défaut. Ce numéro peut être changé avec l'option -idtrace_file. Il est important de savoir, qu'à ce jour, sous les version 5.x de IRIX, les exécutables ne doivent pas faire appel à des librairies partagées, ni utiliser des instructions *mips2*, pour pouvoir être tracés. Il faut donc obligatoirement les compiler avec les options -non_shared -mips1.

2.2 Pixstats et Prof

Ces deux utilitaires sont des analyseurs des caractéristiques dynamiques d'un programme. Leur utilisation essentielle concerne l'optimisation de programmes. Les fonctions principales de *pixstats* ont été incorporées dans *prof* et *pixstats* n'existera plus dans les futures versions du système d'exploitation en raison de ce double emploi. Un exemple d'utilisation est le suivant :

```
pixie pgnom
setenv LD_LIBRARY_PATH .
pgnom.pixie
pixstats pgnom
```

ou

```
prof -pixie pgnom pgnom.Counts
```

Ces outils sont correctement décrits dans les *man pages*.

3 Génération de traces

3.1 Capture des traces

Pour créer un exécutable générant les traces dynamiques des instructions et des données vers le descripteur de fichier 19, il suffit de lancer la commande suivante [1]:

pixie -idtrace pgnom

Ceci génère un fichier pgnom.pixie qui, lors de son exécution, créera un fichier pgnom.Counts et une trace dans le descripteur 19. Cette trace doit être redirigée car sinon l'exécution s'interrompt puisque ce descripteur n'est pas ouvert. Une manière facile de le faire est d'utiliser un *shell Bourne*. Il est alors nécessaire de la rediriger vers un descripteur inférieur à 10, 5 par exemple, comme suit :

```
pixie -idtrace_file 5 -idtrace pgnom
sh
pgnom.pixie 5>pgnom.seq
```

La trace est ainsi capturée dans le fichier pgnom.seq. Une autre manière consiste à créer deux processus communiquant par un tube UNIX. L'exemple 1 suivant présente un tel procédé, le premier argument donne le nom du fichier, le second le nombre d'instructions que l'on exécute sans enregistrer à partir de la première instruction, et enfin le troisième le nombre d'instructions à enregistrer à la suite. Le calcul du nombre d'instructions présuppose de connaître le format de la trace. Ceci sera explicité dans la sous-section suivante. Les traces sont enregistrées dans le fichier sortie.

```
#include <stdio.h>
#include <fcntl.h>

#define Carton_Rouge(a,b) { fprintf(stderr,a);exit(b);}

void main(argc,argv)
int argc;
char **argv;
{
        int TUBE[2];
        int PERE;
        int temp;
        FILE *descr;
        int nb_dep;
        int nb_enr;
        int nbt;

        if (argc<4)
                Carton_Rouge("Nombre d'arguments\n",1);
        argv++;
        nb_dep=atoi(argv[0]); /* nb d'instructions a depasser */
        argv++;
        nb_enr=atoi(argv[0]); /* nb d'instructions a enregistrer */
        argv++;

        printf("Nombre d'instructions depassees: %d\n",nb_dep);
        printf("Nombre d'instructions enregistrees: %d\n",nb_enr);

        if (pipe(TUBE)==-1)
```

[1] L'option -idtrace précise la génération de traces d'instructions et de données. On peut utiliser l'option -itrace pour générer seulement des traces d'instructions, ou -dtrace pour des traces de données.

```
                Carton_Rouge("Creation Pipe\n",2);
        if ((PERE=fork())==-1)
                Carton_Rouge("Echec du Fork\n",3);
        if (PERE)
        { /* ecrit */
                close(TUBE[0]);
                close(8); /* on a utilise le descripteur 8 pour creer le fichier
pixifie*/
                dup2(TUBE[1],8);
                close(TUBE[1]);

                execv(argv[0],argv);
                printf("OK EMETTEUR\n");
        }
        else
        { /* recoit */
                int nb,ref,cop;

                close(TUBE[1]);
                if ((descr = fopen("sortie","wb"))==NULL)
                        Carton_Rouge("Ouverture fichier\n",4);
/* depassement */
                nbt = 0;

                while ((nbt<nb_dep)&&(read(TUBE[0],&temp,sizeof(int))))
                {
                        cop = temp & 0x00ffffff;
                        nb = (temp>>28) & 0x0000000f;
                        ref= (temp>>24) & 0x0000000f;

                        if (ref==12) /** basic block **/
                                nbt+=nb;
                        else
                                nbt=nbt+nb+1;
                }
/* il faut se mettre au debut du prochain bloc de base avant d'enregistrer --> les
valeurs d'entrees sont des approximations */
                while ((ref!=12)&&(read(TUBE[0],&temp,sizeof(int))))
                {
                        cop = temp & 0x00ffffff;
                        nb = (temp>>28) & 0x0000000f;
                        ref= (temp>>24) & 0x0000000f;

                        if (ref==12) /** basic block **/
                                nbt+=nb;
                        else
                                nbt=nbt+nb+1;
                }
/* on enregistre */
                nbt = nb;
                fwrite(&temp,4,1,descr); /* on enregistre le premier bloc */
                while ((nbt<nb_enr)&&(read(TUBE[0],&temp,sizeof(int))))
                {
```

Nom	Numéro	Fonctionnalité
WORD_READ	0	all load instructions
DOUBLE_READ	1	all load doubles
WORD_WRITE	2	store word
DOUBLE_WRITE	3	store double
BYTE_WRITE	4	store byte
HALF_WRITE	5	store half-word
SWR	6	store word right
SWL	7	store word left
LWC1	8	coprocessor 1 load word
LDC1	9	coprocessor 1 load double
SWC1	10	coprocessor 1 store word
SDC1	11	coprocessor 1 store double
BASIC_BLOCK	12	entered a basic block
ANNUL	13	annuled the branch delay slot
SYSTEM_CALL	14	executed a system call
RESERVED	15	

TAB. 1 - : Liste des types générés par *pixie*

```
                cop = temp & 0x00ffffff;
                nb = (temp>>28) & 0x0000000f;
                ref= (temp>>24) & 0x0000000f;
                if ((nbt %500000)<16)
                        printf("Instruction enregistree: %d\n",nbt);

                if (ref==12) /** basic block **/
                        nbt+=nb;
                else
                        nbt=nbt+nb+1;
                fwrite(&temp,4,1,descr);
        }
        printf("OK RECEPTEUR: %d\n",nbt);
        fclose(descr);
        printf("\n");
        exit(0);
    }
}
```

EXE. 1 -: Capture des traces à partir du descripteur 8 dans le fichier sortie.

3.2 Format des traces

Chaque objet de la trace est un mot de 32 bits comportant trois champs: un compteur (4 bits de poids forts), un type (4 bits), et une adresse (24 bits de poids faibles). Le tableau 1 donne les différentes valeurs possibles du champ type.

En ce qui concerne les objets portant le type BASIC_BLOC, le champ adresse donne l'adresse du bloc de base où on va entrer (adresse mot, donc à décaler de deux positions sur la droite). Le champ compteur renseigne sur le nombre d'instructions entre la première instruction de ce bloc et le prochain objet de la

Objet trace (hexa)	Type	Adresse (octet)	Compteur
4 c 10006c	BASIC_BLOC	0x4001b0	4
3 c 100070	BASIC_BLOC	0x4001c0	3
3 c 100070	BASIC_BLOC	0x4001c0	3
3 c 100070	BASIC_BLOC	0x4001c0	3
3 c 100073	BASIC_BLOC	0x4001cc	3
9 c 100088	BASIC_BLOC	0x400220	9

TAB. 2 - : Trace d'instructions du code exemple

trace. Ce dernier objet peut être soit un autre bloc de base, soit une référence à un accès mémoire. Une telle référence porte un type dont le numéro est compris entre 0 et 11. L'adresse mémoire de l'accès est mentionnée dans le champ adresse, en octets. L'espace d'adressage correspond donc à 24 bits, soit 16 *mega-octets*. De plus, un tel objet ne spécifie la présence que d'un accès. Il dépend donc d'objet précédemment typé BASIC_BLOC pour indiquer l'adresse de l'instruction provoquant l'accès. Le champ compteur d'une référence mémoire correspond à la distance en instructions entre l'accès et le prochain objet, moins un. Par exemple, on associera à un chargement directement suivi par un enregistrement, un objet dont le champ compteur est à 0, et non 1.

En raison du codage, les champs compteur sont limités à 4 bits. Par conséquent, un bloc de base ne peut contenir plus de 15 instructions. Dans le cas contraire, le bloc est divisé en plusieurs blocs de taille fixe égale à 15 instructions hormis le dernier dont la taille est de $(taille - bloc\_base) mod 15$.

Les objets typés ANNUL ne sont produits que lors de trace utilisant le jeu d'instructions *mips2*. La version actuelle de *pixie* ne générant pas ce jeu, de tels objets ne seront jamais produits.

L'exemple 1 précédent montre un exemple de gestion des champs compteur : le nombre total d'instructions exécutées est calculé pour chaque objet de la trace dynamique. L'exemple 2 suivant présente un programme *mips1*. Il contient trois blocs de base aux adresses 0x4001b0, 0x4001c0 et 0x4001cc.

```
0x4001b0  addiu  sp,sp,-24
0x4001b4  sw     r31,20(sp)
0x4001b8  lw     r8,28(sp)
0x4001bc  move   r5,r0

0x4001c0  addiu  r5,r5,1
0x4001c4  bne    r5,r4,0x4001c0
0x4001c8  nop

0x4001cc  sw     r8,0(sp)
0x4001d0  jal    0x400220
0x4001d4  move   r4,r8
```

EXE. 2 - : Code exemple pour jeu d'instructions *mips1*

Ce code exemple a été exécuté après pixification avec l'option -itrace. La table 2 relate une trace possible. Le premier bloc de base est le premier visité. La boucle du second bloc est exécuté pour trois itérations, pour ensuite sortir dans le troisième bloc avant d'effectuer un saut. La trace créée ne contient que des objets référencés BASIC_BLOC, avec l'adresse de base en mot et un champ compteur mentionnant le nombre d'instructions. Le premier objet de la trace indique l'entrée dans le premier bloc de base, qui commence à l'adresse 0x4001b0 et possède quatre instructions. Le second objet mentionne l'exécution du second bloc, et ainsi de suite.

La table 3 relate la trace de la même exécution mais cette fois, le code a été pixifié avec l'option -idtrace. On peut noter l'adjonction de nouveaux objets précisant l'exécution d'instructions d'accès à la mémoire. Cet

Objet trace (hexa)	Type	Adresse (octet)	Compteur
1 c 10006c	BASIC_BLOC	0x4001b0	1
0 2 fff754	WORD_WRITE	0xfff754	0
1 0 fff75c	WORD_READ	0xfff75c	1
3 c 100070	BASIC_BLOC	0x4001c0	3
3 c 100070	BASIC_BLOC	0x4001c0	3
3 c 100070	BASIC_BLOC	0x4001c0	3
0 c 100073	BASIC_BLOC	0x4001cc	0
2 2 fff740	WORD_WRITE	0xfff740	2
9 c 100088	BASIC_BLOC	0x400220	9

TAB. 3 - : Trace d'instructions et de données du code exemple

ajout a provoqué une modification des valeurs des champs `compteur` des objets typés BASIC_BLOC. En additionnant les valeurs des champs `compteur` et `adresse` du premier objet typé BASIC_BLOC (0x10006c + 1 = 0x10006d), on obtient l'adresse en mot de l'instruction d'enregistrement (0x10006d mots = 0x4001c0 octets). Cet enregistrement range la valeur de `r31` dans l'adresse mémoire `0xfff754` comme indiqué par l'adresse octet du champ `adresse` du second objet. Le troisième objet correspond à l'instruction de la prochaine adresse séquentielle puisque le champ `compteur` est à 0.

3.3 Génération de traces d'instructions et de données complètes

Le fichier **pgnom.seq**, précédemment créé, permet de récupérer les adresses des instructions séquentiellement exécutées ainsi que des accès mémoire réalisés, ce que nous appelons la trace dynamique. Il peut ainsi être nécessaire de connaître le code opération des instructions, pour dériver par exemple des informations concernant l'utilisation des registres ... La trace statique liste donc l'adresse puis le code opération de chacune des instructions du programme. La commande MIPS `dis` désassemble un programme exécutable et produit donc les informations requises en mode texte: le fichier créé est dénommé **pgnom.dis**. On peut néanmoins profiter du fait que toutes les instructions d'un programme qui n'appelle pas de librairie partagée, sont contigües. Il suffit donc de produire l'adresse de la première instruction suivie de l'ensemble des codes opération. Ceci est le fichier **pgnom.cop**, qui est produit à partir du fichier **pgnom.dis**. L'exemple 3 suivant décrit une telle transformation. Il est à noter que le fichier final n'est plus en mode texte, pour des raisons bien sûr d'économie de place disque.

```
#include <stdio.h>
#include <stdlib.h>

#define Carton_Rouge(a,b,c) { fprintf(stderr,a,b); exit(c); }
#define Carton_Jaune(a,b,c) { fprintf(stderr,a,b,c); }

#define V_LA_LES_FLICS -22


/**    DIStoCOP    **/

void main(argc,argv)
 int argc; char **argv;
{
FILE   *entree;
char   nom_entree[256];
FILE   *sortie;
```

```
char   nom_sortie[256];
char   ligne[256];
int    nb_element;
char   chaine[6][256];
int    codeop,adresse,adresse_precedente;
int    i;
int    dechet = 0;

   if (argc!=2)
      Carton_Rouge("Mauvais usage: %s <nom de fichier sans extension>\n",argv[0],1);
   strcpy(nom_entree,argv[1]);
   strcat(nom_entree,".dis");
   strcpy(nom_sortie,argv[1]);
   strcat(nom_sortie,".cop");

   if ((entree = fopen(nom_entree,"r")) == NULL)
      Carton_Rouge("Impossible d'ouvrir en lecture l'entree : %s\n",nom_entree,2);

   if ((sortie = fopen(nom_sortie,"w")) == NULL)
      Carton_Rouge("Impossible d'ouvrir en ecriture la sortie: %s\n",nom_sortie,3);

   adresse_precedente = V_LA_LES_FLICS;

   while ((fgets(ligne,256,entree)) != NULL)
   {
      nb_element = sscanf(ligne,"%s %s %s %s %s %s",chaine[0],chaine[1],chaine[2],
         chaine[3],chaine[4],chaine[5]);
      i = 0;
      while ((i<nb_element)&&(strncmp(chaine[i],"0x",2)!=0))
         i++;
      if (i!=nb_element)
      { /* on a trouve un code_op */
         adresse = strtol(chaine[i],(char **)NULL,16);
         codeop = strtol(chaine[i+1],(char **)NULL,16);
         if ((adresse!=adresse_precedente+4)&&(adresse_precedente!=V_LA_LES_FLICS))
         {
            while (adresse!=adresse_precedente+4)
            {
               Carton_Jaune("Adresses non consec.: %d %d\n",adresse_precedente,adresse);
               fwrite((char *) &dechet,4,1,sortie);
               adresse_precedente+=4;
            }
         }
         else
         {
            if (adresse_precedente == V_LA_LES_FLICS)
            {
               fwrite((char *) &adresse,4,1,sortie);
               fwrite((char *) &codeop,4,1,sortie);
               adresse_precedente = adresse - 4;
            }
            else
               fwrite((char *) &codeop,4,1,sortie);
```

```
         }
         adresse_precedente = adresse_precedente +4;
      }
   }
   close(entree);
}
```

Exe. 3 - Production de la trace statique

L'exemple 4 suivant présente un ensemble de fonctions qui renvoient un à un les codes opération des instructions exécutées. Le fichier *inst.h* est celui standard fourni par SGI.

```
#include "stdio.h"
#include "inst.h"
#include "cartons.h" /* carton_rouge -> erreur, carton_jaune -> warning,
      carton_vert -> message qcq */

#define TAILLE_PROGRAMME 1000000

int   adresse_courante;  /** adresse du PC **/
int   nombre_instruction; /** nb d'inst avant la prochaine lecture **/
int   offset;  /** adresse de la premiere instruction du pg **/
int   item_courant;
union mips_instruction *PROGRAM;
FILE  *COP;
FILE  *SEQ;

/*--------------------------------------------*/
/* Lit les fichiers .cop et .seq              */
/* Alloue la memoire                          */
/* Renvoie PROGRAM, offset, adresse_courante  */
/*--------------------------------------------*/
void Init(nom)
   char *nom;
{
   char nom_cop[256], nom_seq[256]; int i;

   strcpy(nom_cop, nom);
   strcat(nom_cop, ".cop");
   if ((COP = fopen(nom_cop,"r"))==NULL)
      Carton_Rouge("fichier %s impossible a ouvrir",nom_cop,10);

   strcpy(nom_seq, nom);
   strcat(nom_seq, ".seq");
   if ((SEQ = fopen(nom_seq,"r"))==NULL)
      Carton_Rouge("fichier %s impossible a ouvrir",nom_seq,11);

   if ((fread((char *) &adresse_courante,sizeof(int),1,COP))!=1)
      Carton_Rouge("lecture impossible de l'adresse dans le fichier %s",nom_cop,12);
   offset = adresse_courante;
   if ((PROGRAM =(union mips_instruction *)malloc(TAILLE_PROGRAMME*sizeof(union
```

```
      mips_instruction)))==NULL)
    Carton_Rouge("impossible d'allouer la memoire programme",0,13);
  Carton_Vert("lecture du fichier %s",nom_cop);
  i=0;
  while ((fread((char *) &PROGRAM[i],sizeof(int),1,COP))==1)
    i++;
  Carton_Vert("%d instructions",i);
  fclose(COP);
}

/*---------------------------------------------*/
/* Histoire de terminer proprement             */
/* on ferme les fichiers et on libere la memoire*/
/*---------------------------------------------*/
void Exit()
{
  free(PROGRAM);
  fclose(SEQ);
}

/*---------------------------------------------*/
/* Lecture du bloc d'instructions courant */
/*---------------------------------------------*/
bool Lecture(cop,add,longueur) /** retourne faux si fin du programme **/
  union mips_instruction *cop; /** code op **/
  int *add; /** adresse load_store **/
  int *longueur; /** longueur load_store en octet**/
{
  int reftype;
  bool booleen;

  if (nombre_instruction != 0)
  {
    cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
    adresse_courante +=4;
    nombre_instruction--;
    return(vrai);
  }
  else
  { /** recherche d'un nouvel item **/
    if ((fread((char *) &item_courant,sizeof(int),1,SEQ))!=1)
      return(faux); /** FIN DU PROGRAMME **/
    *add = item_courant & 0x00ffffff;
    nombre_instruction = (item_courant>>28)&0x0000000f;
    reftype = (item_courant>>24)&0x0000000f;

    switch(reftype)
    {
    case 4: /** store byte **/
      cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
      adresse_courante +=4;
      *longueur = 1;
      return(vrai);
```

```
   break;
case 5: /** store half word **/
   cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
   adresse_courante +=4;
   *longueur = 2;
   return(vrai);
   break;
case 0: /** all load instructions **/
case 2: /** store word **/
case 6: /** store word right **/
case 7: /** store word left **/
case 8: /** coprocessor 1 load word **/
case 10: /** coprocessor 1 store word **/
   cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
   adresse_courante +=4;
   *longueur = 4;
   return(vrai);
   break;
case 1: /** all load doubles **/
case 3: /** store double **/
case 9: /** coprocessor 1 load double **/
case 11: /** coprocessor 1 store double **/
   cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
   adresse_courante +=4;
   *longueur = 8;
   return(vrai);
   break;
case 12: /** entered a basic block **/
   if (nombre_instruction!=0)
   {
      adresse_courante = (*add)<<2;
      cop->word = PROGRAM [(adresse_courante-offset)>>2].word;
      adresse_courante +=4;
      nombre_instruction--;
      return(vrai);
   }
   else /** load en premier, bb sert a donner juste la nouvelle adresse **/
   {
      adresse_courante = (*add)<<2;
      booleen = Lecture(cop,add,longueur); /** appel recursif **/
      return(booleen);
   }
   break;
case 13: /** annuled the branch delay slot **/
   Carton_Rouge("Bccl is a mips2 instruction",0,20);
   break;
case 15: /** unimplemented **/
   Carton_Rouge("numero de reftype 15 non implemente",0,21);
   break;
case 14: /** syscall call **/
   booleen = Lecture(cop,add,longueur); /** appel recursif **/
   return(booleen);
   break;
```

Nom	Description
008.espresso	Génère et optimise des PALs
022.li	Utilise un interpréteur LISP pour résoudre le problème des 8 reines
023.eqntott	Traduit une représentation logique d'une équation booléenne en table de vérité
026.compress	Réduit la taille d'un fichier par codage de Lempel-Ziv
072.sc	Calcul de budgets dans un tableur basé sur le paquetage CURSES
085.gcc	Traduit des fichiers source C en assembleur SUN 3 optimisés

TAB. 4 - : Paquetage CINT92

```
      default:
         Carton_Rouge("I think a big problem occurs! NO REFTYPE %d !",reftype,22);
         break;
      }
   }
}
```

EXE. 4 - Traitement des traces et renvoi des codes-op

4 Génération de traces à partir des SPEC92

4.1 Présentation des SPEC92

La première version du paquetage SPEC fut commercialisée en 1989. Elle contenait dix programmes test : quatre programmes *entier* et six *flottant*. La métrique ne faisait par contre aucune distinction.

En 1992, la seconde et présente version fut introduite. Six programmes *entier* (ensemble **CINT92**) sont présents [2] ainsi que 14 programmes *flottant* (ensemble **CFP92**) [3]. Les métriques existent pour les deux types de programmes, et sont de plus basées comme nous le verrons à la section suivante, sur la moyenne géométrique des résultats individuels.

Les CINT92 et CFP92 consistent donc en des ensembles de nouveaux programmes ou de versions mises à jour des SPEC89. Les facteurs considérés lors du choix des programmes incluent la classe d'application, le langage de programmation, les données manipulées, l'utilisation de fichier d'entrée ainsi que la distribution des instructions. De manière identique aux SPEC89, ce paquetage se focalise sur le calcul intensif. Il ne teste pas la capacité d'un système à gérer les entrées/sorties, à manipuler intensivement du graphique ou à communiquer via des réseaux. La plupart des programmes du paquetage SPEC92 dérivent d'applications du domaine public, bien que modifiées afin de ne favoriser aucun système en particulier. Les tables 4 et 5 décrivent succintement ces programmes. Une nouvelle version du paquetage étiquetée SPEC95 devrait être lancée pour la fin de l'année 1995.

4.2 Caractéristiques des traces générées

L'ensemble des programmes que nous venons de présenter a été compilé avec les *makefiles* standards fournis par SPEC (optimisation maximale) pour le jeu d'instruction du Mips R4000 [4]. Néanmoins les limitations dues à *pixie* ont été respectées et donc les drapeaux de compilation -mips1 -non_shared ont été utilisés. Les jeux d'entrée utilisés sont les minimums, ou des versions légèrement modifiées afin que l'exécution reste courte. **Par contre, les sources des programmes n'ont jamais subi de modification**. Nous avons utilisé la méthode précédemment décrite pour récupérer la trace de l'exécution de chaque programme (les fichiers **pgnom.seq** et **pgnom.cop**). Il est à noter que les NOPs ont été écartés des traces. De plus, l'utilisation de petits jeux d'entrée nous a permis de tracer **les exécutions complètes des programmes**

Nom	Description
013.spice2g6	Simulation de circuits analogiques
015.doduc	Simulation de Monte-Carlo de l'évolution du temps d'un modèle thermo-hydraulique pour un composant d'un réacteur nucléaire
034.mdljdp2	Résolution d'équations de mouvement dans un modèle de 500 atomes en interaction en utilisant le potentiel idéal de Lennard-Jones (double précision)
039.wave5	Résolution d'équations de Maxwell pour des particules sur une grille cartésienne
047.tomcatv	Génère un système de coordonnées 2D maillé autour d'un domaine géométrique général
048.ora	Lancer de rayons à travers une surface optique contenant des surfaces sphériques et planes
052.alvinn	Exerce un réseau de neurones par rétropropagation
056.ear	Simulation de l'oreille humaine par conversion d'un fichier de son en cochléogramme en utilisant des FFTs et d'autres fonctions mathématiques
077.mdljsp2	Similaire à 034.mdljdp2 (simple précision)
078.swm256	Résolution d'équations d'écoulement superficiel par approximation des différences finies
089.su2cor	Calcul de masse élémentaire de particules dans le cadre de la théorie de Quark Gluon
090.hydro2d	Calcul des flux galactiques par résolution de l'équation hydrodynamique de Navier-Stockes
093.nasa7	Exécution de sept noyaux exécutés fréquemment dans les applications de la NASA
094.fpppp	Calcul de dérivées intégrales de plusieurs électrons

Tab. 5 - : Paquetage CFP92

tout en gardant une taille raisonnable des traces (à l'exception de *backprop,dnasa7,wave5* et *spice2g6*). L'ensemble des traces représente tout de même un peu moins de 600 millions d'instructions.

Afin de classifier les instructions, nous avons défini la taxonomie suivante en accord avec le jeu d'instructions du MC 88110 [5], la table 6 présentant les distributions des programmes suivant cette taxonomie :

- RU — saut ou tout autre instruction de rupture non conditionnel
- CO — branchement conditionnel
- AR — instruction arithmétique entière
- LO — instruction de calcul logique
- DE — instruction de décalage
- MU — instruction de multiplication entière
- DI — instruction de division entière
- CH — instruction de chargement mémoire
- EN — instruction d'enregistrement mémoire
- FA — instruction arithmétique flottante
- FM — instruction de multiplication flottante
- FD — instruction de division flottante
- FC — instruction de conversion

	nb insts	RU	CO	AR	LO	DE	MU	DI	CH	EN	FA	FM	FD	FC
cc1	27041587	2,93	18,31	30,21	6,71	3,33	0,13	0,51	24,81	13,54	0	0	0	0
compress	6919540	0,89	15,77	27,33	10,93	10,23	0	0	22,42	12,44	0	0	0	0
eqntott	8054151	2,14	22,39	27,04	12,35	0,7	0,35	0,01	25,25	9,78	0	0	0	0
espresso	26513708	2,63	16,07	28,04	15,92	8,41	0,07	0	21,9	6,95	0	0	0	0
sc	36765301	1,21	21,33	31,5	5,77	4,11	0,04	0	25,61	10,4	0	0	0	0,01
xlisp	11329121	5,63	18,04	16,01	9,89	0,76	0	0	30,5	19,16	0	0	0	0
CINT92	116623408	2,57	18,65	26,69	10,26	4,59	0,1	0,09	25,08	12,05	0	0	0	0
backprop	39988855	0,02	9,81	29,88	0,04	0,04	0	0	29,43	10,11	10,01	10,17	0,01	0,48
dnasa7	36921200	0,56	3,66	17,53	0,34	0,5	0	0	48,75	6,31	10,68	10,68	0,11	0,89
doduc	12382313	1,1	8,05	21,45	3,11	1,72	0,03	0,02	28,48	9,81	9,65	8,79	1,72	6,07
ear	26324009	1,31	9,12	25,43	4,26	3,13	0	0,07	18,16	9,62	9,91	7,43	0,64	10,93
fpppp	11520298	0,19	1,31	3,46	0,51	0,45	0,03	0,02	48,6	15,65	13,51	15,12	0,25	0,9
hydro2d	7322720	1,12	9,98	21,71	2,99	3,19	0,08	0,03	29,4	12,17	6,1	5,22	1,19	6,8
mdljdp2	44100676	0,04	8,88	5,92	0,29	0,81	0	0	37,87	13,93	14,19	9,3	0,64	8,12
mdljsp2	33855068	0,05	11,63	16,22	0,38	1,03	0	0	20,37	8,34	18,54	12,03	0,84	10,58
ora	19499394	1,52	8,29	23,13	6,29	4,19	0	0	21,21	7,22	9,07	6,34	3,7	9,03
spice2g6	47521520	2,06	15,52	32,64	2,84	11,4	0,45	0,02	24,89	8,99	0,41	0,04	0,01	0,74
su2cor	45032283	0,23	2,56	15,85	0,97	2,47	0,01	0	42,89	12,48	9,23	11,29	0,59	1,41
swm256	22530485	2,37	4,02	12,5	2,28	0,02	0	0	25,78	4,21	20,87	16,08	0,87	11
tomcatv	38708920	0,01	2,18	23,95	0,09	0,03	0	0	35,35	13,53	15,12	8,73	0,34	0,67
wave5	48754639	2,97	7,54	23,5	0,62	4,08	0,63	0,32	29,9	10,58	5,51	4,72	1,45	8,19
CFP92	434462380	0,97	7,33	19,51	1,79	2,36	0,09	0,03	31,51	10,21	10,91	9	0,9	5,4

TAB. 6 - : Distribution suivant les classes d'instructions

5 Calcul de moyennes

La présence de plusieurs programmes test entraîne le problème du choix de la moyenne : on voudrait associer une *performance moyenne* à une configuration lors d'une évaluation. Cette performance moyenne dépend des performances individuelles sur chacun des programmes test. Nous relatons dans cette section des travaux publiés dans [6] qui concluent que la moyenne géométrique est la seule moyenne correcte pour ce cas de figure. En effet, il va sans dire que le calcul de distributions, tailles et autres valeurs de ce type, est effectué grâce à la moyenne arithmétique standard.

5.1 Calcul d'accélérations

Soient P_1 et P_2 les performances mesurées pour deux configurations différentes (*en seconde par exemple*). L'accélération amenée par la seconde configuration sur la première est alors définie comme étant :

$$S = \frac{P_1}{P_2} \tag{1}$$

Le problème vient alors de la présence de plusieurs programmes test (on supposera leur nombre égal à n). Il est donc nécessaire de calculer une *accélération moyenne*, A, en fonction des accélérations individuelles définies par l'équation 1 précédente :

$$A = F(s_1, s_2, s_3, \ldots, s_n) \tag{2}$$

Les conditions que doit respecter la fonction F sont les suivantes :

Condition 1 $F(x, \ldots, x) = x$

Condition 2 $F(x_1, \ldots, x_k) \cdot F(y_1, \ldots, y_k) = F(x_1 y_1, \ldots, x_k y_k)$

Condition 3 *Pour tout vecteur $\vec{x}$ de dimension k et toute permutation π de $\{1, \ldots, k\}$*

$$F(x_1, \ldots, x_k) = F\big(x_{\pi(1)}, \ldots, x_{\pi(k)}\big)$$

La condition 1 est évidente. Pour comprendre la condition 2, imaginons trois configurations a, b et c. Si $A(a, b)$ désigne l'accélération moyenne de la configuration b vis à vis de la configuration a, il est alors nécessaire que $A(a, c) = A(a, b) \cdot A(b, c)$. La condition 3 assure que l'ordre des x_i n'affecte pas l'accélération.

Lemme 1 *L'unique fonction $F : \mathbb{R}_+^k \rightarrow \mathbb{R}_+$ qui remplit les conditions A1, A2 et A3 est la moyenne géométrique*

$$F(x_1, \ldots, x_k) = \sqrt[k]{x_1 \cdot \ldots \cdot x_k}$$

Démonstration : Il est facile de voir que la moyenne géométrique remplit les conditions A1, A2 et A3. La suite d'équations suivante est vérifiée pour toute solution f de A1, A2 et A3 :

$$\begin{aligned} f(x_1,\ldots,x_k)^k &= f(x_1,\ldots,x_k)\cdot f(x_2,\ldots,x_k,x_1)\cdot\ldots\cdot f(x_k,x_1,\ldots,x_{k-1}) \\ &= f(x_1\cdot\ldots\cdot x_k,\ldots,x_1\cdot\ldots\cdot x_k) \\ &= x_1\cdot\ldots\cdot x_k \end{aligned}$$

Pour la première équation, nous avons utilisé A3 pour écrire $f(x_1,\ldots,x_k)^k$ comme le produit de f appliqué à toutes les permutations cycliques de $x_1,\ldots,x_k$. La seconde équation est l'application de A2 $k-1$ fois. La dernière ligne est l'application de la réflexivité (A1). On en déduit le lemme, puisque f est une fonction strictement positive. $\square$.

5.2 Moyennes des programmes test

L'accélération moyenne entre deux configurations a et b est donc la moyenne géométrique des accélérations individuelles obtenues pour chacun des programmes test :

$$A(a,b) = F(s_1,\ldots,s_k) = \sqrt[k]{\frac{P_{a1}}{P_{b1}}\cdot\ldots\cdot\frac{P_{ak}}{P_{bk}}} = \frac{F(P_{a1},\ldots,P_{ak})}{F(P_{b1},\ldots,P_{bk})} \tag{3}$$

Si nous considérons qu'il existe un *programme test moyen*, d'après les équations 1 et 3, la performance de ce programme pour une configuration donnée correspondrait à la moyenne géométrique[2] des performances obtenues individuellement sur chacun des programmes test (avec un coefficient multiplicatif que nous considèrerons trivialement égal à 1).

Enfin, il est à noter que les mesures des performances peuvent être effectuées en *cycles par instruction* puisque le nombre d'instruction est un invariant pour chaque programme. Une mesure en *instructions par cycle* nécessite l'inversion du résultat final.

6 Application : Impact des dépendances de données

Hennessy et Patterson [7] ont bien montré l'utilité de l'analyse du code dans le processus de conception d'un microprocesseur. Toutefois, la tendance architecturale actuelle consiste à exécuter simultanément plusieurs instructions essentiellement grâce au modèle superscalaire mais aussi avec les modèles VLIW, découplé ou *data-flow*. Si l'on a pu se contenter d'analyses dynamiques de fréquence des instructions pour la conception des microprocesseurs RISC de première génération, la conception de nouvelles architectures nécessite aussi une étude des dépendances des programmes. Aujourd'hui, les programmes les plus fréquemment utilisés pour l'évaluation d'architectures de processeurs sont extraits du paquetage SPEC92 qui a été décrit précédemment. Cette section présente donc d'une part la distribution des instructions mais aussi l'incidence des dépendances de données sur le parallélisme auquel on peut prétendre dans ces programmes test.

On rappelera donc tout d'abord les types de dépendances existants puis les hypothèses faites et les moyens disponibles pour minimiser l'impact de chaque type de dépendance. Nous présenterons ensuite la méthode utilisée pour cette étude et, enfin, les résultats obtenus.

6.1 Les dépendances

Lors de l'exécution d'un programme apparaissent toutes sortes de dépendances influant plus ou moins sur le parallélisme du programme. Le modèle de programmation considéré ici est celui de Von Neumann étendu par les concepts associés aux processeurs RISC. Ainsi, la sémantique du jeu d'instructions spécifie uniquement trois formats qui sont :

- $R \times R \rightarrow R$

[2] Le calcul de la performance moyenne effectué dans les SPEC92 correspond aussi à la moyenne géométrique des résultats individuels

- $R \times M \rightarrow R(load)$
- $R \times R \rightarrow M(store)$

où R est une mémoire de faible capacité (les registres) et M la mémoire centrale. R et M constituent les uniques emplacements de mémorisation.

6.1.1 Dépendances vraies

Il existe une relation de dépendance vraie entre deux instructions lorsqu'une instruction produit un résultat consommé par l'autre. Cela se présente donc lorsqu'une instruction utilise un emplacement de mémorisation précédemment modifié par une instruction antérieure. Ce type de dépendance impose donc un ordre total entre les opérations.

6.1.2 Anti-dépendances

L'anti-dépendance est une dépendance qui est due à des possibilités de mémorisation limitées. En effet, cette dépendance apparaît lorsqu'une écriture suit une lecture au même emplacement de mémorisation. Une synchronisation est alors nécessaire sinon la lecture peut être faite sur une mauvaise donnée.

6.1.3 Dépendances de sortie

La dépendance de sortie est du même genre que la précédente mais elle intervient entre deux écritures dans le même emplacement. Ces deux écritures doivent être ordonnées afin que la donnée dans l'emplacement de mémorisation soit correcte.

6.1.4 Dépendances de contrôle

La dépendance de contrôle se présente lorsqu'on ne sait pas quelle instruction va être exécutée tant qu'une instruction précédente n'est pas terminée. Il y a alors rupture du flot d'instructions vers une adresse connue dynamiquement. Ces dépendances sont dues aux instructions de branchement conditionnelles.

6.1.5 Dépendances de ressources

Une dépendance de ressource (également appelée *aléa structurel*) apparaît lorsque l'exécution d'une instruction doit être retardée parce qu'une ressource physique n'est pas disponible. Cette ressource peut être, par exemple, une unité fonctionnelle, la fenêtre d'instruction ou la mémoire.

Les trois premiers types de dépendance sont regroupés dans ce que l'on appelle les dépendances de données. Toutefois, les anti-dépendances et les dépendances de sortie peuvent être considérées comme des dépendances de ressources puisqu'elles sont dues à un espace de mémorisation limité qui bloque la mise à feu d'instructions.

6.1.6 Gestion des dépendances

Minimiser, voire supprimer les dépendances implique différentes méthodes suivant le type de dépendance.

Bien entendu, il n'est pas possible de supprimer les dépendances vraies car elles sont implicites au modèle de programmation. Le seul moyen de minimiser leur impact est de diminuer la latence des opérations qui produisent ces dépendances.

Les anti-dépendances et les dépendances de sortie peuvent être supprimées par un renommage des emplacements de mémorisation, que ce soit les registres ou la mémoire. Ce renommage consiste en une association des emplacements précisés par le modèle de programmation vers des emplacements physiques mis en œuvre dans le modèle d'exécution. Néanmoins, on ne cherche pas ici à définir des éléments d'architecture comme le nombre de registres nécessaires, on se limitera donc à envisager les cas sans renommage ou alors avec un renommage vers un nombre de registres physiques infini.

La solution architecturale pour minimiser l'impact des dépendances de contrôle est un mécanisme de prédiction de branchement. [8] montre bien l'influence de ces mécanismes sur le parallélisme disponible de quelques programmes test dont certains sont extraits du paquetage SPEC89.

Enfin, pour les dépendances de ressources, leur nombre dépend de l'architecture considérée. Par exemple, nous rapportons dans [9] l'influence du nombre des unités fonctionnelles dans un processeur superscalaire à initiation non ordonnée.

6.2 Le simulateur

Le simulateur utilisé est présenté dans l'exemple 5. Il utilise les routines de l'exemple 4 ainsi qu'une fonction de décodage non présentement documentée. Les options de compilation concernent les choix de renommage, mais aussi celui des latences. Cette dernière option est codée dans la fonction de décodage et permet de choisir soit des latences idéales (toutes égale à 1), soit celles du PowerPC 604 [10] qui sont :

- 1 cycle : RU, CO, AR, LO, DE, FC
- 2 cycles : CH
- 3 cycles : MU, EN, FA, FM
- 18 cycles : FD simple précision
- 20 cycles : DI
- 31 cycles : FD double précision

Le source du simulateur est le suivant :

```
/*
 * Temps d'execution du programme sans aucune contrainte sur l'architecture
 * du processeur.
 * On suppose que la taille de la fenetre d'instructions est infinie.
 * On ne tient pas compte des exceptions.
 * Il n'y a aucune dependance de ressource, seules comptent les
 * dependances de donnees.
 * Renommage de registres et de la memoire.
 * Par consequent, il s'agit de data flow. Des que les operandes
 * d'une instruction sont prets, celle-ci calcule son resultat et le
 * fournit n cycles plus tard (n etant la latence de l'instruction).
 *
 * OPTION:
 *
 * - PESS
 *   Lorsqu'on a un syscall : appel systeme, la variable num_cycle_barrier
 *   est initialisee au cycle maximum . Ceci est donc une barriere et les instructions
 *   anterieures ne pourront etre emises avant la barriere
 *
 * - SRR
 *   sans renommage de registres
 *
 * - SRM
 *   sans renommage de memoire
 *
 * TEMPS CALCUL ADDRESSE MEMOIRE (base+depl) = 1 cycle
 */
```

```
typedef struct  {
        union mips_instruction  inst;
        uf      type;           /* classe de l'instruction */
        int     source1;        /* registres sources */
        int     source2;
        int     destination;    /* registre destination */
        int     latence;        /* duree de l'execution */
        int     adresse;        /* adresse de base de l'acces a la memoire */
        int     longueur; /* taille en octet de l'acces memoire */
} inst_dec;

#define MIN(a,b) (a<b) ? a : b ;
#define MAX(a,b) (a>b) ? a : b ;

#define reg_HI 32
#define reg_LO 33
#define reg_HILO 34

/*
 * Note sur l'adressage des registres:
 *      registre:       0-31    registres entiers
 *                      32      HI
 *                      33      LO
 *                      34      HI&LO
 *                      35      PC
 *                      36-67   registres COP1
 *                      68-99   registres COP0
 *                      100     Condition COP1
 *
 */

#ifdef PESS
   int num_cycle_barrier = 0;
#endif

inst_dec instruction_courante;

int *MEMORY; /* tableau initialise a 0, la taille etant l'espace d'adressage en octet,
                il contient le cycle ou l'octet adresse sera disponible */
int REGISTER[101];
int cycle = 0;

/*
 * Lecture et decodage de l'instruction suivante
 */

bool Instruction(in)
   inst_dec *in;
{
   bool booleen;

   booleen = Lecture(&in->inst,&in->adresse,&in->longueur);
   while ((booleen==vrai)&&(IS_NOOP(in->inst)==vrai))
```

```
   {
      booleen = Lecture(&in->inst,&in->adresse,&in->longueur);
   }
   if (booleen==vrai)
      Decode(in);
   return(booleen);
}

void main(argc,argv)
   int argc;
   char **argv;
{
   int i;
   bool est_une_instruction;
   int inst = 0;
   int a,b,c;

   if (argc!=2)
      Carton_Rouge("nombre inexact d'arguments: %d",argc,1);

#ifndef PESS
   printf("OPTIMISTE");
#else
   printf("PESSIMISTE");
#endif

#ifdef L604
   printf(" - LATENCE 604");
#else
   printf(" - LATENCE IDEAL");
#endif

#ifdef SRR
   printf(" - ss renommage registre");
#endif

#ifdef SRM
   printf(" - ss renommage memoire");
#endif
   printf("\n");

   Init(argv[1]);
   for (i=0;i<101;i++)
      REGISTER[i] = 0;
   est_une_instruction = Instruction(&instruction_courante);
   while (est_une_instruction == vrai)
   {
      /* Rapport de progression */
      if ((inst%5000000)==0)
      {
         printf("INSTRUCTION: %d\t",inst);
         printf ("Resultat: cycle: %d instruction: %d, moyenne: %f IPC\n",
            cycle,inst,(float)inst/(float)cycle);
```

```
      }
      switch(instruction_courante.type)
      {
      case enreg: /* enregistrement */
         a = REGISTER[instruction_courante.source1];
#ifdef PESS
         a = MAX(a,num_cycle_barrier);
#endif
         b = REGISTER[instruction_courante.source2];
         c = MAX(a,b);
         /* dependance de sortie */
#ifdef SRM
         c = MAX(c,MEMORY[instruction_courante.adresse]);
         for (i=1;i<instruction_courante.longueur;i++)
            c = MAX (c,MEMORY[instruction_courante.adresse+i]);
#endif
         /* a : cycle ou on connait la base et le deplacement
          * c : cycle ou on peut lancer l'execution
          */
         if ((instruction_courante.latence > 1)&&(a<c))
            instruction_courante.latence--; /* recouvrement du calcul de l'adresse avec
             l'acces a la memoire fait par l'instruction STORE precedente(meme adresse)*/
         for (i=0;i<instruction_courante.longueur;i++)
            MEMORY[instruction_courante.adresse+i]=c+instruction_courante.latence;
         /* antidependance */
#ifdef SRR
         REGISTER[instruction_courante.source1]=c;
         REGISTER[instruction_courante.source2]=c;
#endif
         break;
      case charg: /* chargement */
         a = REGISTER[instruction_courante.source1];
#ifdef PESS
         a = MAX(a,num_cycle_barrier);
#endif
         b = MEMORY[instruction_courante.adresse];
         for (i=1;i<instruction_courante.longueur;i++)
            b = MAX (b,MEMORY[instruction_courante.adresse+i]);
         c = MAX(a,b);
         /* dependance de sortie */
#ifdef SRR
         c = MAX(c,REGISTER[instruction_courante.destination]);
#endif
         /* a : cycle ou on connait la base et le deplacement
          * c : cycle ou on peut lancer l'execution
          */
         if ((instruction_courante.latence > 1)&&(a<c))
            instruction_courante.latence--; /* recouvrement du calcul de l'adresse avec
             l'acces a la memoire fait par l'instruction STORE precedente(meme adresse)*/
         if (instruction_courante.destination == reg_HILO)
         {
            REGISTER[reg_HI]=c+instruction_courante.latence;
            REGISTER[reg_LO]=c+instruction_courante.latence;
```

```
         }
         else if (instruction_courante.destination != 0)
            REGISTER[instruction_courante.destination]=c+instruction_courante.latence;
         /** antidependance **/
#ifdef SRR
         REGISTER[instruction_courante.source1]=c;
#endif
#ifdef SRM
         for (i=0;i<instruction_courante.longueur;i++)
            MEMORY[instruction_courante.adresse+i]=c; /* un subsequent store peut se
               faire dans le meme cycle */
#endif
         break;
      default: /* tout autre operation */
         a = REGISTER[instruction_courante.source1];
         b = REGISTER[instruction_courante.source2];
         c = MAX(a,b);
#ifdef PESS
         c = MAX(c,num_cycle_barrier);
#endif
         /* dependance de sortie */
#ifdef SRR
         c = MAX(c,REGISTER[instruction_courante.destination]);
#endif
         if (instruction_courante.destination == reg_HILO)
         {
            REGISTER[reg_HI]=c+instruction_courante.latence;
            REGISTER[reg_LO]=c+instruction_courante.latence;
         }
            else if (instruction_courante.destination != 0) /* si dest nulle,
               pas de destination */
         REGISTER[instruction_courante.destination]=c+instruction_courante.latence;
         /* antidependance */
#ifdef SRR
         REGISTER[instruction_courante.source1]=c;
         REGISTER[instruction_courante.source2]=c;
#endif
      }
      cycle = MAX(cycle,c+instruction_courante.latence);
      inst++;
      est_une_instruction = Instruction(&instruction_courante);
   }
   printf ("Resultat: cycle: %d instruction: %d, moyenne: %f IPC\n",
      cycle,inst,(float)inst/(float)cycle);
   Exit();
}
```

Exe. 5 - Simulateur

	Latences idéales					Latences du PowerPC 604				
	OPT-RM	PES-RM	PES-M	PES-R	PES	OPT-RM	PES-RM	PES-M	PES-R	PES
cc1	55,34	39,76	4,04	32,94	4,03	55,20	31,62	3,11	24,69	3,10
compress	23,06	22,99	3,58	22,99	3,58	17,30	17,25	2,83	17,25	2,83
eqntott	71,19	68,66	4,58	68,37	4,58	51,42	49,03	3,65	48,23	3,65
espresso	62,96	61,62	2,86	51,04	2,85	62,96	61,01	2,32	46,95	2,31
sc	114,82	92,22	4,37	24,41	4,37	114,82	89,67	3,50	19,60	3,50
xlisp	348,97	97,16	4,60	46,77	4,60	317,87	76,55	3,21	35,35	3,21
CINT92	78,23	57,10	3,95	38,01	3,95	69,51	47,30	3,07	29,57	3,06
%	100	72,99	5,04	48,59	5,04	88,86	60,46	3,93	37,79	3,92
backprop	837,11	332,75	3,42	137,18	3,42	661,43	294,79	1,46	82,88	1,46
dnasa7	179,64	178,49	8,20	178,35	8,20	89,91	89,57	6,04	89,48	6,04
doduc	98,63	86,30	4,37	48,42	4,37	19,87	18,98	1,20	11,92	1,20
ear	126,57	99,28	5,10	33,04	5,10	60,96	50,57	2,61	17,52	2,61
fpppp	250,35	243,74	4,45	67,29	4,45	59,89	59,30	1,66	22,40	1,66
hydro2d	149,49	85,58	5,55	72,29	5,55	135,26	50,63	2,20	17,70	2,20
mdljdp2	53,91	53,76	4,68	51,03	4,68	26,94	26,88	1,46	25,58	1,46
mdljsp2	195,95	191,55	3,59	39,07	3,59	91,66	90,19	1,30	19,58	1,30
ora	1316,19	1261,36	4,08	7,63	4,08	579,24	560,84	0,68	0,75	0,68
spice2g6	239,46	81,83	2,64	17,89	2,64	233,43	69,39	2,09	13,40	2,09
su2cor	288,46	266,46	6,22	192,38	6,22	78,09	76,09	2,47	49,04	2,47
swm256	309,60	280,20	5,46	279,99	5,46	105,03	100,65	2,56	100,55	2,56
tomcatv	97,62	97,32	10,3	97,19	10,3	32,28	32,24	5,92	32,24	5,92
wave5	35,27	35,24	3,39	35,24	3,39	9,62	9,62	1,33	9,62	1,33
CFP92	188,58	149,89	4,79	60,86	4,79	80,69	63,42	1,97	21,58	1,97
%	100	79,48	2,54	32,27	2,54	42,79	33,63	1,05	11,44	1,05

TAB. 7 - : Parallélismes disponibles

6.3 Résultats et analyses

La table 7 donne les résultats des simulations. Ces simulations portent d'une part sur le renommage des emplacements de mémorisation et d'autre part sur l'influence des temps de latence. Dans le cas idéal, toutes les latences sont considérées égales à 1. Sinon, nous utilisons celles du PowerPC 604, un des seuls processeurs à initiation non ordonnée à ce jour.

La table 7 montre le parallélisme offert par les programmes test sous diverses hypothèses restrictives. Néanmoins, bien que traçant les appels de bibliothèque (*libc, libm...*), *pixie* ne rapporte pas les instructions exécutées lors d'un appel système : il ne fait qu'indiquer l'occurence de ces appels. L'hypothèse OPT correspond à une configuration où l'on suppose que les appels systèmes ne modifient aucune donnée. L'hypothèse PES se situe dans le cas diamétralement opposé où l'on suppose que tout appel système utilise toutes les données connues au moment de l'appel : on oblige donc toute instruction postérieure qui utilise une donnée (la grande majorité des instructions) à être exécutée après l'appel. Un concept imagé serait la mise en place d'une barrière à chaque appel. Bien entendu, ces deux solutions sont des cas extrêmes : le parallélisme véritablement disponible est quelque part entre les deux. L'hypothèse R signifie que le renommage des registres est effectif en supposant un nombre infini de registres physiques. L'hypothèse M est la même mais pour la mémoire. Enfin, les vingt programmes doivent être groupés en deux sous-ensembles suivant les données qu'ils manipulent. Le premier, comprenant les six premiers programmes, travaille sur des données entières (*CINT92*), alors que l'autre manipule principalement des données flottantes (*CFP92*).

Nous analysons dans un premier temps les CINT92 puis les CFP92.

On peut à première vue remarquer des différences assez importantes en ce qui concerne le parallélisme maximal disponible des programmes CINT92 (de 20 à 350 *instructions par cycle*). La moyenne géométrique de ce parallélisme se situe quelque part entre 70 et 80 suivant l'influence des appels systèmes. Le parallélisme moyen disponible sur les CINT92 est donc assez faible et l'influence de ces appels également. Naturellement, certaines applications comme *xlisp* ont de nombreux appels systèmes et souffrent grandement de l'hypothèse pessimiste. L'impact des renommages (anti-dépendances et dépendances de sortie) est très important puisqu'en leur absence, le parallélisme baisse de plus de 93 % en moyenne. L'absence de renommage des registres est en grande partie responsable de cette perte puisque la suppression du renommage mémoire ne diminue le parallélisme en moyenne que de 33 %. Là aussi, il y a des disparités puisque cette perte est très faible pour quatre des six programmes test, alors que pour *sc*, elle s'élève à presque 75 %. Par contre, le renommage de la mémoire à lui seul n'offre vraiment aucun gain ; ceci est essentiellement dû au fait que les instructions d'accès à la mémoire utilisent toujours des registres (*pour l'adresse et la donnée*). De plus, la perte de parallélisme occasionnée par des latences plus grandes, en l'occurence celles du PowerPC 604, est proche de 20 %. On aurait pu s'attendre à une distribution équitable des instructions sur le chemin critique suivant les chiffres donnés dans la table 6. La perte aurait alors été de 40 %. La différence entre ces deux chiffres laisse donc supposer l'écrasante majorité des instructions de latence 1 sur le chemin critique : les accès mémoire semblent donc minoritaires sur ce chemin. Néanmoins, il faut savoir que le premier cycle de l'exécution de ces instructions d'accès à la mémoire est un calcul d'adresse qui peut être totalement recouvert si l'adresse est connue assez tôt : la latence de ces instructions est alors décrémentée de 1. Ces instructions peuvent donc être présentes, dans une certaine mesure, dans le chemin critique. Enfin, on remarque que si on raisonne en proportions, l'impact des anti-dépendances et des dépendances de sortie est relativement indépendant des latences considérées.

En ce qui concerne les CFP92, beaucoup plus nombreux, le parallélisme moyen est nettement plus important puisque se situant dans la fourchette 150-190 et le parallélisme maximal varie de 30 à 1300 ! Il va de soi que des programmes travaillant sur des matrices présentent un parallélisme potentiellement important. Rappelons que l'on a pris les jeux d'entrée minimaux, le parallélisme peut donc être encore plus important. De plus, l'influence du renommage est plus grande pour cet ensemble de programmes. En effet, le parallélisme obtenu en absence de renommage est du même ordre de grandeur, voire inférieur, que pour les CINT92. *tomcatv* fait néanmoins figure d'exception puisqu'il conserve un parallélisme de 10 en l'absence de tout renommage. Ceci vient du fait qu'il travaille sur de grands tableaux en mémoire sur lequel des opérations indépendantes sont effectuées. En ce qui concerne la perte du renommage mémoire seul, on a, dans l'ensemble, les mêmes résultats à l'exception notable de *ora* dont le parallélisme chute drastiquement. Lui aussi travaille sur de grands tableaux mais les opérations sont dépendantes. L'influence des latences se fait plus ressentir dans le cas des CFP92. En effet, la perte de parallélisme induite par la mise en œuvre des latences du PowerPC 604, est de l'ordre de 60 %. Contrairement aux CINT92, on a ici une distribution à peu près équitable des classes d'instructions sur le chemin critique, puisque la latence moyenne suivant les distributions de la table 6 est de 2,1.

Références

[1] M. D. Smith, "Tracing with Pixie," Rapport technique, Stanford University, 1991.

[2] "SPEC CINT92 Release v1.1 : Technical Manual," Rapport technique, SPEC, Dec. 1992.

[3] "SPEC CFP92 Release v1.1 : Technical Manual," Rapport technique, SPEC, Dec. 1992.

[4] J. Heinrich, *MIPS R4000 Microprocessor User's Manual.* MIPS Technologies Inc., 2nde ed., 1993.

[5] *MC88110: Second Generation RISC Microprocessor User's Manual.* Motorola Inc., 1ere ed., 1991.

[6] W. Ertel, "On the definition of Speedup," Rapport technique TR-93-069, Berkeley University, Dec. 1993.

[7] J. L. Hennessy et D. A. Patterson, *Architectures des ordinateurs : Une approche quantitative.* McGraw-Hill, 1ere ed., 1992.

[8] D. W. Wall, "Limits of Instruction-Level Parallelism," dans *Proc. 4th Architectural Support for Programming Languages*, (California), 1991.

[9] S. Jourdan, P. Sainrat, et D. Litaize, "Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor," dans *Proc. 22th Annual Symosium on Computer Architecture*, (Italy), Juin 1995.

[10] "PowerPC 604 RISC Microprocessor Technical Summary," Rapport technique MPR604TSU-01, IBM Microelectronics, 1994.