Compte Rendu TP d'Architecture : Simulation d'une mémoire cache.

De :

Jean-Baptiste DURIEZ
Jordane QUINCY

Calcul des codes permettant de savoir le type de mémoire cache :

Code : 'D' + 'Q' = 68 + 81 = 149, Gestion Ecriture = 149%2 = 1 = WB, Remplacement des blocs = 149%4 = 1 = LRU

Notre mémoire cache aura donc une gestion d'écriture en Write Back et un remplacement des blocs en Least Recently Used.

1) Effet de la taille d'un bloc sur les performances

Pour chacun des résultats suivants nous avons une mémoire cache ayant pour taille totale : 2048, pour degré d'associativité : 2 et une taille de bloc qui va varier entre 8 et 1024 en prenant les puissances de 2. Nous allons pour chaque trace, étudier la valeur des cycles perdus par la mémoire pour voir quelle combinaison de paramétrage est la meilleure.

a) Trace : multTrace10x10

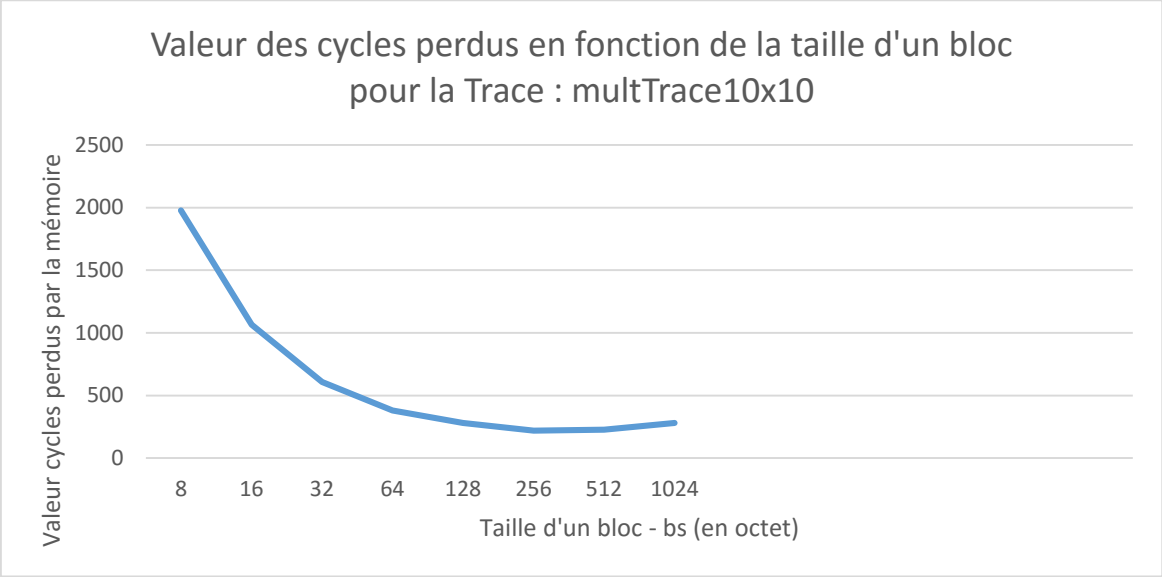
Il y a pour cette trace 5200 lectures et 1200 écritures

Tableau des résultats :

(voir page suivante)

Taille d'un bloc	8	16	32	64	128	256	512	1024
valeurCyclesPerdusMultTrace10x10	1976	1064	608	380	280	220	228	280
multTrace10x10Miss	152	76	38	19	10	5	3	2
multTrace10x10CopyInMemory	0	0	0	0	0	0	0	0
multTrace10x10Success	6248	6324	6362	6381	6390	6395	6397	6398
multTrace10x10DefautsLecture	101	51	26	13	7	4	2	1
multTrace10x10DefautsEcriture	51	25	12	6	3	1	1	1

Graphique : valeur des cycles perdus par la mémoire en fonction de la taille d'un bloc :



On constate que pour cette trace, plus on augmente la taille d'un bloc et plus la valeur des cycles perdus par la mémoire diminue (et donc plus les performances augmentent) mais cela uniquement jusqu'à une certaine taille de bloc. En effet au-delà d'une taille de bloc de 256 octets, les performances diminuent.

Donc pour cette trace la meilleure taille de bloc est de 256 octets.

On constate que le nombre de défauts et de succès est en relation avec la valeur des cycles perdus par la mémoire puisque quand les performances augmentent on a bien un nombre de défauts moins important et donc un nombre de succès plus important. De même pour la suppression d'une ligne du cache avec copie en mémoire.

Il y a plus de défaut en lecture qu'en écriture ce qui semble logique puis qu'on réalise beaucoup plus de lectures que d'écritures.

b) Trace : mergesort2000Trace

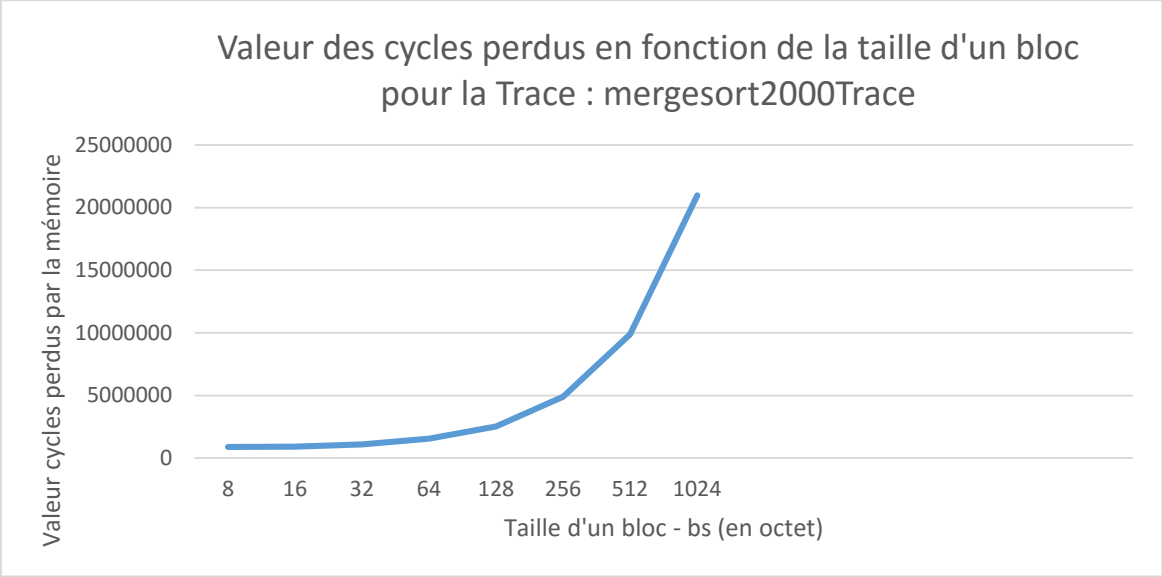
Il y a pour cette trace 85240 lectures et 65856 écritures

Tableau des résultats :

(voir page suivante)

Taille d'un bloc	8	16	32	64	128	256	512	1024
valeurCyclesPerdusMergesort2000Trace	894504	903224	1082576	1551160	2528596	4909256	9886688	20979840
mergesort2000TraceMiss	56632	54638	57482	64399	73149	86084	97456	105952
mergesort2000TraceCopyInMemory	12176	9878	10179	13159	17125	25490	32632	43904
mergesort2000TraceSuccess	94464	96458	93614	86697	77947	65012	53640	45144
mergesort2000DefautsLecture	44434	44749	47292	51238	55995	60608	64855	62300
mergesort2000DefautsEcritue	12198	9889	10190	13161	17154	25476	32601	43652

Graphique : valeur des cycles perdus par la mémoire en fonction de la taille d'un bloc :



Contrairement à la trace précédente, plus on augmente la taille d'un bloc et plus les performances diminuent ! Ainsi, pour cette trace, la meilleure taille de bloc est de 8 octets.

Au niveau des défauts en lecture et des défauts en écriture, on a à nouveau bien plus de défauts de lecture que de défauts d'écriture mais cette fois la différence entre le nombre de lectures et le nombre d'écritures est moins flagrante.

c) Trace : multTrace

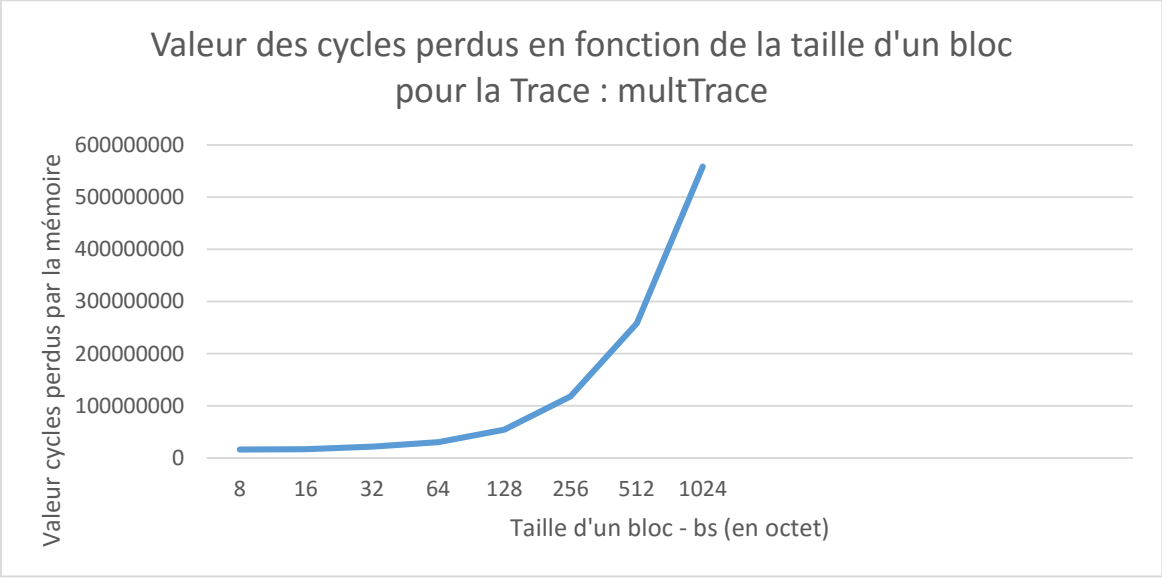
Il y a pour cette trace 5020000 lectures et 1020000 écritures.

Tableau des résultats :

(voir page suivante)

Taille d'un bloc	8	16	32	64	128	256	512	1024
valeurCyclesPerdusMultTrace	16646227	17242904	21856336	30754880	54699204	118196936	258324152	558430880
multTraceMiss	1248089	1188786	1282983	1375228	1640491	2125070	2572288	2988885
multTraceCopyInMemory	32390	42850	83038	162516	313052	561224	826714	999907
multTraceSuccess	4791911	4851214	4757017	4664772	4399509	3914930	3467712	3051115
multTraceDefautsLecture	1215678	1145910	1199925	1212706	1327437	1563844	1745573	1988977
multTraceDefautsEcriture	32411	42876	83058	162552	313054	561226	826715	999908

Graphique : valeur des cycles perdus par la mémoire en fonction de la taille d'un bloc :



Cette fois-ci, comme pour la trace précédente, plus on augmente la taille d'un bloc, plus la performance de la mémoire cache diminue. Et la valeur de la taille d'un bloc pour que la mémoire soit la plus performante possible est de 8 octets.

Encore une fois on a bien plus de défauts en lecture qu'en écriture.

Pour conclure il semblerait que pour des traces avec peu de lectures et d'écritures, une grande taille de bloc permet d'augmenter la performance de la mémoire cache, en revanche pour des traces de grande taille, c'est la valeur de la taille d'un bloc la plus basse qui permet d'avoir la meilleure performance.

2) Effet du degré d'associativité : trace : mergesort2000Trace

D'après les calculs, si la mémoire cache était infinie, le temps d'exécution du programme serait de 604380 nano secondes.

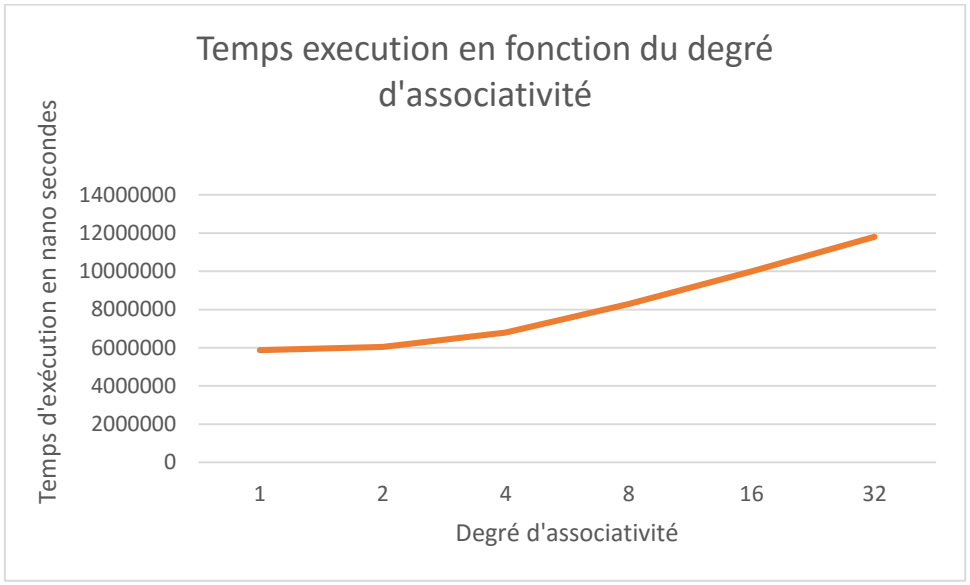
Nous allons maintenant simuler une mémoire cache avec comme taille 2048, comme taille de bloc 64 et on fait varier le degré d'associativité de 1 à 32.

Voici les résultats :

(voir page suivante)

Degré d'associativité	1	2	4	8	16	32
temps exec	5872690	6048438	6790728	8278426	9985892	11802975
success	81774	86697	84587	75257	65196	55513
défautsEcriture	55119	51238	54429	58833	63970	69825
défautsLecture	14203	13161	12080	17006	21930	25758

Graphique : valeur des cycles perdus par la mémoire en fonction de la taille d'un bloc :



On constate tout d'abord qu'avec une taille infinie de cache le programme s'exécute beaucoup plus vite qu'avec une taille limitée de cache. Ensuite sur le graphe on voit clairement que plus on augmente le degré d'associativité de la mémoire cache, plus le temps d'exécution du programme est long, on peut effectivement remarquer qu'en augmentant l'associativité le nombre de défauts augmente ce qui ralenti le temps d'exécution.

Code source : Vous l'avez également dans un fichier c

```
/*
   TP mémoire cache - Jordane QUINCY et Jean-Baptiste DURIEZ
   Etat : Fini
   Code : 'D' + 'Q' = 68 + 81 = 149, Gestion Ecriture = 149%2 = 1 = WB, Remplacement des blocs = 149%4
= 1 = LRU
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
typedef struct s {
   int valid;
   double tag;
   //On a besoin d'un compteur pour gérer le LRU
   int compteur;
   int M;
}bloc;

typedef struct {
   int bs;
   int cs;
   int asso;
   bloc **Cache;
   long nbrFailReading;
   long nbrFailWriting;
   long nbrSuppCache;
   long nbrCopyInMemoryAfterSuppCache;
   long nbrHitReading;
   long nbrHitWriting;
   long nbrOfReading;
   long nbrOfWriting;
}ModelCache;


ModelCache initializeCache (int cs, int asso, int bs) {
   ModelCache C;
   C.cs = cs;
   C.asso = asso;
   C.bs = bs;
   C.nbrFailReading = 0;
   C.nbrCopyInMemoryAfterSuppCache = 0;
   C.nbrFailWriting = 0;
   C.nbrSuppCache = 0;
   C.nbrHitReading = 0;
   C.nbrHitWriting = 0;
```

```
   C.nbrOfReading = 0;
   C.nbrOfWriting = 0;
   int nbrElement = cs / (asso*bs);
   int i, j;
   C.Cache = malloc(nbrElement * sizeof (bloc));
   for (i = 0; i < nbrElement; i++) {
      C.Cache[i] = malloc(asso * sizeof(bloc));
   }
   for (i = 0; i < nbrElement; i++) {
      for (j = 0; j < asso; j++) {
         C.Cache[i][j].valid = 0;
         C.Cache[i][j].tag = (double)-1;
      }
   }
   return C;
};

long getNbCyclePerdu (int bs, long nbDefautLecture, long nbDefautEcriture, long
nbLigneSupprDuCache)
{
          //Pénalité d'un défaut = (12 + bs/8) cycles
          //Cycles perdu par la mémoire =
          // nombre de cycles dues à la pénalité des défauts * (nombre défauts lecture + nombre défauts
écriture + nombre de lignes supprimées du cache)
   return (12 + bs/8) * (nbDefautLecture + nbDefautEcriture + nbLigneSupprDuCache);
};

// Reading Gestion
void addressTreatment (int index, double tag, ModelCache *C, int isWrite) {
   int i;
   //allZero vaut 1 si tout les bits valid à chaque bloc de l'index vaut 0, allZero vaut 0 sinon
   int allZero = 1;
   //indexToReplaceLRU va prendre la valeur de l'index(j) à remplacer si tous les bits valid valent 1 et
qu'aucun tag ne correspond (il n'est donc pas toujours utilisé)
   int indexToReplaceLRU = 0;
   //indexToReplace va prendre la valeur de l'index(j) à remplacer si les tags ne correspond pas et si il
y a au moins un bit valid à 0 (dans ce cas on utilise pas le LRU)
   int indexToReplace = -1;
   //tagFounded vaut 0 si on ne trouve pas le tag déjà présent, 1 sinon
   int tagFounded = 0;
   //indexTagFounded prend la valeur de l'index(j) si un bloc a le même tag que celui à lire (afin
d'incrémenter le compteur pour le LRU)
   int indexTagFounded = -1;
   if (isWrite) {
      C->nbrOfWriting++;
   }
   else {
      C->nbrOfReading++;
```

```
}
//Loop for to determine in which case we are (+ save util data for the LRU)
for (i = 0; i < C->asso; i++) {
   if (C->Cache[index][i].valid == 0) {
         indexToReplace = i;
   }
   else {
      //There is one valid bit set to 1
      allZero = 0;
      //Find the index of the LRU
      if (C->Cache[index][indexToReplaceLRU].compteur > C->Cache[index][i].compteur) {
         indexToReplaceLRU = i;
      }
      //Check if the tag is the same
      if (C->Cache[index][i].tag == tag) {
         tagFounded = 1;
         indexTagFounded = i;
      }
   }
}
//Manage the cache memory
//Case when all valid bit are 0
if (allZero) {
   C->Cache[index][indexToReplace].valid = 1;
   C->Cache[index][indexToReplace].tag = tag;
   C->Cache[index][indexToReplace].M = 0;
   C->Cache[index][indexToReplace].compteur = 0;
   if (isWrite) {
      C->nbrFailWriting++;
      C->Cache[index][indexToReplace].M = 1;
   }
   else {
      C->nbrFailReading++;
   }
}
else {
   if (tagFounded) { //It is a hit !
      C->Cache[index][indexTagFounded].compteur++;
      if (isWrite) {
         C->nbrHitWriting++;
         C->Cache[index][indexTagFounded].M = 1;
      }
      else {
         C->nbrHitReading++;
      }
   }
   else if (indexToReplace >= 0) { //There is at least one valid bit sets to 0
      C->Cache[index][indexToReplace].valid = 1;
```

```
            C->Cache[index][indexToReplace].tag = tag;
            C->Cache[index][indexToReplace].M = 0;
            C->Cache[index][indexToReplace].compteur = 0;
            if (isWrite) {
               C->Cache[index][indexToReplace].M = 1;
               C->nbrFailWriting++;
            }
            else {
               C->nbrFailReading++;
            }
         }
         else { //All valid bits are 1 and it is a fail so use LRU and replace one bloc
            C->nbrSuppCache++;
            if (C->Cache[index][indexToReplaceLRU].M) {
               C->nbrCopyInMemoryAfterSuppCache++;
            }
            C->Cache[index][indexToReplaceLRU].valid = 1;
            C->Cache[index][indexToReplaceLRU].tag = tag;
            C->Cache[index][indexToReplaceLRU].M = 0;
            C->Cache[index][indexToReplaceLRU].compteur = 0;
            if (isWrite) {
               C->Cache[index][indexToReplaceLRU].M = 1;
               C->nbrFailWriting++;
            }
            else {
               C->nbrFailReading++;
            }
         }
      }
   };



   // Address analysis
   void addressAnalysis (char car ,long address, ModelCache *C) {
      int isWrite = 0;
      long addressBase10 = address;
      long numBloc = addressBase10 / C->bs;
      int nbrEntree = C->cs / (C->bs * C->asso);
      int index = numBloc % nbrEntree;
      double tag = (double)numBloc / nbrEntree;
      // Check if it is a reading or a writing
      if (car == 'W')  {
         isWrite = 1;
      }
      addressTreatment(index, tag, C, isWrite);
   };
```

```
double calculTempsExec(ModelCache *C){
   double nbrDeHit = (C->nbrHitReading + C->nbrHitWriting) * 40/100;
   double nbrDeMiss = (C->nbrFailReading + C->nbrFailWriting) * 40/100;
   return (nbrDeHit * (10 + log2(C->asso))) + (nbrDeMiss * 20 * (10 + log2(C->asso)));
}

void main(int argc, char *argv[]) {
   //Test the number of arguments
   if (argc != 5) {
      printf("Le nombre d'arguments est invalide, il doit etre egal à 5 !!\n");
   }
   else {
      int cs = atoi(argv[1]);
      int bs = atoi(argv[2]);
      int asso = atoi(argv[3]);
      char* trace = argv[4];
      char car;
      long adre;
      ModelCache C = initializeCache(cs, asso, bs);
      printf("Les donnees sont :\nTaille de la memoire cache : %d octets\nTaille d'un bloc : %d
octets\nDegre d'associativite : %d\nNom du fichier analyse : %s\n", cs, bs, asso, trace);
      FILE* tr = fopen(trace, "r");
      while(!feof(tr)) {
         fscanf(tr, "%c%X\n", &car, &adre);
         addressAnalysis(car, adre, &C);
      }
      fclose(tr);
      printf("\nResultats apres l'analyse du fichier d'addresses :\n");
      printf("Nombre de lectures : %ld\nNbr d'ecritures %ld", C.nbrOfReading, C.nbrOfWriting);
      printf("\n\nnbr Fail LECTURE : %ld\n", C.nbrFailReading);
      printf("nbr Fail ECRITURE : %ld\n\n", C.nbrFailWriting);
      printf("nbr succes : %ld\n", C.nbrHitReading + C.nbrHitWriting);
      printf("Compy in memory : %ld \n", C.nbrCopyInMemoryAfterSuppCache);
      printf("Nbr de supp : %ld \n", C.nbrSuppCache);
      printf("Valeur cycle perdu %ld\n", getNbCyclePerdu(bs, C.nbrFailReading, C.nbrFailWriting,
C.nbrCopyInMemoryAfterSuppCache));
      //nbr d'instruction SW et LW
      double nbrInstruction = (C.nbrFailReading + C.nbrFailWriting + C.nbrHitReading + C.nbrHitWriting)
*40/100;
      double tempsExecInfini = nbrInstruction * 10;
      printf("temps exec taille infinie : %lf ns\n", tempsExecInfini);
      printf("tmps exec : %lf\n", calculTempsExec(&C));
   }
}
```