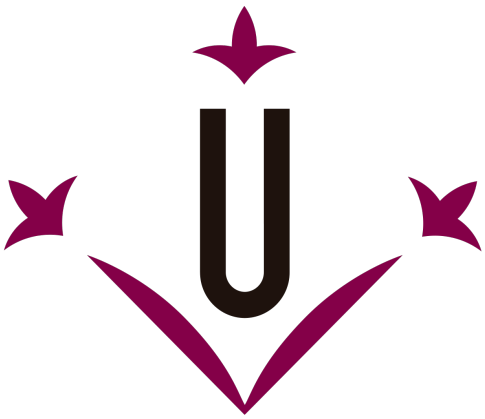

Universitat de Lleida

Jerarquia de Memòria

Arquitectura de Dades

Escola Politècnica Superior

Grau en Enginyeria Informàtica

Yhislaine Nataly, Jaya Salazar

Valeria Adriana, Escalera Flores

14.03.2024

Índex

1 Introducció

En aquesta pràctica s'analitza l'efectivitat d'un sistema de memòria cache (MC), considerant la localització de referència dels programes en estudi i els paràmetres de disseny de la pròpia MC. Per això, s'utilitza el simulador `SMPcaché`, que permet avaluar el comportament dels accessos a memòria mitjançant l'anàlisi d'un fitxer de traça generat per l'execució d'un programa.

L'estudi se centra en la generació d'una traça d'accessos a memòria mitjançant un programa en llenguatge C anomenat `calcul_array.c`. Aquest programa llegeix un conjunt de valors des d'un fitxer, els organitza en diferents estructures de dades i registra els accessos a memòria en un fitxer de traça anomenat `tr_gestiona_vector.prg`. Posteriorment, aquesta traça s'empra per avaluar l'impacte de diferents configuracions de memòria cau i algoritmes de substitució en el rendiment del sistema.

A partir dels resultats obtinguts, es realitza una anàlisi del percentatge de fallades en memòria cau, considerant diferents mides de memòria, mides de bloc i polítiques de substitució. Finalment, es presenten conclusions sobre el comportament de la jerarquia de memòria en funció dels paràmetres avaluats.

2 Avaluació de paràmetres de disseny de la MC

En aquesta secció s'analitza el rendiment de diferents configuracions de memòria cau mitjançant l'avaluació del percentatge de fallades generades pel fitxer de traça . Aquest procés es realitza utilitzant el simulador `SMPcache`, configurant una memòria principal amb les característiques següents:

- Mida: 16 Kbytes
- Mida paraula: 1 byte

La memòria cau pot tenir diferents configuracions en relació amb la mida total, la mida del bloc i el tipus d'organització, segons els valors indicats a la taula corresponent. L'algorisme de substitució de blocs utilitzat en les configuracions que ho requereixen és *Least Recently Used* (LRU).

Mida MC	Mida bloc	directa	2 blocs/conjunt	4 blocs/conjunt
64 bytes	2bytes	42,250%	42,250%	42,250%
	8bytes	21,214%	21,133%	21,129%
	16bytes	10,768%	10,575%	10,567%
128 bytes	2bytes	42,250%	42,250%	42,250%
	8bytes	21,177%	21,125%	21,125%
	16bytes	10,663%	10,563%	10,563%
1Kbyte	2bytes	0,805%	0,805%	0,805%
	8bytes	0,402%	0,402%	0,402%
	16bytes	0,201%	0,201%	0,201%

Figura 1: Taula de Resultats

2.1 Directa

En el gràfic es pot apreciar que, en aquest cas, en mides de MC petites, té un índex de fallades més elevat que en els altres algorismes de correspondència, encara que no sigui per una diferència significable. Això es deu al fet que un bloc concret només es pot ubicar en una posició determinada de MC, aquesta limitació té una afectació important, ja que si coincideix que es necessiten simultàniament dos blocs que s'ubiquen en la mateixa posició de MC hauran d'anar-se alternant i el nombre de fallades augmentarà.

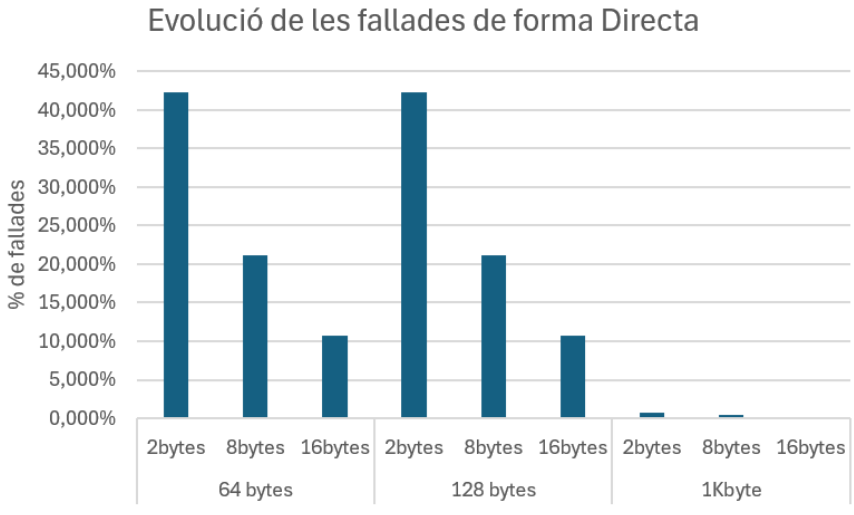

Figura 2: Gràfic de fallades en Directa

2.2 2 blocs/conjunt i 4 blocs/conjunts

Entre les opcions associatives per conjunt observem que l'índex de fallades és molt similar, però és més baix en els dos casos, ja que és més flexible permetent que un bloc determinat es pugui ubicar en dues possibles posicions en el cas de 2blocs/conjunts i en quatre posicions en el cas de 4blocs/conjunts.

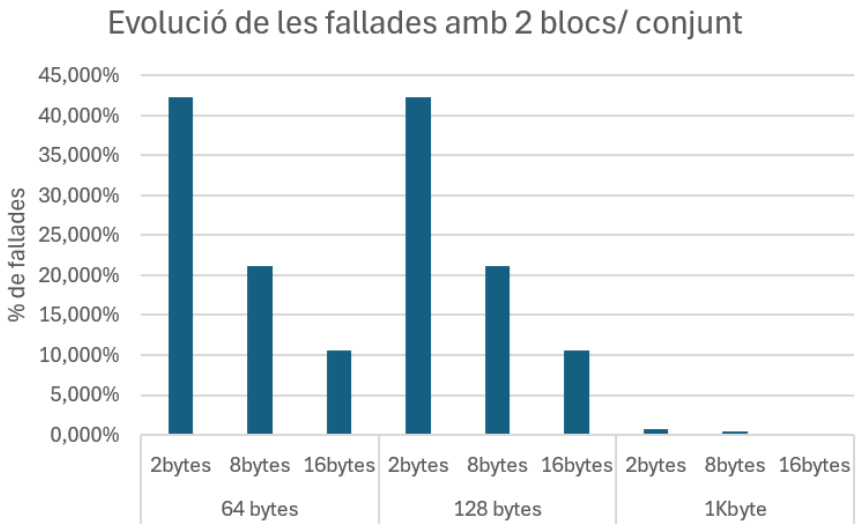

Figura 3: Gràfic de fallades en 2 blocs/conjunt

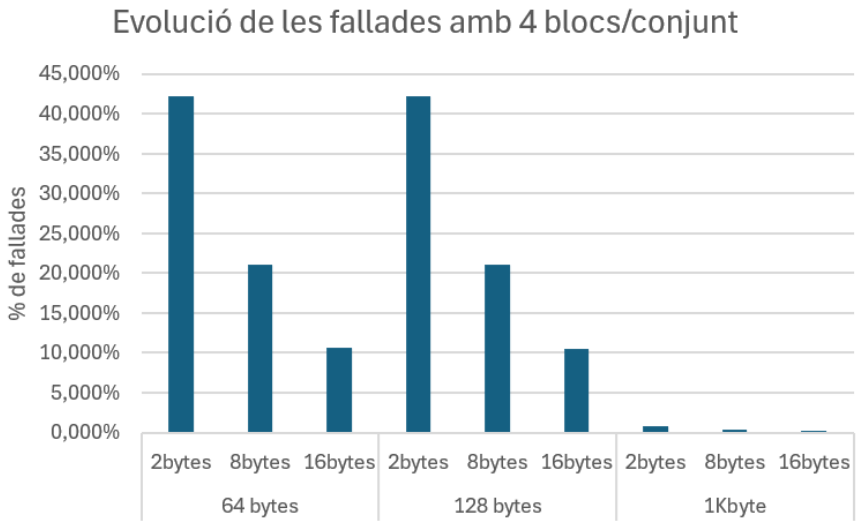

Figura 4: Gràfic de fallades en 4 blocs/conjunt

2.3 Conclusions sobre la configuració de la MC

D'altra banda, també s'ha de tenir en compte la complexitat de la circuiteria, la qual afecta al cost. La circuiteria depèn de l'algorisme, en el cas de la directa només requereix un comparador ja que un bloc determinat només pot estar en una posició concreta, en el cas dels algorismes d'associació per conjunts hi haurà tants comparadors com bloc pugui tenir un conjunt. Per tant, podem concloure que l'opció més òptima serà la correspondència directa, ja que aquesta serà la més econòmica amb només un comparador en la circuiteria, ja que la reducció de l'índex de fallades observada en les opcions de associativitat per conjunts no és proporcional amb l'increment de comparadors i en conseqüència amb el cost. A més a més, amb una memòria més gran la diferència en l'índex és pràcticament inexistent.

En conclusió, un cop analitzat detalladament els índexs de fallada segons les diferents configuracions de MC, la combinació que proporciona el balanç més ideal entre rendiment i cost és una MC de mida de 1KB, mida de bloc de 16 bytes amb l'algorisme de correspondència directa.

3 Anàlisi de la influència dels diferents algorismes de substitució

En aquesta secció s'estudia l'impacte de diferents algorismes de substitució en el rendiment de la memòria cau. Els algorismes analitzats són *Least Recently Used* (LRU), Aleatori, *First In First Out* (FIFO) i *Least Frequently Used* (LFU). L'objectiu és determinar quina estratègia ofereix un menor percentatge de fallades en funció de les configuracions de la memòria caché (MC) utilitzades.

Mida MC	Mida bloc	LRU	Aleatori	FIFO	LFU
64 bytes	2 bytes	42,250%	39,868%	42,250%	42,250%
	8 bytes	21,552%	27,008%	21,552%	21,552%
	16 bytes	10,993%	14,055%	10,989%	11,005%
128 bytes	2 bytes	42,250%	21,785%	42,250%	41,043%
	8 bytes	21,552%	21,548%	21,552%	21,552%
	16 bytes	10,989%	12,240%	10,989%	10,989%
1 Kbytes	2 bytes	0,805%	3,320%	0,805%	0,805%
	8 bytes	0,414%	0,503%	0,414%	0,414%
	16 bytes	0,213%	0,237%	0,237%	0,215%

Figura 5: Taula resultats

3.1 LRU (Least Recently Used)

L'algorisme de substitució *Least Recently Used* (LRU) generalment ofereix bons resultats. Això es deu al fet que reemplaça les dades menys utilitzades recentment, captant correctament la localitat temporal. És a dir, si una variable ha estat referenciada o accedida recentment, és probable que torni a ser utilitzada en el futur i, per tant, encara es trobarà a la Memòria Cau (MC). Per aquest motiu, LRU es manté estable i eficient en diferents configuracions de mida de memòria cau i mida de bloc.

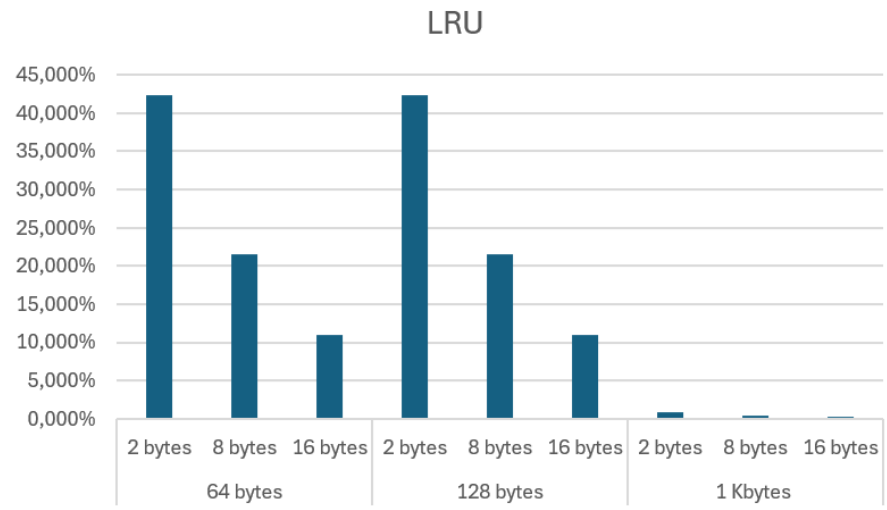

Figura 6: Resultats amb LRU

3.2 Aleatori

L'algorisme de substitució aleatori presenta un rendiment menys eficient en comparació amb els altres, ja que elimina blocs útils a l'atzar, generant taxes de fallades més elevades. Atès que no segueix cap criteri específic per a la substitució, no es pot identificar cap patró ni extreure conclusions clares sobre la seva eficàcia. El seu comportament varia en cada execució.

Tanmateix, pot tenir una lleugera avantatge en termes de temps de resposta, ja que no necessita aplicar cap estratègia de cerca. Destaca que en la memòria cau de 128 bytes, el seu rendiment es manté similar en les dues primeres configuracions analitzades.

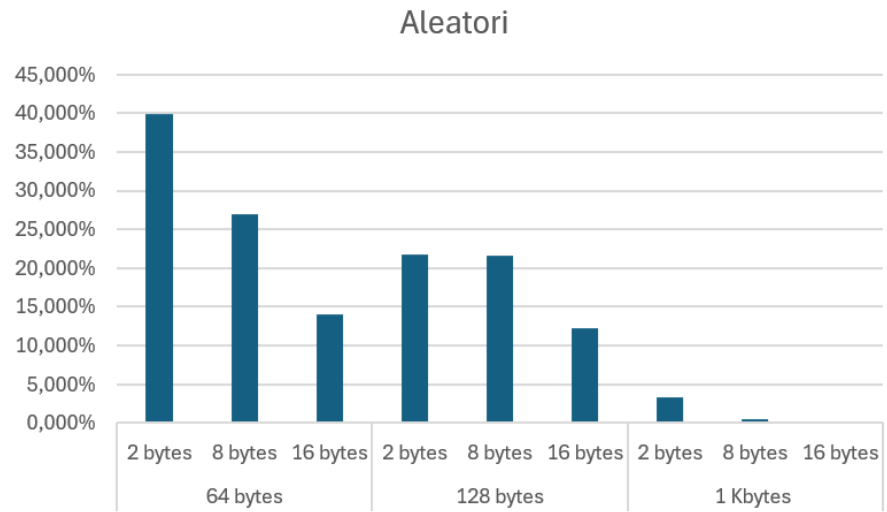

Figura 7: Resultats amb Aleatori

3.3 FIFO (First In First Out)

FIFO presenta un rendiment similar a LRU en alguns casos, però pot ser menys eficient en situacions en què les dades més antigues continuen sent rellevants. En el cas d'aquest algorisme, els resultats són força òptims, la qual cosa podria ser deguda a particularitats del patró d'accés de la traça analitzada.

És possible que, quan un bloc és eliminat de la memòria cau perquè ha estat el primer en entrar, el programa ja hagi finalitzat els accessos necessaris a aquest bloc. Això significaria que la seva eliminació no afecta negativament el rendiment global.

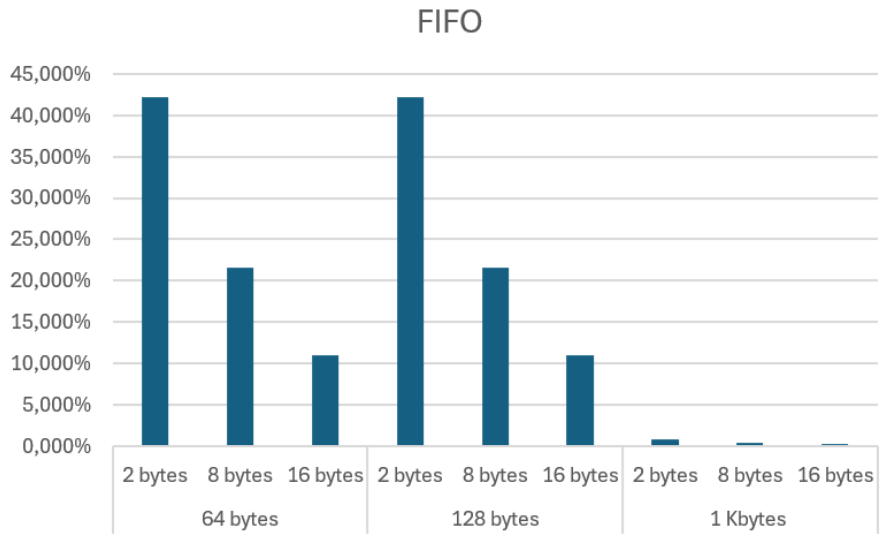

Figura 8: Resultats amb FIFO

3.4 LFU (Least Frequently Used)

LFU presenta un comportament similar a LRU, però pot veure's afectat si determinats blocs són accedits intensament en un període curt de temps i després ja no es tornen a utilitzar. Un inconvenient d'aquest algorisme és que un bloc determinat, després d'haver estat referenciat moltes vegades, pot romandre innecessàriament a la memòria cau durant molt de temps, fins que altres blocs el superin en freqüència d'ús.

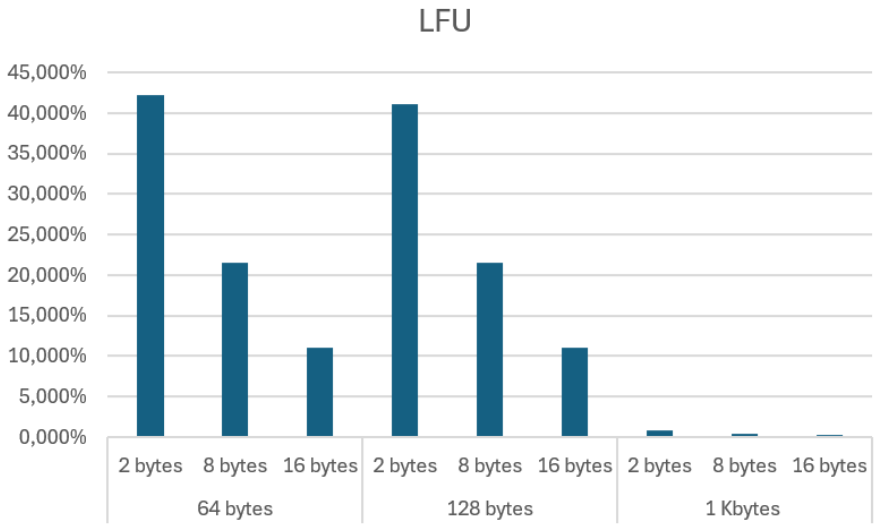

Figura 9: Resultats amb LFU

3.5 Consideracions

- Per a una memòria cau de 64 bytes, el percentatge de fallades és elevat en tots els algorismes, ja que la capacitat d'emmagatzematge és massa petita.
- En una memòria cau de 1 KB amb blocs de 16 bytes, les errades són mínimes en tots els algorismes, destacant la millora aconseguida amb l'augment de la memòria i la mida del bloc.

3.6 Conclusions sobre els algorismes de substitució

L'augment de la mida de la memòria cau i del tamany dels blocs redueix el percentatge de fallades i millora el rendiment. **LRU i LFU** són, en general, més eficients que **FIFO i Aleatori**. **FIFO** pot ser menys eficient en determinats casos, especialment quan es treballa amb patrons d'accés que requereixen reutilitzar dades antigues. Finalment,

l'algorisme aleatori presenta una gran variabilitat i, per tant, no és la millor opció en termes de predictibilitat i eficiència.

D'acord amb els resultats obtinguts, es pot concloure que **LFU i LRU és l'algorisme lleugerement més òptim** per a l'execució del programa analitzat i la seva traça d'accés associada. Això es deu al fet que capta adequadament la localitat temporal i el seu bon rendiment queda reflectit en els gràfics.

A més, s'observa que **la influència dels algorismes de substitució disminueix a mesura que augmenta la mida de la memòria cau**. En el cas analitzat, la configuració amb **1 KB de memòria cau i blocs de 8 bytes** ha mostrat un bon comportament.

Finalment, cal tenir en compte que una memòria cau totalment associativa pot millorar encara més el rendiment, però implica un cost elevat a causa de la necessitat d'un comparador per cada entrada de la memòria cau.

4 Annex

```c
#include <stdio.h>
#include <string.h>

#define N 100

// ORDENACI DE L'ARRAY
void ordenacio_array(int v[], int n, FILE *fitxer){
  int temp;
  for(int i = 0; i < n; i++){
    for(int j = 0; j < n - 1; j++){
      fprintf(fitxer, "%d %p\n", 2, (void*)&v[j]); // Lectura
      fprintf(fitxer, "%d %p\n", 2, (void*)&v[j+1]); // Lectura
      if (v[j] < v[j + 1]){
        temp = v[j];
        fprintf(fitxer, "%d %p\n", 3, (void*)&v[j]); //
           Escriptura
        v[j] = v[j + 1];
        fprintf(fitxer, "%d %p\n", 3, (void*)&v[j+1]); //
           Escriptura
        v[j + 1] = temp;
      }
    }
  }
}

//ARRAY AMB NOMBRES PARELLS
void parells(int v[], int p[], int n, FILE *fitxer) {
  int j = 0;
  for(int i = 0; i < n; i++){
    fprintf(fitxer, "%d %p\n", 2, (void*)&v[i]); // Lectura
    if(v[i] % 2 == 0){
      p[j] = v[i];
```

```
31        fprintf(fitxer, "%d %p\n", 3, (void*)&p[j]); // Escriptura
32        j++;
33      }
34    }
35  }
36
37  int obtenir_nombre_parells(int v[], int n, FILE *fitxer){
38    int numParells = 0;
39    for (int i = 0; i < n; i++){
40      fprintf(fitxer, "%d %p\n", 2, (void*)&v[i]); // Lectura
41      if (v[i] % 2 == 0 ){
42        numParells++;
43      }
44    }
45    return numParells;
46  }
47
48  // ARRAY AMB NOMBRES SENARS
49  void senars(int v[], int p[], int n, FILE *fitxer) {
50    int j = 0;
51    for(int i = 0; i < n; i++){
52      fprintf(fitxer, "%d %p\n", 2, (void*)&v[i]); // Lectura
53      if(v[i] % 2 == 1){
54        p[j] = v[i];
55        fprintf(fitxer, "%d %p\n", 3, (void*)&p[j]); // Escriptura
56        j++;
57      }
58    }
59  }
60
61  int obtenir_nombre_senars(int v[], int n, FILE *fitxer){
62    int numSenars = 0;
63    for (int i = 0; i < n; i++){
64      fprintf(fitxer, "%d %p\n", 2, (void*)&v[i]); // Lectura
```

```
65      if (v[i] % 2 == 1 ){
66        numSenars++;
67      }
68    }
69    return numSenars;
70  }
71
72  int main (){
73    // Obrir el fitxer de  t r a a  amb el nom correcte
74    FILE *fitxer = fopen("tr_calcul_array.prg", "w");
75    if (fitxer == NULL) {
76      printf("Error obrint el fitxer de  t r a a .\n");
77      return 1;
78    }
79
80    int valors[N] = {
81          23, 12, 33, 93, 126, 45, 125, 669, 12, 4,
82          9, 25, 7, 89, 112, 23, 269, 59, 458, 47,
83          15, 35, 48, 126, 240, 125, 230, 120, 111, 2,
84          55, 598, 88, 44, 69, 78, 15, 124, 128, 450,
85          460, 5, 8, 97, 5, 12, 11, 33, 66, 125,
86          658, 458, 98, 69, 548, 59, 69, 230, 220, 365,
87          5, 6, 8, 40, 10, 69, 58, 45, 89, 2,
88          6, 9, 8, 45, 55, 66, 87, 1200, 125, 69,
89          69, 25, 83, 84, 59, 66, 98, 2, 9, 7,
90          89, 56, 6, 45, 69, 8, 100, 25, 200, 23
91      };
92
93      int p[N];
94      int s[N];
95
96      // O R D E N A C I  ARRAY
97      ordenacio_array(valors, N, fitxer);
98
```

```
99      // ARRAY NOMBRES PARELLS
100     int numParells = obtenir_nombre_parells(valors, N, fitxer);
101     parells(valors, p, N, fitxer);
102
103     // ARRAY NOMBRES SENARS
104     int numSenars = obtenir_nombre_senars(valors, N, fitxer);
105     senars(valors, s, N, fitxer);
106
107     // Tanquem el fitxer
108     fclose(fitxer);
109     printf("Traa generada correctament a tr_calcul_array.prg\n
            ");
110
111     return 0;
112 }
```

5 Bibliografía

Concepció Roig Mateu, *Jerarquia de Memòria*, 2025.

W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 10a edició, Pearson, 2015.

A. J. Smith, *Cache Memories*, ACM Computing Surveys, vol. 14, núm. 3, pp. 473-530, 1982.