Vol. (6) 2: pp. 48-55

SIMULACIÓN DE LA MEMORIA CACHE

** Edgar Ruiz L.*
** Eduardo Raffo L.*

RESUMEN

El artículo presenta un programa que simula la memoria cache considerando el porcentaje de éxitos y el tiempo medio de acceso del sistema. El programa encuentra el tamaño óptimo de la memoria cache con respecto a l tamaño de la memoria RAM el cual puede utilizarse como un criterio válido para cuestiones del diseño de ambas memorias en una computadora.

Palabras Claves: Memoria cache. Memoria RAM. Distribución uniforme. Tiempo de acceso medio.

ABSTRACT

This article presents a program simulating the Cache Memory, by considering the system's success percentage and average access time.The program finds the Cache Memory's optimal size with regard to the RAM Memory size, which can be used as a valid criterion in designing both kinds of Memory within a computer.

Key Words: Cache memory. RAM memory. Uniform distribution. Average access time.

INTRODUCCIÓN

El trabajo consiste, en un programa escrito en lenguaje C++; utilizando un compilador Borland® C++ v.3.5 para Windows y DOS. La carta de estructuras para el programa se muestra en la Figura 1.

OBJETIVO

Simular el porcentaje de aciertos acerca de que un dato se encuentre o no en la cache, así como, estimar el tiempo de acceso medio a la cache

El módulo principal del programa es el archivo fuente *cache.cpp* el cual pide como datos:

1. Ingresar tamaño de la memoria.
2. Ingresar tamaño de la cache

Luego se inicializa la cache, la cual es modelada como una estructura de datos del tipo FIFO(del ingles First In First Out), más conocida como una cola. Finalmente, simula una distribución uniforme de los bloques de memoria principal los cuales tendrán la misma oportunidad de ingresar a la memoria cache e imprime el porcentaje de éxitos y calcula el tiempo de acceso medio a la *cache*.

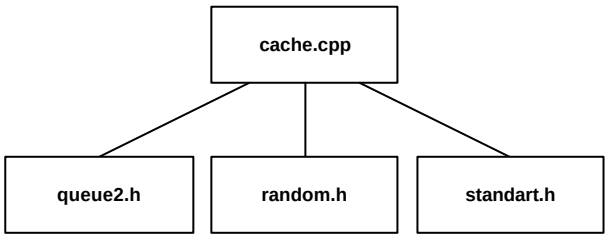

Figura 1. Carta de estructuras para el programa que simula la memoria *cache*

* Instituto de Investigación
Facultad de Ingeniería Industrial, UNMSM
E-mail: eraffol@unmsm.edu.pe

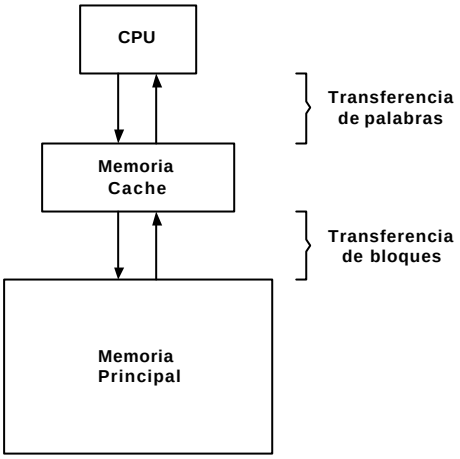

Figura 2. Memoria pricipal y cache

Cuadro 1. Elementos de diseño de la memoria cache

Tamaño de la Cache	
Función de Correspondencia	
I)	Directa
II)	Asociativa
III)	Asociativa por conjuntos
Algoritmo de Sustitución	
IV)	Utilizado menos recientemente (LAN)
V)	Primero en entrar - Primero en salir (FIFO)
VI)	Utilizado menos frecuentemente (LFU)
VII)	Aleatorio
Política de escritura	
VIII)	Escritura inmediata
IX)	Post- escritura
X)	Escritura única
Tamaño del bloque	
Número de caches	
XI)	Uno o dos niveles
XII)	Unificada o partida

FUNDAMENTOS TEÓRICOS [1]

Debido a que la memoria principal es relativamente más grande y más lenta, la memoria *cache* se diseña para que sea más pequeña y rápida. La *cache* tiene una copia de partes de la memoria principal. Cuando la CPU intenta leer una palabra de la memoria, se hace una comprobación para determinar si la palabra está en la cache. Si es así se entrega dicha palabra a la CPU. Si no; un bloque de memoria principal, consistente en un cierto número de palabras, se transfiere a la *cache* y después la palabra es entregada a la CPU. La Figura 2 muestra la idea básica de las memorias principal y cache.

CRITERIOS PARA EL DISEÑO DE LA CACHE

Además del costo total medio por *bit* para encontrar el tamaño "optimo" de la cache, existen las siguientes alternativas de diseño de la cache mostradas en el Cuadro 1.

Para el programa se asume una función de correspondencia directa donde cada bloque K de la RAM tiene la misma oportunidad que otro de ser ingresado a la cache. El algoritmo de sustitución empleado es una estructura del tipo FIFO que se traduce en un Tipo Abstracto de Datos conocido como TAD Cola.

[1] Los fundamentos teóricos han sido tomados de la referencia (2) citada en la bibliografía.

EL PROGRAMA DE SIMULACIÓN

Como ya se indico en la introducción el programa está escrito en lenguaje de programación C++, bajo el paradigma de la programación orientada al objeto, y donde se hace uso de librerías (archivos *.h) escritos por los autores, que pueden servir para este y otros programas. La Figura 3 muestra el diagrama de flujo del programa.

Después de leer los datos: tamaño de memoria (variable *nmemoria*) y tamaño del *cache* (variable *ncache*) se inicializa o llena la cache invocando el procedimiento *inicia* cuya tarea consiste en insertar el bloque en la cola.

La función *simula* genera muestras para las posiciones en la memoria RAM utilizando una distribución uniforme de probabilidades y así ubicar el bloque *j* en la *cache* (ver Figura 5), luego averigua si el bloque *j* pertenece a la cache, de ser así se incrementa el número de aciertos o *hit*, caso contrario se remueve el bloque de la cola y se inserta el bloque *j*.

El sustento de la distribución uniforme (archivo *random.h*), está dado por la siguiente definición:

Sea:

$$x \in [a,b]$$

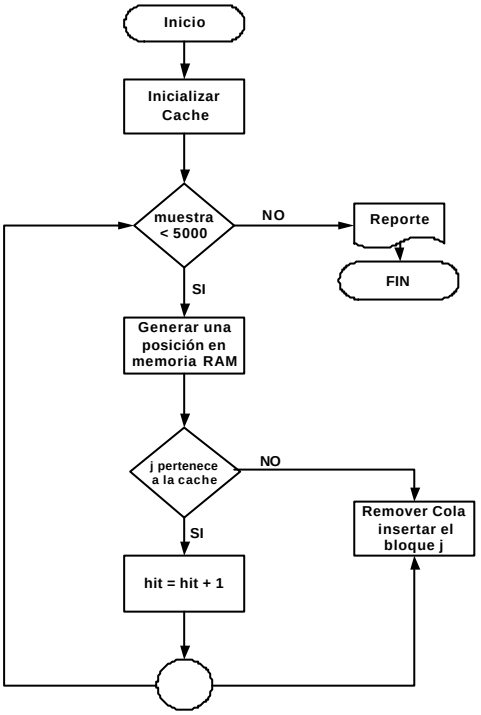

Figura 3. Diagrama de flujo del programa

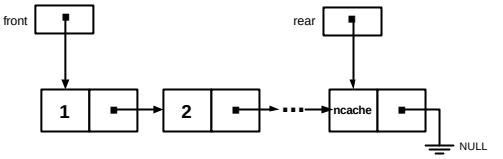

Figura 4. El TAD Cola

Donde cualquier valor de x tiene igual oportunidad de ser elegido, luego:

$$F(x) = \int_a^x f(x)dx = \int_a^x \frac{dx}{b-a} = \frac{x-a}{b-a}$$

como $F(X) \in [0,1]$, entonces $r = F(x)$, $r \in [0,1]$ y $r = \frac{x-a}{b-a}$, con lo cual:

$$x = a + r*(b-a)$$

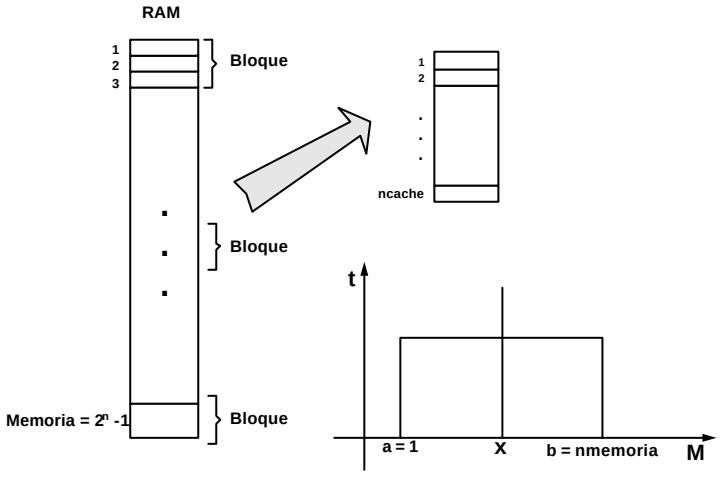

Figura 5. Memoria RAM, memoria cache y distribución uniforme

```
// Archivo fuente: cache.cpp
// simulacion de la memoria cache
#include <iostream.h>
#include <iomanip.h>
#include "a:queue2.h"
#include "a:standart.h"
#include "a:random.h"

void inicio(int);
double simule(int);

Queue<int> cache;
const int BLOQUES=20;
void main()          // cache.cpp
{
    int nmemoria,ncache;      // nmemoria = tamanno memoria
                              // ncache = tamanno de la cache
    cout << "ingrese el tamaño de la memoria > ";
    cin  >> nmemoria;
    cout << "ingrese el tamaño del cache > ";
    cin  >> ncache;
    inicio(ncache);  // llamar a inicio
    // reporte
    double hit=simule(nmemoria);
    cout << endl << setw(10) << setprecision(4)
         << "porcentaje de exitos   " << (hit*100)
         << endl;
    // se asume un tiempo de acceso T=10
    double tacceso=1*hit+(1-hit)*10;
    cout << endl << setw(10) << setprecision(4)
         << "el tiempo de acceso es " << tacceso
         << " Useg." << endl;
}
void inicio(int ncache)
{
    for(int i=1;i<=ncache;i++)
        cache.insert(i);
}
double simule(int nmemoria)
{
    int n=0,m;
    for(int i=1;i<=5000;i++){
        // unifr=distribucion uniforme
        m=unifr(1,nmemoria);
        int j=(m/BLOQUES)+1;
        if(cache.search(j))
            n++;
        else {
            cache.remove();
            cache.insert(j);
        }
    }
    return (double)n/5000;
}
```

Figura 6. Listado del archivo *cache.cpp*

La función *rnd( )*, permite generar un número aleatorio de la forma lineal dado por L(G).

$$Z_{n+1} = aZ_n \pmod{m}$$

$$Z_{n+1} = residuo\,(a * Z_n, m)$$

Si $m = 2^b$ y $a = 8k + 3$ ó
$a = 8k+5$ donde *período* = 2^{b-2}; entonces
$m = HIGHVALUE = 2^{15} - 1$ por lo tanto los residuos están entre $(0,2^{15} - 1)$ es decir (0,32767)

Todo este proceso se repite para 5 000 muestras, con lo cual se prepara el reporte para indicar el porcentaje de éxitos mediante la fórmula hit*100 y estimar el tiempo de acceso medio del sistema con:

$$t_{acceso} = T_1 * hit + (1 - hit) * T_1$$

Donde se asume que el tiempo de acceso medio M es de 10 unidades.

A continuación se presenta el código del programa completo *cache.cpp* y el de las librerías *queue2.h*, *random.h*, y *standart.h*

```
// queue2.h
#ifndef  QUEUE_H
#define  QUEUE_H
#include <iostream.h>
#include <assert.h>
#include "standart.h"
// Node
#define  NODE  Node<T>*

template <class T>
class Node {
    T info;
    NODE next;
public :
    Node():next(0){};
    Node(const T&);
    NODE getnext()const;
    // sobrecargando flujo de salida
    friend ostream& operator<<(ostream&,const Node<T>&);
    friend class Queue<T>;
};
template <class T>
Node<T>::Node(const T& a):info(a)
```

Figura 7a. Listado del archivo *queue2.h*

```
{
    next=0;
}
template <class T>
NODE Node<T>::getnext()const
{
    return next;
}
//sobrecarga del operador <<
template <class T>
ostream& operator<<(ostream& os,const Node<T>& p)
{
    os << p.info << "->";
    return os;
}
// Queue
template <class T>
class Queue {
      NODE front;
      NODE rear;
public:
      // Inicializacion
      Queue();
      // Acceso y Modificacion
      void insert(const T&);
      T remove();
      Boolean empty()const;
      Boolean search(const T&);
      // Salida
        friend ostream& operator<<(ostream&,const
    Queue<T>&);
};
template <class T>
Queue<T>::Queue()          // Inicializando
{
    front=0;
    rear=0;
}
template <class T>
Boolean Queue<T>::empty()const
{
    return (rear==0);
}
// insertar bloque en la cola
template <class T>
void Queue<T>::insert(const T& a)
{
    NODE p=new Node<T>(a);
    if(rear==0)
        front=p;
    else
        rear->next=p;
    rear=p;
}
// sacar bloque de cola
template <class T>
T Queue<T>::remove()
{
    NODE p;
    p=front->next;
    T x=front->info;
    delete front;
    front=p;
    if(front==0)
        rear==0;
    return x;
}
// buscar si el bloque esta o no en la cola
template <class T>
Boolean Queue<T>::search(const T& p)
{
    NODE q=front;
    while(q)
        if(q->info==p)
            return TRUE;
        else
            q=q->getnext();
    return FALSE;
}
template <class T>
ostream& operator<<(ostream& os,const Queue<T>& p)
{
    if(p.empty())
        return os;
    NODE q=p.front;
    while(q) {
        os << *q ;
        q=q->getnext();
    }
    os << endl;
    return os;
}
#endif
```

Figura 7b. Listado del archivo *queue2.h*

```
// random.h
#ifndef  RANDOM_H
#define  RANDOM_H
#include  <math.h>
#include  "standart.h"
double  rnd()
{
     static int ix=1;
     ix*=899;
     if(ix<0)
          ix+=32767+1;
     return  ix/32768.0;
}
// metodo del cuadrado central
int midsqr(int& ix) // middle_square
{
     long int iy;
     iy=(long)ix*ix;
     iy/=100;
     ix=(int)iy%1000;
     return ix;
}
// metodo de la distribucion uniforme
double  unifr(float  a,float  b)
{
     return  a+rnd()*(b-a);
}
// metodo de la distribucion exponencial
double  expon(float  ex)
{
     double r;
     do {
          r=rnd();
     }while(r==0);
     return  -ex*log(r);
}
// metodo de la distribucion normal
double  normal(float  ex,float  std)
{
     double suma=0;
     for(int  i=1;i<=12;i++)
          suma+=rnd();
     return  ex+std*(suma-6.0);
}
// distribucion en forma polar
double polar(float ex,float std,double& x1,int& flag)
{
     double  r1,r2,v1,v2,w,y,pol;
     if(!flag)  {
          do {
               r1=rnd();
               r2=rnd();
               v1=2*r1-1;
               v2=2*r2-1;
               w=pow(v1,2)+pow(v2,2);
          }while(!(w<1 && w>0));
          y=sqrt(-2*log(w)/w);
          pol=v1*y;
          x1=v2*y;
          flag=TRUE;
     }
     else {
          pol=x1;
          flag=FALSE;
     }
     return  ex+std*pol;
}
double  empirica(float*  x,float*  fx)
{
     double  r=rnd();
     int i=0;
     while(r>=fx[i])
          i++;
     return x[i];
}
#endif
```

Figura 8. Listado del archivo *random.h*

```
// standart.h
#ifndef  STANDART_H
#define  STANDART_H

#define  NELEM(x)  sizeof(x)/sizeof(*x)
// definicion de un tipo enum para devolver valor de verdad o
     falso
enum  {FALSE,TRUE};        // FALSE=0, TRUE=1
class Boolean {       //clase  Boolean
     int boolean;
public :
     Boolean(int b=0){boolean=b!=0;} // constructor
     operator int(){return boolean!=0;}// sobrecarga al tipo int
};
// Comparacion
template  <class  T>
int Cmp(T a,T b)
{
      if (a<b)
            return  -1;
      else if (a==b)
            return  0;
      else
            return  1;
}
// El maximo
template  <class  T>
T Max(T a,T b)
{
     return  (a>b)?a:b;
}
// El minimo
template  <class  T>
T Min(T a,T b)
{
     return  (a<b)?a:b;
}
// Swaping: Algoritmo de intercambio
template  <class  T>
void SwapTo(T& a,T& b)
{
     T  temp=a;
     a=b;
     b=temp;
}
#endif
```

Figura 9. Listado del archivo *standart.h*

ANÁLISIS Y DISCUSIÓN DE RESULTADOS

A continuación se presenta una corrida del programa para los datos nmemoria = 64 000 y ncache = 128

Figura 10. Una corrida del programa *cache.cpp*

El programa ha sido corrido para un determinado tamaño de memoria, variando el tamaño de la *cache*. Los resultados se muestran en los cuadros 2, 3 y 4.

Cuadro 2. Análisis de Resultados (64000 Mb)

Tamaño de Memoria RAM	Tamaño de Cache	% de éxitos	Tiempo de Acceso
64000	32	40.92	6.317 µ seg.
64000	64	82.26	2.597 µ seg.
64000	128	98.46	1.139 µ seg.
64000	256	98.46	1.139 µ seg.
64000	512	98.46	1.139 µ seg.

Cuadro 3. Análisis de Resultados (128000 Mb)

Tamaño de Memoria RAM	Tamaño de Cache	% de éxitos	Tiempo de Acceso
128000	32	20.48	8.157 µ seg.
128000	64	40.26	6.377 µ seg.
128000	128	80.24	2.778 µ seg.
128000	256	96.92	1.277 µ seg.
128000	512	96.92	1.277 µ seg.
128000	1024	96.92	1.277 µ seg.

Cuadro 3. Análisis de Resultados (256000 Mb)

Tamaño de Memoria RAM	Tamaño de Cache	% de éxitos	Tiempo de Acceso
256000	32	10.32	9.071 µ seg.
256000	64	19.96	8.204 µ seg.
256000	128	40.16	6.386 µ seg.
256000	256	79.14	2.877 µ seg.
256000	512	93.84	1.554 µ seg.
256000	1024	93.84	1.554 µ seg.
256000	2048	93.84	1.554 µ seg.

Como puede verse en las tablas de resultados se encuentra que para un determinado tamaño de memoria RAM, existe un tamaño de *cache* "óptimo" a partir del cual el porcentaje de éxitos y el tiempo de acceso medio del sistema no varía; es decir, permanece constante aunque se aumente el tamaño de la *cache*. Dichos resultados a modo de resumen se presentan en la Cuadro 5.

Cuadro 5. Resumen de resultados óptimos de la cache obtenidos con el programa

Tamaño de Memoria	Tamaño de Cache optimo	% de éxitos	Tiempo de Acceso
64000	128	98.46	1.139 µ seg.
128000	256	96.92	1.277 µ seg.
256000	512	93.84	1.554 µ seg.

A decir verdad, el tamaño de la *cache* depende de otras variables. como el costo por *bit* para implementar en la tarjeta madre. Sin embargo, el programa presentado da una buena aproximación de las cuestiones teóricas inmersas en el diseño.

CONCLUSIONES Y RECOMENDACIONES

El programa de simulación de la *cache* implementado muestra que para un determinado tamaño de memoria RAM, existe un tamaño de memoria *cache* óptimo, asociado por encima del cual el porcentaje de éxitos no mejora. De allí que no es conveniente aumentar el tamaño de la cache por encima del óptimo pues no se consigue mejora en el porcentaje de éxitos ni en el tiempo de acceso y el costo de implementar un tamaño de *cache* mayor en la tarjeta madre aumenta considerablemente.

El programa puede también utilizarse para probar con más de una cache utilizando la fórmula adecuada para los aciertos y los tiempos medios de acceso a cada uno de ellos.

BIBLIOGRAFÍA

1. **Raffo Lecca, Eduardo. (1998).** Algoritmos y Estructuras de Datos con C++. 1ra Edic. Raffo Lecca Editores. Lima, Perú.

2. **Stallings, William. (2003).** Computer Organization and Architecture: Designing for Performance. 6ta Edic. Editorial Prentice Hall International, USA.

3. **Stroustrup, Bjarne. (2002).** El Lenguaje de Programación C++ Edición especial. 3ra Edic. España. Addison Wesley - Pearson Educación S.A. España.