Estudo do efecto do principio de localidade nos accesos a memoria caché

Pablo Martínez Rodríguez

Pablo Rey Fernández

Table of Contents

1. - Introducción

Coas arquitecturas dos microprocesadores e dos sistemas informáticos en xeral dos que dispoñemos actualmente, faise evidente a importancia de conceptos como a localidade espacial dos datos en memoria como fundamento para a mellora do rendemento en operacións de calquera ámbito. Dacordo ás distintas tecnoloxías de fabricación, os dispositivos de almacenamento presentes na actualidade presentan un moi amplo abano de prestacións nas que é fundamental destacar a relación entre a capacidade e os tempos de acceso (ben para escritura ou ben para lectura) que ofrece cada un. En xeral, aqueles dispositivos que ofrecen capacidades moi grandes, discos duros mecánicos formados por discos superpostos que son lidos mediante un cabezal móbil, penalizan en canto ao tempo de acceso aos datos pero presentan unha boa realación entre a capacidade e o custo. Doutra banda, os dispositivos baseados en tecnoloxías SRAM ou DRAM presentan un alto custo pero os seus tempos de acceso melloran en gran medida aos de aqueles dispositivos que ofrecen maiores capacidades. Coa fin de obter sistemas informáticos con capacidades que permitan traballar con importantes cantidades de información e, do mesmo xeito, tendo en conta que un dos principais freos no rendemento dos sistemas é o tempo de espera cando se realiza un acceso a disco, as arquitecturas actuais deseñan a organización dos dispositvos de memoria dos computadores dacordo a unha estrutura xerárquica como a mostrada na figura 1.

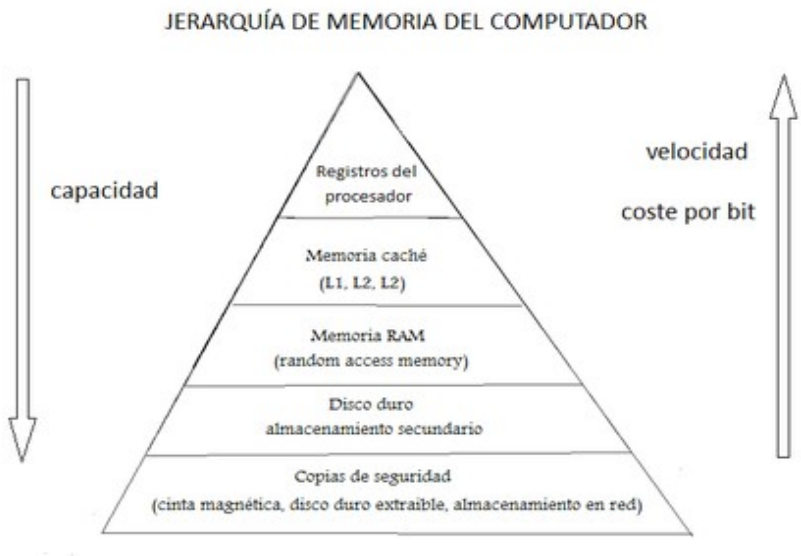

Figura 1: Xerarquía de memoria

A división en niveis deste xeito da memoria dun computador actual é eficiente e pode darse grazas ao principio de localidade. Ademais, é fundamental para manter bos tempos de execución dado que os microprocesadores operan a unhas velocidades moito maiores que a memoria e esta non pode ler instrucións a un ritmo que permita manter ao procesador ocupado en todo momento. O principio de localidade é un termo experimental que nace froito das estruturas de datos e do xeito de programar que se empregan de forma predominante. En particular, o principio de localidade espacial enuncia que, se nun momento dado se fai unha referencia a un elemento en memoria, entón é altamente probable que se vaian facer referencias a posicións achegadas no futuro próximo. Se durante a súa execución un programa require a lectura ou a escritura dun dato e procesador non ten unha copia do mesmo nalgún dos seus rexistros, entón traerase unha copia da caché. Se o dato tampouco está na memoria caché, pedirase á memoria principal, e así sucesivamente ata chegar ao disco. A vantaxe que se extrae do principio de localidade é que, se ao trasladar unha copia dun dato dun nivel inferior a outro superior levamos tamén copias de datos almacenados en posicións contiguas, co paso do tempo aforraremos accesos a memoria de niveis inferiores, operacións que adoito consumen moitos ciclos de reloxo, e conseguiremos obter así ciclos de reloxo útiles cos que se poderá mellorar o rendemento na execución das aplicacións.

O principio de localidade dá pé a diversas tecnoloxías de optimización como son a caché, a precarga ou o predictor de saltos.

1.1 - Obxectivos

O obxectivo principal desta primeira práctica da asignatura de Fundamentos de Computadores é o de observar o comportamento das memorias caché do microprocesador tendo en conta o principio de localidade e os mecanismos de precarga. Para poder experimentar nas condicións dadas débese ter en conta que nas arquitecturas modernas o normal é que as cachés veñan integradas no procesador e organizadas en varios niveis. As cachés divídense en liñas, unidades indivisibles que se cargan completamente unha vez se requira algún dos seus elementos. Este feito, así como a posibilidade de

observar algún tipo de precarga é algo que está directamente relacionado co principio de localidade espacial exposto anteriormente e que responde á motivación da práctica.

1.2 - Procedemento

Para facer as experimentacións desenvolverase un programa na linguaxe C no que se sumarán distintos elementos dun vector de números de tipo *double* variando o patrón de acceso e o número de datos do vector. Compararanse os tempos entre distintas execucións con parámetros variables coa fin de comprobar os efectos do principio de localidade e dos mecanismos de precarga dun sistema coma os dos laboratorios.

Os equipos nos que se realizarán as probas están equipados cun procesador Intel Core 2 Quad Processor Q9550 a 2.83GHz e as probas fanse nun entorno con kernel Linux versión 3.5.0-37. Dacordo a información obtida na ruta */sys/devices/system/cpu/* dispoñemos dunha caché de tres niveis L1, L2 e L3 (*Data*, *Instructions*, *Unified*) por cada unha das 4 CPU do procesador. Para a caché de nivel 1 indícase un tamaño de liña de 64 bytes e un tamaño total de 32KB logo a caché de nivel 1 ten 512 liñas. No caso da caché de nivel 2 mantense o tamaño de liña, pero o tamaño máximo é agora de 6144KB (∼6MB) polo que dispoñemos de 98304 liñas de caché deste nivel. Tanto nun caso coma noutro, nunha liña poderemos colocar 8 valores de tipo *double*.

- Definición das variables:

- L : representa o número de liñas caché que serán lidas. Virá determinado por un factor que multiplicará ao tamaño de liña das caché de primeiro e segundo nivel.
- D : representa o valor enteiro que define o tamaño dos saltos entre as seleccións de datos. Varía entre 1 e 100 e recibirase como argumento. E necesario que os distintos valores de D non sexan próximos entre si.
- R : número de accesos ao vector determinado en función do valor de D. No caso de que D sexa menor que 8 calculase da seguinte forma $R.D \geq L.T$ sendo $L = x.512$ e T o número de elementos que entran

nunha liña cache. Neste caso é 8 polo que $R \geq \frac{L.T}{D}$. Isto é así debido a que hay que garantir que sempre se fan L saltos. No caso contrario $R=L$ debido a que se non se fixese deste xeito o número de accesos que se realizasen sería menor que o número de liñas que deben ser lidas. Así garantizamos o acceso a L liñas.

- Vector A : é vector que conten os elementos que serán sumados. Ten un tamaño de $R \cdot D$, xa que se realizan R accesos saltando de D en D elementos. O vector está inicializado con *doubles* aleatorios entre 1 e 2.
- Vector e : é o vector que almacena os índices dos elementos do vector A aos que se debe acceder en cada execución. Ten un tamaño de 10 · R e está composto por 10 secuencias de R indices da forma 0, D, 2D, 3D, etc.

Unha vez realizadas as probas con distintos parámetros R e D, realizarase unha segunda experimentación variando as condicións. A idea clave nesta segunda proba será a de ter no vector e secuencias aleatorias de índices, é dicir, facer accesos non secuencias aos elementos do vector A.

2. - Desenvolvemento das probas

Para a realización de cada un dos dous apartados da práctica desenvolvéronse dos códigos en linguaxe C independentes. No primeiro código os accesos ao vector onde se almacenan os valores para a suma fanse de xeito secuencial dacordo aos valores do parámetro D introducidos por liña de comandos. No apartado 2, os accesos realízanse de forma aleatoria, enchendo o vector e de valores múltiplos de D en orde aleatoria.

2.1 – Apartado 1

```
/*** main.c ***/

#include "rutinas_clock.h"
```

```c
#include "time.h"

// Macro definida para xestionar os erros de execución
#define handle_error(msg) \
   do { perror(msg); exit(EXIT_FAILURE); } while (0)

// Función para xerar un double aleatorio entre 1 e 2
double genera_aleatorio() {
   double min = 1.0;
   double max = 2.0;
   double rango = max - min;

   double random = (double)rand()/(double)RAND_MAX;

   return random * rango + min;
}

int main(int argc, char *argv[]) {
      // Control dos parámetros de entrada
      // e asignación das variables D e L
      if(argc != 3 || atoi(argv[1]) < 1 || atoll(argv[2]) < 1)
            handle_error("uso_correcto: test <D> <L>");

      int D = atoi(argv[1]);
      double L = atoll(argv[2]);

      // Asignación da variable R que define o número de
      // accesos ao vector A dacordo aos parámetros de entrada
```

```
int R;
if(D < 8)
     R = (int)(((L*8)/D)+1);
else
     R = L;

// Apertura do arquivo onde gardaremos os resultados coa
// opción 'append' para ir engadindo ao mesmo os resultados de
// cada execución
FILE *fp;
if((fp = fopen("puntos.txt","a+")) == NULL)
     handle_error("na apertura do arquivo");

// Declaración e inicialización das variables e, S, A e da
// variable que servirá de contador para medir os ciclos
int *e=(int *)malloc(10*R*sizeof(int));
double S[10];
double ck=0;
double *A = NULL;
A = _mm_malloc(R*D*8,512);
 int i=0,j=0,k=0,z=0;

// Fase de quencemento. Realízase unha chamada á función
// srand co parámetro tempo actual co fin de obter
// distintos valores entre execucións e inicialización
// dos elementos do vector con valores aleatorios
 srand((unsigned int)time(NULL));
```

```
for(z=0; z<R*D;z++)
     *(A+z) = genera_aleatorio();

// Xeración do patrón de acceso aos datos. Neste caso,
// xéranse 10 secuencias iguais de índices que van dende
// 0 ata R
e[0]=0;
for(i=1;i<10*R;i++){
     if(i%10==0)
          k++;
     e[i]=k*D;
}
// Bucle principal de computación da suma dos elementos
// seleccionados segundo o vector e. En cada execución
// realízanse e mídese o tempo de 10 operacións completas
// deste tipo.
for(i=0;i<10;i++){
     // Suma dos elementos nas posicións seleccionadas
     // polo vector e (cada R índices). Cómputo do tempo
     // consumido na variable ck e gardado do resultado
     // da suma no índice que lle corresponde.
     double sum = 0;
     start_counter();
     for(j=0;j<R*10;j=j+10)
          sum += *(A+e[j]);
     ck=get_counter();
     S[i]=sum;
```

```
            // Impresión dos resultados no arquivo de texto
            fprintf(fp,"%f  ", ck/(R));
            fprintf(fp,"%f  ", L);
            fprintf(fp,"%d\n", D);
      }

      mhz(1, 1);
      // Impresión dos resultados co fin de evitar
      // optimizacións do compilador
      for(i=0;i<10;i++)
            printf("Resultado %d: %f\n", i+1,S[i]);

      // Liberación da memoria ocupada polas variables A e e
      _mm_free(A);
      free(e);

      return 0;
}
```

2.2 – Apartado 2

As diferenzas respecto ao apartado 1 están na xeración dos elementos do vector *e*, xa que agora interesa sumar elementos do vector *A* seleccionados de forma aleatoria. Para acadar isto desenvolveuse a seguinte función para a mistura aleatoria dos índices unha vez xerados:

```
void mistura_indices(int *array, int tam) {
      if(tam > 1) {
      int i;
```

```
    for(i=0; i<(tam-1); i++) {
        int j = i + rand() / (RAND_MAX / (tam-i)+1);
        int t = array[j];
        array[j] = array[i];
        array[i] = t;
    }
  }
}
```

O proceso de asignación tamén variou agora xa que *k*, o factor polo que multiplicábamos D para obter os índices e que se ía incrementando en un en cada iteración para obter acceso secuencial defínese agora coma un array de enteiros definidos entre 0 e R e que se misturan coa chamada á función *mistura_indices()*:

```
/*** apartado2.c ***/
for(i=0;i<R;i++)
    *(k+i)=i;
mistura_indices(k, R);
j = -1;
for(i=0; i<10*R; i++) {
    if(i%10==0) {
    j++;
    }
    e[i]=k[j]*D;
}
```

O resto do código é totalmente análogo ao do primeiro apartado.

2.3 – Códigos auxiliares

Para a automatización das probas desenvolveuse un script en linguaxe BASH no que se declaran e inicializan os vectores que almacenarán os sucesivos valores de L e D a pasar como argumento en cada execución do programa. Ademais, escribiuse un Makefile no que se definen os arquivos necesarios para a compilación do programa e as regras de compilación a especificar para que GCC no faga optimizacións non desexadas.

- Script de automatización:

```
#!/bin/bash

L=(256 768 49152 73728 196608 393216 786432)

D=(5 29 53 77 95)


rm puntos.txt


for i in "${L[@]}"; do

     for j in "${D[@]}"; do

          echo -e "\n\n*** Prueba con $j, $i ***"

          ./test $j $i

     done

done
```

- Makefile:

```
CC = gcc -Wall -O0 -msse3

MAIN = test

SRCS = apartado2.c rutinas_clock.c

DEPS = rutinas_clock.h
```

```
OBJS = $(SRCS:.c=.o)


$(MAIN): $(OBJS)
	rm -f $(MAIN)
	rm -rf objects/
	$(CC) -o $(MAIN) $(OBJS)
	mkdir objects/
	mv *.o objects/


%.o: %.c $(DEPS)
	$(CC) -c $<


clean:
	rm -f $(MAIN)
	rm -rf objects/
```

Ademais dos códigos reproducidos no documento, utilizouse un pequeno código en C para o tratamento dos resultados almacenados no arquivo de texto co fin de obter arquivos en formato *.csv* interpretables por programas de tratamento de follas de cálculo dende os que se xeraron as gráficas.

3 – Resultados e conclusións

Para poder analizar os resultados, dos 10 tempos obtidos por cada unha das 35 execucións do programa realizadas a través do script, elexiuse como mostra representativa a media xeométrica dos tres mellores tempos de cada execución. Os resultados obtidos cando os accesos ao vector A se fan de xeito secuencial represéntanse na seguinte gráfica:

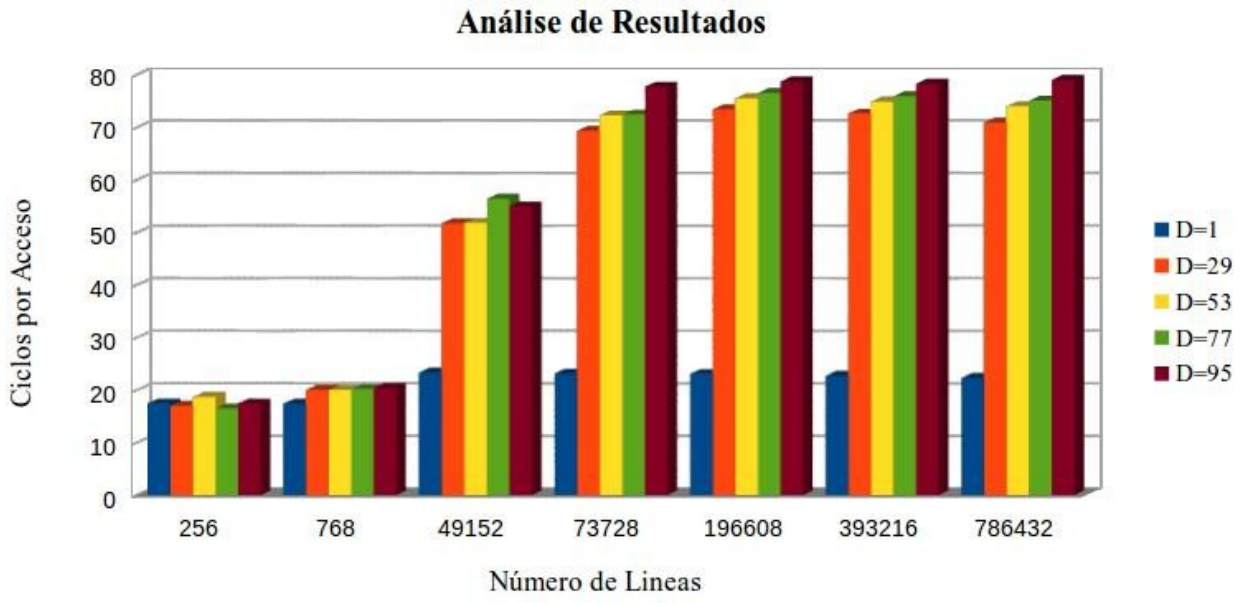

Figura 2: Resultados para as probas con accesos secuenciais

Na gráfica pode apreciarse o efecto da precarga de liñas. Para unha *D* moi baixa case todos os datos encóntranse na caché polo que, a diferencia dotros valores de *D*, cando aumentamos o número de liñas o tempo de acceso varía en moi pouca medida.

Cando maior é o crecemento dos ciclos por acceso é no salto de 768 liñas a 49152. Isto débese a que neste punto a caché de nivel 1 e de nivel 2 teñen todas as súas liñas ocupadas, polo que neste momento entrán en xogo os algoritmos de remplazo. Como para unha *D* pequena o efecto do principio de localidade e a precarga fan que os ciclos por acceso non aumenten de forma tan brusca como se sucede con *D* maiores. Neste caso, ao non cumprirse o principio de localidade e a precarga non facer efecto, é necesario remprazar moitas máis liñas polo que os ciclos por acceso aumentan considerablemente.

Os resultados cando o acceso aos elementos do vector *A* se realizan de xeito aleatorio móstranse a continuación:

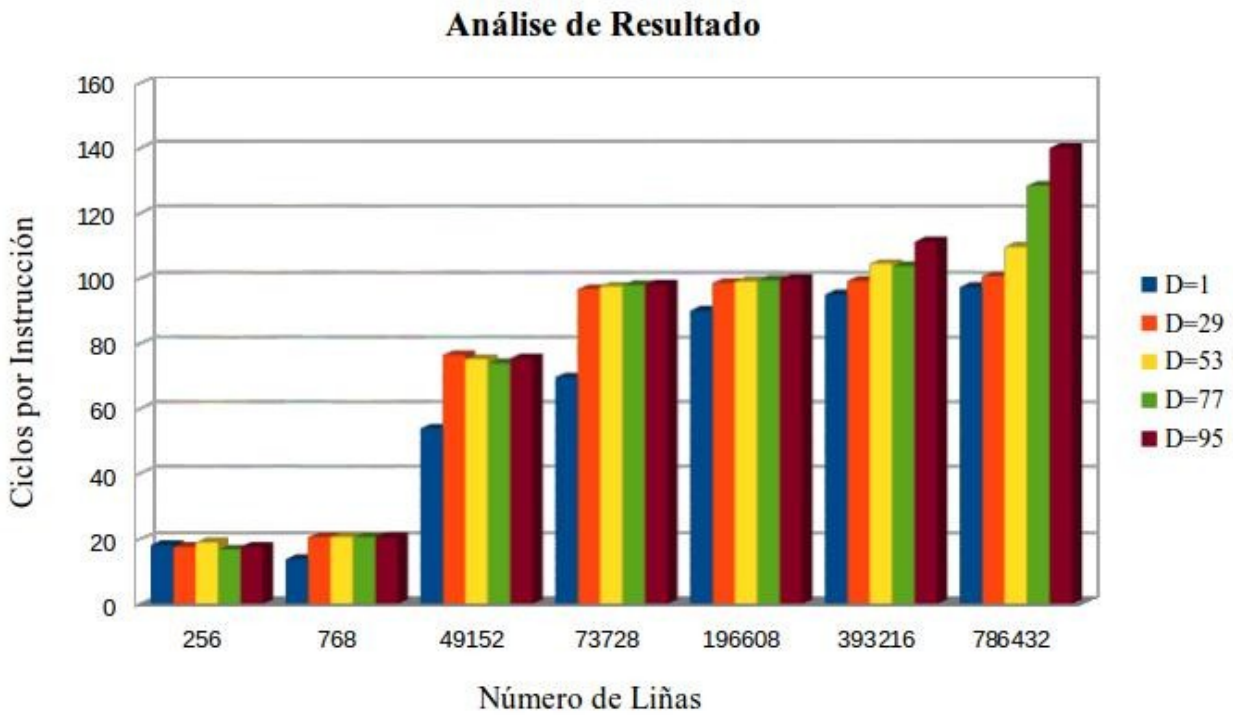

Figura 3: Resultados para as probas con accesos aleatorios

Neste caso os accesos son a valores seleccionados de xeito aleatorio, polo que o principio de localidade e a precarga non teñen por que facer efecto. Coma no caso anterior, para os primeiros valores de liñas a acceder, os ciclos por instrucción non varían demasiado, pero, cando a cache está chea e entran en xogo os algoritmos de remplazo, a gran diferenza con respecto ao anterior apartado é o valor que alcanza os ciclos por instrucción con $D=1$. Neste caso o principio de localidade e a precaga non melloran o rendemento polo que, do mesmo xeito que sucede para o resto de valores de *D,* o procesador debe esperar a que se fagan os remprazos para poder acceder a un dato.

Bibliografía e enlaces

Hennesy, Patterson: *Computer Architecture, Fifth Edition,* Apéndice B

Wikipedia.org: *Memory Hierarchy*

https://en.wikipedia.org/wiki/Memory_hierarchy

FutureChips.org: *Prefetching*

http://www.futurechips.org/chip-design-for-all/prefetching.html

Lee, J., Kim, H., and Vuduc*, R. 2012. When prefetching works, when it doesn't, and why. ACM Trans.* Archit. Code Optim. 9, 1, Article 2 (March 2012), 29 pages
http://www.cc.gatech.edu/~hyesoon/lee_taco12.pdf

Figura 1

https://es.wikipedia.org/wiki/Jerarqu%C3%ADa_de_memoria