Efecto de la localidad de los accesos a memoria caché en las prestaciones de programas en microprocesadores

Xiana Carrera Alonso, Ana Carsi González

Arquitectura de Computadores
Grupo 01
xiana.carrera@rai.usc.es | ana.carsi@rai.usc.es

25 de marzo de 2022

Resumen

Con el propósito de minimizar la latencia en los accesos a memoria, múltiples estudios han tenido por objetivo identificar las métricas del rendimiento caché más cruciales. Se ha comprobado que factores como la localidad en el acceso, la organización de la jerarquía de memoria y la precarga tienen impacto sobre la eficiencia. Para esclarecer este aspecto y demostrar su significación, este trabajo estudia las consecuencias de la elección del patrón de acceso a los elementos de un vector N-dimensional en el coste temporal de lectura, así como los efectos de parámetros físicos del procesador y de la memoria cache.

Palabras clave: Jerarquía de memoria, caché, localidad temporal, localidad espacial, precarga (prefetching), eficiencia, fallo caché, reducción...

I. Introducción

En este documento se expone un análisis acerca de la influencia de la localidad de las referencias a datos sobre los tiempos de acceso a memoria caché. En concreto, se estudian las alteraciones causadas por distintos grados de localidad espacial y temporal en el acceso a los elementos de un array de datos en punto flotante para la realización de una operación de reducción sobre el mismo.

En la evaluación de los resultados han resultado fundamentales conceptos tales como la organización de la jerarquía de memoria y, en particular, de la memoria caché (para lo cual se ha empleado como referencia el manual de la asignatura, [1], así como el libro de consulta de Fundamentos de Computadores, [2]). Asimismo, ha sido preciso comprender la naturaleza de los fallos caché ([3]), los fundamentos del principio de localidad (con particular énfasis en la localidad espacial, siendo de especial interés como referencia el análisis de [4]), las propiedades de las operaciones de precarga (en base a las indicaciones al respecto de [1]) y los efectos provocados por distintas opciones de compilación del código (véase [5]).

En última instancia, el objetivo será determinar en qué medida afectan distintos parámetros del programa, tales como la separación de los sumandos, el tamaño del array, el número de líneas a acceder, el medio de ejecución de las pruebas, etc., a la eficiencia del código. Todo ello deberá ser examinado bajo la perspectiva de las especificaciones técnicas del equipo de pruebas. Se empleará una metodología basada en la repetición de experimentos para diferentes condiciones iniciales, su filtrado a través de la eliminación de valores atípicos y la interpretación de los resultados finales de forma gráfica.

La estructura del documento comienza, en la sección *II. Metodología de las pruebas*, por una descripción de la organización estática del programa, indicando el significado y las características de sus diferentes componentes, y justificando las decisiones que tengan impacto ulterior en las mediciones. A continuación, en la sección *III. Ejecución*, se describen

las circunstancias dinámicas de realización de las pruebas, esto es, las características del equipo empleado, las condiciones del entorno de ejecución y cualquier elección peculiar al respecto, así como los efectos de la técnica de precarga empleada por el procesador. Se da paso entonces a *IV. Resultados*, donde se aplican los conocimientos teóricos adquiridos en la interpretación de los valores obtenidos, utilizando como apoyo diferentes representaciones gráficas. Finalmente, en *V. Conclusiones*, se expone un breve resumen del estudio, de las deducciones realizadas a partir de él, y se indican posibles líneas de investigación futuras.

II. Metodología de las pruebas

En esta sección se describe el problema planteado y la organización de los programas desarrollados para afrontarlo. Asimismo, se indican y describen distintas fórmulas que fueron deducidas de las condiciones de partida y que son empleadas para determinar valores de parámetros intrínsecos al problema, variables a determinar según el objetivo del experimento para la ejecución.

A. Descripción del problema

El factor clave de la implementación del problema gira alrededor de la realización de una operación de reducción de suma de punto flotante sobre R elementos de un vector de *doubles* A[] con tamaño N, estando el acceso a estos determinado por un parámetro D de forma que las referencias siguen la sucesión A[0], A[D], A[$2 * D$], ..., A[$(R-1) * D$]. Otro parámetro de entrada a considerar es el número de líneas caché distintas (L) que se desea leer al emplear los elementos del vector.

En total, se realizan 35 experimentos, con 5 valores distintos de D (a elegir por los programadores, en el rango de enteros entre 1 y 100) y 7 de L: $0{,}5 * S1$, $1{,}5 * S1$, $0{,}5 * S2$, $0{,}75 * S2$, $2 * S2$, $4 * S2$, $8 * S2$, siendo S1 y S2 el número de líneas caché que caben en la caché L1 y L2 de datos respectivamente. Cada uno de estos experimentos se repite 10 veces para medir el número de ciclos de reloj precisados y, a continuación, se calcula la mediana de dichas 10 pruebas, de forma que finalmente se obtienen 35 valores, uno por cada par (D, L).

Además, en cada una de las 350 pruebas totales la operación de reducción se lleva a efecto 10 veces (lo cual incrementará la uniformizad en la cronometración de los resultados, pero también será un factor de análisis sobre los mismos). Los totales de la reducción se almacenan en un vector S[] sobre el cual se calcula la media y del que se imprimen todos los elementos, a fin de evitar que la compilación ignore operaciones o realice cambios que puedan alterar las condiciones de los ensayos.

El número de sumandos de cada prueba, R, es un valor determinado por la elección de D y L. Se remiten los detalles de su obtención al subapartado B.

Por otra parte, la referencia a los elementos a sumar de A[] no es directa, sino que ocurre a través de un vector de índices, ind[].

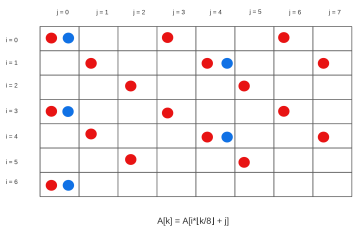
Nótese que el resultado de cada prueba no es el número de ciclos de reloj de las 10 reducciones asociadas, sino el número de ciclos por acceso a cada elemento. Si denotamos como ck el tiempo total en ciclos y como ck_acceso el tiempo en ciclos por acceso, se tiene:

$$ck\_acceso = \frac{ck}{10 * R},$$

puesto que por cada reducción se leen R elementos de A[].

B. Obtención de R

Figura 1: *Representación gráfica de la dependencia de R del rango en el que se encuentra D para el equipo de pruebas: [1, 7] u [8, 100]. Empleando L=7, en rojo se muestran los accesos necesarios cuando D=3; en azul, cuando D=12.*



Para calcular el valor del parámetro R en un experimento dado (esto es, con el par (D, L) fijado), se ha empleado la siguiente fórmula:

$$R = \begin{cases} \lceil \frac{B*(L-1)}{D*sizeof(double)} \rceil & D < \frac{B}{sizeof(double)} \\ L & D \geq \frac{B}{sizeof(double)} \end{cases}$$

El motivo de la distinción de dos regiones para D se fundamenta en la diferencia que supone que el espacio entre elementos de A sea menor o mayor que el número de *doubles* que caben en una línea caché, $\frac{B}{sizeof(double)}$. En el equipo en el que se han realizado las pruebas, se tiene que $\frac{B}{sizeof(double)} = \frac{64 \text{ bytes}}{8 \text{ bytes}} = 8$ (véase el subapartado A de *III. Ejecución*).

Supóngase que los elementos del vector A[] se encuentran en líneas caché consecutivas, uno detrás de otro, como se indica en la figura 1.

Si $D > 8$, podrá haber líneas enteras de A[] a las que no se acceda. Por otro lado, en aquellas a las que sí se accede únicamente se leerá un elemento. Dado que el número de líneas accedidas es, por definición, L, este será exactamente el valor de R, sin influencia alguna de D.

De forma similar, si $D = 8$, se accederá a un y solo un elemento de cada línea. Por consiguiente, se deberán tomar tantos sumandos como líneas: $R = L$.

Ahora bien, si $D < 8$, podrá seleccionarse más de un dato por línea. Se deberá dividir el espacio en bloques de longitud D. No obstante, dicha longitud puede no ser divisora exacta de 8. En consecuencia, se tendrá que emplear la función ceiling() para asegurar que también se incluyen los bloques "interlínea". De hecho, se puede afinar aún más con respecto a la última línea, debido a que siempre hará falta únicamente un dato para asegurar que se accede a ella. De ahí que únicamente se multiplique por L-1 dentro de ceiling() y se sume 1 *a posteriori*.

C. Obtención de N

El tamaño total del array, N, se calculará en función de R y D:

$$N = D * (R - 1) + 1$$

De nuevo, se puede visualizar su significado pensando en bloques de D elementos. Habrá un número R de dichos bloques, con la salvedad de que en el último de ellos únicamente hace falta incluir un elemento (el primero, que será el R-ésimo, ya que no se precisará leer ninguno de los siguientes). En consecuencia, se definirá un array de tamaño $D * (R - 1)$ con un elemento extra.

D. Estructura del programa

La estructura del programa se ha dividido en 4 componentes:

- **reduccion.c**, el programa principal. Su propósito es llevar a cabo un experimento para el que se han fijado los valores de D y L, que se introducen como argumentos. Tras ejecutar la reducción, medir sus ciclos y calcular el tiempo por acceso, escribe los resultados en un archivo temporal. Las funciones referentes al cronometrado se encuentran en la librería *counter.h* y fueron proporcionadas como material de la asignatura.
- **mediana.c**, un programa de filtro de los resultados. Este actúa sobre 10 medidas obtenidas a partir de *reduccion.c* utilizando los mismos valores de D y L. El objetivo es calcular la mediana de esas medidas, eliminando los posibles valores atípicos. Asimismo, se logra centralizar los resultados en un solo dato para cada par (D, L). Estos valores se escriben sobre un archivo de resultados definitivos.
- **bash_script.sh**, un script de bash que automatiza la ejecución. Tras preseleccionar manualmente las 5 constantes a utilizar como D, se itera sobre las 7 constantes de L especificadas en el estudio. Para cada par (D, L), se llama al programa de *reduccion* 10 veces. Tras las $5 * 7 * 10 = 350$ pruebas, se habrá completado el registro del archivo temporal. En ese punto, se llama a *mediana* para obtener los resultados definitivos.
- **graficas.R**, un script de R de visualización de datos. A través de él, se representan diferentes medidas y relaciones del programa en forma de 3 gráficas 2D y 2 gráficas 3D, con el propósito de ayu-

dar y servir de soporte en el análisis y la comprensión del estudio.

A continuación, se exponen las características más significativas de cada una de las componentes a las que se ha hecho referencia:

D.1. reduccion.c

Código asociado en A.

En primer lugar, cabe destacar que la reserva de memoria dinámica se ha realizado con la función _mm_malloc() para alinear el inicio del vector con el comienzo de la línea caché y asegurar el correcto estudio de la localidad.

El vector A[] se inicializa con valores aleatorios en el conjunto $(-2,-1]\bigcup[1,2)$. De esta forma, se tiene la seguridad de que antes de iniciar las pruebas reales ya hay datos previamente cargados en los distintos niveles de la jerarquía de memoria (caché, TLBs, etc.), evitando posibles alteraciones en las mediciones por estar estos vacíos y precisarse acciones adicionales. Con esta precaución, se asegura que cada línea se traerá en las mismas condiciones: en las pruebas no habrá fallos obligatorios ([2], [3]), pues todos y cada uno de los elementos de A[] ya habrán sido referenciados con anterioridad. Únicamente se consideran fallos de capacidad y de conflicto.

Dado que en la mayoría de equipos el tamaño de línea es invariante entre cachés, se ha operado bajo el supuesto de que esto es cierto en general. En caso contrario, nótese que sería necesario realizar un refinamiento sobre el cálculo de parámetros.

Se ha implementado una función que tiene por objetivo desordenar los elementos del vector ind[], que almacena los índices del vector A[] correspondientes a los elementos a emplear en la reducción. La justificación completa y el análisis de los resultados se remiten a IV. Es importante destacar que para asegurar que el vector ind[] realmente es aleatorio, se repite el proceso de desorden hasta que satisfaga los estándares admitidos (se compara con el vector ordenado y se vuelve a desordenar si el porcentaje de coincidencias no es inferior al 15 %). El código de randomización fue desarrollado por Ben Plaff ([7]), y se trata de un refinamiento del algoritmo de Fisher-Yates.

D.2. mediana.c

Código asociado en B.

Se ha elegido la mediana como medida de centralización de los datos. El motivo fundamental subyace en su robustez (esto es, que se vea afectada en baja medida por valores atípicos) frente a otros estimadores como podría ser la media. De esta forma, se minimiza el efecto de las interferencias que pudieran tener otros procesos, picos o caídas en la frecuencia del procesador, etc. Aplicar la mediana asegura que el valor obtenido para unas ciertas condiciones iniciales tiene un alto grado de fiabilidad.

Al ser el número de ejecuciones para cada dupla (D, L) par (10 mediciones), ha tenido que aplicarse una corrección para el cálculo. Se ha optado por interpolar los dos valores centrales y emplear entonces la media de ambos. Esta sigue siendo una opción bastante segura y estándar estadísticamente, pues al haber ordenado previamente los datos se continúan desechando los resultados extremos.

En este programa se emplea el algoritmo de inserción para ordenar los datos. La elección buscar optimizar el tiempo de ejecución teniendo en cuenta que se trabaja con arrays de tamaño reducido ([8]).

D.3. bash_script.sh

Código asociado en C.

Las 10 ejecuciones para el mismo par (D, L) no se efectúan de manera consecutiva, sino que se intercalan con otros valores. La explicación de esta decisión se remite al apartado F.

D.4. graficas.R

Código asociado en E.

Se ha elegido emplear R como lenguaje para la representación de los datos por su amplio abanico de posibilidades para la visualización. En el código se emplean las librerías gráficas *plotly*, *plot3D* y *rgl*, que deben estar instaladas de antemano.

III. Ejecución

En esta sección se especifican las características técnicas del procesador y de la jerarquía de memoria del equipo utilizado para la ejecución del código asociado a la práctica, se comentan las condiciones impuestas sobre el entorno externo al programa durante los experimentos, y se ofrece un comentario sobre la precarga tanto de forma teórica como con respecto al procesador en cuestión.

A. Datos técnicos del equipo de medición

En lo referente al hardware utilizado, las pruebas se han realizado en un Lenovo V130-15IKB con las siguientes especificaciones:

- Procesador: Intel® Core™ i7-7500U de cuatro núcleos y 2.70 GHz.
- Sistema Operativo: Ubuntu 20.04.4 LTS.
- Tipo de Sistema Operativo: 64 bits.
- Kernel: 5.13.0-35-generic.

Cabe resaltar que se trata de un ordenador portátil, y que su procesador es de 7ª generación y de bajo consumo (U), esto es, su diseño antepone la duración de la batería al rendimiento general.

Cuadro 1: *Características técnicas de la caché del equipo y consecuentes valores de L.*
L1 I y L1 D corresponden a las cachés L1 de instrucciones y datos, respectivamente.
S_i representa el número de lineas caché de cada nivel.

caché	Tamaño	Tamaño de línea
L1 I	2^{15} bytes	64 bytes
L1 D	2^{15} bytes	64 bytes
L2	2^{18} bytes	64 bytes
L3	2^{22} bytes	64 bytes
caché	S_i	**Tipo de caché**
L1 I	$512 = 2^9$	Asociativa por conjuntos
L1 D	$512 = 2^9$	Asociativa por conjuntos
L2	$4096 = 2^{12}$	Asociativa por conjuntos
L3	$65536 = 2^{16}$	Asociativa por conjuntos
caché	**Nº de vías**	**Nº de conjuntos**
L1 I	8	64
L1 D	8	64
L2	4	1024
L3	16	4096

Se han comprobado también los diferentes atributos de la jerarquía de memoria del equipo. Este dispone de una memoria RAM de 7.1 GiB y de 3 niveles de caché, todos ellos totalmente asociativos. Existen varias formas de comprobar sus detalles técnicos, como puede ser a través del Administrador de Tareas de Windows, con el programa CPU-Z, etc. Se ha optado por emplear la función sysconf() en un programa de C especialmente dedicado a este propósito (*info_cache.c*, que se incluye como el apartado D del apéndice).

sysconf() permite corroborar el número de vías (por ejemplo, para la L2, con _SC_LEVEL2_caché_ASSOC), el tamaño total en bytes (_SC_LEVEL2_caché_SIZE) y el tamaño de línea en bytes (_SC_LEVEL2_caché_LINESIZE). Así, empleando que el número de líneas (S1, S2) será igual al tamaño total en bytes dividido por el tamaño de línea en bytes, se tiene:

$$B = \text{sysconf(\_SC\_LEVEL1\_Dcaché\_LINESIZE)}$$

o

$$B = \text{sysconf(\_SC\_LEVEL2\_caché\_LINESIZE)},$$

y

$$S1 = \text{sysconf(\_SC\_LEVEL1\_Dcaché\_SIZE)} * \frac{1}{B}$$

$$S2 = \text{sysconf(\_SC\_LEVEL2\_caché\_SIZE)} * \frac{1}{B}$$

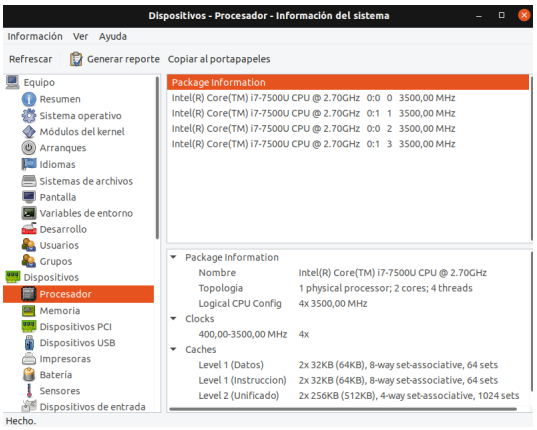
Se pueden observar los resultados obtenidos en el cuadro 1. Nótese que para el cálculo del número de conjuntos se utiliza la relación:

$$\text{Nº de conjuntos} = \frac{\text{Nº de líneas}}{\text{Nº de vías}} = \frac{\text{Tamaño total}}{\text{Tamaño de línea * Nº de vías}}$$

En base a los datos del cuadro 1, se tiene que los valores de L a emplear en el programa son:

- 0.5 * S1 = 0.5 * 512 = 256
- 1.5 * S1 = 1.5 * 512 = 768
- 0.5 * S2 = 0.5 * 4096 = 2048
- 0.75 * S2 = 0.75 * 4096 = 3072
- 2 * S2 = 2 * 4096 = 8192
- 4 * S2 = 4 * 4096 = 16384
- 8 * S2 = 8 * 4096 = 32768

Figura 2: *Información de los niveles de caché obtenida con hardinfo. Obsérvese la multiplicidad de los niveles L1 de instrucciones y datos y del nivel L2.*



Como detalle adicional, cabe destacar que en este equipo cada una de las cachés L1 y L2 se encuentran compartidas entre 2 cores. Por la peculiaridad de este hecho con respecto a la estructura habitual de las jerarquías de memoria, se ha comprobado tanto a través de la información disponible en el directorio *sys/devices/system/cpu/* de Linux, como a través de *hardinfo*: véase la figura 2.

B. Condiciones de ejecución

Todas las ejecuciones del script *bash_script.sh* se han realizado con el menor número de procesos en ejecución posible (un proceso del shell para poder lanzar el programa, y aquellos que sean imprescindibles para el sistema operativo). Se trata de una medida cautelaria que busca minimizar el impacto de tareas en segundo plano, puesto que los experimentos son sensibles a interrupciones, uso de la memoria por otros procesos, etc.

Asimismo, para evitar cualquier tipo de interferencia entre pruebas resultado de las decisiones del planificador del sistema operativo, no se paraleliza ninguna de ellas.

También se ha ejecutado cada prueba en un nuevo proceso, en lugar de ejecutar varias veces el mismo programa, para forzar que todos los experimentos operaran bajo condiciones iniciales lo más similares posibles y que no se reutilizaran líneas de ejecuciones anteriores, sino que tuvieran que cargarse desde cero.

Por otra parte, se ha operado bajo dos conjuntos de restricciones de compilación distintos:

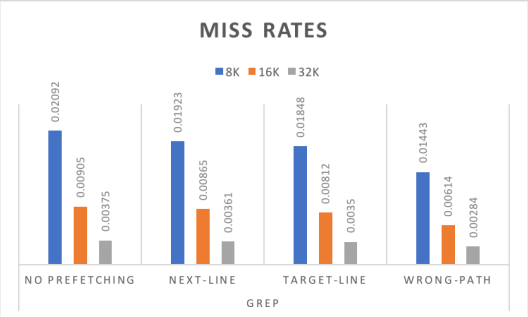
En primer lugar, tal y como requería la práctica, se ha empleado el compilador gcc con la opción -O0. Esta evita determinadas optimizaciones del compilador como, por ejemplo, la precarga software, la no inclusión de funciones sin efectos secundarios en el código ensamblador, el desenrollamiento de bucles (que optimiza la ejecución de los mismos eliminando o reduciendo ciertas instrucciones), el uso de instrucciones vectoriales, etc. ([5]). Así, se pretende que el proceso de traducción del código en C a lenguaje ensamblador mantenga una alta fidelidad con respecto al original, para que los usuarios puedan analizar el comportamiento interno del ordenador en base a las instrucciones programadas.

La compilación con la opción -O0 se ha hecho siempre acompañada de un proceso de randomizado del vector de índices ind[]. Por tanto, las ejecuciones asociadas buscan minimizar todo lo posible la localidad del programa.

En segundo lugar, se ha optado por emplear la opción -O3 y no desordenar el array de índices. Esta elección tiene el propósito contrario al anterior: maximizar las optimizaciones y la eficiencia derivada de la localidad, con el objetivo de poder contraponer los resultados a los primeros. Con la opción -O3 se estará permitiendo el desenrollamiento de bucles en un amplio abanico de situaciones, se permite que el compilador reemplace llamadas a funciones con el código de las mismas, no se impiden optimizaciones 'agresivas' en la compilación, etc.

C. Precarga

Figura 3: *Benchmarks de tasa de fallo del comando grep aplicando diferentes algoritmos de precarga. Los detalles técnicos pueden consultarse en [9].*



La precarga es una técnica de reducción de fallos caché mediante la cual se predicen futuras solicitudes de datos, que se traen a memoria caché antes de que el procesador realmente las requiera ([1]). De esta forma, si las predicciones son buenas (lo cual está estrechamente relacionado con que se cumpla el principio de localidad), el procesador tendrá disponibles en caché los datos que vaya pidiendo a medida que ejecute el programa, sin sufrir una penalización en términos de tiempo por tener que buscarlos en memoria principal. Se puede observar un ejemplo del impacto de este perfeccionamiento en la figura 3.

El principio básico de la precarga se basa en traer de memoria principal a la caché no solo los datos solicitados, sino estos y algunos cercanos o que se han determinado, por técnicas de predicción, con altas probabilidades de ser requeridos en un futuro próximo.

Se puede distinguir entre precarga software y precarga hardware. La primera es controlada por el compilador y utiliza técnicas como *loop unrolling* (desenrollamiento de iteraciones de bucles) para evitar los riesgos de datos para saltos en el pipelining. Sin embargo, es importante equilibrar los efectos positivos que ofrece la precarga con el *overhead* que suponen sus operaciones adicionales, de manera que acabe suponiendo una mejora en el rendimiento y no un detrimento.

Por contra, la precarga hardware es responsabilidad del propio procesador. Habitualmente, como ocurre en los niveles L1 y L2 de los Intel Core i7 (precisamente el procesador del equipo de pruebas), este proceso se realiza incorporando la línea caché inmediatamente posterior a la solicitada al buffer de la caché ([1]). Se razonará en el apartado G por qué, en base a los resultados obtenidos, este es el criterio que se está empleando con casi total seguridad.

En realidad, la situación ideal de ejecución sería eliminar completamente la precarga. La opción -O0 de compilación nos permite evitar las optimizaciones por software, pero no es posible eludir el impacto que tendrá el *prefetching* por hardware. Se trata sencillamente de un factor más a tener en cuenta a la hora de analizar los resultados.

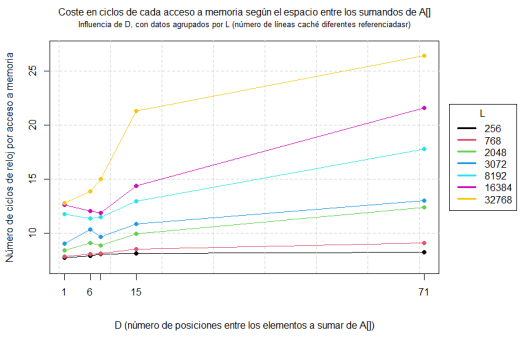
IV. Resultados

En este apartado se presentan, interpretan y justifican los distintos resultados obtenidos en la práctica. Para ello, se incluirán distintas gráficas construidas en R y un mapa de calor obtenido a través de Excel.

La sección presente se estructura en distintos subapartados que hacen referencia a condicionantes sobre los resultados y aspectos de interés en los mismos.

A. Elección de D

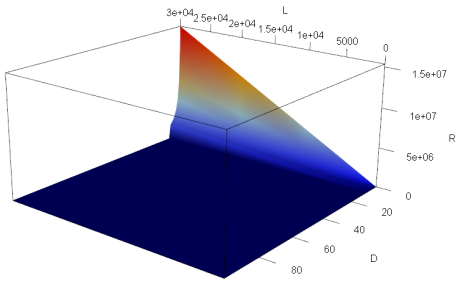
Figura 4: *Representación gráfica de los tiempos por acceso para los valores de D 1, 6, 8, 15 y 71 con la opción de ejecución -O0 y con el array ind[] desordenado.*



Una de las observaciones más claras es que los tiempos por acceso empeoran a medida que D crece, como se puede apreciar en la

figura 4. La explicación se encuentra en la propia naturaleza de D: si aumenta, se estarán solicitando elementos cada vez más alejados entre sí, de modo que la localidad disminuye significativamente.

Figura 5: *Representación gráfica de la función de obtención de R (eje Z) con D (eje Y) y L (eje X) como entradas.*

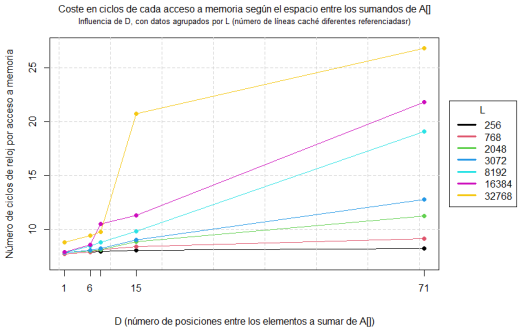


Cuando se accede a los elementos de forma ordenada y $D = 1$, se tiene una situación de localidad secuencial (un caso particular de localidad espacial), ya que se están leyendo elementos consecutivos. Además, se tendrá, en cada operación de reducción, localidad temporal con respecto a la misma línea caché, pues se empleará $\frac{B}{sizeof(double)} = \frac{64}{8} = 8$ veces seguidas cada línea.

En el caso de las ejecuciones de -O3 y no randomizado de ind[], se observa que en esta situación se obtienen tiempos de acceso muy bajos para cualquier valor de L. Puesto que los movimientos de datos en la jerarquía caché se realizan en bloque (como líneas completas), por cada fallo habría asegurados 7 aciertos posteriores, y en consecuencia el rendimiento es alto.

Si se ejecuta con -O0 y randomizamos ind[], los resultados son significativamente peores: se pasa de una media de 7 segundos (casi independientemente de L) a superar los 14 segundos para el mayor valor de L. Es de esperar, pues con esta última opción ya no se está accediendo secuencialmente a los datos y, además, se tiene un número de sumandos muy elevado (N), ya que D es inversamente proporcional a R (véase la gráfica 5), de manera que se acaba necesitando reemplazar un gran número de líneas, sobre todo para L grandes.

Figura 6: *Representación gráfica de los tiempos por acceso para los valores de D 1, 6, 8, 15 y 71 con la opción de ejecución -O3 y con el array ind[] ordenado.*



Cuando $1 < D < 8$, apenas llega a apreciarse un deterioro en el rendimiento. En el caso de -O0, se observa que hay entre 1 y 2 segundos de diferencia para la mayoría de valores de L con respecto a $D = 1$. No obstante, los tiempos siguen siendo bajos y no llegan a sobrepasar los 15 segundos. El fundamento es que, aunque ahora se accede a menos elementos por línea caché, se reutilizan líneas varias veces en cada reducción. Cuando L es pequeño, esto es especialmente significativo, pues hay menos probabilidades de que las líneas se reemplacen y, por tanto, los datos seguirán estando disponibles en caché. El caso en el que ind[] está ordenado es aún mejor (obsérvese la figura 6), incluso sin tener en cuenta el resto de optimizaciones de -O3, pues con casi total seguridad la línea estará disponible para todos los accesos que se requieran de ella, al producirse de forma ordenada.

Cuando $8 \leq D < 16$, la fórmula de R cambia: ahora R pasa a ser igual a L. Es decir, de cada línea solamente se toma a lo sumo un elemento (si $D = 8$ se toma exactamente uno, si $D > 8$ puede haber líneas de A[] que no se empleen). En este punto empiezan a notarse los efectos de la precarga. Con anterioridad, se tenía la seguridad de que el sumando de índice inmediatamente posterior a uno dado iba a estar en la misma línea caché o en la siguiente. Tomando por cierto el supuesto de que el *prefetching* hardware trae dos líneas, la requerida y la que le sucede, este suponía una

alta mejora en el rendimiento. Incluso en el caso en el que ind[] no está ordenado, cuando L es pequeño y no hay fallos caché de capacidad ([2], [3]), como en el caso $L = 0{,}5 * S1$, o estos son muy escasos, como en el caso $L = 1{,}5 * S1$, las líneas caché necesarias estarán disponibles para su uso por haberse traído con anterioridad.

Ahora que hay líneas que ya no se utilizan esto deja de ser cierto, y la magnitud de los beneficios de la precarga disminuye. De hecho, uno de los argumentos que sustentan la hipótesis de la línea extra por precarga hardware es el notable empeoramiento en los resultados cuando $D \geq 16$, pues en este caso la línea extra no se emplea nunca. La disminución del rendimiento ocurre tanto para la opción -O0 como para la -O3, pero la primera es la verdaderamente esclarecedora, pues la precarga software de la segunda nos impide definir con precisión las causas de las diferencias encontradas.

De este valor en adelante, a medida que D aumenta los resultados no dejan de empeorar. Se aprecia un comportamiento casi lineal y con una pendiente que se agudiza a medida que se incrementa L. De hecho, apenas se aprecian diferencias entre los dos tipos de ejecución cuando L es muy elevado, y en esta muestra ambos alcanzan un valor máximo de alrededor de 26 segundos.

En base a las explicaciones anteriores y diversas ejecuciones, se estima que las conclusiones más significativas surgen al elegir 5 valores de D en rangos más bajos. Se ha considerado que el conjunto $\{1, 6, 8, 15, 71\}$ es una muestra representativa de las peculiaridades explicadas, y es la que reflejan las figuras de esta sección [1].

B. Localidad

La localidad ([4]) de mayor peso en este estudio es, sin duda, la espacial. La localidad temporal sobre un mismo dato únicamente

[1] Aunque en este documento solo se muestran representaciones del conjunto 1, 6, 8, 15 y 71 como valores de D, también se ha considerado de interés emplear 1, 3, 7, 8 y 16; 10, 30, 50, 70 y 90; y 14, 15, 16, 17 y 18. Sus diagramas asociados en los dos modos de ejecución descritos se incluyen como anexo, y se recomienda su comprobación para visualizar con más minuciosidad los detalles descritos en este estudio.

Figura 7: *Mapa de calor de los 35 resultados obtenidos con la opción de ejecución -O0 y con el array ind[] ordenado.*

L	D	R	N	CK
256	1	2041	2041	7,731455
768	1	6137	6137	7,808538
2048	1	16377	16377	8,397329
3072	1	24569	24569	9,014288
8192	1	65529	65529	11,74426
16384	1	131065	131065	12,6101
32768	1	262137	262137	12,77117
256	6	341	2041	7,903812
768	6	1024	6139	8,059668
2048	6	2731	16381	9,085701
3072	6	4096	24571	10,3483
8192	6	10923	65533	11,33084
16384	6	21845	131065	12,03586
32768	6	43691	262141	13,84754
256	8	256	2041	8,046093
768	8	768	6137	8,129817
2048	8	2048	16377	8,822876
3072	8	3072	24569	9,663151
8192	8	8192	65529	11,49526
16384	8	16384	131065	11,88933
32768	8	32768	262137	15,02415
256	15	256	3826	8,082812
768	15	768	11506	8,490299
2048	15	2048	30706	9,918774
3072	15	3072	46066	10,86379
8192	15	8192	122866	12,93031
16384	15	16384	245746	14,39343
32768	15	32768	491506	21,32397
256	71	256	18106	8,215039
768	71	768	54458	9,092578
2048	71	2048	145338	12,38833
3072	71	3072	218042	12,97951
8192	71	8192	581562	17,80656
16384	71	16384	1163194	21,61374
32768	71	32768	2326458	26,43212

llega a tener efecto 10 veces, una por reducción, pues cada sumando se emplea una única vez en cada operación. Aun si consideramos localidad temporal sobre cada línea caché en conjunto, esta solo tendrá efecto en una única reducción cuando $D < 8$.

En contraposición, D controla la localidad espacial de manera directa. Cuanto más pequeño sea, más se cumple este principio, pues se precisarán datos del array cercanos. Si L es pequeño, lo que implica que la probabilidad de reemplazo de líneas caché no es demasiado alto, se podrá utilizar este hecho aun en el caso de desordenar ind[] para no sufrir la penalización de ir a memoria principal, gracias al *prefetching*. En el caso de utilizar precarga hardware, este efecto solo será aprovechado si D es menor a 16, pero con precarga software los algoritmos podrían adaptarse a tamaños más variados.

Este análisis se puede efectuar, por ejemplo, a través del mapa de calor de la figura 7.

C. Picos para D=1

Cuando se emplea la opción -O0, es relativamente frecuente encontrar que los valores más pequeños de D (como 1 o 3) ofrecen un

rendimiento peor que elecciones algo superiores, como $D = 6$ o $D = 7$, especialmente cuando L toma valores altos. Ello provoca que en las representaciones de tiempos por acceso en función de D, se observe una tendencia inicial decreciente.

La explicación se encuentra en la expresión de la fórmula de R cuando $D < 8$ (B). La gráfica de la función (5) delata el motivo: se puede ver que R se dispara cuando D es pequeño y L grande, pues se podría aproximar R como $R \approx 8 * (L - 1)$. Observando también la fórmula de N, $N = D * (R - 1) + 1$, tampoco es descabellado asumir $R \approx N$.

En consecuencia, el número de sumandos es exageradamente elevado, alcanzando cotas del orden de 131065 para (D=1, L=16384), o 262137, para (D=1, L=32768). A la larga, esto provocará un elevado número de reemplazos en las caches por fallos de capacidad y conflicto. También se reduce la probabilidad de poder reutilizar una línea entre distintas reducciones del grupo de 10 por prueba.

Cuando ind[] está ordenado las consecuencias no son tan graves, al aprovecharse una localidad secuencial o casi secuencial, pero si ind[] está desordenado los efectos son catastróficos, y de ahí los picos en las gráficas de -O0.

Cuando aumentamos D ligeramente, hasta alcanzar por ejemplo D=6, R ya no supera el rango de los 50000 para ningún valor de L. En este caso, el número de sumandos es asumible y la asíntota de la función de R en $D = 0$ deja de tener tanto efecto.

D. Valores de L y R

Se observa que un incremento en el valor de L provoca también un mayor número de ciclos de acceso, ya sea tanto en las ejecuciones de -O0 como en las de -O3. En esta subsección se deducirá, en base a las propiedades teóricas de la jerarquía de memoria, por qué el número de fallos de capacidad y conflicto es dependiente de L

Cuando L es pequeño, la mayoría de líneas caché de A[] tendrán cabida en los niveles de caché más cercanos a las CPUs. Dado que en las 10 reducciones los únicos datos empleados son el vector A[], dos variables de iteración (i y j) y el vector S[] de resultados, las líneas caché empleadas se podrán alojar en la caché L1 de datos sin que tengan lugar fallos de capacidad y conflicto. En particular, esto es lo que ocurre cuando $L = 0{,}5 * S1$.

Si L crece, el número de líneas caché que será necesario emplear en cada reducción no podrá almacenarse por completo en el nivel L1 de datos. Por ende, se precisará efectuar movimientos desde niveles inferiores. En concreto, como mínimo se usará la caché L2, puesto que cada dato debe estar disponible en todos los estratos inferiores.

Este hecho incurre en una penalización por fallo, al tener que sustituir una línea y al ser la velocidad de acceso menor cuanto más se baje en la jerarquía ([2]). La repercusión sobre el tiempo de acceso asociado es directa.

Por añadidura, en este equipo la caché L1 es la única que se encuentra dividida entre datos e instrucciones. Emplear los niveles L2 y L3 implica que los datos se encontrarán en competencia directa por el espacio con las instrucciones, lo cual provoca un agravamiento del número de fallos de conflicto. A pesar de ello, en el caso de este código este no es un factor demasiado preocupante, pues se emplea un lazo de pocas instrucciones que acabarán quedando completamente contenidas en la L1 de instrucciones.

Lo que sí afecta, en cambio, es la asociatividad: se comentaba en el cuadro 1 que la caché L2 dispone únicamente de 4 vías, mientras que cada L1 cuenta con 8. Este menor número de líneas disponibles por cada conjunto repercute en un aumento del número de fallos de conflicto.

La base teórica descrita es coherente con las especificaciones del problema y con los resultados. L afecta directamente al valor de R, tanto si $D < 8$ como si $D \geq 8$, en el primer caso de forma amortiguada y en el segundo directamente. Un incremento de R implica la necesidad de mover más líneas entre los niveles de la jerarquía de memoria (y, por tanto, la aparición de más fallos de capacidad).

Adicionalmente, si ind[] está desordenado, ni siquiera se tiene una probabilidad aceptable de que una línea se pueda llegar a utilizar más de una vez entre las 10 reducciones, pues un elevado L provoca tantos fallos que para ese momento es probable que la línea en

Figura 8: *Representación gráfica de los tiempos por acceso para cada valor de L, 0.5*S1, 1.5*S1, 0.5*S2, 0.75*S2, 2*S2, 4*S2 y 8*S, con la opción de ejecución -O0 y con el array ind[] ordenado.*

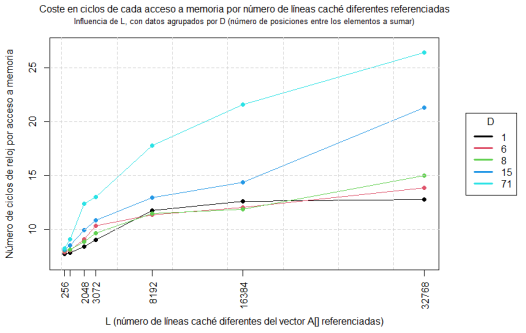
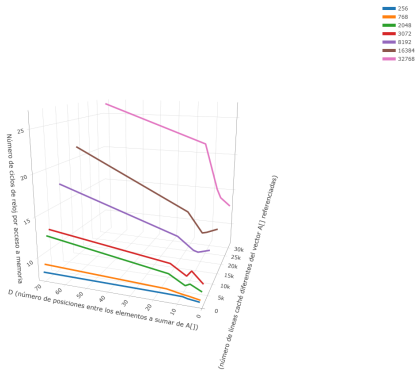


Figura 9: *Representación gráfica 3D de los tiempos (eje Z) para cada valor de L (eje X) y de D (eje Y), con la opción de ejecución -O0 y con el array ind[] ordenado.*



cuestión ya se haya reemplazado.

Por ese motivo, los elevados valores de L (y de R) están asociados con comportamientos mucho más extremos en los datos. Basta comparar, por ejemplo, $L = 32768$ con el valor inmediatamente anterior, $L = 16384$, para observar diferencias de unos 3 segundos para valores de D reducidos, y de entre 5 y 7 segundos para los más altos. Por contra, los valores inferiores de L presentan un comportamiento aceptable, especialmente en los casos en los que $D < 8$, con elevada reutilización, o $D < 16$, donde tiene efecto la precarga hardware. Estas inferencias provienen de figuras como 8 y 9, siendo esta última una captura de un recurso interactivo que se incluye como documento anexo para permitir su manejo.

La interpretación aquí descrita es consecuente con investigaciones en el campo tales como [6].

E. Tamaño de A[]

El valor de N también es significativo, con especial énfasis en los casos en los que tenemos precarga software e ind[] está ordenado. Cuando aumenta el tamaño del array, habrá más datos de referencia con respecto a los cuales basar las predicciones de la precarga, de forma que eventualmente se hará más efectiva (si ind[] está desordenado, el impacto se reduciría o desaparecería por completo).

Ahora bien, puesto que $N = D * (R - 1) + 1$, si queremos maximizar N habrá que tener un valor de D grande (o, al menos, superior a 7, para que R sea igual a L y no sea inversamente proporcional con D) y un valor de L grande. Es por ello que las pendientes de las gráficas de -O3 son menos pronunciadas que las de -O0 en rangos altos de D.

F. Orden de ejecución

Al iniciar el proceso de pruebas, se comenzó probando un script de bash que ejecutaba las pruebas mediante el siguiente lazo:

```
para un L ∈ listaL
    para un D ∈ listaD
      para k=0..9
        realizar prueba
```

Sin embargo, se observaban numerosas irregularidades en los resultados entre pruebas. Se hipotetizó entonces que el orden del lazo podría ser significativo para las mediciones. En efecto, se comprobó que cambiando el orden a:

```
para k=0..9
    para un D ∈ listaD
      para un L ∈ listaL
        realizar prueba ,
```

las gráficas se volvían mucho más uniformes. Se dedujo que la razón principal era una incorrecta suposición de independencia entre las pruebas: como inicialmente las 10 repeticiones de cada experimento (un par (L, D) fijo) se lanzaban de forma consecutiva, cualquier proceso en segundo plano que el

equipo estuviera ejecutando en ese momento para desconocimiento del programador acabaría afectando a varias de las pruebas, con lo que la mayoría se verían desplazadas y, en consecuencia, también la mediana. Una vez distribuidas y alejadas las pruebas en el tiempo, la probabilidad de que un proceso secundario afecte a varias de ellas significativamente se minimiza.

G. Precarga hardware

Como ya se ha comentado en el apartado A, la precarga hardware con una línea extra es responsable de gran parte de la diferencia en los tiempos cuando D pasa del rango [1, 15] al rango [16, 100]. En este punto, la localidad es tan baja que el *overhead* de las operaciones adicionales que supone acaba sobreponiéndose a sus efectos sobre la eficiencia, que desaparecen si consideramos cierta la hipótesis sobre el número de líneas traídas por predicción, lo cual es bastante probable en base a los resultados.

La consecuencia más evidente en las visualizaciones es la agudización de las pendientes de los tiempos por acceso cuando se supera el umbral de $D = 16$. A partir de ese punto, el incremento es prácticamente lineal.

Durante un tiempo se estudió la hipótesis de que el paso de D de un múltiplo de 8 a otro afectaría notablemente a las mediciones. No obstante, no se encontraron pruebas estadísticamente significativas al respecto y la idea se acabó desechando. Los únicos múltiplos de 8 con consecuencias reales son 8 y 16, por los motivos previamente expuestos.

H. Proporción R/L

El cociente R/L se puede entender como una estimación de la localidad del programa: representa el número de sumandos que se toman por línea caché requerida. Se ha elaborado una representación al respecto que demuestra la relación inversamente proporcional entre localidad y tiempos por acceso. Nótese que la alta proporción de puntos para $R/L = 1$ se debe a que $R = L$ para $D \geq 8$, de modo que los valores realmente interesantes de esta gráfica se encuentran en el rango $R/L > 1$.

Figura 10: *Representación gráfica de los tiempos por acceso para la proporción R/L, representativa de la localidad (en especial, la localidad espacial), con la opción de ejecución -O0 y con el array ind[] ordenado.*

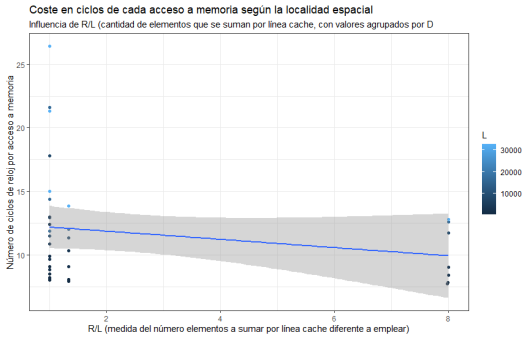
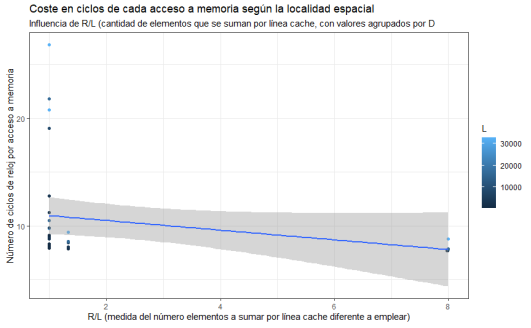


Figura 11: *Representación gráfica de los tiempos por acceso para la proporción R/L, representativa de la localidad (en especial, la localidad espacial), con la opción de ejecución -O0 y con el array ind[] ordenado.*



Aunque R/L es indicativo de la localidad en general, es especialmente interesante interpretarlo a través de la localidad espacial en particular, pues es el criterio sobre el que tiene efecto directo y palpable.

En las gráficas 10 y 11 se representa una recta de regresión simple que aproxima la relación entre localidad y tiempo por acceso bajo el supuesto de carácter lineal. Se incluye también un intervalo de confianza para la localización de la recta.

V. Conclusiones

En este informe se ha estudiado un tema de gran envergadura a la hora de reducir la latencia en microprocesadores, aspecto que ha sido recientemente tratado desde el punto de vista de utilización de la caché [10], [11] aportando este trabajo una visión complementaria. Así, se ha analizado el efecto de la localidad espacial y temporal, de la arquitectura de memoria y de otros factores (precarga, número de acceso a datos, independencia de las pruebas de ejecución, etc.) sobre la eficiencia en los tiempos de acceso a memoria.

Los requerimientos del problema en cuestión, los detalles técnicos del hardware empleado y la base teórica de los puntos mencionados se han empleado como base para el análisis de los resultados. Estos reflejan una fuerte dependencia de los tiempos de acceso con respecto a la localidad, así como del porcentaje de utilización de cada nivel caché, dos factores estrechamente relacionados. Una baja localidad (D elevado) repercute en un gran número de movimientos de datos entre distintos niveles de la jerarquía, y si la propia cantidad de datos es alta (L y R de considerable magnitud) los fallos de conflicto y capacidad provocan importantes penalizaciones en los tiempos de acceso.

Asimismo, se ha observado la relevancia que supone el control de la independencia en el entorno de pruebas y factores de menor envergadura como el tamaño de las estructuras de datos (que facilita la precarga y aumenta R) o la proporción entre R y L (indicativa de la localidad).

Finalmente, el tipo de precarga hardware implementada en el computador en el que se han realizado las pruebas ha resultado fundamental para comprender las predicciones basadas en localidad (véase sección A de IV. Resultados). Por ello, se considera que el estudio de las especificaciones de esta precarga, especialmente la referencia de líneas caché consecutivas, una cuestión de interés para estudios futuros, así como la ejecución del programa adjunto en otras arquitecturas para analizar los efectos de las jerarquías de memoria (tipos de memoria caché, compartición entre núcleos y distinciones entre datos e instrucciones).

Referencias

[1] Hennessy, John L. et al. (5ª Edición). Arquitectura de Computadores: Un enfoque cuantitativo. Cap. 2: Diseño de la jerarquía de memoria. (pp. 78-105). Editorial Morgan Kaufmann, Elsevier. *Intel virtualization technology.* Computer, vol. 38, no. 5, pp. 48–56, 2005.

[2] Patterson, David A. y Hennessy, John L. (5ª Edición). Computer Organization and Design: MIPS Edition. Chapter 5: Large and Fast: Exploiting Memory Hierarchy. (pp 372-499). Editorial Morgan Kaufmann, Elsevier. 2014.

[3] Brihadiswaran, Gunavaran. Taxonomy of caché Misses. *The High Performance Computer Forum.* 2020. `shorturl.at/ehET9`, [online] última visita 23 de marzo de 2022.

[4] Harris, Sarah L. y Harris, David Money. Digital Design and Computer Architecture. 8 - Memory Systems. (pp 486-529). Editorial Morgan Kaufmann. 2016.

[5] Selecting Optimization Options. *Arm Keil, Compiling Getting Started Guide.* Versión 6.16. `https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm` [online] última visita 22 de marzo de 2022.

[6] Morris, Gerald R. The effect of caché on memory access time. *ERDC DSRC, DoD Supercomputing Resource Center.* 2003. `https://www.erdc.hpc.mil/docs/Tips/caché20030711.pdf`, [online] última visita 23 de marzo de 2022.

[7] Plaff, Ben. How can I shuffle the contents of an array? *Ben Plaff* 2004 `https://benpfaff.org/writings/clc/shuffle.html`, [online] última visita 20 de marzo de 2022.

[8] Tang, Daisy. CS241 – Lecture Notes: Sorting Algorithm *CalPolyPomona, CS241: Data Structures and Algorithms II* Referencia del curso: Carrano, Frank M. Data Structures and Abstractions with Java. 4th Edition,

2014. https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm, [online] última visita 20 de marzo de 2022.

[9] Alsultaan, Heba et al. Final Report Implementing and Testing Four Prefetching Cache Algorithms. *University of Colorado Denver. Advance Computer Architecture.* 2017 https://www.researchgate.net/publication/328597173_Final_Report_Implementing_and_Testing_Four_Prefetching_Cache_Algorithms [online] última visita 23 de marzo de 2022.

[10] Perarnau, Swann Tchiboukdjian, Marc Huard, Guillaume. (2011). Controlling Cache Utilization of HPC Applications. Proceedings of the International Conference on Supercomputing. 295-304. 10.1145/1995896.1995942.

[11] Mobile Edge Cache Strategy Based on Neural Collaborative Filtering - Scientific Figure on ResearchGate. Performance comparison of content cache space utilization of different algorithms.

VI. Código de los programas

A. reduccion.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pmmintrin.h>       // Requiere
    la opción -msse3 al compilar
#include <time.h>
#include <unistd.h>
#include <math.h>            // Función
    ceil()
#include <string.h>          // Función
    memcpy()
#include "counter.h"


#define N_RED 10             // Número de
    veces que se repite la operación de
    reducción


/*
 * Xiana Carrera Alonso, Ana Carsi
     González
 * Arquitectura de Computadores - Curso
     2021/2022
 * Práctica 1 - Jerarquía de memoria y
     comportamiento de memoria caché:
     Estudio del efecto de la localidad
     de los
 * accesos a memoria en las prestaciones
     de programas en microprocesadores
 *
 * Este programa mide el número de ciclos
     medio por acceso a memoria durante
     una operación de reducción de suma
 * de punto flotante sobre R elementos de
     un vector de doubles. Dichos
     elementos están separados en D
     posiciones y,
 * en total, su uso supone el acceso a L
     líneas caché diferentes.
 * En última instancia, el objetivo es
     comprobar cómo diferentes factores
     que actúan sobre la localidad
     espacial en
 * el acceso a los datos puede alterar la
     eficiencia de los programas software.
 */


// Función que fija valores de algunos de
    los parámetros a emplear durante el
    programa (L, B, R, N y S1 óS2)
// en base a los datos pasados como
    argumento al programa (factor, caché
    y D).
void fijar_param(long * L, int * B, long
    * R, long * N, long * S1, long * S2,
    float factor, int caché, int D);

// Función que calcula el valor medio de
    los resultados de 10 operaciones de
    reducción
void inicializar_A(double A[], long N);

// Función auxiliar que simplifica la
    finalización del programa con error
void salir(char * msg);

// Función que determina los índices de
    los sumandos de la reducción en el
    vector A[] y los almacena como
    elementos
// del vector ind[]
void calcular_indices(int ind[], int D,
    long R);

// Función que calcula la media de los
    elementos del vector de resultados
```

```
double calcular_media(double S[]);

// Función que almacena los resultados
    del programa como una línea de
    formato "L D R N ck" sobre un
    archivo temporal,
// el cual será posteriormente procesado
    por el programa medicana.c
void escribir_resultados(long L, int D,
    long R, long N, double ck);

// Función que mide el grado de similitud
    entre dos vectores (con el objetivo
    de verificar la aleatoriedad)
double comprobar_random(int ind[], int
    copia[], long R);

// Función que randomiza los elementos
    del vector de índices para empeorar
    la localidad espacial de los datos
void desordenar(int ind[], long R);


int main(int argc, char * argv[]) {
    double * A;       // Array de doubles
    int * ind;        // Array de
        referencias a los elementos a
        sumar de A
    double * S;       // Array que
        contendrá los 10 resultados de la
        reducción
    double ck;        // Tiempo de 10
        operaciones de reducción en
        ciclos de reloj
    double ck_acceso; // Tiempo por
        acceso en ciclos de reloj (=
        ck/(10*R))

    long N;           // Tamaño del vector
        A
    long R;           // Número de
        elementos de A a sumar
    int D;            // Parámetro de
        espaciado entre los elementos a
        sumar: A[0], A[D], A[2*D], ...,
        A[(R-1)*D]

    long S1;          // Número de líneas
        caché que caben en la caché L1 de
        datos
    long S2;          // Número de líneas
        caché que caben en la caché L2
    int B;            // Tamaño de línea
        caché en bytes
    long L;           // Número de líneas
        caché diferentes que se deben
        leer en el acceso a A

    int i, j;         // Variables de
        iteración


    /*****************************
        PREPARATIVOS
        ****************************/

    // Comprobamos que la entrada del
        programa sea correcta
    if (argc != 4)
        salir("Error: el programa debe
            recibir tres argumentos:\n"
            "\t1) El valor de D a
                emplear"
            "\t2) Un factor sobre S1 o
                S2\n"
            "\t3) 1 o 2, para indicar L1
                de datos (S1) o L2
                (S2)\n"
            "Ejemplo: ./programa 0.5 1
                50\n"
            "(Equivale a L=0.5*S1;
                D=50)\n\n");

    // Fijamos la semilla para generar
        números aleatorios como el número
        de segundos desde Epoch (tiempo
        actual)
    srand((unsigned int) time(NULL));

    printf("\n\n\n********************
        EXPERIMENTO DE LOCALIDAD
        ********************\n");

    // Imprimimos el valor de los
        parámetros determinados por los
        argumentos introducidos por línea
        de comandos
    printf("Parámetros fijados: D = %d,
        L=%f*S%d\n", atoi(argv[1]),
        atof(argv[2]), atoi(argv[3]));

    D = atoi(argv[1]);        // D se
        puede guardar directamente en
        base a los argumentos del programa
    // fijar_param escanea los niveles de
        caché para determinar sus
        características físicas (B, S1,
        S2)
    // Además, calcula L, R y N en
        función del resto de parámetros y
        de los argumentos introducidos al
        programa
```

```
fijar_param(&L, &B, &R, &N, &S1, &S2,
    atof(argv[2]), atoi(argv[3]), D);

// En la reserva de memoria de A se
    alinea el inicio del vector con
    el inicio de la línea caché, para
    un mayor
// control de los resultados
// Pasamos el número de bytes a
    reservar (N*sizeof(double)) y el
    tamaño de la línea caché (B)
if ((A = (double *) _mm_malloc(N *
    sizeof(double), B)) == NULL)
   // Si tiene lugar algún error,
       cortamos la ejecución e
       imprimimos un error con
       salir()
   salir("Error: no se ha podido
       reservar memoria para A");

// El tamaño máximo que puede tener
    ind es N (si apunta a todas las
    posiciones de A)
if ((ind = (int *) malloc(N *
    sizeof(int))) == NULL)
   salir("Error: no se ha podido
       reservar memoria para ind");

// S tendrá 10 posiciones
if ((S = (double *) malloc(N_RED *
    sizeof(double))) == NULL)
   salir("Error: no se ha podido
       reservar memoria para S");


calcular_indices(ind, D, R);  //
    Almacenamos los índices de los
    elementos de A a sumar según el D
    elegido
// Desordenamos ind[] de forma
    aleatoria para minimizar la
    localidad espacial
// Se verifica que el vector quede
    realmente desordenado y, si no es
    así, se repite el proceso hasta
    que lo esté
desordenar(ind, R);
// Guardamos N valores aleatorios con
    signo aleatorio y valor absoluto
    en el rango [1,2) para evitar
    posibles
// efectos de inicialización de los
    niveles de la jerarquía de memoria
inicializar_A(A, N);

/*****************************
    REDUCCIÓN
    ****************************/

start_counter();         // Iniciamos
    la medición del tiempo de acceso
    total

// Realizamos 10 reducciones. Cada
    resultado se guarda como un
    elemento distinto de S
for (i = 0; i < N_RED; i++) {
   for (j = 0; j < R; j++){
       // Sumamos los R elementos
       determinados
       S[i] += A[ind[j]];
           // Accedemos a A[] a
           través de ind[], de forma
           aleatoria
   }
}

ck = get_counter();      // Paramos
    el contador


/*****************************
    ANÁLISIS DE RESULTADOS
    ****************************/


printf("\nTiempo de 10 reducciones en
    ciclos de reloj = %1.10lf\n", ck);

// Imprimimos la frecuencia de reloj
    estimada
mhz(1,1);

// Imprimimos los 10 resultados (que
    deberían ser iguales) y su media
    para evitar optimizaciones del
    compilador,
// que en caso contrario podría optar
    por no ejecutar determinadas
    operaciones
for (i = 0; i < N_RED; i++)
    printf("S[%d] = %f\n", i, S[i]);
printf("Media: %f\n",
    calcular_media(S));

// Cada reducción realiza R accesos
    al vector A[]
// Por cada llamada a reducir() se
    realizan 10 reducciones (que
    resultan en los 10 elementos de
    S[])
```

```
    // Por tanto, el tiempo por acceso en
        ciclos es ck/(10*R), esto es, el
        tiempo total entre el número de
        accesos
    ck_acceso = ck/(10*R);

    // Escribimos los resultados junto a
        los parámetros utilizados en un
        archivo temporal, que después
        será procesado
    // por un programa de cálculo de la
        mediana
    escribir_resultados(L, D, R, N,
        ck_acceso);



    // Liberamos la memoria reservada
    _mm_free(A);
    free(ind);
    free(S);

    return 0;
}


/*
 * Función fija los valores de los
     parámetros que se emplearán en la
     ejecución del programa.
 * S1, S2 y B (el tamaño de la línea
     caché) se leen a través de una
     llamada al sistema.
 * L depende del factor y la caché
     indicados como argumentos del
     programa, y del valor de S1 o S2,
     según corresponda.
 * El valor de R se calcula en función de
     B, L y D.
 * El valor de N se calcula en función de
     D y R.
 *
 * @param L Número de líneas caché
     distintas de las que se deben leer
     datos.
 * @param B Tamaño de la línea caché.
 * @param R Numero de elementos del
     vector A[] a sumar.
 * @param N Tamaño del vector A[].
 * @param S1 Numero de lineas caché que
     caben en la caché L1 de datos.
 * @param S2 Numero de lineas caché que
     cabaen en la caché L2.
 * @param factor Factor que se multiplica
     por S1 o S2 para calcular L (entrada
     del programa).
 * @param caché caché a utilizar: 1 -> L1
     de datos; 2 -> L2 (entrada del
     programa).
 * @param D Numero de posiciones que
     separan los elementos a sumar del
     vector A[] (entrada del programa).
 */
void fijar_param(long * L, int * B, long
    * R, long * N, long * S1, long * S2,
    float factor, int caché, int D){
    if (caché == 1) {       //
        Determinamos el valor actual de
        los parámetros de la caché L1 de
        datos
        // Guardamos en B el tamaño de
            línea de la caché L1 de datos
            en bytes
        *B = (int)
            sysconf(_SC_LEVEL1_Dcaché_LINESIZE);
        // S1 almacenará el número de
            líneas (tamaño total de la
            caché en bytes entre tamaño
            de cada línea en bytes)
        *S1 =
            sysconf(_SC_LEVEL1_Dcaché_SIZE)
            / *B;
        *L = *S1 * factor;    // L es
            igual al número de líneas
            escalado por un determinado
            factor
    } else {                  //
        Determinar el valor actual de los
        parámetros de la caché L2
        // Análogo al caso de la L1 de
            datos
        *B = (int)
            sysconf(_SC_LEVEL2_caché_LINESIZE);
        *S2 =
            sysconf(_SC_LEVEL2_caché_SIZE)
            / *B;
        *L = *S2 * factor;
    }

    /*
     * Obtenemos el número de sumandos de
         A[] que debemos emplear para que
         se acceda a L líneas caché y los
         elementos
     * estén separados por D posiciones.
     * Si D es menor que
         *B/sizeof(double)
         (presumiblemente 8), el valor de
         R dependerá no solo de L, sino
         también de
     * D, pues se está cogiendo más de un
         elemento por cada línea caché.
```

```
        Es necesario emplear la función
        ceiling, pues
    * puede que un bloque de D
        posiciones esté partido entre
        varias líneas caché, y es
        necesario tomarlo en su
    * posición iniciales. Nótese además
        que de la última línea solamente
        es necesario tomar un elemento
        para cumplir
    * con la restricción, de ahí el 1
        añadido a mayores.
    * Si D es igual a 8, se toma
        exactamente un elemento por cada
        línea: R = L.
    * Si D es mayor que 8, eventualmente
        podrá haber líneas caché en las
        que esté almacenado A[] y a las
        que no se
    * acceda. Es decir, R será
        exactamente igual al número de
        líneas que se pretende usar, L.
    */
   *R = D >= *B/sizeof(double)? *L :
       (int) (ceil(*B * (*L - 1) /
       (double) (D * sizeof(double)))) +
       1;
   // Calculamos el tamaño total que
       deberá tener el array A[]. Puesto
       que cada elemento "ocupará" D
       posiciones,
   // necesitaremos D * (R - 1)
       posiciones para cubrir todos los
       sumandos menos el último. Este se
       puede sumar
   // en solitario, ya que a partir de é
       l no se empleará ningún índice
       más.
   *N = D * (*R-1) + 1;
}


/*
 * Función que escribe los resultados del
     experimento en el archivo
     res_temporales.txt con el formato
 * "L D R N ciclos_por_acceso".
 * @param L Número de líneas caché
     distintas de las que se deben leer
     datos.
 * @param D Numero de posiciones que
     separan los elementos a sumar del
     vector A[].
 * @param R Numero de elementos del
     vector A[] a sumar.
 * @param N Tamaño del vector A[].
 * @param ck_acceso Tiempo por acceso
     obtenido en la medición.
 */
void escribir_resultados(long L, int D,
    long R, long N, double ck_acceso){
   FILE *fp;      // Puntero al archivo

   // Abrimos el archivo en modo append,
       para continuar escribiendo a
       continuación de ejecuciones
       previas de este
   // mismo programa. Si
       res_temporales.txt no existía, se
       crea.
   if ((fp = fopen("res_temporales.txt",
       "a")) == NULL)
      salir("Error: no se ha podido
          abrir el archivo de
          resultados");

   fprintf(fp, "%ld %d %ld %ld %f\n",
       L, D, R, N, ck_acceso);

   if (fclose(fp)) salir ("Error: no se
       ha podido cerrar el archivo de
       resultados");
}

/*
 * Código de referencia:
     https://benpfaff.org/writings/clc/shuffle.html
 * Esta función randomiza los índices de
     los sumandos de A[]. Su corrección
     se basa en que N será siempre
     bastante
 * inferior a RAND_MAX
 * Si los resultados no son lo
     suficientemente aleatorios, se
     repite el proceso hasta que lo sean
 * @param ind Vector a randomizar
 * @param R Número de elementos
     almacenados en ind
 */
void desordenar(int ind[], long R){
   size_t i, j;
   long temp;
   int copia[R];

   // Copiamos todo el vector de índices
       (que llega ya ordenado)
   memcpy((void *) copia, (void *) ind,
       (size_t) R * sizeof(int));

   do{
      for (i = 0; i < R - 1; i++) {
         j = i + rand() / (RAND_MAX /
```

```
                (R - i) + 1); // Buscamos
                otro índice, aleatorio
            // Intercambiamos los elementos
            temp = ind[j];
            ind[j] = ind[i];
            ind[i] = temp;
        }
    } while (comprobar_random(ind, copia,
        R) > 0.15); // Si el grado de
        coincidencia con el vector
        ordenado es
    // superior al 15%, repetimos el
        randomizado
}

/*
 * Función que comprueba el grado de
     coincidencia entre dos vectores y
     devuelve la proporción de la misma
     con respecto
 * al tamaño total, que debe coincidir
     entre ambos.
 * @param ind Vector 1 a comparar
 * @param copia Vector 2 a comparar
 * @param R Tamaño de ambos vectores
 */
double comprobar_random(int ind[], int
    copia[], long R){
    int contador_iguales = 0;
    int i;

    for (i = 0; i < R; i++){
        // Si los dos vectores guardan el
            mismo valor en la posición i,
            aumentamos el contador
        if (ind[i] == copia[i])
            contador_iguales++;
    }

    // Devolvemos la proporción de
        coincidencias
    return contador_iguales / (double) R;
}

/*
 * Función que devuelve la media de los
     N_RED = 10 elementos de S[].
 * El objetivo de esta función es
     utilizar los elementos de S[] para
     evitar optimizaciones del compilador.
 * @param S Array de doubles sobre el que
     se realiza la media
 */
double calcular_media(double S[]){
    double media = 0.0;
    int i;

    for (i = 0; i < N_RED; i++) media +=
        S[i];

    return media / N_RED;
}

/*
 * Función que inicializa A con valores
     aleatorios con valor absoluto en el
     intervalo [1,2).
 * @param A Vector a inicializar
 * @param N Tamaño de A
 */
void inicializar_A(double A[], long N){
    int i;

    for (i = 0; i < N; i++) {
        // rand genera un entero
            aleatorio en [0, RAND_MAX)
        // Pasamos a double y dividimos
            entre RAND_MAX para obtener
            un número real en [0, 1)
        // Sumamos 1.0 para trasladar a
            [1, 2)
        A[i] = 1.0 + (double) rand() /
            RAND_MAX;

        // Multlicamos A[i] por 1 o -1
            para dar signo aleatorio.
        // rand % 2 tiene dos resultados:
            {0, 1}. Multiplicando por 2,
            obtenemos {0, 2}. Sumando -1,
            tenemos {-1, 1}.
        A[i] *= (rand() % 2) * 2 - 1;
    }
}

/*
 * Función que almacena en ind[] los í
     ndices de los elementos de A[] que
     se deben sumar: 0, D, 2*D, ...,
     (R-1)*D.
 * @param ind Vector en el que se
     guardarán los índices de los
     sumandos (de forma ordenada).
 * @param R Número de sumandos.
 */
void calcular_indices(int ind[], int D,
    long R){
    int i;

    for (i = 0; i < R; i++) ind[i] = i *
        D;
}
```

```
/*
 * Función auxiliar que cierra el
     programa e imprime un mensaje de
     error.
 * @param msg Mensaje de error a imprimir.
 */
void salir(char * msg){
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}
```

B. mediana.c

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


/*
 * Xiana Carrera Alonso, Ana Carsi
     González
 * Arquitectura de Computadores - Curso
     2021/2022
 * Práctica 1 - Jerarquía de memoria y
     comportamiento de memoria caché:
     Estudio del efecto de la localidad
     de los
 * accesos a memoria en las prestaciones
     de programas en microprocesadores
 *
 * Este programa calcula la mediana sobre
     los resultados de 10 ejecuciones de
     reduccion.c. Dichas medidas se leen
 * de un archivo temporal,
     res_temporales.txt, y se encuentran
     desordenados de forma intencionada,
     para aumentar la
 * independencia estadística al minimizar
     el efecto de procesos en segundo
     plano.
 * La mediana, que es una medida de
     centralización robusta, disminuye
     significativamente el efecto de
     valores atípicos.
 * Los resultados finales se guardan en
     el archivo res_totales.txt.
 */


#define NL 7     // Número de valores de
    L a probar
#define NE 10    // Número de
    experimentos para cada par (D, L)
#define ND 5     // Número de valores que
    toma D


// Función que ordena un array empleando
    insertion sort (que, aunque tiene
    complejidad O(n^2), se ha comprobado
    que
// es uno de los algoritmos más
    eficientes para arrays de pequeño
    tamaño, como en este caso)
// @param CK Array con los resultados de
    tiempos en ciclos de reloj por
    acceso que se quiere ordenar
void sort(double CK[NE]){
    double temp;
    int i;


    for(i = 1; i < NE; i++){
        temp = CK[i];      // Guardamos un
            elemento temporal
        j = i - 1;
        while(temp < CK[j] && j >= 0){
            // Vamos retrocediendo por el
                array e desplazamos una
                posicio hacia delante
                todos los elementos
            // mayores que temp, para
                hacerle hueco
            number[j+1] = number[j];
            j = j-1;
       }
       number[j+1] = temp; // Insertamos
           temp justo después del primer
           elemento encontrado que sea
           mayor o igual que él
       // De esta forma, el inicio del
           array (hasta el índice i)
           queda ordenado
    }
}


int main(int argc, char * argv[]){
    FILE *res_temporales;        //
        Archivo de resultados sin procesar
    FILE *res_finales;           //
        Archivo de los resultados
        obtenidos con la mediana
    long L[NL*ND];               // 35
        valores de L, el número de líneas
        caché diferentes a acceder
    int D[NL*ND];                // 35
        valores de D, el número de
        posiciones entre sumandos
    long R[NL*ND];               // 35
        valores de R, el número de
```

```
    sumandos
long N[NL*ND];              // 35
    valores de N, el tamaño del array
double CK[NL*ND][NE];       // 350
    valores de CK, el tiempo en
    ciclos de reloj para cada
    experimento
double mediana;             //
    Mediana de 10 experimentos con
    los mismos valores de L, D, R y N
int i, j;                   //
    Variables de iteración

// El archivo de resultados
    temporales sobre el que
    reduccion.c ha escrito los 350
    resultados se abre en modo
// lectura
if ((res_temporales =
    fopen("res_temporales.txt", "r"))
    == NULL){
    fprintf(stderr, "Error al abrir
        el archivo de resultados
        temporales\n");
    exit(EXIT_FAILURE);
}

// El archivo de resultados finales
    sobre el que se escribirán las 35
    medianas a obtener se abre en modo
// escritura. Si no existía
    previamente, se crea.
if ((res_finales =
    fopen("res_totales.txt", "w")) ==
    NULL){
    fprintf(stderr, "Error al abrir
        el archivo de resultados
        finales\n");
    exit(EXIT_FAILURE);
}

// Los resultados están
    "desordenados": los 10 resultados
    de cada experimento no están
    juntos, sino que el archivo
// se divide en 10 bloques de 35
    entradas cada uno. Por tanto, el
    bucle exterior para la lectura es
    de los 10
// bloques, y el interior es sobre
    los 35 experimentos.
// (El motivo de esta organización es
    incrementar la independencia de
    las pruebas).
for (j = 0; j < NE; j++){
    for (i = 0; i < NL * ND; i++){
        // Almacenamos los parámetros
        // L, D, R y N no varían para
            distintos j
        fscanf(res_temporales, "%ld
            %d %ld %ld %lf", &L[i],
            &D[i], &R[i], &N[i],
            &CK[i][j]);
    }
}

// Para cada experimento, obtenemos
    la mediana
for (i = 0; i < NL * ND; i++){
    sort(CK[i]);      // Ordenamos CK
        con un algoritmo de inserción
    // Como hay un número par de
        valores, realizamos la media
        de los 2 centrales
    mediana = (CK[i][4] + CK[i][5]) /
        ((double) 2);
    // Escribimos los resultados
        sobre el archivo
        res_totales.txt
    fprintf(res_finales, "%ld %d %ld
        %ld %f\n", L[i], D[i], R[i],
        N[i], mediana);
}

// Reabrimos el archivo de resultados
    temporales, esta vez en modo
    escritura, para borrarlo por
    completo
freopen("res_temporales.txt", "w",
    res_temporales);

// Cerramos ambos archivos
fclose(res_temporales);
fclose(res_finales);

exit(EXIT_SUCCESS);
}
```

C. bash_script.sh

```
#!/usr/bin/env bash


# Xiana Carrera Alonso, Ana Carsi
    González
# Arquitectura de Computadores -
    Curso 2021/2022
# Práctica 1 - Jerarquía de memoria
    y comportamiento de memoria
```

```
    caché: Estudio del efecto de la
    localidad de los
# accesos a memoria en las
    prestaciones de programas en
    microprocesadores
#
# Este programa de bash automatiza
    las mediciones ejecutando las 350
    pruebas requeridas y un programa
    de cálculo
# de la mediana sobre los 35
    experimentos, obteniendo
    finalmente un archivo de
    resultados con los 35 valores
# de ciclos de reloj finales.
# La ejecución de las pruebas se
    desordena (es decir, se alterna
    constantemente entre valores de D
    y L diferentes
# en lugar de realizar las 10
    mediciones de forma consecutiva)
    con el objetivo de minimizar el
    posible impacto
# de procesos en segundo plano e
    incrementar la independencia. Se
    ha observado que esto tiene un
    impacto
# importante sobre los resultados.


factor_L=(0.5 1.5 0.5 0.75 2.0 4.0
    8.0)    # Factores a multiplicar
    sobre S1 o S2
cache=(1 1 2 2 2 2 2)           # Í
    ndice de S correspondiente a cada
    factor
lista_D="10 30 50 70 90"        #
    Valores seleccionados de D


# Borramos res_temporales y
    res_totales para eliminar
    cualquier valor anterior
rm res_temporales.txt
rm res_totales.txt

# Creamos el archivo res_totales.txt
touch res_totales.txt
# Escribimos una cabecera para poder
    interpretar los resultados con
    facilidad en R
echo "L D R N CK" >> res_totales.txt

# No hace falta volver a crear
    res_temporales, ya que mediana
# lo crea si no existe (lo abre en
    modo "w").

for k in {0..9}       # Cada par (D,
    L) tiene 10 pruebas asociadas
do
    for D in $lista_D     # Iteramos
        sobre los 5 valores de D
    do
        for i in {0..6}   # Iteramos
            sobre los 7 valores de L
        do
            ./accesos_caché $D
                ${factor_L[i]}
                ${caché[i]}  #
                Programa principal
        done
    done
done

# Una vez obtenidos todos los
    resultados, el programa de la
    mediana procesa el archivo
    correspondiente
# (res_temporales.txt) y calcula los
    valores finales, que se guardan
    en res_totales.txt
./mediana2
```

D. info_cache.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>            //
    Función log
#include <unistd.h>

/*
 * Xiana Carrera Alonso, Ana Carsi
     González
 * Arquitectura de Computadores -
     Curso 2021/2022
 * Práctica 1 - Jerarquía de memoria
     y comportamiento de memoria
     caché: Estudio del efecto de la
     localidad de los
```

```
 * accesos a memoria en las
     prestaciones de programas en
     microprocesadores
 *
 * Este programa permite comprobar
     las características de los
     niveles L1 (datos e
     instrucciones), L2 y L3 de la
 * jerarquía caché del equipo. En
     concreto, se determinan el
     tamaño total, el tamaño de línea
     y el número de vías
 * de cada nivel.
 * Aunque realmente en el estudio no
     son necesarios todos estos
     valores, se ha querido
     comprobarlos igualmente para
 * asegurar la correcta comprensión
     de la estructura particular de
     memoria en el equipo de
     ejecución.
 */


int main(int argc, char * argv[]) {
    long total;               //
        Tamaño total de cada nivel,
        en bytes
    long vias;                //
        Número de vías de cada nivel
    long tam_linea;           //
        Tamaño de línea de cada nivel
    long n_lineas_L1I;        //
        Número de líneas de la caché
        L1 de instrucciones
    long n_lineas_L1D;        //
        Número de líneas de la caché
        L1 de datos
    long n_lineas_L2;         //
        Número de líneas de la caché
        L2
    long n_conjuntos_L1I;     //
        Número de conjuntos de la
        caché L1 de instrucciones
    long n_conjuntos_L1D;     //
        Número de conjuntos de la
        caché L1 de datos
    long n_conjuntos_L2;      //
        Número de conjuntos de la
        caché L2

    /***** Caché L1 de instrucciones
        *****/

    // Tamaño total
    total =
        sysconf(_SC_LEVEL1_Icaché_SIZE);
    //  N  de vías
    vias =
        sysconf(_SC_LEVEL1_Icaché_ASSOC);
    // Tamaño de línea
    tam_linea =
        sysconf(_SC_LEVEL1_Icaché_LINESIZE);

    printf("\n**** Datos caché L1 de
        instrucciones: ****\n");
    printf("\tTamanho total: %ld =
        2^(%d)\n", total, (int)
        (log(total) / log(2)));
    printf("\tN de vías: %ld\n",
        vias);
    printf("\tTamanho de linea:
        %ld\n\n", tam_linea);

    // El número de líneas será igual
        al tamaño total dividido por
        el tamaño de cada línea caché
    n_lineas_L1I = total / tam_linea;

    // El número de conjuntos será
        igual al número de líneas
        dividido por el número de vías
    n_conjuntos_L1I = n_lineas_L1I /
        vias;

    /***** Caché L1 de datos *****/

    total =
        sysconf(_SC_LEVEL1_Dcaché_SIZE);
    vias =
        sysconf(_SC_LEVEL1_Dcaché_ASSOC);
    tam_linea =
        sysconf(_SC_LEVEL1_Dcaché_LINESIZE);

    printf("**** Datos caché L1 de
        datos: ****\n");
    printf("\tTamanho total: %ld =
        2^(%d)\n", total, (int)
        (log(total) / log(2)));
    printf("\tN de vías: %ld\n",
        vias);
    printf("\tTamanho de linea:
```

```
    %ld\n\n", tam_linea);

n_lineas_L1D = total / tam_linea;
n_conjuntos_L1D = n_lineas_L1D /
    vias;

/***** Caché L2 *****/

total =
    sysconf(_SC_LEVEL2_caché_SIZE);
vias =
    sysconf(_SC_LEVEL2_caché_ASSOC);
tam_linea =
    sysconf(_SC_LEVEL2_caché_LINESIZE);

printf("**** Datos caché L2:
    ****\n");
printf("\tTamanho total: %ld =
    2^(%d)\n", total, (int)
    (log(total) / log(2)));
printf("\tN de vías: %ld\n",
    vias);
printf("\tTamanho de linea:
    %ld\n\n", tam_linea);

n_lineas_L2 = total / tam_linea;
n_conjuntos_L2 = n_lineas_L2 /
    vias;


/***** Caché L3 *****/

total =
    sysconf(_SC_LEVEL3_caché_SIZE);
vias =
    sysconf(_SC_LEVEL3_caché_ASSOC);
tam_linea =
    sysconf(_SC_LEVEL3_caché_LINESIZE);

printf("**** Datos caché L3:
    ****\n");
printf("\tTamanho total: %ld =
    2^(%d)\n", total, (int)
    (log(total) / log(2)));
printf("\tN de vías: %ld\n",
    vias);
printf("\tTamanho de linea:
    %ld\n\n", tam_linea);

// Se imprime el número de líneas
    de cada caché
    printf("S1I = n líneas L1I =
        %ld\n", n_lineas_L1I);
    printf("S1D = n líneas L1D =
        %ld\n", n_lineas_L1D);
    printf("S2 = n líneas L2 =
        %ld\n", n_lineas_L2);
    printf("S3 = n líneas L3 =
        %ld\n\n", total / tam_linea);
        // total y tam_linea hacen
        referencia a la L3

    // E imprimimos también el número
        de conjuntos
    printf("N conjuntos L1I =
        %ld\n", n_conjuntos_L1I);
    printf("N conjuntos L1D =
        %ld\n", n_conjuntos_L1D);
    printf("N conjuntos L2 = %ld\n",
        n_conjuntos_L2);
    printf("N conjuntos L3 =
        %ld\n\n", total / (tam_linea
        * vias));
}
```

E. graficas.R

```
# Xiana Carrera Alonso, Ana Carsi González
# Arquitectura de Computadores - Curso
    2021/2022
# Práctica 1 - Jerarquía de memoria y
    comportamiento de memoria caché:
    Estudio del
# efecto de la localidad de los accesos a
    memoria en las prestaciones de
    programas
# en microprocesadores.
#
# Este programa de R permite visualizar
    los resultados obtenidos en forma de
    5
# gráficas, 3 en 2D y 2 en 3D.
# Requiere las librerías plotly, plot y
    rgl.

# Leemos los datos, con una cabecera
    inicial, separados por un espacio
# y con un punto como marca decimal
datos <- read.table("1 3 7 8 16 00
    D.txt", header=T, sep=" ", dec=".")
attach(datos)    # Convertimos los
    componentes de datos en variables
head(datos)      # Lectura de una muestra
```

```
    de los datos
summary(datos)  # Resumen estadístico de
    los datos
str(datos)      # Estructura (tipo de
    dato) de la muestra



#************************************
# Gráfica 1 (2D): Número de ciclos (eje
    Y) en función de L (eje X) para
# cada D (colores).
#************************************

lista_D <- unique(D) # Distintos valores
    que toma D
lista_D

# Añadimos espacio extra de márgenes
# Con xpd, controlamos que el plot esté
    anclado a la región de dibujo
par(mar=c(5, 4, 4, 8), xpd=TRUE)

# Vamos a representar la relación entre L
    y CK, utilizando una línea distinta
# para cada valor de D (5 líneas en total)
# No indicamos eje X para configurarlo a
    continuación manualmente
# En este plot representamos en primer
    lugar el valor más bajo de D
# El tipo de representación es
    "overplotted" (puntos y líneas)
plot(L[D==min(D)], CK[D==min(D)],
    type="o", xaxt="n",
    ylim=c(floor(min(CK)),
        ceiling(max(CK))),
    col=1, pch=19,
    xlab = "",
    ylab = "Número de ciclos de reloj
        por acceso a memoria")

# Recorremos todos los elementos de
    lista_D menos el primero [-1]
for (i in lista_D[-1]){
  # Con which obtenemos el índice del
      elemento de la lista que es igual a
      i
  lines(L[D==i], CK[D==i], type="o",
      col=which(i==lista_D), pch=19)
}

par(xpd=F)
# Quitamos los márgenes para incluir una
    rejilla únicamente
# dentro del área gráfica
grid(nx = NULL, ny = NULL, lty = 2, col =
    "lightgray", lwd = 1)
# Volvemos a activar los márgenes
par(xpd=T)

# Añadimos una leyenda de colores para D
legend("right", legend=levels(factor(D)),
    lty=1, lwd=3, col = 1:5,
      title = "D", inset=c(-0.2, 0))

# Introducimos un título y un subtítulo
mtext(side = 3, line = 2, cex=1,
     "Coste en ciclos de cada acceso a
         memoria por número de líneas
         caché diferentes referenciadas")
mtext(side = 3, line = 1, cex=0.8,
     "Influencia de L, con datos
         agrupados por D (número de
         posiciones entre los elementos
         a sumar)")

# Especificamos el texto del eje X con
    mtext para controlar su posición
# (side = 1 -> debajo del gráfico)
mtext(side = 1, line = 3.8, cex = 1,
     "L (número de líneas caché
         diferentes del vector A[]
         referenciadas)")
axis(1, at=L, las=3, lwd=1)



#************************************
# Gráfica 2 (2D): Número de ciclos (eje
    Y) en función de D (eje X) para
# cada L (colores).
#************************************

lista_L <- unique(L)  # Distintos valores
    que toma L
lista_L


# Añadimos espacio extra de márgenes
# Con xpd, controlamos que el plot esté
    anclado a la región de dibujo
par(mar=c(5, 4, 4, 8), xpd=TRUE)

# De forma análoga a la gráfica 1,
    imprimimos una línea con puntos para
# cada valor de L, comenzando por aquella
    de menor valor
plot(D[L==min(L)], CK[L==min(L)],
    type="o", xaxt="n",
    col=1, pch=19,
    ylim=c(floor(min(CK)),
        ceiling(max(CK))),
```

```
     xlab = "",
     ylab = "Número de ciclos de reloj
         por acceso a memoria")

par(xpd=F)
# Quitamos los márgenes para incluir una
    rejilla únicamente
# dentro del área gráfica
grid(nx = NULL, ny = NULL, lty = 2, col =
    "lightgray", lwd = 1)
# Volvemos a activar los márgenes
par(xpd=T)

# Añadimos un título y un subtítulo
mtext(side = 3, line = 2, cex=1,
      "Coste en ciclos de cada acceso a
          memoria según el espacio entre
          los sumandos de A[]")
mtext(side = 3, line = 1, cex=0.8,
      "Influencia de D, con datos
          agrupados por L (número de
          líneas caché diferentes
          referenciadasr)")

# Recorremos todos los elementos de
    lista_L menos el primero [-1]
for (i in lista_L[-1]){
  # Con which obtenemos el índice del
      elemento de la lista que es igual a
      i
  lines(D[L==i], CK[L==i], type="o",
      col=which(i==lista_L), pch=19)
}

# Añadimos una leyenda en el margen
    derecho
legend("right", legend=levels(factor(L)),
    lty=1, lwd=3, col = 1:7,
      title = "L", inset=c(-0.2, 0))

# Configuramos el eje X y cambiamos las
    etiquetas para que coincidan con los
# valores de D
mtext(side = 1, line = 3.8, cex = 1,
      "D (número de posiciones entre los
          elementos a sumar de A[])")
axis(1, at=D, las=1, lwd=1)


#************************************
# Gráfica 3 (3D): Número de ciclos (eje
    Z) en función de L (eje X) y en
# función de D (eje Y)
#************************************

library(plotly) # Cargamos la librería
    plotly para representar en 3D

# El tipo de gráfico es scatter3d y
    utilizamos el modo líneas para poder
# unir los puntos
p3 <- plot_ly(datos, x=~datos$L,
    y=~datos$D, z=CK, split=~datos$L,
            type="scatter3d",
                mode="lines",
                line=list(width=7)) %>%
    layout(scene=list(
    xaxis = list(title = "L (número de
        líneas caché diferentes del
        vector A[] referenciadas)"),
    yaxis = list(title = "D (número de
        posiciones entre los elementos a
        sumar de A[])"),
    zaxis = list(title = "Número de
        ciclos de reloj por acceso a
        memoria")))
p3     # Imprimimos el gráfico

# Exportamos el gráfico en forma de un
    recurso html
htmlwidgets::saveWidget(as_widget(p3),
    "ciclos3D.html")


#************************************
# Gráfica 4 (3D): Función de obtención de
    R a partir de L y D, con B fijo.
#************************************

# Cargamos plot3D y rgl para representar
    funciones en 3D
library(plot3D)
library(rgl)

# Definimos la función que se utiliza
    para obtener R
funcion_R <- function(x, y){
  # x=L, y=D
  # Asumimos B=64
  ifelse(y < 8, ceiling(64*(x-1)/y*8)+1,
      x)
}

open3d()  # Abrimos una ventana para
    imprimir en 3D
# Utilizamos un gradiente de colores para
    representar intensidad en función
# de la magnitud de R
plot3d(funcion_R,
    col=colorRampPalette(c("blue",
    "lightblue", "orange", "red")),
```

```
      xlab="L", ylab="D", zlab="R",
          xlim=c(0,30000), ylim=c(1,99),
      aspect=c(1,1,0.5))

# Guardamos una captura del gráfico
rgl.snapshot('R_plot.png', fmt = 'png')


#************************************
# Gráfica 5 (2D): Número de ciclos (eje
    Y) en función de R/L (eje X)
#************************************

# R/L representa un factor de "localidad
    espacial". Cuanto mayor es el número
# de datos que tenemos que coger por
    línea, mayor es la localidad
    espacial.

library(ggplot2)  # Para este gráfico
    emplearemos ggplot, que nos permite
# añadir fácilmente una recta de regresión

loc_esp = R/L  # Creamos una variable que
    guarda R/L

theme_set(theme_bw())  # Tema estético
    del gráfico

# Representamos con ggplot, poniendo
    loc_esp en el eje X y los ciclos de
    reloj
# en el eje Y.
# Añadimos una recta de regresión y una
    región de confianza
# A continuación, configuramos el título
    y las etiquetas de los ejes.
ggplot(data = datos, mapping = aes(x =
    loc_esp, y = CK)) +
        geom_point(aes(color=L, group=L))
            +
        geom_smooth(method="lm",
            formula=y~x) +
        stat_summary(fun.data=mean_cl_normal)
            +
        labs(subtitle="Influencia de R/L
            (cantidad de elementos que
            se suman por línea caché,
            con valores agrupados por D",
            y = "Número de ciclos de
                reloj por acceso a
                memoria",
            x = "R/L (medida del número
                elementos a sumar por
                línea caché diferente a
                emplear)",
            title = "Coste en ciclos de
                cada acceso a memoria
                según la localidad
                espacial")
```