PRÁCTICAS ESTRUCTURA DE COMPUTADORES II

II, ITIS - Curso 2008-2009

Optimización Software del Rendimiento Caché

-Guía de Actividades Propuestas-

1. Primer día

Antes de intentar mejorar la localidad de los códigos deberíamos comprender la influencia que tienen los parámetros de la caché sobre la tasa de fallos. Para ello usamos el simulador `dineroIII` ubicado en `/usr/local/bin` y por ejemplo dos de las trazas que se encuentran dentro del archivo `r2000.zip` en el directorio `/usr/local/dineroIII/trazas`. Para informarse sobre el manejo de este simulador puede consultarse su manual (`man dineroIII`) y leer el apéndice I de esta guía de prácticas. En esta clase se harán tres gráficas usando cachés unificadas.

En la primera gráfica, fijaremos un tamaño de caché pequeño (p.j. 1KB), un tamaño de línea relativamente pequeño (p.ej. 16 bytes) e iremos trazando la tasa de fallos que obtenemos con distintas asociatividades (1, 2, 4, ...). Debería ponerse de manifiesto que los aumentos iniciales del nivel de asociatividad tienen una influencia sobre la tasa de fallos que van disminuyendo a medida que la asociatividad aumenta, hasta llegar al punto de quedar estancados.

En la segunda gráfica seleccionaremos una caché de correspondencia directa y un tamaño de línea pequeño o mediano (32 bytes, 64 bytes, ...) e iremos trazando la tasa de fallos que obtenemos con distintos tamaños de la caché. De forma similar a la gráfica anterior, típicamente los aumentos iniciales del tamaño de la caché son los que más ayudan a reducir la tasa de fallos, y al llegar a cierto tamaño ésta tiende a estancarse.

Por último debemos estudiar el efecto del tamaño de la línea sobre la tasa de fallos de una caché unificada de correspondencia directa de pequeño tamaño. Los aumentos iniciales deberían ayudar a reducir la tasa de fallos, pero a partir de cierto tamaño ésta debe empezar a aumentar.

¿Puedes explicar estos efectos?

2. Restantes días

Esta guía de actividades está acompañada por una descripción de una serie de técnicas software que pueden ayudar a mejorar la localidad espacial y temporal de los programas y por tanto reducir su tiempo de ejecución, o al menos, como es el caso de la prebúsqueda, ayudar a ocultar la latencia de los accesos a memoria. Estas técnicas también habrán sido explicadas en clase de teoría.

Proponemos que los restantes días de prácticas de laboratorio de la asignatura se dediquen a comprender la aplicación de estas técnicas. Para el análisis de cada técnica, cada grupo podrá realizar las siguientes actividades:

1. Escribir códigos en C en los que se aplican las distintas técnicas de optimización y comparar el comportamiento de la versión optimizada y la original. Debe intentarse escoger códigos que resalten la bondad de la optimización, a ser posible a mayores de los proporcionados por los profesores. No obstante también es de interés estudiar códigos en los cuales se suponía que una técnica dada iba a mejorar el rendimiento y no lo hace para comprender el motivo de este comportamiento. Como ejemplos se proporcionan los de las figuras de esta guía de prácticas así como un código de ejemplo sin optimizar y optimizado para cada técnica en `http://www.des.udc.es/~basilio/DOCEN/page1/page4/page4.html` que incluye mediciones del tiempo de ejecución de cada versión de los códigos.

2. Hacer simulaciones con dineroIII utilizando las trazas[1] generadas mediante una versión instrumentada de los códigos confeccionada siguiendo lo explicado en el apéndice II empleando varias configuraciones distintas de caché. Se buscarán combinaciones de códigos y cachés donde se pueda observar una mejora al aplicar las técnicas. En concreto deberían compararse las tasas de fallos obtenidas antes y después de aplicar la optimización.

3. Medir el tiempo de ejecución de los códigos (sin instrumentación) en el ordenador `torno` que se esté usando del laboratorio 0.3, que tiene un Athlon. Deberán compararse los tiempos de ejecución del código antes y después de aplicar la optimización. Sugerimos que en las compilaciones se use el flag `-O` o `-O2` para que el compilador aplique algunas optimizaciones básicas.

4. Medir el rendimiento de ambas versiones del código (sin instrumentación) utilizando la librería PAPI que está instalada en los ordenadores `torno` del laboratorio 0.3. Esta herramienta permite acceder a contadores hardware que miden diversos tipos de eventos durante la ejecución de aplicaciones. El apéndice III contiene una introducción al uso de esta librería.

Lo importante no es simplemente experimentar con las transformaciones de código que proponemos midiendo sus resultados, sino razonar sobre los motivos de los comportamientos observados a fin de comprender cómo y por qué funcionan las diversas técnicas y la interacción entre nuestros códigos, las transformaciones del compilador y la forma en que funciona el hardware del computador. Algunas preguntas pueden ser genéricas e igualmente aplicables a todas las técnicas. Ejemplos de preguntas de este tipo son:

- ¿Qué condiciones deben verificar un código y la jerarquía de memoria sobre la que se ejecuta para que cada técnica de optimización sea efectiva en ellos?
- Si para un código dado una optimización mejora la tasa de fallos en cierta configuración de la caché pero no lo hace en otra, ¿cuál puede ser el motivo?

[1] Por defecto, dineroIII sólo procesa las primeras 10000000 referencias que recibe y se detiene. Asegúrate de generar trazas con menos referencias, o bien aumenta el número máximo de referencias a simular mediante la opción -z (mira el manual).

- ¿Cómo es posible que si se observa una reducción de la tasa real de fallos aplicando cierta técnica a cierto código (por ejemplo usando `PAPI`) sin embargo aumente el tiempo total de ejecución?
- ¿Cómo afecta la prebúsqueda hardware que tienen hoy en día muchos procesadores (p.ej. los Pentium 4 y los Athlon) a las distintas técnicas de optimización?

Otras preguntas pueden ser específicas de cada técnica. Ejemplos de preguntas de este tipo son:

- Técnica de rellenado y alineado de estructuras:
  - ¿Por qué si se accede sencuencialmente a un array de estructuras la técnica de rellenado y alineado de estructuras no sólo no nos ayuda sino que empeora el comportamiento del programa?
  - ¿Es útil rellenar las estructuras si no se alinea el array?
  - ¿Y es útil alinear el array si no se rellenan las estructuras?
  - ¿Cuánto debería uno rellenar en la técnica de rellenado y alineado de estructuras? Es decir, ¿hay alguna regla sencilla para estimar la cantidad óptima de relleno que se debe efectuar en cada estructura?
- Técnica de rellenado de arrays
  - ¿Cuál es efecto de rellenar en las distintas dimensiones de un array?
  - ¿Vale rellenar en cualquier dimensión?
  - ¿Hay alguna regla sencilla para determinar cuánto relleno aplicarle a un array?
- Técnica de partición en bloques
  - ¿Mejora o empeora el tiempo de ejecución con respecto al código sin bloques cuando se tiene un bloque muy pequeño (por ejemplo de 1, 2, 3 elementos)?
  - ¿Y si el bloque es muy grande (por ejemplo casi del tamaño de la matriz)?
  - Así pues, ¿Cómo influye el tamaño del bloque en la tasa de fallos y en el tiempo de ejecución en la técnicas de partición en bloques?
  - ¿Se podría estimar de forma aproximada un tamaño de bloque óptimo o de los mejores? ¿Cómo?
- Técnica de prebúsqueda
  - ¿Es beneficioso prebuscar sistemáticamente todos los datos que va a requerir el procesador?
  - ¿Mejora el rendimiento si se prebuscan datos que están a punto de ser usados?
  - ¿Y si se prebuscan datos que no van a usarse hasta dentro de mucho tiempo?
  - ¿Es benificioso prebuscar para cualquier patrón de acceso o sólo para algunos? Por patrón de acceso entendemos el acceso aleatorio (`a[1]`, `a[89]`, `a[26]`, ...), el acceso a datos consecutivos (`a[1]`, `a[2]`, `a[3]`, ...) , el acceso a una serie de datos que están separados por una distancia constante (`a[1]`, `a[101]`, `a[201]`, ldots), etc.

Técnicas para mejorar el comportamiento caché

Una vez localizada la sección de datos y estructuras de datos que pueden causar una degradación del rendimiento, existen varias técnicas para modificar los programas que pueden ayudar a eliminar parte de los fallos caché de los mismos.

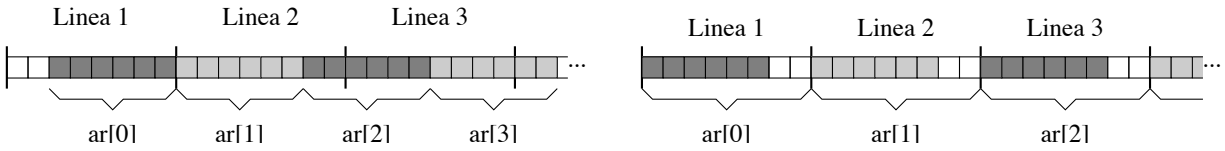
Las modificaciones que más conviene realizar en un programa dependen del tipo de fallos caché a eliminar. Los fallos de conflicto pueden reducirse mediante la unión de arrays, el rellenado y alineado de estructuras, el rellenado de arrays, el empaquetado de arrays y estructuras, el intercambio de bucles y la partición en bloques. Las cuatro primeras técnicas cambian la localización de las estructuras de datos, mientras que las dos últimas modifican el orden en el que las estructuras de datos son accedidas. Los fallos de capacidad pueden ser eliminados mediante modificaciones en el programa de modo que reutilice los datos antes de que estos sean reemplazados de la caché, como la fusión de bucles, la partición en bloques, el empaquetado de arrays y estructuras, y el intercambio de bucles. Otra posibilidad para aliviar el peso de las latencias de acceso a memoria en el tiempo de ejecución de un código es solaparlas con computaciones aplicando la técnica de prebúsqueda. A continuación se describen estas técnicas:

1. **Fusión de arrays**. Algunos programas referencian al mismo tiempo dos o más arrays de la misma dimensión usando los mismos índices. En el caso de que las posiciones correspondientes de dichos arrays estuvieran mapeadas a los mismos conjuntos de la caché esto daría lugar una serie de fallos de conflicto sistemáticos. Una situación típica en que da este efecto es en el caso de arrays con un tamaño que es una potencia de dos y se han declarado de forma consecutiva en el programa. La fusión de arrays mezcla múltiples arrays en un único array compuesto. En lenguaje C esto se realiza declarando un array de estructuras en lugar de dos o más arrays.

```
/*Declaracion original de dos arrays */
int val[TAMAÑO];
int key[TAMAÑO];

/* Nueva declaracion con array de estructuras */
struct miregistro {
         int val;
         int key;
       }
struct miregistro union_array[TAMAÑO]
```

2. **Rellenado y Alineado de estructuras**. Las referencias a una estructura de datos que se encuentra situada a caballo de dos bloques caché puede provocar dos fallos caché, aún cuando la estructura en si sea menor que el tamaño de bloque. Así, en el ejemplo mostrado en la izquierda de la figura inferior puede verse un array `ar` con registros que ocupan 6 bytes que comienza en una posición arbitraria de memoria en un sistema con líneas de caché con una capacidad de 8 bytes, correspondiendo cada cuadrado en la figura a un byte. Puede apreciarse que si por ejemplo se accediese a los componentes `ar[0]` y `ar[2]` sería necesario traer las líneas de memoria 1, 2 y 3 a la caché para satisfacer dichos

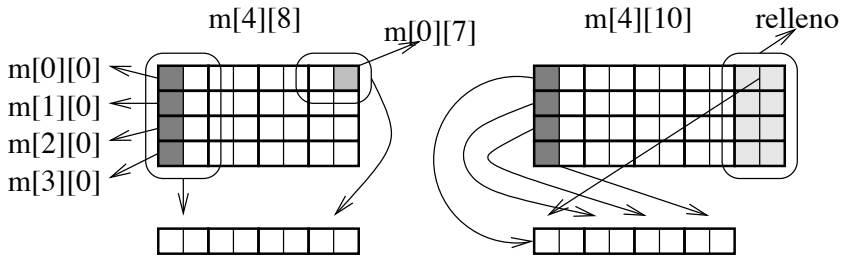
accesos. Rellenando las estructuras de modo que su tamaño sea un múltiplo del tamaño de bloque y alineándolas con los límites de los bloques pueden eliminarse los fallos de "desalineación". El rellenado es fácil de realizar en C declarando campos extra de relleno. El alineado es más difícil, dado que la dirección de una estructura debe ser un múltiplo del tamaño de bloque caché. Las estructuras declaradas estáticamente generalmente requieren apoyo del compilador. Las estructuras asignadas dinámicamente pueden ser alineadas por el programador usando aritmética de punteros. Obsérvese que algunas funciones de asignación de memoria (como algunas versiones de *malloc( )*) devuelven direcciones de memoria alineadas con un bloque caché. Siguiendo con nuestro ejemplo, a la derecha de la figura puede verse que una vez se ha garantizado que el array `ar` está alineado con el incio de una línea de la caché y que cada registro del mismo tiene 8 bytes, el acceso a los componentes `ar[0]` y `ar[2]` ya sólo requiere el acceso a las líneas de memoria 1 y 3.




```
/*Declaracion original de una estructura de seis bytes */
struct ex_struct {
         short int val1, val2, val3;
      }


/*Declaracion de estructura alineada a un tamaño de
  bloque-cache de 8 bytes*/
struct ex_struct {
         short int  val1, val2, val3;
         char pad[2];
      }


/*Asignacion original que no garantiza alineamiento */
ar = (struct ex_struct *) malloc(sizeof(struct ex_struct)
      * TAMAÑO);


/*Nuevo codigo que garantiza el alineamiento de
  estructuras para bloques de BLOCK bytes */
ar = (struct ex_struct *) malloc(sizeof(struct ex_struct)
      * TAMAÑO + BLOCK);
ar = (struct ex_struct *) ((( (int) ar + BLOCK -1) / BLOCK ) * BLOCK);
```

3. **Rellenado de arrays**. Cuando no es posible acceder de forma secuencial a los contenidos de un array, puede darse el caso de que debido a las dimensiones del mismo los accesos sobre él generen fallos de conflicto de forma sistemática. Un ejemplo típico de esta situación es el acceso por columnas a una matriz bidimensional en C (puesto que en

C los arrays bidimensionales se almacenan por filas) que tiene un tamaño de fila que es una potencia de dos. En esta situación muchas o incluso todas las líneas que contienen los elementos de cada columna de la matriz recaen en el mismo conjunto de la caché, haciendo imposible su reuso para cuando se va a acceder a la siguiente columna. La figura inferior ilustra este problema para una caché de correspondencia directa con 4 líneas en la que cada línea puede contener dos de los elementos del array `m[4][8]`. Como puede verse en la ilustración de la izquierda, en esta situación las líneas de memoria que contienen los datos de una columna dada del array están todas mapeadas a la misma línea de la caché, con lo que cada vez que se va acceder a un nuevo elemento de una columna, la línea asociada al elemento precedente debe ser expulada de la caché para hacerle sitio. De esta forma, cuando se va a acceder a la siguiente columna, es imposible reusar ninguna de las líneas que se trajeron durante el procesamiento de la columna precedente y deben volver a traerse todas de la memoria. Este problema puede resolverse rellenando el array aumentando el tamaño de la fila de la matriz, de forma que las líneas de cada columna se distribuyan entre más conjuntos. En la ilustración de la derecha hemos pasado a declarar `m` como una matriz `m[4][10]` aunque realmente sólo sigamos usando sus 8 primeras columnas. El cometido de las dos columnas de relleno es simplemente que, como puede apreciarse, gracias a ellas ahora cada línea de una columna de la matriz va a parar a una línea diferente de la caché. De esta forma es posible reusar todas cuando se va a proceder a procesar la siguiente columna de `m`.



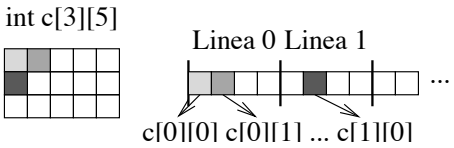

```
/*Declaracion original de una matriz con una fila
 con un tamaño multiplo o divisor del de la cache */
float m[4][8];

/*bucle que recorre el array por columnas*/

for(j=0; j<8;j++)
  for(i=0; i<4; i++)
    q = q * 1.2 + m[i][j];

/*Declaracion nueva de un array cuyas filas no
  tienen un tamaño multiplo o divisor del de la cache  */
float m[4][10];
```

4. **Empaquetado**. El empaquetado o compactación es lo opuesto al rellenado: empaquetando un array en el menor espacio posible. En el siguiente ejemplo el programador observa que el valor de los elementos del array nunca son mayores que 255, y por tanto puede

almacenarlos en un *unsigned char*, que requiere 8 bits, en lugar de un *unsigned int* que normalmente ocupa 32 bits. En una máquina con un tamaño de bloque de 16 bytes, el código del ejemplo permite almacenar 16 elementos por bloque, en lugar de 4, reduciendo el número máximo de fallos de caché en un factor de 4.

```
/*Declaracion original de un array de enteros sin signo */
unsigned int valores[10000];

/*bucle que recorre el array */
for(i=0; i<10000; i++)
    valores[i] = i % 256;

/*Declaracion nueva de un array de caracteres sin signo */
unsigned char valores[10000];
```

5. **Intercambio de bucles**. Algunos programas tienen bucles anidados que acceden a los datos en memoria en un orden no secuencial. Simplemente intercambiando el orden del anidamiento se puede conseguir que el código acceda a los datos en el orden en que están almacenados. Por ejemplo, en FORTRAN el compilador almacena los datos por columnas, con lo que el orden ideal de acceso de los datos es columna a columna. Por el contrario, en C y C++ los datos se almacenan consecutivamente a lo largo de la última dimensión de un array, a continuación a lo largo de la antepenúltima, y así consecutivamente. Así, en el caso de un array bidimensional, los datos de cada fila están almacenados en posiciones consecutivas en memoria, por lo que un acceso por columnas sería contraproducente, puesto que entre cada dos elementos consecutivos en una columna tenemos almacenados todos los de una fila en la memoria. La figura inferior ilustra esta explicación.




```
/*Antes*/
for( j=0; j<N; j++)
    for( i=0; i<M; i++)
        x[i][j] = 2 * x[i][j];

/*Despues*/
for( i=0; i<M; i++)
    for( j=0; j<N; j++)
        x[i][j] = 2 * x[i][j];
```

6. **Fusión de bucles**. Los programas numéricos a menudo consisten en varias operaciones sobre los mismos datos, codificadas como múltiples bucles sobre los mismos arrays.

   La fusión de bucles es lo contrario de la fisión de bucles, una transformación del programa que divide porciones independientes de un bucle en dos bucles separados. La fisión de

```
/*Bucles separados*/
for( i=0; i<N; i++)
    for( j=0; j<N; j++)
        a[i][j] = 1 / b[i][j] * c[i][j];

for( i=0; i<N; i++)
    for( j=0; j<N; j++)
        d[i][j] = a[i][j] + c[i][j];

/*Bucles fusionados*/
for( i=0; i<N; i++)
    for( j=0; j<N; j++) {
        a[i][j] = 1 / b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

bucles ayuda al compilador a detectar bucles para explotar el hardware de vectores en algunos supercomputadores.

7. **Partición en bloques (Blocking)**. La partición en bloques es una técnica general de reestructuración de un programa para reutilizar bloques de datos que se encuentran en la caché antes de que sean reemplazados. Imaginemos un programa con varios arrays, algunos accedidos por columnas y otros por filas. Almacenando los datos por filas o columnas (según la característica del compilador) no resolvemos nada, pues tanto filas como columnas son utilizadas en cada iteración. Por esta misma razón, el intercambio de bucles tampoco nos sirve en este caso.

```
/* Antes */
for( x=0; x<N; x++)
    for( y=0; y<N; y++) {
         acum = c[x][y];
         for( z=0; z<N; z++)
             acum += a[x][z] * b[z][y];
         c[x][y] = acum;
       }

/* Particion en bloques */
for( yy=0; yy<N; yy+=T)
    for( zz=0; zz<N; zz+=T)
        for( x=0; x<N; x++)
            for( y=yy; y<min(yy+T,N); y++) {
                 acum = c[x][y];
                 for( z=zz; z<min(zz+T,N); z++)
                     acum += a[x][z] * b[z][y];
                 c[x][y] = acum;
             }
```

Para asegurar que los elementos que están siendo accedidos están en la caché, el código original debe de ser modificado para trabajar con una sub-matriz de tamaño T×T, de modo que los dos bucles internos realicen los cálculos en pasos de tamaño T en vez de recorrer de principio a fin los arrays A y B. T es el llamado factor de bloqueo.

Como puede apreciarse, esta técnica conlleva el aumento del tamaño del código así como del número de cálculos a efectuar. Así, dependiendo del valor de T puede suceder que si bien el número de fallos de caché se reduzca, el tiempo total de ejecución del programa aumente debido a los ciclos de CPU adicionales requeridos. Por ello debe intentarse escribir el código generado por esta técnica de la manera más óptima posible.

```
 /* Particion en bloques optimizada*/
for( yy=0; yy<N; yy+=T) {
    lim_y = min(yy+T, N);
    for( zz=0; zz<N; zz+=T) {
        lim_z = min(zz+T, N);
        for( x=0; x<N; x++)
            for( y=yy; y<lim_y; y++) {
                acum = c[x][y];
                for( z=zz; z<lim_z; z++)
                    acum += a[x][z] * b[z][y];
                c[x][y] = acum;
             }
    }
}
```

8. **Prebúsqueda**. Esta técnica en lugar de buscar reducir el número de fallos de la caché, intenta solapar la latencia de acceso a memoria de los mismos con computaciones de la CPU. Consiste en solicitar a la memoria datos en cuyo acceso se supone que hay muchas probabilidades de que se produzcan fallos caché con antelación al uso de los mismos, de manera que el procesador pueda ejecutar otras instrucciones mientras estos son traídos de la memoria.

```
/* Codigo sin prebusqueda */
for( p = lista; p != NULL; p = p->sig )
    computa(p);

/* Codigo con prebusqueda para gcc */
for( p = lista; p != NULL; p = p->sig ) {
    __builtin_prefetch(p->sig);
    computa(p);
}
```

9. **Evitar la compartición falsa**. En programas con múltiples threads de ejecución que se ejecuten en sistemas multiprocesador y/o multinúcleo deben evitarse situaciones en las cuales diferentes threads estén actualizando posiciones de memoria localizadas en la misma línea caché. El motivo es que la línea caché define la granularidad de los protocolos

hardware involucrados en el mantenimiento de la coherencia de las copias de los datos en las distintas cachés. Así, siempre que un thread que se esté ejecutando en un cierto procesador modifique una posición en una cierta línea caché, todos aquellos otros procesadores que tengan copias de esa línea deberán ser informados por el protocolo hardware y realizar las consiguientes actualizaciones o invalidaciones dependiendo del protocolo utilizado. Y este proceso, que es relativamente costoso, ocurre independientemente de que los threads que se ejecutan en esos otros procesadoresrealmente nunca utilizen la porción de la línea que es modificada por otros threads.

```
/*Con comparticion falsa*/
double resultadoparcial[NUMTHREADS]; /* compartido */
void f(int minumthread)
{ int i;
  for( i = 0; i < M; i++)
    resultadoparcial[minumthread] += f(i);
}

/*Sin comparticion falsa*/
#define DOUBLESPORLINEA (TAMANO_LINEA / sizeof(double))
double resultadoparcial[NUMTHREADS * DOUBLESPORLINEA]; /* compartido */
void f(int minumthread)
{ int i;
  for( i = 0; i < M; i++)
    resultadoparcial[minumthread * DOUBLESPORLINEA] += f(i);
}

/*Otra forma minimizando comparticion falsa */
double resultadoparcial[NUMTHREADS]; /* compartido */
void f(int minumthread)
{ int i;
  double temporal = resultadoparcial[minumthread];
  for( i = 0; i < M; i++)
    temporal += f(i);
  resultadoparcial[minumthread] = temporal;
}
```

APÉNDICE I: COMENTARIOS ADICIONALES SOBRE EL `dineroIII`

- El simulador DineroIII lee trazas en formato *din*.

label	address
0	7d50
2	3808
0	41c0
3	9774
3	977c
0	c2f0
0	c300
0	4c7c
0	4c80
...	

  label indica el tipo de acceso de una referencia.

0	lectura dato
1	escritura dato
2	busqueda instrucción
3	acceso de tipo desconocido
4	caché flush

  address es la dirección en hexadecimal entre **0** y **ffffffff**.

- Los parámetros de la caché se introducen en la línea de comando. Obligatorios:
  - Tamaño del bloque: -b16
  - Tamaño de la caché unificada: -u16K
  - Caché separada: -i16384, -d1M
- Por defecto la caché es de correspondencia directa (asociatividad = 1) -a1, LRU.
- Ejemplo para procesar la traza en el fichero ale_0.prg:

```
dineroIII -b32 -u4K < ale_0.prg
```

- Varios niveles de caché:

`cat ficherotraza | dineroIII -o2` *arg1* `| awk -f twolevel.awk | dineroIII` *arg2*

Interpretación de la salida de dineroIII

La salida típica de dineroIII tiene la forma:

```
Metrics                 Access Type:
(totals,fraction)       Total   Instrn   Data    Read   Write    Misc
-----------------       ------  ------  ------  ------  ------  ------
Demand Fetches           20000    1452   18548   18041       0     507
                        1.0000  0.0726  0.9274  0.9021  0.0000  0.0254

Demand Misses              797     361     436     406       0      30
                        0.0398  0.2486  0.0235  0.0225  0.0000  0.0592

Words From Memory         6376
( / Demand Fetches)     0.3188
Words Copied-Back            0
( / Demand Writes)      0.0000
Total Traffic (words)     6376
( / Demand Fetches)     0.3188
```

Significados relevantes:

- En la primera columna de arriba a abajo vemos que hubo un total de 20000 accesos, de los cuales resultaron en fallos 797, lo que constituye el 3,98 % del total. Es decir, la tasa de fallos fue del 3,98 %. Estos 797 fallos trajeron a la caché del nivel inferior de la jerarquía de memoria un total de 6376 palabras, siendo la palabra de 4 bytes siempre en este simulador.
- La segunda columna indica que 1452 de los 20000 accesos (es decir, el 7.26 %) fueron accesos a instrucciones. 361 de ellos resultaron en fallos. Así, la tasa de fallos dentro del conjunto de accesos a instrucciones fue $361/1452 \times 100 = 24{,}86\,\%$.
- La tercera columna indica que 18548 de los 20000 accesos (es decir, el 92.74 %) fueron accesos a datos. De ellos fallaron 436, siendo la tasa de fallos dentro del conjunto de accesos a datos $436/18548 \times 100 = 2{,}35\,\%$
- La cuarta columna indica que en concreto 18041 de los accesos a datos fueron lecturas, constituyendo el 90,21 % de los accesos totales. De ellos 406 accesos resultaron en fallo, con lo que la tasa de fallos en las lecturas de datos fue $406/18041 \times 100 = 2{,}25\,\%$.
- Según la quinta columna, no hubo ninguna escritura.
- La sexta columna indica los accesos misceláneos (los del tipo 3), que no revisten interés al ser siempre una parte mínima del total. En este caso hubo 507, lo que constituye el $2,54\,\%$ del total. De ellos 30 resultaron en fallos, con lo que la tasa de fallos dentro de este grupo fue del $30/507 \times 100 = 5{,}92\,\%$

APÉNDICE II: INSTRUMENTACIÓN PARA LA OBTENCIÓN DE TRAZAS

Para obtener la traza de accesos a datos generadas por un programa en C, deben seguirse estos tres pasos:

1. Inclusión en el fichero C de la declaración de las macros[2]

   #define traza_rd(iindex) printf("0 %lx \n", (iindex))
   #define traza_wr(iindex) printf("1 %lx \n", (iindex))

   que se usarán respectivamente para generar la impresión de los accesos de lectura y de escritura. Nota: Es importante dejar un espacio en blanco entre el 0 y el 1 y el símbolo de porcentaje.

2. Inserción de una llamada **traza_rd(&(X[expr1][expr2]...))** precediendo cada línea de código donde se efectúe una lectura sobre **X[exp1][expr2]...**.

3. Comportamiento recíproco frente a la escritura usando **traza_wr** tras cada acceso de este tipo.

Nótese que sólo instrumentamos los accessos a vectores, matrices y en general arrays multidimensionales, y a punteros, pues asumimos que las variables escalares están alojadas en registros del procesador y por tanto sus accesos no afectan a la caché.

Por otro lado, dado que la salida estándar del programa va a ser alimentada como traza de entrada para *dineroIII*, el programa no debe imprimir nada por pantalla, puesto que dicha impresión también sería redireccionada al simulador.

Una vez compilado el programa, su salida puede redireccionarse al simulador usando el carácter `|`. Así, por ejemplo, si el ejecutable de nuestro programa se llama `a.out`, pasaríamos la traza al simulador para ver su comportamiento en una caché unificada de correspondencia directa de 2KB con 64 bytes por línea mediante

```
a.out | dineroIII -u2K -b64 -a1
```

[2]Dado que emplean `printf`, debe incluirse el archivo de cabecera `stdio.h` en el código antes de las macros.

APÉNDICE III: Performance Application Programming Interface (PAPI)

La librería PAPI (`http://icl.cs.utk.edu/papi/`) permite a los programadores observar la correlación entre sus códigos, su rendimiento y los eventos que suceden en el programa durante la ejecución de sus aplicaciones. La librería tiene un interface de bajo nivel y otro de alto nivel, siendo éste último el que proponemos usar en las prácticas. Se pueden descargar o leer on-line los manuales completos de la librería de su sitio WWW. Para los propósitos de nuestras prácticas recomendamos consultar la guía de usuario, y en menor medida, la guía software, de los URL:

`http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.htm`
`http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_Software_Spec.htm`

En concreto, la partes de mayor interés son la introducción, y la explicación sobre el uso del interface de alto nivel de la guía de usuario.

No obstante, para la realización de nuestras prácticas basta con utilizar el esqueleto de programa del código de ejemplo proporcionado por los profesores de la asignatura en

`http://www.des.udc.es/~basilio/DOCEN/page1/page4/assets/ejemplo1.c`

Este programa verifica que los contadores hardware están disponibles, y los programa para medir el número de ciclos, de instrucciones y de fallos de accesos a datos en las cachés del primer y del segundo nivel durante la ejecución de dos porciones de código. Se pueden encontrar otros eventos medibles en la referencia de programación y en la especificación software de PAPI. Puedes aprovechar este programa para medir con él el número de fallos en distintas cachés y ciclos de ejecución de los distintos códigos que escribas durante las prácticas. Para compilar este código usa el comando[3]:

```
gcc -O -I/opt/papi-3.5.0/include/ -L/opt/papi-3.5.0/lib/
    -o ejemplo1 ejemplo1.c -lpapi
```

Antes de ejecutarlo debes asegurarte de

1. haber arrancado el sistema en Linux usando el kernel que ha sido parcheado con el parche `PerfCtr`, pues la librería lo requiere para su funcionamiento. En el caso de que no sepas si el kernel que estás usando tiene el parche o no, usa el comando `uname -r`, que informa de la versión del sistema operativo. Si la cadena `perfctr` forma parte de la versión, indica que el kernel tiene el parche; en caso contrario debes reiniciar con el kernel correcto.

2. que tu variable de entorno `LD_LIBRARY_PATH` contenga el directorio `/opt/papi-3.5.0/lib/`, puesto que contiene una librería dinámica que el ejecutable carga durante su ejecución. De no ser así, si tu *shell* es `ksh` o `bash` puedes añadirla mendiante el comando

   `export LD_LIBRARY_PATH=/opt/papi-3.5.0/lib/:${LD_LIBRARY_PATH}`

   Sugerimos que para no tener que teclear este comando en cada sesión, podrías añadirlo a tu fichero de inicialización del *shell*.

[3]Esstas instrucciones se refieren al directorio `/opt/papi-3.5.0` pero el nombre del directorio va variando según la versión de PAPI instalada. Haz un `ls /opt` para ver de cuál dispones