Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 3 – Solución propuesta

Profesor: Yadran Francisco Eterovic Solano

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice 'solución propuesta' por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Precalentamiento

1. Explique los principios de localidad espacial y localidad temporal con sus propias palabras, dando un ejemplo para cada caso.

   El principio de localidad espacial nos dice que si un dato es accedido, es probable que se soliciten datos cercanos al mismo. Un ejemplo de esto sería un elemento de un arreglo, con todos sus componentes contiguos en memoria.

   Por otra parte, el principio de localidad temporal nos dice que si un dato es accedido, es probable que se vuelva a solicitar el mismo en el corto plazo. Un ejemplo de esto sería una variable utilizada dentro de un ciclo.

2. **(I3 - I/2017)** Sin considerar el precio, ¿por qué no tiene sentido usar una caché infinita?

   El objetivo principal de la memoria caché es disminuir el tiempo de acceso, aprovechando los principios de localidad espacial y temporal. Si se tuviera una caché infinita, a la larga se terminarían por copiar todos los bloques de la memoria principal a las líneas de la caché. Esto implicará un tiempo de búsqueda similar de una dirección dentro de la memoria, por lo que no se sacaría provecho de los principios antes mencionados y se tendría una ganancia en la eficiencia prácticamente nula.

3. Suponga que tiene una memoria principal de 16 bytes, y una memoria caché de 8 bytes y 4 líneas. Además, asuma que tiene un programa que accede, en este orden, a las direcciones de la memoria principal: 0,1,5,7,10,13,4,6.

   Obtenga el estado final de la memoria caché (en una tabla) y el hit-rate para cada una de las siguientes funciones de correspondencia:

a. Directly mapped.

b. Fully associative.

c. 2-way associative.

Puede asumir una política de reemplazo LRU, en caso de necesitarla.

Antes de comenzar con el desarrollo, notaremos ciertos elementos que nos permitirán avanzar más rápido en cada ejercicio:

- Al ser la memoria de 16 bytes, tendremos 2^4 direcciones de datos posibles. Es decir, las direcciones son de 4 bits.
- Al ser la memoria caché de 8 bytes, y poseer 4 líneas, tendremos $\frac{8}{4} = 2^1$ palabras (bytes) por línea. Es decir, posicionamos cada palabra (offset) con un solo bit, y para obtener la línea correspondiente, al ser 2^2, necesitamos de 2 bits.

Para un mejor entendimiento de esta parte, usaremos colores para identificar cada elemento en una dirección:

- Posición
- Línea
- Tag
- Conjunto

a. **Directly mapped:** Cada bloque de la memoria principal tiene una línea asociada en la caché. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Por otra parte, para obtener la línea correspondiente, usamos los dos bits siguientes. Los bits restantes (en este caso, solo el bit más significativo) serán el tag. Entonces, veamos cada acceso:

   1) 0 = 0000. Obtenemos un miss al tener un bit de validez = 0. Guardamos el tag 0 en la línea 00, almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   2) 1 = 0001. Obtenemos un hit al encontrar el dato en la posición correspondiente.
   3) 5 = 0101. Obtenemos un miss al tener un bit de validez = 0. Guardamos el tag 0 en la línea 10, almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   4) 7 = 0111. Obtenemos un miss al tener un bit de validez = 0. Guardamos el tag 0 en la línea 11, almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   5) 10 = 1010. Obtenemos un miss al tener un bit de validez = 0. Guardamos el tag 1 en la línea 01, almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   6) 13 = 1101. Obtenemos un miss al ver que en la línea 10 y en la posición 1 de ella, el tag difiere del nuestro (¡tenemos datos distintos!). Reescribimos el tag 1 en la línea 10 y los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea.

7) $4 = 0100$. Obtenemos un miss al ver que en la línea 10 y en la posición 1 de ella, el tag difiere del nuestro. Reescribimos el tag 0 en la línea 10 y los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea.
8) $6 = 0110$. Obtenemos un hit al encontrar el dato en la posición correspondiente.

Finalmente, tenemos un hit-rate de $\frac{2}{8}$, y el estado de la caché de la siguiente forma:

Índice línea	Ubicación palabra	Bit validez	Tag	Dato
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	1	1	Mem[10] Mem[11]
10	0 1	1	0	Mem[4] Mem[5]
11	0 1	1	0	Mem[6] Mem[7]

b. **Fully associative:** Cada bloque de la memoria principal puede ser asociado a cualquier línea de la caché. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Los bits restantes (en este caso, los tres que quedaron) serán el tag. Acá, además será necesario tener constancia del tiempo en que cada línea fue accedida por última vez, para poder utilizar la política de reemplazo LRU. Entonces, veamos cada acceso:

- $0 = 0000$. Obtenemos un miss al no encontrar el dato en ninguna línea. Guardamos el tag 000 en la línea 00 (primer espacio disponible), almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 1.
- $1 = 0001$. Obtenemos un hit al encontrar el dato en la línea 00. Actualizamos el tiempo de acceso a la línea a 2.
- $5 = 0101$. Obtenemos un miss al no encontrar el dato en ninguna línea. Guardamos el tag 010 en la línea 01 (primer espacio disponible), almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 3.
- $7 = 0111$. Obtenemos un miss al no encontrar el dato en ninguna línea. Guardamos el tag 011 en la línea 10 (primer espacio disponible), almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 4.
- $10 = 1010$. Obtenemos un miss al no encontrar el dato en ninguna línea. Guardamos el tag 101 en la línea 11 (primer espacio disponible), almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 5.
- $13 = 1101$. Obtenemos un miss al no encontrar el dato en ninguna línea. Al quedarnos sin espacio, por LRU reescribimos la línea accedida hace más tiempo. Reescribimos el tag 110 en la línea 00, almacenando los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea. Finalmente, actualizamos el tiempo de acceso a la línea a 6.

- $4 = 0100$. Obtenemos un hit al encontrar el dato en la línea 01. Actualizamos el tiempo de acceso a la línea a 7.
- $6 = 0110$. Obtenemos un hit al encontrar el dato en la línea 10. Actualizamos el tiempo de acceso a la línea a 8.

Finalmente, tenemos un hit-rate de $\frac{3}{8}$, y el estado de la caché de la siguiente forma:

Índice línea	Ubicación palabra	Bit validez	Tag	Dato	Tiempo de acceso
00	0 1	1	110	Mem[12] Mem[13]	6
01	0 1	1	010	Mem[4] Mem[5]	7
10	0 1	1	011	Mem[6] Mem[7]	8
11	0 1	1	101	Mem[10] Mem[11]	5

c. **2-way associative:** Cada bloque de la memoria principal está asociado a conjuntos de dos líneas cada uno, pudiendo ser almacenado en cualquiera de estas. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Para obtener el conjunto, al poseer 4 líneas y 2 líneas por conjuntos, tendremos solo 2, por lo que nos bastará un bit para identificarlo (el que se encuentra después del de posición). Los bits restantes (en este caso, los dos que quedaron) serán el tag. Acá, además será necesario tener constancia del tiempo en que cada línea fue accedida por última vez, para poder utilizar la política de reemplazo LRU. Entonces, veamos cada acceso:

- $0 = 0000$. Obtenemos un miss al no encontrar el dato en ninguna línea del conjunto 0. Guardamos el tag 00 en la línea 00 del primer conjunto (primer espacio disponible), almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 1.
- $1 = 0001$. Obtenemos un hit al encontrar el dato en la línea 00 del conjunto 0. Actualizamos el tiempo de acceso a la línea a 2.
- $5 = 0101$. Obtenemos un miss al no encontrar el dato en ninguna línea del conjunto 0. Guardamos el tag 01 en la línea 01 del conjunto 0 (primer espacio disponible), almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 3.
- $7 = 0111$. Obtenemos un miss al no encontrar el dato en ninguna línea del conjunto 1. Guardamos el tag 01 en la línea 10 del conjunto 1 (primer espacio disponible), almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 4.
- $10 = 1010$. Obtenemos un miss al no encontrar el dato en ninguna línea del conjunto 1. Guardamos el tag 10 en la línea 11 del conjunto 1 (primer espacio disponible), almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 5.

- 13 = 1101. Obtenemos un miss al no encontrar el dato en ninguna línea del conjunto 0. Al quedarnos sin espacio, por LRU reescribimos la línea accedida hace más tiempo dentro del conjunto (ya que debemos mantener consistencia con el mapeo a conjuntos). Reescribimos el tag 11 en la línea 00, almacenando los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea. Finalmente, actualizamos el tiempo de acceso a la línea a 6.
- 4 = 0100. Obtenemos un hit al encontrar el dato en la línea 01 del conjunto 0. Actualizamos el tiempo de acceso a la línea a 7.
- 6 = 0110. Obtenemos un hit al encontrar el dato en la línea 10 del conjunto 1. Actualizamos el tiempo de acceso a la línea a 8.

Finalmente, tenemos un hit-rate de $\frac{3}{8}$, y el estado de la caché de la siguiente forma:

Conjunto	Índice línea	Ubicación palabra	Bit validez	Tag	Dato	Tiempo de acceso
0	00	0 1	1	11	Mem[12] Mem[13]	6
	01	0 1	1	01	Mem[4] Mem[5]	7
1	10	0 1	1	01	Mem[6] Mem[7]	8
	11	0 1	1	10	Mem[10] Mem[11]	5

4. **(I3 - I/2016)** Comente sobre las ventajas de tener una memoria caché split en vez de una unified conectada a la memoria de datos del computador básico.

   No existe ninguna ventaja si está conectada a la **memoria de datos**. La gracia de la caché split es poder tener separada la memoria de datos con la de instrucciones. Conectarla directamente a la memoria de datos no tiene una utilidad práctica, ya que utilizaríamos la mitad del espacio en instrucciones inexistentes.

Preguntas

1. a. **(I3 - I/2016)** Un computador tiene una memoria caché de 16KB, con líneas de 32 bytes que almacenan 8 palabras, y un tiempo de acceso de 10ns. La memoria caché está conectada a la memoria principal mediante un bus capaz de transferir 8 bytes en 120ns. ¿Cuál es el hit-rate que debe tener la memoria caché para tener un tiempo de acceso promedio de 20ns?

   Nos guiamos por la siguiente fórmula:

   $$TP = HR * HT + (1 - HR) * (HT + MP)$$

   Donde:

   - TP : Tiempo promedio.
   - HR : Hit-rate.
   - HT : Hit-time.
   - MP : Miss penalty.

   Ahora, agrupando términos, tenemos que:

   $$TP = HT + (1 - HR) * MP$$

   Como el tiempo de acceso es 10ns, $HT = 10ns$, y como la transferencia del bus es de 120ns por 8 bytes, $MP = \frac{32}{8} * 120ns = 480ns$. Finalmente, como queremos que $TP = 20ns$:

   $$20ns = 10ns + (1 - HR) * 480$$
   $$HR = \frac{470}{480} \approx 0{,}98$$

   Nota: Recordar que el tiempo de acceso al tener un miss incluye el tiempo de acceso en la caché, ya que se busca el dato de todas formas, pero no se encuentra.

   b. **(I3 - I/2015)** Un computador de 64 bits tiene una memoria caché de 32KB, con 1024 líneas de 32 palabras. ¿Cuánto espacio de esta caché es usado por información distinta de los datos?

   Primero, notemos que esto depende del tipo de función de asociación que tengamos. Para poder ejemplificar bien, asumamos que tenemos una función 4-way associative.

   - Al tener 2^5 palabras por línea, necesitamos 5 bits para identificar una específica dentro de una línea en especial (offset).
   - Al ser 4-way associative, tendremos conjuntos de 4 líneas. Si tenemos 2^{10} líneas, entonces tendremos 2^8 conjuntos (es decir, necesitamos 8 bits para identificar el conjunto).
   - Finalmente, usamos 13 bits para posicionar (5 bits offset + 8 bits conjunto), por lo que el resto lo utilizamos para el tag: 51 bits.

   Lo que almacenaremos en la tabla serán finalmente los 51 bits del tag para cada línea (pues todas las palabras en la línea lo comparten), además de un bit de validez (para ver si el dato fue escrito efectivamente desde la memoria principal). Finalmente, tendremos un espacio total de 52 * 1024 bits = 6.5 KB.

Veamos qué habría pasado si hubiera sido fully associative: Solo utilizaríamos los 5 bits de posicionamiento dentro de una línea (ya que todo bloque de memoria puede ser asignado a cualquier línea de la caché). Entonces, ahora nuestro tag tendrá 59 bits y, por ende, incluyendo el bit de validez ocupará un espacio de 60 * 1024 bits = 7.5 KB.

Puede realizar el mismo análisis para otros tipos de asociación (directly mapped, 2-way associative, etc.).

c. **(I3 - I/2017)** Considere una memoria caché *fully-associative*, con *hit-time* igual a $16L^3 - 100L$ ns, donde L es la cantidad de líneas de la caché. Si el *hit-rate* de esta memoria es de 0.95, ¿cuál es la cantidad de líneas que genera el tiempo de acceso promedio mínimo?

Se hará uso de la misma fórmula antes descrita:

$$TP(L) = HR * HT(L) + (1 - HR) * (HT(L) + MP)$$

Hay que notar dos detalles importantes para la resolución:

1) Al estar el *hit-time* en función de la cantidad de líneas en la caché, el tiempo promedio también lo está.
2) El *miss-penalty* es constante, debido a que corresponde simplemente al tiempo que toma copiar un bloque de la memoria principal en una línea de la caché.

Agrupando términos:

$$TP(L) = HT(L) + (1 - HR) * (MP)$$

De esta forma, para encontrar el mínimo, derivamos e igualamos a cero:

$$TP'(L) = HT'(L) = 0$$

Los términos $(1 - HR)$ y MP desaparecen al ser constantes.

Reemplazando:

$$TP'(L) = 48 * L^2 - 100 = 0$$

Despejando:

$$L = \sqrt{\frac{100}{48}} = 1{,}443$$

Como necesitamos un número entero para la cantidad de líneas, vemos el valor de $HT(1)$ y $HT(2)$:

$$HT(1) = 16 * 1^3 - 100 * 1 = -84$$

$$HT(2) = 16 * 2^3 - 100 * 2 = -72$$

Aquí notamos algo importante: Para los valores de L más cercanos al mínimo, el *hit-time* es **negativo**, lo que no puede suceder en la práctica[1]. Como el tiempo va aumentando con

[1] Salvo que conozcan un mecanismo para que su computador haga cómputos en el pasado.

el incremento en el número de líneas (según lo observado con los cálculos anteriores), nos quedamos con el mínimo valor L para el que $HT(L) \geq 0$:

$$HT(3) = 16 * 3^3 - 100 * 3 = 132$$

Por lo tanto, el número de líneas que minimiza el tiempo promedio es $L = 3$, con un *hit-time* promedio de 132 ns.

d. **(I3 - I/2015)** Considere un computador con microarquitectura Von Neumann, donde la tasa de ciclos de clock por instrucción es igual a N, cuando todos los accesos a memoria producen hits en la caché. La memoria caché tiene miss-rate de 4 % y miss-penalty de 25*N ciclos de clock. Si en un programa de K instrucciones, el 50 % de estas realizan lectura de un dato en memoria, ¿cuántos ciclos de clock menos tomaría la ejecución del programa, si todas las instrucciones produjeran hits en la memoria caché?

Primero, veamos el mejor caso: Solo tenemos hits, por lo que la cantidad de ciclos de clock total que se tendrá será igual al número de instrucciones por el número de ciclos por instrucción, es decir:

$$Ciclos_{mejor\_caso} = N * K$$

Ahora, tomemos el caso real: Tenemos un miss-rate de un 4 %. Esto significa que en un 4 % de nuestras instrucciones tendremos un miss. Ahora, cuando tenemos un miss, **también** se ejecuta la cantidad de ciclos de un hit (ya que se buscó, en primer lugar, el dato en la caché).

Entonces, hasta ahora nuestro número de ciclos se ve de la siguiente forma:

$$Ciclos_{real} = N * K + Ciclos_{miss}$$

Notemos que la cantidad de ciclos será de 25*N por cada miss. Como estamos trabajando con la arquitectura Von Neumann, tendremos **al menos** un acceso a memoria por instrucción (necesitamos obtener el opcode de la memoria). Entonces, tenemos en primera instancia $0,04 * K * 25 * N$ ciclos añadidos. Luego, nos dicen que en la mitad de las instrucciones, además, se tienen lecturas de memoria, por lo que en la mitad de las instrucciones tendremos **dos accesos a memoria**. Por ende, debemos incluir una cantidad de ciclos equivalente a $\frac{1}{2} * 0,04 * K * 25 * N$. Entonces:

$$Ciclos_{miss} = 0,04 * K * 25 * N + 0,02 * K * 25 * N = 0,06 * K * 25 * N = 1,5 * N * K$$

Finalmente, reemplazamos:

$$Ciclos_{real} = N * K + 1,5 * N * K = 2,5 * N * K$$

Concluimos entonces que se tendrán $1,5 * N * K$ ciclos menos en el caso ideal.

2. a. **(I3 - I/2013)** El principio de localidad espacial explica en parte el buen funcionamiento de la memoria caché. Sin embargo, es posible no cumplir este principio, disminuyendo el rendimiento de la memoria. Describa un ejemplo específico de esto y explique por qué se produce.

   Un ejemplo particular de esto es el caso de almacenar una matriz dentro de la memoria caché, guardando en cada línea una fila de esta, pero sin que quepan todas en el espacio disponible. Asumiendo lo anterior, si recorriéramos la matriz por columna, tendríamos el caso de que por cada acceso a un elemento en particular, copiaríamos la fila completa. Esto, ya que en el acceso anterior se agregó la fila según lo estipulado al principio **y no la columna que contiene al siguiente dato que busca ser accedido**. Entonces, tendremos un rendimiento de memoria mucho peor en comparación con el que se tendría al recorrer la matriz por filas.

   b. **(I3 - I/2017)** Suponga que está escribiendo una subrutina que realiza el producto punto entre cada par de vectores de un conjunto que contiene N de estos. Explique detalladamente qué consideraciones se deben tomar para maximizar el hit-rate.

   Nota: Con detalladamente, se espera que otra persona sea capaz de replicar los resultados.

   La idea principal será modificar la dirección del espacio de memoria de los vectores de forma que podamos aprovechar el principio de localidad espacial. Sin pérdida de generalidad, se asumirá que cada vector posee 2 dimensiones y se definirá un ejemplo en `Python` como sigue:

```python
# Se define la clase principal a utilizar.
class VectorCoords:
    def __init__(self, dimension):
        self.dimension = dimension
        self.values = []

    def add_value(self, value):
        self.values.append(value)

# Se crean las clases de ejemplo
x_dim = VectorCoords('x')
y_dim = VectorCoords('y')
# Asumimos que se tiene un archivo de texto donde cada
# linea representa un vector.
length = 0
with open("example.txt",'r') as f:
    line = f.readline()
    while line:
        length += 1
        # Separamos las coordenadas del vector.
        vector = line.split(',')
        # Agregamos los valores.
        x_dim.add_value(vector[0])
        y_dim.add_value(vector[1])
        line = f.readline()
```

```
# Obtenemos el producto punto
x_prod = 1
y_prod = 1
# Al ejecutarlo de esta forma, se traen a memoria primero
# los elementos de x, teniendo mas hits. Lo mismo al
# hacerlo luego para y.
for i in range(length):
    x_prod *= x_dim.values[i]
for j in range(length):
    y_prod *= y_dim.values[j]
result = x_prod + y_prod
```

A grandes rasgos, lo que se hace es crear una nueva clase que agrupe todos los valores de una dimensión particular para un conjunto de vectores, lo que permitirá realizar el producto entre estos de forma directa, para finalmente hacer la suma. Al agrupar el conjunto de valores a multiplicar en una misma clase, por el principio de localidad todos serán almacenados en bloques contiguos, aumentando el número de *hits* al escribir la mayoría de estos en la caché[2].

c. **(I3 - I/2015)** ¿Cómo es el rendimiento de una memoria caché, si el patrón de accesos a memoria distribuye de manera uniforme sobre todas las posibles direcciones? Ejemplifique el o los posibles casos.

Antes de ejemplificar, notemos que una distribución uniforme implica que se accede a cada posición en memoria en la misma proporción, sin importar el orden en el que se haga. Sabiendo esto, podemos ver dos casos generales. Para ejemplificar mejor, asumiremos una memoria caché con líneas de 8 palabras:

- **Caso 1:** Se accede a los datos de memoria en forma secuencial: 0,1,2,3,...,N. En este caso, al partir en el acceso 0, tendremos un miss, pero guardaremos en la caché los datos desde la posición 0 a la 7 en la primera línea, por lo que en los accesos 1,2,...,7 solo tendremos hits. Luego, tendremos lo mismo: Tratamos de acceder a la posición 8 con un miss, guardamos en la siguiente línea los datos desde la posición 8 a la 15, y en los accesos tendremos 9,10,...,15 solo tendremos hits. Esto se repetirá constantemente hasta el último dato, por lo que tendremos un hit-rate muy alto.
- **Caso 2:** Se accede a los datos de memoria de la siguiente forma: 0,8,16,...,N-7,1,9,17,...,N-6,...,N. Al partir en el acceso 0, tendremos un miss, y guardamos en la caché los datos desde la posición 0 a la 7 en la primera línea. Luego, en el acceso 8, tendremos otro miss, almacenando los datos desde la posición 8 a la 15, y así sucesivamente. Si no contáramos con el espacio suficiente en la caché para toda la memoria accedida, y siguiéramos una política de reemplazo LRU, las primeras líneas utilizadas serían sobreescritas, y al llegar al acceso 1, tendríamos un miss nuevamente por la sobreescritura. Esto, finalmente, conlleva a un hit-rate prácticamente nulo.

Bajo estos ejemplos, llegamos a la conclusión de que no es posible catalogar el rendimiento dada la información del enunciado, ya que puede ser muy alto o muy bajo, dependiendo del caso.

[2] Notar que este ejercicio es bastante libre, pueden haber más formas de aumentar el número de hits (por ejemplo, agrupando todos los vectores en una sola clase). Para más soluciones posibles, revisen la sección 4.2 de los apuntes de Jerarquía de Memoria.

d. **(I3 - I/2017)** La contención de bloques es un problema del esquema de mapeo directo, donde 2 o más bloques pelean por la misma línea, existiendo otras líneas no utilizadas en la caché. ¿Existe un problema similar en el esquema *N-way*? Si su respuesta es negativa, justifíquela y, si es positiva, indique detalladamente un caso en que esto se de.

   Sí, existe un problema similar, pero en menor grado. Al ser un bloque asociado a un **conjunto** de líneas, podría darse el caso en el que una secuencia de accesos genere que uno solo de estos conjuntos se llene rápidamente y sea necesario efectuar las políticas de reemplazo, a pesar de existir otros conjuntos con espacio disponible.

   Un ejemplo de esto sería la secuencia 0,4,8,0,4,8,0,4,8,... para una memoria principal de 16 bytes, una memoria caché de 8 bytes y 4 líneas, una política de reemplazo LRU y, por último, un esquema *2-way associative*[3]. En este caso, los accesos a 0 y 4 harán que las dos líneas del conjunto **0** se ocupen y, luego, por LRU se irá reemplazando la primera línea ocupada del mismo conjunto ya que el acceso a 8 será mapeado al mismo conjunto de los dos anteriores. Una secuencia como la descrita generará reemplazos constantes en el conjunto **0**, dejando al conjunto **1** prácticamente vacío.

e. **(I3 - II/2014)** El algoritmo de reemplazo **MRU** (Most Recently Used), a diferencia de LRU, descarta primero los elementos que han sido ocupados más recientemente. ¿En qué casos podría ser útil el uso de este esquema?

   Un caso específico en el que puede ser útil, es en la iteración constante de un arreglo. Supongamos que el arreglo completo no cabe en la caché, así que una vez que se nos agote el espacio, reemplazamos la última posición iterada por la siguiente a acceder. Hacemos esto hasta llegar al final, y luego, al reiniciar, tendremos en memoria una porción considerable del mismo, dándonos un buen hit-rate (que concluirá al llegar a la posición donde se realizó la primera reescritura).

   Notar que este caso convendría siempre y cuando la memoria caché pueda contener más de la mitad del arreglo (en otro caso, el miss rate sería mayor).

f. **(I3 - II/2014)** Describa al menos dos posibles soluciones para el problema de consistencia de memoria que se genera al tener un esquema de escritura de caché write-back.

   - **Solución 1:** Tener un bit dentro de la memoria principal que indique si el dato fue modificado dentro de la caché o no (dirty bit). Entonces, al querer revisar dicha posición, si se tiene un dirty bit igual a 1, se puede realizar una interrupción que realice una sobreescritura en la memoria principal desde la caché, devolviéndole al bit su estado original igual a 0. De esta forma, la inconsistencia se arreglaría solo cuando fuera necesario.
   - **Solución 2:** En general, tenemos problemas cuando otros dispositivos tratan de acceder de forma directa a la memoria (DMA). Entonces, podemos hacer que la DMA interactúe de forma directa con la caché, por lo que siempre tendría datos consistentes (sin embargo, en caso de necesitar datos almacenados en la memoria principal, sería necesario aplicar políticas de reemplazo).

[3] Notar que es el mismo ejemplo resuelto en el precalentamiento.