Taller de Caché

Organización del Computador 1

1. Introducción

El presente taller consiste en una serie de ejercicios en los cuales se deberá realizar el seguimiento del estado de diferentes tipos de caché. Será posible utilizar el simulador de caché para validar los resultados. Luego, se implementará una nueva política de desalojo dentro del simulador y se realizarán comparaciones.

2. El simulador

Para el taller implementamos una versión super beta de un simulador de Caché. El simulador se puede descargar de la página de la materia. Éste implementa tres tipos de caches posibles y tiene la restricción de que la memoria que usa siempre direcciona a **palabra**. El simulador consiste de varias clases de Python explicadas a continuación.

Para utilizar el simulador se debe descargar el archivo `cache.py` y desde una consola –en el directorio donde se encuentra `cache.py`– ejecutar las siguientes instrucciones para abrir la consola interactiva de Python e importar todas las clases del simulador:

```
# Abre una nueva sesion interactiva de Python
ipython

# Luego, dentro de la consola de Python, importar las clases del simulador
from cache import *
```

2.1. Clases de Caches implementadas

- **CacheCorrespondenciaDirecta**

  Esta clase simula una Caché de correspondencia directa. Sus parámetros son:

  - *memory*: Lista de enteros que representan el contenido de la memoria
  - *cacheSize*: Tamaño de la caché (cantidad de unidades direccionables)
  - *nLines*: Número de líneas de la caché

  Por ejemplo, para crear un simulador de caché de correspondencia directa con: tamaño de caché de 64 bytes, dos líneas y memoria principal de 1 KB direccionable a byte con todos ceros, se debe hacer:

  ```
  memory = [0 for i in range(2**10)]
  c = CacheCorrespondenciaDirecta(memory=memory, cacheSize=64, nLines=2)
  ```

- **CacheTotalmenteAsociativa**

  Esta clase simula una Caché totalmente asociativa y como parámetros toma:

  - *memory*: Lista de enteros que representan la memoria

- *cacheSize*: Tamaño de la caché (cantidad de unidades direccionables)
- *nLines*: Número de líneas de la caché
- *cacheAlg*: Función que implementa el algoritmo de sustitución

Por ejemplo, para crear un simulador de cache totalmente asociativa con: tamaño de cache de 32 *bytes*, 4 líneas, algoritmo de sustitución FIFO y memoria principal de 1 MB direccionable a byte con todos ceros, se debe hacer:

```
memory = [0 for i in range(2**20)]
c = CacheTotalmenteAsociativa(memory=memory, cacheSize=32,
                              nLines=4, cacheAlg=FIFO)
```

- **CacheAsociativa_NWays**

  Esta clase simula una Caché asociativa de N vías y toma como parámetros:

  - *memory*: Lista de enteros que representan la memoria
  - *cacheSize*: Cantidad de *bytes* de la caché
  - *nWays*: Cantidad de vías
  - *nSets*: Cantidad de sets
  - *cacheAlg*: Función que implementa el algoritmo de sustitución

  Por ejemplo, para crear un simulador de caché asociativa de 4 vías de tamaño de cache de 64 *bytes*, 2 sets, algoritmo de sustitución MRU y memoria principal de 1 MB con todos ceros, se debe hacer:

```
memory = [0 for i in range(2**20)]
c = CacheAsociativa_NWays(memory=memory, cacheSize=64, nWays=4,
                          nSets=2, cacheAlg=MRU)
```

2.2. Métodos implementados

Cada clase de caché simulada tiene algunos métodos implementados.

`c.fetch(`*`address`*`)`
: Este método simula un pedido a memoria, toma como parámetro la dirección o *address* del dato a buscar y devuelve dicho valor.

`print c`
: Imprime el estado actual de la cache.

`c.infoCache(line=N)`
: Permite inspeccionar el contenido de la línea N de cache, direcciones que abarca, etc.

`c.mostrarLog()`
: Muestra el historial de operaciones realizadas por la cache (*miss* o *hit* producido por cada fetch).

`c.hitRate()`
: Devuelve la tasa de *hits*

2.3. Ejemplo de uso

A continuación presentamos un ejemplo de uso de la cache.

Memoria

B0	**0**	**1**	**2**	**3**
	0x0	0xA	0x0	0x0
B1	**4**	**5**	**6**	**7**
	0x0	0x0	0xF	0x0
B2	**8**	**9**	**10**	**11**
	0x0	0x0	0x5	0x0
B3	**12**	**13**	**14**	**15**
	0x0	0x0	0x0	0x0

Cache

Indice → Tag ↓	0	1	2	3
-	-	-	-	-
-	-	-	-	-

```
from cache import *

# Esto es un comentario en python

# Creamos una memoria de solo 16 bytes
memory = [0,0xA,0,0,0,0,0xF,0,0,0,0x5,0,0,0,0,0]

# Creamos la memoria cache
ca = CacheTotalmenteAsociativa(memory=memory, cacheSize=16, nLines=2,
    cacheAlg=FIFO)

# Imprimo el estado actual de la cache
print ca

#  ----------------Step:-1---------------
#|               Invalido                |
#|               Invalido                |
#  ---------------------------------------
#Hit rate: -
```

La cache almacena el valor del **step**, que se aumenta cada vez que se hace un fetch, de este modo, podemos tener registro de en qué paso de la ejecución se creó y usó la cache. Como todavía no se realizó un fetch, este valor por ahora es -1.

A continuación, hacemos un fetch de la dirección 0x1. El estado de la cache actual será entonces:

Cache

Indice → Tag ↓	0	1	2	3
0	0x0	0xA	0x0	0x0
-	-	-	-	-

```
# Accedo a la direccion 0x1 y obtengo el dato correspondiente
ca.fetch(0x1)
# 0xA
```

```
# Veo como cambio la cache
print ca

#  ----------------Step:0--------------
#|         Tag:0 First:0 Change:0         |
#|               Invalido                 |
#  ----------------------------------------
#Hit rate: 0.0

# Me fijo el historial de accesos y veo que se produjo un miss
ca.mostrarLog()

#|   Step: 0    |    Miss    |    Linea: 0    |   Direccion: 1    |

# Y miro el contenido de la linea 0
ca.infoCache(line=0)

#Linea 0:
#Dir. pedida  1
#Valido  True
#Tag  0
#Step 1st use  0
#Step change  0
#Linea (Dir. Memoria y contenido):
#| 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 |
#| 0   | 10  | 0   | 0   | 0   | 0   | 15  | 0   |
```

Y ahora cargamos otra dirección más, que producirá otro miss.

Cache

Indice → Tag ↓	0	1	2	3
0	0x0	0xA	0x0	0x0
1	0x0	0x0	0x5	0x0

```
# Hago el fetch
ca.fetch(0xA)
# 0x5

# Me fijo como quedo la cache
print ca

#  ----------------Step:1--------------
#|         Tag:0 First:0 Change:0         |
#|         Tag:1 First:1 Change:1         |
#  ----------------------------------------
```

```
# Veo el nuevo miss
ca.mostrarLog()

#|  Step: 0    |    Miss    |   Linea: 0    |     Direccion: 1    |
#|  Step: 1    |    Miss    |   Linea: 1    |     Direccion: 10   |
```

3. Ejercicios

3.1. Ejercicio 1 - Seguimiento de Caché

3.1.1. Caché de correspondencia directa

Considerar la máquina de ORGA1 (palabras y direccionamiento de 16 *bits*, memoria de 128 KB), y con una memoria caché de correspondencia directa de 128 B con líneas de 32 B. Suponiendo que la caché comienza vacía, determinar para la siguiente lista de accesos si se produce un hit o un miss en la caché completando la siguiente tabla.

Se accede a las siguientes direcciones de memoria en este orden: 0x0009, 0x001D, 0x000A, 0x0101, 0x0113, 0x000A, 0x001E, 0x0102, 0x0114

Dirección	Tag	Linea (*bits*/decimal)	Indice	Direcciones de la línea	Hit/Miss
0x0009					
0x001D					
0x000A					
0x0101					
0x0113					
0x000A					
0x001E					
0x0102					
0x0114					

NOTA: completar la tabla manualmente, sin utilizar el simulador. Una vez completa, verificar los resultados con el mismo.

> **Utilización del simulador**
> Para validar las respuestas es posible utilizar el simulador.
>
> Para crear la cache, tener en cuenta que el simulador maneja todos los tamaños en palabras. Para inicializar la memoria, utilice la siguiente función.
>
> ```
> memoria = range(0, tamanio)
> ```
>
> Para visualizar los campos que utiliza la caché para una direccion de memoria en particular, es posible utilizar la función $getFields(address)$
>
> ```
> # muestra los campos de cache usados para la direccion de memoria 0x0009
> ca.getFields(0x0009) # te regalamos una solucion =)
>
> # tag: 0 set: 0 index: 9
> ```

3.1.2. Caché completamente asociativa

Utilizando el simulador y la misma lista de accesos a memoria que en el punto anterior, complete la siguiente tabla utilizando a una memoria caché completamente asociativa de 128 B, líneas de 32 B y una política de desalojo FIFO.

Dirección	Tag	Indice	# Linea	Direcciones de la línea	Hit/Miss
0x0009					
0x001D					
0x000A					
0x0101					
0x0113					
0x000A					
0x001E					
0x0102					
0x0114					

a) ¿En qué casos funciona mejor una memoria completamente asociativa frente a una de correspondencia directa? Dé un ejemplo.

b) ¿Qué pasa si sólo uso caché para los datos? ¿Y si sólo la uso para el código?

3.2. Ejercicio 2 - Políticas de desalojo

En este ejercicio queremos evaluar el impacto de usar distintas políticas de desalojo.

3.2.1. Implementación de LRU

El simulador permite configurar las memoria caché para que utilice diferentes políticas de desalojo. Actualmente se encuentran implementadas las politicas FIFO y RANDOM. Puede verse su implementación en el archivo `politicas.py`

Para implementar una política se utilizan los siguientes atributos asociados a una línea (o set/vía) de cache:

- `valid`: Es un flag que establece si la línea es válida o no
- `stepChange`: Guarda en que *step* se accedió por última vez una línea
- `stepFirstUse`: Guarda en que *step* se cargó en la cache esta línea
- `address_req`: Guarda la dirección que se pidió para llenar esta línea
- `mem`: Guarda los valores de memoria que se almacenaron en esta línea

Utilizando las políticas ya existentes en el simulador, implemente una nueva política: Least-Recently-Used (LRU) que desaloje la línea que fue utilizada hace más tiempo. Opcionalmente, implemente también la política MRU (Most-Recently-Used). Como ejemplo, pueden observar la implementación de la política FIFO en `politicas.py`.

3.2.2. Hit Rate

Para analizar el comportamiento de la cache utilizando las distintas politicas existentes, podemos medir el *hit rate* que se obtiene para una serie de accesos a memoria. Para ello, tomaremos los accesos producidos por dos variantes de un mismo algoritmo, que se encarga de recorrer una matriz cuadrada y realizar una cierta operación. A continuación, podemos ver el pseudo-código de cada caso:

```
<operaciones>
for i=1:64:
  for j=1:64:
    <operaciones con acceso a matriz(i,j)>
    <mas operaciones con acceso a matriz(i,j)>
    <otras operaciones>
```

```
<operaciones>
for i=1:64:
  for j=1:64:
    <operaciones con acceso a matriz(i,j)>
    <operaciones con acceso a matriz(j,i)>
    <otras operaciones>
```

Los accesos a memoria asociados a cada ejecución se encuentran en los archivos `benchmark_matrix_igual.list` y `benchmark_matrix_mix.list`, respectivamente.

Para una configuración de memoria de 128KB y una caché totalmente asociativa configurada como en el ejercicio anterior, se pide:

a) Medir el *hit rate* que se produce para ambos códigos, con las políticas FIFO, RANDOM y LRU.

b) Explique la diferencia de performance de la cache encontrada entre ambos códigos, independientemente de la política utilizada.

c) Explique cuál es el beneficio que obtiene entre utilizar FIFO y LRU, tras analizar el hit rate en ambos casos.

Para evaluar el hit rate asociado a cada secuencia de accesos, tras inicializar la cache con la configuración correspondiente, utilizar la función $fetchFrom$, que toma como parametro el nombre de un archivo y ejecuta $fetch$ para cada una de las direcciones definidas en el archivo y luego calcular el *hit rate* utilizando $hitRate$.

```
# Traigo de memoria todas las direcciones en el order definido
    ↪ en el archivo elegido
ca.fetchFrom('benchmark_matrix_igual.list')

# Corroboro el hitRate de la cache
ca.hitRate()
```

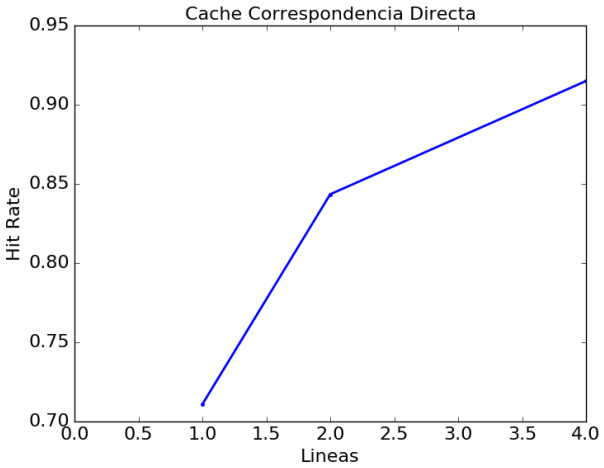
3.3. Ejercicio 3 - Análisis de Caché

Si quisiéramos decidir qué cantidad de líneas ponerle a nuestra caché, podemos utilizar el simulador para verificar el comportamiento esperado de antemano. Por ejemplo, se pueden graficar los resultados del hit rate para una serie de accesos a memoria (representativos de un código típico), mediante el siguiente código:

```
from cache import *
import numpy as np
import pylab

# Correspondencia directa, vario distintas lineas y guardo la
    ↪ hit rate
# para cada configuracion usando la secuencia de fetchs
    ↪ definida en
# benchmark.list
dom = [1,2,4]
res = []
for l in dom:
    ca = CacheCorrespondenciaDirecta(memory=range(0, 2**16),
    cacheSize=128, nLines=l)
    ca.fetchFrom('benchmark.list')
    res.append(  ca.hitRate() )
    print "Lineas:", l, "HitRate:", ca.hitRate()

# res tiene los distintos valores de hit rates para cada
    ↪ configuracion
# uso pylab para plotear
pylab.plot(dom,res,'.-',lw=2,label='
    ↪ CacheCorrespondenciaDirecta - FIFO')
pylab.xticks(size=16)
pylab.yticks(size=16)
pylab.xlim([0,np.max(dom)])
pylab.xlabel('Lineas',size=16)
pylab.ylabel('Hit Rate',size=16)
pylab.title('Cache Correspondencia Directa',size=16)
pylab.show()
```



- A partir del resultado que se observa, ¿se puede decir que a mayor cantidad de lineas, mejor funcionamiento de la cache? ¿Para verificar su hipótesis, que pasa si tenemos más de 16 líneas (**nota:** al menos debe quedar un bit para índice)? Explique qué sucede.