Práctica Cache

Dalia Camacho, Gabriela Vargas, Luis Alpízar, Erick Chávez

Marzo 2019

1 Introducción

La memoria Caché surge con el fin de atender las discrepancias en el desarrollo de dos componentes fundamentales en la arquitectura de computadoras: el CPU y la Memoria Principal (RAM). Derivado de la diferencia entre la velocidad de procesamiento de instrucciones del CPU y la capacidad de almacenamiento de la RAM, surge la necesidad de compensar un problema importante: el procesador ahora es capaz de procesar una mayor cantidad de instrucciones y datos que lo que la memoria RAM es capaz de alimentar. En este sentido, resulta importante mejorar la velocidad de acceso y búsqueda de palabras en la RAM. Lo anterior lo podemos lograr con la memoria Caché y las estrategias de jerarquía de memoria.

La jerarquía de memoria refiere al diseño de computadoras con más de una memoria. Tal diseño se visualiza por niveles donde las memorias que se encuentran en los niveles superiores refieren mayor cercanía con el procesador. Los niveles inferiores indican memorias más lejanas y resultan más tardadas de acceder, así, el Caché se ubica en el nivel superior mientras que la RAM se ubica en el inferior. Destaca que esta estrategia de jerarquía depende del tamaño de las memorias, por lo general, entre más alejadas estén las memorias del procesador, mayor tamaño suelen tener.

En la jerarquía de memoria, el Caché se ubica físicamente entre el CPU y la RAM. Por un lado, puede transmitir las palabras requeridas por el CPU en menor tiempo; por otro lado, puede copiar bloques de palabras de la RAM para su posterior uso.

Por lo anterior, se tienen principios de localidad que refieren al manejo de los accesos a memoria principal, los cuales tienen el objetivo de ayudar al procesador en la ejecución de programas. Los principios se dividen en dos:

- **Localidad temporal.** Tendencia a referenciar una misma localidad de memoria más de una vez en el corto plazo.
- **Localidad espacial.** Tendencia a referenciar localidades de memoria adyacentes en el corto plazo.

El Caché funciona al copiar información de la RAM y ponerla a disposición del CPU con el fin de otorgar rápido acceso. Para lograrlo, la RAM es dividida en bloques, es decir, se crean conjuntos de palabras de igual tamaño que posteriormente serán copiadas en el Caché. Esta división en bloques permite manejar la información de la RAM de manera ordenada y segura donde se presta especial atención al principio de localidad espacial.

De esta forma, el Caché realiza los accesos necesarios a memoria y trae los bloques que contengan las palabras que el procesador requiere. Esta tarea la debe realizar de manera periódica pues el Caché tiene un tamaño inferior a la RAM y debe reemplazar los bloques cuando la palabra requerida no se ha copiado previamente.

La siguiente figura muestra el uso de bloques entre las memorias descritas.

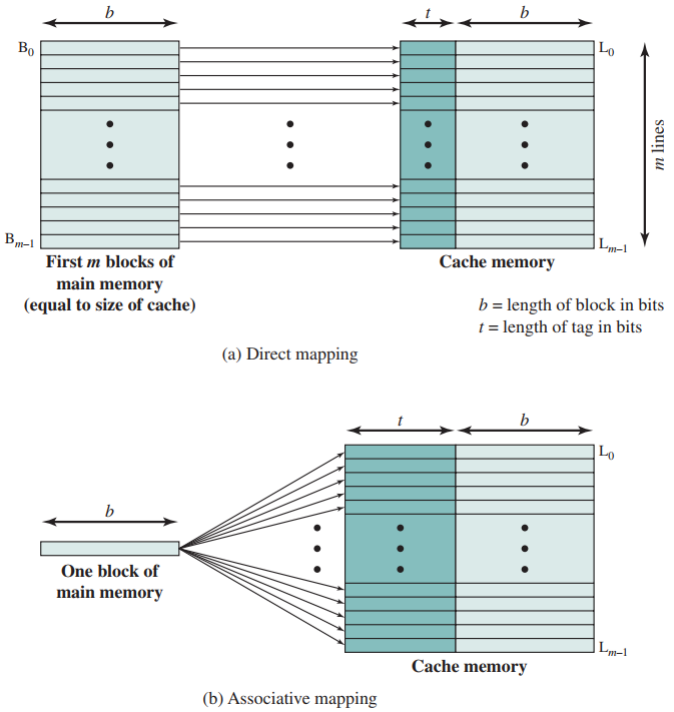

Figure 1: Estructura básica de una memaoria Cache tomada de Stallings [1]

Para determinar si la palabra se encuentra en el Caché, se realiza una comparación de la etiqueta de la palabra buscada con la etiqueta del bloque adecuado, cuando la comparación es positiva, se dice que se tiene un Hit, en otro caso, se tiene un Miss. Ante un miss, la memoria Caché debe reemplazar su bloque completo con el bloque en la RAM que contiene la palabra requerida. Estas acciones son importantes, pues mediante su análisis estadístico, podemos

comparar el desempeño de diversas arquitecturas de memoria.

La arquitectura de memoria Caché se puede representar de diferentes formas:

- Diseño de Caché
- Partición
- Estrategias de escritura
- Estrategias de reemplazo

En primer lugar, los diseños para representar la memoria Caché incluyen el mapeo directo, el mapeo asociativo por conjunto y el mapeo asociativo completo. El modelo más sencillo de memoria Caché es el mapeo directo, en el que cada localidad de memoria es copiada exactamente a una localidad particular del Caché. Este modelo tiene la ventaja de ser fácil de implementar, sin embargo, tiene el problema de constante colisión (miss) y reemplazo entre localidades. Un modelo mejor que el anterior es el asociativo por conjunto, en el que las localidades de memoria tienen ciertas ubicaciones específicas en Caché en las que pueden ser copiadas. Este modelo tiene la ventaja de reducir las colisiones entre localidades, sin embargo, tiene el problema de dividir el Caché por lo que debe hacer más comparaciones al reemplazar información. El modelo más conveniente en cuanto a almacenamiento es el de mapeo asociativo completo, en el que cada localidad de memoria puede copiarse en cualquier lugar del Caché. Este modelo tiene la ventaja de optimizar el copiado de localidades, sin embargo, ante un reemplazo requiere de comparaciones con todas las localidades, pues no existe orden en su guardado.

En segundo lugar, existen modelos diferentes de Caché respecto al guardado de su información. En este sentido, se tienen Cachés unificados y separados. Los Caché unificados tienen la característica de almacenar instrucciones y datos. La ventaja de este tipo es que incrementa la tasa hit. Los Caché separados dividen entre información y datos. La ventaja de este tipo es que facilitan la ejecución de programas al hacer esta distinción.

En tercer lugar, se definen estrategias de escritura ante un caso de hit o miss. Para el hit, se tienen opciones de escritura write through y write back. En el write through, se realiza la escritura tanto en Caché como en memoria principal. En el write back, se realiza la escritura solamente en Caché. Para el miss, se tienen opciones de no write allocate y write allocate. En el not write allocate, solamente se escribe en memoria principal sin cargar el bloque por reemplazar. En el write allocate, se busca y escribe el nuevo bloque. Las combinaciones de estrategias más comunes son write through y not write allocate o write back y write allocate.

Por último, algunas de las opciones que se tienen para el reemplazo de bloques son las siguientes: de manera aleatoria, Least Recently Used, First in

First Out, entre otros. Lo anterior va a depender del diseño del Caché, pues el mapeo directo no tiene alternativa y el reemplazo se realiza tradicionalmente.

A continuación, se realiza un simulador de la organización y desempeño de una memoria Caché. Se utiliza el lenguaje de programación C para simular arquitecturas de Caché y evaluar su desempeño al variar sus características. Adicionalmente, se atienden una serie de preguntas indicadas por el simulador.

2 Diseño

El propósito de la practica es entender el impacto que tienen los diferentes parámetros de un cache en el desempeño del mismo. Los parámetros flexibles en el trabajo son: tipo de arquitectura (Von Newman y Harvard), tamaño de cache, tamaño de bloque, asociatividad y políticas de lectura y escritura.

El desempeño del cache es medido con las siguientes métricas:

- Demmand fetches: número de veces que consultamos un bloque en cache.
- Hit: acceso a un bloque que está en cache.
- Miss: consulta de un bloque que no está en cache.
- Replacements: total de ocurrencias donde al haber un miss de conflicto, tenemos que cargar el bloque de memoria principal a cache, esto generá un remplazo.
- Copies back: total de ocurrencias donde al realizar una escritura, independientemente de la política escogida, tenemos que copiar el valor generado por el procesador a la memoria principal.

El simulador se implementa a través de 4 archivos. El archivo, main.c contiene los procedimientos necesarios para leer comandos de la terminal, los cuales fijan los parámetros del cache que se va a construir. Cache.c contiene las funciones necesarias para simular el comportamiento del cache con base en los parámetros seleccionados. Por último, tenemos dos archivos tipo *header*, main.h y cache.h, los cuales sirven como interface entre los archivos de main.c y cache.c. Así, la implementación del Caché se realiza con 2 estructuras.

La primera, *cache_line_*, es una lista doblemente ligada la cual abstrae la funcionalidad de los bancos de datos del cache. Esta estructura se compone de cuatro miembros.

- tag: representa el tag de la línea y banco
- dirty: utilizado en la política de *write back* para tener control de los bloques sobre los que se haya escrito. Al sustituirlos los tenemos que regresar la memoria principal para mantener la consistencia de información.

- Pcache_line *LRU_next: apuntador de tipo apuntador a cache_line,
- Pcache_line *LRU_prev:

La segunda estructura llamada *cache_* permite crear la simulación de esta memoria por lo que requiere de los siguientes elementos:

- Tamaño de la memoria Caché: calculado en bytes se recibe desde la línea de comando al ejecutar el programa
- Asociatividad: utilizado para determinar el grado de asociatividad para este tipo de diseño de Caché
- Número de sets del Caché: representa el número de líneas del Caché simulado al dividir el tamaño sobre la asociatividad y el tamaño de bloque
- Máscara para obtener el índice: auxiliar para obtener el índice de un set
- Máscara para obtener el offset: auxiliar para obtener el offset de un set
- Entradas válidas del Caché: representa todas las entradas contenidas en el Caché simulado
- Apuntador a cabeza de la lista de cada set: utilizado para implementar la política LRU en caso de reemplazo con diseño de Caché mapeo directo y asociativo
- Apuntador a cola de la lista de cada set: utilizado para implementar la política LRU en caso de reemplazo con diseño de Caché asociativo

Las funciones principales del Caché son $initCache()$ y $performAccess()$, las cuales junto con $flush()$ y $printStats()$ realizan las acciones que permiten medir el desempeño de los distintos escenarios de memorias Caché. En este sentido, los puntos relevantes de las funciones mencionadas son los siguientes.

La función *initCache()* tiene como objetivos definir el número de líneas en el cache, el tamaño de la máscara, crear las estructruras que conforman al cache e inicializar las estadísticas.

Para poder cubrir los casos de mapeo directo y mapeo *set-associative* creamos una función auxiliar *init_cache_spec()* en la que realizamos la inicialización de la estructura tipo cache.

A partir del tamaño del bloque, del tamaño del caché y la asociatividad se obtiene el número de líneas en el cache de la siguiente manera

$$líneas = \frac{Tamaño\ del\ Cache}{Asociatividad \cdot Tamaño\ del\ Bloque}.$$

Habiendo calculado el número de líneas obtenemos el número de bits necesarios (set) para identificar la línea.

$$set = log_2(líneas).$$

La máscara la obtenemos creando un arreglo de bits que desplaza un bit de uno en $set + offset$ posiciones. Con esto tenemos un arreglo de bits que tiene un uno en la posición $set + offset + 1$ (empezando el conteo con el bit menos signigicativo). Después le restamos uno, con lo que tenemos un arreglo de bits con unos únicamente de la posición 1 a la posición $set + offset$ (empezando el conteo con el bit menos significativo). Esto se escribe en una sola línea como:

$$mask = (1 << (set + offset)) - 1.$$

De esta manera si hacemos un *and* entre la máscara y alguna dirección los bits de la dirección correspondientes al tag quedan todos en cero. Esto último lo hacemos en la función *perform_access*.

Por último definimos las estructuras de tipo cache_line_. Reservamos espacio en la memoria para la estructura cache line que se utilizará para representar cada línea del cache. Para simular el caso de una memoria con mapeo directo o asociativo, utilizamos una lista doblmente ligada con dos vectores de apuntadores a su cabeza y cola. La figura 2 es una representación de las estructuras de datos empleadas para la implementación del caché. Como se mencionó, las listas doblemente ligadas abstraen la idea de los bancos del caché. En nuestro caso, la política de LRU se implementó a través de actualizar los tags y el dirty bit. Se realizó de está manera ya que al analizar las funciones previamente programadas *insert*() y *delete*() no se removian los nodos anteriores, lo que podía generear filtraciones de memoria. De está manera, sólo se inicializaron las listas al principio del programa y se fueron actualizando con base en la política LRU.

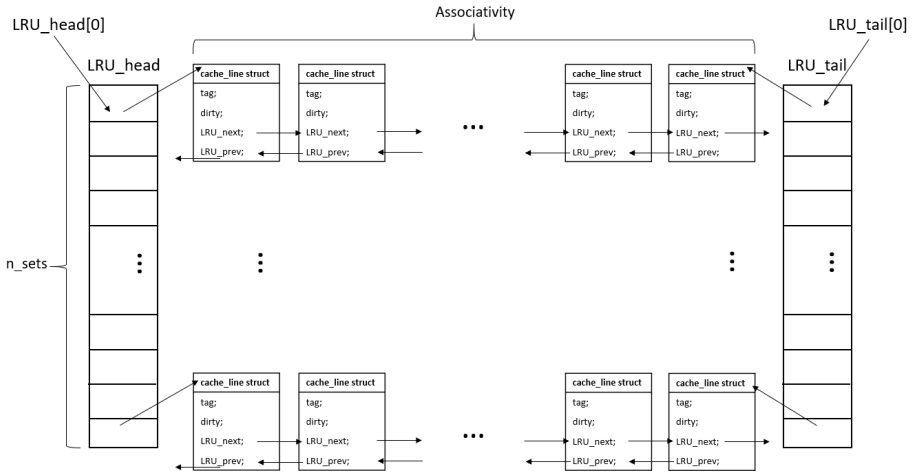

Figure 2: Diagrama de estructuras de datos

Al accesar al elemento k de arreglo dinámico LRU-head ($k - ésima$ línea del cache), se apunta al primer nodo de la lista doblemente ligada (es decir el

primer banco). Podemos ir recorriendo la lista al guardar el nodo actual como un temporal y apuntar a la siguiente, después guardamos este nuevo nodo como el temporal y apuntamos al siguiente, etc. Para regresar, de manera análoga al ejemplo previo, vamos apuntando al nodo anterior y guardando. Con estas funcionalidades, podemos evaluar si un cierto bloque de cache se encuentra en el primer banco con el siguiente código $LRU\_head[index] -> tag == tag$, este ejemplo analiza el primer nodo, para probar con los siguientes bancos es necesario accesar a los siguientes nodos y evaluar $temp -> tag == tag$. Esto se realizo con las funciones que se presentan a continuación.

La función *performAcces()* detecta si la instrucción corresponde a una lectura de datos o instruccion o a una escritura en datos. Revisa si el bloque al que corresponde la dirección está en el cache. Simula el funcionamiento del cache dependiendo de las políticas en curso de escritura y reemplazo bajo una política LRU y actualiza las estadísticas según corresponda.

El flujo de instrucciones para los casos de escritura se muestra en la Figura 3, mientras que el flujo para lo casos de lectura se muestran en la Figura 4.

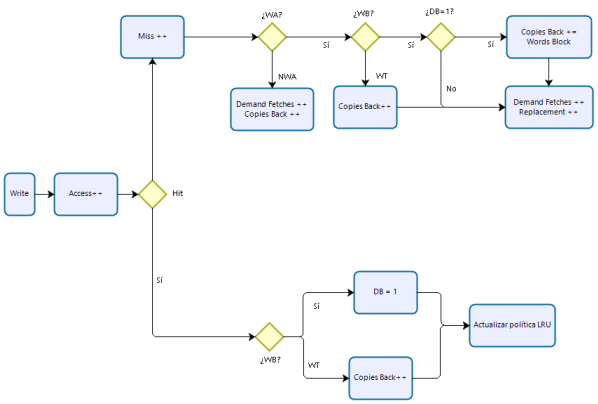

Figure 3: Escritura de datos

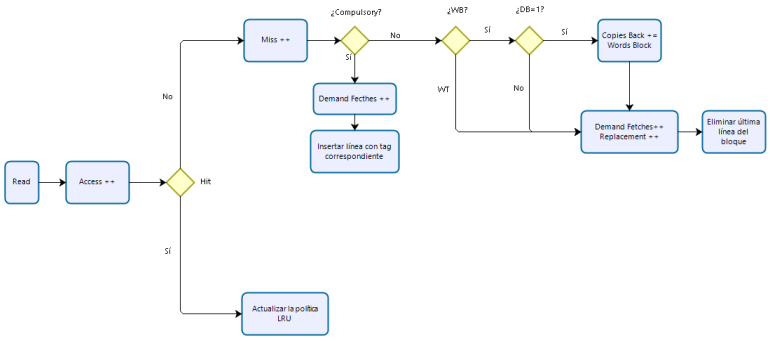

Figure 4: Escritura de datos

Con la función flush(), vaciamos lo que se encuentra tanto en el cache de instrucciones como en el de datos en caso de que ambos se usen o bien el cache con el que se ha trabajado si éste es unificado. Antes de vaciarlo debemos ver si el *dirty bit* está prendido en caso de que estemos bajo una política de tipo *writeback*, si este es el caso actualizamos la estadística *copies back*. Una vez hecho esto al tag se le asigna el valor -1 y al *dirty bit* el valor cero. A continuación pasamos los resultados a un archivo llamado "programa.out" y ponemos las estadísticas en cero.

3 Resultados

A continuación, se muestran los resultados obtenidos al ejecutar los escenarios que la práctica indica.

Primer Escenario - Impacto por tamaño del set

Parámetro	**Valor**
Tamaño de Caché	4 bytes (incremento en potencias de 2)
Tipo de Caché	separado
Tamaño del bloque	4 bytes
Política de escritura	write back - write allocate

Table 1: Características del Caché para el primer escenario

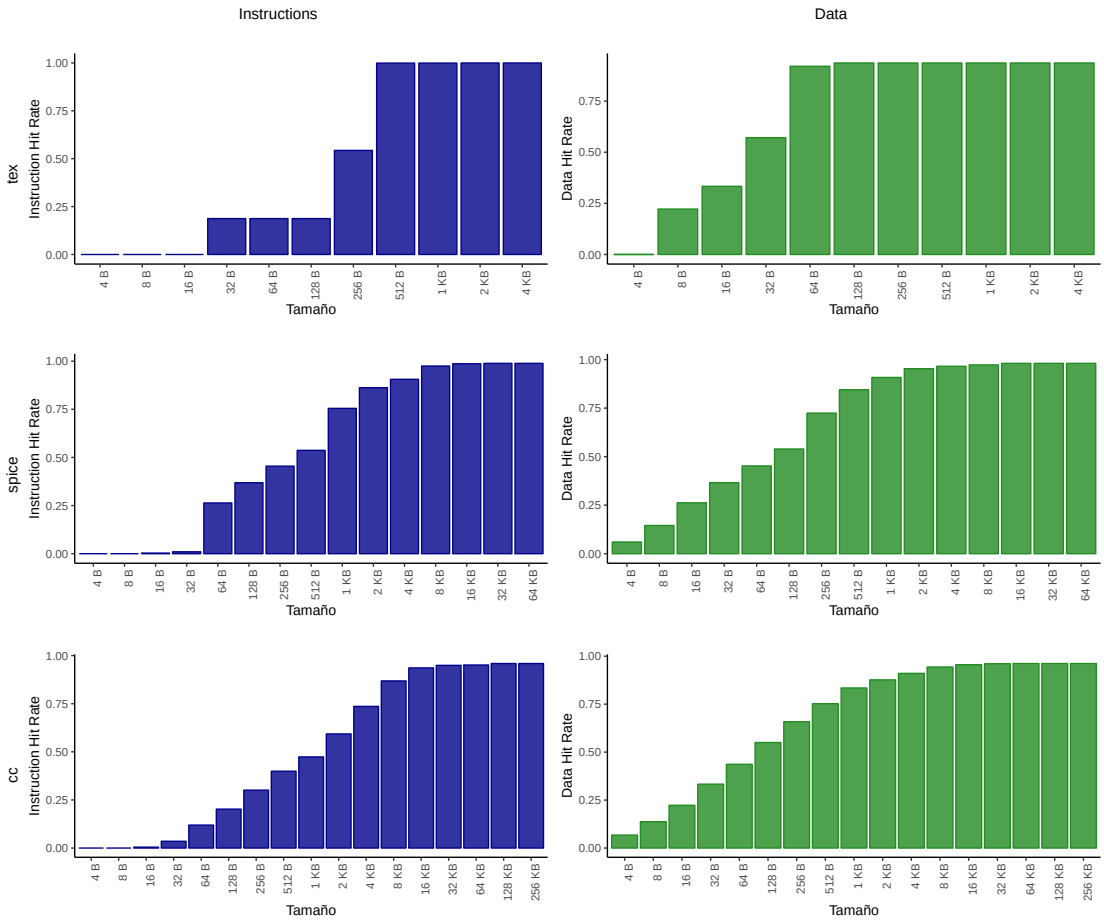

Figure 5:

Preguntas a contestar

- ¿Qué hace el ejercicio? ¿Cómo se relaciona el tamaño de Caché con la tasa de Hit? Nos ayuda a entender cómo el tamaño del Caché, dejando lo demás constante, aumenta la tasa de hit, sin embargo, este aumento es marginalmente decreciente. De esta manera, podemos tomar decisiones de costo/beneficio para entender de qué tamaño implementar nuestro caché. La tasa de Hit aumenta conforme aumenta el tamaño de Caché, no obstante, a partir de cierto punto ya no es relevante.
- ¿Cuál es el tamaño del conjunto de instrucciones y datos de los Caché en cada una de las tres muestras? En la Tabla 2 mostramos el número de instucciones y de datos en cada una de las muestras.

Muestra	Instrucciones	Datos
Tex	597,309	23,5168
Spice	782,764	217,237
CC	757,341	242,661

Table 2: Número de instrucciones y datos en cada muestra

Segundo Escenario - Impacto por tamaño del bloque

Parámetro	Valor
Tamaño de Caché	4 K-bytes
Asociatividad	2-ways
Tipo de Caché	separado
Tamaño del bloque	4 bytes (incremento en potencias de 2)
Política de escritura	write back - write allocate

Table 3: Características del Caché para el segundo escenario

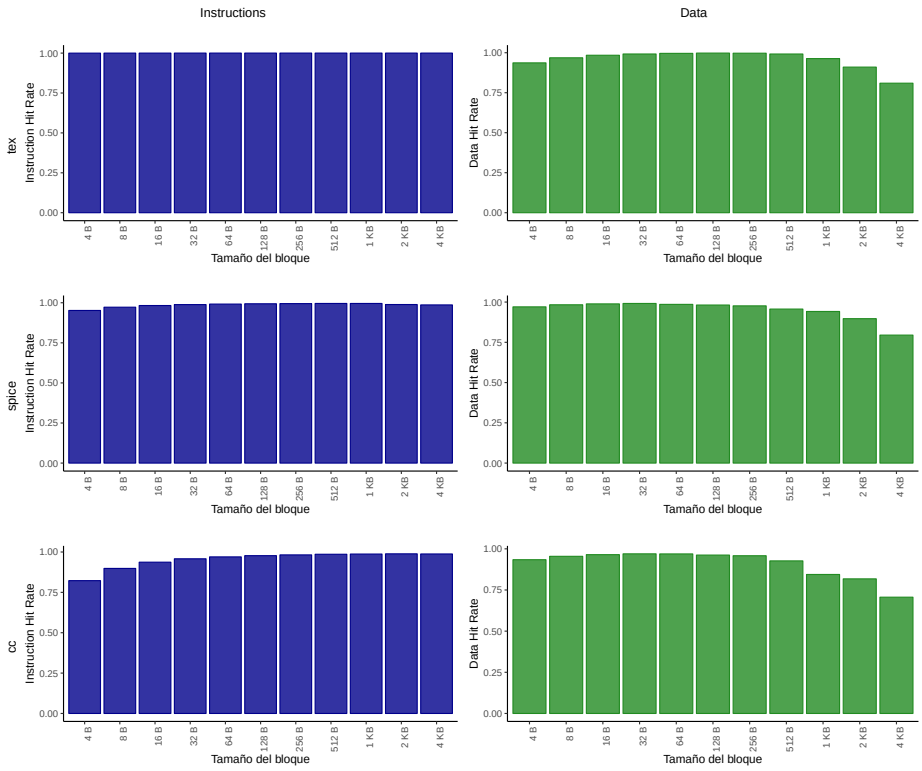

Figure 6: Hit rate del Caché de instrucciones por muestra

Preguntas a contestar

- ¿Por qué se observó la figura mostrada en las gráficas? ¿En qué afecta la localidad espacial? El tamaño del bloque no afectó demasiado, no obstante, es importante mencionar que la tasa de hit comienza a disminuir con bloques muy grandes debido a que no se está utilizando su información. Tenemos instrucciones y datos que no estamos utilizando en Caché.

- ¿Cuál es el tamaño óptimo de bloque por muestra? Para la muestra tex, el óptimo es 64B. Para la muestra spice, el óptimo es 128B. Para la muestra CC, el óptimo es 64B.

- ¿El tamaño óptimo del bloque para los Caché de instrucciones y datos es diferente? ¿Qué podemos inferir respecto a las referencias a instrucciones vs referencias a datos? Para las muestras tex y spice no ha sido diferente. Para la muestra CC, hay una ligera diferencia en una potencia más. No obstante, la tasa de hit es muy cercana a 1 en estos casos óptimos. Así mismo, se puede observar que el tamaño de caché para datos alcanza un

hit rate máximo cuando el tamaño del bloque es de 64B. Sin embargo, el bloque de instrucciones no parece tener un máximo (en estos ejercicios), ya que mientras más aumenta el tamaño del bloque el hit rate sigue aumentando.

Tercer Escenario - Impacto por asociatividad

Parámetro	**Valor**
Tamaño de Caché	4 K-bytes
Asociatividad	1 (incremento en potencias de 2)
Tipo de Caché	separado
Tamaño del bloque	128 bytes
Política de escritura	write back - write allocate

Table 4: Características del Caché para el tercer escenario

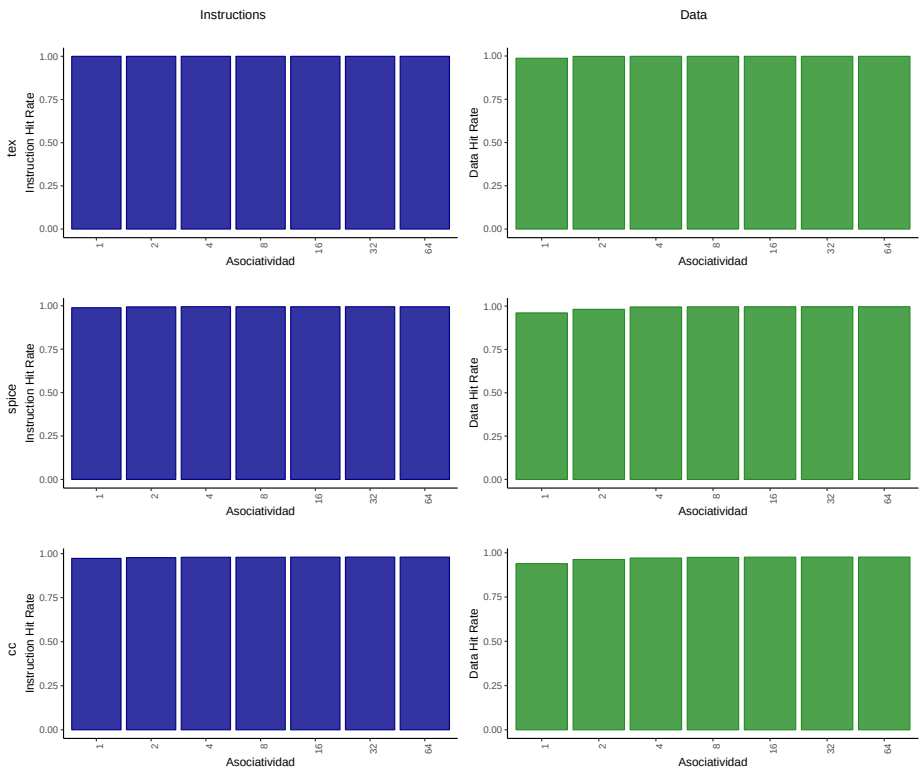

Figure 7: Hit rate del Caché de instrucciones por muestra

Preguntas a contestar

- ¿Por qué se observó la figura mostrada en las gráficas? El tamaño del bloque es el óptimo del ejercicio anterior, la tasa de hit es practicamente 1.
- ¿La gráfica para los Caché de instrucciones y datos es diferente? ¿Qué podemos inferir respecto al impacto que tiene la asociatividad en las referencias de instrucciones vs referencias de datos? No existe diferen i impartante entre las gráficas, el cambio en la asociatividad no afecta la tasa de hit cuando el tamaño del bloque es el óptimo

Cuarto Escenario - Ancho de banda en memoria

Parámetro	**Valor**
Tamaño de Caché	8-16 K-bytes
Asociatividad	2-4
Tipo de Caché	separado
Tamaño del bloque	64-128 bytes
Política de escritura	write through vs write back

Table 5: Características del Caché para el cuarto escenario **modalidad write allocate**

Preguntas a contestar

- ¿Qué caché tiene la menor cantidad de tráfico entre write through y write back? ¿Por qué? Como se puede observar en las tablas 7 y 8, la política write through tiene mayor tráfico debido al número de accesos que debe realizar en memoria principal y Caché.
- ¿Existe un escenario donde la respuesta anterior sea diferente? ¿Cuál? No existe un escenario diferente dentro de los casos evaluados. Sin embargo si el tamaño del cache fuera pequeño o bien si no se cumplen las propiedades de localidad espacial y temporal la tasa de miss sería alta y con la política writeback es necesario copiar todo el bloque a memoria principal. Supongamos el caso extremo en que el caché de datos tiene una sola línea con $n = 4$ palabras, el programa lee un dato, reescribe ese dato, lee otro dato que no está en caché, escribe sobre ese dato, nuevamente lee un dato que no está en caché y así sucesivamente. Con este programa se tiene que copiar todo el bloque a memoria principal cada dos instrucciones. Por lo que cada dos instrucciones se copian cuatro palabras a memoria principal.

Bajo una política de write through se pasa una sola palabra a memoria principal cada dos instrucciones. Por lo que se podría considerar un cambio de política a write through si la tasa de *misses* es alta y si hay una cantidad alta de escrituras.

Parámetro	**Valor**
Tamaño de Caché	8-16 K-bytes
Asociatividad	2-4
Tipo de Caché	separado
Tamaño del bloque	64-128 bytes
Política de escritura	write allocate vs write no allocate

Table 6: Características del Caché para el cuarto escenario **modalidad write back**

- ¿Qué caché tiene la menor cantidad de tráfico entre write allocate y write no allocate? ¿Por qué? La política write allocate debido a que no debe acceder a memoria principal, pues el bloque se escribe solo en Caché. Nuevamente, se apuesta por los principios de localidad, en este caso, el temporal.

- ¿Existe un escenario donde la respuesta anterior sea diferente? ¿Cuál? Dentro de los casos evaluados no hay ningún caso en que haya más tráfico en el write allocate que en el no write allocate. Sin embargo consideremos el caso en que las localidades de memoria principal que se leen no pertenecen al mismo bloque de las localidades de memoria que se reescriben y que las escrituras se encuentran tan dispersas en el programa que cada vez que hay una instrucción de escritura el bloque al que corresponde no se encuentra en caché. Por lo tanto se pasa todo el bloque de memoria principal a caché y éste se desechará antes de volverse a usar. Por lo que si la localidad espacial y temporal no se cumplen para las direcciones que contienen instrucciones de escritura, sería recomendable una política no write allocate.

WP Miss	WP Hit	CS	BS	Assoc	Fetches	Copies Back
wna	wt	8,192	64	2	2,880	104,513
wna	wt	16,384	64	2	2,464	104,513
wna	wt	8,192	128	2	3,488	104,513
wna	wt	16,384	128	2	2,656	104,513
wna	wt	8,192	64	4	3,312	104,513
wna	wt	16,384	64	4	2,464	104,513
wna	wt	8,192	128	4	4,064	104,513
wna	wt	16,384	128	4	2,656	104,513
wa	wt	8,192	64	2	15,696	104,105
wa	wt	16,384	64	2	11,696	104,173
wa	wt	8,192	128	2	18,528	104,280
wa	wt	16,384	128	2	13,696	104,323
wa	wt	8,192	64	4	15,408	104,104
wa	wt	16,384	64	4	10,480	104,167
wa	wt	8,192	128	4	15,684	104,306
wa	wt	16,384	128	4	11,200	104,337

Table 7: Características del Caché para el cuarto escenario **modalidad write through**

WP Miss	WP Hit	CS	BS	Assoc	Fetches	Copies Back
wna	wb	8192	64	2	2880	29,903
wna	wb	16,384	64	2	2,464	29,909
wna	wb	8,192	128	2	3,488	29,935
wna	wb	16,384	128	2	2,656	29,957
wna	wb	8,192	64	4	3,312	29,903
wna	wb	16,384	64	4	2,464	29,909
wna	wb	8,192	128	4	4,064	29,935
wna	wb	16,384	128	4	2,656	29,957
wa	wb	8,192	64	2	15,696	7,664
wa	wb	16,384	64	2	11,696	7,616
wa	wb	8,192	128	2	18,528	8,608
wa	wb	16,384	128	2	13,696	8,256
wa	wb	8,192	64	4	15,408	7,568
wa	wb	16,384	64	4	10,480	7,568
wa	wb	8,192	128	4	15,648	7,648
wa	wb	16,384	128	4	11,200	7,648

Table 8: Características del Caché para el cuarto escenario **modalidad write back**

References

[1] William Stallings. *Computer organization and architecture: principles of structure and function.* Macmillan, 1987.