Arquitectura de Computadoras

Organización de caché y evaluación de desempeño

Mario Becerra
Luis Daniel Hernández

Marzo de 2017

1. Introducción

En esta práctica se implementa un simulador de caché basado en trazas con el objetivo de reforzar la teoría vista en clase sobre cómo funcionan los cachés. Con el simulador implementado, se analizan distintas configuraciones de caché, las cuales también fueron discutidas en clase.

El simulador toma como entrada una serie de lecturas y escrituras a memoria, las cuales fueron proporcionados por el profesor. El simulador es configurable y se le puede modificar las características de tamaño de caché, de tamaño de bloque, si es un caché unificado o separado por instrucciones y datos (I-D), políticas de escritura *write-back*, *write-through*, *write-allocate* o *write-no-allocate*. El simulador rastrea el número de referencias de instrucciones y de datos, número de *misses* de datos y de instrucciones, número de palabras adquiridas de memoria y número de palabras escritas en memoria. En los casos en que la asociatividad es mayor a 1, la política de reemplazo utilizada es *Least-Recently-Used* (LRU).

2. Marco teórico

Se espera que el lector de este trabajo esté familiarizado con la funcionalidad de un caché y sus tecnicismos. Sin embargo, a continuación se mencionan y explican algunos de los conceptos más utilizados en este trabajo.

Un caché unificado es uno en el que se guardan instrucciones y datos, mientras que un caché I-D es uno en el que en realidad se tienen dos cachés: uno para instrucciones y uno para datos; todas las referencias que tienen que ver con datos se hacen con el caché D, mientras que todas las que tienen que ver con instrucciones se hacen con el caché I.

Las políticas de escritura *write-back* y *write-through* se establecen para cuando se tiene un *hit* en una referencia a escritura de un dato. En la política *write-through* se escribe la palabra en caché y en memoria, mientras que en *write-back* se escribe solo en caché, por lo cual se tiene un *dirty bit* que indica si el bloque de memoria es inconsistente con el bloque que está en caché.

Análogamente, las políticas de escritura *write-allocate* y *write-no-allocate* se establecen para cuando se tiene un *miss* en una referencia de escritura. En el caso de *write-allocate*, se trae a caché el bloque en el que se encuentra la localidad de memoria en la cual se quiere escribir para después modificar la memoria únicamente en el caché. Por el contrario, en la política *write-no-allocate*, se escribe únicamente en la memoria principal.

Usualmente, las combinaciones que se utilizan son *write-back* con *write-allocate* y *write-through* con *write-no-allocate*.

La política de reemplazo LRU es de la siguiente forma: cuando se tiene una colisión en un *set*, si todas las páginas de ese *set* están ocupadas, se reemplaza la página que fue usada hace más tiempo. Para poder implementar esta política, se tiene que saber en todo momento el orden en que han sido utilizadas las páginas. En el caso de esta práctica se hace con una lista doblemente ligada en la cual se elimina la cola cuando hay que reemplazar, y cada vez que se accede a una página, se pone en la cabeza de la lista para que así en la cabeza siempre esté la página que se usó más recientemente y en la cola la que menos.

3. Evaluación de desempeño

En esta sección se evalúa el desempeño para cada uno de los archivos de trazas indicados, como función de distintos parámetros, manteniendo todo lo demás constante. Se presenta una gráfica para cada archivo, y en cada gráfica se muestran los resultados para el caché de datos (D) y el de instrucciones (I).

3.1. Tamaño de caché

En esta subsección, se analiza el desempeño como función del tamaño. Se utiliza un caché *full-associative* con un tamaño de bloque de 4 bytes, políticas de escritura *write-back* y *write-allocate*. En la figura 1 se puede ver la tasa de *hit* en el eje y como función del tamaño en el eje x. Este eje está en escala logarítmica con base 2 y va desde 4 hasta 1,073,741,824 bytes. Se puede ver que conforme aumenta el tamaño del caché, aumenta la tasa de *hit*, esto debido a que mientras más grande es el caché, mayor es la probabilidad de que contenga al dato que se necesita; sin embargo, se llega a una asíntota en el 1 en la cual no importa cuánto aumente el tamaño, la tasa de *hit* ya llegó al máximo. El punto en el que la tasa de *hit* llega al máximo varía según el programa y según si se trata de instrucciones o de datos.

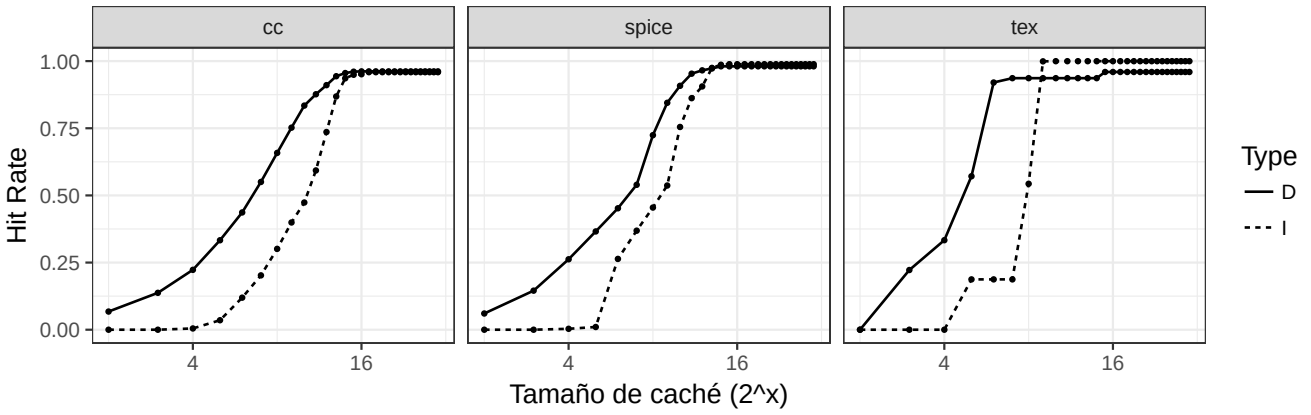

Figura 1: Tamaño de caché

En [1] mencionan que el *working set size* es la cantidad total de memoria utilizada en cierto momento. En este caso, si se quisiera saber el *working set size* de cada programa, se tendría que ver el archivo de trazas y contar el número de localidades de memoria utilizadas. En la tabla 1 se muestran estos números: distinguiendo si el tipo de acceso a memoria es de instrucción (I) o dato (D), y además se muestra el número de localidades únicas en cada caso. Cada localidad de memoria es de 32 bits, es decir, 4 bytes, por lo que si se quisiera saber el tamaño en bytes, se tendría que multiplicar el número de la tabla por 4.

Tabla 1: Número de localidades de memoria utilizadas en cada archivo para accesos a datos (D) o instrucciones (I).

Archivo	Tipo	Localidades únicas	Localidades totales
`cc`	D	11,856	242,661
`cc`	I	31,195	757,341
`spice`	D	6,356	217,237
`spice`	I	8,964	782,764
`tex`	D	38,026	235,168
`tex`	I	160	597,309

3.2. Impacto del Tamaño del Bloque

Ahora se analiza el desempeño dependiendo del tamaño del bloque. En este caso, se utilizó un caché de datos y uno de instrucciones de tamaño 8K-bytes cada uno, asociatividad 2 y políticas de escritura *write-back* y *write-allocate*.

En la figura 2 se muestra la tasa de *hit* como función del tamaño del bloque, el cual va desde 4 bytes hasta 4 K-bytes. Se observa que en un principio, con mayor tamaño de bloque se tiene mayor tasa de *hit*, pero que esta tasa disminuye después de cierto punto. Esta disminución es más evidente en el caso del caché de datos, en el cual la tasa de *hit* disminuye drásticamente a partir de más o menos el tamaño 128 para los tres archivos. La forma de esta curva tiene una razón sencilla, y esta es la del principio de localidad espacial, el cual dice que las instrucciones van a estar en localidades de memoria cercanas entre ellas debido a que se tienen instrucciones secuenciales y pocos saltos, además de variables agrupadas en los mismos bloques de memoria (por ejemplo, los arreglos). Debido a este principio, si se aumenta el tamaño de bloque, se cargan más localidades de memoria que van a ser utilizadas; pero si el tamaño del bloque es muy grande, al tener un tamaño fijo de caché, bloques más grandes implican menos *sets* en un mismo caché, por lo cual se tienen que hacer más reemplazos.

En el caso de este simulador con el tamaño de caché que se utiliza, el tamaño óptimo de bloque para cada archivo sería:

- `cc`: Para el caché de datos un tamaño de 32, y para el de instrucciones de 2048.
- `spice`: Para el caché de datos un tamaño de 32, y para el de instrucciones de 512 o 1024.
- `tex`: Para el caché de datos un tamaño de 128, y para el de instrucciones de 32 a 2048 (todos tienen tasa de *hit* igual a 1).

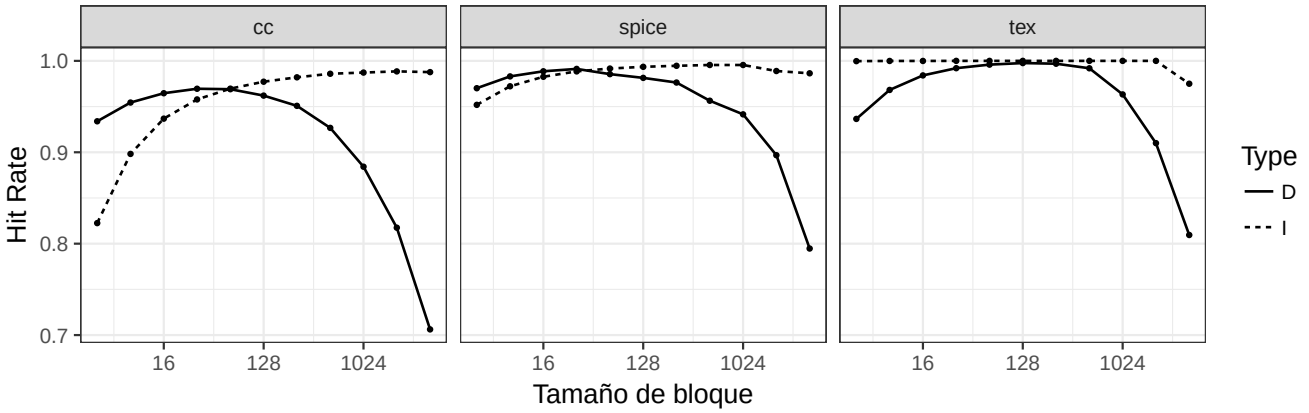

Figura 2: Tamaño de bloque

3.3. Impacto de la Asociatividad

En esta subsección se analiza el impacto que tiene la asociatividad en la tasa de *hit*. Para este análisis se simula un caché I-D de 8K cada uno con tamaño de bloque de 128 con políticas de *write-back* y *write-allocate*. En la figura 3 se muestran las tasas de *hit* como función de la asociatividad. En todos los casos, se puede ver que aumentar la asociatividad aumenta la tasa de *hit*. El tener mayor asociatividad mejora la tasa de *hit* siempre porque se tienen menos colisiones, por lo que hay menos reemplazos y menos *misses*. Esto, naturalmente, viene con un costo, el cual es el de tener que buscar en todos las páginas de una misma línea para encontrar el *tag* correspondiente al bloque de memoria en que se encuentra la palabra a la cual se quiere acceder.

Se puede apreciar que casi siempre la tasa de *hit* es mayor en el caché de instrucciones que en el de datos, a excepción de `spice` a partir de asociatividad 4, sin embargo, estas diferencias se van haciendo más chicas conforme aumenta la asociatividad.

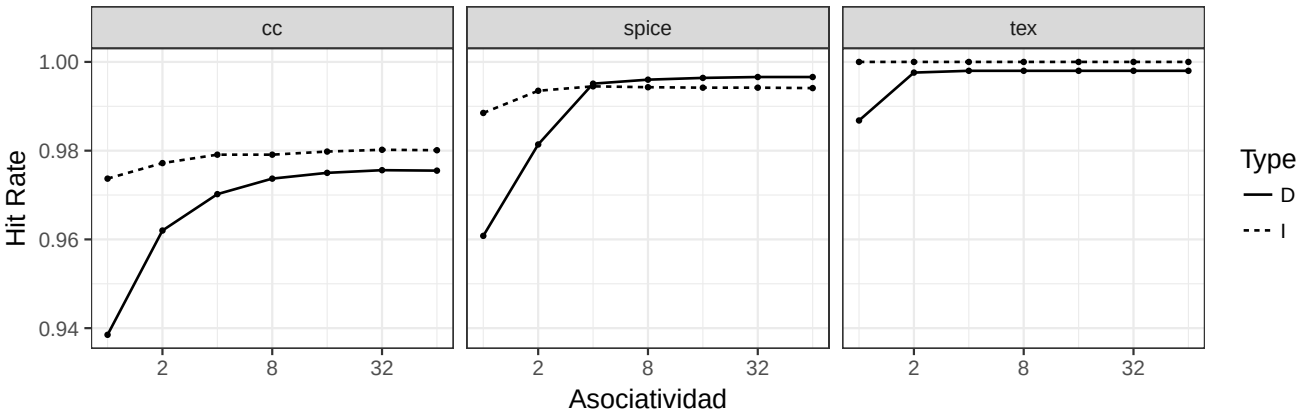

Figura 3: Asociatividad

3.4. Ancho de Banda en Memoria

En las figuras 4, 5 y 6 se muestra la cantidad de *copies back* y *demand fetches* en un caché I-D simulado, para cada uno de los archivos. Cada figura corresponde a un archivo diferente, y cada cuadrante de la figura corresponde a una configuración de tamaño de caché y tamaño de bloque, y a su vez, cada subcuadrante corresponde al número de *copies back* y de *demand fetches* con política de escritura *write-back* (WB) o *write-through* (WT). Dentro de cada subcuadrante hay dos barras, cada una de las cuales corresponde a política *write-allocate* (WA) o *no-write-allocate* (NW).

Se puede ver que para el mismo tamaño de bloque, de caché y la misma asociatividad, la política de *write-back* tiene exactamente el mismo número de *demand fetches* que la política de *write-through*; sin embargo, el número de *copies back* es menor con la política de *write-back*. Estas dos observaciones tienen razones: la razón de la primera es bastante simple, y es que la operación de *demand fetch*, al ser una operación de **lectura**, no es afectada por la política de **escritura**, y por esto los números son iguales; la razón de la segunda observación es que en el caso de un *hit*, con la política *write-through* se tiene que escribir en la memoria principal en cada operación de escritura, mientras que en la política *write-back* se escribe solamente en el caché, por lo que se reduce el número de accesos a memoria para escribir, es decir, los *copies back*. Estos números podrían voltearse, i.e. que el número de *copies back* fuera mayor en *write-back* que en *write-through*, si se ejecutara un programa en el que se la mayoría de las operaciones fueran de escritura en oposición a las de lectura.

Similarmente al caso anterior, para el mismo tamaño de bloque, de caché y la misma asociatividad, las políticas *write-allocate* y *write-no-allocate* tienen exactamente el mismo número de *copies-back* si se sigue la política *write-through*, pero con *write-back*, no se ve un patrón claro del número de *copies back*. En el caso de los *demand fetches*, en todos los casos el valor de la política *write-allocate* es mayor que el de *write-no-allocate*, esto porque en el caso de *miss*, la política *write-allocate* trae todo un bloque de memoria a caché para luego escribir en este, mientras que *write-no-allocate* solo escribe la palabra directamente en memoria; por lo que se tienen muchos más *demand fetches*.

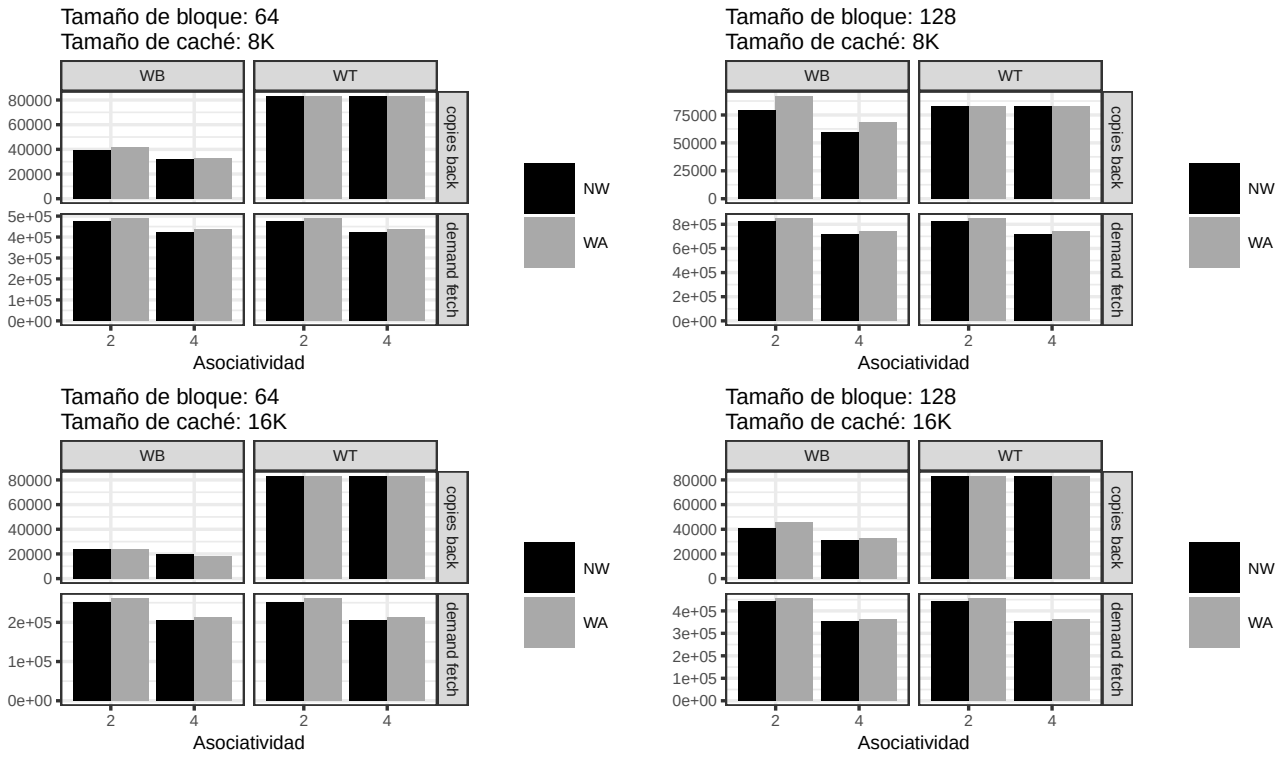

Figura 4: Cantidad de *copies back* y *demand fetches* para archivo `cc`.

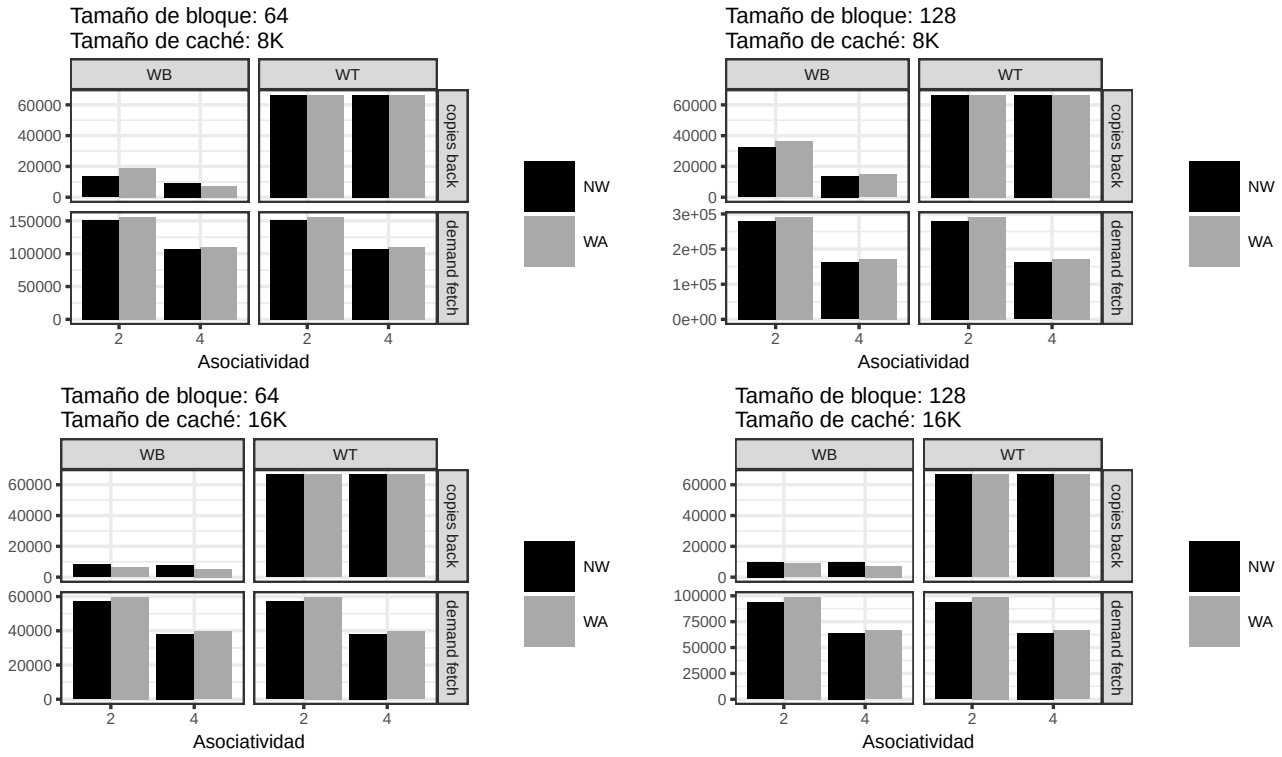

Figura 5: Cantidad de *copies back* y *demand fetches* para archivo `spice`.

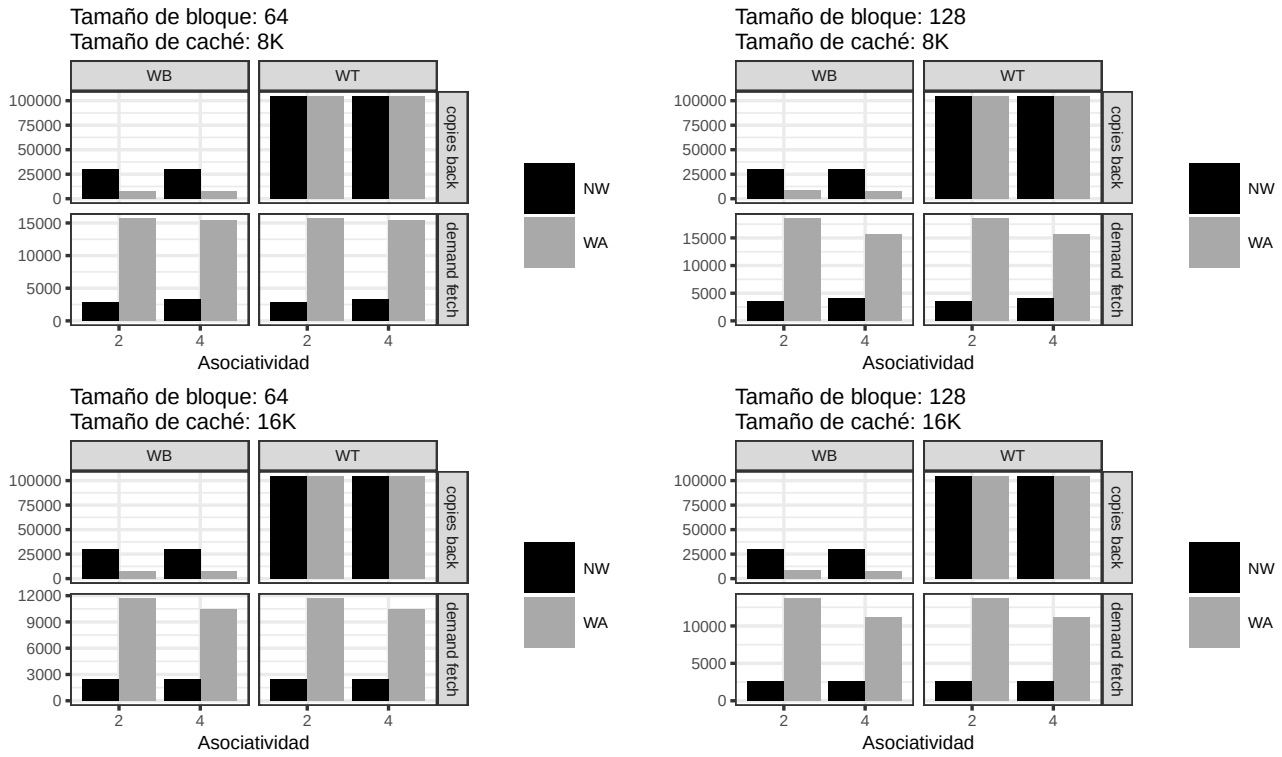

Figura 6: Cantidad de *copies back* y *demand fetches* para archivo `tex`.

4. Conclusiones

Este proyecto fue una buena experiencia para entender cómo funciona la memoria caché y reforzar los conceptos vistos en clase. Sin embargo, se tiene una curva de aprendizaje un tanto profunda con los archivos iniciales y, en particular, con la programación en C. Aún así, la práctica fue una experiencia enriquecedora llena de aprendizaje, con sus momentos de alta frustración.

Referencias

[1] Ulrich Drepper. «What every programmer should know about memory (2007)». En: *URL http://people. redhat. com/drepper/cpumemory. pdf* (2010).

[2] John L. Hennessy y David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123704901.

[3] William Stallings. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005. ISBN: 0131856448.