Evaluación de una caché de metadatos para sistemas de archivos

Rubén H. Ullón
ESPOL- FIEC
Guayaquil-Ecuador
ruben.ullon.ec@ieee.org

Cesar San Lucas
ESPOL- FIEC
Guayaquil-Ecuador
csan@espol.edu.ec

Luis Jácome
ESPOL-FIEC
Guayaquil-Ecuador
lejacome@espol.edu.ec

I. Introducción

Los sistemas de archivos (por ejemplo, ext3) almacenan los metadatos asociados a un archivo de manera independiente de los datos o contenido del archivo. Estos metadatos incluyen el nombre del archivo, su longitud, y su ubicación en la jerarquía de archivos (ej.: /home/user/cabad/varios.txt).

Dichos metadatos son almacenados en disco, en archivos con una estructura propietaria o en una base de datos ligera. Sin embargo, consultar esta base de datos cada vez que un usuario quiere abrir un archivo resulta costoso debido a lo lento que es leer del disco duro.

Por esta razón, los sistemas de archivos modernos, utilizan cachés en memoria que contienen información parcial de los metadatos. Cuando un usuario desea abrir un archivo, primero se consulta los metadatos en la caché, y solamente si no se encuentra la información en la misma (cache miss), entonces se consulta los metadatos en disco.

El sistema de archivos ficticio OperativosFS actualmente almacena todos los metadatos de la estructura jerárquica de directorios en memoria. Por lo que se considera implementar una caché para dicha información, y se evaluara el rendimiento de cuatro políticas de desalojo de caché: ÓPTIMO, LRU, CLOCK y una variante de LRU seleccionada por Ud. (por ejemplo, LRUK, SLRU, etc.

II. Metodología

Algoritmo CLOCK. Para el desarrollo de este algoritmo se hizo uso de las siguientes estructuras:

Se creó una **Lista**, para representar la información que va ser almacenada en la cache, donde el elemento contenido en la lista fue un representado por la siguiente clase:

PageTableEntry. La cual mantuvo la siguiente estructura:

-Bit de referencia

-Bit de validez

Los cuales son necesarios para representar la simulación de dicho algoritmo.

Si bien es cierto la información que iba a ser leída como se describió en la introducción exigía un alto rendimiento en la búsqueda para verificar si la información que se iba procesando ya había sido cargada a la cache.

Para solucionar este problema se planteó hacer uso de una estructura de tipo **HashTable**, la cual sigue el paradigma [clave, valor], recordando que esta estructura almacena la información dependiendo del hashcode asignado a cada llave ingresada a la tabla.

Esta tabla necesitaba tener fijo un tamaño, el cual dependía del tamaño de la cache con que se trabajara, entonces como solución a evitar un alto número de colisiones en las entradas a la tabla hash se usó el siguiente número primo mayor al tamaño de la cache. Por ejemplo: Cache de tamaño 50000 el siguiente primo 65521

Como parte de la descripción que contiene nuestra estructura de HashTable tenemos que:

Key: La llave representa la clase creada StringMH la cual mantiene referencia al string leído y a su hashcode el cual es calculado realizando una multiplicación de los valores numéricos acumulados de cada carácter en la cadena.

Value: Representa la posición en la lista de la cache donde se encuentra el StringMH que ha sido ingresado a la misma.

Algoritmo LRU. Para el desarrollo de este algoritmo se hizo uso de las siguientes estructuras**:**

Esta vez la información almacenada de la cache va ser representada por una **Lista doblemente enlazada,** debido a que recordando cómo trabaja el algoritmo LRU donde en cada reemplazo sacamos al menos recientemente usado que en nuestro caso representa el primer nodo que se encontrara en la lista, entonces una de las problemáticas que ayudo a solucionar esta estructura fue, que al momento de darse un hit facilita la eliminación del nodo para colocarlo al final de la lista.

De igual manera que en el Algoritmo CLOCK se usó la misma estructura **HashTable** con la única diferencia que ahora el valor es una referencia al nodo agregado en la lista doblemente enlazada que representa a la cache.

III. RESULTADOS

Una vez ejecutados los algoritmos estos pasaron por un proceso de prueba con archivos y cache de diferentes tamaños.

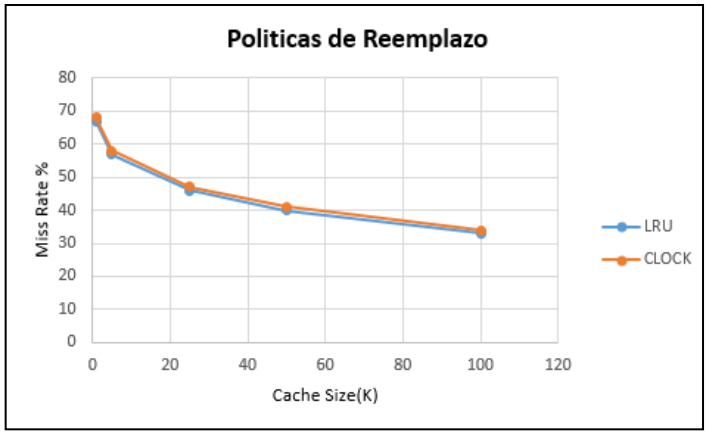

Figura 1. *Porcentaje de Miss Rate para cada política de reemplazo, dependiendo el tamaño de la cache*

Tiempos obtenidos para los diferentes tamaños de cache de cada política de reemplazo:

TIEMPOS		
	LRU	**CLOCK**
1K	1min 23.764seg	1min 17.661seg
5K	1min 24.252seg	1min 17.729seg
25K	1min 21.153seg	1min 12.768seg
50K	1min 22.960Seg	1min 12.982seg
100K	1min 37.143seg	1min 9.921seg

IV. CONCLUSIONES

- Los tiempos obtenidos para ambas políticas fueron bastante similares, siendo la diferencia tan solo en unidades de segundos.
- Los porcentajes de miss rate para cada tamaño de cache de las políticas de reemplazo evaluadas de igual manera que el tiempo fueron muy aproximadas.
- La escala del porcentaje de Miss Rate comparado en cada tamaño utilizado para la cache fue reduciendo en un factor de 10% por cada aumento de cache

REFERENCES

[1] Algoritmos de Reemplazo en cache
Web: https://en.wikipedia.org/wiki/Cache_algorithms

[2] Algoritmos de Reemplazo en cache
Web: https://en.wikipedia.org/wiki/Cache_algorithms

[3] Algoritmos 4th Edition, (2015), Cap. 3.4
Web: http://algs4.cs.princeton.edu/home/