ESCUELA SUPERIOR POLITECNICA DEL LITORAL

FACULTAD DE INGENIERIA EN ELECTRICIDAD Y COMPUTACION

SISTEMAS OPERATIVOS

**Autores:** Gustavo Totoy y Frank Hermida

Descripción breve

En el siguiente documento describe la evaluación de una cache de metadatos para un sistema de archivos, implementado en el lenguaje de programación C++

Documentacion de Sprout

Evaluación de una cache de metadatos para sistema de archivos

Contenido

[**Introducción** 2](#_Toc396960912)

[**Descripción del problema** 2](#_Toc396960913)

[**Metodología** 3](#_Toc396960914)

[**Algoritmo First In, First Out (FIFO)** 3](#_Toc396960915)

[**Algoritmo Least Recently Used (LRU)** 4](#_Toc396960916)

[**Algoritmo Least Frequently Used (LFU)** 5](#_Toc396960917)

[**Algoritmo Random** 6](#_Toc396960918)

[Resultados 0](#_Toc396960919)

[**Conclusiones** 0](#_Toc396960920)

# **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.

# **Descripción del problema**

El sistema de archivos de **Sproud** actualmente almacena todos los metadatos de la estructura jerárquica de directorios en memoria. Sus diseñadores están considerando implementar una caché para dicha información, y desean evaluar el rendimiento de cuatro políticas de desalojo de caché (cache eviction policies): FIFO, LRU, LFU

# **Metodología**

Durante el procesamiento del workload, al ejecutar el algoritmo FIFO y leído un path del mismo. Este automáticamente lo agrega a la memoria cache. Por otra parte esto no sucede en los restantes algoritmos de alojamiento, se debe esperar que se procese todo el workload y una vez realizado esta tarea, el conjunto de path son trasformadas en un archivo de enteros para ser insertados en la cache.

## **Algoritmo First In, First Out (FIFO)**

Para la implementación de la política de desalojo FIFO, dentro del header **fifo\_cache\_t.h** encontramos la función **find(x),** la cual se encuentra implementada la política de reemplazo FIFO.

**Estructuras utilizadas:** vector y Hashmap.

**Descripción:** La estrategia de utilizar dos estructuras en la implementación se debe a que, optimizamos de mejor manera el desalojo de elementos en la estructura vector al mantener un puntero al elemento que se va a desalojar; y el cálculo del siguiente elemento a desalojar es el incremento del puntero. La estructura hashmap se utilizó básicamente para determinar la existencia o no de un miss.

**Diseño propuesto**

**Workload**

**container**

**map**

**Find(x)**

**replacement**\_**it**\_

**Find(x)**

**cache\_miss**

**X**

Variables

**container** = vector

**map** = unordered\_map

**replace\_it\_** = puntero

**Capacity** = capacidad de la cache

**cache\_miss** = variable booleana que verifica si hay un miss al buscar en la cache

## **Algoritmo Least Recently Used (LRU)**

Para la implementación de la política de desalojo LRU, dentro del header **lru\_cache\_t.h** encontramos la función **find(x),** la cual se encuentra implementada la política de reemplazo LRU.

**Estructuras utilizadas:** Hashmap en c++ se lo conoce como unordered\_map.

**Descripción:** Para una mejor optimización del algoritmo se utiliza un hashmap que usa como “key” los elementos del workload, y como “value” una variable de tipo time\_mark que representa el último tiempo de uso de cada elemento de la cache. Esta variable es relevante para la aplicación de la política ya que con ella podemos hacer el reemplazo respectivo buscando aquel elemento de la cache que tiene el mínimo time\_mark(LRU). Inicialmente time\_mark = 0

**Diseño propuesto**

**Workload**

**map**

**X**

**Find(x)**

**replacement**\_**it**\_

**Find(x)**

**cache\_miss**

**time\_mark**

**0 1 2 3 4**

## **Algoritmo Least Frequently Used (LFU)**

Para la implementación de la política de desalojo LFU, dentro del header **lfu\_cache\_t.h** encontramos la función **find(x),** la cual se encuentra implementada la política de reemplazo LFU.

**Estructuras utilizadas:** Hashmap en c++ es se lo conoce como unordered\_map.

**Descripción:** Para una mejor optimización del algoritmo se utiliza un hashmap que usa como “key” los elementos del workload, y como “value” una variable de tipo time\_mark que representa la frecuencia con la cual fue utilizado el último elemento de la cache. Esta variable es relevante para la aplicación de la política ya que con ella podemos hacer el reemplazo respectivo buscando aquel elemento de la cache que tiene es la mínima frecuencia time\_mark(LFU). Inicialmente time\_mark = 0

**Diseño propuesto**

**Workload**

**map**

**X**

**Find(x)**

**replacement**\_**it**\_

**Find(x)**

**cache\_miss**

**F = 0 1 2 3 4**

## **Algoritmo Random**

La razón principal para la selección del algoritmo fue debido a la poca complejidad en su implementación debido a razones de tiempo, además notamos que en tiempos de ejecución y la cantidad de misses se acerca mucho al algoritmo FIFO. Básicamente lo que el algoritmo hace es seleccionar una página de manera aleatoria para ser desalojada de la memoria.

**Estructuras utilizadas:** Vector y Hashmap, este último se lo conoce en c++ como unordered\_map.

**Descripción:** La implementación del algoritmo es similar a la explicada anteriormente en el algoritmo FIFO, se utilizan las dos estructuras para una mejor utilización del algoritmo al momento de realizar la búsqueda. También tenemos una variable que selecciona de manera randomica una página en la memoria cache para ser reemplaza usando una distribución uniforme.

**Diseño propuesto**

**Workload**

**map**

**X**

**Find(x)**

**replacement**\_**it**\_

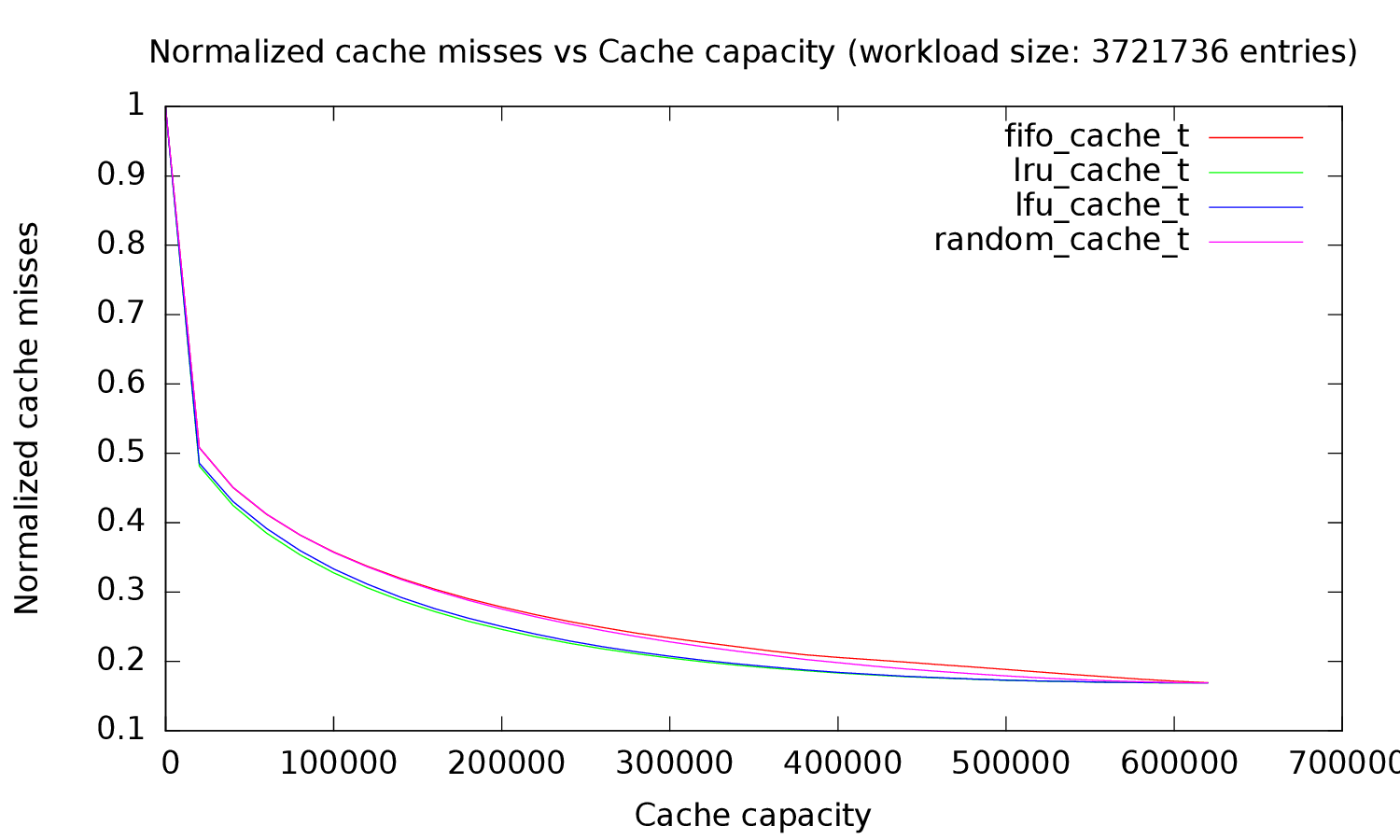
**Find(x)**

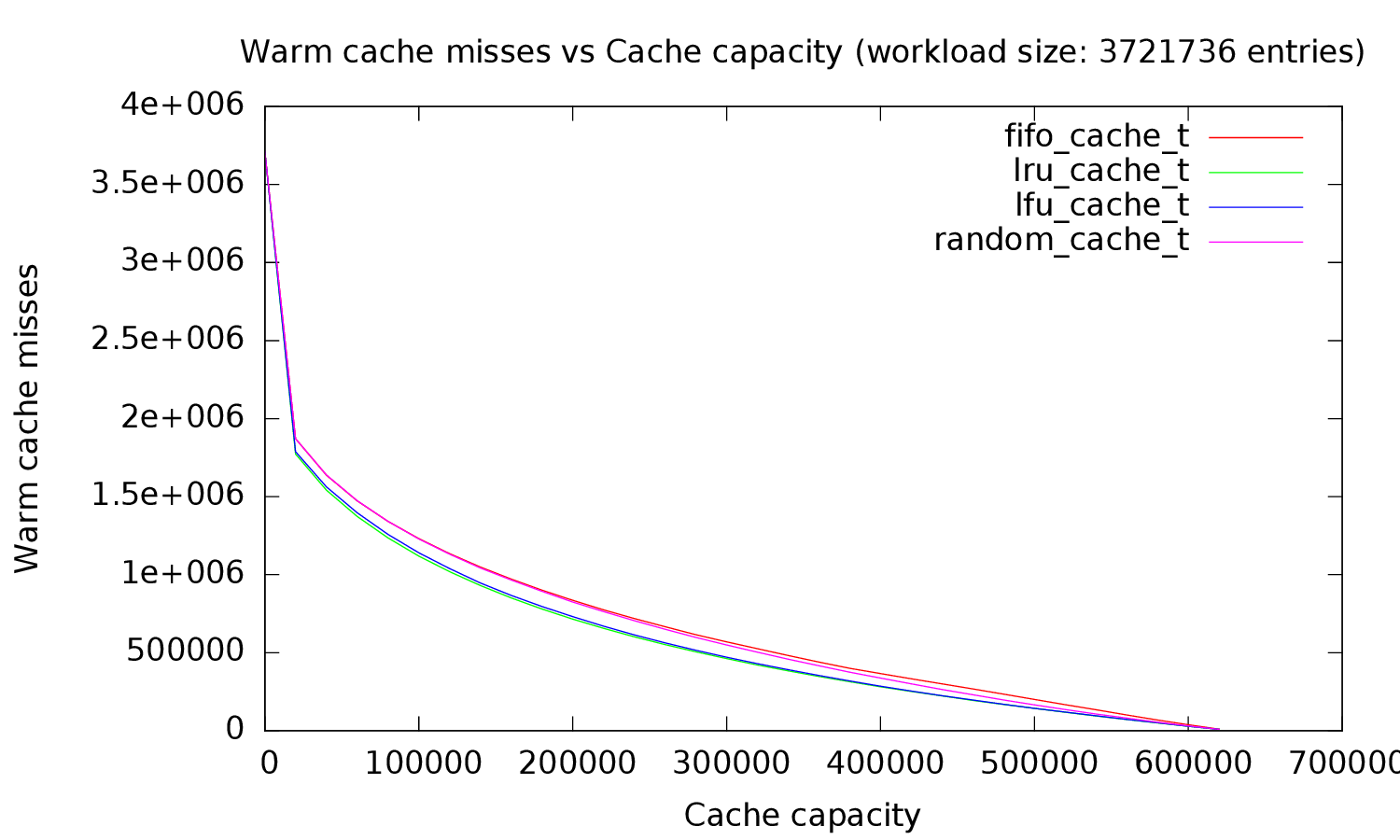
**cache\_miss**

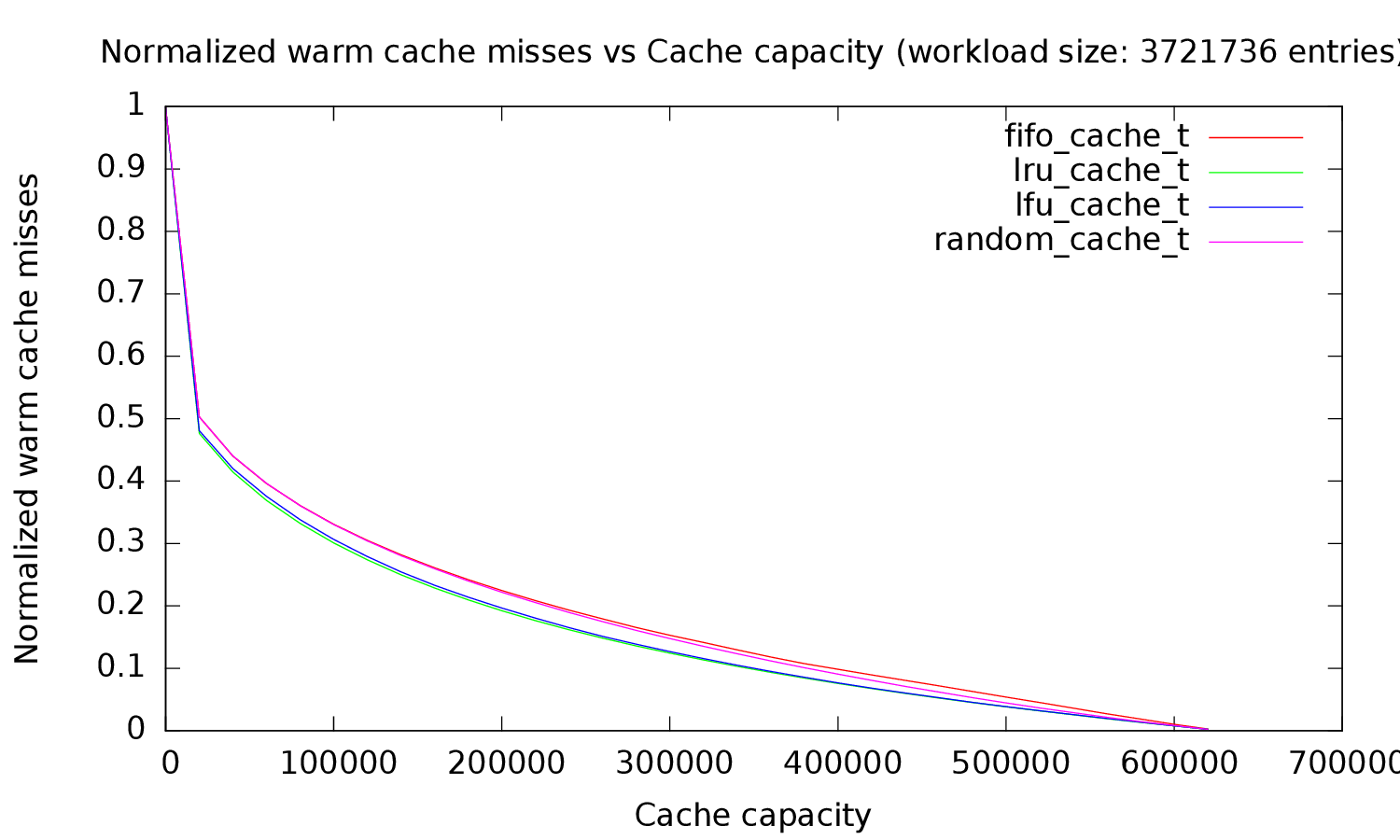
**Dis(random\_generator)**

**container**

# **Resultados**







# **Resultados y Conclusiones**

* En los gráficos se observa que a mayor capacidad de la cache es menor la cantidad de misses. Con el workload utilizado no se presentó la anomalía de Belady
* Con el uso de este workload no hubo una diferencia significativa de misses en las políticas de desalojamiento del LFU Y LRU.
* A medida que la cache va creciendo aproximadamente a la mitad de su capacidad, las políticas de desalojo FIFO y RANDOM presentan una ligera dispersión de los datos.

# **Bibliografía**

|  |  |
| --- | --- |
| [1] | Wikipedia, «Page replacement algorithm,» [En línea]. Available: http://en.wikipedia.org/wiki/Page\_replacement\_algorithm#Random. |
| [2] | Uregina, «Page Replacement Algorithms,» [En línea]. Available: http://www2.cs.uregina.ca/~hamilton/courses/330/notes/memory/page\_replacement.html. |