Simulación de Caché N-Vías Asociativa con Política FIFO

Tulio Cordero y Carlos Zárate

17 de octubre de 2025

1. Introducción y Conceptos Fundamentales

Este documento presenta el diseño y la implementación de un simulador de memoria caché. El objetivo principal es modelar una caché con organización **N-Vías Asociativa** utilizando la política de reemplazo **First-In, First-Out (FIFO)**.

1.1. ¿Qué es una Memoria Caché?

La caché es una memoria pequeña, rápida y costosa que actúa como intermediaria entre la CPU y la Memoria Principal (RAM). Su propósito es reducir la latencia de acceso a los datos, aprovechando los principios de **localidad temporal** y **localidad espacial**.

1.2. Caché N-Vías Asociativa

En este diseño, la caché está organizada en **conjuntos**. Cada conjunto contiene N "vías"(en nuestro caso, $N = 4$).

- Un bloque de Memoria Principal puede ubicarse en cualquier de las N vías de un conjunto específico.
- Esto balancea la velocidad de una caché de mapeo directo con la flexibilidad de una caché totalmente asociativa.

1.3. Política de Reemplazo: FIFO

FIFO (First-In, First-Out) es el algoritmo de reemplazo seleccionado. Cuando un conjunto está lleno y ocurre un fallo de caché, la línea más antigua (la primera que se cargó) es desalojada para dar espacio al nuevo bloque.

→ **Ventaja Clave:** Su **simplicidad** y **bajo costo computacional** en la simulación (**O**(**1**) para reemplazo).

→ **Desventaja y Alternativas:** No considera la frecuencia de uso, a diferencia de **LRU** (Least Recently Used), pero es ideal para un entorno educativo.

2. Arquitectura de la Simulación

2.1. Modelo de Dirección y Desglose de Bits

Utilizamos una dirección simulada de **16 bits**. El desglose de la dirección en sus campos principales es crucial para el mapeo:

$$\text{Dirección} = \underbrace{\text{Tag}}_{\text{Etiqueta}} \Big| \underbrace{\text{Index}}_{\text{Índice del Conjunto}} \Big| \underbrace{\text{Offset}}_{\text{Desplazamiento dentro del Bloque}}$$

- La función `calcularDireccion` realiza el cálculo:
  - **Offset**: Se obtiene mediante la operación de módulo ($\%$) con el tamaño del bloque.
  - **Index**: Se obtiene mediante `shift right` ($\gg$) de la dirección seguido de la operación módulo ($\%$) con el número de conjuntos.
  - **Tag**: El valor restante, utilizado para la comparación.

2.2. Estructuras de Datos en C++

La caché se modela utilizando estructuras de datos de la librería estándar de C++:

1. `struct LineaCache`: Representa un bloque, conteniendo el `tag`, el `dato` y el bit `valida`.
2. `using ConjuntoCache = std::list<LineaCache>;`: La `std::list` es ideal para el conjunto, ya que permite implementar **FIFO** con alta eficiencia.
3. `using Cache = std::unordered_map<int, ConjuntoCache>;`: El `unordered_map` (tabla hash) utiliza el **Index** como clave para un acceso rápido ($\mathbf{O(1)}$) al conjunto.

2.3. Diagrama de la Estructura de Datos

El siguiente diagrama simplifica la arquitectura de la simulación.

3. Funcionalidad Avanzada: Prefetching

La simulación incorpora una técnica de optimización de rendimiento: la carga anticipada o **Prefetching**.

3.1. Mecanismo de Prefetching

Después de cada **fallo de caché** (*Miss*), el simulador llama a la función `cargarPrefetch`.

- ✓ Se predice la **localidad espacial**: Dirección Anticipada = Dirección Actual+Tamaño del Bloque.
- ✓ El bloque anticipado solo se carga si **no está ya en la caché** y si **hay espacio libre** en su conjunto.

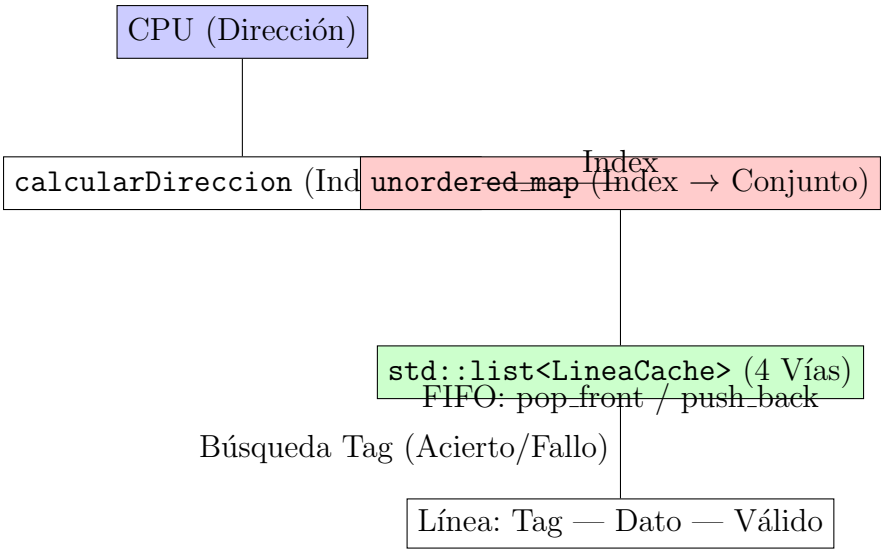

Figura 1: Flujo de acceso y estructuras de datos principales.

4. Análisis de Rendimiento

La métrica principal de la simulación es la **Tasa de Aciertos** (Hit Rate).

$$\text{Tasa de Aciertos} = \frac{\text{Aciertos}}{\text{Accesos Totales}} \times 100\,\%$$

4.1. Simulaciones Prácticas

El menú de operaciones permite probar dos escenarios de carga que explotan la localidad:

1. `simularMatrizMips()`: Acceso anidado a una matriz, probando la interacción entre localidad espacial y temporal.

2. `simularCargaMipsReducida()`: Acceso secuencial a un array, ideal para evaluar la efectividad de FIFO y del Prefetching.

5. Código Fuente del Simulador (Extracto)

5.1. Extracto del Código Fuente (Función accederCache)

La siguiente implementación muestra cómo se aplica la lógica de reemplazo FIFO y la carga del nuevo bloque tras un fallo de caché.

```
    // 2. Fallo de cach
    fallos++;
    cout << " FALLO: Dir " << dirFormato
         << " (Tag: " << tagFormato
         << ", Index: " << indexFormato
         << ", Offset: " << offsetFormato << "). ";

    // Obtener el conjunto
```

```
9       ConjuntoCache& conjunto = miCache[index];
10
11      // 3. Pol tica FIFO (Reemplazo)
12      if (conjunto.size() >= nVias) {
13          LineaCache victima = conjunto.front();
14          conjunto.pop_front();
15
16          // Mostrar el tag de la v ctima en el formato correcto
17          string tagVictimaFormato = obtenerValorFormato(victima.
                tag, numBitsTag);
18
19          cout << "Reemplaza (FIFO) la l nea con Tag: " <<
                tagVictimaFormato << ". ";
20      } else {
21            cout << "Hay espacio. ";
22      }
23
24      // 4. Cargar nuevo bloque
25      // ... (l gica de carga) ...
26      LineaCache nuevaLinea(tag, datoCargado);
27      conjunto.push_back(nuevaLinea); // Insertar al final (el m s
            nuevo)
28      cout << "Carga nueva l nea con Dato: " << datoCargado << "."
            << endl;
```

Listing 1: Implementación de la política FIFO y carga del nuevo bloque.