Universidad de Carabobo

Computación (Arquitectura del Computador)

PROYECTO 10 - Estrategias de Escritura en Caché (Write-Through vs Write-Back)

Informe Técnico

Autor:
José M. Díaz M. (25.682.785)
Sergio José Noguera (30.572.211)

14 de octubre de 2025

1. Problema

Contexto General :

En la arquitectura de computadores modernos, el rendimiento del sistema depende en gran medida del manejo eficiente de la jerarquía de memoria, en particular de la memoria caché, que actúa como intermediaria entre la CPU y la memoria principal (RAM). Dado que el acceso a la RAM es significativamente más lento que el acceso a la caché, el comportamiento de esta última influye directamente en el desempeño general del sistema. Una de las decisiones críticas en el diseño de una caché es la estrategia de escritura que se emplea para manejar las operaciones de escritura (write). Dos técnicas ampliamente utilizadas para este fin son:

- **Write-Through:** Escritura que se realiza tanto en la caché como inmediatamente en la RAM.
- **Write-Back:** Escritura que se realiza solo en la caché, marcando el bloque como "sucio"(dirty), y se escribe a la memoria principal solo cuando ese bloque es reemplazado.

Para elegir qué bloque reemplazar cuando la caché está llena, una política eficiente es la de LRU (Least Recently Used), que reemplaza el bloque que menos recientemente se ha usado.

Cada una de estas estrategias presenta ventajas y desventajas en términos de rendimiento, coherencia y frecuencia de acceso a la memoria principal. En el caso de Write-Through, garantiza la coherencia, pero genera una mayor latencia en el proceso (tiempo de espera por la escritura dual) y puede tener un alto tráfico de datos hacia la RAM. Mientras que Write-Back es más rápida y puede reducir el tráfico hacia la RAM, pero requiere mecanismos adicionales para asegurar la coherencia y manejar los bloques sucios.

Objetivo del Proyecto :

Con el fin de analizar estas estrategias de escritura de forma práctica, se requiere el diseño e implementación de una herramienta de simulación de caché en lenguaje C++11 o superior, que permita emular el funcionamiento de una caché de datos para observar y comparar el comportamiento de ambas estrategias de escritura bajo condiciones controladas, incluyendo una política de reemplazo LRU (Least Recently Used).

En este simulador se compararán ambas estrategias bajo un conjunto predefinido de accesos a memoria (datos sintéticos), es decir, secuencias de direcciones

y valores de escrituras generadas manualmente; también, debe permitir analizar los resultados mediante estadísticas y métricas clave (tiempo de ejecución, cantidad de escrituras a RAM, coherencia de datos, etc.).

La simulación deberá cumplir con lo siguiente:

- Emular el comportamiento de la caché bajo ambas estrategias de escritura usando la política LRU (Least Recently Used) para reemplazos.
- Comparar la eficiencia de ambas estrategias de escritura en cuanto a:
  - La cantidad de escrituras efectivas realizadas en la RAM.
  - El tiempo total de ejecución.
  - La preservación de coherencia de los datos entre la caché y la memoria principal.
- Mostrar en consola el proceso de cargas, escrituras, reemplazos y verificaciones de coherencia.

Problema Específico :

Elaborar un programa en lenguaje C++ (o superior) que simule el funcionamiento de un sistema de memoria, que permita analizar y comparar el comportamiento de ambas estrategias de escritura en caché bajo condiciones controladas y predefinidas. Esta herramienta de simulación debe cumplir con las siguientes características:

1. Caché con mapeo asociativo total de tamaño fijo (4 bloques).
2. Bloques de memoria de tamaño fijo (16 bytes).
3. Memoria principal (RAM) simulada con 256 bloques.
4. Implementación de dos estrategias de escritura: write-through y write-back.
5. Política de reemplazo de bloques usando el LRU (Least Recently Used).
6. Registro de estadísticas, incluyendo:
   - Número total de escrituras a RAM.
   - Número de reemplazos con escritura a RAM en write-back.
7. Medición del tiempo que toma cada estrategia en realizar las operaciones de escritura.
8. Verificación de la coherencia entre la caché y la memoria RAM al final de cada simulación.

Se trabajará con las siguientes secuencias predefinidas:

```
// Direcciones y valores para las escrituras simuladas
std::vector<uint32_t> writes = {0, 16, 32, 0, 48, 64, 16, 80};
std::vector<int> values = {10, 20, 30, 40, 50, 60, 70, 80};
```

Figura 1: Secuencias

Estas secuencias permiten simular:

- Accesos repetidos a las mismas direcciones (Ej: 0, 16), lo cual es útil para observar el efecto del cache hit.
- Reemplazos inevitables debido al tamaño limitado de la caché (4 bloques).
- Situaciones donde los bloques sucios deben ser escritos en RAM (en Write-Back), permitiendo analizar el costo adicional del mantenimiento de coherencia.

Preguntas que orientan el problema :

- ¿Qué diferencia práctica existe en la cantidad de escrituras a RAM entre Write-Through y Write-Back?
- ¿Qué estrategia ofrece mayor eficiencia en términos de tiempo de ejecución?
- ¿Cómo impacta el uso de una política de reemplazo como LRU en la coherencia de datos al emplear Write-Back?
- ¿Cuál de las estrategias resulta más adecuada en escenarios con accesos frecuentes o repetitivos a las mismas direcciones?

2. Técnicas Computacionales para su Solución (algoritmos, estructuras de datos, etc.)

La implementación del simulador de estrategias de escritura en memoria caché (Write-Through y Write-Back), desarrollada en C++11, se apoya en el uso de diversas técnicas computacionales que abarcan desde estructuras de datos eficientes hasta algoritmos de reemplazo y simulación temporal. A continuación se describen los elementos clave utilizados en la solución.

Estructuras de Datos :

- Estructura CacheLine:

  Cada línea de caché (también llamada línea de bloque) está representada por una estructura CacheLine, que encapsula los siguientes atributos necesarios para la simulación:

```
// Estructura que representa una línea de caché
struct CacheLine {
    bool valid = false;      // Indica si la línea tiene datos válidos
    uint32_t tag = 0;        // Etiqueta que identifica el bloque almacenado
    bool dirty = false;      // Marca si la línea ha sido modificada (solo para write-back)
    int data = 0;            // Dato almacenado (aquí un entero para simplificar)
    int last_used = 0;       // Contador para política LRU (cuándo fue usada por última vez)
};
```

  - valid (bool/int): Indica si la línea contiene un bloque con datos válidos.
  - tag (uint32_t): Identificador del bloque de memoria correspondiente.
  - dirty (bool/int): Usado solo en Write-Back; para marcar si el bloque ha sido modificado pero aún no escrita a RAM.
  - data (int): Dato almacenado en la caché (simulado como un entero).
  - last_used (int): Contador (marca temporal) usado para implementar LRU y para determinar cuál línea fue la menos recientemente utilizada.

- Arreglo de Caché y Memoria Principal:

  Se utilizan dos estructuras "std::vector":

```
// Vector que representa la caché con CACHE_SIZE líneas
std::vector<CacheLine> cache(CACHE_SIZE);

// Vector que representa la memoria RAM, inicializada con ceros
std::vector<int> RAM(RAM_SIZE, 0);
```

- RAM: Arreglo de enteros que simula la memoria principal (256 bloques), usada para lecturas/escrituras.
- Cache: Representa la memoria caché con una cantidad fija de líneas (4 en este caso).
- Variables globales para:
  - Contar el número de escrituras a RAM.
  - Contar reemplazos con escritura a RAM (para write-back).
  - Mantener un contador global para el algoritmo LRU.

Funciones :

- Inicialización (init_cache_ram):
  - Inicializar todas las líneas de la caché como inválidas.
  - Inicializar RAM con valores cero.
  - Reiniciar contador global de accesos.

- Búsqueda de Línea en caché (find_line):
  - Buscar si un bloque de memoria ya se encuentra en caché usando el campo tag solicitado y válido.
  - Si el bloque se encuentra en cache, hay que actualizar el contador global de acceso (last_used) para esa línea y se debe retornar el índice de esa misma línea, o retornar -1 si el bloque no existe.
  - Se actualiza el last_used para efectos de LRU.

- Búsqueda de Línea Libre (find_free_line):
  - Recorrer la caché para buscar la 1ra línea inválida (libre o vacía).
  - Si la línea se encuentra, hay que actualizar el last_used y se debe retornar el índice de la línea para su uso inmediato.

- Si no hay línea, hay que retornar -1.

- Reemplazo de Línea con LRU (replace_line_lru):
  - Cuando la caché está llena, se aplica la política Least Recently Used (LRU) para buscar la línea con el menor valor en last_used (la menos recientemente utilizada).
  - Si la línea seleccionada está sucia, es decir, marcada como dirty (dirty == true), en el caso de Write-Back, se fuerza la escritura en RAM antes de reemplazarla.
  - Se invalida la línea reemplazada y resetea sus campos.
  - Se retorna el índice de la línea reemplazada.

- Estrategia de Escritura Write-Through (write_through):
  - Se calcula el tag (etiqueta que identifica el bloque almacenado) a partir de la dirección.
  - Se busca el bloque en caché con la función find_line.
  - Si no está en caché, hay que buscar línea libre con la función find_free_line; si no hay, reemplazar con la función replace_line_lru.
  - Se carga el bloque desde RAM.
  - Se escribe el dato en caché y en RAM inmediatamente.
  - Se incrementa el contador de escrituras a RAM.

- Estrategia de Escritura Write-Back (write_back):
  - Es similar a write-through, con la diferencia de que en la escritura solo se modifica la caché y se marca la línea de bloque como sucia (dirty = true).
  - La escritura para actualizar la RAM ocurre únicamente cuando esa línea se reemplaza.
  - Se reduce el tráfico a la memoria principal, pero se requiere una validación de coherencia.
  - Se utiliza dirty y last_used de forma activa.

- Verificación de Coherencia (verificar_coherencia):

  Esta función analiza si los datos en caché y RAM están correctamente sincronizados.

  - Para cada línea válida:
    - En write-through, los datos de caché y RAM deben coincidir (ser iguales).
    - En write-back, hay que verificar que caché y RAM sean iguales si el bloque no está sucio (de lo contrario hay incoherencia). Si el bloque está sucio, caché y RAM pueden ser diferentes.
  - Reportar si hay incoherencias.

Manejo del Tiempo :

- Se utiliza la librería ¡ctime¿para medir el tiempo de ejecución de cada estrategia.
- Se utiliza clock() para medir el tiempo en milisegundos que toma ejecutar las escrituras.
- Se muestran métricas como:
  - Cantidad de escrituras efectivas a la RAM.
  - Total de reemplazos forzados con escritura en Write-Back.
  - Coherencia de los datos al finalizar la ejecución.

Main (Algoritmo Principal) :

- Definir arreglos con direcciones y valores a escribir.
- Inicializar caché y RAM.
- Ejecutar simulación write-through:
  - Realizar las escrituras.
  - Mostrar estadísticas.
  - Verificar coherencia.
  - Medir tiempo de ejecución.
- Ejecutar simulación write-back:

- Realizar las escrituras.
- Hacer flush final para escribir bloques sucios restantes.
- Mostrar estadísticas.
- Verificar coherencia.
- Medir tiempo de ejecución.

Nota: Un flush (vaciado de búfer) es una operación fundamental en las estrategias de escritura de memoria que consiste en forzar la transferencia inmediata de datos desde un búfer temporal hacia su destino final.

Conclusión de esta sección :

El simulador utiliza un diseño modular y claro, apoyado en estructuras simples pero eficientes (vector, struct, enteros y booleanos) para representar una caché de mapeo asociativo total con política de reemplazo LRU. Las estrategias de escritura están claramente diferenciadas y permiten su análisis práctico a partir de estadísticas precisas y datos sintéticos predefinidos.

3. Análisis y Comparación de Resultados

Contexto del experimento :

El proyecto tuvo como propósito comparar el comportamiento de las estrategias de escritura en caché Write-Through y Write-Back, utilizando un mapeo asociativo total y una política de reemplazo LRU (Least Recently Used).

La simulación se desarrolló en C++, compilada con G++ en entorno GNU/Linux (Ubuntu), siguiendo las reglas de un archivo Makefile para automatizar la compilación y ejecución.

Los datos utilizados fueron datos sintéticos, generados de manera controlada para producir una secuencia específica de direcciones y valores de escritura, lo que permite analizar con precisión el comportamiento de la caché bajo condiciones reproducibles.

Resultados obtenidos :

PARÁMETRO	WRITE-THROUGH	WRITE-BACK
Total escrituras en RAM	8	7
Reemplazos con escritura en RAM	—	3
Coherencia	Garantizada	Garantizada
Tiempo total (ms)	Varia (mayor/menor)	Varia (menor/mayor)

Análisis detallado de los resultados :

- Cantidad de escrituras en RAM:
  - En Write-Through, cada operación de escritura de actualización realizada en la caché se replica inmediatamente en la memoria principal, lo que explica que el número total de escrituras (8) coincida con el número de accesos de escritura realizados.
  - En Write-Back, las escrituras solo se envían a RAM cuando un bloque sucio ("dirty") debe ser reemplazado o al finalizar la simulación. Por ello, el total de escrituras, incluyendo las de reemplazo, es de 7 (que es menor), evidenciando mayor eficiencia en el uso de la RAM.

- Reemplazos de bloques y política LRU:

- La función LRU (replace_line_lru) reemplazó correctamente los bloques menos utilizados recientemente cuando la caché estaba llena, demostrando un manejo coherente de la política de reemplazo.
- En Write-Back, se produjeron 3 reemplazos, lo cual concuerda con la configuración de 4 líneas de caché y la secuencia de accesos. Esto también implicó tener que escribir el bloque sucio en RAM antes de liberar la línea.

- Coherencia de datos:

  En ambas estrategias de escritura, la función de verificación (verificar_coherencia) confirmó que la información entre la caché y la RAM es consistente al finalizar la ejecución, garantizando la integridad y coherencia de los datos. Esto demuestra que:

  - Write-Through logra coherencia inmediata por diseño.
  - Write-Back mantiene coherencia diferida pero controlada, gracias al uso de la marca dirty y al proceso de escritura final de bloques sucios.

- Tiempos total de ejecución:

  En algunas de las ejecuciones, Write-Through fue ligeramente más rápido (Tiempo de Write-Through < Tiempo de Write-Back), mientras que en otras Write-Back fue más eficiente (Tiempo de Write-Through > Tiempo de Write-Back). Estas diferencias de tiempo son esperables y se deben a factores que son externos a la lógica del código, como los siguientes:

  - El uso del objeto de flujo de salida estándar (std::cout) para mostrar datos de las operaciones de salida en la consola o en la pantalla; es decir, muchos prints pueden afectar la medición.
  - Variaciones mínimas en el rendimiento del CPU y en el entorno del sistema operativo sistema operativo durante cada ejecución.

  En general:

  - Esto no afecta la corrección funcional de la simulación.
  - En simulaciones pequeñas, los tiempos son dominados por la sobrecarga del sistema y no por el costo real de memoria.

- En simulaciones grandes (cientos o miles de accesos), siempre y cuando se efectúen en un entorno mas controlado y sin tantas salidas por pantalla, Write-Back tiende a ser consistentemente más eficiente en tiempo, ya que reduce el número de escrituras en memoria principal (RAM).

- Respuestas de las preguntas:

  - ¿Qué diferencia práctica existe en la cantidad de escrituras a RAM entre Write-Through y Write-Back?

    En los resultados obtenidos, la simulación del modo Write-Through realizó 8 escrituras en la memoria principal (RAM), mientras que el modo Write-Back efectuó 7 escrituras efectivas, acompañadas de 3 reemplazos de bloques sucios.

    Esto ocurre porque en Write-Through cada operación de escritura actualiza inmediatamente tanto la caché como la memoria principal, generando un mayor número de accesos a RAM. En cambio, Write-Back acumula los cambios en la caché y solo escribe en RAM cuando un bloque modificado ("dirty") debe reemplazarse o al finalizar la simulación.

    En términos prácticos, Write-Back representa una mayor eficiencia en el uso del ancho de banda de memoria y una reducción en el tráfico hacia la RAM.

  - ¿Qué estrategia ofrece mejor latencia y mejor coherencia?

    - Write-Back ofrece mejor latencia.

      En Write-Back, las escrituras se realizan solo en la caché, y la actualización a la memoria principal (RAM) se difiere hasta que el bloque modificado ("sucio") deba ser reemplazado. Esto significa que la mayoría de las operaciones de escritura no requieren acceso inmediato a la RAM, lo que reduce los tiempos de espera y mejora la latencia promedio del sistema.

      En cambio, en Write-Through, cada vez que se escribe un dato, la operación se envía simultáneamente a la RAM, generando más tráfico y mayor tiempo por acceso, ya que la RAM es más lenta que la caché.

- Write-Through ofrece mejor coherencia.

  Write-Through actualiza la RAM al mismo tiempo que la caché. Por tanto, los datos en ambas memorias siempre están sincronizados, eliminando el riesgo de que la caché contenga información diferente a la memoria principal.

  En cambio, Write-Back retrasar la actualización hasta que se reemplace un bloque sucio, lo cual puede provocar incoherencias temporales entre caché y RAM (aunque el sistema lo controla mediante la marca dirty).

- ¿Cómo impacta el uso de una política de reemplazo como LRU en la coherencia de datos al emplear Write-Back?

  La política LRU (Least Recently Used) garantiza que, al reemplazar bloques en la caché, siempre se expulse el bloque menos utilizado recientemente.

  En el contexto de Write-Back, LRU ayuda a mantener la coherencia porque antes de reemplazar un bloque sucio, el simulador fuerza su escritura en RAM, asegurando que los datos modificados no se pierdan.

  Así, LRU colabora directamente con la coherencia, asegurando que la memoria principal siempre contenga la versión más actualizada antes de eliminar el bloque de la caché.

- ¿Cuál de las estrategias resulta más adecuada en escenarios con accesos frecuentes o repetitivos a las mismas direcciones?

  En escenarios donde las mismas direcciones de memoria son accedidas repetidamente (alta localidad temporal), la estrategia Write-Back resulta más adecuada.

  Esto se debe a que las escrituras repetidas afectan solo la copia del dato en caché, y la memoria principal se actualiza una sola vez cuando el bloque es reemplazado, reduciendo drásticamente el tráfico de escritura.

  En cambio, Write-Through escribiría en memoria principal cada vez, incluso si se trata del mismo dato, generando más latencia y menor eficiencia.

- Conclusiones Finales:

  En conjunto, este proyecto permitió entender de manera práctica cómo las estrategias de escritura en memoria caché impactan el equilibrio entre rendimiento y coherencia de datos.

  La experiencia mostró que Write-Back es más eficiente, mientras que Write-Through es más seguro, y que la elección entre ambos depende del tipo de sistema y las prioridades del diseño.

  Además, la implementación en C++ fortaleció las habilidades de programación estructurada y el razonamiento lógico, consolidando los conocimientos sobre arquitectura de computadores, algoritmos y gestión eficiente de memoria.