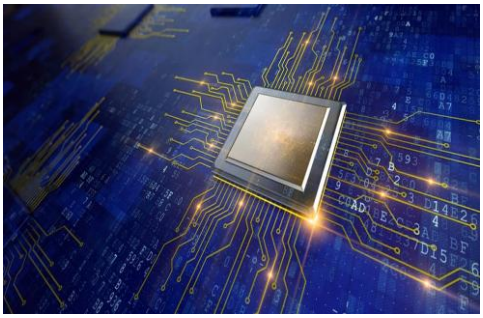
IMPLEMENTACION DE UN CACHE MULTINIVEL(L1, L2, L3)

Bryan Mirabal 31.200.232

Juan Gomez 30.958.654

¿QUE ES UNA CACHE MULTINIVEL?

Una caché multinivel es una arquitectura de almacenamiento en la que se utilizan múltiples niveles de caché para mejorar el rendimiento de acceso a datos en sistemas computacionales. Cada nivel de caché tiene diferentes características en términos de velocidad, tamaño y costo.



1. CACHÉ L1:

- **Ubicación**: Integrada en el procesador.
- **Velocidad**: Muy rápida (nanosegundos).
- **Tamaño**: Pequeño (generalmente de 16 KB a 64 KB).
- **Propósito**: Almacenar datos e instrucciones de uso frecuente.

2. CACHÉ L2:

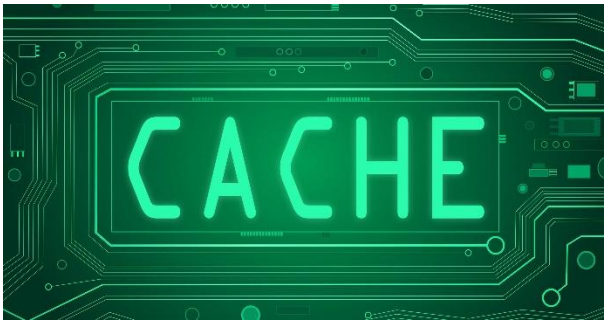
- **Ubicación**: Puede estar en el procesador o en un chip separado.
- **Velocidad**: Rápida (microsegundos).
- **Tamaño**: Más grande que L1 (de 256 KB a varios MB).
- **Propósito**: Actúa como un intermediario entre L1 y la memoria principal.

3. CACHÉ L3:

- **Ubicación**: Generalmente compartida entre núcleos de procesador.
- **Velocidad**: Más lenta que L2, pero aún rápida.
- **Tamaño**: Más grande (varios MB).
- **Propósito**: Mejorar el acceso a datos menos frecuentes que no caben en L1 o L2.

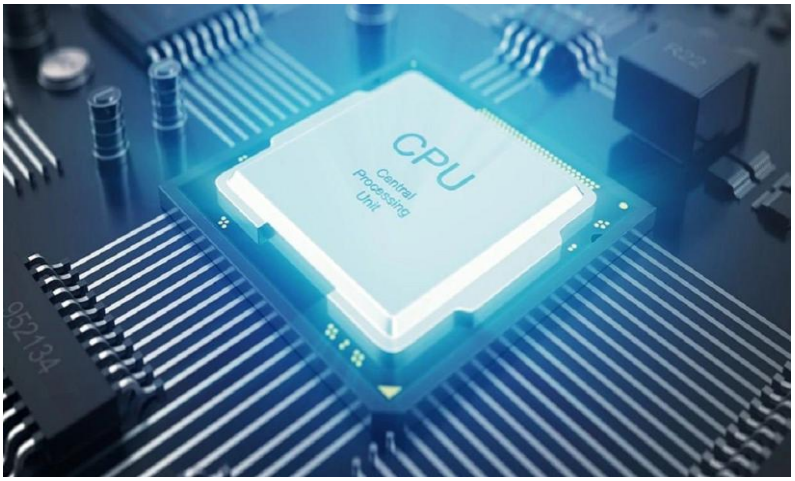
VENTAJA

La caché multinivel ofrece varias ventajas significativas, como la mejora del rendimiento al reducir el tiempo de acceso a datos, lo que acelera el procesamiento y minimiza el tráfico entre el procesador y la memoria principal. Además, permite una reducción de la latencia al almacenar datos cercanos al procesador, aumentando así la eficiencia general del sistema. Sin embargo, es importante considerar aspectos como la coherencia de caché en sistemas multinúcleo, ya que mantener la consistencia entre las diferentes cachés es crucial. También se deben implementar políticas de reemplazo efectivas para gestionar qué datos se mantienen en caché, lo que añade complejidad y costo al diseño del sistema.



¿ QUE HACE NUESTRO ALGORITMO DE CACHE INCLUSIVA ?

Este algoritmo implementa un simulador de caché en C++, diseñado para gestionar el acceso a datos en una jerarquía de memoria que incluye tres niveles de caché (L1, L2 y L3) y una memoria principal. Su objetivo principal es optimizar el acceso a la memoria mediante el uso de una política de reemplazo LRU (Least Recently Used), que permite mantener en la caché los datos más utilizados, mejorando así la eficiencia y la velocidad de acceso a la información.



ESTE ALGORITMO SE COMPONE DE DOS CASES, LAS CUALES SON:

La clase Cache es responsable de gestionar el almacenamiento de datos a través de una estructura que simula el comportamiento de una caché en un sistema de memoria. Esta clase incluye atributos como size, que define el tamaño total de la caché en bytes, associativity, que determina cuántas líneas pueden almacenar un bloque de datos, y lineSize, que especifica el tamaño de cada línea de caché. Utiliza un mapa (data) para almacenar las direcciones de memoria en forma de tags junto con sus direcciones completas, y una lista (lruList) que implementa la política de reemplazo LRU (Least Recently Used) para mantener el orden de uso de los tags. El método access(int address) gestiona el acceso a las direcciones de memoria, identificando si hay un acierto o un fallo en la caché, y actualiza la lista LRU en consecuencia. Además, proporciona métodos para verificar si la caché está llena y para imprimir su estado actual, mostrando qué líneas están ocupadas y cuáles están vacías. En conjunto, la clase Cache juega un papel fundamental en la optimización del acceso a datos, asegurando que los elementos más utilizados se mantengan disponibles para un acceso rápido.

La clase Memory representa la memoria principal del sistema, simulando su funcionamiento mediante un vector que almacena datos en un tamaño predefinido, inicializado con ceros. Su atributo principal, data, se utiliza para almacenar los valores de las direcciones de memoria, permitiendo la lectura de estos valores a través del método read(int address), que devuelve el dato correspondiente a una dirección específica. Esta clase actúa como el último recurso en el acceso a datos, proporcionando información cuando los datos solicitados no se encuentran en ninguna de las cachés (L1, L2 o L3). La implementación de la clase Memory es esencial para completar la jerarquía de memoria del sistema, asegurando que, aunque los accesos a la memoria principal son más lentos, se pueda acceder a todos los datos necesarios cuando sea requerido.

LLENADO DE LA CACHÉ

A medida que el usuario ingresa direcciones de memoria, las cachés se llenan según la política LRU. Cuando una caché alcanza su capacidad máxima, el algoritmo elimina el bloque menos recientemente usado, asegurando que siempre se mantengan los datos más relevantes. Esto permite que el sistema sea eficiente en el manejo de datos, minimizando los accesos a la memoria principal, que son más lentos.

Al ingresar direcciones de memoria en el algoritmo de simulación de caché, el usuario interactúa directamente con el sistema a través de un bucle que solicita continuamente una dirección. Este proceso es fundamental, ya que permite simular el acceso a datos en tiempo real. Cuando el usuario introduce una dirección, el algoritmo intenta acceder primero a la caché L1. Si el dato no se encuentra allí (lo que se considera un fallo), se procede a las cachés L2 y L3 en secuencia. Este enfoque jerárquico refleja cómo funcionan realmente los sistemas de memoria en computadoras, donde se prioriza el acceso a las cachés más cercanas al procesador para optimizar el rendimiento.

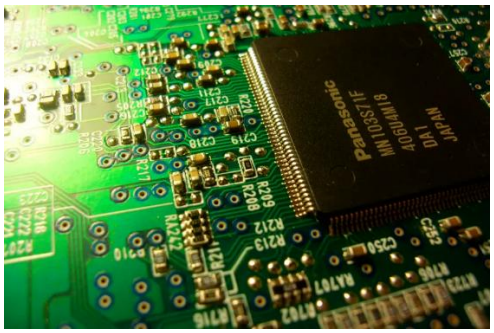
Además, el ingreso de direcciones permite observar cómo se llenan las cachés y cómo se manejan los fallos y aciertos. Cada acceso a una dirección puede resultar en un acierto, que promueve el dato a niveles superiores de caché, o en un fallo, que requiere acceder a la memoria principal. Este proceso no solo ilustra el funcionamiento de la caché, sino que también permite al usuario entender la eficiencia del sistema y cómo se optimizan los accesos a la memoria. Al finalizar la ejecución, el estado de las cachés se imprime, proporcionando una visión clara de cómo las direcciones ingresadas afectaron la gestión de la memoria y el rendimiento general del sistema

ESTADO FINAL DE LAS CACHÉS

Al finalizar el programa, se imprime el estado de cada caché, mostrando cuántas líneas están ocupadas y cuáles están vacías. Esto proporciona una visión clara de cómo se ha comportado el sistema durante la ejecución y cuán efectiva ha sido la gestión de la memoria.

AHORA QUE HACE NUESTRO ALGORITMO DE CACHE EXCLUSIVA

El algoritmo presentado es una simulación de un sistema de caché multinivel que involucra tres niveles de caché (L1, L2 y L3) y una memoria principal. Su objetivo es gestionar el almacenamiento y la recuperación de datos de manera eficiente, minimizando el tiempo de acceso a la memoria. Este tipo de arquitectura es fundamental en los sistemas computacionales modernos, donde la velocidad de procesamiento es crucial.

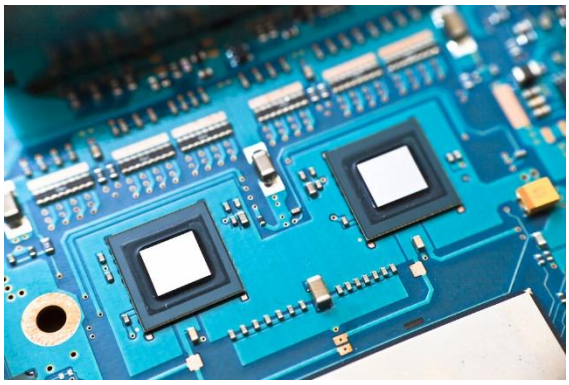


FUNCIONAMIENTO

El algoritmo se basa en una serie de clases que representan la caché y la memoria. La clase Cache maneja las operaciones básicas de la caché, como la inserción, eliminación y verificación de la presencia de datos, mientras que la clase Memory simula una memoria principal.

ESTRUCTURA CLASE CACHE

La clase **Cache** está diseñada para gestionar el almacenamiento y la recuperación de datos en un sistema de caché. Sus atributos principales incluyen size, que define el tamaño total de la caché en bytes, y lineSize, que especifica el tamaño de cada línea de caché. Utiliza un std::map<int, int> llamado data, que almacena pares de tag y dirección completa, donde el tag se obtiene dividiendo la dirección por el tamaño de línea. La clase incluye varios métodos: contains(int address), que verifica si una dirección está presente en la caché; insert(int address), que inserta una dirección y devuelve la dirección reemplazada si es necesario; remove(int address), que elimina una dirección específica; y printCacheState(), que imprime el estado actual de la caché, mostrando cuáles líneas están ocupadas y cuáles están vacías. Esta estructura permite una gestión eficiente de los datos y un acceso rápido a la información almacenada en la caché.



. PROCESO DE ACCESO A DATOS

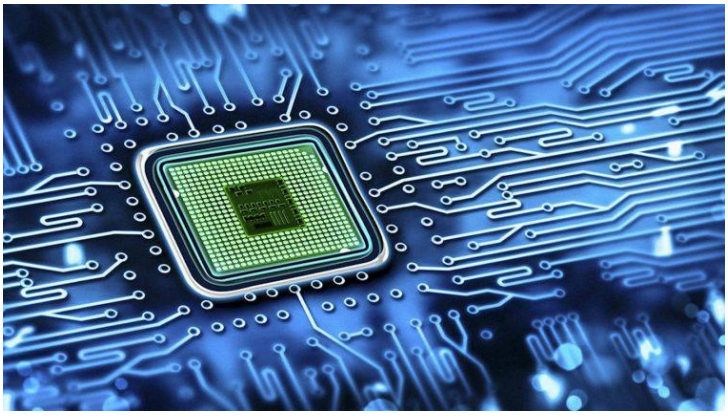
El algoritmo comienza solicitando al usuario que ingrese una dirección de memoria. Este proceso se repite hasta que el usuario decida salir (ingresando -1). Para cada dirección ingresada, el algoritmo sigue estos pasos:

1. **Verificación en L1**: Se verifica si la dirección está presente en la caché L1. Si se encuentra, se registra un "acierto" y se continúa.

2. **Fallo en L1**: Si no se encuentra en L1, se verifica en L2. Si está presente, se registra un "acierto" en L2 y se mueve el bloque de L2 a L1, eliminando el bloque de L2 y posiblemente de L3.

3. **Fallo en L2**: Si no está en L2, se verifica en L3. Un acierto en L3 resulta en mover el bloque a L1, eliminando el bloque de L3.

4. **Fallo en L3**: Si la dirección no está en ninguna de las cachés, se realiza un acceso a la memoria principal, se registra un "fallo" y se inserta el bloque en L1. Si L1 está llena, el bloque reemplazado se mueve a L2, y si L2 también está llena, se mueve a L3.



El algoritmo utiliza una estrategia FIFO (First In, First Out) para el reemplazo de bloques en la caché. Esto significa que el bloque más antiguo en la caché es el primero en ser reemplazado cuando se necesita espacio para un nuevo bloque.

El programa permite al usuario ingresar direcciones de memoria de forma interactiva, lo que facilita la simulación del acceso a la caché. Esto también permite observar el comportamiento de la caché en tiempo real y cómo se llenan las diferentes líneas de la caché a medida que se accede a diferentes direcciones.

ESTADO FINAL DE LA CACHÉ: Al final de la ejecución, el estado de cada nivel de caché se imprime, proporcionando una visión clara de qué datos están almacenados y cómo se han organizado

DIFERENCIAS DE ESTOS DOS ALGORITMOS

En cuestión de funcionamiento general, ambos algoritmos están diseñados para simular el funcionamiento de un sistema de caché, que actúa como un intermediario entre la CPU y la memoria principal, optimizando el acceso a los datos. Sin embargo, implementan diferentes estrategias de gestión de caché y acceso a memoria.

POR PARTE DE LA CACHE INCLUSIVA

En este algoritmo, se utiliza un enfoque de caché simple y directo, donde se implementa una política de reemplazo basada en el concepto de **Least Recently Used (LRU)**. La clase Cache mantiene un mapa que asocia etiquetas (tags) con direcciones de memoria. Cuando se accede a una dirección, se verifica si el tag correspondiente ya está presente en la caché. Si es así, se considera un acierto y se actualiza la lista LRU; si no, se produce un fallo y se intenta cargar el dato desde la siguiente caché (L2 o L3) o desde la memoria principal. Este algoritmo es efectivo para situaciones simples, pero sufre de limitaciones en términos de eficiencia debido a la falta de asociatividad.

POR PARTE DE LA CACHE EXCLUSIVA

Por otro lado, este algoritmo adopta un enfoque más sofisticado, utilizando una política de reemplazo FIFO (First In, First Out) y una estructura de gestión más clara. La clase Cache permite verificar si un bloque está presente, insertar nuevos bloques y eliminar bloques existentes. El método moveBlock facilita la transferencia de datos entre diferentes niveles de caché, lo que permite que los datos se promuevan de niveles inferiores a superiores cuando se producen aciertos. Este enfoque mejora la eficiencia al permitir una mejor gestión de los datos en múltiples niveles de caché, lo que reduce la latencia en el acceso a datos.

DIFERENCIAS

1. POLÍTICA DE REEMPLAZO:

- **Cache inclusiva**: Utiliza una política LRU, que puede ser más eficiente en ciertos patrones de acceso, pero también puede ser más costosa en términos de complejidad al mantener el orden de los elementos.
- **Cache exclusiva**: Implementa una política FIFO, que es más sencilla de gestionar y puede ser suficiente para ciertos escenarios, aunque puede no ser tan eficiente en comparación con LRU en todos los casos.

2. Gestión de datos:

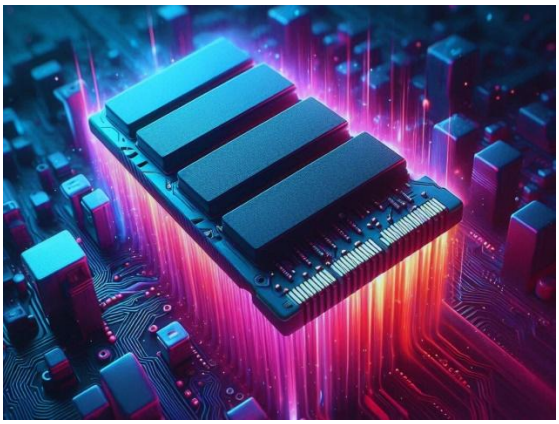
- **Cache inclusiva**: Se enfoca en un acceso secuencial y directo a las cachés, sin un manejo explícito de la promoción de datos entre niveles de caché.
- **Cache exclusiva**: Permite una gestión activa de los datos mediante el método moveBlock, que asegura que los datos se muevan adecuadamente entre las cachés, mejorando la eficiencia del sistema.

3. Interacción con la memoria principal:

- **Cache inclusiva**: Carga datos desde la memoria principal solo si hay un fallo en todas las cachés, lo que puede resultar en un mayor tiempo de espera si se producen múltiples fallos.
- **Cache exclusiva**: Maneja la carga de datos de manera más eficiente, insertando directamente en la caché L1 y promoviendo los datos a niveles superiores según sea necesario, lo que puede reducir el tiempo de acceso en situaciones de fallo.

4. Manejo de errores y excepciones

- **Cache inclusiva**: La gestión de errores es mínima. Si ocurre un fallo en el acceso a la memoria, el algoritmo simplemente intenta acceder a la siguiente caché o a la memoria principal, sin proporcionar retroalimentación detallada o manejo de excepciones.
- **Cache exclusiva**: Incluye un manejo más robusto de errores, permitiendo una mejor detección de fallos y proporcionando información más detallada sobre el estado de la caché y los datos que se están moviendo entre niveles. Esto puede ser útil para depurar y optimizar el rendimiento.



CONCLUSIÓN

La comparación entre los algoritmos de caché inclusiva y cache exclusiva revela no solo diferencias en su implementación técnica, sino también variaciones significativas en su enfoque hacia la gestión de datos y la optimización del rendimiento. Ambos algoritmos abordan el desafío fundamental de la latencia en el acceso a la memoria, pero lo hacen a través de paradigmas distintos que reflejan sus respectivas filosofías de diseño.