UNIVERSIDADE ESTADUAL DE MARINGÁ - UEM

Nome: Gustavo Amaro Niehues RA: 138894

Nome: Bruno Viana Felix da Silva RA:139175

**TRABALHO: SIMULADOR DE MEMÓRIA CACHE**

MARINGÁ - PR

2025

Nome: Gustavo Amaro Niehues RA: 138894

Nome: Bruno Viana Felix da Silva RA:139175

**TRABALHO: SIMULADOR DE MEMÓRIA CACHE**

Trabalho do 2º Semestre do curso de Informática na Universidade Estadual de Maringá, referente a disciplina de Arquitetura de Computadores, ministrada pelo Professor: Maurilio Martins Campano Junior

MARINGÁ – PR

2024

**INTRODUÇÃO E OBJETIVOS DO TRABALHO**

A evolução da computação trouxe consigo a necessidade de dispositivos de armazenamento cada vez mais rápidos e eficientes. Dentro desse contexto, as memórias cache desempenham um papel essencial na melhoria do desempenho dos sistemas, funcionando como um intermediário entre a CPU e a memória principal. Este trabalho tem como **objetivo** a implementação de um simulador de memória cache, abordando aspectos teóricos e práticos relacionados à organização e arquitetura de computadores. Durante a simulação, serão explorados conceitos fundamentais, como funções de mapeamento associativo, algoritmos de substituição de cache: FIFO, LRU, e LFU, políticas de escrita write-back, com o objetivo de proporcionar uma compreensão mais profunda do funcionamento interno desses sistemas. E por fim documentar as decisões de projeto, justificando a escolha de estruturas de dados, tamanho da memória cache e estratégias de implementação, para facilitar o entendimento e a replicação do trabalho.

**TRABALHO: SIMULADOR DE MEMÓRIA CACHE**

Identificação:

Nome: Gustavo Amaro Niehues RA: 138894

Nome: Bruno Viana Felix da Silva RA:139175

1. **Funcionamento da Memória Cache**

A memória cache é uma camada intermediária entre a CPU e a memória principal, projetada para otimizar o desempenho ao armazenar temporariamente os dados mais frequentemente acessados. Quando a CPU solicita um dado, a cache verifica sua disponibilidade, resultando em um "cache hit" (dado encontrado na cache) ou um "cache miss" (dado ausente, exigindo busca na memória principal). Esse processo utiliza técnicas de mapeamento associativo para organizar os dados e garantir acessos rápidos.

Quando a cache atinge sua capacidade máxima, torna-se necessário substituir blocos para liberar espaço, o que é feito por meio de algoritmos como FIFO (First-In-First-Out), LRU (Least Recently Used) e LFU (Least Frequently Used). Esses algoritmos, amplamente usados na gestão de memória, determinam qual bloco será descartado, priorizando eficiência e desempenho.

Além disso, a política de escrita write-back é frequentemente adotada, permitindo que alterações sejam realizadas apenas na cache e escritas na memória principal somente quando o bloco modificado é substituído. Isso reduz a frequência de acessos à memória principal, otimizando o uso do sistema. Esses mecanismos ilustram como a memória cache integra eficiência e estratégia para melhorar a performance computacional, aplicando conceitos fundamentais de organização e arquitetura de computadores. E é exatamente isso que queremos retratar neste trabalho.

1. **Funções de Mapeamento**

def Mapeamento(bloco:Bloco):

    global tamanho\_cache

    indice = random.randint(0,3)

    while cache[indice].LFU != 0:

        indice = random.randint(0,3)

    inserir = Cache(indice,bloco,1,tamanho\_cache,1)

    cache[indice] = inserir

A função **Mapeamento** realiza a inserção de um bloco de dados da memória principal em uma posição disponível na memória cache. Primeiramente, ela gera um índice aleatório entre 0 e 3 para selecionar uma posição no cache. Em seguida, verifica se o valor de **LFU** (Least Frequently Used) do item armazenado na posição escolhida é igual a 0. Nesse caso, significa que o bloco está vazio, ou seja, a posição na cache não está ocupada. Caso o valor de **LFU** seja diferente de 0, isso indica que a posição está sendo ocupada por outro bloco de dados, e a função gera um novo índice aleatório, repetindo o processo até encontrar uma posição disponível.

Quando uma posição válida é encontrada, a função cria um novo objeto **Cache** utilizando o índice da posição encontrada, o bloco de dados a ser inserido, e atualiza os valores de **LFU** e **LRU** para 1, indicando que o bloco foi utilizado recentemente. Além disso, o valor de **first** é definido como o valor atual de **tamanho\_cache**, refletindo a ordem de inserção do bloco no cache. Por fim, o novo objeto **Cache** substitui o item existente na posição do cache, garantindo que o bloco seja corretamente mapeado e inserido no índice disponível.

1. **Algoritimos de Substituição**

def sub(bloco: Bloco, cache: list[Cache]) -> Bloco:

    global política

    def lru(cache: list[Cache]) -> Cache:

        menor = cache[0]

        for item in cache:

            if item.LRU < menor.LRU:

                menor = deepcopy(item)

            elif item.LRU == menor.LRU and item.first < menor.first:

                menor = deepcopy(item)

        for item in cache:

            if item.first > menor.first:

                item.first -= 1

        return menor

    def lfu(cache: list[Cache]) -> Cache:

        menor = cache[0]

        for item in cache:

            if item.LFU < menor.LFU:

                menor = deepcopy(item)

            elif item.LFU == menor.LFU and item.first < menor.first:

                menor = deepcopy(item)

        for item in cache:

            if item.first > menor.first:

                item.first -= 1

        return menor

    def fifo(cache: list[Cache]) -> Cache:

        for item in cache:

            if item.first != 1:

                item.first -= 1

            else:

                x = deepcopy(item)

        return x

    if política == 1:

        bloco\_cache = fifo(cache)

    elif política == 2:

        bloco\_cache = lfu(cache)

    else:

        bloco\_cache = lru(cache)

    bloco\_retirado = bloco\_cache.bloco

    indice = bloco\_cache.index

    cache[indice] = Cache(indice, bloco, 1, 4, bloco\_cache.LRU)

    return bloco\_retirado

A função sub tem como objetivo substituir um bloco de dados no cache de acordo com uma política de substituição, que pode ser FIFO, LFU ou LRU, dependendo do valor da variável global política. Para isso, ela define três funções internas: lru, lfu e fifo, cada uma responsável por implementar uma das políticas de substituição.

A função lru (Least Recently Used) escolhe o bloco que foi acessado há mais tempo. Ela começa assumindo que o item na posição 0 é o de menor valor de LRU. Em seguida, percorre todos os itens no cache, comparando o valor de LRU de cada um. Caso dois itens tenham o mesmo valor de LRU, o critério adicional de first é usado para decidir qual bloco será substituído. Após a escolha do bloco a ser substituído, a função decrementa o valor de first dos itens mais recentes. A função retorna o bloco a ser substituído.

A função lfu (Least Frequently Used) realiza uma operação similar, mas usando o valor de LFU. Ela escolhe o bloco com o menor valor de LFU, e, em caso de empate, o critério adicional de first é utilizado. Assim como a função lru, ela também ajusta o valor de first de todos os itens mais recentes e retorna o bloco a ser retirado.

Já a função fifo (First In, First Out) segue uma lógica mais simples: ela escolhe o primeiro item que foi inserido no cache, identificando-o pelo valor de first. Quando o item com first == 1 é encontrado, ele é retirado do cache e a função retorna esse bloco.

Com base no valor da variável global política, a função principal escolhe qual política de substituição aplicar. Se política == '1', a função fifo é chamada; se política == '2', a função lfu é chamada; e, caso contrário, a função lru é utilizada.

Após selecionar o bloco a ser substituído de acordo com a política escolhida, a função obtém o índice do item retirado e substitui esse item no cache pelo novo bloco. O valor de LRU do item retirado é preservado, se necessário, e o bloco retirado é armazenado em uma variável para ser retornado ao final da execução.

Por fim, a função retorna o bloco que foi retirado do cache, ou seja, o bloco que foi substituído. Assim, a função sub gerencia a substituição de blocos no cache, utilizando uma das três políticas de substituição definidas e retornando o bloco retirado após a substituição.

1. **Política de escrita write-back**

A política de escrita write-back é uma abordagem fundamental no funcionamento de memórias cache e desempenha um papel crucial na eficiência do sistema. Essa técnica visa reduzir a quantidade de acessos à memória principal, uma vez que tais operações são consideravelmente mais lentas em comparação com acessos à memória cache. O conceito principal do write-back é simples: quando um dado na cache é alterado, essa modificação não é imediatamente refletida na memória principal. Em vez disso, a atualização é adiada até que o bloco correspondente seja removido da cache por um algoritmo de substituição, como LRU (Least Recently Used), LFU (Least Frequently Used) ou FIFO (First In, First Out).

No programa apresentado, a função Write\_Back implementa precisamente essa política. Quando um bloco precisa ser substituído na cache devido à falta de espaço ou à necessidade de acessar um novo dado, a função é chamada para escrever de volta na memória principal qualquer modificação feita naquele bloco específico enquanto estava na cache. Esse mecanismo garante que a memória principal mantenha os dados atualizados apenas quando estritamente necessário, evitando operações de escrita excessivas e desnecessárias. Por exemplo, se um dado "232" foi alterado na cache e, posteriormente, o bloco que contém esse dado foi escolhido para ser substituído, somente então a função Write\_Back atualiza o valor na memória principal.

A vantagem da política write-back é clara no contexto de desempenho, especialmente em sistemas onde os acessos à memória cache são significativamente mais rápidos que os acessos à memória principal. Como a maioria das operações ocorre dentro da cache, a necessidade de acessar a memória principal é reduzida, minimizando atrasos e otimizando a utilização de recursos do sistema. No entanto, essa abordagem também possui seus desafios. Em cenários de falha de energia ou interrupção abrupta, existe o risco de perder dados que foram alterados na cache, mas que ainda não foram gravados na memória principal. Para mitigar esses riscos, sistemas modernos frequentemente utilizam buffers ou mecanismos de backup para proteger os dados antes que sejam permanentemente perdidos.

No contexto do trabalho, a função Write\_Back exemplifica a integração prática dessa política em um simulador de memórias. Ao associar essa técnica com algoritmos de substituição (LRU, LFU, FIFO), o programa é capaz de decidir estrategicamente qual bloco remover da cache, enquanto assegura que a memória principal seja atualizada apenas no momento adequado. Essa abordagem proporciona um equilíbrio entre desempenho e consistência, sendo amplamente empregada em sistemas computacionais modernos.

1. **Decisões de projeto para a implementação**

A implementação da simulação de memória cache foi guiada por decisões de projeto cuidadosamente elaboradas para equilibrar eficiência, clareza de código e funcionalidade. Uma das primeiras decisões foi a utilização de quatro espaços de memória cache, cada um com capacidade para armazenar quatro dados. Essa escolha permitiu uma simulação compacta e funcional, suficiente para demonstrar as interações entre memória principal e cache, sem comprometer a escalabilidade do programa. Combinado a isso, a memória principal foi projetada para conter 300 blocos, cada um composto por quatro dados aleatórios, totalizando 1200 posições de memória. Essa arquitetura proporcionou um espaço adequado para a simulação de acessos e substituições, mantendo o programa eficiente.

Outro aspecto fundamental foi a adoção de dataclasses para a definição das estruturas de dados. Foram implementadas as classes Bloco e Cache para modelar os blocos da memória principal e as entradas da memória cache, respectivamente. A classe Bloco encapsula os dados de cada posição da memória principal, enquanto a classe Cache armazena, além do bloco alocado, três valores adicionais: LFU (Least Frequently Used), LRU (Least Recently Used) e FIRST, que são utilizados pelos algoritmos de substituição. Essa decisão foi tomada para permitir que a função de substituição (def sub) operasse com flexibilidade, dependendo da política de substituição ativa, sem necessidade de redefinir a estrutura de dados para cada algoritmo.

Uma mudança significativa no projeto foi a substituição da matriz utilizada anteriormente para rastrear o LRU por um contador associado a cada linha da cache. Essa abordagem simplifica o rastreamento de quais linhas foram usadas mais recentemente, eliminando a necessidade de manipular estruturas complexas. Diferentemente do comportamento do LFU, onde o contador é reiniciado a cada acesso, no LRU o contador continua incrementando, refletindo de maneira mais precisa o histórico de uso de cada linha. Essa modificação não só otimizou o código, mas também melhorou a clareza do comportamento do algoritmo.

Por fim, a escolha de implementar apenas uma função de substituição, a def sub, foi outra decisão central. Essa função encapsula a lógica para todas as políticas de substituição (FIFO, LFU e LRU), utilizando condições internas para determinar qual algoritmo aplicar. Essa abordagem reduz a duplicação de código, facilitando manutenção e extensões futuras. Assim, o projeto não só atendeu aos requisitos da simulação, mas também resultou em uma implementação modular, clara e eficiente.

O tamanho da cache e a política de substituição são variáveis essenciais para a implementação da simulação e foram amplamente utilizadas em todo o código. O tamanho da cache define o número máximo de linhas disponíveis para armazenar dados da memória principal. Essa variável foi projetada para permitir flexibilidade, ajustando-se conforme as necessidades da simulação, começando com capacidade para quatro linhas, o que reflete uma configuração compacta e prática para testes. Já a política de substituição, que pode ser selecionada entre FIFO (First In, First Out), LFU (Least Frequently Used) ou LRU (Least Recently Used), determina a estratégia utilizada para decidir qual linha da cache será descartada quando a cache estiver cheia. Essa variável é passada como parâmetro para várias funções, garantindo que o comportamento da cache seja consistente com a política selecionada pelo usuário. Essa abordagem modular permitiu uma implementação clara e reutilizável, onde o tamanho da cache e a política atuam como elementos centrais da lógica de funcionamento.

**CONCLUSÃO**

A implementação do simulador de memória cache proporcionou uma oportunidade prática de explorar os conceitos fundamentais da arquitetura de computadores, como funções de mapeamento, algoritmos de substituição e políticas de escrita. O projeto evidenciou a importância de decisões estruturais claras, como o uso de classes bem definidas e a escolha por uma memória cache compacta e funcional, que permitiram uma simulação eficiente e didática.

O uso das variáveis centrais — tamanho da cache e política de substituição — destacou a flexibilidade do sistema, permitindo ajustes dinâmicos e personalizáveis para diferentes cenários. Além disso, a política de escrita write-back e a unificação das lógicas de substituição em uma única função (sub) demonstraram como soluções otimizadas podem equilibrar eficiência e simplicidade, resultando em um sistema modular e de fácil manutenção.

Por fim, a simulação alcançou seu objetivo ao reproduzir, de forma prática e detalhada, os processos internos de uma memória cache, permitindo não apenas uma melhor compreensão acadêmica do tema, mas também fornecendo uma base sólida para futuras expansões e aplicações mais complexas.

**REFERÊNCIAS**

STALLINGS, William. *Arquitetura e organização de computadores*. 8. ed. São Paulo: Pearson, [ano de publicação].

TANENBAUM, Andrew S.; BOS, Herbert. *Modern operating systems*. 4. ed. Boston: Pearson, 2015**.**