UNIVERSIDADE ESTADUAL DE MARINGÁ

DEPARTAMENTO DE INFORMÁTICA

CIÊNCIA DA COMPUTAÇÃO

Relatório do Trabalho Prático

ARQ. E ORG. DE COMPUTADORES II (12029)

Professora: Sandra Cossul

Alunos:

Isadora Dantas Bruchmam - RA. 140870

Letícia Akemi Nakahati Vieira - RA. 140535

Pedro Henrique Pereira da Silva - RA. 139781

Sumário

1. Introdução

De modo geral, os processadores modernos são muito mais rápidos que a memória RAM, considerando limitações tecnológicas, custo e a diferença de arquitetura. Enquanto processadores evoluíram rapidamente em velocidade de processamento e complexidade, a RAM segue um desenvolvimento com maior foco em capacidade e menor custo. Essa discrepância de velocidade faz com que o processador passe por longos períodos ociosos, esperando os dados serem transferidos pela RAM. Isso cria um gargalo no sistema, desperdiçando a capacidade do processador e diminuindo sua performance.

A solução dessa problemática é a utilização da memória cache, que é menor e está integrada diretamente ao mesmo chip do processador. Desse modo, um dos maiores fatores de alta latência (a distância física) é reduzido drasticamente, considerando que a cache é bem mais próxima em relação à RAM. Além disso, a memória cache utiliza um tipo diferente de tecnologia, que é muito mais rápida e menos densa que a RAM, apesar de ser mais cara.

A memória cache reduz a latência ao atuar como um buffer entre a CPU e a RAM, armazenando informações acessadas recentemente. Assim, caso o conteúdo já esteja disponível nela, o acesso torna-se muito mais rápido do que uma nova busca na memória principal. Isso aumenta o desempenho, pois, ao diminuir o tempo de espera, permite que o processador execute mais instruções por unidade de tempo, resultando em uma performance geral do sistema superior. Além disso, esse recurso mantém a unidade de processamento constantemente atendida, assegurando sua operação em capacidade máxima e evitando a inatividade enquanto aguarda o fornecimento de dados.

Em sistemas com múltiplos processadores, quando vários núcleos possuem cópias do mesmo dado na cache, surge o problema da incoerência de cache: um processador altera a informação e os demais continuam com a versão inválida. Para garantir a integridade do sistema, utilizam-se protocolos de coerência baseados em hardware e um deles é o MOESI (Modified, Owned, Exclusive, Shared, Invalid), uma extensão do protocolo MESI.

O protocolo MOESI destaca-se pela introdução do estado Owned, que permite o compartilhamento direto de dados modificados entre caches sem a necessidade de escrever imediatamente na memória RAM. Isso faz com que o tráfego no barramento e a latência de acesso em aplicações com alto grau compartilhado de dados reduzam significativamente. Nesse contexto, será descrita a implementação de um simulador de memória cache operando sob o protocolo MOESI neste trabalho. Para demonstrar a aplicação prática destes conceitos teóricos, o simulador foi integrado a um cenário de Gestão Hospitalar, em que múltiplos profissionais (processadores) acessam e atualizam prontuários médicos (dados compartilhados) simultaneamente

2. Objetivos

O objetivo deste trabalho é desenvolver um simulador de coerência de cache utilizando a extensão de protocolo MOESI, aplicado em um sistema de Gestão Hospitalar, garantindo a integridade de dados críticos entre múltiplos profissionais de saúde. Contudo, para alcançar esse objetivo, é preciso a realização de algumas atividades:

- **Simular a Arquitetura de Hardware:** Implementar uma estrutura computacional composta por três processadores e uma memória principal compartilhada.
- **Implementar o Protocolo MOESI:** Programar o gerenciamento das transições entre os estados M, O, E , S e I, permitindo a transferência eficiente de dados entre as caches sem a necessidade de acesso constante à memória principal.
- **Aplicar Políticas de Gerenciamento de Memória:** Utilizar o algoritmo de substituição FIFO para gerenciar a rotatividade dos prontuários nas caches e a política de escrita Write-Back para otimizar o salvamento de dados alterados apenas quando necessário.
- **Validar a Integridade da Aplicação:** Demonstrar, através de uma interface de linha de comando, cenários de conflito de dados, como leitura simultânea e escrita exclusiva, comprovando que o simulador evita o uso de informações obsoletas no tratamento de pacientes.

3. Funcionamento da Cache

A implementação da memória cache foi dividida em duas estruturas de dados principais: a classe CacheLine, que representa a unidade mínima de armazenamento, e a classe Cache, que gerencia o conjunto de linhas associadas a um processador.

3.1 Estrutura da CacheLine

Cada linha da cache foi projetada para armazenar um bloco de dados, e não apenas um único valor. A classe CacheLine possui os seguintes atributos:

- **endereco_base:** Atua como a Tag da linha. Armazena o endereço inicial do bloco na Memória Principal.
- **dados:** Uma estrutura de lista com capacidade para 5 inteiros. Isso significa que cada linha de cache carrega o status de 5 pacientes consecutivos.
- **protocolo:** Uma string que armazena o estado atual do protocolo MOESI para aquele bloco.

A classe também implementa métodos auxiliares, como o **contem_endereco (endereco)**, que verifica logicamente se um endereço específico pertence ao intervalo do bloco carregado, facilitando a busca.

3.2 Gerenciamento da Cache

A classe Cache simula uma memória totalmente associativa com capacidade padrão para 5 linhas. Seu funcionamento se baseia nos seguintes mecanismos implementados:

- **Armazenamento:** Utiliza uma lista inicializada com linhas no estado "I", simulando o estado vazio inicial do hardware.
- **Busca Associativa:** O método busca percorre linearmente todas as linhas da lista para verificar se alguma delas contém o endereço requisitado e se o estado é válido, ou seja, diferente de "I".
- **Política de Substituição:** Para determinar qual linha deve ser removida quando a cache está cheia, o sistema utiliza um ponteiro denominado self.topo. Este ponteiro controla uma fila circular: a cada substituição, o índice é incrementado utilizando aritmética modular. Isso garante que o ponteiro retorne ao início da lista automaticamente após atingir o final, implementando a política FIFO, sem a necessidade de deslocar elementos na memória.
- **Inserção e Write-Back:** O método insere implementa uma lógica de três etapas:
    1. Verifica se o bloco já existe para apenas utilizá-lo.
    2. Procura por slots vazios ou inválidos para inserção direta.
    3. Caso a cache esteja cheia, executa a substituição, chamando o método remove para obter a linha, insere o novo dado na posição liberada e retorna a linha expulsa. Esse retorno permite que o barramento verifique se há necessidade de utilizar o Write-Back.

4. Funcionamento da aplicação

O simulador implementa a lógica de um Sistema de Gestão Hospitalar para demonstrar o funcionamento do protocolo de coerência de cache. Neste cenário, a integridade dos dados é mantida através do mapeamento dos componentes de hardware para elementos do cotidiano hospitalar.

A abstração do sistema associa os componentes da seguinte forma:

- **Memória Principal:** Representa o arquivo central de prontuários. Contém o registro oficial de todos os pacientes.
- **Memória Cache:** Representa a prancheta individual de cada profissional. É onde são mantidas as cópias temporárias dos prontuários que estão sendo atendidos no momento para acesso rápido.
- **Barramento:** Representa o sistema de comunicação interna. É por onde os profissionais trocam atualizações e solicitam prontuários uns aos outros.

O sistema simula a interação concorrente de três processadores, identificados na camada de aplicação como profissionais de saúde:

- Processador 0 (Médico)
- Processador 1 (Enfermeiro)
- Processador 2 (Farmacêutico)

Do ponto de vista da implementação, todos os processadores podem realizar operações de leitura ou escrita em qualquer endereço de memória a qualquer momento, ou seja, ambos podem consultar um status ou alterar um status. Essa liberdade de operação serve para simular um ambiente dinâmico e imprevisível, onde o protocolo MOESI deve ser capaz de manter a coerência independentemente de quem origina a ação, seja um médico ou enfermeiro, ambos podem iniciar uma alteração de dados e gerar um dado modificado.

Cada endereço de memória corresponde ao número do prontuário de um paciente. O dado nesse prontuário é um número de 1 a 4 que representa o status clínico:

1. Em triagem
2. Em atendimento
3. Em medicação
4. Em alta

Para demonstrar a eficácia do protocolo, a aplicação permite criar conflitos de dados em tempo real. Um exemplo de fluxo possível no simulador é:

1. O Enfermeiro consulta o Prontuário 10.
2. O Médico altera o status do Prontuário 10 para “Em alta”.
3. Automaticamente, a cópia do prontuário na prancheta do Enfermeiro é invalidada.
4. Se o Enfermeiro tentar reler esse prontuário, o sistema detecta a invalidez e força a busca da atualização.

5. Decisões de projeto para a implementação

A implementação do simulador foi guiada por princípios de modularidade e orientação a objetos, visando criar um código que fosse ao mesmo tempo fiel aos conceitos de arquitetura de computadores e flexível para a modelagem da aplicação hospitalar.

O simulador foi desenvolvido em Python. A escolha desta linguagem foi feita pela robustez na manipulação de estrutura de dados e pela clareza sintática, permitindo focar o esforço de desenvolvimento na lógica da máquina de estados do protocolo MOESI. O paradigma de orientação a objetos foi essencial para encapsular os componentes de hardware em classes distintas, facilitando a manutenção e a escalabilidade do código.

A implementação da lógica de Snooping e de coerência foi centralizada na classe Barramento. Diferente de um barramento físico que acaba sendo um meio passivo, na simulação o barramento atua como um controlador ativo que possui referências para todas as caches e para a RAM. Essa decisão técnica simplificou a implementação das transferências do estado Owner e das invalidações, pois o barramento pode acessar e modificar diretamente os estados das linhas nas caches conectadas.

A classe CacheLine foi projetada para encapsular um bloco de memória, ou seja, armazenando uma lista de 5 inteiros, permitindo que o bloco de memória seja tratado como uma unidade atômica.

Para atender ao requisito da política de substituição FIFO, foi implementada uma Fila Circular na classe Cache, evitando a necessidade de deslocar elementos a cada inserção e remoção, permitindo que o custo computacional para a realização dessas ações seja bem menor.

Para a implementação do Write-Back funcionar de forma eficiente, o método de inserção da cache retorna o objeto removido para o barramento, que faz a verificação do estado do protocolo e decide se realiza a persistência na RAM.

6. Conclusão

Este trabalho apresentou a implementação de um simulador de coerência de cache operando sob o protocolo MOESI. A modelagem de um sistema hospitalar serviu como base para a validação dos mecanismos de controle de concorrência e integridade de dados entre processadores distintos.

A análise dos resultados confirmou a eficácia do estado *Owned* na otimização do compartilhamento de dados "sujos". No cenário simulado, a transferência direta de dados entre as caches eliminou a necessidade de gravações desnecessárias na Memória Principal, resultando em um uso mais eficiente da largura de banda do barramento.

A arquitetura do software, estruturada sobre o paradigma de Orientação a Objetos, permitiu o isolamento das responsabilidades de cada componente (Barramento, Cache e Processador). A implementação da política de substituição FIFO e do mecanismo de *Write-Back* assegurou que o simulador refletisse o comportamento de memórias reais.

Por fim, a simulação demonstrou que a garantia de coerência é um requisito importante para a correção da execução paralela. O sistema impediu a leitura de valores obsoletos, validando a integridade das transações simuladas e reforçando a aplicabilidade do protocolo MOESI em arquiteturas modernas.

7. Referências

PATTERSON, David A.; HENNESSY, John L. Arquitetura de Computadores: Uma Abordagem Quantitativa. 4. ed. Rio de Janeiro: Campus-Elsevier, 2008.

PATTERSON, David A.; HENNESSY, John L. Organização e Projeto de Computadores: A Interface Hardware/Software. 3. ed. Rio de Janeiro: Campus, 2005.

PYTHON SOFTWARE FOUNDATION. Python 3.12 Documentation. Disponível em: https://docs.python.org/3/. Acesso em: nov. 2025.

STALLINGS, William. Computer Organization and Architecture: Design for Performance. 11. ed. Pearson, 2021.

COSSUL, Sandra. Materiais disponibilizados em aula.