Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Departamento de Engenharia de Computação e Automação

Componente curricular: Arquitetura de Computadores - DCA0104
Docente responsável: Diogo Pinheiro Fernandes Pedrosa

Ernane Ferreira Rocha Junior - ernane.ferreira.100@ufrn.edu.br
Quelita Míriam Nunes Ferraz - quelita.miriam.705@ufrn.edu.br
Thiago Lopes do Nascimento - thiago.lopes.702@ufrn.edu.br

RELATÓRIO DE ATIVIDADE: SIMULAÇÃO DE MEMÓRIA CACHE

NATAL/RN - 2023

1. INTRODUÇÃO

A presente tarefa visa construir uma simulação abrangente de um sistema computacional com uma única camada de cache, utilizando a linguagem Python e a biblioteca pyCacheSim, conforme mencionado na "Atividade 1". Para essa simulação, consideraremos um sistema hipotético com as seguintes especificações:

ESPECIFICAÇÕES	
Memória Principal	O sistema possui uma memória principal de 64K bytes de tamanho, endereçada a byte. Isso significa que o tamanho da palavra desse sistema é igual a 1 byte.
Processador	O sistema é composto por um único processador.
Cache	A cache de dados é do tipo simples, com capacidade de 4K bytes.

Tabela 1. Especificações de projeto

Com base nessas configurações, a simulação permitirá testar diversas combinações de parâmetros, como tamanho da cache, política de substituição de dados e organização física da memória cache. Serão obtidas informações precisas sobre as taxas de acerto e falha associadas a cada configuração.

Utilizando a biblioteca PyCacheSim em Python, poderemos implementar a lógica necessária para simular o comportamento do sistema com as diferentes configurações de cache. A simulação irá gerar dados sobre as taxas de acerto, que representam a proporção de solicitações de dados atendidas pela cache, e as taxas de falha, que indicam quantas vezes a cache não possui os dados necessários e precisa buscar na memória principal.

Além disso, a simulação nos permitirá avaliar o desempenho do sistema em termos de latência de acesso à memória e taxa de transferência de dados. Serão realizados experimentos nos quais diferentes tipos de aplicações serão executados, possibilitando a análise de como as configurações de cache influenciam o tempo de resposta do sistema.

Com base nos resultados obtidos, será possível identificar as configurações de cache mais eficientes para diferentes cenários de uso. Essas informações serão valiosas para os projetistas de sistemas computacionais, auxiliando-os na tomada de decisões sobre o dimensionamento adequado da cache e suas características internas.

Assim, a construção dessa simulação em Python, utilizando a biblioteca PyCacheSim, proporcionará uma compreensão mais abrangente dos sistemas computacionais com cache única.

2. METODOLOGIA

A construção desse projeto foi realizada com a linguagem de programação Python e o sucesso e velocidade de produção ocorreu graças à biblioteca PyCacheSim que é uma ferramenta poderosa e versátil para simulação de sistemas de cache em computação. Com ela, é possível construir simulações completas e realistas de caches de dados, explorando diferentes configurações e analisando o desempenho do sistema.

A PyCacheSim permite que os desenvolvedores experimentem com parâmetros como tamanho da cache, política de substituição de dados e organização física da memória cache. Isso possibilita e facilita a obtenção de informações precisas sobre as taxas de acerto e falha da cache em diferentes cenários de uso. Além disso, a biblioteca busca facilitar o desenvolvimento em alguns aspectos como, por exemplo, em nenhum momento foi necessário dividir os endereços de memória a serem lidos para explicitar se o tipo de mapeamento utilizado é direto, associativo por conjuntos ou totalmente associativo. Isso porque, pela própria forma que a biblioteca foi implementada, o algoritmo de mapeamento é simulado conforme a forma que os parâmetros de construção na classe *Cache* são apresentados.

Ao utilizar a PyCacheSim, é possível avaliar o desempenho do sistema em termos de latência de acesso à memória e taxa de transferência de dados. A biblioteca permite a execução de experimentos com diferentes tipos de aplicações, o que possibilita uma análise abrangente do comportamento da cache e suas influências no tempo de resposta do sistema.

Em resumo, a PyCacheSim é uma biblioteca essencial para quem deseja explorar e compreender o funcionamento de sistemas de cache em computação. Com ela, é possível obter dados valiosos sobre as taxas de acerto e falha, analisar o desempenho do sistema e tomar decisões informadas para otimização e aprimoramento de sistemas computacionais.

2.1. CÓDIGO UTILIZADO

O código produzido consiste em um programa que realiza simulações de diferentes configurações de mapeamento de cache e políticas de substituição. O objetivo é analisar o desempenho dessas configurações e gerar relatórios com os resultados.

Para isso, foi desenvolvido métodos adicionais para permitir executar as simulações e gerar os resultados para análise, sendo eles: *simulate, file_handler e report_data*. A função principal `main()` é definida logo após as importações. Essa função coordena a execução das simulações e a geração dos relatórios de dados.

```
1    from simulator import simulate
2    from file_handler import clear_file, append_data_to_file
3    from report_data import reports_direct_and_fully_associative_mapping, report_associative_mapping_by_sets
4
```

Figura 1. Importação de métodos

Antes de iniciar as simulações, o programa realiza algumas operações de preparação. Primeiro, chama a função *clear_file()* do módulo *file_handler* para limpar o conteúdo do arquivo chamado "result.csv" que será onde estará o resultado da simulação atual. Isso é feito para garantir que o conteúdo do arquivo corresponderá somente à execução atual após o seu término. Em seguida, utiliza a função *append_data_to_file()* para adicionar um cabeçalho ao arquivo, especificando as colunas dos dados que serão registrados.

```
5   def main():
6       clear_file("./data/result.csv")
7       append_data_to_file(f"./data/result.csv", 'mapping,num_sets,num_ways,block_size,hit_percent,miss_percent,rep_policy')
```

Figura 2. preparando o arquivo de resultados

A variável *cache_size* é definida com o valor 4096 (bytes), que representa o tamanho total da cache utilizada nas simulações. A variável *policies* é uma lista que contém três políticas de substituição de cache previstas para simulação: 'FIFO', 'LRU' e 'RR' (do inglês First in - first out, Least Recently Used e Random Replacement, respectivamente).

```
9       cache_size = 4096 # (B)
10      policies = ['FIFO', 'LRU', 'RR']
```

Figura 3. variáveis da arquitetura

A primeira parte das simulações é realizada em um loop, que executa quatro iterações. Cada iteração corresponde a um teste de mapeamento direto com tamanhos de bloco diferentes. O valor dos bits referenciado por “n” é calculado como 8 multiplicado por 2 elevado à i-ésima potência, onde `i` varia de 0 a 3. A função *simulate()* é chamada para realizar a simulação de um mapeamento direto ('dm') com os parâmetros especificados.

```
12      for i in range(4): # => Direct with 8B, 16B, 32B and 64B blocks
13          n = 8 * 2**i
14          simulate('dm', cache_size//n, 1, n, 'LRU')
```

Figura 4. Simulação com mapeamento direto

A segunda parte das simulações é realizada em um loop aninhado. O loop externo itera sobre as políticas de substituição de cache presentes na lista *policies*. O loop interno executa quatro iterações, semelhante à parte anterior das simulações. Cada iteração corresponde a um teste de mapeamento associativo por conjunto com diferentes números de vias. A função *simulate()* é chamada novamente para realizar a simulação de um mapeamento associativo por conjunto ('as') com os parâmetros adequados.

```
16      for policy in policies: # => Associative per set with 2, 4, 8 and 16 ways
17          for i in range(4):
18              n = 8 * 2**i
19              simulate('as', 4*cache_size//(n*n), n//4, n, policy)
```

Figura 5. Mapeamento associativo por conjuntos

Em seguida, o valor da variável *cache_size* é atualizado para 1024 (bytes) e são realizadas as simulações de mapeamento totalmente associativo. O loop externo itera sobre as políticas de substituição de cache presentes em *policies*, e o loop interno executa quatro iterações semelhantes às anteriores. A função *simulate()* é chamada novamente para simular o mapeamento totalmente associativo ('fas') com os parâmetros apropriados.

```
21      cache_size = 1024
22
23      for policy in policies: # => Fully associative with 8B, 16B, 32B, and 64B blocks
24        for i in range(4):
25            n = 8 * 2**i
26            simulate('fas', 1, cache_size//n, n, policy)
```

Figura 6. Simulações de mapeamento totalmente associativo.

Após a conclusão das simulações, são chamadas algumas funções específicas. A função *reports_direct_and_fully_associative_mapping()* do módulo *report_data* é chamada quatro vezes, passando diferentes argumentos para gerar relatórios sobre os resultados do mapeamento direto e totalmente associativo. A função *report_associative_mapping_by_sets()* também é chamada três vezes para gerar relatórios sobre os resultados do mapeamento associativo por conjunto. [1]

```
28      reports_direct_and_fully_associative_mapping("Mapeamento direto","dm","LRU")
29      reports_direct_and_fully_associative_mapping("Totalmente associativo (FIFO)","fa","FIFO")
30      reports_direct_and_fully_associative_mapping("Totalmente associativo (LRU)","fa","LRU")
31      reports_direct_and_fully_associative_mapping("Totalmente associativo (Aleatório)","fa","RR")
32
33      report_associative_mapping_by_sets("Associativo por conjuntos (FIFO)","FIFO")
34      report_associative_mapping_by_sets("Associativo por conjuntos (LRU)","LRU")
35      report_associative_mapping_by_sets("Associativo por conjuntos (Aleatório)","RR")
36
37  main()
```

Figura 7. Chamada de relatórios e execução do programa

3. RESULTADOS

Para cada simulação de cache realizada no código apresentado, o endereço da memória principal é dividido em campos que são interpretados de maneira diferente, dependendo do tipo de mapeamento de cache.

No mapeamento direto, cada bloco de memória principal é mapeado em uma única entrada de cache. Isso significa que um bloco de memória só pode ser armazenado em uma posição específica na cache. Por ser uma implementação mais simples, de baixo custo e que apresenta baixa latência de acesso quando ocorre um acerto. Entretanto, apresenta baixa taxa de acertos quando vários blocos de

[1] Para mais detalhes sobre os métodos utilizados e averiguar o código na íntegra acesse o link: https://github.com/ErnaneJ/cachesim-mapping-ca

memória tentam ser armazenados no mesmo conjunto da cache, resultando em uma alta taxa de substituição, como demonstrado na figura 8.

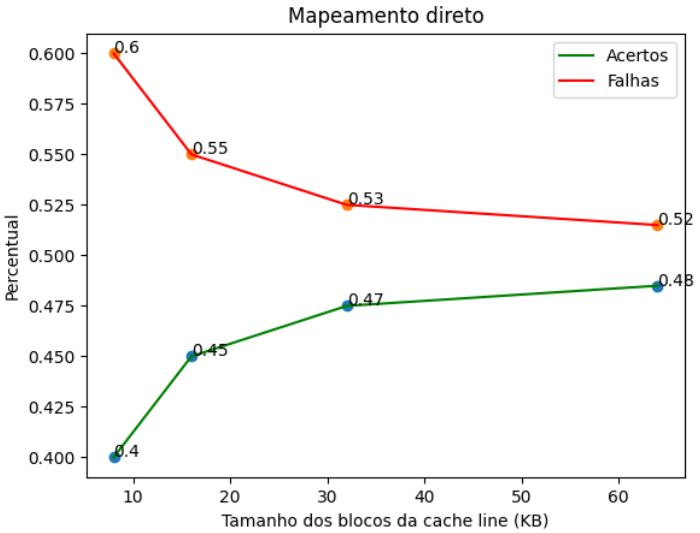

Figura 8: Resultado do mapeamento direto

No mapeamento totalmente associativo, onde qualquer bloco de memória pode ser armazenado em qualquer posição na cache, não há conjuntos fixos, e a cache procura em todas as linhas para encontrar um bloco solicitado. Isso elimina os conflitos de mapeamento, mas aumenta a complexidade e o custo do hardware.

Sendo assim, para o mapeamento totalmente associativo com a política de substituição FIFO, tem-se alta taxa de acertos em situações em que os programas apresentam padrões de acesso sequenciais ou localidade temporal, onde os blocos acessados recentemente têm maior probabilidade de serem acessados novamente no futuro próximo. Entretanto, a política FIFO não leva em consideração a frequência ou a relevância do acesso a um bloco de memória, apresentando assim um desempenho baixo e, consequentemente, a uma menor taxa de acertos. Isso pode ser descrito graficamente como demonstrado na figura 9.

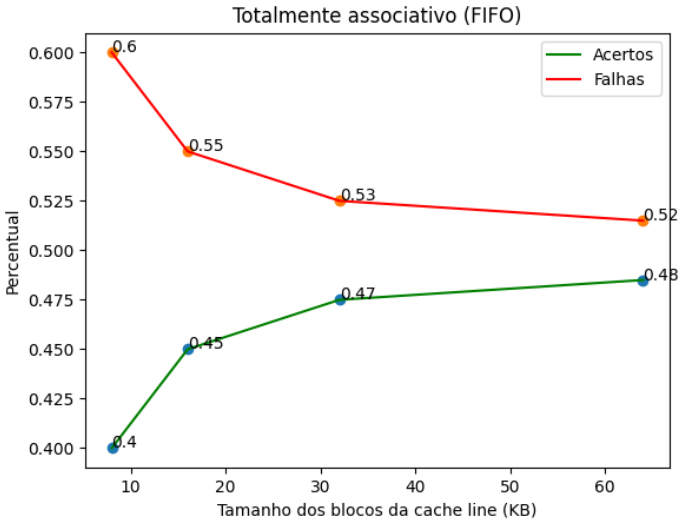

Figura 9: Resultado do mapeamento totalmente associativo (FIFO)

O mapeamento totalmente associativo, com a política de substituição LRU, geralmente apresenta um desempenho melhor em comparação com a política FIFO apresentando alta taxa de acertos em programas com padrões de acesso não uniformes ou com alta localidade espacial, pois leva em consideração a frequência de acesso aos blocos de memória, mantendo os blocos mais recentemente acessados na cache por mais tempo. Entretanto, por ser uma implementação mais complexa e custosa do que a política FIFO, tem-se um possível impacto na latência de acesso e, consequentemente, uma menor taxa de erros, pois a política LRU também requer acompanhamento e atualização constante da ordem dos blocos na cache.

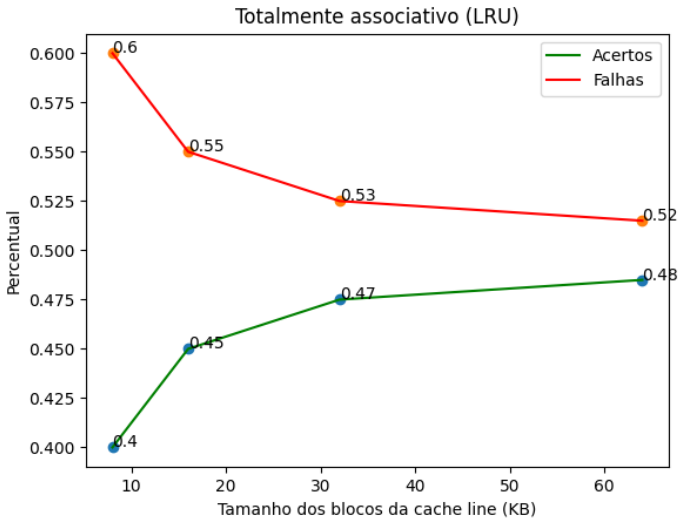

FIgura 10: Resultado do mapeamento totalmente associativo (LRU)

A política de substituição Random Replacement seleciona aleatoriamente um bloco da cache para ser substituído quando ele não está presente na cache e precisa ser buscado na memória principal. Essa política é especialmente útil em cenários com padrões de acesso aleatórios, onde não há uma

correlação clara entre a ordem de acesso aos blocos e sua importância ou frequência de acesso. No entanto, em programas com padrões de acesso com alta localidade temporal, a política de substituição aleatória pode ter um desempenho inferior, com uma menor taxa de acertos, em comparação com políticas mais adaptativas, como LRU, que consideram a ordem de acesso dos blocos, como pode ser descrito graficamente na figura abaixo.

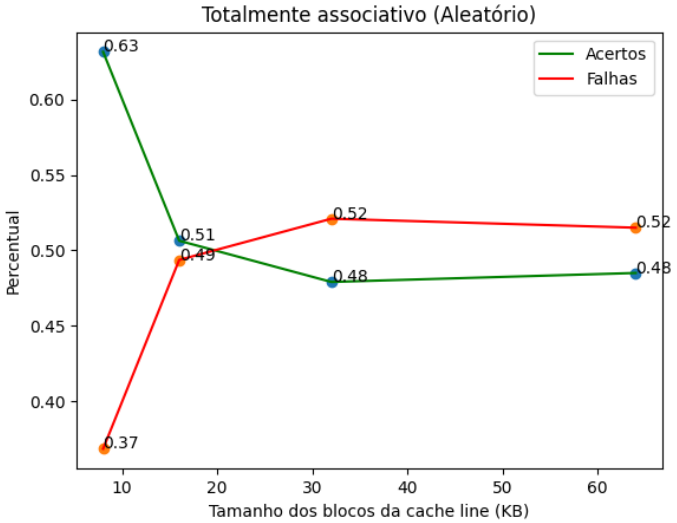

Figura 11: Resultado do mapeamento totalmente associativo (RR)

A política FIFO no mapeamento associativo por conjunto seleciona o bloco mais antigo dentro de um conjunto para ser substituído quando o bloco de memória solicitado não está presente na cache e precisa ser buscado na memória principal. Isso garante que o bloco que está há mais tempo no conjunto seja substituído primeiro. Em geral, o mapeamento associativo por conjunto com a política FIFO oferece uma taxa de acertos razoável em programas com padrões de acesso com alguma localidade espacial. No entanto, em programas com conflitos frequentes em conjuntos específicos ou com alta localidade espacial, pode ocorrer uma alta taxa de substituição de blocos, reduzindo o desempenho da cache.

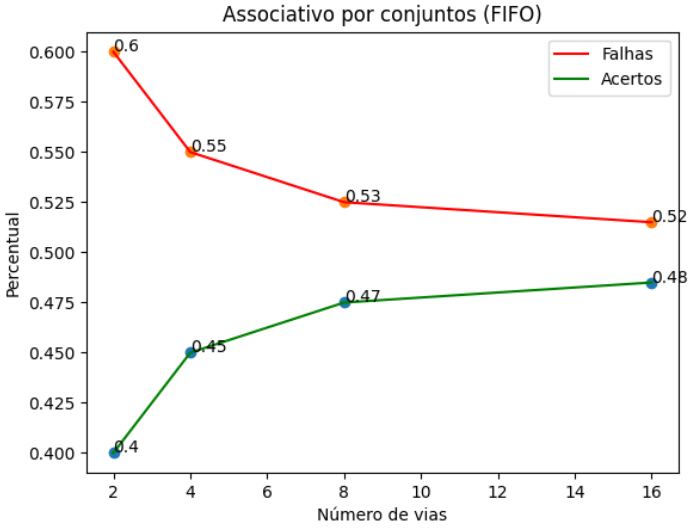

Figura 12: Resultado do mapeamento associativo por conjuntos (FIFO)

A política LRU no mapeamento associativo por conjunto mantém o controle da ordem de acesso aos blocos dentro de cada conjunto. Quando um bloco de memória não está presente na cache e precisa ser buscado na memória principal, o bloco menos recentemente usado dentro do conjunto é selecionado para ser substituído pelo novo bloco.

Este tipo de configuração geralmente oferece um bom desempenho em uma ampla variedade de padrões de acesso à memória, como pode-se notar na figura 13, especialmente em programas com alta localidade espacial ou com a repetição frequente de acessos a um conjunto específico de blocos. No entanto, é importante destacar que a implementação da política LRU é mais complexa e custosa do que a política FIFO e de possível impacto na latência de acesso, pois a política LRU requer acompanhamento e atualização constante da ordem dos blocos na cache.

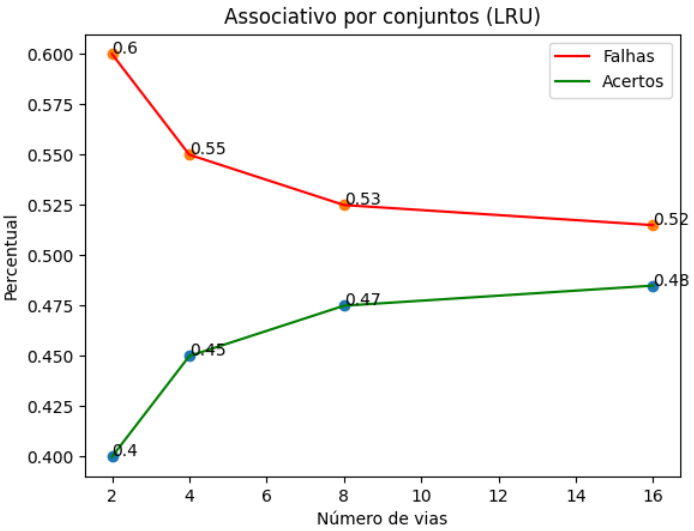

FIgura 13: Resultado do mapeamento associativo por conjuntos (LRU)

Para o mapeamento associativo por conjuntos com a política de substituição aleatória (RR), tem-se simplicidade de implementação, pois não requer o acompanhamento da ordem de acesso dos blocos e ausência de padrões previsíveis de substituição, o que pode ser benéfico em programas com padrões de acesso aleatórios ou não uniformes. Entretanto, apresenta menor taxa de acertos em comparação com políticas mais adaptativas, como LRU, em programas com padrões de acesso que exibem localidade temporal e baixa capacidade de aprendizado sobre padrões de acesso, o que pode resultar em uma taxa de substituição maior do que o necessário.

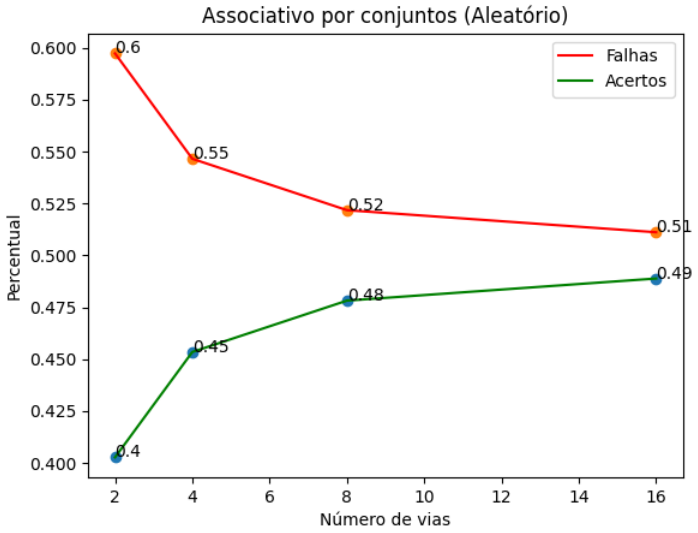

Figura 14: Resultado do mapeamento associativo por conjuntos (RR)

4. CONCLUSÃO

Esse trabalho permitiu explorar e analisar de forma detalhada o desempenho da memória cache por meio de simulações e geração de gráficos. As dificuldades encontradas na escrita do código foram superadas com o uso da biblioteca *PyCacheSim,* que se mostrou uma ferramenta útil e eficiente. A sua interface intuitiva e a disponibilidade de funções específicas para simulações de memória cache facilitaram o desenvolvimento do código. No entanto, algumas dificuldades foram encontradas, especialmente no que diz respeito à configuração correta dos parâmetros da cache, como tamanho, associatividade e política de substituição. Os resultados obtidos, apresentados em termos do total de erros e acertos observados, forneceram insights sobre o funcionamento da memória cache e a importância dos parâmetros de configuração. Além disso, as comparações realizadas entre as diferentes estratégias de mapeamento evidenciaram suas características e desempenhos distintos. Essas informações são relevantes para o projeto e otimização de sistemas de memória cache, contribuindo para a melhoria do desempenho dos computadores.