Unidade 3 - Análise Comparativa de Algoritmos de Substituição de Páginas em um Simulador de Memória Virtual

Emerson da S. Santos[1], João Pedro dos S. Medeiros[1], João Batista B. Neto[2]

[1]Bacharelado em Sistemas de Informação (BSI)
Universidade Federal do Rio Grande do Norte (UFRN) – Campus CERES
Rua Joaquim Gregório, 296, Penedo, 59.300-000 – Caicó – RN – Brasil

[2]Departamento de Computação e Tecnologia (DCT)
Universidade Federal do Rio Grande do Norte (UFRN) – Campus CERES
Rua Joaquim Gregório, 296, Penedo, 59.300-000 – Caicó – RN – Brasil

emerson.santos.125@ufrn.edu.br, jopesame@gmail.com, joao.borges@ufrn.br

Abstract. *This paper presents the implementation and a comparative performance analysis of six page replacement algorithms within a virtual memory simulator. The work was based on an existing simulator, extending it to support First-In, First-Out (FIFO), Least Recently Used (LRU), Second Chance (SC), Least Frequently Used (LFU), and Most Frequently Used (MFU), in addition to the pre-existing Random algorithm. The performance analysis was conducted through automated simulations using four different memory access traces and varying the number of available memory frames from 2 to 64. The primary metrics for evaluation were the total number of Page Faults and the number of Page Writes to disk. The results are presented in line graphs, allowing for a clear visual assessment of the performance and I/O cost of each algorithm under different workloads.*

Resumo. *Este trabalho apresenta a implementação e a análise de performance comparativa de seis algoritmos de substituição de páginas em um simulador de memória virtual. O trabalho foi baseado em um simulador existente, estendendo-o para suportar os algoritmos First-In, First-Out (FIFO), Least Recently Used (LRU), Second Chance (SC), Least Frequently Used (LFU) e Most Frequently Used (MFU), além do algoritmo Aleatório (Random) pré-existente. A análise de desempenho foi realizada por meio de simulações automatizadas, utilizando quatro diferentes traces de acesso à memória e variando a quantidade de frames de memória disponíveis de 2 a 64. As métricas principais de avaliação foram o número total de Faltas de Página (Page Faults) e o número de Páginas Escritas em disco (Page Writes). Os resultados são apresentados em gráficos de linha, permitindo a análise visual do desempenho e do custo de E/S de cada algoritmo sob diferentes cargas de trabalho.*

1. Introdução

O Sistema Operacional (SO) atua como um gerenciador de recursos complexo, sendo uma de suas responsabilidades mais críticas o *gerenciamento de memória*. A memória virtual permite a um processo utilizar um espaço de endereçamento muito maior do que a memória física (RAM) realmente disponível. Quando um processo tenta aceder uma

porção de seu espaço de endereçamento que não está na memória física, ocorre uma *falta de página (page fault)*. O SO deve então tratar essa falta, carregando a página do disco para a memória. Se não houver frames de memória livres, o SO precisa selecionar uma página ”vítima”para ser removida. Se esta página vítima foi modificada (está ”suja”), ela precisa ser escrita de volta no disco antes de ser substituída, gerando uma operação de *page write*, que é uma das operações mais custosas em um sistema. A eficiência do *algoritmo de substituição de páginas* é, portanto, crucial não apenas para minimizar as faltas de página, mas também para reduzir o I/O de escrita, evitando o *thrashing*.

Este trabalho visa a implementação e análise de cinco algoritmos de substituição: *FIFO*, *LRU*, *SC*, *LFU* e *MFU*. A metodologia envolve a execução automatizada de cada algoritmo sob diferentes cargas de trabalho (traces) e com um número crescente de frames (2 a 64), analisando tanto as faltas de página quanto as escritas em disco.

2. Metodologia

A implementação dos algoritmos foi desenvolvida em C no simulador `virtmem_sim`. Quando uma falta de página ocorre sem frames livres, uma função de seleção de vítima é chamada.

2.1. Algoritmo FIFO (First-In, First-Out)

Seleciona a página que está na memória há mais tempo, baseado no campo `load_time`.

```
int selectVictimFIFO(void) {
  int i, victim = 0;
  unsigned int oldest_load_time = -1;
  for (i = 0; i < n_frames; i++) {
    if (frameTable[i].load_time < oldest_load_time) {
      oldest_load_time = frameTable[i].load_time;
      victim = i;
    }
  }
  return victim;
}
```

Listing 1. Algoritmo FIFO

2.2. Algoritmo LRU (Least Recently Used)

Seleciona a página que não é acedida há mais tempo, baseado no campo `last_reference_time`.

```
int selectVictimLRU(void) {
  int i, victim = 0;
  unsigned int oldest_ref_time = -1;
  for (i = 0; i < n_frames; i++) {
    if (frameTable[i].last_reference_time < oldest_ref_time) {
      oldest_ref_time = frameTable[i].last_reference_time;
      victim = i;
    }
  }
```

```
  return victim;
}
```

Listing 2. Algoritmo LRU

2.3. Algoritmo SC (Second Chance)

Melhoria do FIFO que usa um ponteiro circular e um *bit de referência* (`referenced`) para dar uma segunda chance a páginas recentemente acedidas.

```
int selectVictimSC(void) {
  static int current_victim = 0;
  while(1){
    if(frameTable[current_victim].referenced == 0){
        int victim = current_victim;
        current_victim = (current_victim + 1) % n_frames;
        return victim;
    } else {
        frameTable[current_victim].referenced = 0;
        current_victim = (current_victim + 1) % n_frames;
    }
  }
}
```

Listing 3. Algoritmo Segunda Chance

2.4. Algoritmo LFU (Least Frequently Used)

Seleciona a página acedida o menor número de vezes, baseado no contador `reference_num`.

```
int selectVictimLFU(void) {
  int i, victim = 0;
  unsigned int min_references = -1;
  for (i = 0; i < n_frames; i++) {
    if (frameTable[i].reference_num < min_references) {
      min_references = frameTable[i].reference_num;
      victim = i;
    }
  }
  return victim;
}
```

Listing 4. Algoritmo LFU

2.5. Algoritmo MFU (Most Frequently Used)

Inverso do LFU, seleciona a página acedida o maior número de vezes.

```
int selectVictimMFU(void) {
  int i, victim = 0;
  unsigned int max_references = 0;
```

```
4     for (i = 0; i < n_frames; i++) {
5       if (frameTable[i].reference_num > max_references) {
6         max_references = frameTable[i].reference_num;
7         victim = i;
8       }
9     }
10    return victim;
11  }
```

Listing 5. Algoritmo MFU

3. Resultados e Discussão

Nesta seção, são apresentados os resultados para as duas métricas: Faltas de Página e Páginas Escritas.

3.1. Análise de Faltas de Página (Page Faults)

O número de page faults indica a frequência com que o sistema precisa buscar dados no disco, impactando diretamente a latência. A análise das Figuras 1 a 4 confirma o comportamento esperado:

- **Superioridade do LRU/LFU:** Em todos os cenários, os algoritmos LRU e LFU demonstram ser os mais eficientes, minimizando as faltas ao reterem páginas relevantes, o que evidencia sua excelente adaptação ao princípio da localidade de referência.
- **Ineficácia do MFU:** O MFU consistentemente apresenta o pior desempenho. Sua heurística de remover a página mais usada é contraproducente para padrões de acesso comuns, onde páginas frequentemente usadas têm alta probabilidade de serem acedidas novamente.
- **Comportamento Intermediário:** FIFO, SC e Random formam um grupo intermediário. O SC confirma sua vantagem teórica sobre o FIFO, apresentando um número de falhas ligeiramente menor na maioria dos casos.
- **Impacto dos Frames:** Com o aumento do número de frames, todos os algoritmos (exceto o MFU) melhoram, pois mais páginas do ”working set”do processo cabem na memória.

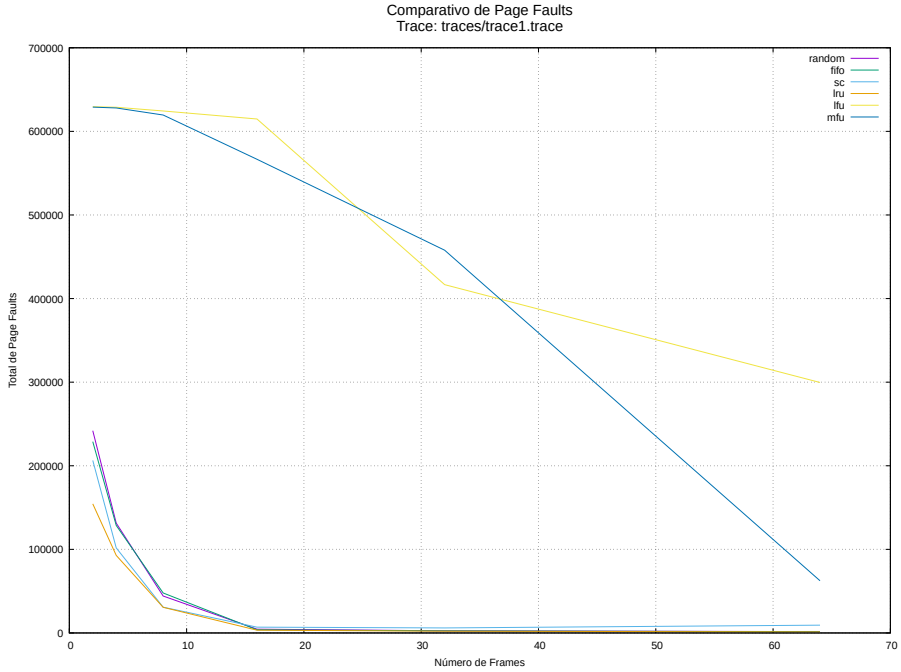

Figura 1. Page Faults para o trace1.trace.

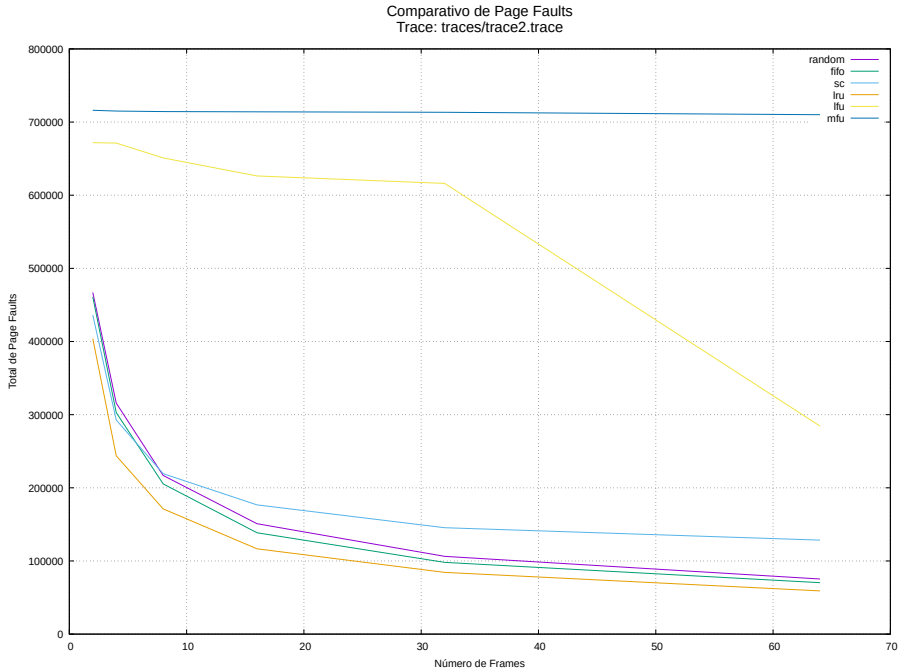

Figura 2. Page Faults para o trace2.trace.

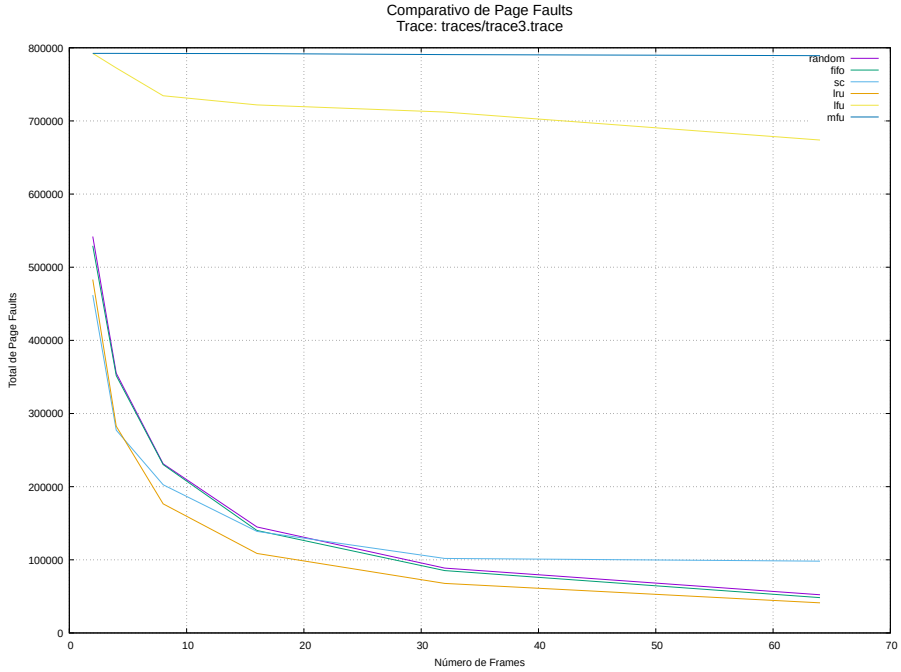

Figura 3. Page Faults para o trace3.trace.

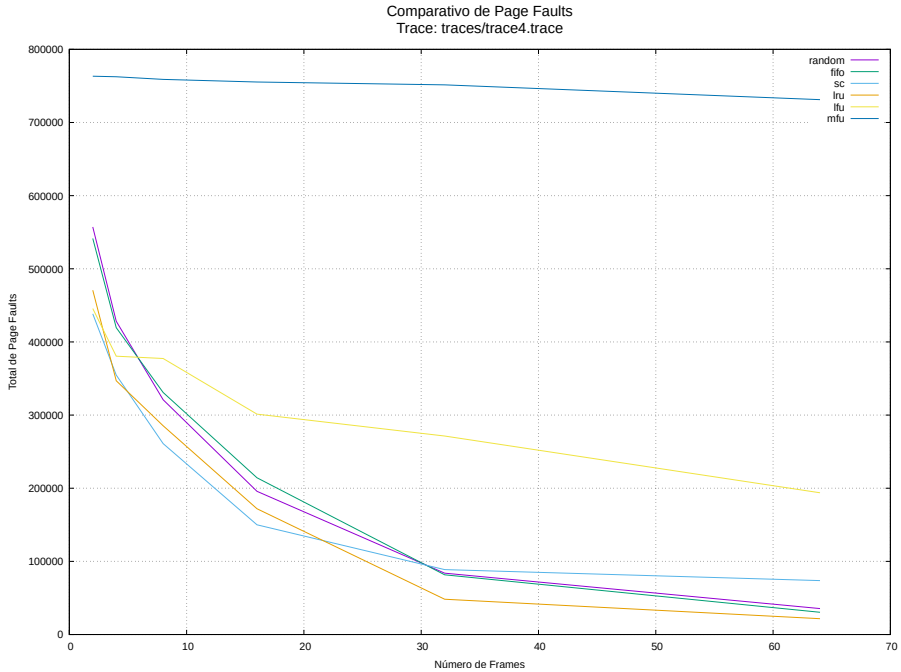

Figura 4. Page Faults para o trace4.trace.

3.2. Análise de Páginas Escritas (Page Writes)

O número de page writes representa o custo de I/O de escrita em disco, uma das operações mais lentas do sistema. Um bom algoritmo deve minimizar a necessidade de salvar páginas ”sujas”(modificadas). A análise das Figuras 5 a 8 revela:

- **Correlação com Page Faults:** A hierarquia de desempenho para page writes é muito semelhante à dos page faults. Algoritmos que causam menos faltas de página também tendem a realizar menos escritas em disco.

- **Eficiência do LRU/LFU:** Por manterem as páginas mais importantes na memória por mais tempo, LRU e LFU diminuem a probabilidade de que uma página frequentemente modificada seja removida prematuramente, reduzindo assim o número de escritas necessárias.

- **Custo do MFU:** O MFU, ao remover páginas ativas, não só aumenta os page faults, mas também eleva a chance de descartar uma página ”suja”, resultando em mais operações de escrita e maior sobrecarga de I/O.

Esta forte correlação entre as duas métricas era esperada. Menos substituições de página significam menos decisões sobre o que remover e, consequentemente, menos oportunidades para que uma página suja precise ser escrita no disco.

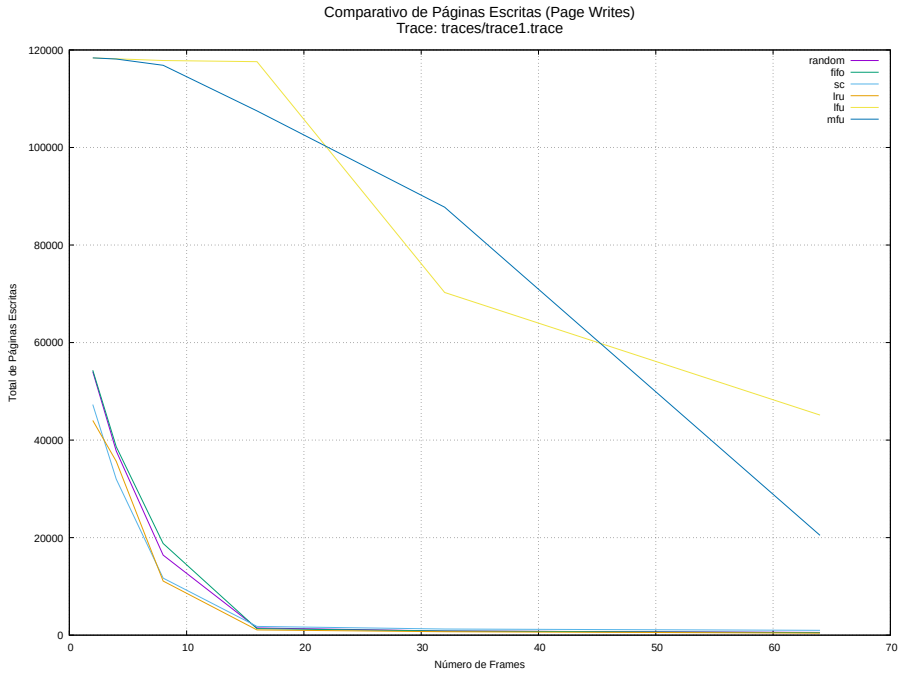

Figura 5. Page Writes para o trace1.trace.

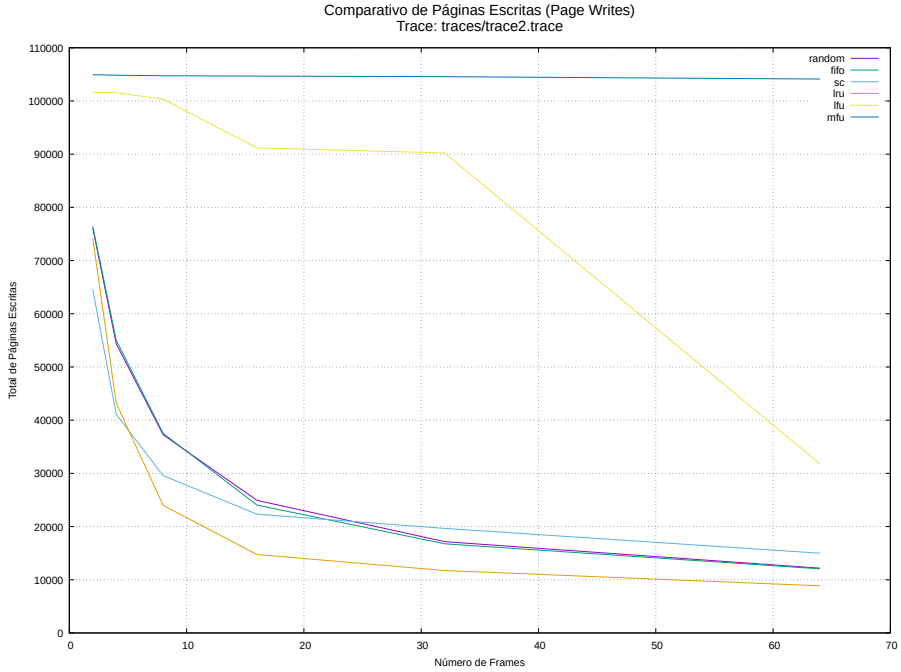

Figura 6. Page Writes para o trace2.trace.

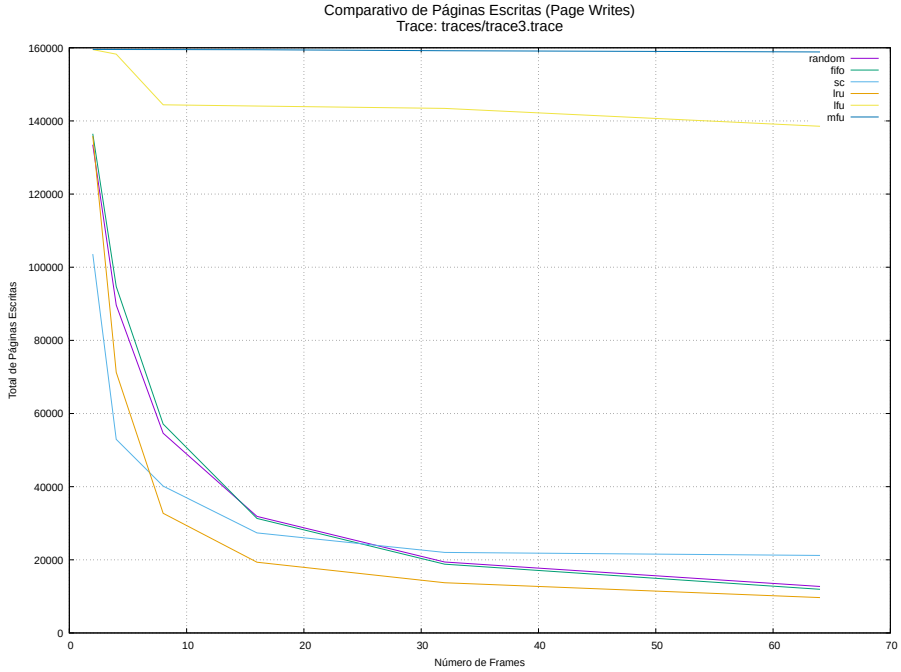

Figura 7. Page Writes para o trace3.trace.

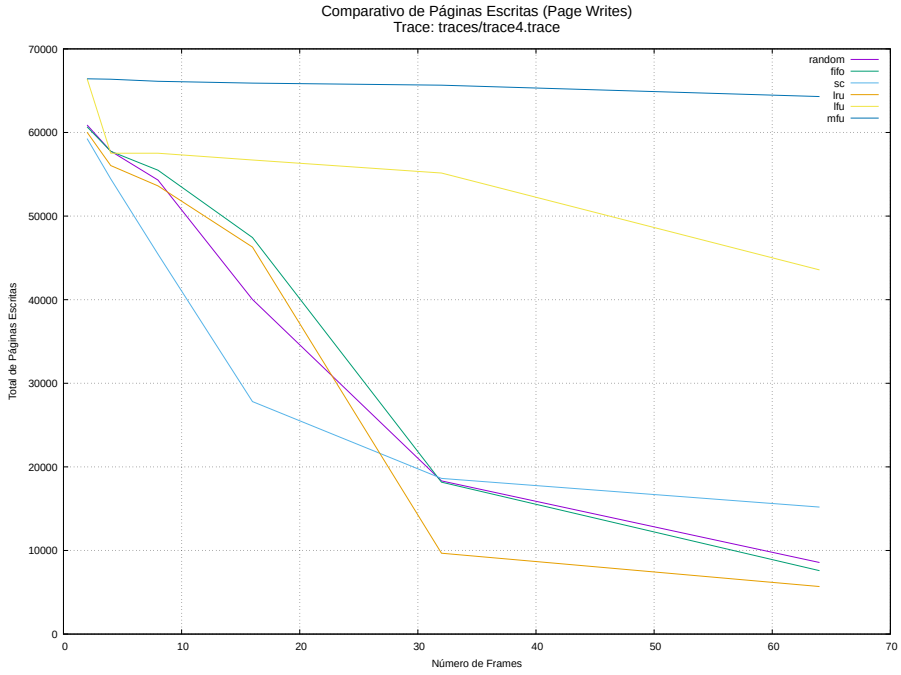

Figura 8. Page Writes para o trace4.trace.

4. Conclusão

Este trabalho aborda a implementação e análise de cinco algoritmos de substituição de páginas, avaliando-os com base nas métricas de Faltas de Página e Páginas Escritas. Os resultados validaram a teoria de sistemas operativos: algoritmos como LRU e LFU, que

exploram a localidade de referência, são mais eficientes na redução de ambas as métricas, diminuindo tanto a latência (menos page faults) quanto o custo de I/O (menos page writes). Em contrapartida, o MFU provou ser contraproducente em todos os cenários.

Referências

Silberschatz, A., Galvin, P. B., and Gagne, G. (2015). *Fundamentos de Sistemas Operacionais*. LTC, Rio de Janeiro, 9ª edition.