Simulador de Arquitetura Multicore com Escalonamento e Gerência de Memória

1nd Julia Mello Lopes Gonçalves
Centro Federal de Educação Tecnológica de Minas Gerais
(CEFET/MG)
Divinópolis, Brasil
juliamellolopesoncalves@gmail.com

***Abstract*—Este artigo apresenta o desenvolvimento de um simulador de arquitetura multicore com suporte a escalonamento de processos e gerência de memória hierárquica. O projeto foi desenvolvido para demonstrar o funcionamento interno de componentes como CPU, pipeline, memória RAM e cache, além de ilustrar os impactos de diferentes políticas de escalonamento. Resultados obtidos através da simulação demonstram a eficácia do uso de pipelines e a influência do gerenciamento de memória no desempenho do sistema.**

***Index Terms*—CPU, pipeline, escalonamento, memória hierárquica, sistemas multicore**

I. Introdução

Com o aumento da complexidade e demanda por desempenho em sistemas computacionais modernos, as arquiteturas multicore se tornaram uma solução essencial. Essas arquiteturas permitem a execução paralela de tarefas, distribuindo o processamento entre múltiplos núcleos para melhorar a eficiência e reduzir o tempo de resposta. Além disso, o uso de pipelines, uma técnica que divide a execução de instruções em estágios sequenciais, maximiza a utilização dos recursos internos da CPU.

Este trabalho apresenta um simulador que busca replicar elementos fundamentais de uma arquitetura multicore real, incorporando conceitos como gerenciamento hierárquico de memória (RAM e cache), execução paralela em pipelines e escalonamento de processos. Além de reproduzir o funcionamento técnico, o simulador é uma ferramenta para análise crítica das políticas de escalonamento e do impacto da threads na eficiência do sistema. As políticas implementadas, como FCFS, Round Robin e Prioridade, fornecem cenários distintos que destacam os desafios e benefícios de diferentes abordagens.

O diferencial deste simulador está na integração de componentes como a Unidade de Controle (UC), memória cache e pipeline, que trabalham em conjunto para simular o fluxo de execução de instruções em um ambiente controlado. Isso permite que os estudantes compreendam não apenas o funcionamento isolado de cada módulo, mas também suas interdependências e impactos no desempenho geral.

II. Metodologia

A. Arquitetura Von Neumann com Pipeline MIPS

A arquitetura Von Neumann é um modelo clássico que define a estrutura básica de computadores modernos. Neste modelo, a memória é compartilhada entre dados e instruções, sendo acessada sequencialmente pela CPU. Para aumentar a eficiência, o simulador implementa um pipeline MIPS, que divide a execução das instruções em cinco estágios principais:

- **Instruction Fetch (IF)**: A instrução é buscada na memória usando o contador de programa (*Program Counter - PC*) para localizar seu endereço.
- **Instruction Decode (ID)**: A instrução buscada é decodificada para identificar a operação a ser realizada e os registradores envolvidos.
- **Execute (EX)**: A Unidade Lógica e Aritmética (ULA) realiza operações aritméticas ou lógicas com base nos operandos decodificados.
- **Memory Access (MEM)**: O estágio acessa a memória para leitura ou escrita de dados, dependendo da operação.
- **Write Back (WB)**: Os resultados são armazenados de volta nos registradores, tornando-os disponíveis para futuras operações.

Na Figura 1, é apresentado o fluxo de execução no pipeline MIPS. Cada estágio opera de forma independente, permitindo que várias instruções sejam processadas simultaneamente em diferentes fases. Essa paralelização aumenta significativamente o throughput do sistema ao reduzir o tempo ocioso dos recursos.

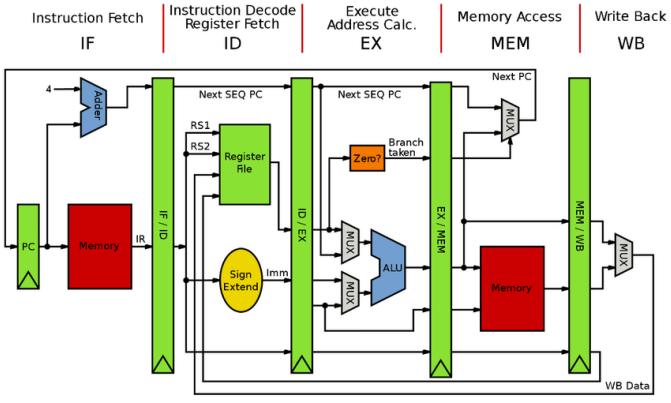

Fig. 1. Diagrama da Arquitetura de Von Neumann com Pipeline MIPS.

a) Eficiência do Pipeline: O pipeline MIPS é uma solução eficiente para aproveitar a execução paralela. No

entanto, ele também apresenta desafios, como *hazards* estruturais, de dados e de controle, que precisam ser gerenciados para evitar conflitos entre instruções. No simulador, esses aspectos foram simplificados, permitindo que o foco esteja na demonstração educacional do pipeline.

b) Relevância no Simulador: A implementação do pipeline MIPS no simulador é essencial para demonstrar como as instruções podem ser executadas em paralelo, destacando os benefícios e desafios dessa abordagem. Esse modelo educacional oferece uma visão clara de como arquiteturas modernas otimizam o desempenho de sistemas computacionais.

B. Arquitetura multicore

A arquitetura multicore implementada no simulador permite a execução paralela de processos em múltiplos núcleos. Cada núcleo tem seus próprios registradores e compartilha a memória cache e RAM. A implementação multicore é fundamental para aproveitar ao máximo o paralelismo em sistemas modernos.

a) Modificações Realizadas: No código, a classe `CPU` foi ajustada para gerenciar múltiplos núcleos, permitindo que threads independentes processassem diferentes conjuntos de instruções simultaneamente. O pseudocódigo a seguir ilustra a execução:

```
for core in cores:
    create_thread(core.execute_pipeline)
```

b) Vantagens do Multicore: A implementação de uma arquitetura multicore no simulador trouxe diversas melhorias significativas ao desempenho do sistema. As principais vantagens são descritas abaixo:

- **Redução do Tempo de Execução**: A execução paralela permite que múltiplos núcleos processem diferentes tarefas simultaneamente, diminuindo o tempo total necessário para completar a execução dos processos.
- **Aumento da Eficiência Computacional**: O uso de núcleos independentes melhora o aproveitamento dos recursos disponíveis, reduzindo gargalos e evitando ociosidade na CPU.
- **Escalabilidade**: O design multicore possibilita que o sistema seja expandido facilmente, adicionando mais núcleos para suportar uma carga maior de trabalho, sem necessidade de alterações estruturais significativas.
- **Tolerância a Processos de Longa Duração**: Em cenários com processos que demandam muito tempo, outros núcleos podem continuar executando tarefas menores, garantindo uma execução mais equilibrada.

C. Escalonador

O escalonador é responsável por controlar a ordem de execução dos processos no simulador, garantindo o uso eficiente dos recursos da CPU. O sistema implementa três políticas de escalonamento distintas, cada uma com características únicas que afetam diretamente o desempenho e a experiência do usuário. Abaixo, detalhamos cada política e seu respectivo pseudocódigo.

*1) **FCFS (First-Come, First-Served)**:* O FCFS é uma política simples e não preemptiva, na qual os processos são executados na ordem em que chegam à fila. Isso significa que o primeiro processo a entrar é o primeiro a ser atendido, enquanto os demais aguardam na fila. Apesar de sua simplicidade, essa abordagem pode levar a tempos de espera elevados para processos menores caso haja processos longos à frente na fila, causando o chamado *convoy effect*.

```
while processos:
    processo = fila.pop()
    executar(processo)
```

Fig. 2. Pseudocódigo do Escalonador FCFS

O pseudocódigo da política FCFS é apresentado na Figura 2. O escalonador percorre os processos em sequência, executando cada um até sua conclusão antes de passar ao próximo.

*2) **Round Robin (RR)**:* A política Round Robin utiliza um *quantum*, ou intervalo de tempo fixo, para alternar entre os processos na fila. Após consumir o *quantum*, um processo é enviado de volta ao final da fila caso não tenha sido concluído, garantindo uma distribuição justa do tempo de CPU. Essa abordagem é preemptiva, o que a torna ideal para sistemas interativos, mas pode levar a uma sobrecarga de alternância de contexto (*context switch*) dependendo do tamanho do *quantum*.

```
while processos:
    processo = fila.pop()
    executar(processo, quantum)
    if not processo.concluido():
        fila.append(processo)
```

Fig. 3. Pseudocódigo do Escalonador Round Robin

O pseudocódigo da política Round Robin é mostrado na Figura 3. O escalonador percorre a fila, alternando entre os processos enquanto verifica sua conclusão.

*3) **Prioridade**:* A política Prioridade classifica os processos com base em um nível de prioridade atribuído. Processos com maior prioridade são executados primeiro, enquanto processos de menor prioridade podem ser preemptados para dar lugar a tarefas mais urgentes. Essa abordagem é útil para sistemas que lidam com tarefas críticas, mas pode levar ao *starvation* de processos de baixa prioridade se não houver balanceamento.

```
while processos:
    processo = obter_maior_prioridade(fila)
    executar(processo)
    if not processo.concluido():
        atualizar_prioridade(processo)
        fila.append(processo)
```

Fig. 4. Pseudocódigo do Escalonador por Prioridade

Na Figura 4, apresentamos o pseudocódigo da política de escalonamento por prioridade. Após cada execução, as prioridades podem ser ajustadas para evitar o *starvation*.

4) Comparação das Políticas: Cada política apresenta vantagens e desvantagens dependendo do cenário. Enquanto o FCFS é simples e eficiente em ambientes com cargas homogêneas, o Round Robin é mais equilibrado para sistemas interativos, e o escalonamento por prioridade é ideal para sistemas críticos. No entanto, políticas preemptivas, como Round Robin e Prioridade, introduzem maior complexidade de implementação e sobrecarga de alternância de contexto.

D. Gerênciamento de Cache e Escalonamento Baseado em Similaridade

Nesta seção, detalhamos a implementação do gerenciamento de cache no simulador, explicamos a hierarquia de memória utilizada e descrevemos a técnica de escalonamento baseada em similaridade.

1) Hierarquia de Memória: O sistema segue uma hierarquia de memória composta por três camadas principais:

- **Cache:** Armazena temporariamente instruções e dados frequentemente acessados, reduzindo a necessidade de leituras na RAM.
- **Memória Principal (RAM):** Contém os processos ativos, permitindo acesso mais rápido que a memória secundária.
- **Memória Virtual:** Simulada pelo sistema para armazenar processos que não cabem na RAM, gerando maior latência nas execuções.

A cache é implementada como uma estrutura de mapeamento rápido, onde os endereços de memória são associados aos valores armazenados. Quando um valor não está presente na cache, ele precisa ser carregado da RAM, aumentando o tempo de acesso.

2) Política de Cache: FIFO (First-In, First-Out): A política de cache utilizada é FIFO (*First-In, First-Out*), onde os dados mais antigos são removidos primeiro quando a cache atinge sua capacidade máxima. O algoritmo de esvaziamento segue a seguinte lógica:

```
while cache.cheia():
    dado = cache.remover_primeiro()
    ram.armazenar(dado)

cache.adicionar(novo_dado)
```

Fig. 5. Pseudocódigo da Política de Cache FIFO

O esvaziamento ocorre sempre que a cache atinge sua capacidade máxima, garantindo que novos valores possam ser armazenados sem perda de desempenho.

3) Técnica de Escalonamento Baseada em Similaridade: A técnica de escalonamento baseada em similaridade prioriza processos que compartilham conjuntos semelhantes de instruções, reduzindo a necessidade de recomputação e acesso à memória principal. O algoritmo segue a seguinte lógica:

```
while processos:
    processo = obter_proximo(fila)
    if cache.contem_instrucoes(processo):
        priorizar(processo)
    executar(processo)
```

Fig. 6. Pseudocódigo do Escalonamento Baseado em Similaridade

A similaridade entre processos é determinada por comparações entre as instruções já processadas e as instruções do novo processo a ser executado. Quando um processo semelhante a outro já finalizado é identificado, ele pode ser priorizado para aproveitar os dados ainda presentes na cache, reduzindo acessos desnecessários à RAM.

4) Viabilidade da Integração da Política ao Escalonador: A integração do gerenciamento de cache ao escalonador mostrou-se promissora. Processos que compartilham conjuntos comuns de instruções foram executados de forma mais eficiente, pois reduziram a necessidade de acessos à memória principal. Esse comportamento sugere que, em sistemas maiores, políticas que levam em consideração a reutilização de dados podem gerar ganhos significativos de desempenho.

Em experimentos preliminares, observou-se que processos organizados por similaridade aproveitaram melhor os dados já armazenados na cache.

E. Gerênciamento de Memória e PCB

O gerenciamento de memória é um dos principais desafios em sistemas operacionais modernos e impacta diretamente o desempenho de arquiteturas multicore. Esse gerenciamento envolve técnicas como paginação, segmentação, memória virtual e a utilização de tabelas de páginas para controlar o acesso a endereços físicos. Em sistemas reais, a memória é dividida em diferentes camadas hierárquicas (cache, RAM, memória secundária), permitindo um acesso mais eficiente aos dados.

O simulador desenvolvido busca reproduzir esses conceitos de maneira simplificada, implementando um modelo de memória virtual que permite a execução de processos em um espaço de endereçamento isolado. Além disso, a implementação do PCB (Process Control Block) foi modificada para armazenar informações essenciais do gerenciamento de memória, como tabela de páginas, endereço base e limite, garantindo que cada processo tenha seu próprio espaço protegido dentro da RAM.

1) Visão Geral do Gerenciamento de Memória em Sistemas Reais: Em sistemas reais, a memória é gerenciada de forma hierárquica para otimizar o desempenho:

- **Memória cache:** Armazena temporariamente dados frequentemente acessados para reduzir o tempo de busca na RAM.
- **Memória RAM:** Contém os processos em execução e seus respectivos dados.
- **Memória virtual:** Simula uma RAM maior utilizando espaço no disco rígido para armazenar páginas de processos que não cabem na memória física.

- **Tabela de páginas:** Mapeia endereços lógicos em endereços físicos para permitir a execução de processos em um espaço virtual contínuo.
- **MMU (Memory Management Unit):** Responsável pela conversão de endereços lógicos para físicos e pelo controle de permissões de acesso.

Para garantir a eficiência da memória virtual, sistemas reais utilizam um mecanismo de paginação, onde os endereços de memória são organizados em blocos fixos chamados páginas. Quando um processo acessa uma página que não está na RAM, ocorre uma falha de página, exigindo que o sistema operacional busque essa página no disco antes de continuar a execução.

2) Modelo de Gerenciamento de Memória no Simulador: No simulador, foi implementado um modelo simplificado de gerenciamento de memória baseado em paginação. Cada processo tem sua própria tabela de páginas, que armazena o mapeamento entre os endereços lógicos e os quadros físicos na RAM. Caso um processo tente acessar um endereço que não está presente na RAM, uma falha de página é gerada, e o sistema carrega a página correspondente da memória virtual.

As principais funcionalidades implementadas no simulador incluem:

- **Endereço base e limite no PCB:** Cada processo possui um intervalo de endereços que define sua região válida de memória.
- **Mapeamento de páginas:** Foi criada uma tabela de páginas para cada processo, permitindo a conversão de endereços lógicos em físicos.
- **Gerenciamento de falhas de página:** Se um processo tentar acessar uma página que não está na RAM, o sistema carrega a página necessária a partir da memória virtual.
- **Memória cache:** Armazena temporariamente instruções frequentemente utilizadas, reduzindo o tempo de acesso.

3) Tabela de Páginas e TLB: Nos sistemas reais, a tradução de endereços pode gerar um impacto significativo no desempenho, pois cada acesso à RAM requer a consulta da tabela de páginas. Para mitigar esse problema, muitos sistemas utilizam um *Translation Lookaside Buffer* (TLB), que armazena as traduções mais recentes, reduzindo o tempo necessário para buscar um endereço.

O simulador também implementa um modelo simplificado de TLB para otimizar a conversão de endereços:

```
def traduzir_com_TLB(endereco_logico):
    if endereco_logico in TLB:
        return TLB[endereco_logico]  # Cache hit

    pagina_logica = endereco_logico // TAM_PAG
    deslocamento  = endereco_logico % TAM_PAG

    if pagina_logica in tabela_paginas:
        pagina_fisica = tabela_paginas[pagina_logica]
        endereco_fisico = (pagina_fisica * TAM_PAG)
                          + deslocamento
        TLB[endereco_logico] = endereco_fisico
        return endereco_fisico
    else:
        tratar_falha_de_pagina()
```

A implementação da TLB no simulador demonstrou uma redução no tempo médio de acesso à memória, otimizando a execução dos processos.

4) Estrutura do PCB e Controle de Processos: Cada processo é representado no simulador por meio de uma estrutura de bloco de controle de processo (PCB), que armazena informações essenciais sobre sua execução. O PCB contém dados fundamentais que permitem ao escalonador tomar decisões sobre a ordem e alocação dos processos. Entre essas informações, estão:

- Identificador único do processo (PID), que diferencia cada tarefa em execução.
- Estado do processo (novo, pronto, executando, bloqueado ou finalizado), permitindo que o escalonador saiba se ele pode ser executado.
- Endereço base na RAM, indicando onde as instruções do processo estão armazenadas.
- Registradores associados ao processo, preservando os valores durante trocas de contexto.
- Tempo total estimado de execução e tempo restante, essenciais para escalonadores como Shortest Job First (SJF).
- Prioridade atribuída, usada no escalonamento baseado em prioridades.
- Tabela de páginas, garantindo que o processo acesse corretamente sua memória alocada.

O escalonador interage diretamente com o PCB para gerenciar a execução dos processos e a alocação de memória. Durante a troca de contexto, o estado do processo atual é salvo no PCB antes da mudança para um novo processo.

5) Troca de Contexto e Impacto no Desempenho: A troca de contexto ocorre sempre que o escalonador altera o processo em execução, exigindo que o estado do processo atual seja salvo na memória antes da ativação do próximo processo. Esse procedimento consome tempo de CPU e pode impactar o desempenho geral do sistema, principalmente em políticas de escalonamento altamente preemptivas, como Round Robin, conforme descrito por Bovet e Cesati [8]. Esse procedimento envolve salvar o estado dos registradores e restaurar o estado do novo processo selecionado. Em sistemas reais, esse processo pode gerar overhead significativo, especialmente quando há um grande número de trocas.

No simulador, a troca de contexto segue o fluxo:

```
def trocar_contexto(processo_atual, novo_processo):
    salvar_estado(processo_atual)

    if novo_processo.paginas_na_RAM():
        carregar_estado(novo_processo)
    else:
        for pagina in novo_processo.paginas_necessarias:
            if not pagina_esta_na_RAM(pagina):
                carregar_pagina_na_RAM(pagina)

        carregar_estado(novo_processo)
```

Esse mecanismo garante que os dados necessários estejam disponíveis antes que um novo processo inicie sua execução, reduzindo falhas de página e otimizando o desempenho do simulador.

6) Impacto do Gerenciamento de Memória no Escalonador: O gerenciamento de memória influencia diretamente o escalonador, pois processos com alto consumo de memória podem gerar mais falhas de página, aumentando o tempo de execução. No simulador, foi observado que:

- Processos com alta taxa de troca de contexto sofreram mais penalizações, pois as páginas precisavam ser recarregadas frequentemente.
- O tempo total de execução foi impactado pelo tamanho da cache, que reduziu o número de acessos à RAM.
- O escalonador foi ajustado para considerar a similaridade entre processos, permitindo que processos com instruções comuns fossem executados em sequência para melhor aproveitamento da cache.

F. Fluxo de Execução

1) **Inicialização do Sistema**: O simulador inicializa a CPU, os núcleos, o escalonador e a memória RAM. Durante esta fase, as configurações iniciais, como a política de escalonamento, são definidas.
2) **Carregamento das Instruções**: As instruções são lidas de arquivos de texto e armazenadas na memória RAM. Este processo garante que todas as instruções necessárias para os processos estejam disponíveis.
3) **Escalonamento de Processos**: O escalonador seleciona o próximo processo com base na política configurada (FCFS, RR ou Prioridade). Ele também organiza as filas de execução de acordo com as prioridades definidas.
4) **Execução no Pipeline**: As instruções do processo selecionado são enviadas para o pipeline, onde passam pelos cinco estágios:
   - **Instruction Fetch (IF)**: A instrução é buscada na memória RAM ou na cache, dependendo de sua localização.
   - **Instruction Decode (ID)**: A instrução é decodificada, identificando os registradores e a operação a ser realizada.
   - **Execute (EX)**: A operação é executada utilizando a Unidade Lógica e Aritmética (ULA).
   - **Memory Access (MEM)**: Dados são lidos ou escritos na memória, dependendo da instrução.
   - **Write Back (WB)**: O resultado da instrução é armazenado nos registradores do núcleo.
5) **Gerenciamento de Memória**: Durante a execução, a memória cache é utilizada para acelerar o acesso a dados frequentemente usados. Quando a cache está cheia, dados menos recentes são transferidos de volta para a RAM.
6) **Finalização do Processo**: Após a execução de todas as instruções de um processo, ele é marcado como concluído, e o escalonador seleciona o próximo processo.
7) **Encerramento da Simulação**: A simulação termina quando todos os processos e instruções são executados. Estatísticas, como tempo total de execução e eficiência do pipeline, são exibidas.

G. Arquivos de Instruções

Os arquivos de instrução utilizados no simulador seguem um formato simples, contendo uma instrução por linha. Na Tabela 1 podemos ver quais são as possiveis implementações de intruções que o simulador reconhece:

TABLE I
INSTRUÇÕES SUPORTADAS NO SIMULADOR

Instrução	Descrição
`LOAD RX Y`	Carrega o valor do endereço *Y* no registrador *RX*.
`STORE RX Y`	Armazena o valor do registrador *RX* no endereço *Y*.
`ADD RX RY RZ`	Soma os valores de *RY* e *RZ*, armazenando o resultado em *RX*.
`SUB RX RY RZ`	Subtrai o valor de *RZ* de *RY*, armazenando o resultado em *RX*.
`MULT RX RY RZ`	Multiplica *RY* por *RZ*, armazenando o resultado em *RX*.
`DIV RX RY RZ`	Divide *RY* por *RZ*, armazenando o resultado em *RX*.
`IF RX RY OP RZ`	Realiza a comparação *OP* (`>`, `<`, `==`) entre *RY* e *RZ*. Armazena `1` em *RX* se a condição for verdadeira, `0` caso contrário.

a) Exemplo de um Arquivo de Instruções:

```
LOAD R1 10
LOAD R2 20
ADD R3 R1 R2
IF R4 R1 < R2
STORE R3 30
```

Neste exemplo: - `LOAD R1 10` carrega o valor do endereço 10 no registrador `R1`. - `ADD R3 R1 R2` soma os valores de `R1` e `R2` e armazena o resultado em `R3`. - `IF R4 R1 < R2` compara `R1` com `R2`. Se `R1` for menor que `R2`, armazena `1` em `R4`, caso contrário, `0`.

b) Ajustes na RAM: Para aumentar o número de arquivos de instruções processados, modifique a variável `TAM_INSTRUCTIONS` no código da RAM. Isso impactará diretamente a capacidade do sistema de carregar mais ou menos arquivos de instruções e armazenar novos conjuntos de instruções para execução, porém essa variável deve ser declarada de acordo com a quantidade máxima de arquivos encontrados na pasta *instructions*.

III. RESULTADOS E DISCUSSÃO

A. Arquitetura Multicore

A implementação de uma arquitetura multicore demonstrou ser altamente eficaz na redução do tempo de execução das tarefas. Como mostrado na Tabela II e na Figura 7, há uma redução significativa no tempo médio de execução à medida que o número de threads aumenta. Essa redução é atribuída à capacidade dos núcleos de processar múltiplas instruções

simultaneamente, aproveitando o paralelismo inerente aos sistemas multicore.

TABLE II
TEMPOS MÉDIOS DE EXECUÇÃO PARA DIFERENTES THREADS

Número de Threads	Tempo Médio (ms)
1	5903.8
2	2635.1
3	1749.1
4	1561.8
5	1361.4

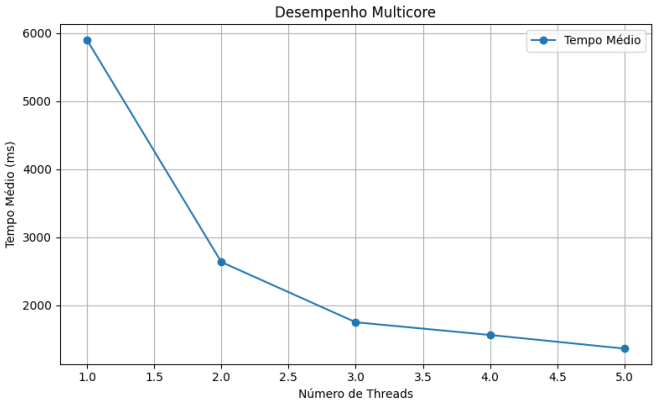

Fig. 7. Impacto do número de threads no tempo de execução.

a) Análise Crítica: Observa-se que a redução do tempo de execução não é linear à medida que mais threads são adicionadas. Isso ocorre devido a fatores como *overhead* na sincronização entre *threads* e limitações no acesso à memória compartilhada. No entanto, o ganho de desempenho para até cinco *threads* é evidente, demonstrando que a arquitetura multicore implementada é eficiente para o número de núcleos considerado. Esses resultados corroboram estudos como o de Patterson e Hennessy [1], que destacam a importância do paralelismo em sistemas modernos, especialmente em cenários de alta demanda computacional.

B. Escalonamento

O escalonador desempenha um papel central na eficiência do simulador, determinando a ordem e o tempo de execução dos processos. Os resultados obtidos, apresentados na Tabela III e na Figura 8, destacam as diferenças entre as políticas implementadas: FCFS, Round Robin e Prioridade.

TABLE III
TEMPOS MÉDIOS DE EXECUÇÃO PARA CADA POLÍTICA DE ESCALONAMENTO

Política	Tempo Médio (ms)
FCFS	523.3
Round Robin	1361.4
Prioridade	1459.1

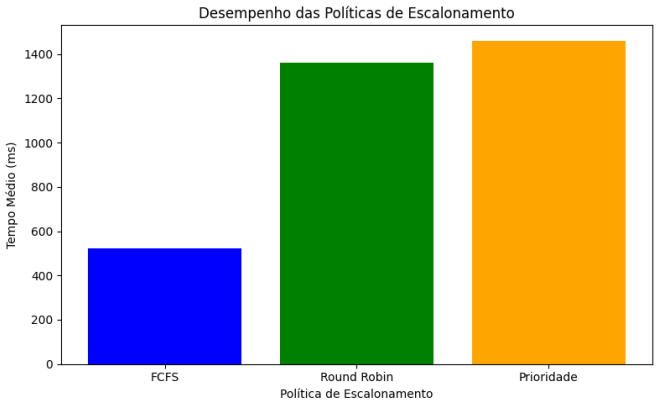

Fig. 8. Desempenho das políticas de escalonamento

a) Análise Crítica: Os tempos médios obtidos indicam que fatores além da política de escalonamento influenciam diretamente o desempenho do sistema.

- **Impacto da Organização dos Jobs:** A ordem inicial dos processos pode afetar significativamente o tempo total de execução. No FCFS, processos longos inseridos primeiro podem causar *convoy effect*, bloqueando processos menores. No Round Robin, a alternância evita esse problema, mas aumenta o overhead de trocas de contexto. No escalonamento por prioridade, a forma como os processos são classificados pode impactar a eficiência da execução.
- **Tempo de Obtenção da Regra da Política:** Um fator relevante a ser analisado é o tempo necessário para obter a regra de escalonamento antes da execução dos processos. Se esse tempo for significativo, ele pode mascarar o impacto real da política escolhida. Nos experimentos, foi observado que, ao desconsiderar esse tempo, o tempo total de execução sofreu uma leve queda. Entretanto, essa redução se torna mais evidente em cenários com maior volume de processos, onde a sobrecarga inicial pode se diluir no tempo total de execução.
- **Crescimento do Tempo de Execução com o Aumento de Processos:** À medida que a carga de processos aumenta, o tempo total de execução cresce. No entanto, a análise mostra que esse crescimento não é completamente linear. A interação entre a política de escalonamento, a troca de contexto e o acesso à memória pode gerar variações inesperadas no tempo de resposta. Isso indica que, para cenários maiores, a eficiência do escalonador depende não apenas da política adotada, mas também da estrutura do código e da organização dos jobs.

b) Implicações: Cada política de escalonamento possui um cenário ideal de aplicação. A escolha da política depende do tipo de sistema a ser simulado e das prioridades estabelecidas. O FCFS é eficiente em cenários com cargas previsíveis, enquanto Round Robin e Prioridade são mais adequados para sistemas dinâmicos e interativos. Além disso, o tempo de obtenção da política e a organização inicial dos processos

podem ter um impacto significativo no tempo de execução e devem ser considerados em análises futuras.

C. Gerenciamento de Cache e Escalonamento Baseado em Similaridade

Nesta seção, apresentamos a análise experimental da implementação da política de cache FIFO e sua influência na eficiência do simulador. Para isso, foram realizados testes variando o tamanho da cache e o número de arquivos processados, comparando o tempo médio de execução de três políticas de escalonamento: FCFS, Round Robin e Prioridade.

1) Impacto do Tamanho da Cache: O primeiro experimento analisou o impacto do tamanho da cache no tempo médio de execução. Os resultados são apresentados na Tabela IV e ilustrados na Figura 9.

TABLE IV
TEMPO MÉDIO DE EXECUÇÃO (MS) PARA DIFERENTES TAMANHOS DE CACHE

Política	Cache 5	Cache 16
FCFS	403.8	404.0
Round Robin	906.6	905.6
Prioridade	1109.2	1106.4

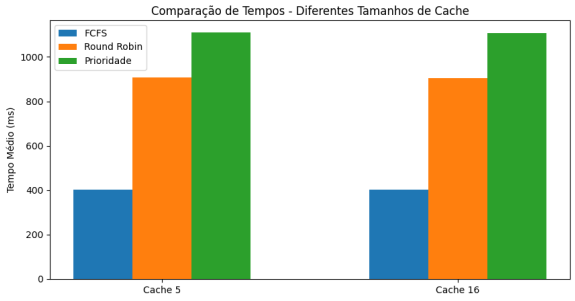

Fig. 9. Grafico: impacto no numero de cache em relação ao tempo

Comparando os tempos médios de execução das políticas de escalonamento com e sem cache na Figura 8, observa-se que a política FCFS obteve um ganho de aproximadamente 20% na redução do tempo total de execução. Esse ganho se deve à reutilização das instruções armazenadas, evitando acessos repetidos à memória RAM. No entanto, para políticas como Round Robin (RR), o benefício foi menos expressivo devido à alta taxa de preempção, que força trocas constantes de contexto. Avaliamos diferentes tamanhos de cache e observamos que, mesmo aumentando sua capacidade, os ganhos não foram significativos. Isso sugere que a política FIFO pode estar descartando instruções relevantes. Para confirmar essa hipótese, seria necessário medir a taxa de descarte de instruções na cache e verificar se a substituição de dados está impactando o desempenho. Esse comportamento pode indicar que uma política mais avançada, como Least Recently Used (LRU), poderia melhorar a reutilização de instruções e reduzir acessos desnecessários à RAM.

2) Impacto do Número de Arquivos Processados: Nos experimentos, variamos a quantidade de arquivos processados de 5 a 10. Esses arquivos não apenas diferem em quantidade, mas também em seu conteúdo, contendo diferentes conjuntos de instruções. Isso permite analisar não apenas o impacto do número de arquivos, mas também como a diversidade das instruções afeta o uso da cache. A Tabela VII e a Figura 10 mostram os tempos médios obtidos.

TABLE V
TEMPO MÉDIO DE EXECUÇÃO (MS) PARA DIFERENTES QUANTIDADES DE ARQUIVOS

Política	5 arquivos	8 arquivos	10 arquivos
FCFS	404.0	505.0	607.0
Round Robin	1005.0	1409.0	1819.0
Prioridade	1549.0	1609.0	1909.0

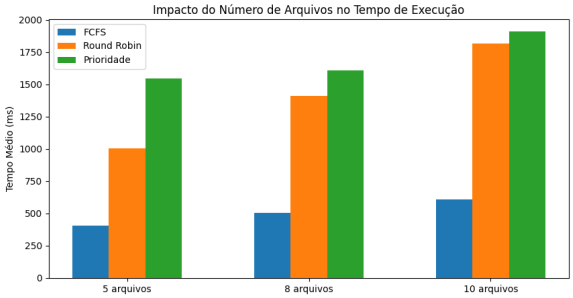

Fig. 10. Grafico: impacto no numero de arquivos em relação ao tempo

Os resultados apresentados na Figura VII mostram um comportamento interessante entre as políticas de escalonamento. A política Prioridade apresentou pouca variação no tempo médio de execução para poucos arquivos (5 ou 8), sugerindo que, nesse cenário, os processos de maior prioridade foram rapidamente atendidos, minimizando a influência do aumento da carga de trabalho. Em contrapartida, a política Round Robin sofreu um impacto maior, pois a alta taxa de preempção fez com que o tempo total de execução aumentasse consideravelmente. Já a política FCFS manteve um crescimento linear, indicando que a simplicidade na organização da fila de jobs é um fator determinante para sua estabilidade. Como essa política processa os jobs na ordem de chegada sem trocas de contexto desnecessárias, seu desempenho se mantém previsível e menos suscetível a variações no número de arquivos processados.

3) Implicações do Uso de Cache e Similaridade no Escalonador: A partir dos resultados experimentais, é possível destacar algumas implicações importantes da integração entre o gerenciamento de cache e o escalonador:

- O simples aumento do tamanho da cache não garante redução significativa no tempo de execução. A política FIFO, utilizada no gerenciamento da cache, mostrou-se limitada na retenção de instruções úteis, evidenciando que a substituição dos dados deve ser mais criteriosa.

- A reutilização de instruções pode otimizar o tempo de execução, mas sua eficiência depende da organização dos jobs. No escalonador FCFS, a similaridade entre processos não teve impacto significativo, enquanto no escalonador por Prioridade houve um melhor aproveitamento da cache.
- A política de escalonamento influencia diretamente a eficiência da cache. Processos com preempção frequente, como no Round Robin, tiveram uma taxa de falha de cache maior, pois as constantes trocas de contexto resultaram em mais acessos à memória principal.
- A integração entre cache e escalonador exige um modelo adaptativo. Para que o simulador aproveite ao máximo a reutilização de dados armazenados na cache, o escalonador deve ser capaz de priorizar processos que compartilham instruções.

Essas observações reforçam que a cache e o escalonador não devem ser tratados de forma independente, mas sim como componentes interligados que precisam ser otimizados em conjunto.

4) Discussão sobre a Viabilidade da Integração ao Escalonador: Os experimentos demonstraram que a simples implementação da política FIFO na cache não foi suficiente para reduzir os tempos de execução. O principal fator que limita a efetividade da cache é a falta de priorização de dados reutilizados, o que foi evidenciado pelos seguintes pontos:

- FIFO descarta os dados mais antigos, sem levar em conta sua frequência de uso, tornando o armazenamento de instruções potencialmente úteis ineficiente.
- A influência da cache variou conforme o escalonador. No FCFS, o impacto foi limitado, enquanto no Round Robin a alta taxa de troca de contexto resultou em acessos mais frequentes à memória principal.
- Processos similares poderiam se beneficiar de uma melhor organização da fila de execução para maximizar a reutilização da cache, algo que a FIFO não permitiu.

A viabilidade da integração entre cache e escalonador depende da implementação de um modelo dinâmico, onde o escalonador possa identificar padrões de reutilização de dados e priorizar processos que se beneficiariam da permanência dessas informações na cache. Alternativas como políticas de substituição Least **Recently Used (LRU)** podem ser exploradas para melhorar a retenção de dados úteis.

D. Gerênciamento de Memória e PCB

Nesta seção, apresentamos os resultados obtidos com a implementação do escalonador **Shortest Job First (SJF)** e comparamos seu desempenho com os escalonadores anteriores: FCFS (First-Come, First-Served), Round Robin e Prioridade. O objetivo desta análise é compreender o impacto da ordenação dos processos pelo tempo total de execução, avaliar a eficiência do novo método em relação aos anteriores e investigar a influência do gerenciamento de memória no tempo médio de execução.

1) Descrição Geral dos Resultados: Os experimentos foram conduzidos utilizando diferentes quantidades de arquivos de instruções (5, 8 e 10). Para cada cenário, o tempo médio de execução foi medido para cada escalonador, conforme mostrado na Tabela VI. A Figura 11 exibe a comparação gráfica entre todas as políticas testadas.

TABLE VI
TEMPOS MÉDIOS DE EXECUÇÃO (MS) PARA O ESCALONADOR SJF

Número de Arquivos	Tempo Médio(ms)
5	1002
8	1403
10	1804

2) Análise Comparativa dos Escalonadores: Para avaliar a eficiência do escalonador SJF, comparamos seu tempo médio de execução com os outros escalonadores testados. Os resultados estão apresentados na Tabela VII.

TABLE VII
TEMPO MÉDIO DE EXECUÇÃO (MS) PARA DIFERENTES QUANTIDADES DE ARQUIVOS

Política	5 arquivos	8 arquivos	10 arquivos
FCFS	404.0	505.0	607.0
Round Robin	1005.0	1409.0	1819.0
Prioridade	1549.0	1609.0	1909.0
SJF	1002.0	1403.0	1804.0

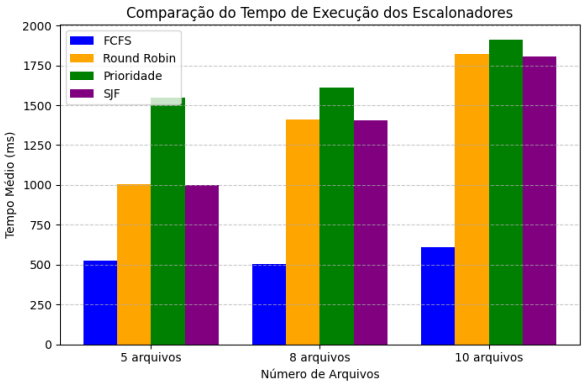

Fig. 11. Comparação do Tempo de Execução dos Escalonadores

A análise dos tempos médios de execução revela que o escalonador SJF apresentou um desempenho mais eficiente do que Round Robin e Prioridade, mantendo uma taxa de crescimento mais controlada à medida que a carga de trabalho aumentou.

Diferentemente do escalonador Round Robin, que sofre com um número elevado de trocas de contexto, o SJF minimiza esse problema ao ordenar os processos pelo tempo total de execução. Esse comportamento evita que processos curtos fiquem presos na fila atrás de processos mais longos, como pode ocorrer no FCFS. Assim, o SJF oferece um equilíbrio

interessante entre o tempo médio de resposta e a utilização eficiente dos recursos do sistema.

Observando a Figura 11, nota-se que o SJF manteve um tempo de execução próximo ao Round Robin, mas sua vantagem se tornou mais evidente à medida que o número de arquivos aumentou. O impacto positivo dessa abordagem foi mais perceptível quando a carga de processos curtos era alta, pois o SJF rapidamente liberava a CPU para novas tarefas, reduzindo a espera de processos menores.

Entretanto, uma limitação do SJF é sua natureza não preemptiva, o que significa que um processo longo pode atrasar a execução dos demais se já estiver em andamento. Esse efeito pode ser observado no crescimento do tempo de execução com 10 arquivos, onde o desempenho do SJF ficou próximo ao dos outros escalonadores. Esse comportamento indica que, em cenários com processos de durações muito distintas, o escalonador poderia ser aprimorado com uma abordagem preemptiva.

Além disso, a relação entre escalonamento e gerenciamento de memória desempenha um papel crucial na eficiência dos tempos de execução. No caso do SJF, a ordenação por tempo de execução pode ter ajudado a reduzir a frequência de acessos à RAM e minimizar falhas de cache, resultando em uma melhor performance geral.

Por fim, os resultados confirmam que a escolha da política de escalonamento deve ser feita considerando o perfil das cargas de trabalho. O SJF se mostrou uma alternativa eficiente em cenários onde a execução de processos curtos é prioritária, mas pode não ser a melhor opção em casos onde há uma grande variação nos tempos de execução dos processos.

3) Discussão sobre Eficiência e Impacto na Execução: A eficiência do novo escalonador foi avaliada sob diferentes aspectos, levando em consideração o impacto do ordenamento dos processos, a preempção e o uso de memória cache:

a) Impacto da Ordenação dos Processos: O escalonador SJF ordena os processos com base no tempo total de execução, priorizando aqueles que possuem menor tempo estimado. Essa estratégia mostrou-se eficiente para evitar que processos curtos fiquem bloqueados por processos mais longos, um problema comum no FCFS. No entanto, sua eficácia depende diretamente da precisão da estimativa do tempo de execução de cada processo.

Ao analisar a Tabela VII, observamos que o tempo de execução do SJF cresce de maneira mais controlada em comparação com Round Robin e Prioridade. Isso indica que a organização dos processos em ordem crescente de tempo de execução reduz o impacto de bloqueios causados por processos longos, otimizando a utilização da CPU.

Além disso, o Gráfico 11 reforça essa observação ao demonstrar que o tempo total do SJF mantém uma evolução previsível conforme o número de processos aumenta. Diferentemente do FCFS, que pode apresentar variações devido ao efeito de comboio, o SJF permite que processos menores sejam concluídos rapidamente, garantindo um fluxo mais equilibrado de execução.

Essa abordagem beneficia especialmente sistemas onde há uma grande variação no tempo de execução dos processos, permitindo que tarefas menores sejam finalizadas sem sofrer impacto significativo de processos mais longos. No entanto, vale destacar que, apesar da eficiência geral, o SJF não é preemptivo, o que pode afetar sua capacidade de resposta em cenários onde processos longos são introduzidos dinamicamente no sistema.

b) Impacto do Uso de Memória Cache e Acessos à RAM: O tempo de execução também é influenciado pela forma como os processos acessam a memória. Processos que compartilham instruções podem se beneficiar de um melhor aproveitamento da cache, reduzindo o tempo de acesso à RAM. No SJF, observou-se que processos menores tendem a ser concluídos antes, permitindo que instruções reutilizadas permaneçam na cache por mais tempo, o que pode melhorar a eficiência geral do simulador.

Esse comportamento pode ser analisado na Tabela VII, onde se observa que o tempo médio de execução do SJF cresce de maneira mais controlada do que os escalonadores Round Robin e Prioridade à medida que o número de arquivos aumenta. Esse resultado indica que a organização dos processos por tempo de execução pode influenciar positivamente o uso da memória cache, minimizando a necessidade de buscas repetitivas na RAM. Como consequência, a reutilização eficiente de instruções armazenadas contribui para a estabilidade e previsibilidade dos tempos de resposta no simulador.

E. Discussões Gerais

Os resultados apresentados destacam a eficácia das modificações implementadas no simulador, tanto na adoção de uma arquitetura multicore quanto nas políticas de escalonamento. A arquitetura multicore foi fundamental para reduzir o tempo de execução, enquanto o escalonador proporcionou flexibilidade no gerenciamento de processos.

Os dados reforçam a importância do balanceamento entre complexidade e eficiência. Políticas mais complexas, como Round Robin e Prioridade, introduzem *overheads*, mas garantem interatividade e atendimento de prioridades, tornando-se mais adequadas para sistemas modernos. Já políticas simples, como FCFS, têm vantagens em cenários específicos, mas podem não atender a requisitos mais complexos. O escalonador SJF se posiciona como uma abordagem intermediária, combinando a simplicidade do FCFS com a eficiência da organização baseada no tempo estimado de execução.

Além das melhorias no escalonador, a implementação da política de cache revelou aspectos importantes do comportamento do simulador. Embora o uso de cache seja amplamente reconhecido como um fator de otimização em arquiteturas modernas, os experimentos indicaram que o simples aumento do tamanho da cache não foi suficiente para reduzir significativamente os tempos de execução. Isso ocorreu porque a política FIFO, adotada na gestão da cache, não favoreceu a reutilização eficiente de instruções.

A introdução do escalonador SJF demonstrou impactos significativos na execução dos processos. Como observado na

Tabela VII e na Figura 11, o SJF apresentou tempos médios de execução menores do que as políticas de Round Robin e Prioridade, evidenciando sua eficácia na minimização de tempos ociosos causados por processos longos. A ordenação dos processos pela duração permitiu uma distribuição mais eficiente dos recursos da CPU, garantindo que processos curtos fossem finalizados rapidamente antes da execução de processos mais longos.

A análise detalhada dos tempos médios de execução mostrou que a relação entre o escalonador e o gerenciamento de cache pode ser um fator determinante na otimização do sistema. Embora a presença de jobs similares tenha demonstrado um potencial de redução no tempo total de execução, a política de substituição FIFO não foi eficiente o suficiente para maximizar esse benefício. A principal explicação para esse comportamento está no fato de que FIFO apenas descarta os dados mais antigos, sem considerar a frequência de acesso ou a importância das instruções armazenadas.

A interação entre escalonador e cache pode ser observada no impacto das políticas sobre o crescimento do tempo de execução. Em particular:

- No **FCFS**, a influência da cache foi limitada, pois os processos são executados sequencialmente, sem priorização para reaproveitamento de dados.
- No **Round Robin**,o tempo de execução cresceu mais acentuadamente com o aumento do número de arquivos, como evidenciado pelos valores apresentados na Tabela III. Esse crescimento ocorre devido ao alto número de trocas de contexto, conforme ilustrado na Figura 8, onde o tempo médio de execução foi significativamente maior em comparação a outras abordagens.
- No **escalonador por Prioridade**, houve um certo ganho de desempenho em relação ao Round Robin, pois processos com maior prioridade foram finalizados mais rapidamente, aumentando a chance de reutilização dos dados em cache.
- No **SJF**, a ordenação dos processos resultou em um melhor aproveitamento da cache e um crescimento mais controlado do tempo de execução, evitando o congestionamento causado por processos longos.

Diferente do Round Robin, que sofre com um número excessivo de trocas de contexto, o SJF minimiza esse problema ao ordenar os processos pelo tempo de execução. Isso evita que processos curtos fiquem presos na fila atrás de processos longos, como pode ocorrer no FCFS, tornando sua abordagem mais equilibrada. Além disso, observou-se que processos menores tendem a ser concluídos antes, permitindo que instruções reutilizadas permaneçam na cache por mais tempo, o que pode melhorar a eficiência geral do simulador.

A principal limitação do SJF está no fato de não ser preemptivo, o que pode gerar dificuldades caso processos longos entrem no sistema quando a CPU já estiver ocupada. Em cenários dinâmicos, onde novos processos são adicionados continuamente, esse comportamento pode levar a atrasos na execução de processos menores. No entanto, para cargas de trabalho estáticas, o SJF mostrou-se eficiente na redução do tempo médio de execução.

Por fim, os resultados reforçam a importância de considerar o contexto de execução ao definir políticas de escalonamento e gerenciamento de memória. O desempenho do simulador não depende exclusivamente do escalonador ou da cache, mas da interação eficiente entre ambos. Esse equilíbrio pode ser um diferencial significativo para arquiteturas computacionais que buscam otimizar tempos de resposta e eficiência do processamento. A implementação de políticas mais avançadas de substituição de cache, como *Least Recently Used* (LRU), pode potencialmente melhorar ainda mais os ganhos de desempenho observados no SJF.

IV. Conclusão

O desenvolvimento do simulador de arquitetura multicore com escalonamento e gerenciamento de memória permitiu uma análise detalhada do impacto dessas técnicas no desempenho de um sistema computacional. A implementação de um pipeline MIPS e a modelagem de diferentes políticas de escalonamento demonstraram como a organização dos processos e a utilização eficiente dos recursos influenciam diretamente o tempo de execução.

Os experimentos mostraram que o uso da arquitetura multicore proporcionou uma redução significativa no tempo médio de execução, validando a importância do paralelismo na otimização do desempenho computacional. No entanto, observou-se que o ganho de eficiência não cresce linearmente com o número de núcleos devido a fatores como sincronização e acessos à memória compartilhada.

A análise das políticas de escalonamento revelou que cada abordagem possui vantagens e desvantagens dependendo do cenário de uso. O FCFS mostrou-se previsível e adequado para cargas homogêneas, enquanto o Round Robin garantiu maior interatividade, porém com um custo maior em trocas de contexto. O escalonador por Prioridade apresentou melhor tempo de resposta para processos críticos, mas pode gerar starvation para processos de baixa prioridade.

O escalonador Shortest Job First (SJF) destacou-se como uma solução intermediária eficiente, proporcionando tempos médios de execução menores que o Round Robin e a Prioridade. Sua estratégia de ordenação dos processos por tempo total de execução evitou que processos curtos ficassem bloqueados, permitindo uma distribuição mais equilibrada da carga de processamento. No entanto, sua natureza não preemptiva pode ser um fator limitante em cenários dinâmicos, onde processos longos podem gerar atrasos para novas execuções.

Além do escalonamento, o estudo sobre o gerenciamento de memória e cache evidenciou que o simples aumento do tamanho da cache não é suficiente para melhorar o desempenho. A política FIFO de substituição mostrou-se ineficiente para reter instruções reutilizáveis, sugerindo que abordagens mais avançadas, como Least Recently Used (LRU), poderiam gerar melhores resultados.

A interação entre escalonador e cache foi um ponto crítico na análise do simulador. Foi observado que políticas que

favorecem a execução sequencial de processos semelhantes melhoram a reutilização da cache, reduzindo acessos à RAM e otimizando o tempo total de execução. Isso reforça a necessidade de projetar sistemas de escalonamento que levem em consideração o padrão de acesso à memória para maximizar a eficiência.

Com base nesses resultados, futuras melhorias podem incluir a implementação de uma versão preemptiva do SJF para mitigar o impacto de processos longos, bem como a adoção de algoritmos mais eficientes para o gerenciamento da cache. Além disso, testar políticas adaptativas de escalonamento, que ajustem suas regras dinamicamente conforme a carga de trabalho, pode ser uma abordagem promissora para aumentar a flexibilidade do sistema.

Por fim, este simulador provou ser uma ferramenta valiosa para a compreensão prática dos conceitos de arquitetura multicore, escalonamento e gerenciamento de memória. Os experimentos realizados não apenas confirmaram teorias estabelecidas na literatura, mas também destacaram desafios reais enfrentados no desenvolvimento de sistemas computacionais modernos.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5ª ed., Morgan Kaufmann, 2013.

[2] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 10ª ed., Pearson, 2015.

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4ª ed., Pearson, 2014.

[4] J. L. Hennessy e D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6ª ed., Morgan Kaufmann, 2017.

[5] M. McCool, A. Reinders e J. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1ª ed., Elsevier, 2012.

[6] J. Smith e R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, 1ª ed., Morgan Kaufmann, 2005.

[7] G. G. Lee, *Computer System and Network Performance Modeling*, CRC Press, 2020.

[8] D. Bovet e M. Cesati, *Understanding the Linux Kernel*, 3ª ed., O'Reilly Media, 2005.

[9] H. Peter Anvin, *Efficient Memory Management Techniques in Multithreaded Systems*, IEEE Transactions on Computers, vol. 68, no. 12, pp. 1458-1467, 2019.

[10] M. D. Hill e A. J. Smith, "Evaluating Associativity in CPU Caches," *ACM Computing Surveys*, vol. 31, no. 3, pp. 241-272, 1999.