Simulador: Arquitetura Von Neuman e Pipeline Mips

1st José Marconi de A. Júnior
Engenharia de Computação
Centro Federal de Educação Tecnológica (CEFET)
Divinópolis, MG
jmarconiadm@gmail.com

***Abstract*—The development of computer systems requires an in-depth understanding of both hardware and software. This paper presents the development of an operating system simulator that emulates a von Neumann architecture with MIPS pipeline support. The simulation includes a implemented CPU, integrating a control unit, opcodes, instructions, RAM, disk, cache, pipeline, and peripherals. The proposed system features a multicore environment where each core has its own CPU and clock, while the RAM is shared among all cores. The study evaluates the efficiency of different scheduling algorithms (FCFS, SJF, and Lottery) under various configurations, including cache optimization and Memory Management Unit (MMU) integration. Results show that the use of cache improves execution time, with the Lottery algorithm achieving the highest reduction (79.32%), albeit at the cost of increased clustering overhead. The MMU implementation, which transitioned from scheduling PCB IDs to binary-based scheduling, demonstrated slight improvements in execution time and clustering efficiency. The findings suggest that cache utilization and MMU-based scheduling can enhance performance without introducing computational costs, making them strategies for optimizing operating system.**

***Index Terms*—Arquitetura de Computadores, Sistemas Operacionais, Pipeline MIPS, Sistemas Multicore, Unidade de Gerenciamento de Memória (MMU), Otimização de Cache, Escalonamento de Processos.**

I. Introduction

O desenvolvimento de sistemas computacionais exige uma compreensão profunda tanto do hardware quanto do software que os suporta. A arquitetura von Neumann, fundamental para a computação moderna, define a base para a interação sequencial entre memória e processador, enquanto o pipeline, como implementado na arquitetura MIPS, permite a execução paralela e eficiente de instruções.

Este artigo apresenta o desenvolvimento de um simulador de sistema operacional que emula, por meio de código, uma arquitetura von Neumann com suporte a pipeline MIPS. A emulação inclui a implementação de uma CPU, abrangendo unidade de controle, opcodes, instruções e uma estrutura modular que integra RAM, Disco, Cache, pipeline e periféricos. O objetivo principal é oferecer uma plataforma para explorar, testar e compreender os princípios fundamentais dessas arquiteturas e sua relação com sistemas operacionais.

Identify applicable funding agency here. If none, delete this.

II. Fundamentação Teórica

A. Arquitetura Von Neumann

A arquitetura de Von Neumann é um modelo de computador clássico que organiza os componentes do sistema de forma a executar programas armazenados na memória (figura 1). Essa organização é a base para a maioria dos computadores modernos.

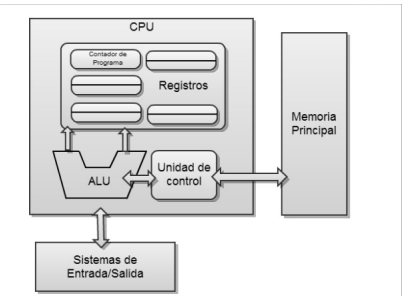

Fig. 1. Exemplo ilustrativo de uma arquitetura clássica de Von Neumann.

1) CPU: A CPU é considerado o ”cérebro” do sistema. Ela executa as instruções armazenadas na memória principal. É composta principalmente por:

- Contador de Programa (Program Counter): guarda o endereço da próxima instrução a ser executada, geralmente sendo um registrador especial do banco de registradores;
- Banco de Registradores(Registros): pequenas áreas de armazenamento temporário, usadas para armazenar dados intermediários e facilitar cálculos;
- ALU (Unidade Lógica e Aritmética): realiza operações matemáticas (soma, subtração, etc.) e lógicas (comparações como AND e OR, por exemplo);
- Unidade de Controle: coordena as operações do sistema. Ela decodifica as instruções da memória e controla o fluxo de dados entre os componentes da CPU, memória principal e dispositivos de entrada/saída.

2) Memória Principal (RAM): É onde os dados e instruções do programa são armazenados. Na arquitetura von Neumann, memória é unificada, ou seja, tanto os dados quanto as instruções do programa compartilham o mesmo espaço de armazenamento.

3) Sistemas de Entrada/Saída: Representam os dispositivos que permitem a interação com o sistema (teclado, monitor, discos, etc.). A CPU controla esses dispositivos por meio da Unidade de Controle.

4) Fluxo de Dados:

- Entre CPU e Memória Principal: as instruções e dados são buscados da memória para serem processados pela CPU. Após o processamento, os resultados podem ser armazenados de volta na memória;
- Entre CPU e Dispositivos de Entrada/Saída: A CPU envia comandos para os dispositivos e recebe dados deles.

B. Pipeline Mips

A pipeline MIPS (figura 2) é uma técnica usada em arquiteturas baseadas no conjunto de instruções MIPS (Microprocessor without Interlocked Pipeline Stages) para melhorar o desempenho do processador. Essa técnica divide a execução de uma instrução em várias etapas, permitindo que várias instruções sejam processadas simultaneamente em diferentes estágios da CPU.

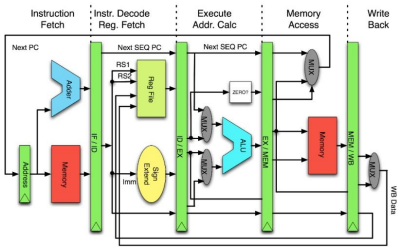

Fig. 2. Exemplo ilustrativo da Pipeline Mips e seus estágios.

1) Estágios da Pipeline:

- IF (Instruction Fetch - Busca da Instrução):
  - A instrução é buscada da memória usando o endereço armazenado no contador de programa (PC);
  - O PC é incrementado para apontar para a próxima instrução.
- D (Instruction Decode/Register Fetch - Decodificação da Instrução):
  - A instrução é decodificada;
  - Os operandos são lidos dos registradores (se necessário);
  - Determina-se o tipo de operação (aritmética, lógica, memória, etc.).
- EX (Execute - Execução):
  - A ALU realiza a operação especificada pela instrução;
  - Exemplo: Soma, subtração, ou cálculo do endereço efetivo (para instruções de memória).
- MEM (Memory Access - Acesso à Memória):
  - Instruções de leitura/escrita acessam a memória;
  - Outras instruções ignoram este estágio.
- WB (Write Back - Escrita no Registrador):
  - O resultado da operação é escrito de volta em um registrador.

A pipeline MIPS permite que múltiplas instruções sejam executadas ao mesmo tempo. Enquanto uma instrução está sendo buscada (IF), outra está sendo decodificada (ID), outra está na execução (EX), e assim por diante. Isso resulta em um aumento significativo na taxa de transferência de instruções.

III. Metodologia

O objetivo inicial foi integrar os conceitos fundamentais de ambas as arquiteturas para desenvolver um emulador com uma estrutura mais robusta. Durante o desenvolvimento, foram incorporadas modificações em cada etapa para aproximar o sistema operacional de um ambiente real, incluindo conceitos de multicore, multithreading e a implementação de políticas de escalonamento.

A. Modificações

1) Multicore: O conceito de ”multicore” refere-se a um processador que contém dois ou mais núcleos de processamento. Cada núcleo pode executar instruções de maneira independente, funcionando como se fosse um processador separado. Essa arquitetura permite a execução simultânea de múltiplas tarefas ou threads, aumentando significativamente o desempenho e a eficiência do sistema (figura 3).

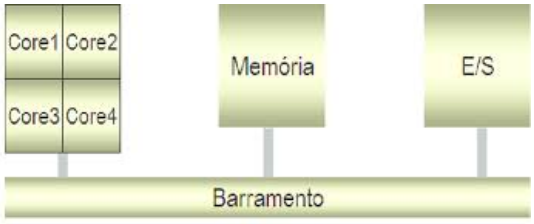

Fig. 3. Exemplo ilustrativo básico de uma estrutura multicore.

No contexto do sistema implementado, cada núcleo (Core) terá sua própria CPU, juntamente com todos os componentes necessários, e um clock exclusivo. A memória principal (RAM), por outro lado, será compartilhada entre todos os núcleos em operação.

Com essa estrutura, o objetivo é permitir que os processos concorram pelos núcleos, disputando os recursos para sua execução.

2) Multithreading: O conceito de multithreading refere-se à capacidade de um processador ou sistema operacional de executar múltiplas threads dentro de um único processo de forma concorrente. Uma thread é a menor unidade de execução de um programa, contendo seu próprio conjunto de instruções, registros e pilha de execução.

Ao adotar o multithreading, um programa pode dividir seu trabalho em várias threads, permitindo que elas sejam executadas simultaneamente em sistemas com múltiplos núcleos.

Dessa forma, como o sistema foi projetado para suportar múltiplos núcleos, é possível distribuir as threads entre os núcleos disponíveis, aproveitando de maneira mais eficiente os recursos disponíveis e aumentando o desempenho na execução de tarefas paralelizadas.

Para implementar o multithreading, foi utilizada a biblioteca ¡thread¿ do C++, que oferece as funcionalidades necessárias para essa aplicação.

3) Quantum e Clock:

- Clock: é a unidade de "tempo" utilizada pelo sistema. Cada ciclo de clock é considerado uma unidade de tempo, ou seja, cada ciclo do clock é um "Clock". Cada etapa do pipeline usa um ciclo de clock, e essa medida será utilizada para avaliar a eficiência de cada escalonador no sistema, além do "tempo" total gasto em cada Core utilizado;
- Quantum: é o número de ciclos de clock alocados para a execução de cada processo. O quantum fixo associado a cada processo determina o número máximo de ciclos de clock que um processo pode usar antes de ser bloqueado. Também existe o "quantum máximo", que é o número total de ciclos de clock usados por um processo para completar sua execução.

O quantum atribuído a cada processo é definido pelo somatório dos pesos de suas instruções, como apresentado na Tabela 1. Esses pesos representam, de forma aproximada, o número de ciclos de clock necessários para a execução de cada instrução. Embora o peso não corresponda exatamente ao total de ciclos utilizados por cada instrução, ele reflete um valor representativo, quanto maior o número de ciclos maior o peso atribuído.

TABLE I
REFERENTE AOS PESOS ASSOCIADOS DE ACORDO COM CADA INSTRUÇÃO.

Instrução	Peso
ADD	3
SUB	3
AND	3
OR	3
STORE	2
LOAD	2
ENQ	2
IF igual	2

Ao avaliar a prioridade dos processos, observa-se que, quanto menor a soma dos pesos, menor o quantum atribuído ao processo e maior o número de "bilhetes de loteria", conforme ilustrado na Tabela 2.

TABLE II
REFERENTE AOS QUANTUS E BILHETES ASSOSCIADOS

Somatório	Quantum	Bilhetes
$0 < Soma \leq 10$	10	5
$11 \geq Soma \leq 20$	15	3
$Soma \geq 21$	20	1

Portanto, o sistema de prioridade adotado segue a lógica de que processos com menores quantums possuem maior prioridade de execução.

4) Escalonamento: O escalonamento de threads é o processo pelo qual o sistema operacional decide qual thread deve ser executada em um dado momento. O objetivo do escalonamento é gerenciar a execução concorrente de várias threads de forma eficiente, garantindo que os recursos do processador sejam utilizados de maneira otimizada.

Existem várias estratégias de escalonamento de threads, para diferentes tipos de sistema. Como a implemnetação em questão, é um sistema em lote, a principio foi utilizado um escalonamento básico de First Come First Service(FCFS) com preempção. Posteriormente foi implementado outras duas outras formas de escalonamento para realizar testes comparativos, foram eles Small Job First(SJF) e Lottery(loteria).

- FCFS: as threads são executadas na ordem de chegada, a partir de uma fila, e cada uma recebe um quantum máximo fixo, determinado pelas instruções que há nesse arquivo, cada instrução tem um peso diferente, logo, o quantum é definido em cima da prioridade total. Quando o quantum é excedido e o processo não é concluído, a thread é bloqueada e o processo é colocado novamente no final da fila. Assim que o processo é concluído, ele é removido da fila;
- SJF: a implementação é semelhante à do FCFS, porém, em vez de uma fila comum, foi utilizada uma fila de prioridade, na qual a prioridade de cada processo está relacionada ao quantum fixo que ele recebe. Quanto menor o quantum, maior a prioridade do processo, pois indica que se trata de um processo de curta duração. Se o quantum for excedido e o processo não for concluído, ele é bloqueado e recolocado na fila. Caso o processo tenha uma prioridade maior, a fila é reorganizada, colocando os processos de maior prioridade à frente para serem executados primeiro;
- Lottery: no escalonamento por lotaria, cada processo recebe um número de bilhetes proporcional à sua prioridade, que, por sua vez, está relacionada ao quantum. Quanto menor o quantum, maior o número de bilhetes atribuídos ao processo. Em seguida, é realizado um sorteio aleatório para determinar qual processo será executado, levando em consideração o peso dos bilhetes. Após a execução, os processos que permanecem na fila recebem mais bilhetes. Quando um processo é bloqueado, ele é recolocado na fila e recebe um novo número de bilhetes, continuando o ciclo até a execução de todos os processos,

5) Gerenciamento de Cache: A gestão de cache foi implementada através da classe `Cache`, utilizando uma tabela hash (`unordered_map`) com as seguintes características:

- Tamanho fixo: 8
- Estrutura de entrada:
  - `C`have: Inteiro representando o Program Counter (PC)

  - Valor: Estrutura contendo:
    * Instrução completa (`Instruction`)
    * Flag `modificado` (booleano) para controle de escrita

A política de gestão emprega dois princípios fundamentais:

- Clusterização por Similaridade:
  - Agrupamento de processos com perfis de instruções semelhantes
  - Limiar de similaridade: $\theta = 0.5$
- Cálculo de Similaridade:
  - Baseado na frequência relativa de opcodes (Equação **??**)
  - Considera A e B como conjuntos de instruções de processos distintos:

$$S(A, B) = \frac{\sum_{k \in \mathcal{O}} \min(F_A(k), F_B(k))}{\max(|A|, |B|)}$$

Como politica de gerenciamento, foi implementado uma estratégia FIFO (*First-In, First-Out*) utilizando uma fila auxiliar. O mecanismo opera da seguinte forma:

- Cada nova entrada é adicionada ao final da fila auxiliar;
- Na substituição:
  - Remove-se a entrada no início da fila (elemento mais antigo);
  - Atualiza-se a fila removendo a referência evadida;
  - Mantém-se a consistência entre a tabela hash e a fila.

6) Memory Management Unit - MMU: Como o próprio nome diz, a MMU é a unidade de gerenciamneto de memória de um sistema operacional. Ela é responsável por traduzir endereços lógicos (virtuais) gerados pela CPU em endereços físicos na memória RAM. Além disso, a MMU desempenha outras funções essenciais.

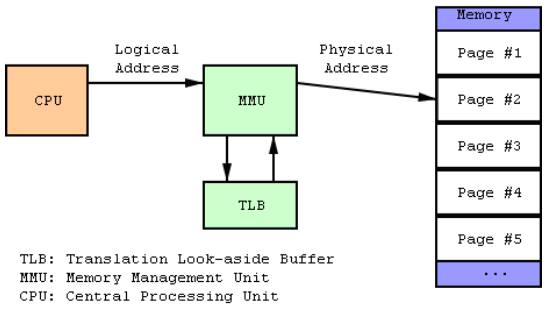

Fig. 4. Imagen ilustrativa do funcionamento de uma MMU.

1) Paginação e Segmentação – Suporte a esquemas de gerenciamento de memória, como paginação e segmentação, para permitir o uso eficiente da RAM:
   - Paginação divide a memória física e os espaços de endereçamento dos processos em blocos fixos chamados páginas (no espaço de endereços lógicos) e quadros (na memória física). Essa abordagem evita fragmentação externa e facilita a alocação dinâmica, permitindo que diferentes partes de um processo possam estar espalhadas na RAM. A MMU gerencia a tabela de páginas, que armazena o mapeamento entre páginas virtuais e seus quadros físicos correspondentes;
   - Segmentação, divide a memória em segmentos de tamanhos variáveis, correspondendo a estruturas lógicas do programa, como código, dados e pilha. Cada segmento tem um descritor, armazenado em uma tabela de segmentos, que contém informações como base, limite e permissões de acesso. Esse método melhora a organização da memória e facilita a implementação de proteção de acesso, mas pode sofrer com fragmentação externa.
2) Proteção de Memória – Impede que processos acessem áreas de memória que não lhes pertencem, prevenindo falhas e ataques:
   - Bits de permissão (leitura, escrita, execução) definidos na tabela de páginas ou na tabela de segmentos, garantindo que processos não modifiquem ou leiam áreas de memória não autorizadas;
   - Isolamento entre processos, impedindo que um programa acesse dados de outro, evitando falhas de segmentação e ataques maliciosos, como buffer overflow;
   - Geração de exceções (page fault ou segmentation fault) quando um processo tenta acessar uma região de memória inválida, permitindo que o sistema operacional tome medidas adequadas, como alocação dinâmica ou encerramento do processo infrator;
3) Memória Virtual – Permite que programas utilizem mais memória do que a RAM disponível, mapeando parte dos dados para o disco (swap):
   - Quando um processo precisa acessar uma página que não está na RAM, ocorre um page fault, e a MMU aciona o sistema operacional para buscar essa página no disco e carregá-la na memória, possivelmente substituindo uma página menos usada (usando algoritmos como LRU – Least Recently Used);
   - Esse mecanismo é essencial para permitir a execução de múltiplos processos simultaneamente, proporcionando uma ilusão de espaço de memória maior para cada programa. No entanto, acessos frequentes ao swap podem causar thrashing, reduzindo drasticamente o desempenho do sistema;
4) Cache de Tradução (TLB - Translation Lookaside Buffer) – A tradução de endereços virtuais para físicos envolve consultas a tabelas de páginas, o que pode ser lento. Para otimizar esse processo, a MMU possui um cache especial chamado TLB (Translation Lookaside Buffer), que armazena as traduções mais recentes:
   - O TLB funciona como uma memória cache de alta velocidade que reduz o tempo necessário para

traduzir endereços virtuais, minimizando acessos repetitivos à tabela de páginas na RAM;
- Se o TLB contém a entrada correspondente a um endereço virtual solicitado (TLB hit), a tradução ocorre rapidamente. Caso contrário (TLB miss), a MMU consulta a tabela de páginas na RAM e atualiza o TLB com a nova entrada;
- O desempenho do sistema melhora com um TLB eficiente, pois a busca em tabelas de páginas pode ser um gargalo, especialmente em sistemas com paginação multinível.

Sem a MMU, sistemas operacionais modernos, como Windows e Linux, enfrentariam grandes desafios para gerenciar múltiplos processos de forma segura e eficiente.

Compreendendo o papel da MMU em um sistema operacional, foi desenvolvida uma implementação inicial de um modelo simplificado, focado apenas no mapeamento dos endereços IDs de cada PCB, sem a utilização de virtualização de memória. Para os testes, adotou-se exclusivamente o escalonamento SJF (Shortest Job First).

A implementação faz uso de um vetor binário e de uma tabela hash, onde a chave corresponde ao ID mapeado em binário, armazenando todos os processos. Assim, à medida que os processos são lidos de um arquivo seus respectivos PCBs (Process Control Blocks) são criados, o ID é gerado, inserido na tabela hash e adicionado à fila de execução.

Após a inserção dos processos, a fila de binários é ordenada com base nos quantums dos processos, acessando essas informações na hash e organizando o vetor de acordo com a política do escalonador(SJF).

IV. Resultados

A saída no terminal exibe as principais informações sobre cada instrução executada por cada processo, (PCB). Além disso, são exibidas mensagens detalhando quando um processo é bloqueado, indicando qual processo foi afetado e em qual Core isso ocorreu. Também são apresentadas notificações informando quando um processo é finalizado, juntamente com o Core responsável pela execução, como o exemplo a seguir:

```
[Processo 1] Executando instrução:
 PC=4
 Opcode=1
 Destino=R10 Valor1=0 Valor2=10
State:1
Arquivo fonte: data/instructions0.txt
Quantum total utilizado: 20
clock = 20

Processo 1 bloqueado no Core 1
```

A saída permite visualizar diversas informações detalhadas sobre a execução dos processos:

- Processo em execução (linha 1): Indica qual processo está sendo executado no momento.
- Endereço da instrução (linha 2): Exibe o endereço da instrução que está em execução.
- Instrução (linha 3): Mostra o opcode correspondente à instrução.
- Registradores utilizados (linha 4): Apresenta os registradores envolvidos, incluindo o registrador de destino e os valores dos registradores utilizados.
- Estado do processo (linha 6): Indica o estado de vida do processo (1 = RUNNING).
- Arquivo fonte (linha 7): Exibe o arquivo fonte ao qual a instrução pertence.
- Quantum utilizado (linha 8): Mostra o total de quantum consumido pelo processo até o momento.
- Clock total do Core (linha 9): Informa o total de ciclos de clock já acumulados pelo Core.
- Mensagem de bloqueio (última linha): Indica que o processo foi bloqueado no Core 1.

A. Comparações sem utilização da Cache

Todos os processos foram concluídos com sucesso utilizando ambos os núcleos. A tabela a seguir apresenta o quantum máximo de ciclos de clock permitidos para cada processo, o total de ciclos efetivamente utilizados por cada núcleo e o tempo de execução correspondente a cada uma das políticas de escalonamento utilizadas. Observa-se que a soma total dos ciclos de clock dos processos deve ser igual à soma total dos ciclos de ambos os núcleos utilizados. Vale ressaltar que os testes foram realizados separadamente, mas é possível executar todos os processos simultaneamente no código. No entanto, como os núcleos e o quantum máximo não são resetados, ao rodar todos os processos juntos, os valores serão a soma total de todos os escalonadores combinados.

TABLE III
Ciclos de Clock Totais e Tempo de Execução para as 3 Políticas de Escalonamento Utilizadas

Processo / Core	FCFS	SJF	Loteria
P1	94	74	54
P2	62	62	62
P3	18	18	18
P4	33	33	33
P5	67	67	67
P6	22	22	22
Total - P	296	276	256
Core 1	149	142	119
Core 2	147	134	137
Total - Cores	296	276	256
Tempo de execução (s)	0.09297	0.082987	0.0143661

Para uma avaliação quantitativa dos algoritmos de escalonamento, três métricas foram analisadas:

1) Total de ciclos de clock consumidos pelos processos;
2) Carga computacional distribuída entre os núcleos;
3) Tempo total de execução do sistema.

A Figura 5 sintetiza visualmente essas relações por meio de uma representação dual:

- Subgráfico superior: Comparação direta dos ciclos de clock totais entre FCFS, SJF e Loteria;
- Subgráfico inferior: Tempo de execução correspondente para cada política.

Valores numéricos são destacados sobre cada elemento gráfico, permitindo uma análise das diferenças percentuais.

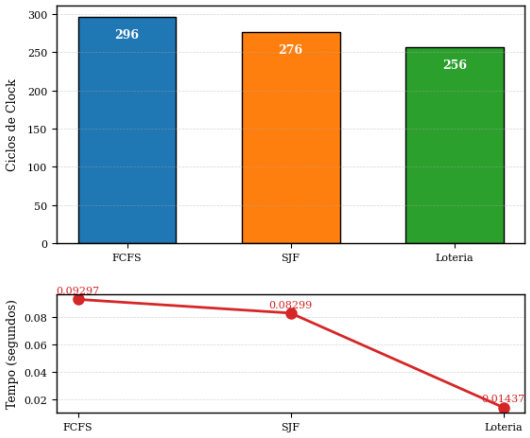

Fig. 5. Comparação entre ciclos de clock e tempo de execução. (A) Total de ciclos de clock. (B) Tempo de execução em segundos.

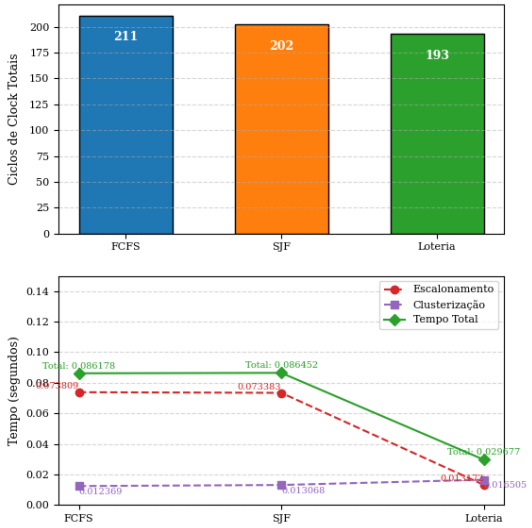

Fig. 6. Comparação entre ciclos de clock e tempo de execução. (A) Total de ciclos de clock. (B) Tempo de execução em segundos.

B. Comparações com utilização da Cache

Novamente os processos foram concluídos com sucesso, e os dados podem ser visualizados na Tabela IV. O tempo de execução total é somado tanto o tempo necessário para realizar a clusterização dos processos quanto o tempo gasto pelo escalonador. O restante dos dados se fazem na mesma perspectiva que da tabela anterior.

TABLE IV
CICLOS DE CLOCK TOTAIS E TEMPO DE EXECUÇÃO PARA AS 3 POLÍTICAS DE ESCALONAMENTO UTILIZADAS - COM CACHE

Processo / Core	FCFS	SJF	Loteria
P1	80	60	60
P2	55	44	35
P3	32	43	44
P4	22	31	30
P5	5	5	5
P6	17	19	19
Total - P	211	202	193
Core 1	102	113	99
Core 2	109	89	94
Total - Cores	211	202	193
Tempo de Escalonamento(s)	0.0738094	0.0733833	0.0131721
Tempo de Clusterização(s)	0,012369	0.0130682	0.0165046
Tempo de execução Total(s)	0.0861784	0.0864515	0.0296767

Para uma avaliação quantitativa dos algoritmos de escalonamento com utilização da cache, as seguintes métricas foram analisadas:

1) Total de ciclos de clock consumidos pelos processos;
2) Carga computacional distribuída entre os núcleos;
3) Tempo total de execução da política de escalonamento;
4) Tempo de encapsulamento de processos;
5) Tempo total de execução do sistema.

1) Utilização da MMU - Análises em cima do SJF: Os processos foram concluídos com sucesso, e os dados podem ser visualizados na Tabela V. O tempo de execução total é somado o tempo necessário para realizar a clusterização dos processos e o tempo gasto pelo escalonador. Importante ressaltar que foi utilizado apenas o SJF para as análises.

TABLE V
CICLOS DE CLOCK TOTAIS E TEMPO DE EXECUÇÃO PARA AS 3 POLÍTICAS DE ESCALONAMENTO UTILIZADAS - COM CACHE

Processo / Core	SJF-MMU	SJF
P1	50	60
P2	53	44
P3	43	43
P4	31	31
P5	5	5
P6	19	19
Total - P	201	202
Core 1	109	113
Core 2	92	89
Total - Cores	201	202
Tempo de Escalonamento(s)	0.0731845	0.0733833
Tempo de Clusterização(s)	0.0123258	0.0130682
Tempo de execução Total(s)	0.0855103	0.0864515

Para uma avaliação quantitativa dos algoritmos de escalonamento com utilização da cache, as seguintes métricas foram analisadas:

1) Total de ciclos de clock consumidos pelos processos;
2) Carga computacional distribuída entre os núcleos;
3) Tempo total de execução da política de escalonamento;
4) Tempo de encapsulamento de processos;
5) Tempo total de execução do sistema.

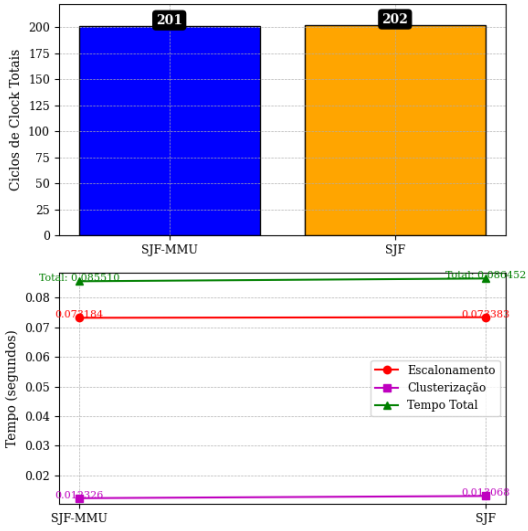

Fig. 7. Comparação entre ciclos de clock e tempo de execução. (A) Total de ciclos de clock. (B) Tempo de execução em segundos.

C. Conclusões

1) Módulo 2 - Escalonamentos: A partir dos resultaddos optidos, podemos visualizar alguns resultados importantes com relação ao que foi proposto:

- Em todos os algoritmos de escalonamento (FCFS, SJF e Loteria), os processos foram distribuídos entre os dois núcleos (Cores), garantindo o uso paralelo e eficiente dos recursos do sistema.
- Nos algoritmos FCFS e SJF, a carga de trabalho está bem equilibrada entre os dois núcleos, com uma diferença muito pequena no total de ciclos executados por cada núcleo. A diferença é inferior a 1% no caso do FCFS e de apenas 2,9% no caso do SJF. Já no algoritmo de Loteria, observa-se uma diferença maior, de 7,04%, o que é esperado devido à natureza probabilística do escalonamento.
- Como evidenciado na Figura 5, é possível analisar as relações entre os ciclos totais, o tempo de execução e a vazão dos processos:
    1) Considerando o total de ciclos, observa-se uma diferença de 6,2% entre FCFS e SJF, e de 13,5% entre Loteria e FCFS. Já a diferença entre Loteria e SJF é de 6,7%.
    2) Em relação ao tempo de execução, a diferença entre FCFS e SJF é de 9,8%, enquanto que entre FCFS e Loteria, a diferença é de 84,6%. A diferença entre Loteria e SJF é de 73,8%
    3) Com base nas métricas anteriores, podemos estimar a vazão de cada escalonador. Observa-se uma diferença de 1,9% a mais de vazão no SJF em comparação ao FCFS, enquanto as diferenças de vazão entre Loteria e os outros escalonadores são bastante significativas: 84,6% a mais em relação ao FCFS e 82,7% a mais em relação ao SJF.

2) Módulo 3 - Gerenciamneto de cachê: A partir dos resultados obtidos, observamos padrões significativos no desempenho dos algoritmos de escalonamento:

- Em todos os algoritmos de escalonamento (FCFS, SJF e Loteria), os processos foram distribuídos entre os dois núcleos (Cores), garantindo o uso paralelo e eficiente dos recursos do sistema.
- Nos algoritmos FCFS e SJF, a carga de trabalho está relativamente equilibrada entre os dois núcleos, com uma diferença de aproximadamente 6,75% no total de ciclos executados por cada núcleo em FCFS e de 26,97% em SJF. No algoritmo de Loteria, a diferença é de 5,05%, o que indica uma distribuição menos desigual entre os núcleos em relação ao SJF, mas ainda maior do que no FCFS.
- Como evidenciado na Tabela 6, é possível analisar as relações entre os ciclos totais, o tempo de execução e a vazão dos processos:
    1) Considerando o total de ciclos, observa-se uma diferença de 4,27% entre FCFS e SJF, e de 8,53% entre Loteria e FCFS. Já a diferença entre Loteria e SJF é de 4,46
    2) Em relação ao tempo de execução total, a diferença entre FCFS e SJF é de 0,31%, enquanto que entre FCFS e Loteria, a diferença é de 65,55%. A diferença entre Loteria e SJF é de 65,66%.
    3) Com base nas métricas anteriores, podemos estimar a vazão de cada escalonador. Observa-se uma diferença de 0,37% a mais de vazão no SJF em comparação ao FCFS, enquanto as diferenças de vazão entre Loteria e os outros escalonadores são bastante significativas: 65,55% a mais em relação ao FCFS e 65,66% a mais em relação ao SJF.
    4) Além disso, ao analisar o tempo de clusterização em relação ao tempo total de execução, observa-se que para o FCFS, a clusterização representa aproximadamente 14,36% do tempo total de execução. No caso do SJF, esse percentual é de 15,12%, enquanto que para o escalonador de Loteria, a clusterização corresponde a 55,61% do tempo total de execução. Isso evidencia que o impacto do tempo de clusterização é consideravelmente maior no algoritmo de Loteria, influenciando significativamente sua eficiência total.

3) Análise de Desempenho com Utilização de Cache: A Figura 8 apresenta a comparação dos ciclos de clock com e sem cache, enquanto a Figura 9 demonstra o impacto no tempo de execução total. Os resultados revelam três aspectos críticos no uso de cache:

Com relação ao impacto no Tempo de Escalonamento, conforme evidenciado na Figura 9, observa-se que:

- Algoritmo de Loteria: Redução de 8.36%: $\Delta t = 0.0132s \rightarrow 0.0121s$
- Melhoria marginal de 5.55%: $\Delta t = 0.0738s \rightarrow 0.0697s$
- Ganho moderado de 11.24%: $\Delta t = 0.0734s \rightarrow 0.0651s$

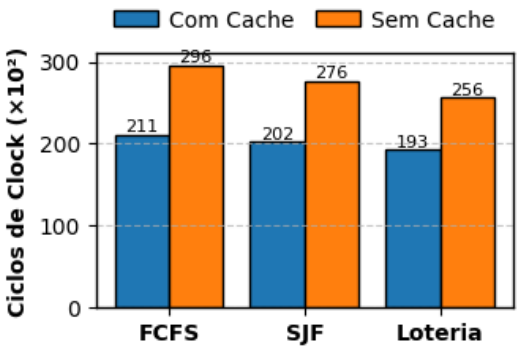

Fig. 8. Comparação entre ciclos de clock com utilização de cache e sem utilização de cache.

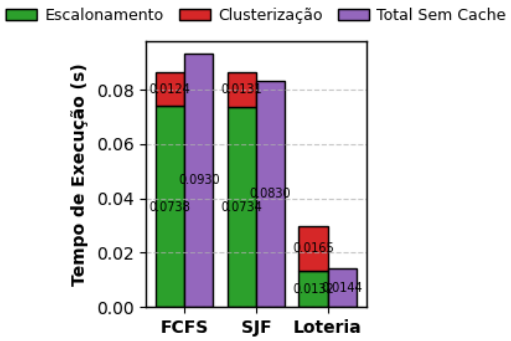

Fig. 9. Comparação entre tempo de execução com utilização de cache e sem utilização de cache.

- Eficiência no Tempo Total de Execução:
  - FCFS: $cache = 6.57\%(0.09297s \rightarrow 0.08618s)$
  - SJF: $cache = 4.17\%(0.08299s \rightarrow 0.07952s)$
  - Loteria: $cache = 79.32\%(0.14367s \rightarrow 0.02968s)$
- Analisando o Overhead de Clusterização, temos:
  - FCFS: $PsemCluster = 13.30\% \rightarrow PcomCluster = 14.36\%(\Delta = +7.98\%)$
  - SJF: $PsemCluster = 15.74\% \rightarrow PcomCluster = 15.12\%(\Delta = 3.94\%)$
  - Loteria: $PsemCluster = 38.42\% \rightarrow PcomCluster = 55.61\%(\Delta = +44.79\%)$
- Redução do ciclos de clock:
  - FCFS: $\gamma = 28.72\%(296 \rightarrow 211)$
  - SJF: $\gamma = 26.81\%(276 \rightarrow 202)$
  - Loteria: $\gamma = 24.61\%(256 \rightarrow 193)$

Os resultados demonstram que a utilização de cache produz ganhos assimétricos dependendo da política de escalonamento:

- Para Loteria, o cache reduz drasticamente o tempo total (79.32%) porém aumenta o overhead relativo de clusterização (+44.79%);
- Em FCFS / SJF, a redução moderada de ciclos (28.72% / 26.81%) não se traduz proporcionalmente em ganhos temporais (<12%);
- O trade-off ótimo ocorre no SJF, combinando redução de ciclos (26.81%) com estabilidade no $P(-3.94\%)$;

4) Módulo 4 - MMU: A Figura 10 apresenta a comparação separadamente dos ciclos de clock entre SJF com MMU e SJF sem a MMU, enquanto a Figura 11 demonstra o impacto no tempo de execução total.

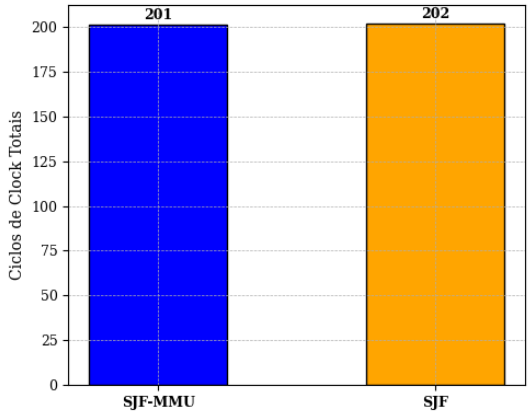

Fig. 10. Comparação entre ciclos de clock entre SJF com MMU e sem MMU.

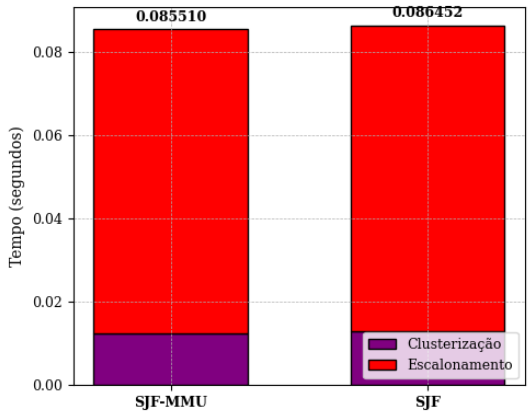

Fig. 11. Comparação entre tempo de execução entre SJF com MMU e sem MMU.

É importante destacar que, no SJF sem a utilização da MMU, o escalonamento era realizado com base em um vetor contendo apenas os IDs das PCBs. Com a introdução da MMU, passou-se a operar diretamente sobre um vetor de binárioso. Essa otimização ocorre porque a movimentação de binários(1 byte) sem acessar e modificar bits internos tem uma performance um pouco melhor com relação ao movimentar inteiros (4 bytes) no contexto do C++ com arquitetura de 64 bits.

Com relação ao impacto no Tempo de Escalonamento, conforme evidenciado nos gráficos, observa-se que a utilização do SJF-MMU resultou em uma leve redução de 0.27% no tempo de escalonamento, passando de 0.0733833s para 0.0731845s. Embora a melhoria seja marginal, indica que a abordagem com MMU não impactou negativamente a velocidade do escalonamento.

Em termos de eficiência no Tempo Total de Execução, o SJF-MMU demonstrou uma redução de 1.0%, reduzindo de 0.0864515s para 0.0855103s. Esse ganho pode ser atribuído a um melhor aproveitamento do cache, mesmo que a diferença absoluta seja pequena.

Ao analisar o Overhead de Clusterização, observou-se uma redução de 5.67%, com o tempo de clusterização caindo de 0.0130682s para 0.0123258s.

Com relação à redução dos ciclos de clock, a mudança foi mínima, registrando uma queda de apenas 0.50%, indo de 202 para 201 ciclos no total. Esse dado indica que a mudança na estratégia de escalonamento não trouxe um impacto significativo no número total de ciclos necessários para a execução.

5) Conclusão geral: A análise dos resultados mostra que a utilização de cache impacta o desempenho dos algoritmos de escalonamento, mas de forma desigual dependendo da política adotada. Em termos gerais, o cache reduz a quantidade total de ciclos de clock para todos os algoritmos, o que sugere um aumento na eficiência do processamento. Entretanto, a relação entre essa redução e a melhoria no tempo total de execução é variável.

Para o algoritmo de Loteria, o cache proporciona a maior redução no tempo total de execução (79.32%), indicando que a política probabilística se beneficia da melhoria no acesso aos dados. No entanto, essa otimização vem acompanhada de um aumento no overhead relativo de clusterização (+44.79%), o que pode comprometer a escalabilidade dependendo da carga de trabalho.

Já nos algoritmos FCFS e SJF, a redução dos ciclos de clock é moderada (28.72% e 26.81%, respectivamente), mas os ganhos no tempo de execução são bem menores, não ultrapassando 12%. Isso sugere que a eficiência geral desses algoritmos não depende apenas da quantidade de ciclos executados, mas também de outros fatores, como a distribuição das tarefas entre os núcleos e o próprio overhead de gerenciamento.

O SJF se destaca como o algoritmo que melhor equilibra os benefícios do cache, pois apresenta uma redução nos ciclos de clock (26.81%) e mantém um overhead de clusterização mais próximo da estabilidade (3.94%). Esse comportamento indica que o SJF consegue aproveitar a otimização do cache sem gerar um custo adicional, tornando-se uma escolha em cenários onde o balanceamento entre desempenho e sobrecarga de gerenciamento possam ser mais relevantes.

Em suma, a utilização de cache é uma estratégia eficiente para otimizar o desempenho do sistema, mas seu impacto depende fortemente da política de escalonamento utilizada. O algoritmo de Loteria apresenta os maiores ganhos absolutos, mas ao custo de um maior overhead, enquanto o SJF se mostra o mais equilibrado, oferecendo melhorias consistentes sem penalidades significativas.

Com relação a utilização da MMU, o resultado geral sugere que a adoção do SJF-MMU, que realiza o escalonamento diretamente sobre um vetor de binários ao invés de operar apenas sobre os IDs dos PCBs, trouxe melhorias sutis. O impacto no tempo total de execução e a redução no overhead de clusterização indicam que a abordagem baseada em MMU pode otimizar a organização e o acesso à memória sem comprometer o desempenho geral do sistema, mesmo aplicando uma implementação rudimentar. A leve melhoria no tempo de escalonamento reforça que a nova técnica não adiciona um custo computacional significativo. Em um cenário de grande volume de processos, essa abordagem pode se tornar ainda mais vantajosa, proporcionando ganhos cumulativos em eficiência e organização da memória, principalmente numa implementação mais robusta.

REFERENCES

[1] W. Stallings, “Arquitetura e Organização de Computadores,” 8th ed. Always Learning, Nov. 2010.

[2] A.S.Tanenbaum and T.Austin, Structure Computer Organization, 6th ed., Pearson, 2013

[3] A. José Marconi, “Simulador Mips: Sistemas Operacionais: Simulador da Arquitetura de Von Neumann e Pipeline MIPS“, Disponível em: https://github.com/josemarconi/Simulador-Mips