UNIVERSIDADE FEDERAL DO PARANÁ

MARCELO FERREIRA DA SILVA CARDOSO

# AVALIAÇÃO DO ESPAÇO PARA MELHORIAS DE POLÍTICAS DE SUBSTITUIÇÃO DE LINHAS DE CACHE

CURITIBA PR

2021

MARCELO FERREIRA DA SILVA CARDOSO

# AVALIAÇÃO DO ESPAÇO PARA MELHORIAS DE POLÍTICAS DE SUBSTITUIÇÃO DE LINHAS DE CACHE

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA PR

2021

## RESUMO

Cache foi o nome dado ao nível de memória intermediário entre o processador e a memória principal, e, hoje, o termo também é usado para se referir a qualquer dispositivo de armazenamento que se aproveite da localidade de acesso. Sendo uma memória de menor latência e maior velocidade, porém de capacidade reduzida, a cache necessita de um bom gerenciamento de suas entradas, visando sempre manter aqueles dados cujo intervalo de reuso é mais curto que outros. Quando cheia, uma das várias entradas da memória cache precisa ser desalocada para que um novo bloco de dados possa ser carregado da memória principal, função exercida por uma política de substituição de linhas de cache.

Ao longo dos anos, muitas políticas foram propostas, cada uma com vantagens e desvantagens, mas nenhuma capaz de atingir um desempenho ótimo, já que este só seria possível com informações de referências futuras a blocos da memória. Contudo pouco se sabe sobre a proximidade da performance desses algoritmos a um ótimo, isto é, é difícil determinar se existem margens para ganhos e, consequentemente, se vale a pena investir recursos no desenvolvimento de novas políticas.

Assim, neste trabalho é proposto um estudo para se determinar as margens de ganhos de um algoritmo ótimo, como o MIN, proposto por Belady (1966), frente àquelas políticas mais comumente utilizadas em processadores modernos. Para tal, foi desenvolvido um simulador funcional de cache, capaz de receber instruções de memória e contabilizar o número total de faltas de dados, viabilizando uma comparação entre os algoritmos selecionados.

Como resultado, foram encontrados ganhos de até 22,6% sobre a média geral de políticas informadas, bem como constatou-se melhorias acima de 10% sobre todas as políticas em pelo menos 80% das aplicações. Desta forma, pôde-se concluir que ainda há grandes margens para melhorias, ainda que seja incerto qual o impacto que uma política ótima de substituição de linhas de cache exerce no desempenho final de um sistema.



Palavras-chave: Memórias Cache; Políticas de Substituição; Avaliação de Desempenho;

## ABSTRACT

Cache memory was the name given to the intermediate level of memory between the processor and the main memory, but, today, this denomination is also used to refer to any storage device that leverages the locality of access. Being a memory with low latency and high speed, but low capacity, the cache requires excellent management of its entries, always looking to maintain those blocks of data whose reuse interval is shorter than others. When it is full, one of the many cache memory entries needs to be evicted in favor of a new block coming from the main memory, task achieved by a cache replacement policy.

Many policies have been proposed throughout the years, each with its advantages and disadvantages, but none capable of achieving optimal performance, since that would only be possible with knowledge of future memory references. However, little is known about these algorithms' performance proximity to that presented by an optimal policy. In other words, it is difficult to determine if there is any margin for gains and, consequently, if an investment in developing new replacement policies is worthwhile.

Therefore, this study intends to determine the gains of an optimal algorithm, like the one proposed by Belady (1966), against those policies that are commonly used in modern processors. For this, a functional cache simulator, capable of receiving memory instructions and counting the total number of cache misses, was developed to compare the selected algorithms.

As results showed, a perfect replacement policy could improve up to 22.6% against the global mean of informed policies, while improvements above 10% against all policies in at least 80% of the input applications were observed. Thus, it was established that there are still significant improvement margins to be extracted, even though the impact of an optimal cache replacement policy in the final system performance is still unkow and requires further investigation.



Keywords: Cache Memories; Replacement Policies; Performance Evaluation;

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ACRÔNIMOS

| | |
|---|---|
| BRRIP | *Bimodal Re-Reference Interval Prediction* |
| CPU | *Central Processing Unit* |
| DInf | Departamento de Informática |
| DRAM | *Dynamic Random Access Memory* |
| DRRIP | *Dynamic Re-Reference Interval Prediction* |
| FIFO | *First in, First out* |
| LFSR | *Linear-Feedback Shift Register* |
| LRU | *Least Recently Used* |
| LTS | *Long Term Support* |
| MRU | *Most Recently Used* |
| NRU | *Not Recently Used* |
| PLRU | *Pseudo Least Recently Used* |
| PLRUm | *MRU (Most Recently Used) based Pseudo Least Recently Used* |
| PLRUt | *Tree based Pseudo Least Recently Used* |
| QLRU | *Quad-age Least Recently Used* |
| PRNGl | *LFSR (Linear-Feedback Shift Register) Pseudo Random Number Generator* |
| RRIP | *Re-Reference Interval Prediction* |
| RRPV | *Re-Reference Prediction Value* |
| PRNGr | *Round Robin Pseudo Random Generator* |
| SPEC | *Standard Performance Evaluation Corporation* |
| SRAM | *Static Random Access Memory* |
| SRRIP | *Static Re-Reference Interval Prediction* |
| UFPR | Universidade Federal do Paraná |

**SUMÁRIO**

# 1 INTRODUÇÃO

A possibilidade de trabalhar com quantidades ilimitadas de memória rápida foi um sonho de muitos programadores durante muito tempo, uma vez que todo o conjunto de trabalho de um programa estaria prontamente disponível ao processador, eliminando a necessidade de otimizações na busca de dados ou instruções. Contudo os altos custos de produção de memórias rápidas e a impossibilidade de tê-las em grandes quantidades sempre inviabilizaram a realização desse desejo. Para criar a ilusão ao programador de que a memória instalada possui essas propriedades, foi, então, idealizada uma hierarquia de memória baseada no princípio da localidade, que afirma que programas acessam apenas pequenos subconjuntos do espaço de endereçamento total em um dado instante (Patterson e Hennessy, 2014).

Uma hierarquia de memória é formada por múltiplos níveis de memórias das mais variadas capacidades, performances e tecnologias. Enquanto a memória principal utiliza a tecnologia DRAM (*Dynamic Random Access Memory*), níveis mais próximos do processador, como as caches, que serão o foco desse trabalho, são implementados com a tecnologia SRAM (*Static Random Access Memory*). Por utilizarem apenas um transistor e um capacitor por bit de armazenamento, as DRAM são muito mais densas e baratas que as SRAM, que precisam de seis transistores por bit, porém são também muito mais lentas, pois necessitam de um endereçamento de dois níveis e de um circuito interno de amplificação de sinal e atualização dos dados, que culminam em um tempo de acesso 5 a 10 vezes maior (Patterson e Hennessy, 2014). Dessa forma, as caches possuem capacidades bem reduzidas, apesar de seus tempos de acesso também menores.

Segundo Patterson e Hennessy (2014), a unidade mais básica de transferência de dados entre níveis da hierarquia de memória é chamada de bloco, ou linha. Dizemos que há um acerto na cache, ou cache *hit*, quando um dado requisitado está instalado nela, mas que há uma falta na cache, ou cache *miss*, caso contrário, e, associada a ela, uma penalidade, oriunda do tempo necessário para buscar o bloco contendo o dado na memória principal, inseri-lo na cache e notificar o processador. Porém, caso não existam mais espaços livres, deve-se, antes de inserir o bloco, desalocar uma linha já instalada, onde a seleção da "vítima", isto é, a linha a ser removida, será feita por um algoritmo de substituição de linhas de cache (Zanata, 2009).

Com a busca incessante por desempenho, diversas técnicas, como *pipeline* e superescalaridade, foram desenvolvidas para se extrair o máximo dos processadores (Zanata, 2009), e, naturalmente, esse progresso chegou às memórias, já que são delas que o processador obtém os dados para exercer sua função. Contudo Damien (2007) menciona uma crescente diferença de performance entre memórias e processadores, com o tempo de acesso dessa sendo ordens de magnitude maior que um ciclo de relógio deste. Para este problema, a cache se apresenta como uma solução (Damien, 2007) ao manter parte do conjunto de trabalho próximo ao processador, mas, por ser de tamanho reduzido, deve gerenciar muito bem seu espaço.

Ao passar dos anos, todavia, esse gerenciamento parece ter estagnado, já que pouco avança na aproximação do comportamento de uma política considerada ótima, ainda que muitas propostas de novos algoritmos de substituição tenham surgido. Segundo AMD (2017), a política usada na cache de nível um de instruções da família de processadores 17h é baseada no algoritmo LRU (*Least Recently Used*), enquanto Abel e Reineke (2020) citam o uso de variações do PLRU (*Pseudo Least Recently Used*) e do QLRU (*Quad-age Least Recently Used*) nos processadores Intel.

Dado o custo intrínseco da penalidade de uma falta de dados, uma escolha ruim do próximo bloco a ser removido pode ser desastrosa para o desempenho final da cache, uma vez que será necessário buscá-lo novamente dos níveis inferiores posteriormente. Ainda que, hoje, seja comumente encontrado até três níveis de cache para amortizar este problema, idealmente busca-se remover da memória aquele bloco que não será mais utilizado, ou que ficará sem uso durante um intervalo de tempo maior que os outros (Al-Zoubi et al., 2004), mas para isso seria necessário ter conhecimento de todas as requisições que ainda seriam feitas. Ter pleno conhecimento do futuro, todavia, é impossível, e, dessa forma, os algoritmos práticos propostos até hoje tomam decisões apenas com base em previsões ou randomizações, podendo alterá-las quando novas informações estiverem disponíveis (Jaleel et al., 2010).

Assim, embora já existam propostas de novas políticas de substituição de linhas de cache, como a apresentada por Jaleel et al. (2010), nota-se uma dificuldade em afirmar qual é o potencial de melhoria que uma nova política pode apresentar frente àquelas já existentes. Para se obter tal informação, é imprescindível o uso de um algoritmo oráculo, isto é, de um algoritmo ótimo, que, embasado em informações sobre futuras requisições, é capaz de tomar a melhor decisão na substituição de dados da cache, possibilitando, portanto, comparar as diversas propostas levantadas e estimar o ganho máximo que um novo algoritmo pode manifestar.

## 1.1 MOTIVAÇÃO

Belady (1966), em seu estudo sobre algoritmos de substituição para memória virtual, apresenta o MIN, um algoritmo ótimo capaz de determinar o número mínimo de vezes que o dispositivo de armazenamento em massa é acessado para se obter páginas de memória. Se a memória principal for vista como uma cache dos níveis subsequentes, uma vez que armazena parte do conjunto de trabalho do processador, o algoritmo pode ser facilmente adaptado ao objeto de estudo dessa monografia, ainda que seja voltado à pesquisa de memórias virtuais.

Além disso, sabe-se que o uso de simuladores baseados em traços de execução de programas reais é bastante comum na arquitetura de computadores, uma vez que a construção de protótipos é inviabilizada pelo alto custo de produção de uma pastilha de silício, bem como torna-se possível saber quais são as próximas referências à memória feitas pelo processador. Com isso, e tendo encontrado um algoritmo ótimo, passa a ser plausível a construção de um simulador que receba instruções de memória e reproduza o comportamento de uma cache, alimentando, quando necessário, adaptações do MIN, ou de outras políticas, com essas instruções. Desta forma, se torna possível extrair dados conclusivos dos testes feitos em um ambiente controlado pela simulação.

## 1.2 OBJETIVO

Ao se considerar todas as despesas envolvidas com pesquisa e desenvolvimento, fica evidente a necessidade de se determinar os ganhos que uma melhoria de um componente pode trazer antes de se alocar recursos para trabalhar em cima dela, afinal os custos de pesquisa são, muitas vezes, muito altos para se obter retornos muito baixos. Dessa maneira, o objetivo desse trabalho é apontar a performance máxima que o MIN pode oferecer e determinar o quão distante dela estão os desempenhos das políticas de substituição de linhas de cache mais comumente utilizadas.

Para responder aos questionamentos apresentados, são levantadas algumas políticas consideradas relevantes. Para testá-las, um simulador simples de cache, capaz de receber

modularmente novas políticas, é implementado. Por fim, seus resultados são comparados àqueles do algoritmo ótimo.

## 1.3 ESTRUTURAÇÃO TEXTUAL

Nos capítulos restantes, dois trabalhos correlatos, e como esta monografia difere deles, são colocados na revisão bibliográfica encontrada no capítulo 2, enquanto os conceitos necessários para o devido desenvolvimento deste estudo, bem como seu entendimento, podem ser encontrados na fundamentação teórica, ou capítulo 3. Já no capítulo 4, a metodologia apresenta os métodos e ferramentas mais relevantes utilizados no decorrer deste trabalho. Por fim, os capítulos 5 e 6, dispõem os dados encontrados, discutem sobre pontos importantes vistos neles e levantam conclusões, bem como apontam o que pode esperar-se de trabalhos futuros.

# 2 REVISÃO BIBLIOGRÁFICA

Apesar de abordarem temas diferentes, os artigos apresentados neste capítulo retratam bem o progresso do desenvolvimento de novas políticas de substituição de linhas de cache. Enquanto Al-Zoubi et al. (2004) fazem uma avaliação dos algoritmos LRU, PLRUt (*Tree based Pseudo Least Recently Used*), PLRUm (*MRU based Pseudo Least Recently Used*), FIFO (*First in, First out*) e aleatório, bem como um ótimo, Jaleel et al. (2010) propõem duas novas políticas de substituição de linhas de cache, baseadas na cadeia RRIP (*Re-Reference Interval Prediction*), que é uma reinterpretação da ordem de referência dos blocos inseridos na cache. Essa monografia busca se inspirar no primeiro, porém difere ao incluir os algoritmos PRNGr (*Round Robin Pseudo Random Generator*), PRNGl (*LFSR Pseudo Random Number Generator*) e QLRU e ao se ater a determinação da diferença de performance entre as políticas mais comumente usadas na atualidade e uma política ótima, como o MIN, apresentado anteriormente. Em seguida, encontram-se, resumidos, ambos os trabalhos.

A avaliação de Al-Zoubi et al. (2004) tenta responder algumas questões, como qual o potencial de melhoria que políticas mais avançadas que a LRU podem apresentar, se há consistência na melhoria apresentada por uma política em relação à outra e quão boa é a aproximação da política PLRU à LRU, levantadas pelos autores a partir de observações comuns sobre políticas de substituição. Para tal, simulam, usando as ferramentas *sim-cache* e *sim-cheetah* da versão alfa do simulador *SimpleScalar*, a execução de programas em caches de variados tamanhos e configurações, a fim de levantar e comparar os números de faltas total e a cada mil instruções.

Os resultados obtidos mostram que, para aplicações baseadas em inteiros, as políticas de substituição atingem um ponto de retornos reduzidos para caches pequenas de 8 vias e caches maiores de 2 e 4 vias, ou seja, para estas configurações e outras de tamanho ou associatividades maiores, os ganhos de desempenho obtidos de políticas de substituição são muito baixos. Enquanto isso, para aplicações baseadas em pontos flutuantes, associatividades maiores são mais benéficas para caches de tamanho limitado em alguns cenários, mas são dispensáveis quando aumenta-se a capacidade.

Além disso, foi observado também que os percentuais de faltas do LRU são aproximadamente os mesmos dos apresentados pelo algoritmo ótimo em questão, quando esse opera em uma cache com duas vezes a capacidade da utilizada neste, mas ambas com mesmo o número de vias, ainda que, para caches de mesmas configurações, a política ótima escolhida tenha oferecido uma diminuição de até 40% no número de *misses*. Finalmente, as heurísticas PLRUt e PLRUm são muito eficientes na aproximação da LRU, onde esta apresenta melhores métricas que essa, proporcionando, assim, o melhor desempenho pelo menor custo dentre as analisadas.

Das políticas propostas por Jaleel et al. (2010), a primeira e mais simples das duas, chamada pelos autores de SRRIP (*Static Re-Reference Interval Prediction*), funciona de forma análoga à NRU (*Not Recently Used*), contudo amplia a granularidade dos intervalos ao permitir $M$ bits para armazenamento das previsões. Para atualizar suas hipóteses na ocorrência de um acerto, a SRRIP pode priorizar *hits*, prevendo um intervalo de reuso mais curto, ou a frequência de uso, assumindo que blocos que sofram mais acertos tenham mais chance de serem reutilizados logo.

Mais complexa que a primeira, a DRRIP (*Dynamic Re-Reference Interval Prediction*) utiliza-se do duelo de conjuntos para determinar o padrão de acesso à memória num dado momento e decidir, dentre duas políticas de substituição, qual será usada. Uma das opções é a

própria SRRIP explicada anteriormente, enquanto a outra é a BRRIP (*Bimodal Re-Reference Interval Prediction*), que prevê que uma parte maior dos blocos será re-referenciada apenas num futuro distante, ao passo que o restante é previsto para ser referenciado novamente num futuro mais próximo. Assim, parte do conjunto de trabalho é preservado quando esta política se defronta com certos padrões de acesso, como o *thrashing*, explicado no capítulo 3. Segundo os autores, tanto este algoritmo quanto o SRRIP são resistentes a *scan*, que pode ser definido como uma sequência de referências a dados cujos reusos acontecem somente num futuro distante.

Os pesquisadores descobriram que sempre prever um intervalo máximo de re-referência apresenta a pior performance, pois a SRRIP não tem tempo suficiente para melhorar a previsão de intervalo do bloco, logo, ao prever um intervalo um pouco mais curto, blocos que não possuem localidade temporal não irão poluir a cache por um extenso período de tempo, o que acarreta em uma redução entre 6 e 10% no número de cache *misses* em comparação à LRU. O artigo aponta ainda que priorizar a frequência de uso faz com que a SRRIP tenha uma redução entre 5 e 18% no número de *misses* a cada mil instruções, conseguindo, assim, ultrapassar a performance da LRU em 2,5% em média, mas priorizar os *hits* permite à política ampliar essa margem para 5%. Dessa forma, fornecendo o dobro de performance de sua alternativa, concluiu-se que o benefício de uma política resistente à *scan* vem da preservação do conjunto ativo de trabalho e não da frequência de uso de seus blocos. Além disso, quando o número de bits do registrador RRPV (*Re-Reference Prediction Value*) é maior que três, algumas cargas sofreram com a degradação da performance, uma vez que blocos sem uso permanecem na cache por mais tempo após seus últimos *hits*, reduzindo sua capacidade efetiva.

Por fim, foi levantado que a DRRIP melhora, em média, 5% a performance apresentada pela SRRIP, mas como é otimizada para a redução do número de cache *misses* e não para o aumento da *throughput*, a política pode reduzir a performance de aplicações quando o custo de uma falta varia, como foi o caso do *photoshop*.

Além dos artigos apresentados, vale destacar, também, as contribuições de campeonatos, como o CRC (2017), para o progresso desta área, ainda que sejam raros, com apenas duas edições tendo acontecido nos anos de 2009 e de 2017. Competições do tipo incentivam propostas como a apresentada por Jain e Lin (2017), chamada pelos altores de *HawkEye*, e desta forma, é possível que edições anuais acelerem ainda mais o desenvolvimento de políticas de substituição de linhas de cache, possibilitando melhores aproximações do ótimo e implementações cada vez menos custosas.

# 3 FUNDAMENTAÇÃO TEÓRICA

Nas seções subsequentes, está situado o embasamento teórico desta monografia, ou seja, estão apresentados os conceitos necessários para o devido desenvolvimento deste trabalho e seu bom entendimento.

## 3.1 HIERARQUIA DE MEMÓRIA

Uma hierarquia de memória consiste de múltiplos níveis de memórias com diferentes tamanhos e tempos de acesso, fazendo com que os dados sejam similarmente hierárquicos. Assim, um nível superior, ou seja, aquele mais próximo do processador, é geralmente um subconjunto de qualquer nível subsequente, enquanto todos os dados são armazenados no nível mais inferior (Patterson e Hennessy, 2014).

A menor unidade de informação que pode estar, ou não, presente em um dado nível da hierarquia é chamada de bloco ou linha, podendo ser transferido apenas entre dois níveis adjacentes por vez. Chama-se de acerto o cenário em que o dado requisitado pelo processador aparece em algum bloco no nível superior, caso contrário diz-se que há uma falta, que ocasiona um acesso ao nível inferior para que o bloco contendo o dado requisitado seja recuperado (Patterson e Hennessy, 2014).

O desempenho da hierarquia de memória pode ser medido através tanto da taxa de acertos, que é a fração de acessos à memória que incidiram em acertos, quanto de seu complemento, chamada de taxa de faltas. Cada acerto demanda um tempo para acessar o nível atual da hierarquia, chamado de tempo de acerto, que inclui o tempo para se determinar se o dado está ou não presente na memória, enquanto cada falta acarreta em uma penalidade de falta, que demanda um tempo maior de acesso, uma vez que é necessário substituir um bloco no nível atual com o bloco correspondente do nível inferior, além de entregar o dado ao processador (Patterson e Hennessy, 2014).

Como performance é uma das razões principais para se ter uma hierarquia de memória, o tempo de acesso à memória é importante; assim, as tecnologias que permitem acessos mais rápidos e maiores velocidades são, geralmente, implementadas nas memórias mais próximas ao processador, mas possuem capacidades menores por serem muito mais caras por bit de armazenamento (Patterson e Hennessy, 2014).

## 3.2 MEMÓRIAS CACHE

Quando o primeiro computador comercial foi vendido com um nível de memória intermediário entre o processador e a memória principal, o nome dado a esse nível extra foi cache. Hoje, apesar de ainda ser o emprego mais comum da palavra, o termo também é usado para se referir a qualquer dispositivo de armazenamento que se aproveite da localidade de acesso (Patterson e Hennessy, 2014).

A cache possui, junto a suas entradas, um conjunto de *tags* para determinar a presença ou não de um dado, que recebem apenas os bits mais significativos do endereço, não usados pela indexação e necessários para identificar unicamente uma palavra na cache. Cada entrada necessita, também, de um bit de validade para indicar se a linha possui dados válidos ou não, fazendo com que uma combinação só seja possível se o bit for configurado. Quando um processador inicia seu

trabalho, a cache não possui dados úteis, e mesmo após executar várias instruções, ainda terá entradas vazias (Patterson e Hennessy, 2014).

A performance final da memória pode ser influenciada pela escolha do tamanho do bloco, onde blocos maiores permitem reduzir a taxa de *misses* ao se aproveitar da localidade espacial, contudo podem também causar seu aumento, caso o tamanho do bloco tome uma fração significativa do tamanho da cache, uma vez que o número de blocos armazenados em cache diminui, ao passo que a competição de uso desses blocos aumenta. Todavia um problema mais sério associado ao aumento do tamanho do bloco é o agravamento da penalidade de uma falta de dados, determinada pelo tempo necessário para buscar o bloco do próximo nível da hierarquia, que contempla a latência para retornar a primeira palavra e o tempo para transferir o restante do bloco, e carregá-lo na cache. O resultado disso é a redução na performance geral da cache, causada quando o agravamento da penalidade de uma falta supera a redução na taxa de faltas para blocos muito grandes (Patterson e Hennessy, 2014).

### 3.2.1 Funções de mapeamento

Após buscar o bloco da memória principal, o controlador da cache deve encontrar um espaço onde irá inseri-lo. Quando cada palavra pode ser instalada em apenas um local na cache, geralmente baseando-se no endereço do dado, diz-se que a cache é diretamente mapeada. Nessa configuração é trivial encontrar a palavra, caso ela esteja na cache (Patterson e Hennessy, 2014).

Contudo, se um bloco pode ser associado com qualquer entrada da cache, diz-se que a cache é totalmente associativa, e para se encontrar um bloco em uma memória nesse esquema, todas as entradas devem ser procuradas, já que um bloco pode estar em qualquer local. Assim, para tornar essa busca viável, ela é feita em paralelo, com um comparador em cada entrada da cache. Esses componentes aumentam o custo do hardware significativamente, tornando esse mapeamento prático apenas para caches pequenas (Patterson e Hennessy, 2014).

Já uma cache associativa por conjuntos é aquela que consiste de um número de conjuntos de entradas de cache, onde o bloco pode ser instalado em qualquer um dos elementos. O conjunto a receber o bloco é selecionado pelo indexador, encontrado em um campo no endereço do dado requisitado, e quando cada um dos conjuntos desse tipo de cache possui $N$ locais de instalação de um bloco, diz-se que a cache é associativa por conjuntos de $N$ vias. Assim, essa função combina mapeamento direto com associatividade total (Patterson e Hennessy, 2014).

Todas as estratégias de instalação de blocos na cache podem ser vistas como variantes da associatividade por conjuntos. Uma cache diretamente mapeada é simplesmente uma cache associativa por conjuntos de uma via, ou seja, cada conjunto associativo possui apenas uma entrada. Já uma cache totalmente associativa de $M$ entradas é simplesmente uma cache associativa por conjuntos de $M$ vias, isto é, possui um único conjunto de $M$ entradas e um bloco pode residir em qualquer entrada desse conjunto (Patterson e Hennessy, 2014).

### 3.2.2 Políticas de escrita

Dada uma situação hipotética em que, durante uma instrução de escrita, o dado fornecido pelo processador é escrito somente na cache de dados, tem-se como resultado um estado onde a memória tem um valor diferente do presente na cache e, assim, diz-se que ambos estariam inconsistentes. Para manter o estado de consistência, o jeito mais simples é sempre escrever o dado tanto na memória principal quanto na cache, método chamado de *write-through* (Patterson e Hennessy, 2014).

Porém o bloco pode também ser alterado apenas na cache e ser escrito na memória quando deve ser substituído. Nesse caso, tem-se o método chamado de *write-back*, de implementação

mais complexa, porém com performance superior, principalmente quando o processador gera escritas mais rápido que a memória principal é capaz de suportar (Patterson e Hennessy, 2014).

É importante considerar o processo que ocorre para se alterar uma palavra na ocorrência de uma falta de escrita: o bloco é buscado da memória principal para que o dado que causou a falta possa ser alterado e, então, ser novamente gravado na memória para refletir a modificação. Apesar das escritas serem facilmente controladas por essa decisão, elas não são feitas de maneira rápida, dadas as limitações da memória principal; assim, para melhorar suas velocidades, usa-se um *buffer* de escrita, cuja função é armazenar o dado aguardando para ser escrito na memória, liberando o processador, depois que este escreve o dado na cache e no *buffer*, para continuar seu trabalho normalmente. Naturalmente, quando uma escrita na memória principal é concluída, a entrada correspondente no *buffer* é liberada, porém caso este esteja cheio, o processador sofre um *stall* até que haja espaço novamente (Patterson e Hennessy, 2014).

Ainda no caso de uma falta de escrita, a estratégia mais comumente adotada é alocar um bloco na cache, buscar seus dados da memória principal e alterar a porção apropriada, esquema que é chamado de *write-allocate*. Porém pode-se adotar, também, uma estratégia chamada de *no write-allocate*, que consiste em atualizar o bloco em memória, mas não trazê-lo à cache, sob a motivação de que programas as vezes escrevem blocos inteiros de dados, tornando a busca associada com o *write miss* desnecessária, o que acontece, por exemplo, quando o sistema operacional zera uma página na memória (Patterson e Hennessy, 2014).

Sob essas definições, nota-se que as politicas *write-back* e *write-allocate* trabalham bem juntas, pois são capazes de barrar parte das atualizações nas memórias cache de níveis inferiores. Por outro lado, as políticas *write-through* e *no write-allocate* apresentam a vantagem de atualizar mais rapidamente toda a hierarquia de memória ao propagarem as escritas.

### 3.2.3 Políticas de substituição

Quando há uma falta de dados na cache, algum bloco deve ser substituído por aquele que foi trazido da memória principal e será usado pelo processador. Em uma cache diretamente mapeada, o bloco requisitado pode ser instalado apenas em um único local, logo aquele que estiver ocupando a posição será substituído, mas quando há associatividade, deve-se escolher a "vítima"a deixar a cache e assim liberar espaço para receber o novo bloco. Em uma cache totalmente associativa, todos os blocos são candidatos a serem substituídos, enquanto em uma cache associativa por conjuntos, apenas os blocos de um conjunto podem ser escolhidos (Patterson e Hennessy, 2014).

Na sequência, encontram-se descrições do funcionamento dos algoritmos de substituição utilizados nesta monografia, suas possíveis implementações e vantagens e desvantagens, quando pertinente.

#### *3.2.3.1 Aleatório (**PRNGr** e **PRNGl**)*

Estes algoritmos dependem da geração de números pseudo-aleatórios para decidir o próximo bloco a deixar a cache, consumindo mais recursos a medida que seus geradores se tornam mais complexos. Geralmente possuem uma performance ruim ao não embasarem suas decisões em informações sobre uso, dependendo fortemente da aleatoriedade da sequência de instruções de memória e chegando a apresentar uma performance 22% pior daquela apresentada pelo LRU (Damien, 2007).

Nesta monografia, escolheu-se utilizar os geradores *round-robin* e LFSR, abreviados deliberadamente como **PRNGr** e **PRNGl**. Enquanto o primeiro se utiliza de um único contador, incrementado a cada acesso, para selecionar o bloco "vítima", o segundo gera seus números a

partir de uma abordagem semelhante a um registrador de deslocamento com realimentação linear (McGrath, 2021), que nada mais é que um registrador de deslocamento cujos bits de entrada são resultados de uma função linear entre dois ou mais estados anteriores (Cusick e Stanica, 2017).

#### *3.2.3.2 Primeiro a entrar, primeiro a sair (**FIFO**)*

Aproveitando-se do princípio da localidade de uma maneira simples, a implementação deste algoritmo se dá através de uma fila, logo os blocos são removidos na ordem em que foram inseridos na cache. Ainda que requeira menos tempo de computação e *hardware* para ser implementado, a maior taxa de faltas de dados em relação ao **LRU**, que pode se apresentar entre 12% e 20% acima da média deste, e o agravante da anomalia de Belady, fenômeno que ocasiona uma perda de performance quando se aumenta o tamanho da cache, se apresentam como uma grande desvantagem para a adoção deste algoritmo (Damien, 2007).

#### *3.2.3.3 Menos recentemente usado (**LRU**)*

Sendo uma das políticas mais comumente utilizadas (Patterson e Hennessy, 2014), o princípio de funcionamento deste algoritmo é substituir o bloco que está há mais tempo sem ser acessado, aproveitando-se da localidade de acesso. Damien (2007) cita que foi demonstrado um limite $p$ no número de faltas de cache quando comparado ao algoritmo ótimo, onde $p$ é proporcional ao tamanho do conjunto associativo. Uma implementação possível é rastrear quando cada elemento no conjunto foi utilizado relativamente aos outros, ainda que seja muito custoso de implementá-lo em hierarquias com altos graus de associatividade (Patterson e Hennessy, 2014).

#### *3.2.3.4 Pseudo menos recentemente usado baseado em árvore binária (**PLRUt**)*

Criado como uma aproximação do LRU, $N - 1$ bits por via, para uma cache associativa por conjuntos de $N$ vias, são necessários para representar uma árvore, cujos ramos indicam o bloco menos recentemente utilizado. Em um acerto, os bits que codificam o caminho à linha recém acessada são invertidos para indicar que o bloco menos recentemente usado não encontra-se mais nesse ramo da árvore, impedindo que o dado recém utilizado seja desalocado. Já na ocorrência de uma falta, os bits são invertidos somente se necessário (Damien, 2007).

#### *3.2.3.5 Pseudo menos recentemente usado baseado em bits MRU (**PLRUm**)*

Necessitando de um bit de status por linha de cache, a concepção deste algoritmo também foi como uma aproximação do LRU, porém cada bit indica a contemporaneidade de cada bloco. Assim, dado um acesso a um bloco da cache, o bit de status correspondente recebe o valor 1; se esse foi o último bit a receber 1 no conjunto associativo, todos os bits de status, menos o último a ser alterado, recebem o valor 0. Numa falta de cache, o elemento mais a esquerda que tenha seu bit de status igual a 0 será substituído (Abel e Reineke, 2020).

#### *3.2.3.6 Menos recentemente usado de quatro tempos (**QLRU**)*

Também chamado de predição do intervalo de re-referência por dois bits, dois bits de status por linha de cache são necessários para representar a idade de um bloco (Abel e Reineke, 2020). Infelizmente, mais detalhes das implementações mais comumente utilizadas nos processadores não está disponível, e, por isso, optou-se pelo uso da implementação apresentada no artigo de Jaleel et al. (2010).

#### *3.2.3.7 Ótimo (**MIN**)*

Proposto por Belady em seu estudo sobre algoritmos de substituição para computadores de memória virtual, este algoritmo é baseado na eliminação de conjuntos completos como candidatos à remoção, podendo ser classificado como um algoritmo guloso e sendo facilmente adaptado à pesquisa de políticas de substituição de linhas de cache. Esses conjuntos completos são constituídos de linhas de cache que serão utilizadas novamente num futuro próximo e, portanto, não podem ser desalocadas. Assim, sempre que um conjunto existe entre duas instruções consecutivas de memória, um novo estado é definido e todos os blocos restantes são substituídos, não importando a ordem em que chegaram na cache (Belady, 1966).

Uma vez que o conjunto associativo está cheio, deve-se coletar informações sobre novas referências à memória. Inicia-se um adiamento na seleção da "vítima"da cache quando um novo bloco deve ser trazido da memória, pois nenhuma linha instalada é uma candidata óbvia a deixá-la. Todavia, se um elemento do conjunto associativo é referenciado, ele deve permanecer instalado e é temporariamente desqualificado como candidato a substituição. Quando $c - 1$ blocos em cache forem desqualificados, a incerteza cai a zero e a "vítima"pode ser, então, substituída pelo novo bloco trazido da memória principal. Junto com o novo bloco, as $c - 1$ desqualificações formam um conjunto completo de $c$ elementos que definem um estado de memória (Belady, 1966).

A seguir, está representado, em pseudo-código, o processo realizado pelo algoritmo. O código é dividido em duas partes e, inicialmente, a variável `table` contém uma tabela *hash* vazia, que mapeia os endereços referenciados aos seus respectivos índices, enquanto as variáveis `complete_set`, `current_index` e `step` recebem os valores 1, 0 e 0, respectivamente. Além disso, a constante `ASSOCIATIVITY` contém o tamanho do conjunto associativo. Percebe-se que após a última referência à memória, a variável `current_index` possui o número mínimo de buscas à memória principal.

```
1  first_part:
2    auto address = data.address;
3    if (!table.find(address) or table[address] < complete_set)
4      current_index += 1;
5      table[address] = current_index;
6      goto first_part;
7
8    if (table[address] == current_index)
9      goto first_part;
10
11   if (table[address] < current_index and table[address] >= complete_set)
12     table[address] = current_index;
13     step = current_index;
14     complete_set = 0;
15     goto second_part;
```

```
second_part:
  for (auto entry : table)
    if (table[entry.value] == step)
      complete_set += 1;

  if (complete_set == ASSOCIATIVITY or step == complete_set)
    complete_set = step;
    goto first_part;

  if (complete_set < ASSOCIATIVITY)
    complete_set -= 1;
    step -= 1;
    goto second_part;
```

Na tabela 3.1 se encontra um exemplo de execução do algoritmo em uma cache associativa por conjuntos de 4 vias, onde a sequência **A**, **B**, **C**, **D**, **E**, **A**, **A**, **B**, **C** de referências a memória, presente no cabeçalho superior, serve de entrada, enquanto no cabeçalho lateral se encontram as entradas da tabela *hash* e as variáveis `complete_set` e `current_index`, representadas pelas letras **c** e **i**, respectivamente. Cada coluna contém os valores correspondentes de cada item da tabela *hash* e das variáveis após uma rodada de execução do algoritmo, com a última acomodando o conjunto completo detectado, que encontra-se sublinhado. Ao final da execução do exemplo, aqueles blocos não pertencentes ao conjunto completo seriam substituídos por aqueles que estariam aguardando uma decisão do algoritmo, ou seja, qualquer nova referência aguarda em uma fila até que o algoritmo encontre um novo estado de memória.

| | **?** | **A** | **B** | **C** | **D** | **E** | **A** | **A** | **B** | **C** |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | ? | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | **5** |
| **B** | ? | ? | 2 | 2 | 2 | 2 | 2 | 2 | 5 | **5** |
| **C** | ? | ? | ? | 3 | 3 | 3 | 3 | 3 | 3 | **5** |
| **D** | ? | ? | ? | ? | 4 | 4 | 4 | 4 | 4 | 4 |
| **E** | ? | ? | ? | ? | ? | 5 | 5 | 5 | 5 | **5** |
| **c** | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 4 |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 |

Tabela 3.1: Exemplo de execução do **MIN**.

### 3.2.4 Padrões de acesso à memória

Durante a execução de uma aplicação, o processador precisará acessar a memória a qualquer momento e mais de uma vez, seja para buscar novas instruções, seja para manipular os dados do programa. Esses acessos seguem padrões, que, ao serem entendidos, auxiliam na identificação dos casos que afetam, ou podem afetar, o desempenho de uma política de substituição de linhas de cache (Jaleel et al., 2010). A lista é incessante, havendo padrões muito mais complexos que os apresentados.

Assim, para a devida definição dos padrões, sejam $a_i$ o endereço de uma linha de cache, $(a_1, ..., a_k)$ uma sequência temporal de referências a $k$ endereços únicos e $(a_1, ..., a_k)^N$ uma sequência temporal que se repete $N$ vezes (Jaleel et al., 2010).

#### *3.2.4.1 Favorável a recência de uso*

Possuindo um intervalo de re-referência imediato, este padrão é facilmente representado por uma sequência $(a_1, ..., a_k, a_k, ..., a_1)^N$, sendo característico de acessos a estruturas baseadas em pilha. Para qualquer valor de $k$, o **LRU** se beneficia fortemente deste padrão de acesso (Jaleel et al., 2010).

#### *3.2.4.2 Batedura*

Do inglês *thrashing*, este padrão de acesso se caracteriza por uma sequência $(a_1, ..., a_k)^N$ e pode ser identificado durante laços de execução ou processamento de vetores. Ao contrário de outros algoritmos, o ótimo é capaz de preservar parte do conjunto de trabalho na cache, caso esta possua menos que $k$ entradas (Jaleel et al., 2010).

#### *3.2.4.3 Transmissão*

Ao se defrontarem com este padrão, conhecido também como *streaming*, nenhum algoritmo é capaz de preservar deliberadamente o conjunto de trabalho do processador, o que acarreta em taxas de acertos iguais ou próximas de 0. A sequência $(a_1, ..., a_k)$, com $k$ tendendo ao infinito, é inerente deste padrão (Jaleel et al., 2010), sendo vista, por exemplo, em bancos de dados cujo fluxo de acessos é muito grande.

#### *3.2.4.4 Misto*

Para este caso, parte das referências possuem intervalos curtos de repetição, enquanto outras possuem intervalos longos (Jaleel et al., 2010). Como o nome já indica, este é um padrão que mistura os outros três descritos anteriormente, estando presente em qualquer programa.

# 4 METODOLOGIA

Para a realização deste estudo, foram testadas, através de um simulador funcional de cache, diferentes combinações de configurações de cache e políticas de substituição. Criado na linguagem *C++*, o simulador é capaz de receber instruções de memória e contabilizar o número total de faltas de dados, porém, buscando isolar as políticas de substituição e, assim, obter resultados referentes apenas ao desempenho delas, a cache simulada conta com apenas um nível e nenhum campo para armazenamento de dados.

Na figura 4.1 encontra-se uma representação do funcionamento de uma das instâncias da política de substituição sendo testada no contexto da cache simulada, onde é possível ver que ela é responsável pelo gerenciamento de um conjunto associativo. Em uma cache real, uma instância do algoritmo é responsável apenas pela substituição de blocos de memória em todos os conjuntos associativos, mas nesta simulação, existem mais de uma instância do mesmo algoritmo e cada uma delas se torna efetivamente controladora de uma pequena cache totalmente associativa. Já o controlador da simulação fica incumbido de apenas dividir os bits de endereçamento e delegar sua execução à uma das várias instâncias da política de substituição.

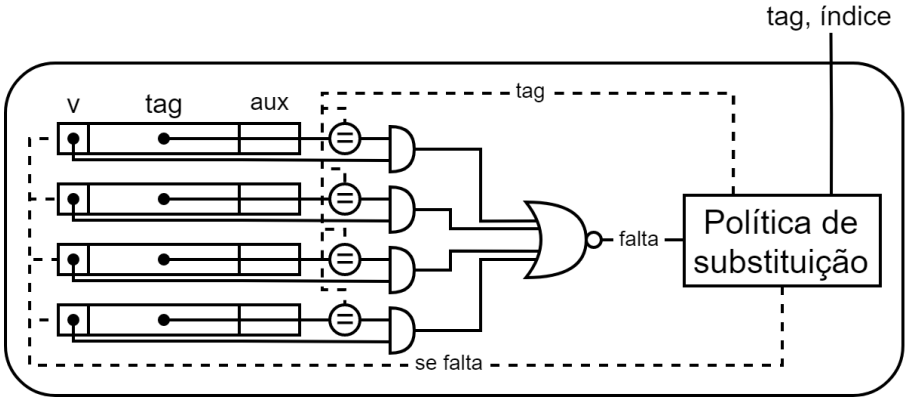


Figura 4.1: Visualização do funcionamento de uma instância da política de substituição.

As políticas selecionadas para este trabalho obedeceram os critérios de facilidade, frequência e independência de implementação em processadores, eliminando, assim, qualquer política dinâmica, isto é, qualquer política híbrida que, baseada em dados de uso dos blocos da cache, possa alternar entre duas ou mais políticas de substituição. Desta forma, os ensaios foram conduzidos nos algoritmos **PRNGr**, **PRNGl**, **FIFO**, **LRU**, **PLRUt**, **PLRUm** e **QLRU**, tendo como base de comparação o **MIN**. A seguir estão alguns detalhes da implementação deles.

- **PRNGr:** a pseudo-aleatoriedade deste algoritmo se dá através de um simples contador global, incrementado após o acesso a qualquer um dos conjuntos associativos;
- **PRNGl:** com seu gerador de números pseudo-aleatórios baseado em um registrador de deslocamento com realimentação linear, foi adotada a função *rand* da biblioteca *glibc*, implementada por McGrath (2021), com o tempo, em segundos, desde a era UNIX até o tempo da execução usado como semente;
- **FIFO:** a adoção de uma fila de duas pontas, encontrada na biblioteca padrão, possibilita inserções e remoções rápidas;

- **PLRUt:** por utilizar uma árvore binária, optou-se pelo mapeamento de uma árvore *heap* nos bits auxiliares das entradas do conjunto associativo;
- **PLRUm:** os bits necessários para seu funcionamento podem ser armazenados facilmente na própria estrutura do conjunto associativo;
- **QLRU:** tem sua implementação baseada na variação do SRRIP que prioriza acertos;
- **LRU:** cada entrada do conjunto associativo recebe um número que indica sua ordem em uma fila de prioridade;
- **MIN:** a tabela empregada para a indexação dos blocos se utiliza, neste projeto, de uma tabela *hash* implementada por Ankerl (2020) através do algoritmo Robin Hood.

O funcionamento do **MIN** foge à regra ao retornar o número mínimo possível de acessos à memória principal, isto é, sem efetivamente gerenciar uma estrutura de memória, o algoritmo é capaz de contabilizar o número mínimo de faltas possível. Segundo Belady (1966), é possível uma adaptação do algoritmo para se obter a sequência de substituições que devem ser feitas, contudo ela foge do escopo deste projeto, uma vez que não foi possível encontrar implementações documentadas.

As instruções de memória usadas como entrada para o simulador provém de traços de memória descompactados utilizando a biblioteca *Boost* e gerados a partir das cargas de trabalho da *suite* SPEC CPU2006, com o auxílio da ferramenta *PinPoints*, que correlaciona grupos de blocos básicos, chamados de vetores de blocos básicos, à aplicação inteira para identificar fases de inicialização e de comportamento periódico dentro do programa *single-threaded* (Zanata, 2014). Os traços de memória são divididos em um campo indicador de leitura ou escrita, tamanho da operação de memória, endereço de memória e número do bloco básico (Zanata, 2014), mas neste estudo, apenas o campo de endereço de memória é utilizado, uma vez que tanto leituras quanto escritas podem gerar faltas.

A SPEC CPU2006 foi projetada com o intuito de fornecer uma forma de medir e comparar a performance computacional intensiva de uma gama de hardwares, usando cargas de trabalho desenvolvidos a partir de aplicações reais (Corporation, 2018). Ela é utilizada como padrão de indústria para aferir a performance de sistemas em duas áreas: operações sobre inteiros e operações sobre pontos flutuantes (Henning, 2006). Nas tabelas 4.1 e 4.2 podem ser encontrados os *benchmarks* de cada área e uma breve descrição de cada um.

| | |
|---|---|
| 400.perlbench | Linguagem de programação PERL |
| 401.bzip2 | Compressão |
| 403.gcc | Compilador C |
| 429.mcf | Otimização Combinatória |
| 445.gobmk | Inteligência artificial (GO) |
| 456.hmmer | Busca de sequência de genes |
| 458.sjeng | Inteligência artificial (Xadrez) |
| 462.libquantum | Computação quântica |
| 463.xalancbmk | Processamento de XML |
| 464.h264ref | Compressão de vídeo |
| 471.omnetpp | Simulação de eventos discretos |
| 473.astar | Algoritmo de busca de caminho |

Tabela 4.1: Aplicações da SPEC CPU2006 usadas na medição de desempenho de sistemas em operações sobre inteiros. Fonte: Henning (2006)

| | |
|---|---|
| 410.bwaves | Dinâmica de fluídos |
| 416.gamess | Química quântica |
| 433.milc | Cromodinâmica quântica |
| 434.zeusmp | Física |
| 435.gromacs | Dinâmica molecular |
| 436.cactusADM | Relatividade geral |
| 437.leslie3d | Dinâmica de fluídos |
| 444.namd | Dinâmica molecular |
| 447.dealII | Análise de elementos finitos |
| 450.soplex | Programação linear |
| 453.povray | Ray tracing |
| 454.calculix | Mecânica estrutual |
| 459.GemsFDTD | Eletromagnetismo computacional |
| 465.tonto | Química quântica |
| 470.lbm | Dinâmica de fluídos |
| 481.wrf | Previsão de tempo |
| 482.sphinx3 | Reconhecimento de fala |

Tabela 4.2: Aplicações da SPEC CPU2006 usadas na medição de desempenho de sistemas em operações sobre pontos flutuantes. Fonte: Henning (2006)

Todas as políticas de substituição sofreram uma bateria de testes sob configurações de cache de 32 KiB e 2 vias, 32 KiB e 4 vias, 32 KiB e 8 vias, e 64 KiB e 8 vias, visando julgar o impacto dessas configurações na performance delas. Uma bateria de testes consiste de ensaios feitos sobre todo o conjunto de *benchmarks* da SPEC CPU2006, onde cada ensaio simula as referências feitas à memória por uma das aplicações da *suite*, dada uma combinação de configuração de cache e política de substituição. É importante destacar a remoção das aplicações *429.mcf*, *433.milc*, *450.soplex* e *470.lbm* das baterias de testes pela alta demanda de tempo para a conclusão dos experimentos realizados com eles, considerando que o tempo de execução é um fator relevante para a realização deste trabalho. Por fim, para que os resultados obtidos mostrem o desempenho de cada política e possam ser comparados entre si, as métricas utilizadas foram o número total e a taxa de faltas de dados, onde o menor valor representa a melhor performance.

# 5 RESULTADOS

Em cada figura apresentada neste capítulo, são expostos dois gráficos contendo os resultados dos testes conduzidos com uma dada configuração de cache, onde o gráfico superior dispõe os percentuais obtidos dos testes sobre aplicações de entrada baseadas majoritariamente em operações sobre inteiros, enquanto o inferior mostra as taxas dos ensaios conduzidos sobre aplicações baseadas majoritariamente sobre pontos flutuantes. Em todos eles, seus eixos horizontais são compostos pelas políticas testadas e categorizadas pelas aplicações de entrada, enquanto seus eixos verticais constituem-se das taxas de faltas computadas em cada teste.

Na figura 5.1, onde encontram-se os percentuais de faltas resultantes dos ensaios com uma cache de 32 KiB e 2 vias, é possível perceber que, em 64% das aplicações de entrada, todos os algoritmos apresentaram taxas de *misses* abaixo de 5%, destacando-se os testes sobre a aplicação *namd*, que mostraram resultados abaixo de 1%, e sobre a aplicação *libquantum*, cujas taxas estabilizaram-se na faixa de 8%.

O **MIN** apresenta uma melhoria de 16%, em média, frente ao **LRU** e suas aproximações, chegando a 29% nos testes realizados sobre a aplicação *zeusmp*. Contudo é interessante apontar que, apesar das baixíssimas taxas de faltas apresentadas pelos algoritmos nos testes realizados sobre a aplicação *namd*, o **MIN** ainda conseguiu apresentar ganhos acima de 10% sobre outras políticas neste mesmo ensaio, ou seja, mesmo em um cenário não muito favorável, o MIN ainda foi capaz de obter ganhos. Enquanto isso, os resultados obtidos da *gcc* não se mostraram tão promissores, já que os ganhos apresentados se mantiveram abaixo das mesmas políticas em outros testes, com exceção da *libquantum*.

Dado o tamanho reduzido dos conjuntos associativos, a taxa de faltas se mostra mais elevada nesta configuração, uma vez que há pouca possibilidade de escolha, percebendo-se uma menor margem para ganhos em relação à outras configurações.

Já em uma configuração de 32 KiB e 4 vias, com resultados dispostos na figura 5.2, os testes sobre as aplicações *GemsFDTD* e *zeusmp* apresentaram reduções relevantes nas taxas de faltas em comparação à configuração anterior, onde este expôs diminuições de até 3% e esse tem seus percentuais reduzidos em até 10,6%. Ainda comparando à uma cache de 32 KiB e 2 vias, o **LRU** apresentou um desempenho pior no teste *bwaves*, enquanto o **MIN** se aproveitou bem do maior número de vias no teste sobre a mesma aplicação, reduzindo em 30% o número de *misses*. Já o **QLRU** novamente desempenhou mal sobre a *GemsFDTD*, chegando a uma diferença de 1,6%, porém, em um ponto fora da curva, o algoritmo **PRNGl** apresentou uma taxa de faltas em torno de 4% menor neste mesmo teste.

Os ganhos do **MIN** passaram, em média, de 16% na configuração anterior para 22% nesta configuração, mostrando-se mais elevados. Sobre a *GemsFDTD*, especialmente, os ganhos se mostraram menores em políticas com escolhas melhor embasadas, com exceção do **QLRU**, onde o ganho se manteve alto.

A figura 5.3, que relaciona os resultados dos testes feitos com uma configuração de 32 KiB e 8 vias, mostra, como esperado, que o aumento na associatividade permitiu uma maior redução geral dos percentuais de faltas, com os resultados dos ensaios sobre a *bwaves* tendo uma expressiva redução. Além disso, é possível perceber um certo nivelamento dos resultados do **LRU** e de suas variantes, isto é, é possível perceber que os resultados produzidos por esses algoritmos são muito próximos, contudo o **QLRU** ainda apresenta um número de faltas mais alto.

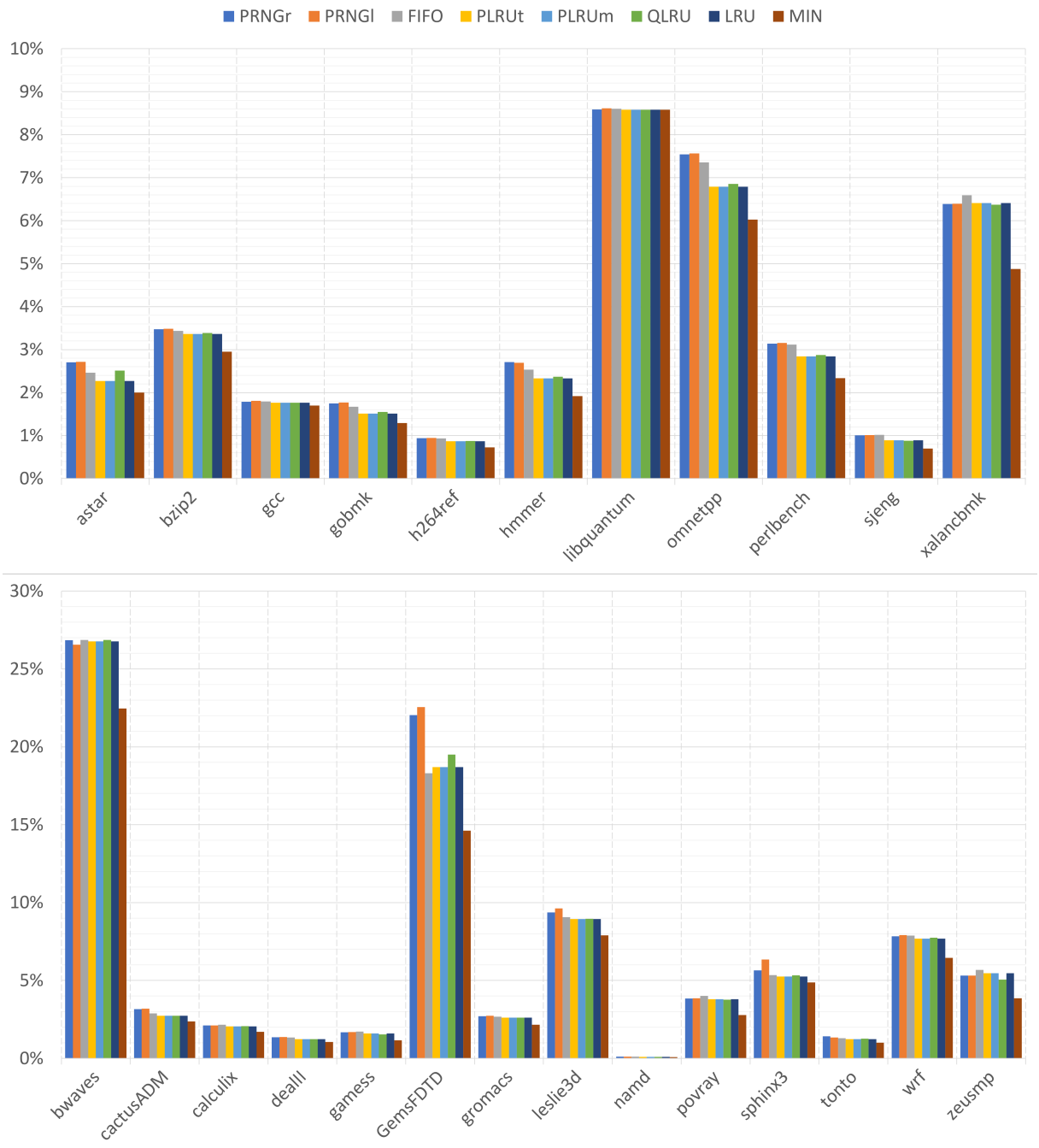


Figura 5.1: Taxas de misses obtidas dos testes em uma cache de 32 KiB e 2 vias.

O nivelamento dos percentuais fez com que os ganhos do **MIN** caíssem nos testes conduzidos sobre as aplicações *bwaves* e *GemsFDTD*, ficando abaixo de 1% no caso deste. Nos testes sobre outras aplicações, contudo, os ganhos podem chegar a 54%, sendo possível identificar uma alta em relação à configuração anterior.

Para uma configuração de 64 KiB e 8 vias, as reduções nas taxas de faltas já não se mostraram tão expressivas quando comparadas àquelas obtidas de configurações anteriores, mas destaca-se percentuais abaixo de 2% em grande parte das políticas testadas sobre 16 das 25 aplicações, como pode ser visto na figura 5.4. Ainda nesta configuração, constata-se maiores ganhos do **MIN** sobre políticas com um processo de decisão pouco informado, como o **PRNGr**, o **PRNGl** e o **FIFO**, com este último chegando a 71%. Todavia os ganhos sobre outras políticas

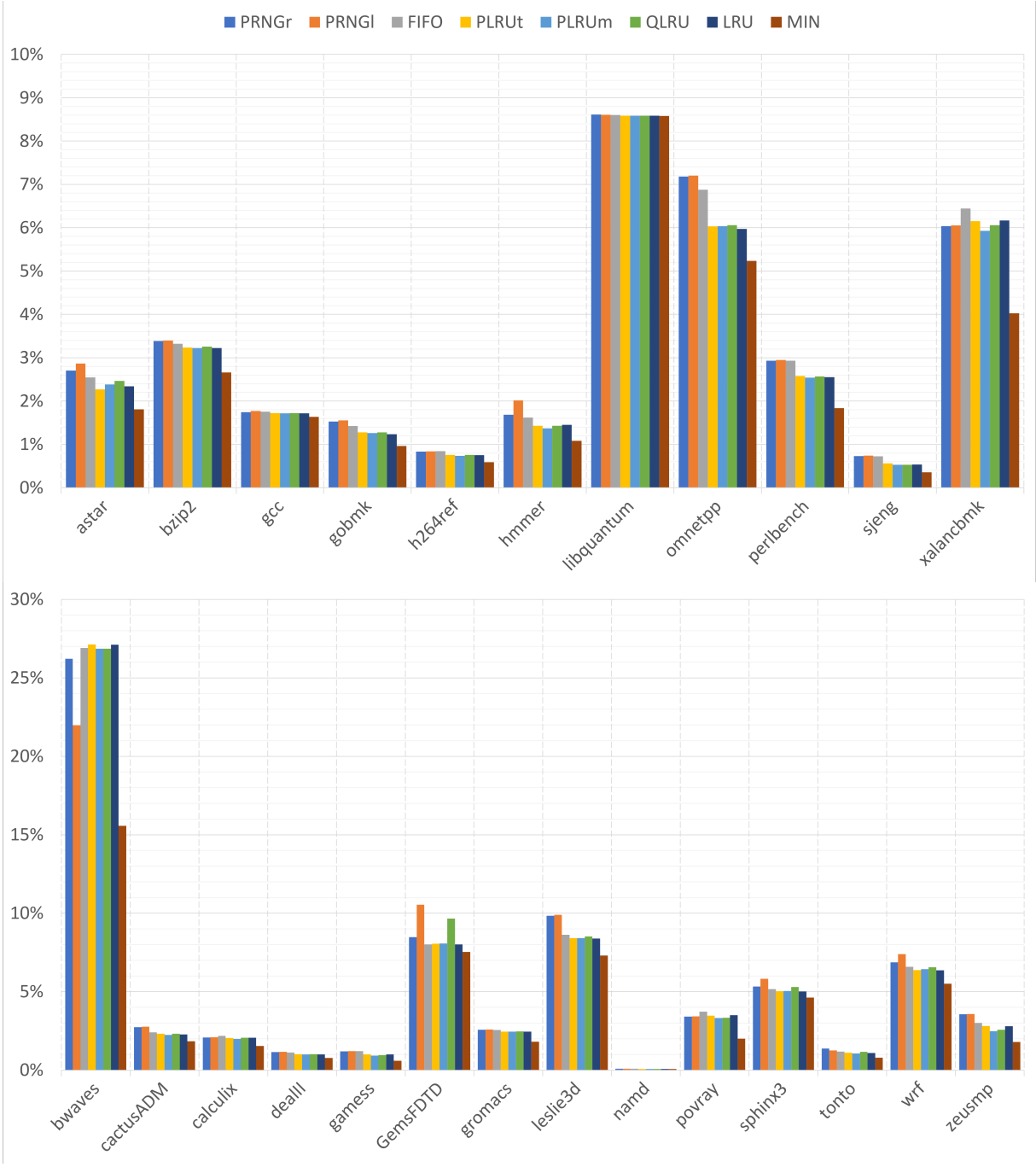


Figura 5.2: Taxas de misses obtidas dos testes em uma cache de 32 KiB e 4 vias.

parece variar de aplicação para aplicação, pois enquanto o **LRU** mostra resultados mais próximos do ótimo no teste sobre a aplicação *zeusmp*, o mesmo algoritmo se distancia no teste sobre a *gromacs*.

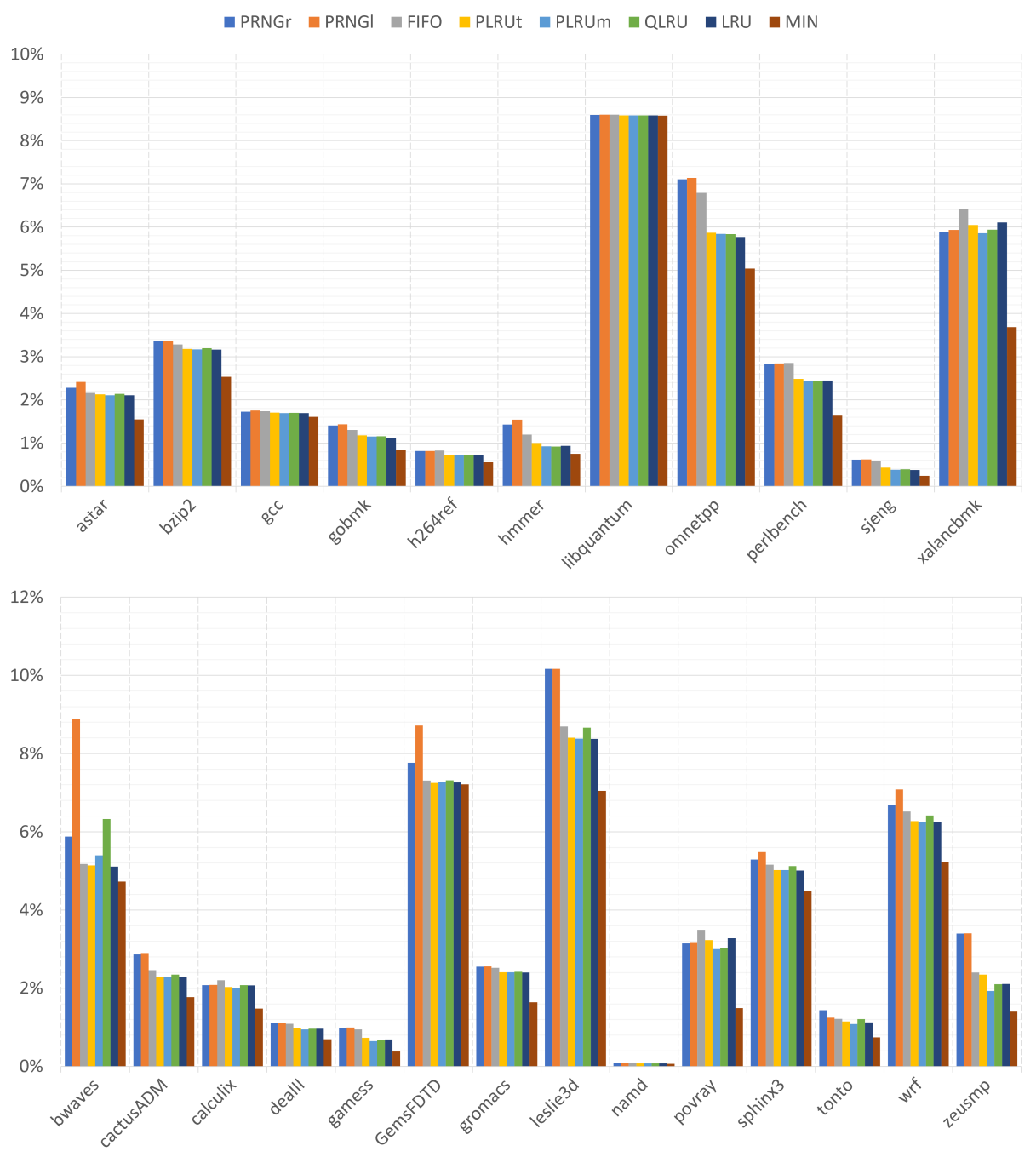


Figura 5.3: Taxas de misses obtidas dos testes em uma cache de 32 KiB e 8 vias.

Finalmente, encontram-se, na figura 5.5, as taxas médias de faltas de todas as políticas de substituição em cada uma das configurações de cache, onde destaca-se o percentual médio do **MIN** abaixo de 2,5% em todas as configurações testadas, bem como a proximidade de desempenho do **LRU** e suas aproximações. O gráfico em linha, presente no eixo secundário, mostra as médias globais de cada política, sendo possível perceber melhorias de 22,6%, 20,9%, 22,6% e 21,6% na média do **MIN** frente às políticas **PLRUt**, **PLRUm**, **QLRU** e **LRU**, respectivamente, mas chegando a 32% frente àquelas pouco ou não informadas.

Em uma visão geral, nota-se uma performance inconsistente, e consequentemente inferior, do **PRNGl** em relação ao **PRNGr**, provavelmente fundamentada na maior aleatoriedade dos números gerados por um LFSR. Além disso, o **QLRU** mostrou um pior desempenho, na

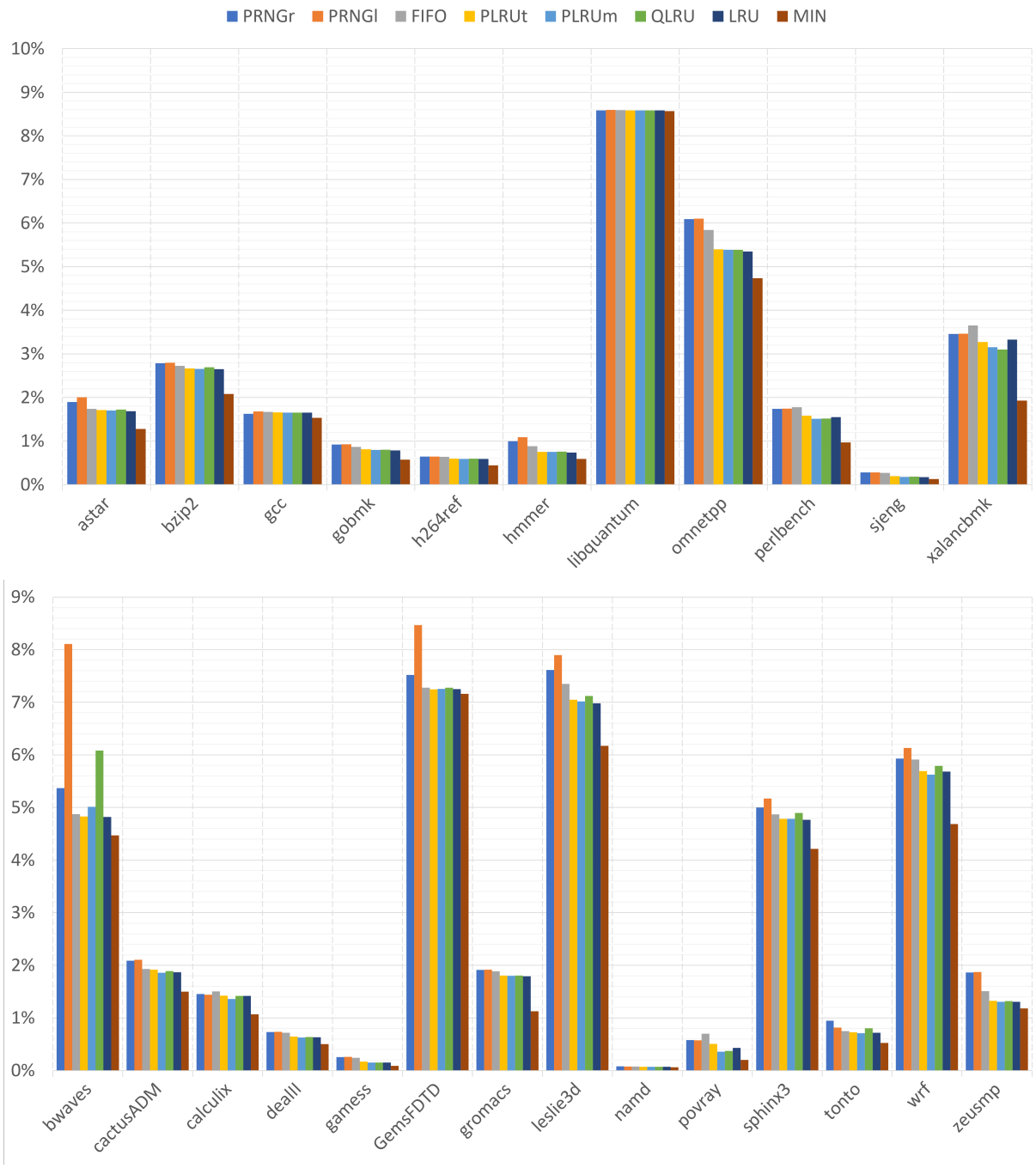


Figura 5.4: Taxas de misses obtidas dos testes em uma cache de 64 KiB e 8 vias.

média, que o **LRU** e outras aproximações, sendo plausível crer que a razão pode estar na sua ineficiência em lidar com o padrão *thrashing* de acesso à memória. Outro ponto interessante a destacar é a performance levemente superior do **PLRUm** frente ao **PLRUt**, chegando a ultrapassar até mesmo o **LRU** em alguns cenários, e sendo, assim, possível corroborar a constatação de Al-Zoubi et al. (2004), quando apontam que esta política apresenta uma melhor relação custo por benefício.

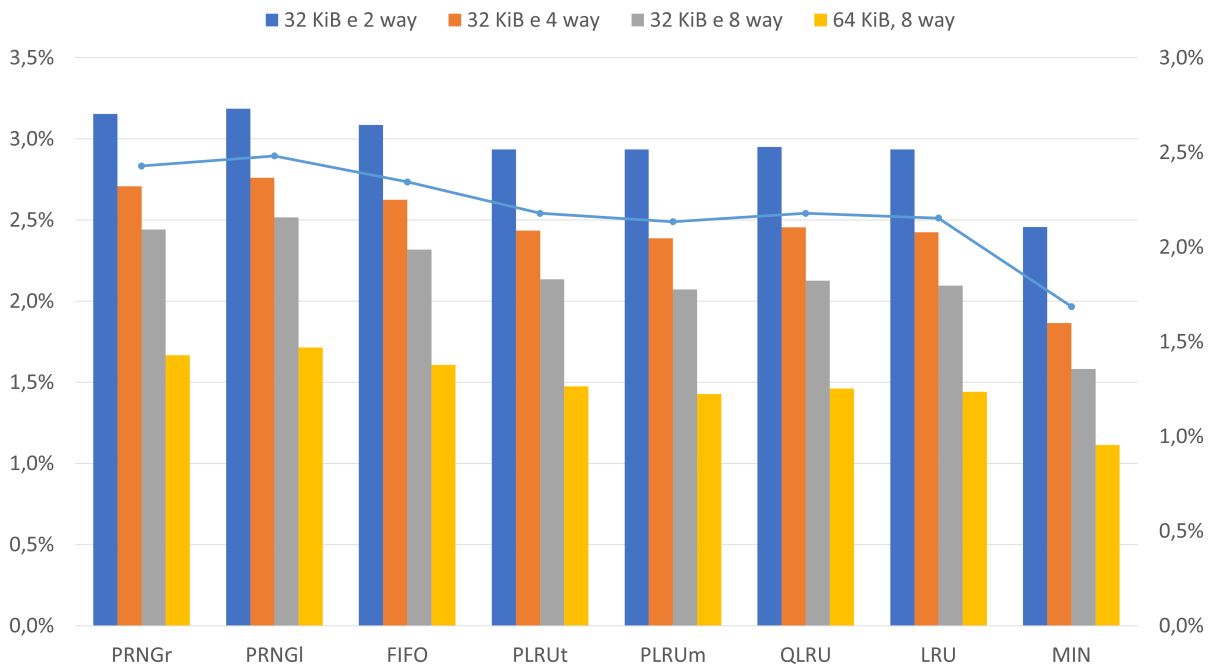


Figura 5.5: Médias geométricas das taxas de misses obtidas de cada bateria de testes.

# 6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho buscou-se determinar a diferença de performance entre políticas de substituição de linhas de cache comumente utilizadas em processadores modernos e uma política ótima, isto é, uma política capaz de sempre retornar a melhor solução, escolhendo, dentre os blocos instalados na cache, sempre aquele que ficará por mais tempo sem uso. Para tal, foi desenvolvido um simulador funcional de cache capaz de determinar a taxa de faltas de uma aplicação de entrada, dada uma política de substituição e uma configuração de cache.

Com os percentuais de faltas em mãos, percebeu-se que o MIN apresenta ganhos acima de 10% sobre todas as políticas em pelo menos 80% das aplicações, chegando a 53% sobre a LRU na aplicação *povray*, para uma configuração de cache de 64 KiB e 8 vias. Constatou-se também que um maior grau de aleatoriedade não parece conduzir a melhores tomadas de decisão, o que parece ter sido a causa da performance inferior do PRNGl frente ao PRNGr, além da performance abaixo do esperado apresentada pelo QLRU, indicando que o algoritmo pode se beneficiar de ajustes mais finos. Apesar disso, a maioria dos resultados obtidos mostram taxas abaixo de 5%, com os piores resultados sendo mitigados a cada aumento na associatividade e tamanho da cache.

Desta forma, pode-se concluir, pelos resultados obtidos durante este estudo, que ainda há grandes margens para melhorias, principalmente naquelas configurações de cache mais comumente encontradas em processadores modernos, como as de 32 KiB de capacidade e associatividade igual a 8. Contudo é incerto qual o impacto que uma política ótima de substituição de linhas de cache exerce no desempenho final de um sistema, uma vez que diversos mecanismos, como a computação fora de ordem e a pré-busca de dados, são usados, hoje, com o intuito de esconder a latência das memórias. Com isso, e considerando que o custo de implementação de soluções mais avançadas continua uma incógnita, torna-se impossível determinar, no escopo deste trabalho, se podem existir políticas viáveis e melhores.

Ao se considerar o que foi mostrado nesta monografia, contempla-se como possíveis caminhos para trabalhos futuros o desenvolvimento de uma política que faça uso de uma inteligência artificial, bem como testar qual seria o impacto na performance final de um sistema caso uma política ótima fosse possível. Além disso, pode-se considerar também uma avaliação da relação custo por benefício que soluções mais avançadas podem apresentar, uma vez que se desconhece o custo de implementação delas, ainda que exista margem para melhoria. Por fim, um estudo como o desta monografia, mas mais abrangente e que inclua políticas mais avançadas a serem testadas, pode ser proveitoso também.

# REFERÊNCIAS


Abel, A. e Reineke, J. (2020). nanobench: A low-overhead tool for running microbenchmarks on x86 systems. Em *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Al-Zoubi, H., Milenkovic, A. e Milenkovic, M. (2004). Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. Em *Proceedings of the 42nd Annual Southeast Regional Conference*, páginas 267–272, Nova Iorque, NY - Estados Unidos da América.

AMD (2017). Software optimization guide for amd family 17h processors. Relatório técnico, Advanced Micro Devices.

Ankerl, M. (2020). Robin hood hashing. `https://github.com/martinus/robin-hood-hashing`. Versão 3.6.0 de 14 de março de 2020.

Belady, L. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101.

Corporation, S. P. E. (2018). Spec cpu2006. `https://www.spec.org/cpu2006/`. Acessado em 21 de janeiro de 2021.

CRC (2017). The 2nd cache replacement championship. `https://crc2.ece.tamu.edu/`. Acessado em 02 de março de 2021.

Cusick, T. e Stanica, P. (2017). *Cryptographic Boolean Functions and Applications*, páginas 7–29. Elsevier.

Damien, G. (2007). Study of different cache line replacement algorithms in embedded systems. Dissertação de Mestrado, KTH Royal Institute of Technology, Estocolmo - Suécia.

Henning, J. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Jain, A. e Lin, C. (2017). Hawkeye: Leveraging belady's algorithm for improved cache replacement. Em *The 2nd Cache Replacement Championship (CRC)*, Toronto, Ontario - Canadá.

Jaleel, A., Theobald, K., Steely, S. e Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3):60–71.

McGrath, R. (2021). The gnu c library. `https://www.gnu.org/software/libc/`. Versão 2.33.9000 de 01 de fevereiro de 2021.

Patterson, D. e Hennessy, J. (2014). *Computer Organization and Design: The Hardware/Software Interface*, páginas 374–499. Elsevier.

Zanata, M. (2009). Avaliação do compartilhamento das memórias cache no desempenho de arquiteturas multi-core. Dissertação de Mestrado, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS - Brasil.

Zanata, M. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. Tese de doutorado, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS - Brasil.