CacheLab: um Simulador Funcional de Memórias Cache

Hugo Jeller Ferreira[1], **Liana Duenha**[1] **(Orientadora)**

[1]Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande - MS - Brasil

{hugo.ferreira,liana.duenha}@ufms.br

Abstract. *This paper presents CacheLab, a functional cache memory simulator developed to support the teaching of Computer Architecture in undergraduate Computer Science courses. In addition to allowing the parameterization of cache designs at various levels, the difference between this simulator and others with the same purpose is that CacheLab simulates cache behavior using memory traces generated from real applications, instead of simulating accesses to random addresses. The tool is optimized to allow low resource utilization during simulations, allowing the evaluation of caches with virtually unlimited size.*

Resumo. *Este artigo apresenta o CacheLab, um simulador funcional de memórias cache, desenvolvido para fins de apoio didático ao ensino de Arquitetura de Computadores em cursos de graduação em Computação. Além de permitir a parametrização do projeto de caches em vários níveis, o diferencial desse simulador, quando comparado a outros que tem a mesma finalidade, é que o CacheLab simula o comportamento de caches sobre traces de memória gerados a partir de aplicações reais, ao invés de simular acessos a endereços aleatórios. A ferramenta tem otimizações que permitem baixa utilização de recursos durante as simulações, permitindo a avaliação de caches com tamanho virtualmente ilimitado.*

1. Introdução

Nos primeiros sistemas computacionais, estavam presentes apenas dois tipos de memória: a memória primária ou principal (memória RAM), responsável pelo armazenamento temporário dos dados e das instruções dos programas em execução, e a memória secundária, destinada ao armazenamento permanente, geralmente representada por discos ou fitas magnéticas. Com o avanço das tecnologias de fabricação de processadores, a disparidade entre a velocidade de processamento e a velocidade com que os dados eram acessados e atualizados nas memórias tornou-se um fator limitante para o desenvolvimento dos sistemas computacionais modernos.

Em resposta a essa limitação, na década de 1960, foi introduzido o conceito de uma memória de acesso mais rápido [Liptay 1968], denominada memória cache, projetada com tecnologias avançadas para melhorar a eficiência na troca de dados entre processador e o sistema de armazenamento. Localizada entre o processador e a memória principal, essa memória tem a função de armazenar e fornecer ao processador os dados e as instruções mais frequentemente utilizados, reduzindo, assim, a necessidade de

comunicação com as camadas mais lentas de memória e, consequentemente, aprimorando o desempenho geral do sistema.

Ao longo dos anos, as memórias cache tornaram-se amplamente populares, mais acessíveis em termos de custo e passaram a ser componentes essenciais em praticamente todos os sistemas que processam dados, desde grandes servidores até processadores embarcados de baixa capacidade. Atualmente, é comum que processadores pessoais e aparelhos de telefonia móvel possuam dois ou mais níveis de cache, com capacidades que podem variar de algumas unidades até dezenas de megabytes.

No contexto das disciplinas de Arquitetura de Computadores, oferecidas em cursos de graduação nas áreas de Computação (como Ciência da Computação, Sistemas de Informação, Engenharia de Computação, Engenharia de Software, entre outros), abordam-se tópicos sobre os diferentes modelos de projetos de cache, políticas de escrita, estratégias de substituição de dados e métricas de desempenho do sistema de memória. Para a consolidação dos conceitos, é comum que os docentes utilizem representações gráficas dos projetos de cache e realizem operações sobre as caches ciclo a ciclo. Essas representações, muitas vezes, são de grande escala, o que demanda um esforço considerável por parte dos docentes, consumindo uma parte significativa do tempo disponível nas aulas. Além disso, a natureza dinâmica das caches exige a execução de várias operações para que os alunos consigam visualizar as vantagens de se utilizar caches no sistema de memória.

Apesar de todo o esforço envolvido, os exemplos ilustrativos por meio de desenhos no quadro ou apresentações em *slides* frequentemente não são suficientes para que os alunos compreendam as métricas de desempenho das caches, como taxas de acerto, taxa de falha e o tempo médio de acesso à memória. De fato, a exploração do impacto de diferentes projetos de cache no desempenho do sistema só é possível após um número considerável de acessos à cache. Por exemplo, nos primeiros acessos à cache, é esperado que ocorram muitas falhas de cache, o que pode levar a uma taxa de falhas muito alta. Se o desempenho da cache for avaliado após apenas alguns acessos, pode-se gerar a impressão equivocada de que o projeto da cache é ineficiente.

Para superar essa limitação, é comum que os docentes recorram a simuladores de memória como ferramenta de apoio ao processo de ensino-aprendizagem. Embora existam diversos simuladores de memória disponíveis, a maioria deles é mais adequada para fins de pesquisa do que para aplicações didáticas, devido à sua complexidade, foco técnico, instalação e interface não amigáveis.

Dessa forma, o presente trabalho tem como objetivo apresentar o CacheLab, um simulador funcional de memória cache, projetado como ferramenta de apoio no ensino de Arquitetura de Computadores e disciplinas correlatas nos cursos da área de Computação. As principais funcionalidades e contribuições deste simulador são as seguintes:

- Permite avaliar *traces* de memória gerados a partir de aplicações reais, ao invés de simular acessos a endereços aleatórios.
- Possibilita que o usuário crie projetos funcionais de caches associativas por conjunto com um ou mais níveis, parametrizando o tamanho dos blocos, a quantidade de blocos e a associatividade de cada uma delas.
- Permite que o usuário escolha entre diferentes políticas de substituição de blocos.

- Permite setar parâmetros de desempenho como *tempo de acerto* ou *hit time* e *penalidade de falha* ou *miss penalty* de cada cache do sistema para fins de exploração de desempenho das memórias.
- Tem suporte à simulação ciclo a ciclo, muito útil para fins didáticos, ou execução completa, mais adequada para análise de desempenho e comparação de distintos projetos de caches.
- Permite visualização da estrutura da cache configurada de forma similar à apresentadas em livros didáticos.
- Possui interface amigável e acesso pela web, dispensando o árduo processo de instalação dependente de compilador, sistema operacional e capacidade técnica do usuário.
- Processamento altamente otimizado para baixa utilização de recursos durante simulações, permitindo simulação de caches com tamanho virtualmente ilimitado.

Este artigo está organizado como segue: a Seção 2 apresenta alguns simuladores de cache que têm relação com as funcionalidades do CacheLab; a Seção 3 oferece a fundamentação teórica necessária para compreensão das funcionalidades implementadas no simulador; a Seção 4 apresenta a metodologia de desenvolvimento do CacheLab e suas principais funcionalidades; a Seção 5 demonstra alguns casos de uso do simulador; por fim, a Seção 6 conclui este artigo.

2. Trabalhos Relacionados

Na revisão bibliográfica realizada durante o desenvolvimento do CacheLab, foram encontrados alguns simuladores de caches com características similares. Esta seção apresenta esses simuladores, suas principais funcionalidades e limitações.

2.1. Simulador Beto

O simulador de memórias cache denominado *Beto* [LeandroGabriel.net 2022] tem em seu nome uma homenagem ao Dr. Robert H. Dennard, inventor da DRAM. O sistema web foi desenvolvido em JavaScript e permite customização de hardware e software, abrindo um leque de possibilidades para testes e demonstrações do funcionamento da cache. Nas configurações de memória, é possível configurar o tamanho das células e blocos, número de células e linhas, tamanho do conjunto associativo, método de substituição (FIFO, LRU e Random) e método de mapeamento (direto, totalmente associativo e associativo por conjunto). Beto possui uma interface de fácil compreensão, com bom uso das cores e com animações que dão fluidez à simulação de acesso à cache. A área de acessos à memória entrega uma visualização didática que, junto à funcionalidade de acompanhamento do passo a passo, auxilia no estudo de como é e quando acontece um hit ou miss na cache.

A maior limitação do Beto é que os endereços de memória que serão acessados devem ser fornecidos diretamente pelo usuário, utilizando números inteiros, o que não permite explorar o comportamento das caches com base em acessos reais de memória (gerados a partir de aplicações reais).

2.2. Simulador da Universidade da Califórnia

O simulador de cache criado na Universidade da Califórnia [Wuterich] é uma ferramenta fornecida pela universidade como recurso para apoio ao aprendizado. É possível adicionar

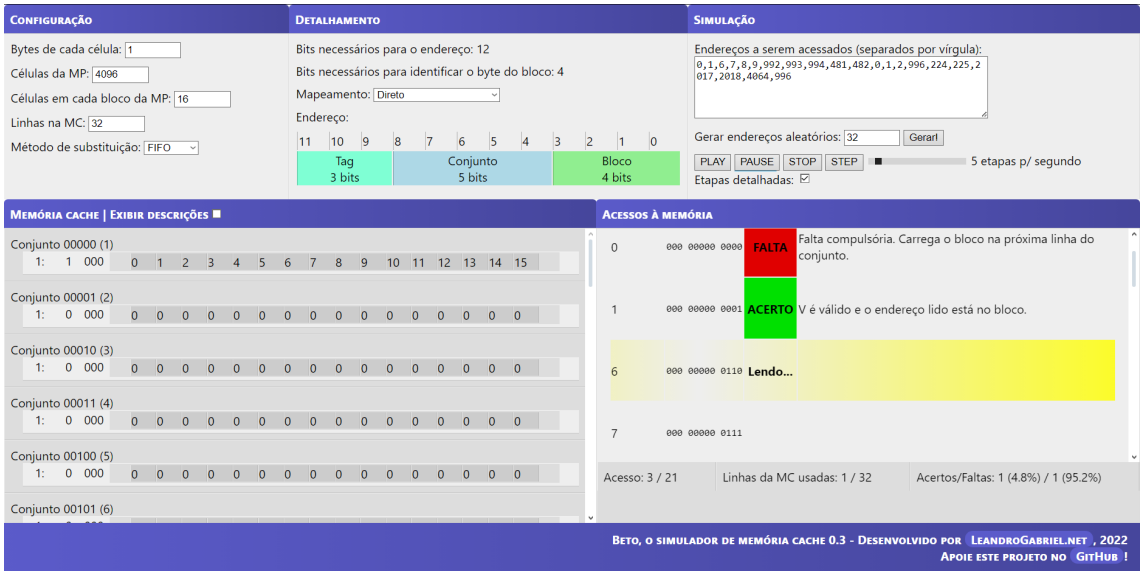

Figura 1. Visão geral do simulador Beto.

caches e configurar a quantidade de blocos, tamanho dos blocos, tamanho do conjunto e tempo de acesso individualmente. Há um campo para visualização geral da memória com estatísticas sobre o desempenho da cache, como taxa de acerto e tempo médio de acesso à memória. No entanto, todos os blocos são mapeamento associativo de duas vias, sem nenhuma maneira de alterar e testar outras configurações de mapeamento, e há um erro no funcionamento do botão de remover blocos de cache. A interface, apesar de simples, consegue ser eficiente na apresentação do conteúdo das caches (Figura 2).

Da mesma maneira que o simulador Beto, o simulador da Universidade da Califórnia necessita de endereços fornecidos manualmente pelo usuário.

2.3. Simulador CSE351 da Universidade de Washington

A Universidade de Washington tem em seu material didático um simulador de cache para apoio ao aprendizado no curso de Interface de Hardware e Software [Washington]. Esse simulador apresenta opções de configuração de sistema como tamanho dos endereços, tamanho da cache, tamanho do bloco, associatividade, política de escrita e método de substituição. Para auxiliar na visualização da simulação e na explicação, o site mostra histórico de acessos e algumas mensagens sobre o que está acontecendo na simulação. A seleção do campo “Explain” providencia mais clareza sobre as ações realizadas pelo simulador, executando passo a passo dos processos e auxiliando o usuário na compreensão de como o sistema está funcionando (Figura 3).

Da mesma maneira que os simuladores apresentados anteriormente, o simulador da Universidade de Washington necessita de endereços fornecidos manualmente pelo usuário, em formato hexadecimal.

2.4. Vantagens do CacheLab

Os simuladores apresentados nesta seção têm funcionalidades similares ao CacheLab e têm como finalidade de apoio didático ao ensino de arquitetura de computadores durante cursos de graduação em computação. Eles também têm suporte a projetos de cache com

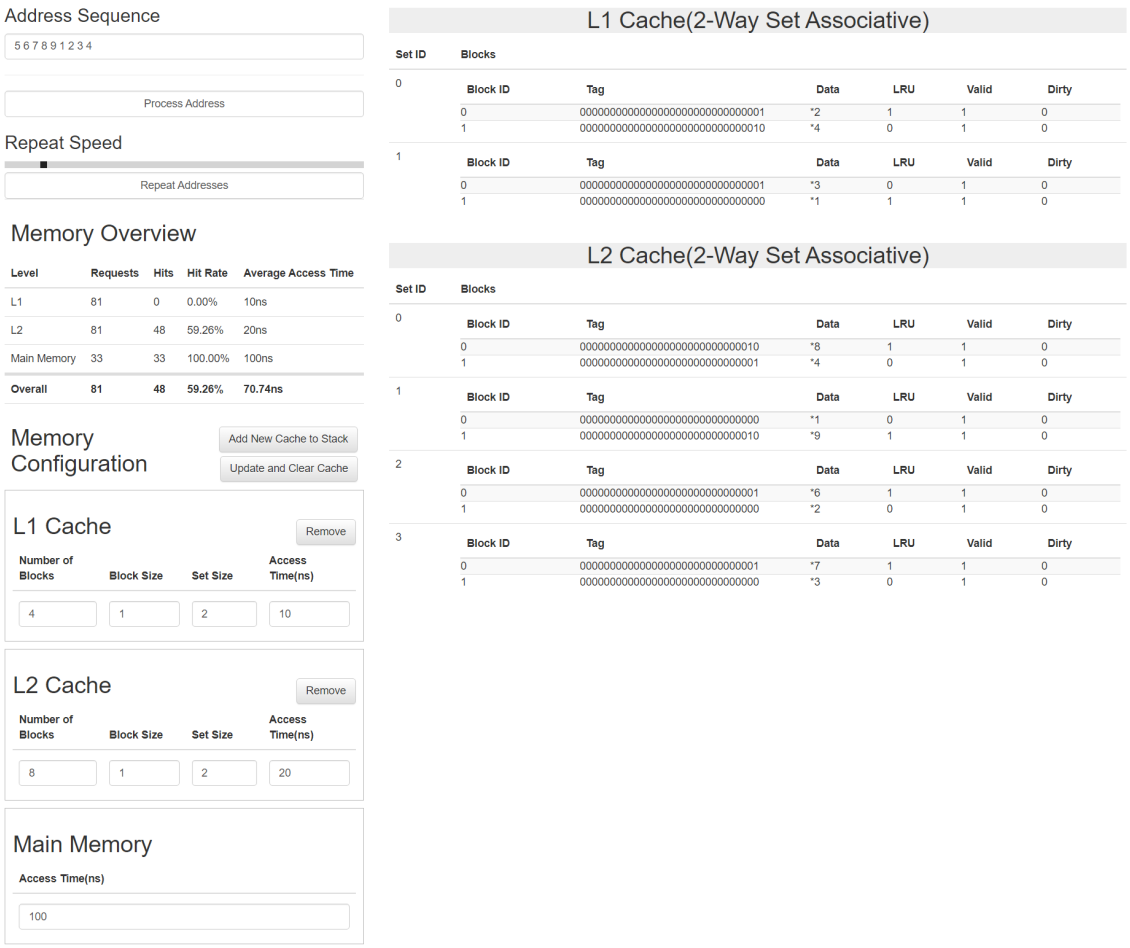

Figura 2. Visão geral do simulador da Universidade da Califórnia.

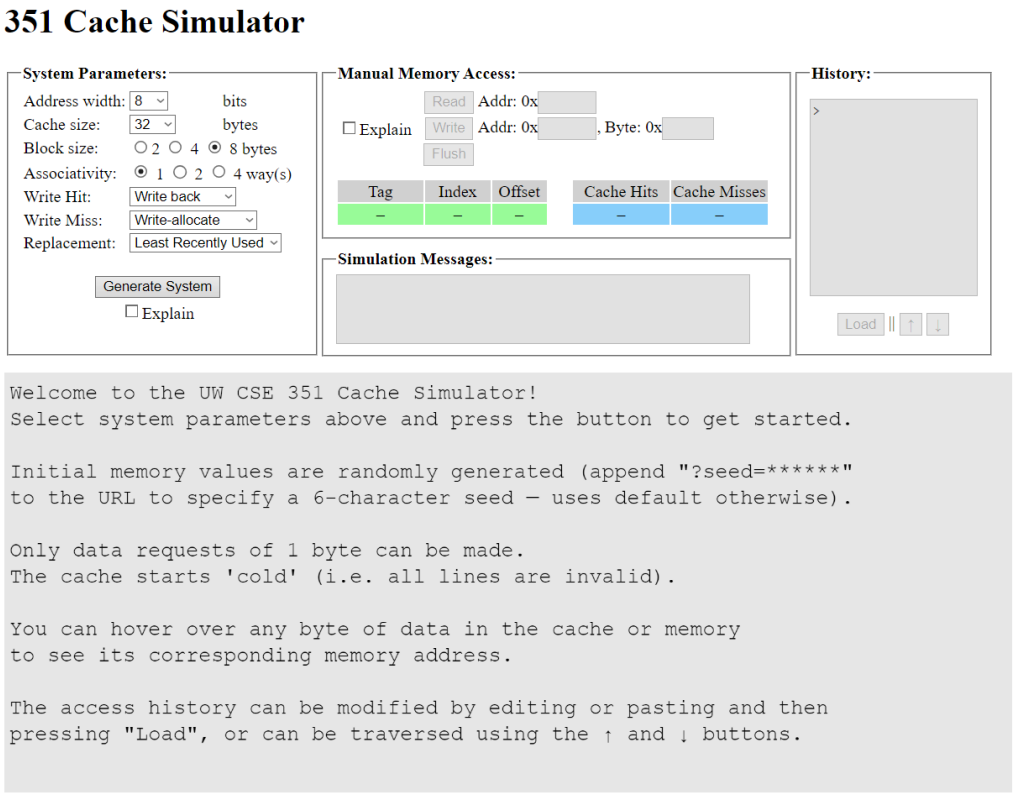

Figura 3. Visão geral do simulador da Universidade de Washington.

parametrização de dados do projeto como tamanho de blocos, associatividade e políticas de substituição de blocos.

Entretanto, a forma como recebem os dados de entrada difere significativamente. Os simuladores apresentados necessitam de requisições de memória geradas manualmente, o que descaracteriza completamente as entradas de caches reais, pois não tiram vantagem do princípio da localidade, ou necessitam que as entradas sejam carregadas uma a uma, o que é ainda menos adequado para avaliação de projetos de cache. Em contrapartida, o CacheLab permite a simulação do projeto de caches com base em *traces* de memória gerados a partir de aplicações reais.

Ao permitir o tratamento de milhares de acessos à memória, foi necessário realizar diversas otimizações para garantir que o desempenho do simulador não desmotivasse o usuário na execução completa para fins de avaliação do desempenho final do sistema de caches.

2.5. Comparações com Outros Simuladores

A comparação direta do CacheLab com outros simuladores não é possível pela diferença fundamental na forma que lidam com endereços de memória. Simuladores desenvolvidos em JavaScript estão sujeitos às limitações do padrão IEEE 754 [IEEE 2019], que limita números inteiros a 53 bits de precisão.

Essa restrição impede que simuladores aqui citados processem adequadamente *traces* de memória de aplicações reais, que exploram todo o espaço de endereçamento de

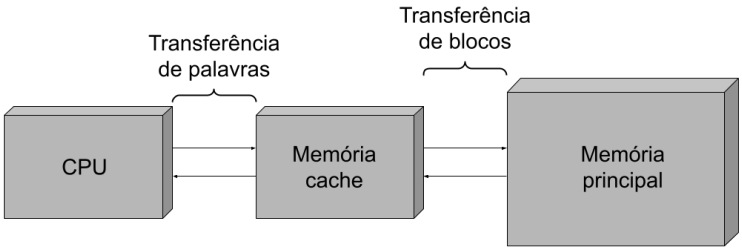

Figura 4. Unidades de Transferência de Dados entre Processador, memória cache e memória principal. Reproduzido a partir de [Stallings 2010].

64 bits. O CacheLab contorna essa limitação fazendo uso extensivo do tipo `BigInt`, que pode representar inteiros com precisão arbitrária.

3. Fundamentação Teórica

A memória cache é uma memória rápida que se beneficia dos princípios da localidade espacial e da localidade temporal para aumentar a velocidade de acesso aos dados e instruções de um programa pelo processador [PARHAMI 2007]. A ideia da localidade temporal reside no fato de que se um dado é referenciado em um dado momento, é muito provável que em breve ele será referenciado novamente. Dados acessados no controle ou no corpo de laços de repetição ou de funções recursivas são exemplos da presença da localidade temporal. O princípio da localidade espacial é bastante presente nos acessos a estruturas de dados como tabelas e vetores, visto que quando um determinado dado é acessado, há uma grande chance de que dados próximos a ele sejam acessados em um futuro próximo. Estes princípios são propriedades inerentes aos programas de computador e os projetos de cache podem beneficiar-se destas propriedades para alcançar melhor desempenho [BAER 2010].

O espaço de endereçamento dos programas em execução é relativo à memória principal. Quando o processador faz uma requisição de leitura ou de escrita de um dado, ele utiliza um endereço de uma *palavra* de memória para referenciá-lo. Uma *palavra* é uma sequência de bytes definida em tempo de projeto, que limita o tamanho dos barramentos do sistema, o tamanho dos registradores, a quantidade de bits dos endereços de memória, entre muitos outros componentes do projeto do sistema de computação.

Ao ser adicionada uma ou mais memórias cache entre o processador e a memória principal, introduz-se o conceito de *bloco* como unidade de transferência de dados entre esses níveis de memória. Um *bloco* é uma sequência fixa de palavras, definida em tempo de projeto do sistema de memória, junto de campos para controle do estado do bloco como bit de validade e tag. A Figura 4 ilustra a unidade de transferência de dados entre processador e memória cache (em palavras) e entre a memória principal e a memória cache (em blocos).

A Figura 5 representa a estrutura de um sistema de memória com uma memória principal e uma memória cache. A memória principal é constituída de 2^n palavras endereçáveis, cada qual com um endereço distinto de n bits. Considere que, nesse sistema, um bloco é definido por um conjunto de K palavras. Desta forma, podemos considerar

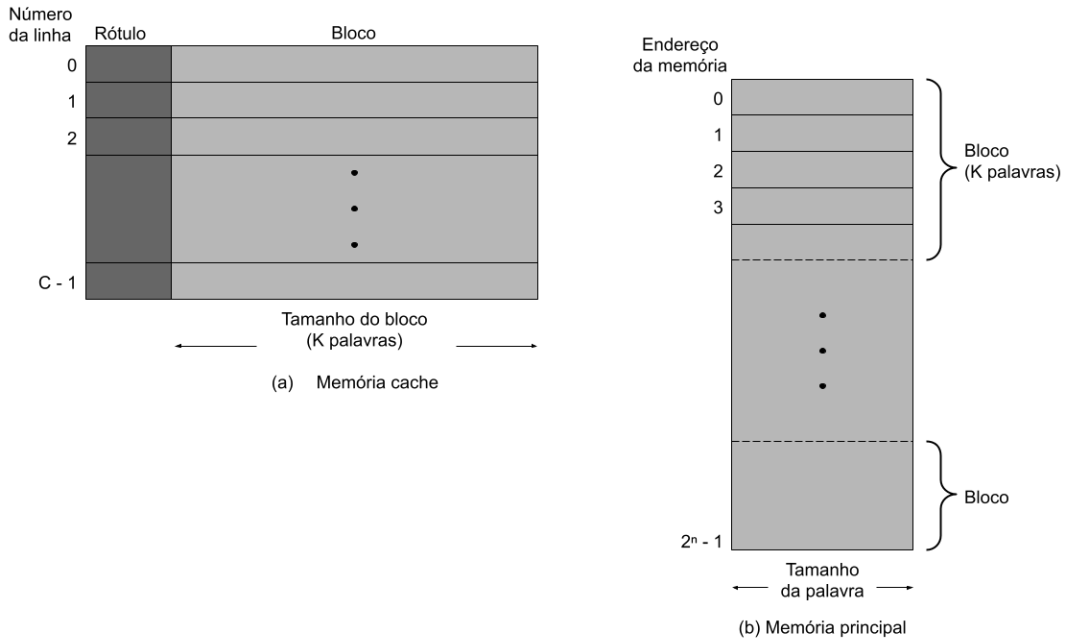

Figura 5. Estrutura da Memória Cache e da Memória Principal. Reproduzido a partir de [Stallings 2010].

que a memória principal possui M blocos de dados, sendo $M = \frac{2^n}{K}$. A memória cache composta por C conjuntos, cada conjunto contendo um bloco de dados (nesse exemplo, o conceito de conjunto e bloco, do ponto de vista da cache, parecem se confundir; porém, as caches com conjuntos de vários blocos serão exemplificadas posteriormente).

Espera-se que, em qualquer hierarquia de memória, a quantidade de linhas de cache seja muito menor do que a quantidade de blocos de memória principal, ou seja, $C << M$. Assim, espera-se que, em qualquer instante do processamento, apenas um pequeno subconjunto de blocos da memória principal esteja presente na cache.

3.1. Como Funcionam as Operações de Leitura e Escrita

As operações realizadas sobre os dados armazenados em qualquer uma das memórias do sistema são classificadas em leitura e escrita. A operação de leitura é identificada por um sinal de controle que indica o tipo de operação (leitura), juntamente com o endereço da palavra solicitada, que será transferida da memória para o processador. Importante destacar que o valor da palavra lida permanece inalterado na memória após a execução da operação. Por outro lado, a operação de escrita requer, além do sinal de controle que indica a operação e o endereço da palavra, uma terceira informação: o novo valor que substituirá o anterior na memória.

Quando uma memória cache é interposta entre o processador e a memória principal, as requisições de leitura e escrita são inicialmente processadas pela cache. A palavra requisitada pode estar presente ou ausente na cache. Caso esteja presente, a operação resulta em um *acerto* ou *hit*; caso contrário, ocorre uma *falha* ou *miss* na cache.

No cenário de um *hit* de leitura, a palavra é diretamente transferida para o pro-

cessador, eliminando a necessidade de acesso à memória principal. Em contrapartida, no caso de um *hit* de escrita, a palavra é atualizada na cache, mas pode ou não ser imediatamente atualizada na memória principal. Essa decisão tem como base uma das duas políticas que regem o tratamento de escritas na cache.

- **Política de escrita *write through*:** estratégia em que cada escrita na cache gera uma escrita na memória principal, mantendo a consistência de dados entre esses dois níveis de memória. Esse é um método muito simples e seguro, porém pouco eficiente, já que todas as escritas na cache geram escritas na memória principal.
- **Política de escrita *write back*:** uma escrita na cache torna o bloco *sujo*, mas não gera uma escrita na memória principal imediatamente, fazendo com que o mesmo bloco tenha valores diferentes na cache e na memória principal. A atualização do bloco sujo na memória principal somente ocorrerá quando for necessário substituí-lo na cache.

Neste trabalho, assume-se que todos os dados requisitados pelo processador durante a execução de um programa estão previamente armazenados na memória principal, descartando a necessidade de considerar níveis inferiores da hierarquia de memória, como a memória virtual. Assim, no caso de um *miss* na cache, a solicitação será encaminhada ao próximo nível da hierarquia até que, no pior cenário, atinja a memória principal, indicando que o dado não foi encontrado em nenhum dos níveis de cache intermediários.

Para explorar o princípio da localidade, o bloco que gerou uma falha deve ser transferido para as caches do sistema, com o objetivo de aumentar a probabilidade de acertos em acessos futuros. Esse bloco é copiado para sua posição correspondente em cada nível de cache ao longo do caminho entre a memória principal e o processador, até que a palavra requisitada seja finalmente disponibilizada ao processador pelo nível de cache mais próximo, geralmente denominada cache L1.

A localização do bloco em uma cache é determinada pela estratégia de mapeamento utilizada para associar um endereço de memória a uma posição específica na cache, sendo essa escolha influenciada pelas características do projeto da cache. Este tema será discutido em detalhes na Seção Mapeamento, após a análise dos princípios do projeto de caches associativas por conjunto, apresentada na Seção 3.2.

3.2. Organização da Cache

Um projeto de cache é definido como um ou mais conjuntos de blocos de dados, além de outros componentes como multiplexadores e comparadores para o controle da cache. Além de ter capacidade para armazenamento de dados, um bloco tem também alguns campos de controle que serão utilizados para o mapeamento dos blocos de memória para a cache e também para aplicação de algoritmos ou políticas de escrita e substituição de blocos.

Um bloco possui, pelo menos, os seguintes campos:

- **Bit de validade**: bit que indica se o bloco está válido (valor 1 ou *verdadeiro*) ou inválido (valor 0 ou *falso*). Inicialmente, todos os blocos da cache são inválidos. Na medida em que os dados são transferidos de outros níveis de memória para cache, tornam-se válidos. A invalidação de um bloco de cache pode ocorrer em sistemas multicore (onde um processador pode invalidar blocos da cache de outro processador, por exemplo).

- **Tag:** Campo que é utilizado para garantir que o respectivo bloco da cache contém os dados do endereço de memória requisitado, ou seja, para garantir que o bloco buscado foi encontrado na cache.
- **Dados:** Campo com uma ou mais palavras de dados que contém os dados do bloco.

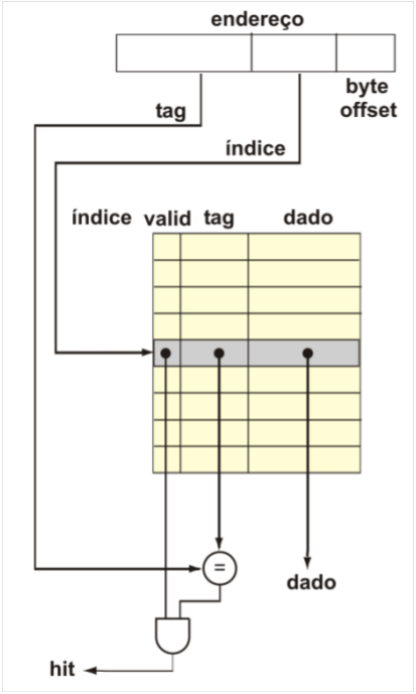

Figura 6. Primeiro projeto de cache. Reproduzido a partir de [Patterson and Hennessy 1994].

Um projeto inicial de cache pode ser visto na Figura 6. Os bits do endereço de memória requisitado são utilizados para mapeamento do bloco na cache. Considerando que estejamos trabalhando com endereçamento de bytes, os últimos dois bits do endereço referem-se ao byte dentro da palavra. Por exemplo, se consideramos palavras de 32 bits ou 4 bytes, o campo *byte_offset* de dois bits servirá para separar o byte desejado dentro de uma palavra. Nesse caso, um endereço terminado em 00 refere-se ao byte 0 da palavra; um endereço terminado em 01 refere-se ao byte 1 da palavra; e assim sucessivamente. Para sistemas com palavras de 8 bytes, serão necessários três bits para *byte_offset*.

O campo *índice*, também chamado de *conjunto* permite o mapeamento do dado requisitado em um determinado conjunto da cache (ou linha da cache). A quantidade de bits desse campo depende da quantidade de conjuntos ou linhas de cache. Por exemplo, uma cache de 256 conjuntos resultará em campo índice de 8 bits ($log_2 256$).

O campo *tag* contém todos os bits restantes do endereço. O campo *tag* do endereço requisitado é comparado com o campo *tag* do bloco de cache mapeado na cache. Se forem iguais, o bloco requisitado foi encontrado na cache. Entretanto, só podemos considerar que houve um *hit* se o bloco estiver válido. Por isso, o resultado do comparador e o bit de validade do bloco são considerados como entrada de uma porta AND que resultará no

sinal *hit*. Se *hit=1 ou verdadeiro* houve acerto na cache; caso contrário, houve falha (ou o bloco não foi encontrado ou ele estava inválido).

3.2.1. Explorando Localidade Espacial

A Figura 7 ilustra uma cache com 256 conjuntos, sendo que cada conjunto possui apenas um bloco de cache. Cada bloco a cache possui dados com 16 palavras e cada palavra possui 32 bits. Os blocos mais largos são úteis para exploração de localidade espacial, visto que a cada acesso à memória serão trazidas várias palavras próximas àquelas que estão sendo requisitadas pelo processador.

Com isso, será necessário incluir mais um campo para o mapeamento do dado na cache. O campo *block_offset* terá quantidade de bits suficientes para identificação de uma das palavras dentro do bloco. Logo, para blocos de k palavras, o campo *block_offset* deverá ter $log_2 k$ bits. No exemplo da Figura 7, o campo *block_offset* tem 4 bits, pois há 16 palavras em cada bloco. A seleção da palavra requisitada é feita por um multiplexador 16x1.

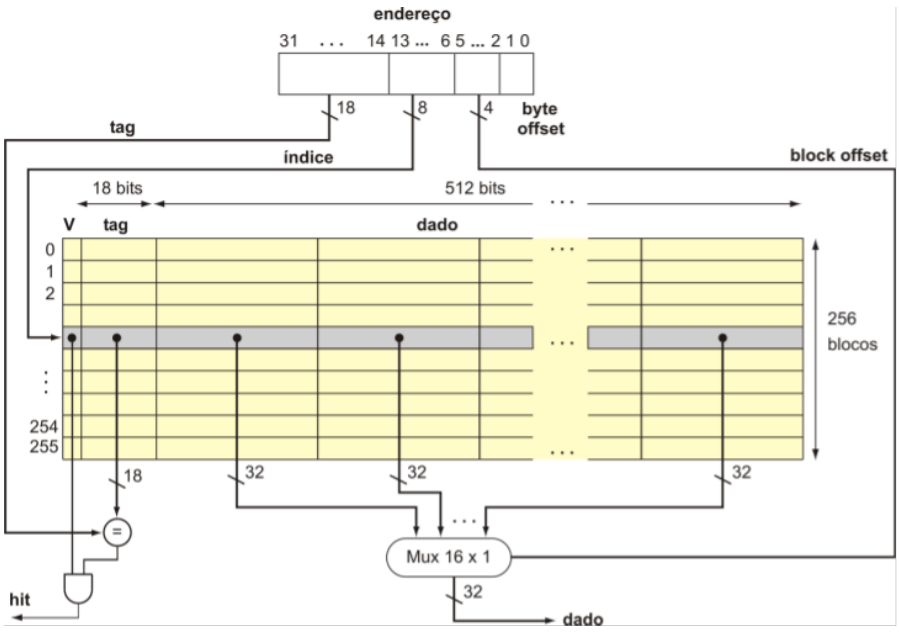

Figura 7. Projeto de Cache. Reproduzido a partir de [Patterson and Hennessy 1994].

3.2.2. Explorando Associatividade

A Figura 8 ilustra uma cache com 256 conjuntos com quatro blocos cada. Para simplificar o exemplo, cada bloco nesse exemplo possui apenas uma palavra de dados, porém nada impediria de explorarmos localidade espacial dentro de cada bloco, aumentando a quantidade de palavras por bloco no projeto.

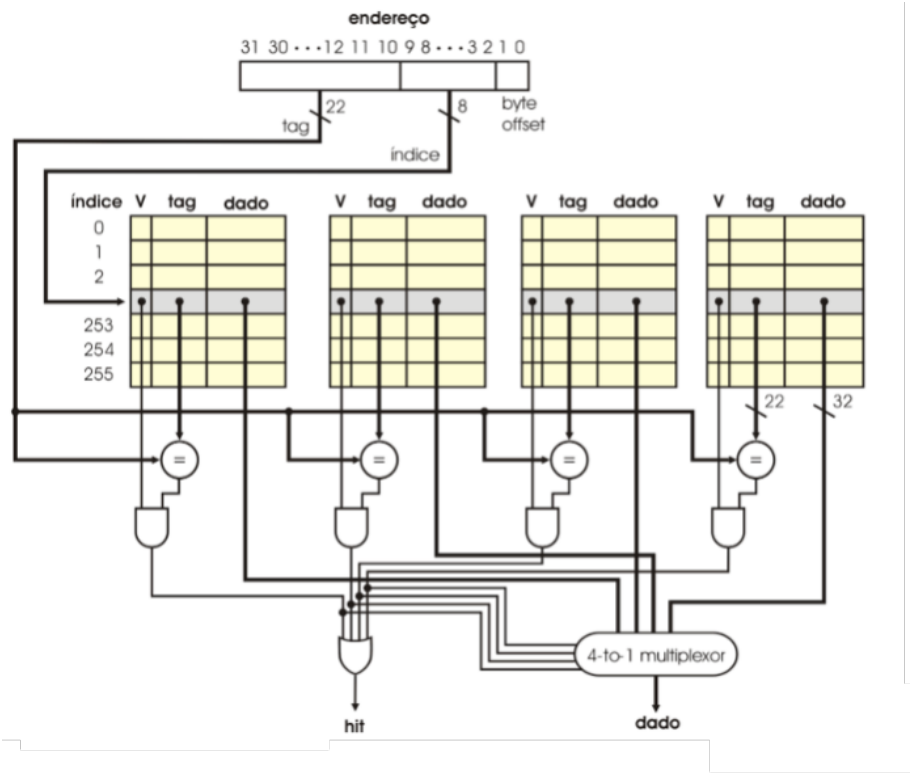

Figura 8. Projeto de Cache Associativa de Quatro vias (4-way). Reproduzido a partir de [Patterson and Hennessy 1994].

Uma cache com associatividade de uma via (*1-way set associative*) é aquela em que cada conjunto possui apenas um bloco [Hennessy and Patterson 2011, Patterson and Hennessy 1994]. Logo, a quantidade de conjuntos e a quantidade de blocos é a mesma. A Figura 7 ilustra esse caso.

Na medida que aumenta a quantidade de blocos por conjunto, dizemos que aumenta a associatividade da cache no projeto. Desta forma, uma cache associativa de duas vias tem dois blocos por conjunto, uma cache associativa de quatro vias tem quatro blocos por conjunto, e assim sucessivamente.

A Figura 8 ilustra uma cache associativa de quatro vias, com 256 conjuntos. Como este projeto tem quatro blocos por conjunto, a quantidade total de blocos nessa cache é 1024 e faz-se necessário o uso de quatro comparadores em paralelo para detectar se o bloco que está sendo requisitado está ou não no conjunto mapeado na cache. Por simplicidade, os blocos deste projeto de cache têm apenas uma palavra, dispensando o campo *block_offset* do endereço de memória.

Uma cache totalmente associativa é aquela em que há um único conjunto contendo todos os blocos da cache. Com isso, não há necessidade de mapeamento para conjunto (já que há um único conjunto) e todas as comparações de *tags* dos blocos com a *tag* do endereço têm que ser feitas em paralelo, tornando-a demasiadamente custosa.

3.2.3. Mapeamento de um Endereço na Cache

Considere uma cache com associatividade m, com total de n blocos, p palavras por bloco e palavra de b bytes. Considere também que o sistema trabalha com endereçamento por bytes, endereços de 32 bits . O mapeamento de um endereço de memória em uma posição da cache é dado da seguinte forma:

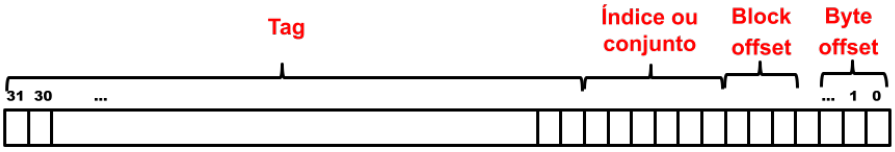

Figura 9. Mapeamento de um endereço de 32 bits para uma cache genérica.

- O campo *byte_offset* de $log_2 b$ bits indica qual o byte dentro da palavra de memória
- O campo *block_offset* de $log_2 p$ bits indica qual a palavra dentro do bloco
- O campo *índice* ou *conjunto* de $log_2(n/m)$ bits indica qual o conjunto de cache do dado requisitado
- O campo *tag* com o restante dos bits $32 - (log_2(n/m) + log_2 p + log_2 b)$.

Como exemplo, suponha que seja uma cache 1024 blocos, associatividade de 8 vias ($m = 8$), 4 palavras por bloco ($p = 4$) e palavras de 8 bytes ($b = 8$). A partir daí, os campos do endereço são definidos com as seguintes quantidades de bits: O campo *byte_offset* terá 3 bits, o campo *block_offset* terá 2 bits e o campo *índice* ou *conjunto* terá 7 bits ($log_2 1024/8 = log_2 128 = 7$).

3.2.4. Política de Substituição de Blocos na Cache

Quanto ocorre uma falha de cache em um projeto com associatividade maior do que um, há necessidade de se escolher qual dos blocos do conjunto deve ser substituído pelo novo bloco. Para isso, há três políticas mais comuns, as quais serão descritas a seguir:

- **LRU:** A política de substituição mais comum é conhecida por LRU (do inglês, *Least Recently Used*), em que o bloco que não foi usado há mais tempo é o escolhido para ser substituído pelo novo bloco. Esse esquema requer bits adicionais de controle de acesso a cada bloco do conjunto.
- **FIFO:** A política de substituição de blocos chamada FIFO (do inglês, *First In, First Out* ou *primeiro a entrar, primeiro a sair*), consiste na substituição de um bloco baseado na ordem de entrada no conjunto. O bloco a ser substituído é o que foi carregado há mais tempo, dentre todos os blocos do conjunto.
- **Random:** Neste esquema, o bloco a ser substituído é escolhido aleatoriamente dentre todos os blocos do conjunto.

3.3. Desempenho de Caches

As falhas de cache são inevitáveis, porém quanto menos falhas, melhor será o desempenho do sistema de memória e, consequentemente, o desempenho do processador. Uma das

métricas para avaliação do desempenho de caches é o *tempo médio de acesso à memória* (t_{mem}), que é definido da seguinte forma:

$$t_{mem} = hit_{time} + miss_{rate} \times miss_{pen} \tag{1}$$

A estimativa do desempenho das caches se dá com base em alguns parâmetros:

- hit_{time}: tempo para acessar a cache e decidir se houve um acerto (hit) ou uma falha (miss).
- $miss_{rate}$: taxa de falhas da cache, que consiste na divisão da quantidade de falhas pelo total de requisições de memória feita pelo processador (ou quantidade de acessos à memória).
- $miss_{pen}$: penalidade de uma falha ou tempo para tratamento de uma falha.

A Figura 10 esboça os parâmetros hit_{time} em preto, decorrente do tempo do processador comunicar-se com a cache L1 (incluindo o tempo da L1 decidir se a requisição é um acerto ou uma falha na cache) e o $miss_{pen}$ em vermelho, que corresponde ao tempo de tratar uma falha na L1 tempo para ir ao próximo nível de memória (nesse caso, a memória principal), copiar o bloco para a L1 e devolver a palavra requisitada ao processador.

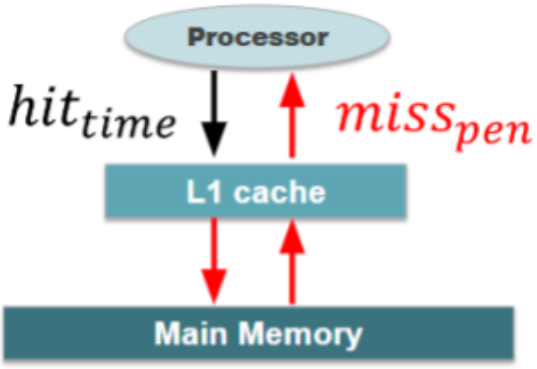

Figura 10. Exemplo dos parâmetros que influenciam no desempenho da hierarquia de memória de um sistema com uma cache L1.

Considerando que o sistema pode ter mais de uma cache, a Equação 3 pode ser generalizada para estimar o tempo médio de acesso à memória da seguinte forma:

$$t_{mem} = hit_{timeL1} + miss_{rateL1} \times miss_{penL1} \tag{2}$$

Porém, a penalidade de falha da L1 é calculada com base nos mesmos parâmetros relativos à L2.

$$miss_{penL1} = hit_{timeL2} + miss_{rateL2} \times miss_{penL2} \tag{3}$$

A partir das Equações 3 e 4, conclui-se que:

$$t_{mem} = hit_{timeL1} + miss_{rateL1} \times (hit_{timeL2} + miss_{rateL2} \times miss_{penL2}) \tag{4}$$

A mesma lógica recorrente pode ser utilizada para estimar o tempo médio de acesso à memória em sistemas com mais do que dois níveis de cache.

4. Metodologia de Desenvolvimento do Simulador

O escopo principal deste simulador é proporcionar uma análise detalhada do funcionamento de diferentes organizações de memórias cache. Essa análise é realizada utilizando padrões de acesso à memória instrumentados de aplicações reais, onde os princípios de localidade espacial e temporal possam ser explorados. As funcionalidades disponíveis, bem como a interface com o usuário, proporcionam o uso do simulador como ferramenta de apoio didático ao ensino de Arquitetura de Computadores.

Para isso, o processo completo de análise de um programa novo dentro do simulador é dividido em três etapas principais. A primeira etapa consiste em instrumentar um programa em tempo de execução para coleta dos padrões de acesso, que chamaremos nesse texto de *traces* de memória. Em seguida, esses *traces* são processados para um formato estruturado compatível com o simulador, contendo todos os acessos à memória de interesse para análise. Por fim, a última etapa envolve a adição da nova entrada e configuração da cache para simulação.

4.1. Tecnologias para o desenvolvimento do simulador

Um dos principais requisitos do projeto é tornar o simulador de fácil acesso para usuários iniciantes. Para atender a esse requisito, optou-se por desenvolver o simulador como uma aplicação *web*, uma vez que elimina a necessidade de instalação pelos usuários, evita a necessidade de atualizações e funciona de forma consistente e independentemente do sistema operacional utilizado.

O simulador também foi desenvolvido restrito em ser apenas *client side*, evitando a necessidade de qualquer infraestrutura envolvendo servidores para que a ferramenta possa ser disponibilizada de forma indeterminada e sem custos associados por serviços como o GitHub Pages [1].

Além disso, foi escolhido *TypeScript* [Hejlsberg] como linguagem principal; sua tipagem estática acelera o desenvolvimento e facilita a manutenção e a geração de documentação técnica do código para futuros contribuintes. Para o ambiente de desenvolvimento, foi utilizada a ferramenta *Vite* [You], otimizada para projetos *web* para um fluxo de trabalho ágil. Testes unitários foram implementados com o *Vitest* [Vladimir and Perkkiö], que oferece uma integração nativa com o *Vite*, proporcionando uma experiência de testes rápida e consistente.

Para a interface do usuário, foi escolhido o *Ant Design* [AntGroup], uma biblioteca de componentes acessíveis e prontos para uso, garantindo uma experiência consistente e alinhada com boas práticas de acessibilidade. Juntamente, *TailwindCSS* [Wathan] foi escolhido para facilitar a prototipação de novos componentes, permitindo a criação rápida e customizável de novos layouts.

4.2. Otimizações de Desempenho

Com o objetivo de reduzir ao máximo a utilização de recursos computacionais durante simulações, fazendo com que o requisito mínimo para um dispositivo conseguir rodar o

[1] https://pages.github.com/

simulador seja rodar um navegador web moderno, diversas otimizações foram implementadas ao longo do desenvolvimento da ferramenta.

Para eliminar um grande volume de alocações de memória, foi proposto que o simulador não iria armazenar os dados sendo acessados pela cache e que manteria apenas uma referência dos endereços de memória que estão carregados.

Ainda em otimizações de alocação de memória, foi implementada uma estrutura de *LazyArray*, que facilita o desenvolvimento de componentes sem alocação durante a inicialização. O resultado disso é que, para simulações de caches de alta capacidade, conjuntos ou blocos que não são acessados não consomem nenhum recurso.

Essas medidas fazem com que a inicialização de uma cache de tamanho arbitrário ocorra sempre em tempo constante.

Além da otimização de memória, endereços de acesso são pré-computados para minimizar *overhead* durante a execução da simulação, o histórico de acessos à cache é limitado para evitar consumo excessivo de memória, e diversas medidas foram tomadas para minimizar atualizações da interface redundantes. Trechos de código com alto potencial de impacto na performance foram monitorados a cada atualização para controle de regressões de desempenho.

4.3. Geração do *Trace* de Memória

Um *trace* de memória é uma sequência de todos os eventos envolvendo acessos à memória durante a execução de um programa. Esses eventos incluem informações como: endereços acessados, modo de operação (escrita ou leitura), o dado que será trafegado e qual o endereço da instrução que ocasionou aquela operação.

Para geração de um *trace* de memória de um novo programa, foi utilizado o *TracerGrind* [SideChannelMarvels 2024], que é uma ferramenta baseada no *Valgrind* [Nethercote and Seward 2007], que permite completa instrumentação das instruções executadas por uma aplicação, com suporte para as arquiteturas `X86`, `X86_64` e `ARM`.

O *TracerGrind* produz um arquivo binário contendo diversas informações sobre a execução da aplicação. Esse arquivo binário é subsequentemente processado pela ferramenta *TextGrind* para produzir um registro em texto de todas as instruções executadas durante a execução da aplicação. Cada linha inicia com um caractere informando o tipo de informação coletada, `I` para instruções de máquina, `M` para acesso à memória e `B` define o final de um bloco de execução e sempre encerra com alguma instrução de desvio de controle.

Nesse registro, uma instrução de máquina começa com o endereço de memória de onde está sendo carregada, seguido do mnemônico correspondente à instrução com os registradores e valores imediatos executados.

Blocos de execução possuem uma identificação única (`EXEC_ID`) associada com a identificação da thread à qual a execução aconteceu (`THREAD_ID`). Os campos `START_ADDRESS` e `END_ADDRESS` apontam para a primeira e última instrução executada naquele bloco, respectivamente.

Acessos à memória começam com o `EXEC_ID` do bloco de execução que foi res-

ponsável pelo acesso, seguido do endereço da instrução associada (INS_ADDRESS). Em seguida, são mostrados: o endereço base de memória (START_ADDRESS), a quantidade de bytes (LENGTH), modo do acesso (MODE) e o dado (DATA) do acesso (valor a ser escrito quando MODE = W e valor lido quando MODE = R).

A seguir, um trecho ilustrativo de uma sequência de operações incluindo instruções, acessos à memória e fim de bloco.

```
[I] 00000000040210da: cmp rax, 0x22
[I] 00000000040210de: jbe 0x40210c9
[M] EXEC_ID:3 INS_ADDRESS:00000000040210c9 START_ADDRESS:000000000403aba0 LENGTH:8
    MODE:W DATA:809e030400000000
[M] EXEC_ID:3 INS_ADDRESS:00000000040210cd START_ADDRESS:0000000004039e90 LENGTH:8
    MODE:R DATA:0400000000000000
[B] EXEC_ID:3 THREAD_ID:0000000000000001 START_ADDRESS:00000000040210c9
    END_ADDRESS:00000000040210de
```

Em conjunto com o *trace* de memória em formato de texto, para auxiliar a próxima etapa de processamento, também é gerado um arquivo auxiliar utilizando a aplicação *readelf* para coleta de endereços do segmento de código e da função main.

O *readelf* permite a leitura de todos os dados armazenados em um binário executável para Linux, dentre esses dados, os necessários para o processamento são o *section headers*, que contém endereço para o segmento de código (.text) e a tabela de símbolos (.symtab), que contém endereço de funções referenciadas durante a execução, incluindo a main.

A seguir, um exemplo truncado da saída do *readelf* com as seções necessárias para o processamento.

```
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  ...
  [17] .text             PROGBITS        00000000000010e0 0010e0 00018a 00  AX  0   0 16


Symbol table '.symtab' contains 42 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
  ...
  36: 000000000000120b    95 FUNC    GLOBAL DEFAULT   17 main
```

Esses endereços são utilizados para calcular o endereço relativo da função main dentro do arquivo binário, permitindo que os blocos de execução e finalização sejam identificados durante a execução, independentemente do endereço base que o sistema operacional escolha para carregar instruções em memória. Dessa forma, é possível filtrar apenas os acessos à memória causados por código de interesse do usuário, evitando acessos causados pelo sistema operacional com instruções de *entrypoint* que preparam o ambiente de execução.

4.4. Processamento do *Trace* de Memória

Separadamente do simulador, também foi desenvolvida uma ferramenta auxiliar para o processamento e validação dos *traces* de acesso à memória, o *TracerGrind-Parser* [Jeller Ferreira 2024]. Esta ferramenta é responsável por transformar o *trace* em formato de texto para um objeto estruturado, onde será realizada a busca pelos blocos de execução de interesse. Também é responsável por quebrar instruções SIMD (*single*

instruction, multiple data) em uma sequência de acessos de largura de uma *word*, que na implementação do simulador se fixou em 64 bits.

A ferramenta calcula endereços relativos para o início e fim da função *main* e utiliza-os como referência para endereços reportados pelo *TracerGrind* durante a instrumentação da aplicação. Todos os blocos entre o bloco de início, que possui sua primeira instrução com endereço relativo igual ao endereço da função *main*, e o bloco de fim, que possui sua última instrução com endereço relativo igual ao endereço da função *main* somado ao seu tamanho.

Como resultado, a ferramenta produz um arquivo no formato JSON em uma estrutura compatível com o simulador, onde pode ser adicionado ao simulador utilizando sua interface gráfica.

4.5. Interface com o Usuário

Para facilitar a compreensão do funcionamento do cache pelo usuário, a interface gráfica foi estruturada em três seções principais: o estado da cache, detalhes da última operação e um histórico.

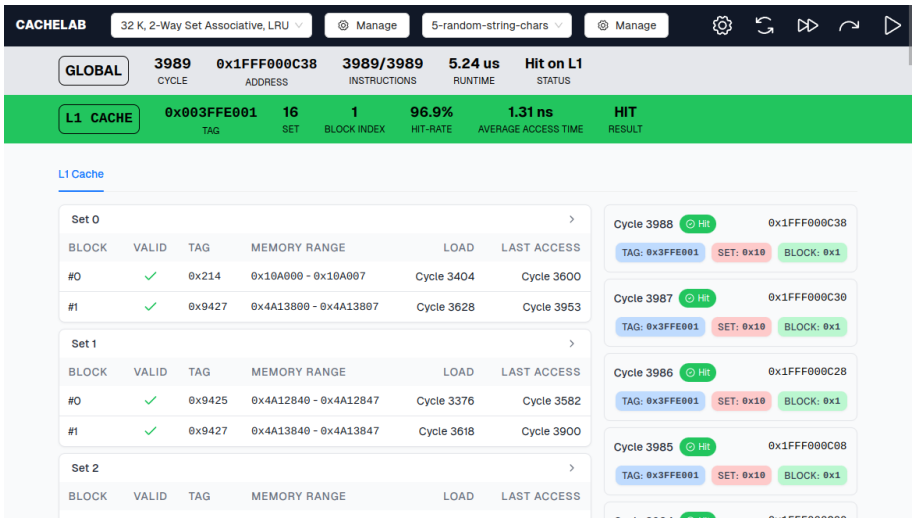

Figura 11. Visão geral do simulador Cachelab.

O estado da cache mostra individualmente cada bloco, agrupado pelos conjuntos de associatividade. É possível visualizar o *range* de endereços carregados no bloco específico, em qual ciclo os dados foram carregados e em qual ciclo foram acessados pela última vez. Durante a execução, esses dados ficam destacados caso o bloco tenha sido acessado durante a última operação.

Os detalhes da última operação mostram qual foi o endereço acessado para cada nível da cache, junto com a *tag*, índice do conjunto calculados com base no endereço e qual bloco foi escolhido com base na política de substituição configurada. Métricas de performance como *hit-rate*, tempo de execução do programa simulado e quantidade de acessos realizados, permitem ao usuário acompanhar a população da cache durante a execução.

Por fim, o histórico mostra os últimos endereços que foram acessados na cache, permite a análise de quais conjuntos e blocos estão sendo mais utilizados e quais

endereços ocasionaram falhas ou *misses*.

Optou-se por utilizar o idioma inglês na interface para facilitar o uso do simulador por universidades do exterior, já que há a intenção de apresentar o simulador em conferência internacional em 2025.

A versão final do simulador CacheLab está publicamente disponível em cachelab.hugo.dev.br e pode ser acessado por qualquer navegador web com suporte a Javascript.

5. Experimentos

5.1. Avaliando a Performance do Simulador

Foram realizados três experimentos para validar as otimizações implementadas no código interno do simulador. O objetivo dos experimentos é comparar o impacto no tempo de execução da simulação, variando apenas um parâmetro da cache. Variando individualmente a quantidade de conjuntos, a associatividade dos conjuntos e os tamanhos dos blocos (medidos em palavras).

O *trace* utilizado para os experimentos conta com 1214 acessos à memória. Os experimentos iniciaram com uma cache simples com 1 conjunto, 1 bloco por conjunto e 1 palavra por bloco. Em cada experimento, um dos parâmetros foi gradualmente incrementado de 1 até 2^{20}, com o objetivo de estabelecer uma relação entre o parâmetro e o tempo necessário para computar a simulação completa.

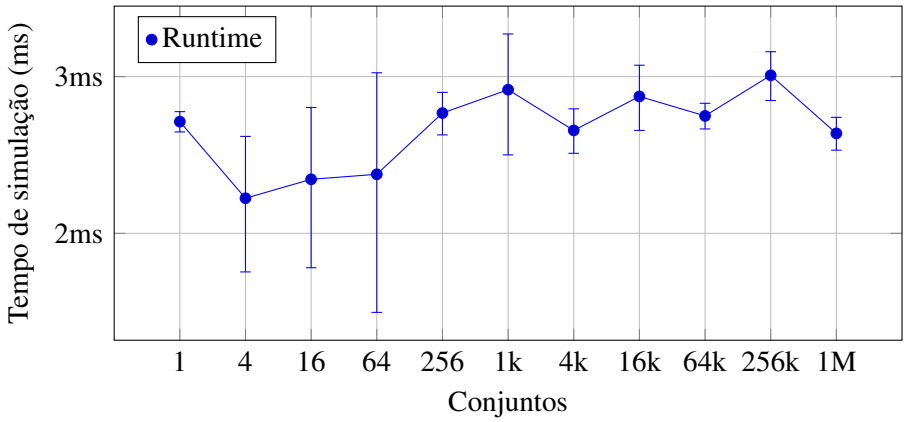

Figura 12. Experimento variando a quantidade de conjuntos.

Os resultados apresentados na Figura 12 mostram o tempo necessário para computar a simulação inteira do experimento e confirmam a invariância com a quantidade de conjuntos configurados para a cache. Isso se deve ao fato de o CacheLab alocar e inicializar estruturas sob demanda durante a simulação, evitando a utilização de recursos simulando conjuntos ou blocos que ainda não foram acessados durante uma simulação.

Variando a associatividade (quantidade de blocos por conjunto) da cache, aumenta o número de *tags* que precisam ser comparadas para verificar se o endereço sendo acessado já está carregado, causando um aumento no tempo de execução da simulação. Os resultados apresentados na Figura 13 mostram uma correlação direta entre a quantidade de blocos simulados em cada conjunto e o tempo de simulação.

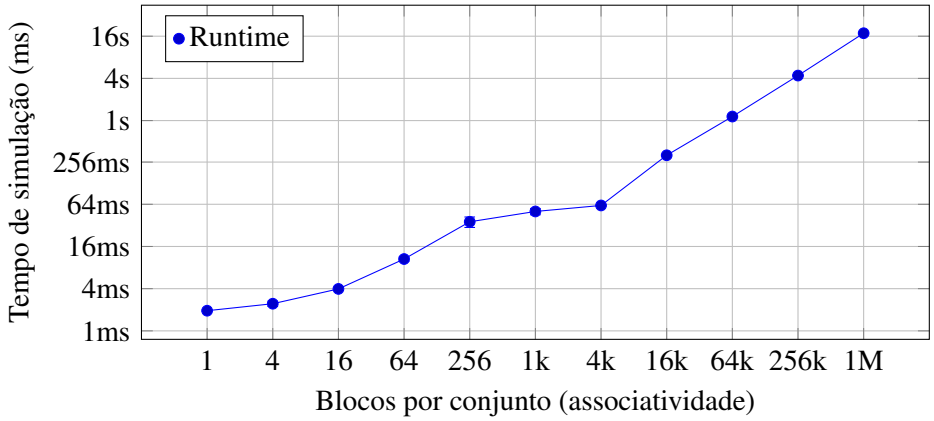

Figura 13. Experimento variando o tamanho dos conjuntos.

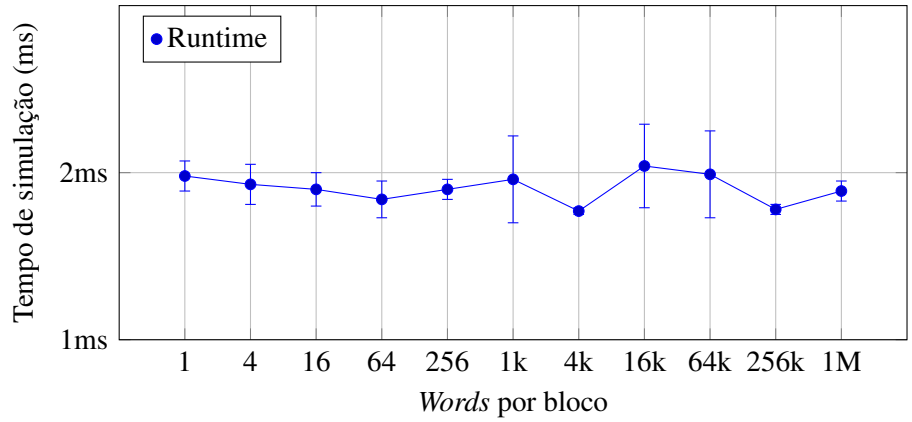

Figura 14. Experimento variando o tamanho do bloco da cache.

Os resultados variando o tamanho dos blocos apresentados na Figura 14 mostram novamente invariância do simulador com a quantidade de palavras em cada bloco. Para minimizar a alocação de memória durante a simulação, CacheLab foi projetado para não armazenar dados acessados pela cache.

Os três experimentos mostram que as otimizações implementadas no simulador foram suficientes para permitir a simulação de caches significativamente maiores do que as que são realisticamente viáveis na prática. Mais relevante para o objetivo didático do simulador, essas otimizações reduzem drasticamente a utilização de recursos computacionais, eliminando requisitos mínimos para usuários utilizarem a ferramenta.

5.2. Ordem Principal em Matrizes

O uso eficiente da memória cache é um dos desafios mais importantes no desenvolvimento de algoritmos de alto desempenho. Entre os fatores que afetam diretamente o desempenho de programas está o fenômeno das falhas de cache (*cache miss*), que ocorrem quando os dados requisitados pelo processador não estão disponíveis na memória cache e precisam ser recuperados da memória principal.

A organização dos dados em memória, como no caso do ordenamento por linhas ou colunas em matrizes, pode impactar significativamente a frequência desses eventos.

Este problema torna-se evidente em somas ou multiplicações de matrizes, onde o acesso sequencial por linhas geralmente minimiza os cache misses devido ao princípio da localidade espacial.

Utilizando o CacheLab como referência para comparação de execuções percorrendo matrizes por linhas e colunas, é possível analisar o impacto direto da organização dos dados no tempo de execução, fornecendo insights para otimizações no uso de caches.

Simulações foram feitas utilizando uma matriz de 250x250 de números inteiros do tipo `long` de 32 bits. O código em C responsável pelo *loop* para gerar os índices de acesso teve o seguinte formato:

```
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        // code
    }
}
```

Acessos por linha foram feitos utilizando `matrix[i][j]`, enquanto acessos por coluna foram feitos com `matrix[j][i]`. Considerando que a linguagem C armazena arranjos bidimensionais com ordem principal em linha, entende-se que o acesso por linha tenha acessos sequenciais na memória, enquanto o acesso por coluna tenha constantes saltos de endereços durante a varredura da matriz.

O tamanho final da cache de 16KB foi determinado experimentalmente, maximizando o impacto da quantidade de palavras dentro de cada bloco. Ao variar o tamanho dos blocos, a quantidade de conjuntos era ajustada para manter o tamanho final da cache normalizado:

$$N_{Sets} = 256/N_{WordsPerBlock}$$

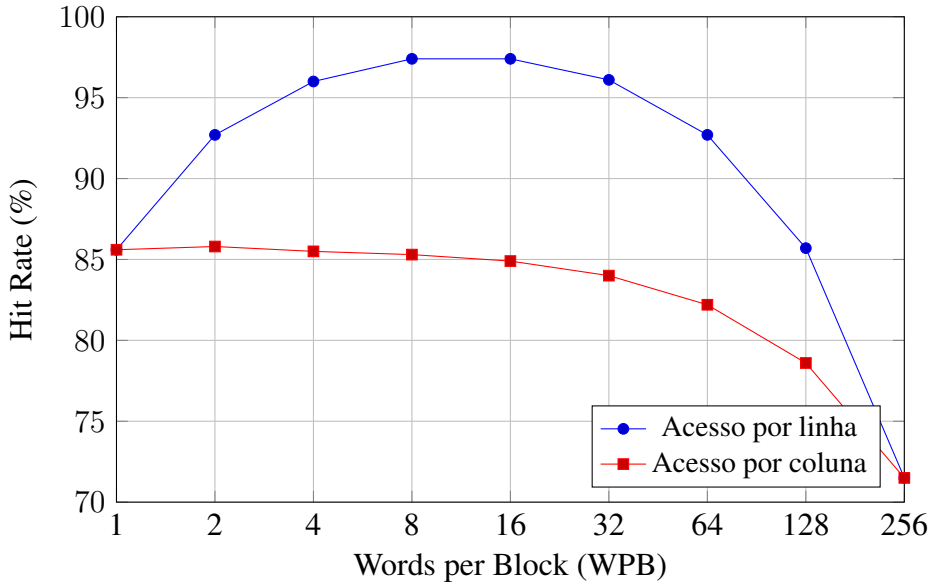

Figura 15. Experimento de varredura de matrizes.

Os resultados apresentados na Figura 15 mostram que o acesso por linha explora o princípio de localidade espacial por estar fazendo acessos sequenciais na memória onde

a matriz está armazenada. O aumento da quantidade de *words* em cada bloco, aumenta a utilização dos dados carregados após um evento de *miss*. Esse comportamento não é observado no acesso por coluna, uma vez que os saltos de endereços de acesso de cada item da matriz não acessam dados já carregados na cache.

Também foi possível observar que, para uma cache de capacidade total fixa, o aumento da capacidade de cada bloco é benéfico até o ponto em que a redução no número de conjuntos começa a aumentar significativamente a quantidade de conflitos na cache durante acessos em outras regiões de memória.

5.3. Quick-sort vs Merge-sort

A comparação entre os algoritmos quick-sort e merge-sort é amplamente relevante por se tratar de duas abordagens de ordenação eficientes e muito estudadas. Ambos oferecem um desempenho semelhante para o caso médio [Esau Taiwo et al. 2020], mas possuem uma distinção na utilização de memória. O quick-sort realiza ordenamento ”in-place”, sem a necessidade de alocar memória auxiliar, enquanto o merge-sort faz uso de memória auxiliar para dividir e mesclar os dados.

Essa diferença impacta diretamente a eficiência da cache, fazendo acessos em uma região mais dispersa de memória e introduzindo maior número de *misses*. Esse experimento explorou o impacto de diferentes níveis de associatividade com dois diferentes tamanhos de bloco (com 1 e 4 *words*) ordenando uma lista de 2000 itens do tipo *long*.

Os *traces* de memória obtidos mostram que o merge-sort precisou de 814 mil acessos à memória, enquanto o quick-sort apenas 589 mil acessos.

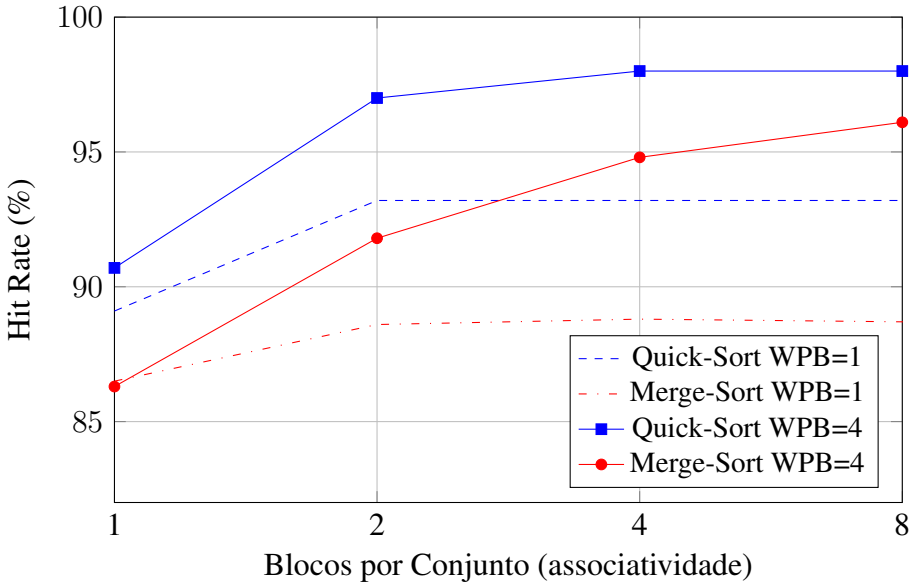

Figura 16. Experimento comparando quick-sort com merge-sort.

Os resultados obtidos na simulação mostram que o quick-sort obteve maior hit rate em todas as diferentes configurações de cache. Vale notar que a capacidade total das caches desse experimento foi determinada de forma experimental com o objetivo

de maximizar os impactos de outros parâmetros. Com isso, foi possível observar que o algoritmo merge-sort exige uma maior capacidade de memória durante sua execução.

Esse experimento também mostrou que a capacidade total da cache não é um parâmetro suficiente para determinar sua eficiência, e sugere um efeito sinérgico entre seus parâmetros. As simulações utilizando apenas 1 *word* por bloco não tiveram melhora no desempenho com aumento da associatividade dos conjuntos. Hit-rates para quick-sort e merge-sort rapidamente atingiram um teto de 93.1% e 89.2%, respectivamente.

Aumentando o tamanho dos blocos para 4 *words*, ainda que ao custo da quantidade total de conjuntos na cache, resultou em uma melhora na eficiência com maiores associatividades, atingindo 98% de hit rate para o quick-sort e 96.1% para o merge-sort.

Para estimar latências reais de uma cache, foi utilizada a ferramenta `perf` do sistema operacional Linux, que expõe diversos contadores de performance. Como referência, um Intel® Core™ i7-4790K com frequência de $4\,\text{GHz}$ foi utilizado, resultando em uma latência média de 4 ciclos para *hits* e aproximadamente 40 ciclos perdidos no evento de um *miss*. Configurando a cache simulada com hit_time de $1ns$ e $miss_{penalty}$ de $10ns$, foi possível comparar o tempo de execução simulado para os casos de blocos com 4 *words*.

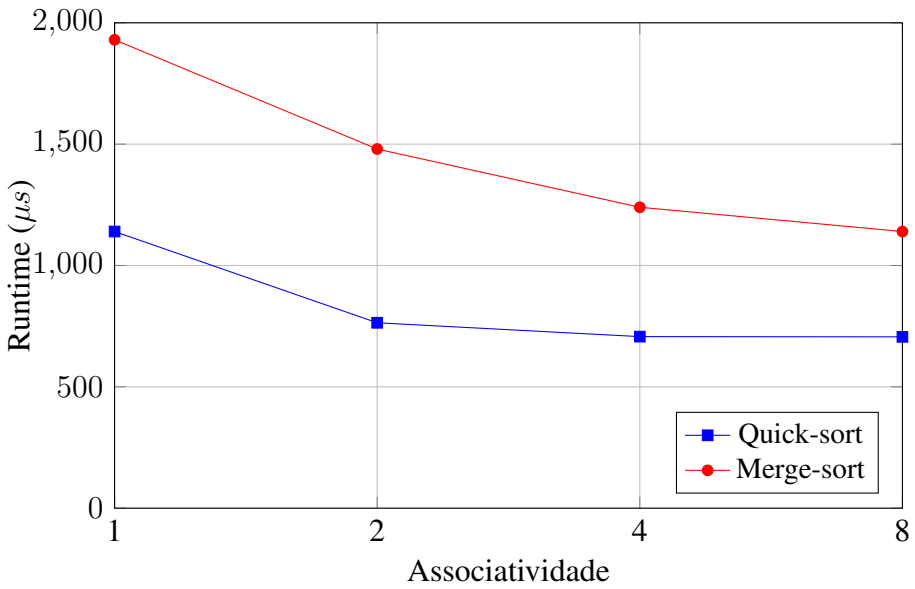

Figura 17. Experimento runtime entre quick-sort e merge-sort.

Por fim, os resultados apresentados na Figura 17 mostram que, em uma situação restritiva de cache, o algoritmo quick-sort obteve um tempo total de execução 38% menor quando comparado com merge-sort para o mesmo conjunto de dados, utilizando como referência a configuração de associatividade 8 que resultou no maior hit_{rate} dos experimentos. Isso se deve ao fato de que o quick-sort teve 27% menos acessos à memória e um tempo médio de acesso a cache de $1.20ns$ comparado com os $1.39ns$ do merge-sort, uma diferença de 14%.

5.4. Cache de 2 níveis

Para analisar a interação de uma cache de 2 níveis, um experimento utilizando a função `strcmp` comparou 2 strings de 6 caracteres, realizando 1026 acessos à memória.

O nível 1 da cache se manteve fixo durante todo o experimento com 1 conjunto totalmente associativo com 32 blocos, cada um contendo 8 palavras, com hit-time de 1 ns. Já o nível 2 teve o número de conjuntos variando de 1 até 128, diretamente mapeada e cada bloco contendo 128 palavras, com hit-time de 3 ns e penalty time de 30 ns.

Como os parâmetros da cache de nível 1 não foram alterados, todas as simulações resultaram em hit rate de 84.5%, fazendo com que seu tempo de acesso médio seja impactado exclusivamente pela performance da cache de nível 2.

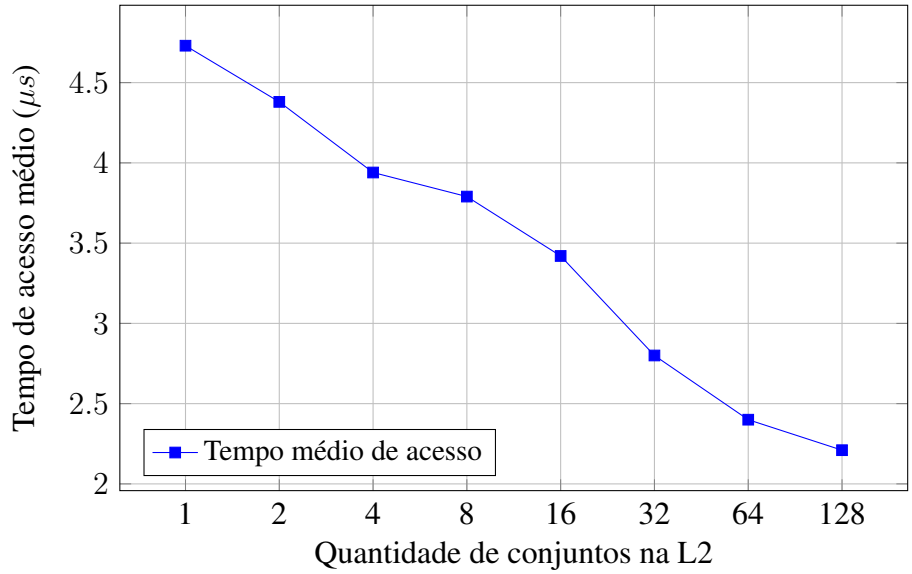

Figura 18. Impacto do tamanho da cache L2 no tempo de acesso médio da cache L1

Os dados do experimento apresentados na Figura 18 mostram que, com o aumento da capacidade do segundo nível da cache, o tempo médio de acesso à memória diminuiu, uma vez que a penalidade de *misses* acaba sendo reduzida.

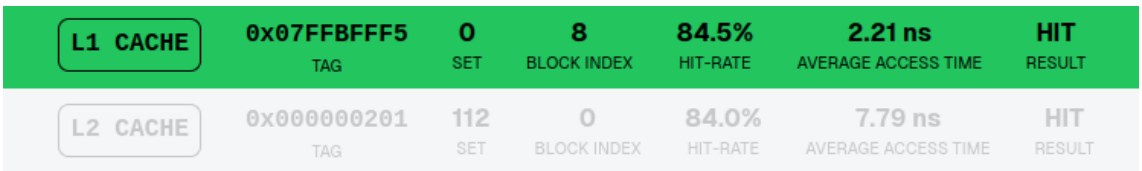

Figura 19. Visão geral do simulador com a cache L2 configurada para 128 conjuntos ao fim da simulação.

6. Conclusão

O objetivo desse trabalho foi implementar um simulador de caches didático de fácil acesso e capaz de simular padrões de acesso à memória de aplicações reais. Foi feito um estudo

de simuladores de cache com propósitos semelhantes e em quais pontos cada um deles poderia ser melhorado.

Ainda que o foco do simulador seja permitir que o usuário navegue a operação de uma cache acesso por acesso, as otimizações implementadas se mostraram suficientes para que *traces* completos com centenas de milhares de endereços fossem executados por completo. Além disso, também é capaz de simular caches de tamanhos impraticáveis sem requisitos computacionais altos.

Por fim, vale destacar que o CacheLab abre possibilidades para o ensino de disciplinas de Arquitetura de Computadores com auxílio de ferramentas com foco para o ensino, ajudando alunos na visualização e experimentação dos diversos conceitos envolvendo caches.

Para trabalhos futuros, a possibilidade de integração do processo de geração dos *traces* de memória diretamente no simulador elimina a necessidade de ferramentas externas de instrumentação e processamento, tornando o fluxo de utilização mais simples e acessível. Essa integração também permite ao usuário escolher qual função presente na `.symtable` do binário ELF (além da `main()`) deve ser utilizada para filtrar acessos à memória, e dessa forma realizar simulações de trechos mais específicos de um programa.

Além disso, a implementação de políticas de escrita *write-back* juntamente com a diferenciação de leituras e escritas permitiria uma análise mais completa e seria especialmente relevante em cenários didáticos durante a simulação de aplicações com alto volume de escritas.

Também pode-se integrar ao simulador funcional algum estimador de área e consumo para podermos avaliar projetos de cache utilizando parâmetros físicos do projeto. Um exemplo seria a integração com o MCPat, ferramenta da HP, que possibilita estimativas físicas de projetos até $22nm$.

Por último, atualizações na interface focando em responsividade podem melhorar a acessibilidade da aplicação para dispositivos móveis com telas pequenas, já que o simulador foi inicialmente desenvolvido para uso em *desktop*.

Referências

[AntGroup] AntGroup. Ant design: An enterprise-class ui design language and react ui library. `https://ant.design`.

[BAER 2010] BAER, J. (2010). *Arquitetura de Microprocessadores: Do Simples Pipeline ao Multiprocessador em Chip*. Editora LTC.

[Esau Taiwo et al. 2020] Esau Taiwo, O., Christianah, A. O., Oluwatobi, A. N., Aderonke, K. A., and kehinde, A. J. (2020). Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort. *Procedia Computer Science*, 171:2532–2540. Third International Conference on Computing and Network Communications (CoCoNet'19).

[Hejlsberg] Hejlsberg, A. Typescript: Javascript with syntax for types. `https://www.typescriptlang.org`.

[Hennessy and Patterson 2011] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[IEEE 2019] IEEE (2019). Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84.

[Jeller Ferreira 2024] Jeller Ferreira, H. (2024). Tracergrind-parser: Tracergrind texttrace dump parser. https://github.com/HugoJF/cache-simulator-tracergrind-parser.

[LeandroGabriel.net 2022] LeandroGabriel.net (2022). Beto: o simulador de memória cache. https://simuladorcache.leandrogabriel.net.

[Liptay 1968] Liptay, J. S. (1968). Structural aspects of the system/360 model 85, ii: The cache. *IBM Systems Journal*, 7(1):15–21.

[Nethercote and Seward 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100.

[PARHAMI 2007] PARHAMI, B. (2007). *Arquitetura de Computadores - de Microprocessadores a Supercomputadores*. McGraw-Hill.

[Patterson and Hennessy 1994] Patterson, D. A. and Hennessy, J. L. (1994). *Computer organization and Design*. Morgan Kaufmann.

[SideChannelMarvels 2024] SideChannelMarvels (2024). Tracer: Set of dynamic binary instrumentation and visualization tools for execution traces. https://github.com/SideChannelMarvels/Tracer.

[Stallings 2010] Stallings, W. (2010). *Arquitetura e Organização de Computadores 8a Edição*. São Paulo: Prentice Hall do Brasil.

[Vladimir and Perkkiö] Vladimir, A. F. and Perkkiö, A. Vitest: Next generation testing framework powered by vite. `https://vitest.dev`.

[Washington] Washington, U. O. Cache simulator. https://courses.cs.washington.edu/courses/cse351/cachesim/.

[Wathan] Wathan, A. Tailwindcss: A utility-first css framework for rapid ui development. `https://tailwindcss.com`.

[Wuterich] Wuterich, A. A configurable cache simulator. https://rivoire.cs.sonoma.edu/cs351/resources.html.

[You] You, E. Vite: Next generation frontend tooling. it's fast! `https://vite.dev`.