Otimização da Eficiência de Cache em Sistemas Transacionais de Alta Performance: Uma Avaliação Experimental em Rust

Fernando Medeiros
Instituto Federal Sul-rio-grandense (IFSul)
Bagé, RS, Brasil
fernandogdemedeirosw@gmail.com

Marcel Corrêa
Instituto Federal Sul-rio-grandense (IFSul)
Bagé, RS, Brasil
marcelcorrea@ifsul.edu.br

Abstract

Modern high-performance systems increasingly face memory latency as a dominant bottleneck, even when running on CPUs with substantial computational capacity. This work investigates how software-level design choices—particularly data layout and concurrency control—directly influence cache efficiency and overall CPU utilization in transactional workloads. We implement a simplified, in-memory clone of the Brazilian Pix instant-payment system in Rust and compare a naïve, lock-bound architecture with an optimized, memory-aware version. Using hardware performance counters collected with perf, we show that applying techniques such as lock sharding, data-oriented struct redesign, and zero-copy mutation reduces the L1 data-cache miss rate from 43.6% to 6.7% and increases IPC by 33% (from 0.48 to 0.64). These results demonstrate that aligning software structure with processor memory hierarchies can shift a workload from latency-bound to compute-bound behavior, yielding substantial throughput gains without changes to hardware. The study highlights the importance of cache-conscious programming in real-time financial systems and provides a reproducible methodology for evaluating memory-centric optimizations in modern multicore environments.

Palavras-chave

Cache Optimization, High-Performance Computing, Lock Sharding

1 Introdução

Ao longo das últimas décadas, pode-se observar um crescimento expressivo no desempenho dos processadores de propósito geral. No entanto, as memórias não evoluíram na mesma velocidade, como ilustrado na Figura 1. Essa discrepância, conhecida como *memory wall*, pode levar a um gargalo significativo: a latência envolvida na comunicação entre processador (*Central Processing Unit* - CPU) e memória principal (*Random Access Memory* - RAM) leva a uma potencial subutilização da capacidade computacional do processador, fazendo com que o mesmo tenha de gastar vários ciclos apenas esperando [1].

Para reduzir esse problema, arquiteturas atuais utilizam-se de diversas estratégias, majoritariamente a nível de projeto de *hardware*. Uma delas foi a migração do controlador de memória da placa-mãe para dentro do processador, com o objetivo de se reduzir drasticamente a latência e aumentar a largura de banda no acesso à RAM. Quando o controlador ficava localizado na placa-mãe, em arquiteturas baseadas em *Front-Side Bus* (FSB), o processador precisava comunicar-se com a memória através de um barramento externo compartilhado, o que se tornava um gargalo à medida que as frequências de CPU aumentavam.

Além desta, uma das estratégias mais importantes é o uso de hierarquia de memória, que utiliza múltiplos níveis de memória *cache* integrados ao processador. Essa organização busca manter os dados mais acessados em memórias mais rápidas, próximas ao CPU, reduzindo a latência média de acesso [4]. A Figura 2 ilustra essa estratégia.

Além da hierarquia de memória, outras técnicas de *hardware* contribuem para minimizar o impacto da latência. O *prefetching* atua de forma preditiva, carregando dados para o *cache* antes que sejam requeridos pelo processador. O *multithreading*, especialmente em implementações como o *Simultaneous Multithreading* (SMT), permite que um núcleo físico execute mais de uma *thread*, de modo que, enquanto uma aguarda dados devido a um *cache miss*, outra pode ser processada, diminuindo o tempo ocioso do núcleo [5]. Já o conceito de *near-memory computing* propõe aproximar a computação da memória, reduzindo drasticamente o tráfego de dados — um dos principais gargalos em aplicações modernas, que em muitos casos possuem alta carga de dados [6].

Contudo, embora a maioria dessas soluções esteja no domínio do *hardware*, sua eficácia depende, em parte, das práticas adotadas no desenvolvimento do *software*. A maneira como os dados são organizados e acessados pelo código pode resultar em um alto índice de acertos no *cache (cache hits)*, maximizando o desempenho, ou, inversamente, em frequentes perdas (*cache misses*), obrigando o processador a buscar informações em um nível inferior da hierarquia, assim, penalizando a performance do sistema [1, 4, 7]. Portanto, otimizações a nível de código para o aproveitamento eficiente das implementações de hardware é fundamental.

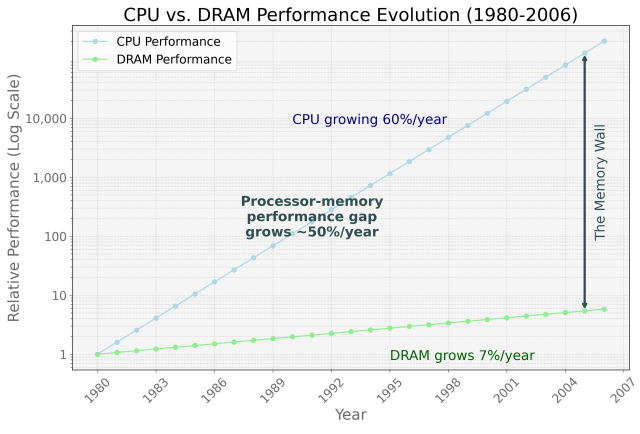

Figura 1: Evolução da performance de processadores vs. memória [2].

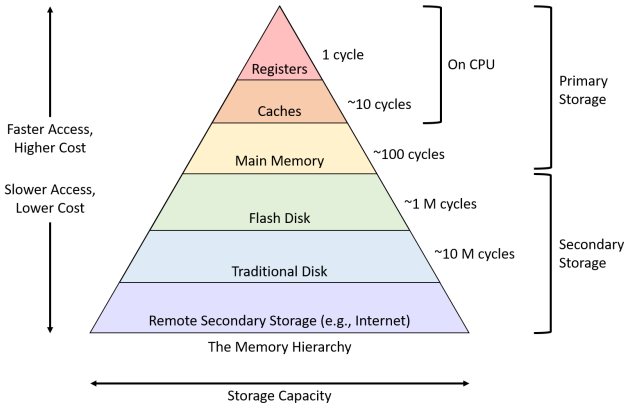

Figura 2: Ilustração da hierarquia de memória [3].

A literatura recente sobre sistemas de alto desempenho aponta consistentemente para a latência de memória como um dos principais gargalos em arquiteturas modernas. Pesquisas focadas em *datacenters* de grande escala demonstram que, mesmo com o avanço no poder de processamento, aplicações críticas (como sistemas de recomendação em larga escala) sofrem com o fenômeno do *memory wall*. Estudos indicam que a ineficiência no *frontend* da CPU e a espera por dados (*stalls*) limitam o desempenho real, sugerindo que otimizações de *software*, como o uso inteligente de *prefetching* e gerenciamento de *threads*, podem mitigar esses efeitos em cenários onde o *hardware* já atingiu seus limites físicos [8].

Outra vertente de investigação concentra-se especificamente na estratégia de otimização para aplicações intensivas em CPU em ambientes virtualizados. Experimentos utilizando modelos preditivos de tempo de execução evidenciam que existe um limiar ótimo de superposição de tarefas; ao ultrapassar a quantidade de núcleos físicos disponíveis, a competição por recursos de processamento degrada a eficiência da execução. Essas descobertas corroboram a importância de estratégias de escalonamento que respeitem a capacidade física do *hardware*, evitando a degradação de performance causada pela concorrência excessiva e pelo *overhead* de gerenciamento de processos [9].

Diante desse cenário, este trabalho investiga o impacto de práticas de programação conscientes do uso da memória *cache* na eficiência de aplicações de alto desempenho. São comparadas duas versões funcionalmente equivalentes: uma abordagem *baseline* e outra otimizada segundo as práticas discutidas. Por fim, define-se um protocolo experimental controlado que permite avaliar, de forma justa e reprodutível, o impacto dessas otimizações.

Este artigo está estruturado da seguinte forma: a Seção 2 define o problema e apresenta os fundamentos teóricos necessários para compreender o impacto da hierarquia de memória no desempenho de sistemas modernos. A Seção 3 descreve a metodologia adotada, incluindo o modelo do sistema implementado e o protocolo experimental. A Seção 4 discute os resultados obtidos a partir das medições de *hardware*. Por fim, a Seção 5 apresenta as conclusões e aponta direções para trabalhos futuros.

2 Justificativa

Para a completa compreensão deste trabalho, é essencial consolidar alguns conceitos fundamentais que formam a base das otimizações de desempenho abordadas. O principal deles é a memória *cache*, um tipo de memória volátil, de tamanho reduzido, porém extremamente rápida, que serve como uma intermediária entre o processador e a memória principal. Sua função é armazenar cópias dos dados e instruções mais frequentemente utilizados, e como seu tempo de acesso é ordens de magnitude menor que o da RAM, seu uso eficaz acelera significativamente o desempenho geral do sistema, mitigando o gargalo de velocidade entre o processador e a memória [7].

A eficácia de uma *cache* é medida por sua capacidade de fornecer os dados que o processador solicita. O *Cache Hit* (acerto de cache) é o cenário ideal: o dado solicitado é encontrado na *cache* e entregue à CPU rapidamente, permitindo que o processamento continue. Por outro lado, o *Cache Miss* (falha de *cache*) ocorre quando o dado não se encontra na *cache*, forçando o sistema a buscá-lo em um nível inferior da hierarquia. Uma busca na memória principal ocorre no pior caso: um *cache miss* em todos níveis de *cache*. Após a busca, o bloco de dados é copiado para a *cache*, mas a ocorrência frequente de *cache misses* degrada a performance do sistema [7].

Para garantir uma alta taxa de acertos, as arquiteturas de *cache* exploram um comportamento previsível dos programas chamado princípio da localidade: A localidade temporal é a tendência de um processador acessar novamente dados que usou recentemente, como uma variável de controle dentro de um laço de repetição (*loop*). Já a localidade espacial é a tendência de acessar dados em posições de memória próximas àquelas acessadas há pouco tempo, como ocorre ao se percorrer os elementos de um vetor [10].

Além dos fundamentos de *hardware* que impactam a performance, a eficiência depende também da arquitetura de comunicação. Nesse âmbito, outro conceito relevante é o gRPC *Remote Procedure Calls* (RPC), que trata-se de um *framework* de código aberto, mantido pela *Cloud Native Computing Foundation* (CNCF), e que moderniza o modelo de RPC tradicional ao utilizar *protocol buffers* (protobuf) para serialização de dados e o protocolo *Hypertext Transfer Protocol version* 2 (HTTP/2) para a comunicação. Em comparação com a arquitetura *Representational State Transfer* (REST), o gRPC se diferencia por ser orientado a ações (serviços invocados pelo cliente) em vez de recursos (manipulados por verbos HTTP como GET e POST). Adicionalmente, enquanto APIs REST geralmente usam *JavaScript Object Notation* (JSON), um formato de texto flexível, o gRPC utiliza protobuf, um formato binário que, embora não legível para humanos, oferece maior performance tanto na transmissão de dados quanto no *parsing* [11].

Apesar das robustas soluções de *hardware* existentes, a sua eficácia é frequentemente limitada por um gargalo no domínio do *software*. O problema central reside no fato de que uma parcela significativa dos desenvolvedores, especialmente aqueles que trabalham com linguagens de programação de alto nível, negligencia ou desconhece os princípios de organização e acesso a dados que favorecem o bom uso da hierarquia de memória. Portanto, torna-se fundamental que os programadores compreendam como suas escolhas de estruturas de dados impactam o comportamento do *cache*. Ignorar esse fator pode levar a anomalias de desempenho,

onde uma estrutura com menor complexidade algorítmica para uma determinada operação acaba, na prática, apresentando uma performance inferior devido a um padrão de acesso à memória que gera um alto índice de *cache misses*.

O crescente nível de abstração das linguagens de programação modernas, embora benéfico para a produtividade, oculta os detalhes da arquitetura de *hardware* subjacente. Com isso, conceitos como localidade espacial e temporal dos dados, que são importantes para a eficiência do *cache*, não fazem parte das preocupações cotidianas do programador. Em linguagens não gerenciadas, como C++, o programador tem controle direto sobre o *layout* dos dados na memória. Por exemplo, é possível alocar um grande bloco contíguo de memória (um *memory pool*) e sub-alocar objetos manualmente dentro dele, garantindo que itens que precisam ser acessados em conjunto permaneçam fisicamente próximos, melhorando a localidade espacial. Outra otimização comum, visando melhorar a eficiência do *cache* em iterações longas, é trocar o padrão *Array of Structures* (AoS) — um único *array* de objetos complexos — pelo padrão *Structure of Arrays* (SoA). Neste último, o programador cria múltiplos *arrays*, de maneira que cada um armazena os valores de um único campo específico de todos os objetos. Dessa forma, ao iterar para acessar apenas um campo (por exemplo, calcular a média de um atributo de todos os objetos), o *prefetcher* do processador carrega apenas dados úteis, maximizando os *cache hits* e evitando a poluição da cache com dados desnecessários.

Tentar aplicar essas mesmas técnicas em linguagens gerenciadas (como Java ou C#) é muito mais complexo, quando não impossível. O alocador de memória não oferece garantias de que os objetos de um *array* serão posicionados sequencialmente na memória e o *garbage collector* não oferece garantias quanto ao tempo levado para liberação de memória. Além disso, a aplicação manual do padrão SoA elimina a integridade e a identidade do objeto (visto que não há mais uma referência única para o objeto original), introduzindo problemas de segurança de memória e, crucialmente, anulando os benefícios do *garbage collection* automático para objetos individuais [12].

A consequência direta dessas práticas é um padrão de acesso a dados ineficiente, que resulta em um baixo índice de acertos no *cache*. Cada *cache miss* força o processador a incorrer em ciclos de espera (*stalls*), aguardando os dados serem buscados em um nível inferior da hierarquia de memória [7]. Em sistemas que necessitam de alto desempenho e alta vazão de dados, como sistemas de pagamento instantâneo, plataformas de *e-commerce* em grande escala ou aplicações de análise de dados em tempo real, esse efeito é amplificado e pode se tornar crítico.

Nesses cenários, a performance não é apenas uma questão de conveniência, mas um requisito fundamental. A latência excessiva, causada pela subutilização da CPU, pode levar à perda de transações por *timeout*, degradação da experiência do usuário e, em última instância, à necessidade de provisionar uma infraestrutura de *hardware* muito mais custosa para compensar a ineficiência do *software*. Como resultado, esses sistemas sofrem com escalabilidade deficiente, degradação de performance ao longo do tempo e subutilização de recursos computacionais [13].

Portanto, o problema central que este trabalho aborda é a lacuna de conhecimento sobre otimizações e uso de memória.

3 Metodologia

Dentre sistemas de pagamento instantâneo, destaca-se o Pix, criado pelo Banco Central (BC) brasileiro. Este sistema permite a transferência de recursos entre contas em poucos segundos, 24 horas por dia, sete dias por semana. Dada a sua natureza de alta disponibilidade, baixa latência e altíssima transacionalidade, o Pix é uma tecnologia disruptiva que rapidamente se tornou um pilar da infraestrutura financeira do Brasil.

Este trabalho inspira-se nas operações e requisitos de alto desempenho do Pix para modelar um sistema financeiro simplificado. O objetivo é explorar otimizações em um ambiente onde o processamento em memória e a eficiência da CPU são críticos, similar aos desafios enfrentados por sistemas reais.

O escopo funcional abrange as seguintes operações: criação de conta e chave, consulta de saldo, transferência entre contas e registro de transação. O sistema deverá manter a consistência de saldo e idempotência para solicitações repetidas.

3.1 Arquitetura e Implementação

Foi desenvolvido um microserviço único com estado mantido em memória, eliminando gargalos de I/O de disco ou rede que poderiam mascarar os efeitos da otimização de *cache*. A comunicação com o serviço foi feita por RPC através do *framework* gRPC, para comunicação de alta performance durante os testes de carga, e transferência de dados em formato JSON através do HTTP/2.

A linguagem utilizada foi *Rust* v1.90.0. Para garantir uma comparação justa entre as versões *baseline* e otimizada, a compilação de ambas foi realizada com as seguintes configurações de otimização:

- Perfil –release: ativa otimizações de produção do compilador Rust (-O3), incluindo *inlining*, vetorização de laços e reordenação de instruções.
- *Link-Time Optimization* (LTO): com lto=true, o *linker* otimiza globalmente todos os módulos como um único programa.
- Unidades de Geração de Código: configuradas como codegen-units=1, aumentam o potencial de otimização ao eliminar paralelismo durante a compilação.

3.1.1 ***Implementação Baseline.*** Nessa versão, foram implementadas interfaces (*traits* em *Rust*) para todos os módulos do sistema. A arquitetura é composta por dois módulos centrais: o *TransactionProcessor*, responsável por orquestrar a lógica de negócio; e o *Ledger*, que atua como camada de persistência em memória.

Para servir como *baseline*, o *Ledger* foi implementado com duas características de implementações ingênuas:

- Contenção de Lock: o uso de um *RwLock* global para proteger o *HashMap* de contas cria um gargalo, já que todas as requisições gRPC precisam do *write lock*, serializando o processamento.
- Layout de Dados: a estrutura *Account* armazena o histórico de transferências (transaction_history: Vec<HistoricTransfer>) na *heap*, resultando em baixa localidade de dados e acesso a regiões de memória distantes.

3.1.2 ***Implementação Otimizada.*** Para a implementação otimizada, foram realizadas refatorações arquiteturais visando eliminar os gargalos identificados.

- Otimização de Layout de Dados: a estrutura *Account* passou a armazenar apenas os identificadores Vec<Uuid> das transações, reduzindo o tamanho da estrutura *hot* e aumentando a capacidade da linha de cache L1.
- Eliminação do *Lock* Global: o mapa do módulo *Ledger* foi substituído pelo *DashMap*, que implementa *sharding* de *locks*, permitindo alterar métodos de &mut self para &self e habilitar processamento paralelo.
- Semântica *Zero-Copy*: o fluxo de transferência foi reescrito para mutações *in-place* nos *shards*, eliminando clonagens desnecessárias em memória.

3.2 Testes de Carga

Para simular um cenário real de alta demanda, foi desenvolvido um cliente de carga sintética utilizando o *runtime* assíncrono da ferramenta *Tokio*. O perfil de tráfego foi modelado para refletir um sistema transacional intensivo, adotando uma distribuição probabilística onde 10% das operações correspondem à criação de contas e chaves, 10% a depósitos, e 80% a transferências entre diferentes contas.

A seleção das contas alvo é realizada de forma aleatória sobre o universo de contas existentes. Essa escolha representa o pior caso para a hierarquia de memória: ao impedir a previsibilidade de acesso e a localidade temporal artificial, força-se o processador a depender da eficiência do *layout* de dados.

Os testes foram executados com alto grau de paralelismo, configurando o cliente para manter 512 *tasks* simultâneas, enviando requisições ininterruptamente. Esse volume de concorrência foi dimensionado para garantir a saturação dos recursos do servidor.

Para medir a eficiência arquitetural, a avaliação concentrou-se na coleta de métricas de *hardware* de baixo nível. Utilizou-se a ferramenta *perf* para coletar os *Performance Monitoring Counters* (PMCs) da CPU. As métricas coletadas foram:

- *Instructions Per Cycle* (IPC): razão entre instruções executadas e ciclos de processamento, principal indicador de eficiência.
- Taxa de *Cache Miss*: derivada dos contadores cache-references e cache-misses, focando no cache de dados de Nível 1.

4 Resultados

Para garantir a consistência e facilitar a reprodutibilidade dos resultados, todos os testes foram conduzidos em um único ambiente computacional: processador AMD Ryzen 9 7900, 32 GB de memória RAM DDR5 e sistema operacional Fedora Linux 42.

O sistema *baseline* e otimizado foram postos sob carga intensa por 10 minutos, com o *perf stat* anexado ao processo.

Os resultados da versão *baseline* revelaram um *cache miss* de 43,6%, ou seja, que aproximadamente metade de todas as tentativas de acesso à memória falharam no *cache*, forçando a CPU a buscar dados em níveis inferiores da hierarquia. Isso corrobora as observações de [14] sobre o impacto de grandes *footprints* de dados. Também foi observado um IPC de 0,48, considerado baixo por situar-se na faixa de cargas de trabalho limitadas por latência [14], ou seja, a CPU está completando menos de meia instrução por ciclo, passando a maior parte do tempo em *stall*.

Já os resultados da versão otimizada mostraram um *cache miss* de apenas 6,7%, isto é, que o sistema passou a acertar quase todos acesso à memória cache L1, comprovando a melhora da localidade espacial por conta das estratégias de otimização propostas. Também foi observado um IPC de 0,64, que equivale a um ganho de 33% em relação a versão *baseline*. Embora um IPC inferior a 1,0 possa parecer modesto, ele é consistente com a média observada em cargas de trabalho reais de *datacenters* [14]. Isso evidencia uma transição de um sistema limitado por latência para um sistema mais limitado por capacidade de processamento.

A Tabela 1 resume os resultados comparativos.

Tabela 1: Comparação de métricas entre implementações

Métrica	Baseline	Otimizada
Taxa de Cache Miss	43,6%	6,7%
IPC	0,48	0,64
ΔMiss	–	-84,6%
ΔIPC	–	+33,3%

A luz da literatura, os resultados obtidos reforçam conclusões referentes a sistemas de larga escala, como as observações de [8, 14], que apontam a latência de memória como o principal limitador de *throughput* em arquiteturas modernas. O comportamento observado na versão *baseline* (IPC baixo e alta taxa de *cache misses*), reproduz exatamente o perfil de aplicações *memory-bound* descrito nesses estudos.

Embora o sistema implementado seja uma abstração simplificada do Pix, os resultados sugerem que sistemas financeiros reais, altamente transacionais, podem sofrer penalidades substanciais caso adotem arquiteturas ingênuas. A redução de 84,6% nos *cache misses* e o aumento de 33% no IPC indicam que otimizações de *layout* e concorrência não apenas melhoram desempenho, mas também reduzem a necessidade de superprovisionamento de *hardware*.

Apesar dos ganhos expressivos, é importante reconhecer que o ambiente experimental foi controlado e não inclui fatores presentes em sistemas reais, como latência de rede, persistência em *storage* distribuído, replicação, auditoria e mecanismos de antifraude. Outro ponto relevante é que as otimizações aplicadas focaram exclusivamente na hierarquia de memória de dados, isto é, não foram exploradas técnicas como reorganização de código para melhorar a *cache* de instruções, afinidade de *threads*, etc.

5 Conclusão

Este estudo mostrou de forma quantitativa como decisões arquiteturais de *software* impactam diretamente a eficiência de utilização do *hardware* subjacente. A análise comparativa entre as implementações *baseline* e otimizada confirmou que a contenção de *locks* e a baixa localidade de dados atuam como gargalos severos em sistemas de alto desempenho.

As otimizações realizadas, focadas na eliminação do bloqueio global via *lock sharding* e na reestruturação do *layout* de memória, provaram-se eficazes. A redução da taxa de *cache misses* de 43,60% para 6,70% evidencia que a compactação das estruturas de dados

permitiu um uso mais eficiente da hierarquia de memória. Simultaneamente, o aumento do IPC demonstra que o sistema transicionou de um perfil limitado por latência para um perfil mais limitado por capacidade de processamento real.

Esses achados têm implicações diretas para a indústria, especialmente em setores que dependem de baixa latência e alta previsibilidade, como o financeiro. Em tais contextos, otimizações de cache e concorrência podem reduzir custos operacionais, aumentar a resiliência sob carga extrema e evitar a necessidade de escalar horizontalmente apenas para compensar ineficiências de *software*.

Como continuidade natural deste trabalho, investigações futuras podem explorar técnicas complementares, como otimizações de *cache* de instruções, afinidade de *threads*, estratégias *Non-Uniform Memory Access* (NUMA) *aware* e mecanismos de *prefetching* explícito. Além disso, a replicação do experimento em ambientes distribuídos permitiria avaliar o impacto dessas otimizações em sistemas reais de missão crítica. Assim, este estudo não apenas quantifica ganhos, mas também abre caminho para oportunidades de pesquisa em *software* consciente de *hardware*.

Agradecimentos

Este trabalho foi apoiado pelo Instituto Federal Sul-rio-grandense (IFSul).

Referências

[1] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, page 162, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137419. doi: 10.1145/977091.977115. URL https://doi.org/10.1145/977091.977115.

[2] Bruce Yellin. Saving the future of moore's law. Technical report, Dell Inc., 2019. URL https://learning.dell.com/content/dam/dell-emc/documents/en-us/2019KS_Yellin-Saving_The_Future_of_Moores_Law.pdf. Knowledge Sharing Article.

[3] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. The memory hierarchy. https://diveintosystems.org/book/C11-MemHierarchy/mem_hierarchy.a, 2023. Chapter from the online book "Dive Into Systems". Accessed: October 16, 2025.

[4] Ranjani Parthasarathi. *Computer Architecture: Engineering And Technology*. INFLIBNET Centre, 2018.

[5] Apostolos Kotsiolis. Simultaneous multithreading: Driving performance and efficiency on amd epyc cpus, 2025. URL https://www.amd.com/en/blogs/2025/simultaneous-multithreading-driving-performance-a.html.

[6] Asif Ali Khan, João Paulo C. De Lima, Hamid Farzaneh, and Jeronimo Castrillon. The landscape of compute-near-memory and compute-in-memory: A research and commercial overview, 2024. URL https://arxiv.org/abs/2401.14428.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[8] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. Optimizing cpu performance for recommendation systems at-scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589112. URL https://doi.org/10.1145/3579371.3589112.

[9] Junjie Peng, Jinbao Chen, Shuai Kong, Danxu Liu, and Meikang Qiu. Resource optimization strategy for cpu intensive applications in cloud computing environment. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 124–128, Beijing, China, 2016. IEEE. doi: 10.1109/CSCloud.2016.29.

[10] Markus Kowarschik and Christian Weiß. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, pages 213–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-36574-7. doi: 10.1007/3-540-36574-5_10. URL https://doi.org/10.1007/3-540-36574-5_10.

[11] Amazon Web Services. What is the difference between grpc and rest? https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/, 2025. Accessed: October 16, 2025.

[12] Alexandros Tasos, Juliana Franco, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. Reshape your layouts, not your programs: A safe language extension for better cache locality. *Science of Computer Programming*, 197:102481, 2020. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2020.102481. URL https://www.sciencedirect.com/science/article/pii/S0167642320300915.

[13] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[14] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169. IEEE, 2015. doi: 10.1145/2749469.2750392.