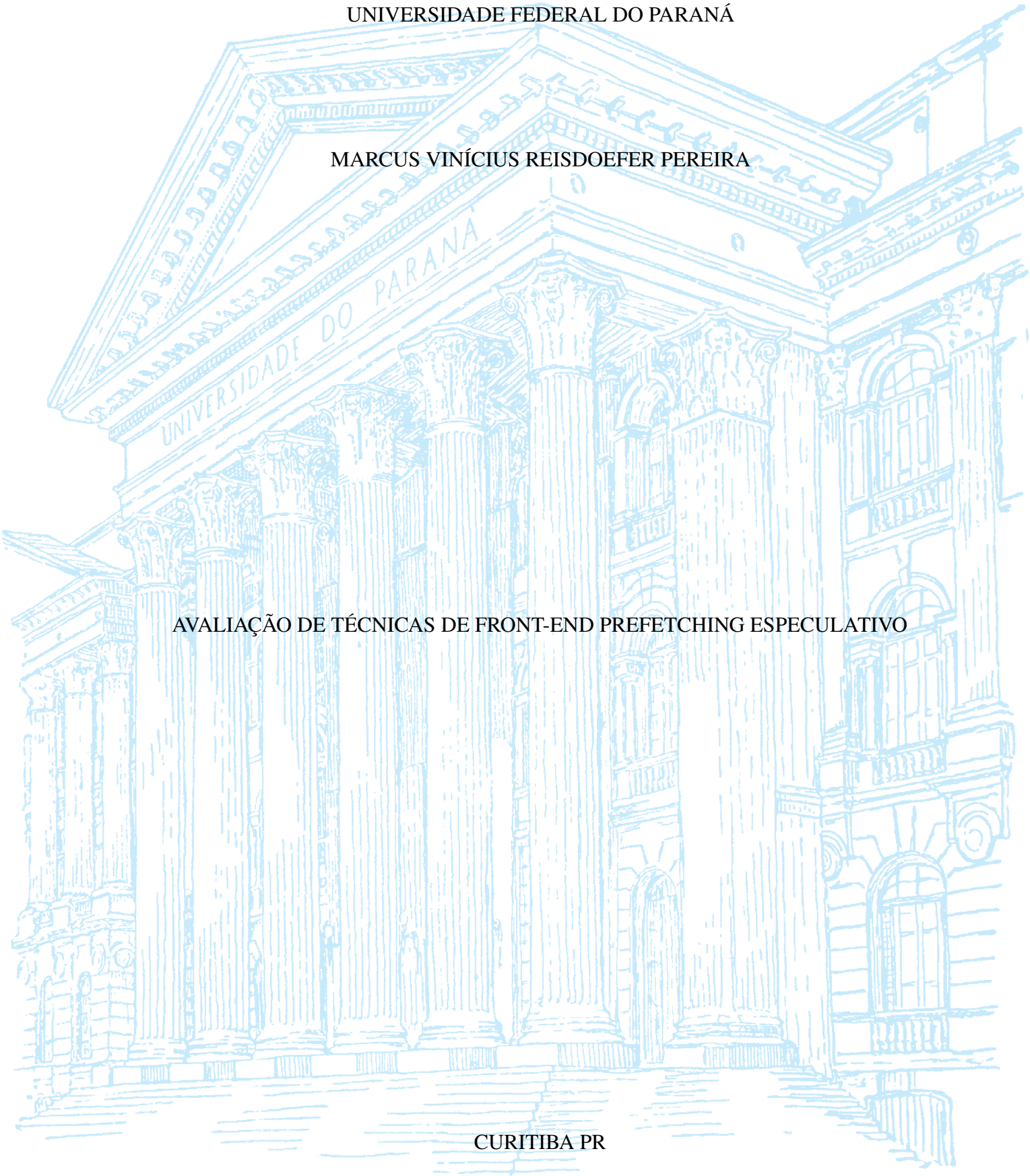
UNIVERSIDADE FEDERAL DO PARANÁ

MARCUS VINÍCIUS REISDOEFER PEREIRA



AVALIAÇÃO DE TÉCNICAS DE FRONT-END PREFETCHING ESPECULATIVO

CURITIBA PR

2024

MARCUS VINÍCIUS REISDOEFER PEREIRA

AVALIAÇÃO DE TÉCNICAS DE FRONT-END PREFETCHING ESPECULATIVO

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: Marco Antonio Zanata Alves.

CURITIBA PR

2024

Universidade Federal do Paraná
Setor de Ciências Exatas
Curso de Ciência da Computação

Ata de Apresentação de Trabalho de Conclusão de Curso 2

Título do Trabalho: AVALIAÇÃO DE TÉCNICAS DE FRONT-END PREFETCHING ESPECULATIVO

Autor: GRR 20211759 Nome: MARCUS VINÍCIUS REISDOEFER PEREIRA

Apresentação: Data: 19/12/2024 Hora: 13h30 Local: Laboratório HiPES - DINF

Orientador: Marco Antonio Zanata Alves _____

Membro 1: Simone Dominico _____

Membro 2: Rodrigo Machniewicz Sokulski _____

(nome) (assinatura)

AVALIAÇÃO – Produto escrito		ORIENTADOR	MEMBRO 1	MEMBRO 2	MÉDIA
Conteúdo	(00-40)				
Referência Bibliográfica	(00-10)				
Formato	(00-05)				
AVALIAÇÃO – Apresentação Oral					
Domínio do Assunto	(00-15)				
Desenvolvimento do Assunto	(00-05)				
Técnica de Apresentação	(00-03)				
Uso do Tempo	(00-02)				
AVALIAÇÃO – Desenvolvimento					
Nota do Orientador	(00-20)		**********	**********	
NOTA FINAL		**********	**********	**********	**90**

Os pesos indicados são sugestões.

Conforme decisão do colegiado do curso de Ciência da Computação, a entrega dos documentos comprobatório de trabalho de Conclusão de Curso 2 deve deve respeitar os seguintes procedimentos: o orientador deve abrir um processo no Sistema Eletrônico de Informações (SEI – UFPR); Selecionar o tipo: *Graduação: Trabalho Conclusão de Curso*; informar os interessados: nome do aluno e o nome do orientador; anexar esta ata escaneada e a versão final do PDF da monografia do aluno; Tramitar o processo para CCOMP (Coordenação de Ciência da Computação).

A alguém...

AGRADECIMENTOS

Agradeço à minha namorada Anna Flora por todos os momentos de acolhimento, por me ouvir explicar não só o projeto inteiro, mas todas as infinitas informações aleatórias que solto sobre Computação para ela. Sou grato por dividir horas e horas de estudo, atenção e dedicação comigo, e também por ser minha referência de uma pessoa extremamente carinhosa e dedicada.

Também agradeço aos meus pais, os maiores responsáveis pela oportunidade de fazer a graduação podendo focar 100% nos estudos, que sempre deram prioridade à minha educação. Meus irmãos também, por me incentivarem sem ter a menor ideia do que está acontecendo, e na prática estão tão perdidos quanto eu.

Finalmente, mas não menos importante, agradeço todo o corpo docente e funcionários do Departamento de Informática pela dedicação e pela oportunidade de aprender. Em especial gostaria de agradecer ao Professor Zanata pela paciência inigualável durante toda a orientação.

RESUMO

Nos últimos anos a velocidade dos processadores aumentou rapidamente, enquanto os avanços na velocidade da memória principal têm sido mais moderados. Esta discrepância cria uma barreira significativa, de modo que o processador frequentemente pausa a execução esperando os dados enquanto a memória é acessada, prejudicando o desempenho geral do sistema. Para mitigar a latência decorrente das diferentes velocidades do processador e da memória, técnicas como *prefetching* de dados se tornam mais relevantes, além de métodos já existentes como algoritmos de substituição de *cache*, execução fora de ordem e paralelismo em nível de memória. Nesse contexto, sabendo que as técnicas existentes reduzem a latência aparente menor do que a latência real da memória, com o objetivo de reduzir ainda mais essa latência aparente este trabalho propõe avaliar a viabilidade de implementar técnicas de *prefetching* especulativo de dados mais cedo no *pipeline* do processador. Enquanto *prefetchers* normalmente são implementados quando já se tem o endereço de memória a ser acessado e fazem uma especulação do próximo, esse trabalho visa especular o endereço de acesso antes mesmo do primeiro ser decodificado. Para essa análise, foi utilizado um simulador arquitetural com precisão de ciclo, que permite uma modelagem detalhada do comportamento do *hardware*. As simulações foram realizadas utilizando *benchmarks* do SPEC (Standard Performance Evaluation Corporation) CPU 2017, que replicam a carga de trabalho uma vasta gama prática de hardware, utilizando cargas de trabalho desenvolvidas a partir de aplicações reais, para fornecer uma medida comparativa da performance de computação de alto desempenho. Com o simulador, foram avaliadas a eficiência e o ganho potencial do *hardware* de processadores ao aplicar essa nova técnica de *prefetching* em relação a métodos existentes para determinar a viabilidade dessa nova abordagem, obtendo um aumento no IPC (Instruções Por Ciclo) de aproximadamente 7.2% e uma redução de aproximadamente 6.5% na taxa de *miss* da *cache*. Ainda, foi estudada a possibilidade de implementação desse *front-end prefetcher* como alternativa ao aumento do tamanho do ROB (Re-Order Buffer). Ao final são apresentadas conclusões sobre os experimentos realizados, suas limitações e possíveis trabalhos futuros.

Palavras-chave: Pré-busca no front-end. Pré-busca. Memória Cache. Simulador. Barreira de Memória.

ABSTRACT

In recent years, the speed of processors has increased rapidly, while advancements in main memory speed have been more moderate. This discrepancy creates a significant barrier, causing the processor to pause execution often while waiting for data as memory is accessed, negatively impacting overall system performance. To mitigate the latency arising from the different speeds of the processor and memory, techniques such as data prefetching are becoming more relevant, alongside existing methods like cache replacement algorithms, out-of-order execution, and memory-level parallelism. In this context, knowing that existing techniques reduce the apparent latency less than the actual memory latency, this work proposes to evaluate the feasibility of implementing speculative data prefetching techniques earlier in the processor pipeline. While prefetchers are typically implemented once the memory address is known and speculate on the next one, this work aims to speculate the access address before even the first one is decoded. A cycle-accurate architectural simulator was used for this analysis, allowing a detailed modeling of hardware behavior. The simulations were conducted using benchmarks from the SPEC (Standard Performance Evaluation Corporation) CPU 2017, which replicates the load of a wide range of practical hardware, using workloads developed from real-world applications to provide a comparative measure of high-performance computing. With the simulator, the efficiency and potential gain of this new prefetching technique were evaluated in comparison to existing methods to determine the viability of this approach, resulting in an IPC (Instructions Per Cycle) increase of approximately 7.2% and a reduction of about 6.5% in the cache miss rate. Additionally, the possibility of implementing this front-end prefetcher as an alternative to increasing the ROB (Re-order Buffer) size is assessed. Finally, conclusions are drawn from the experiments, their limitations, and potential future work.

Keywords: Front-end prefetching. Prefetching. Cache Memory. Simulator. Memory Barrier.

LISTA DE FIGURAS

1.1 Histórico de características de performance da DRAM. 11
1.2 Visualização de pontualidade de pré-buscas. 12

2.1 Histórico de características de performance da DRAM. 15
2.2 Visualização de grau e distância de pré-buscas. 16
2.3 Visualização de pontualidade de pré-buscas. 17
2.4 Visualização de pré-buscas “just-in-time”. 17
2.5 Fluxo de uma instrução.. 19
2.6 Histórico do tamanho do Re-Order Buffer nas arquiteturas da Intel e seu impacto no tempo entre *decode* e *execute* de instruções *load*.. 20

5.1 Diagrama de blocos do Intel Skylake Server (WikiChip, 2017). 30
5.2 Histograma mostrando distribuição de atraso de requisições à memória (*prefecther* perfeito).. 31
5.3 Histograma mostrando distribuição de atraso de requisições à memória (sem *prefecther*). 31
5.4 Histograma mostrando distribuição de atraso de pré-buscas (*pipeline nextline*). . 32
5.5 Histograma mostrando distribuição de atraso de pré-buscas (*cache nextline*). . . . 33
5.6 Gráfico de linhas comparando atraso de pré-buscas entre *prefetchers cache nextline* e *front-end nextline*.. 33
5.7 Latência média obtida nos traços executados. 35
5.8 IPC observado nos traços executados. 36

LISTA DE TABELAS

2.1 Classificação de utilidade de pré-buscas ordenada por utilidade.. 17

3.1 Classificação dos trabalhos correlatos. 1: Eickemeyer e Vassiliadis (1993), 2: González e González (1997), 3: Efnusheva et al. (2017), 4: Alves et al. (2021), 5: Gerogiannis e Torrellas (2024), 6: Yang et al. (2024). 23

5.1 Configuração do sistema base. 29

5.2 Comparação de métricas entre *prefetchers nextline* (*frontend* vs. *cache miss*). . . 34

5.3 Comparação de métricas entre os diferentes tamanhos de ROB e as diferentes implementações de *prefetcher*. (* Tamanho original nos demais experimentos.) . 37

LISTA DE ACRÔNIMOS

CPU	*Central Processing Unit*
DRAM	*Dynamic Random-Access Memory*
HiPES	*High Performance and Efficient Systems*
HPC	*High-Performance Computing*
MAB	*Multi-Armed Bandit*
MOB	*Memory Order Buffer*
OrCS	*Ordinary Computer Simulator*
RAM	*Random-Access Memory*
ROB	*Re-Order Buffer*
SPEC	*Standard Performance Evaluation Corporation*

SUMÁRIO

1 INTRODUÇÃO 11
1.1 PROBLEMA 11
1.2 MOTIVAÇÃO 12
1.3 OBJETIVOS 13
1.4 PRINCIPAIS CONTRIBUIÇÕES 14
1.5 ORGANIZAÇÃO DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 TÉCNICAS DE *HARDWARE PREFETCHING* 17
2.2 OUT-OF-ORDER EXECUTION 18
2.3 DEFININDO MÉTRICAS RELEVANTES 20
3 CORRELATOS E ESTADO DA ARTE 22
3.1 SUMÁRIO DOS TRABALHOS RELEVANTES 22
4 PROPOSTA DE EXPLORAÇÃO E ANÁLISE DO PROBLEMA 24
4.1 PIPELINE FRONT-END PREFETCHER 24
4.1.1 Prefetchers existentes 24
4.2 PROVA DE CONCEITO 24
4.2.1 Multi-Armed-Bandit 25
4.2.2 Multi-Armed Prefetcher 25
5 EXPERIMENTAÇÃO E VALIDAÇÃO 28
5.1 MÉTODO 28
5.2 VERIFICAÇÃO DE OPORTUNIDADES DE PRÉ-BUSCAS 28
5.2.1 Comparação com prefetching regular 32
5.3 ANÁLISE DE PERFORMANCE 34
5.3.1 Multi-Armed Prefetcher 34
5.3.2 Experimentos 35
5.3.3 Exploração do espaço de projeto 36
5.4 LIMITAÇÕES 37
5.4.1 Limitações dos experimentos 38
5.4.2 Limitações do simulador 38
6 CONCLUSÕES E TRABALHOS FUTUROS 39
REFERÊNCIAS 40

1 INTRODUÇÃO

Na década de 1970 foram criadas as memórias do tipo DRAM (*Dynamic Random-Access Memory*), baseada nas células de memória de um transistor inventadas em 1966 por Robert H. Dennard. Um grande avanço que permitiu uma densidade de células de memória muito maior do que designs anteriores de RAM (*Random-Access Memory*). Entretanto, mesmo sendo um salto de tecnologia importante, a diferença de velocidade entre as memórias e os processadores não se reduziu, mas na verdade aumentou com o passar dos anos (Chang, 2017).

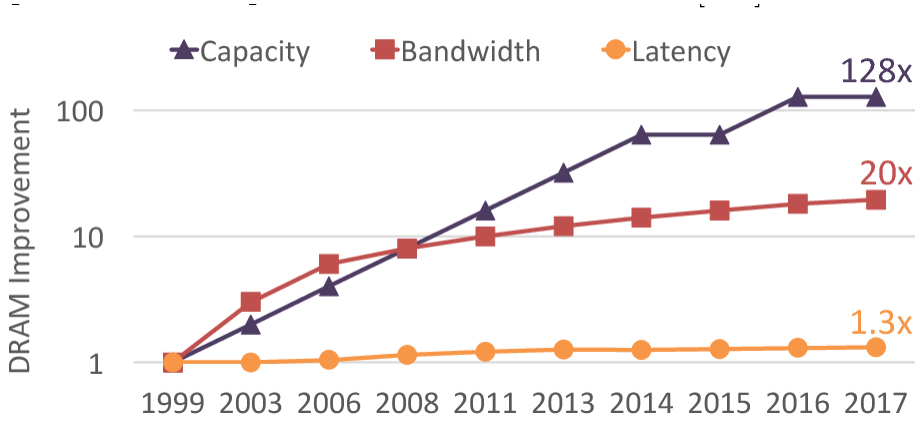


Figura 1.1: Histórico de características de performance da DRAM.
OGAWA, Tadashi. "Understanding and Improving the Latency of DRAM-Based Memory Systems", 2017.

O gráfico da Figura 1.1 mostra graficamente a evolução ao longo dos anos em termos da capacidade, largura de banda e latência da memória. Observa-se que a latência da memória evoluiu muito pouco ao longo dos anos, se comparada com suas outras características.

O descompasso crescente entre a velocidade do processador e da memória é a origem do efeito conhecido como a ***memory wall***. Tal fenômeno subutiliza o processador e leva a um aumento no tempo de processamento, visto que o processador tende a passar diversos ciclos de *clock* esperando pelos dados para toda operação de leitura ou escrita, e isso se torna um gargalo crítico em sistemas de alta performance (Efnusheva et al., 2017).

Diversas técnicas foram propostas no decorrer dos anos a fim de mitigar a discrepância entre as velocidades do processador e da memória. Como por exemplo, a invenção das memórias *caches* que são fundamentais para a performance dos processadores modernos. Ou ainda o paradigma de *Out-of Order Execution* que se tornou essencial para processadores superescalares, além dos *prefetchers*. Essas técnicas levam a uma latência aparente menor do que a latência real da memória. Nesse sentido, o foco deste trabalho é na aplicação das técnicas de *prefetching* em *hardware*.

1.1 PROBLEMA

Mesmo com a invenção das técnicas citadas anteriormente observa-se que a latência de memória ainda é um problema crítico para sistemas de alto desempenho. Estudos recentes, na área de bancos de dados, por exemplo, avaliam o uso de técnicas de *prefetching* em bancos de dados em memória (Zhang et al., 2024) visando mitigar a influência da latência da memória em sistemas de alto desempenho.

A Figura 1.2 fornece uma visualização ilustrando a pontualidade temporal de pré-buscas, que é crucial para manter sua performance. Observa-se que *prefetchers* que possuem uma grande porcentagem de pré-buscas atrasadas ou muito adiantadas afetam negativamente a performance geral do processador ao desperdiçar recursos e tempo de execução dessas operações.

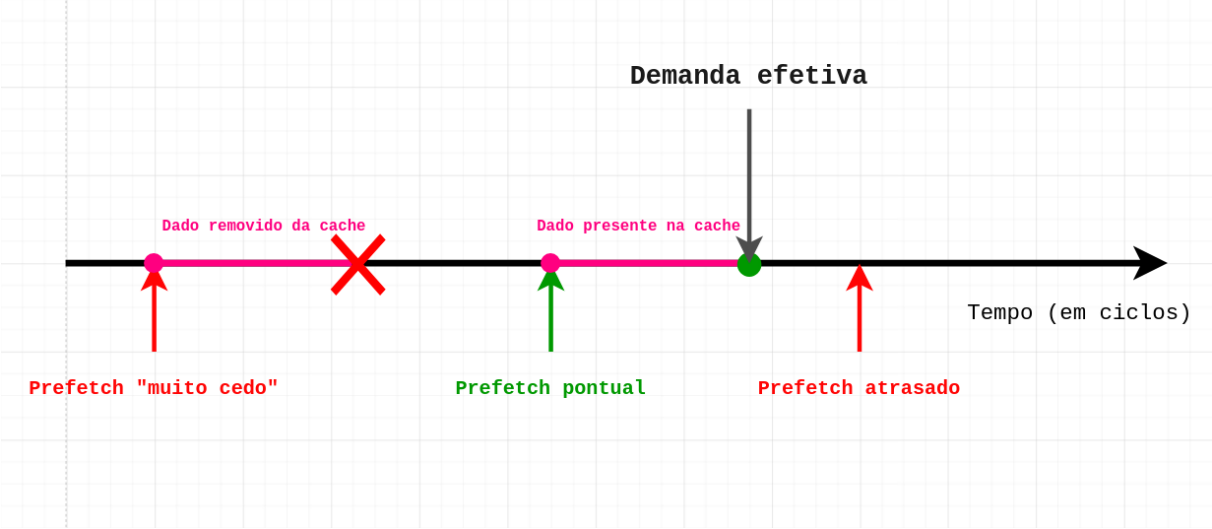


Figura 1.2: Visualização de pontualidade de pré-buscas.

Sendo assim, acredita-se que a introdução de uma nova técnica, buscando por oportunidades não exploradas nos processadores modernos, possa mitigar ainda mais a latência da memória a fim de melhorar a performance de sistemas de alto desempenho modernos.

1.2 MOTIVAÇÃO

Sabendo que *prefetchers* implementados em *hardware* são uma técnica viável para reduzir a latência aparente da memória, esse trabalho visa explorar uma nova oportunidade de implementação de pré-busca de dados, sendo esta diretamente no *front-end* do processador.

Observa-se que mesmo técnicas conhecidas de *prefetching* podem falhar em esconder a latência da memória caso o dado não esteja na *cache* e precise ser buscado na memória principal, por exemplo, escondendo apenas parcialmente a latência da requisição. Com o intuito de reduzir a latência aparente mesmo nesses casos, propõe-se implementar a pré-busca de forma especulativa ainda no estágio de *fetch* do processador.

Dessa forma, a abordagem do *prefetcher* mudará. Ao invés de um acesso efetivo acionar buscas de endereços especulativos que estima-se que ocorrerão no futuro. Com a nossa abordagem bastaria a predição do acesso do *load* atual (o qual ainda está sendo decodificado) para que o *prefetcher* possa buscar esse novo endereço.

Esse acesso poderá proporcionar ainda assim uma janela de tempo para esconder a latência da memória. Em nossos experimentos observamos que o tempo médio entre uma instrução *load* ser decodificada e ser efetivamente executada é mais de 200 ciclos.

Prefetchers tradicionais são implementados para monitorar o acesso da hierarquia de memória, geralmente entre níveis de *cache*. Este novo *prefetcher* seria implementado no *decode* de instruções (no *front-end* do processador), e seria totalmente especulativo, já que inicialmente ele não teria como saber o endereço de memória a ser acessado, mas seu treinamento pode ser melhorado conforme os endereços de memória são devidamente decodificados (no estágio de execução) e os endereços podem ser informados ao *prefetcher* para ajustar suas buscas.

A grande novidade de nossa abordagem é que os métodos tradicionais de *prefetch* deixam de utilizar uma informação importante, que é a direção dos saltos. Quando se implementa o *prefetcher* no estágio de *decode*, em processadores modernos está se tentando prever centenas de instruções no futuro, as quais foram selecionadas por preditores de salto bastante eficientes. Logo, mudanças no fluxo de execução do código serão naturalmente entendidas por um *prefetcher* implementado no estágio de *decode*.

Isso apresenta seus próprios desafios, dos quais destacam-se:

- Especulação do endereço de memória: Como o endereço a ser buscado sequer foi decodificado (é fácil determinar que há uma leitura na memória após a decodificação da instrução, mas é difícil dizer o endereço sem passar pela execução) é necessário realizar um acesso especulativo na *cache*, o que pode prejudicar a performance caso os acessos sejam feitos a endereços errados.

- Buscas *too-early*: Por mais que mover o *prefetcher* para o início do *pipeline* apresente mais oportunidades de esconder a latência da memória, o risco de que o dado buscado seja substituído na *cache* antes mesmo de ser utilizado também aumenta, afinal a janela de tempo para ele ser retirado da *cache* é a mesma que também será dilatada.

O ponto principal, e motivador do trabalho, é que o *pipeline* dos processadores modernos está cada vez mais profundo e o *Re-Order Buffer* também está crescendo, e quando uma instrução *load* trava o ROB por ter que esperar dados da memória a performance do processador também é afetada de forma negativa.

Por consequência desses fatores o tempo entre ler uma instrução *load* da *cache* de instruções e ela efetivamente executar a requisição à *cache* de dados / enviá-la ao *Memory Order Buffer* está grande. Sendo assim, observa-se que existe uma oportunidade de melhorar *prefetchers* ao adiantar as buscas de dados para o *front-end* do *pipeline*, com o objetivo de aproveitar essa janela de tempo entre o *fetch* e o *execute* de instruções de leitura de memória.

1.3 OBJETIVOS

Dessa forma o objetivo final é fazer uma análise sobre a viabilidade e potencial de implementação de um *prefetcher* no *front-end* do processador, e estudar os aspectos que envolvem a adoção dessa técnica de pré-busca de dados mais cedo no *pipeline*. Ainda, é possível resumir os sub-objetivos do trabalho da seguinte maneira:

- Analisar o impacto do aumento do ROB no tempo para executar uma instrução, e as consequências disso para a implementação do *front-end prefetcher*.

- Estudar o **potencial** do *prefetcher* ideal (que acerta 100% das especulações de endereços de memória) no *front-end* no processador.

- Avaliar a **possibilidade de implementação** desse *hardware* fazendo pré-buscas de dados mais cedo no *pipeline*.

Ressalta-se, no entanto, que a proposta de um *hardware* específico para *prefetching* está fora do escopo desta monografia e será considerada um possível trabalho futuro. A análise aqui apresentada concentra-se nos aspectos gerais da viabilidade e do potencial de um *prefetcher* no *front-end* do processador.

1.4 PRINCIPAIS CONTRIBUIÇÕES

Este trabalho apresenta uma análise aprofundada sobre o impacto de diferentes estratégias de otimização no desempenho de processadores modernos, com foco em dois aspectos cruciais: o comportamento do Re-Order Buffer (ROB) e a atuação do *prefetcher* no *front-end* do *pipeline*. Dentre as principais contribuições dessa monografia, destacam-se 3:

1. Quantificar o impacto do aumento do tamanho do ROB (Re-Order Buffer) em processadores modernos quanto ao tempo entre o *fetch* e a execução de instruções.

2. Estudar o **potencial** do *prefetcher* ideal (que acerta 100% das especulações de endereços de memória) no *front-end* no processador.

3. Mostrar que é possível implementar tal *prefetcher* como alternativa ao aumento do tamanho do ROB para ganho de performance.

1.5 ORGANIZAÇÃO DO TRABALHO

No Capítulo 2 será apresentada a fundamentação teórica para a elaboração do trabalho, assim como serão discutidos conceitos fundamentais relacionados às técnicas de *prefetching*.

O Capítulo 3 estudará o estado da arte dos *prefetchers* existentes, e irá abordar como reimplementá-los utilizando a nova abordagem.

O Capítulo 4 será dedicado a descrever como serão implementados os *prefetchers*, o simulador utilizado e os objetivos propostos pelos experimentos, assim como a maneira que serão executados.

No Capítulo 5 apresenta os experimentos e verificações dos resultados gerados pelo processo. Considerações sobre viabilidade e outras análises são feitas nesse capítulo.

Finalmente, no Capítulo 6 serão expostas as considerações finais sobre o desenvolvimento e os resultados do trabalho, assim como sugestões para trabalhos futuros sobre o tema.

2 FUNDAMENTAÇÃO TEÓRICA

Conforme estudos feitos por Chang (2017), com resultados observáveis na Figura 2.1, a latência de acesso a dados da DRAM não melhorou tão significativamente se comparada com a banda de memória ou a capacidade, e menos ainda se comparada com a velocidade dos processadores, o que prejudica o desempenho do computador como um todo, pois o processador perde tempo esperando a resposta da memória.

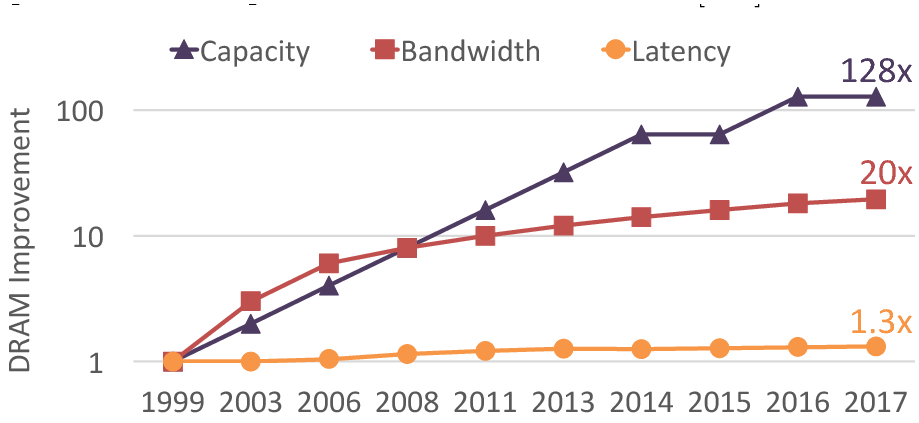


Figura 2.1: Histórico de características de performance da DRAM.
OGAWA, Tadashi. "Understanding and Improving the Latency of DRAM-Based Memory Systems", 2017.

Para mitigar esse problema, técnicas como o uso de *prefetchers* têm sido amplamente adotadas. Técnicas de *prefetching* se baseiam em antecipar as operações de leitura de dados, requisitando-os para a memória *cache* antes mesmo que sejam requisitados pelo código em execução no processador. Ao prever com eficiência os padrões de acesso, elas podem reduzir a latência percebida pelo processador. Dessa forma, os *prefetchers* desempenham um papel crucial no aumento do desempenho, ao minimizar os impactos negativos do tempo de resposta da hierarquia de memória.

No entanto, isso tem um custo em termos de largura de banda, já que o *prefetching* consome espaço na *cache* e também consome largura de banda de memória. A otimização envolve equilibrar o aumento do uso da largura de banda com a redução da latência, visando melhorar o desempenho geral do sistema, especialmente em cenários onde os padrões de acesso a dados são previsíveis e consistentes.

É possível realizar comparações entre os ganhos de se aumentar o paralelismo da DRAM, ou a largura de banda, e a redução da latência. Por mais que essa seja mais uma dimensão para exploração do problema de *prefetching*, essa comparação está fora do escopo dessa monografia.

Separam-se as maneiras de fazer *prefetching* em duas: por *software* e por *hardware*. O *prefetching* realizado por *software* engloba incluir instruções de *prefetch* diretamente no programa, seja pelo programador ou pelo compilador, como uma instrução *load* que não causa *stalls*, ou seja sem carregar o dado para o banco de registradores. Já os *prefetchers* em *hardware* são circuitos do processador responsáveis por monitorar o padrão de acesso à memória e predizer os próximos endereços a serem buscados, mas devem ser feitos de maneira conservadora, pois o custo de pré-buscar dados que não são utilizados pode ser alto.

No geral, *prefetchers* de *hardware* são confiáveis para acessos regulares à memória, enquanto *prefetching* por *software* é utilizado como alternativa quando os padrões de acesso

são extremamente irregulares e a aplicação requer alto desempenho com relação à memória. Isso é possível pois a princípio quem desenvolve a aplicação (ex.: um sistema de gerenciamento de bancos de dados) conseguem estimar onde ocorrem os acessos à *cache*, e podem otimizar o tempo de espera realizando chamadas assíncronas.

Ainda, outras duas classificações para *prefetchers* que se referem a sua adaptabilidade são: *prefetchers* estáticos e *prefetchers* dinâmicos. Resumidamente, *prefetchers* estáticos utilizam regras pré-definidas para as requisições (ex.: pegar a próxima linha da *cache*). Já os *prefetchers* dinâmicos utilizam informações do tempo de execução para as decisões (ex.: *prefetchers* baseados em *buffers* de histórico). É possível também implementar um modelo híbrido de *prefetching*, como em alguns processadores modernos, de modo que quando o padrão de acesso é bastante regular as regras de um *prefetcher* estático são utilizadas, e as regras dinâmicas são utilizadas apenas quando o padrão de acesso se torna mais irregular.

Também ressalta-se o conceito de *prefetch degree* (grau de pré-busca), que está relacionado a agressividade dos *prefetchers* e é visualizado na Figura 2.2, onde cada círculo representa uma i-ésima demanda efetiva à memória. Essencialmente, o grau de uma pré-busca define a quantidade de dados ou a quantidade de requisições que uma única pré-busca antecipa. Já a distância se refere a quantos acessos "a diante" a pré-busca tenta prever.

Um aumento no grau de pré-busca pode reduzir os tempos de latência, pois a previsão de acessos subsequentes se torna mais assertiva. Contudo, existe uma relação direta entre o grau de pré-busca e a agressividade do sistema. Quando o grau de pré-busca é elevado de maneira agressiva, o sistema tende a buscar mais dados do que o necessário, o que pode resultar em desperdício de recursos, aumento de tráfego de memória e até sobrecarga na *cache*, prejudicando o desempenho geral. Portanto, equilibrar o grau de pré-busca e a agressividade é essencial para maximizar a eficiência sem incorrer em custos adicionais desnecessários.

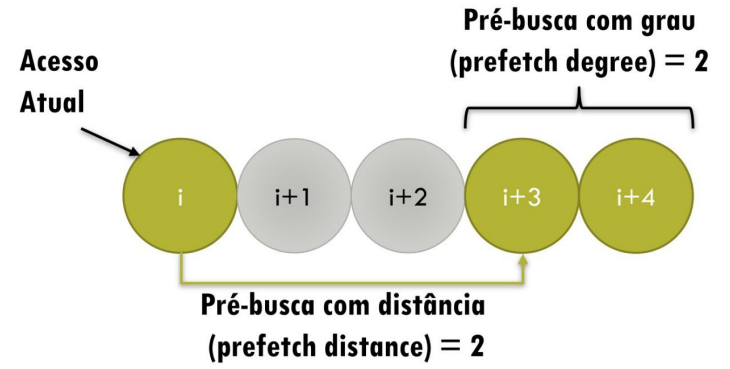


Figura 2.2: Visualização de grau e distância de pré-buscas.

Ainda, de acordo com Lee et al. (2012), é possível classificar pré-buscas em algumas categorias, essas nomenclaturas são utilizadas nesse trabalho, e ajudam a avaliar o quão útil um determinado *prefetch* é. A Tabela 2.1 contém as classificações relevantes para essa monografia. A Figura 2.3 é um diagrama para visualização e melhor compreensão da "pontualidade" de uma pré-busca.

É relevante diferenciar os diferentes tipos de demandas por dados quando se está trabalhando com pré-buscas de dados. As demandas de *prefetching* serão chamadas de **demandas especulativas**, enquanto as requisições de dados do programa original serão denominadas **demandas efetivas**. Essa é a nomenclatura utilizada nessa monografia.

Ainda, uma terminologia utilizada na experimentação dessa monografia é *prefetch "just-in-time"*. Essencialmente, uma pré-busca "*just-in-time*" é uma que ainda não foi satisfeita

Classificação	Acurácia	Pontualidade
Pontual	Correto	Melhor *prefetching*
Redundante MSHR	Correto	Requisição já estava no MSHR
Redundante DC	Correto	Dado já estava na *cache* de dados
Atrasado	Correto	Demanda efetiva chega antes do dado estar pronto
Muito Cedo (Inútil)	Correto	Demanda efetiva chega depois do dado ser retirado da *cache*
Inútil	Incorreto	Dado não é utilizado mesmo depois de sair da *cache*

Tabela 2.1: Classificação de utilidade de pré-buscas ordenada por utilidade.

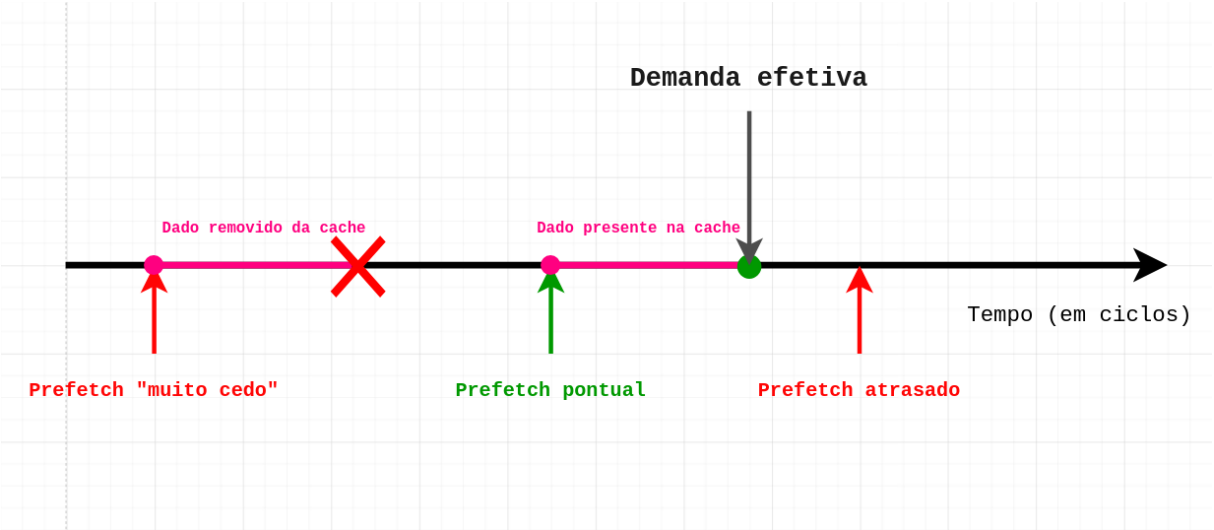


Figura 2.3: Visualização de pontualidade de pré-buscas.

quando uma demanda efetiva que utiliza o dado pré-buscado entra em execução. Entretanto, essa pré-busca termina de escrever a linha da *cache* L1 e satisfaz a demanda efetiva em tempo menor ou igual à própria latência da L1, essencialmente terminando em cima da hora ("*just-in-time*"). A Figura 2.4 apresenta uma visualização para esse tipo de *prefetch*.

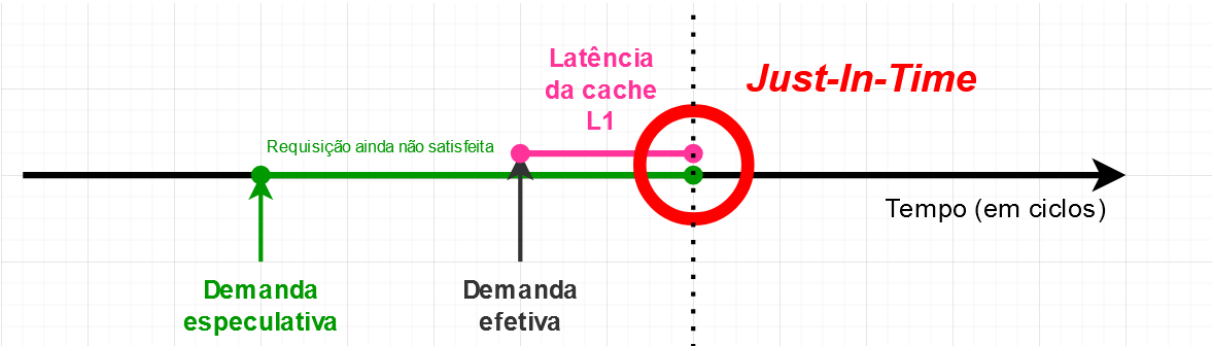


Figura 2.4: Visualização de pré-buscas "just-in-time".

2.1 TÉCNICAS DE *HARDWARE PREFETCHING*

Como esta monografia tem foco em *prefetchers* de *hardware*, ressalta-se que o histórico dos primeiros *prefetchers* pode ser brevemente resumido, de acordo com Nesbit e Smith (2004), da seguinte maneira:

- *Prefetchers* sequenciais: Os primeiros e mais simples *prefetchers*, que após um *cache miss* realizam a pré-busca das linhas de *cache* em sequência da primeira. (Ex.: *nextline prefetchers*)

- *Prefetchers* de tabela: São *prefetchers* que utilizam tabelas para registrar informações históricas para detecção de padrões de acesso, como *strides* entre os últimos *N* acessos, dentre outros. (Ex.: *stride-prefetchers*)

Sabendo disso, ressaltam-se os *prefetchers stride* e *nextline*, que serão utilizados na experimentação desse trabalho. O *nextline prefetcher* simplesmente busca a próxima linha da *cache* quando um endereço é fornecido. Já o *stride prefetcher*, especificamente *PC-based stride prefetcher*, utiliza o registrador da instrução "atual" em execução e os últimos *N* acessos para detectar um *stride* específico, e quando ele é detectado a pré-busca do próximo "pulo" é realizada. Ainda, uma técnica que também será aplicada no presente trabalho para decisões de microarquitetura baseadas em *reinforcement learning* é o algoritmo *multi-armed bandit*. Resumidamente, o objetivo desse algoritmo é "escolher" entre técnicas (nesse caso entre a nextline e a stride *prefetching*) baseada no retorno estimado delas. Ou seja, teoricamente um *prefetcher* que esteja acertando mais será escolhido acima do que esteja acertando menos.

Essa escolha é feita com base em uma tendência recente na literatura de utilizar técnicas de *reinforcement learning* em *hardware* (Yang et al., 2024), especificamente para problemas de decisão na arquitetura. Observou-se que técnicas como *Multi-Armed Bandits* são propostas relativamente eficazes para problemas de sequências de decisões (Gerogiannis e Torrellas, 2024), sendo que *prefetching* se enquadra nessa categoria de problemas.

2.2 OUT-OF-ORDER EXECUTION

Outra técnica fundamental para o aumento da eficiência do processador é a *Out-of-Order Execution*. Em suma, é um paradigma em que o processador executa as instruções em uma ordem definida implicitamente pela disponibilidade dos dados e das unidades de execução, ao invés da ordem original do programa. Como essa execução não pode alterar o resultado final do programa mas as instruções estão sendo executadas fora de ordem, observam-se sempre as dependências entre instruções.

O diagrama na Figura 2.5 tem os estágios pelos quais uma instrução passa para ser devidamente executada. Como essa monografia se trata de *prefetching* de dados, é importante elaborar especificamente sobre o que ocorre com instruções do tipo *load*.

Basicamente, a execução inicia-se no *fetch*, onde a instrução é inicialmente buscada da memória *cache* de instruções e inserida em um *buffer*. Depois disso ela passa pelo estágio de decodificação, a partir de onde é possível dizer que a instrução é realmente um *load*.

Após o *decode* a instrução chega no *back-end* do processador, onde a execução fora de ordem em si ocorre. Existem quatro estágios no *back-end* do processador para a execução efetiva de instruções na arquitetura simulada nesse trabalho:

- ***Rename***: Aplica-se uma técnica para evitar **dependências de nome** (ou "falsas dependências") entre instruções. Resumidamente, não são criadas dependência quando duas instruções utilizam o(s) mesmo(s) registrador(es) mas são independentes.

- ***Dispatch***: A instrução é **enviada da fila de instruções para a unidade de execução** apropriada (ex.: unidade aritmética de inteiros). Em arquiteturas superescalares esse passo pode ser feito concorrentemente para diferentes instruções que utilizem diferentes unidades de execução.

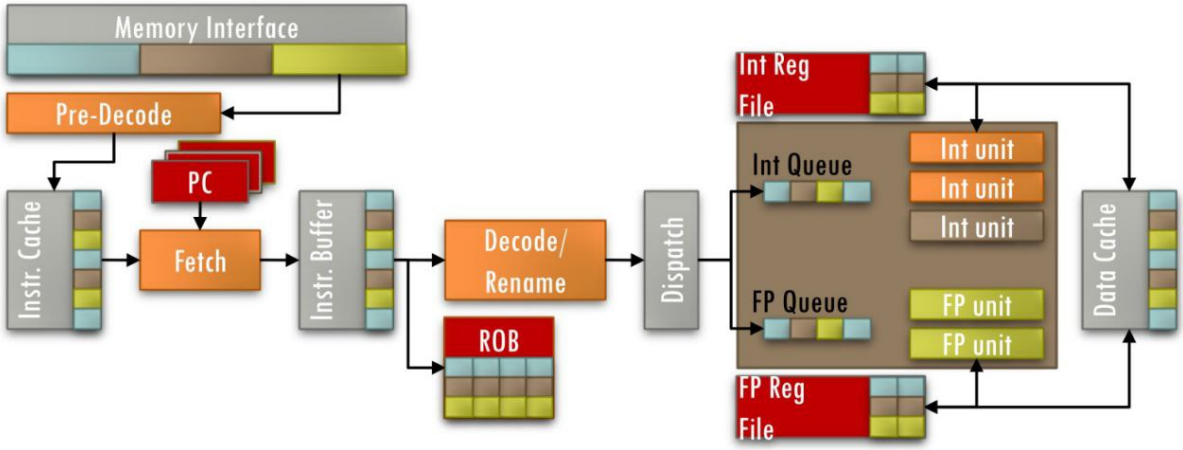


Figura 2.5: Fluxo de uma instrução.

- ***Execute***: A **instrução é devidamente processada pela unidade de execução** da CPU. Após a execução o resultado da computação é armazenado no registrador interno ou na memória.

- ***Commit***: Nesse estágio **os resultados são escritos de volta para os registradores arquiteturais e as instruções são removidas**, **sequencialmente**, para preservar a ordem original do programa executado.

Então, apenas depois do estágio de *renaming* de registradores da instrução, o *load* passa pela AGU (*Address Generation Unit*). Somente então o endereço de memória verdadeiro pode ser obtido, logo antes da instrução ser inserida no ROB (*Re-Order Buffer*) e MOB (*Memoy Order Buffer*).

O principal componente do processador para alcançar a execução fora de ordem é o *Re-Order Buffer* (ROB), um *buffer* circular FIFO (uma fila), que é a unidade de *hardware* capaz de execução fora de ordem e até mesmo especulativa de instruções. O ROB é responsável por realizar os *commits* na ordem do programa. As instruções são inseridas no ROB após o *renaming* dos registradores.

Um componente que auxilia o ROB para instruções que acessam a memória, do tipo *load* ou *store*, é o MOB, que serve como um *buffer* para acessos à memória, ordenando e armazenando as instruções até que elas sejam propriamente enviadas ao subsistema de memória. É o MOB que auxilia a gerenciar dependências entre *loads* e *stores*, dando suporte a execução fora de ordem propriamente dita para instruções de acesso à memória.

Ressaltando que apenas as instruções de *load* podem ser executadas fora de ordem e de forma especulativa, pois elas não apresentam efeitos colaterais com as instruções de escrita. Para instruções de *store*, essas devem ser feitas apenas quando forem a instrução mais antiga presente no ROB, para preservar o estado arquitetural consistente.

Ao longo dos anos a tendência dos processadores modernos é terem um ROB cada vez maior. A microarquitetura *Skylake* (2015) da Intel, referência para a simulação desse trabalho, tinha cerca de 224 entradas no ROB, já para a *Golden Cove* (2021) estima-se 512 entradas, como é possível observar na Figura 2.6. Outro exemplo são os processadores da *Apple*, que estimativas do tamanho do ROB para a CPU A14, lançada em 2020, está na ordem das 630 entradas.

Dessa maneira, é esperado que o tempo entre o *fetch* e o *execute* de uma instrução aumente significativamente, criando uma oportunidade de *prefetching* especulativo no *frontend* do *pipeline* do processador. Um fato observado utilizando o simulador detalhado no Capítulo 5 é

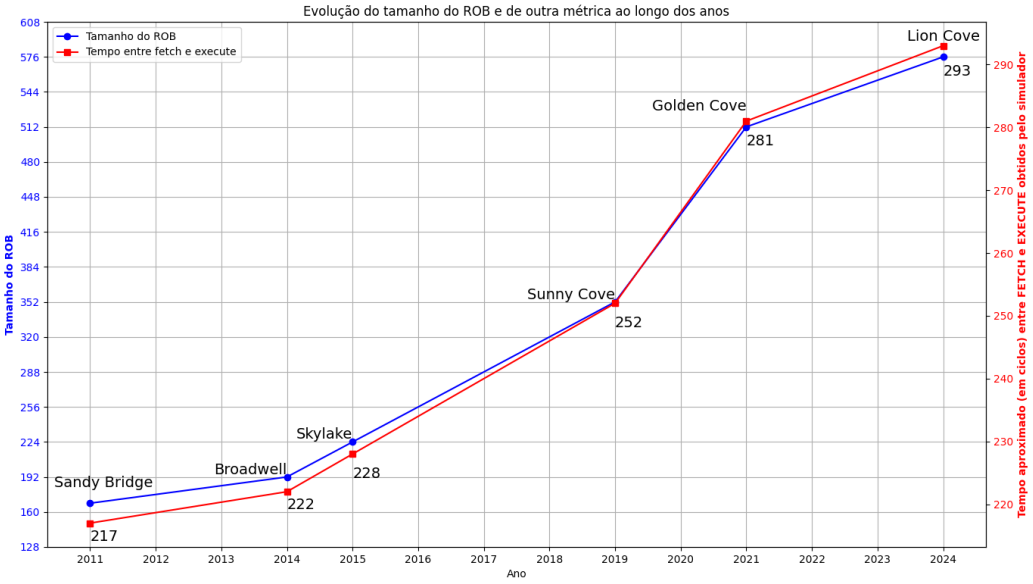


Figura 2.6: Histórico do tamanho do Re-Order Buffer nas arquiteturas da Intel e seu impacto no tempo entre *decode* e *execute* de instruções *load*.

que quando o tamanho do ROB foi aumentado de 128 para 512 entradas a média de tempo em entre o *fetch* e o *execute* de uma instrução aumentou de 210 para 281 ciclos, como observado na Figura 2.6. Sendo assim, a janela de oportunidade para *front-end prefetching* tende a aumentar conforme o tamanho do ROB aumenta e o *pipeline* se torna mais profundo.

2.3 DEFININDO MÉTRICAS RELEVANTES

Para definir a eficiência do *prefetcher* durante a execução do simulador, alguns dados serão analisados. Em especial, visa-se maximizar o IPC e minimizar a latência média da memória, assim como verificar a quantidade de pré-buscas e classificá-las com base em sua utilidade, para definir o aproveitamento do *hadware*. Sendo assim, as seguintes métricas são cruciais para avaliar o desempenho e estimar a eficácia do(s) *prefetcher*(s) desenvolvido(s) no decorrer dos experimentos detalhados no Capítulo 5:

- **Latência média**: Refere-se ao tempo médio que leva para acessar os dados pré-buscados em comparação com o tempo de acesso direto à memória. Uma latência média mais baixa indica que o *prefetcher* está conseguindo entregar os dados de forma eficiente, contribuindo para a redução do tempo total de execução.

- **Porcentagem de pré-buscas úteis**: Essa métrica calcula a fração de dados pré-buscados que são realmente utilizados pelo processador. Uma porcentagem alta sugere que o prefetcher está fazendo um bom trabalho em antecipar as necessidades de dados, enquanto uma porcentagem baixa pode indicar que o algoritmo está prevendo incorretamente os acessos à memória.

- **Atraso médio**: Mede o tempo médio (em ciclos do processador) entre o momento em que um dado é solicitado e o momento em que ele está disponível para uso. Este atraso

deve ser minimizado para que o *prefetcher* contribua efetivamente para a redução da latência aparente.

- **IPC (Instruções Por Ciclo)**: Esta métrica avalia o número de instruções que o processador consegue executar por ciclo de *clock*. Um aumento no IPC após a implementação do *prefetcher* indica que ele está contribuindo para a melhoria do desempenho do sistema, consequência de menos ciclos serem desperdiçados esperando algum dado da *cache*.

- ***Cache miss rate***: Taxa de requisições à *cache* de dados que não foram encontradas e tiveram que ser buscadas no próximo nível da hierarquia de memória.

- **Tempo de *stall* na cabeça do ROB (ciclos)**: Indica quanto tempo, em ciclos de *clock*, *loads* "críticos" (na cabeça do ROB) permanecem travando o ROB, geralmente devido à latência de um *miss*.

Portanto, com base no conhecimento de que técnicas de *prefetching* são eficazes para reduzir a latência de memória, e levando em consideração a profundidade do *pipeline* e o crescente tamanho do ROB, este trabalho propõe o estudo da possibilidade de posicionar um *prefetcher* de *hardware* diretamente no *front-end* do *pipeline*, antes mesmo da decodificação do endereço de memória pela AGU, no estágio de execução da instrução.

Além disso, o estudo abordará a estimativa de ganho de performance desse *hardware* e os desafios envolvidos em sua implementação, avaliando se tal abordagem pode melhorar ainda mais a eficiência no gerenciamento de *loads* em processadores de alto desempenho.

3 CORRELATOS E ESTADO DA ARTE

Neste capítulo serão discutidos os principais trabalhos relacionados à proposta da nova técnica de *prefetching*. Principalmente, serão ressaltadas propostas similares à da presente monografia e outros trabalhos relevantes para a proposta de exploração do problema.

As bases *IEEEXplore* e *Google Scholar* foram as principais utilizadas para a pesquisa, além das buscas feitas em motores de pesquisa regulares, como *DuckDuckGo*. Os principais termos que fizeram parte do processo de pesquisa foram "*memory latency*", "*prefetching*", "*pipeline prefetching*".

3.1 SUMÁRIO DOS TRABALHOS RELEVANTES

Durante a revisão bibliográfica foram encontrados diversos trabalhos relacionados a alguns subtemas desta proposta, como as atuais técnicas de *prefetching* mais eficientes ou outros métodos para esconder a latência da memória que serviram de inspiração ou suporte para o que foi desenvolvido nesse trabalho. Assim, podem-se classificar os trabalhos correlatos em duas categorias:

- Os que analisam e/ou implementam *prefetchers* relevantes para a atual monografia.
- Os que estudam a latência de memória e propõem ou analisam técnicas similares ou relevantes ao que foi proposto nesse trabalho.

Um dos primeiros estudos a propor a técnica de *prefetching* diretamente no *pipeline* do processador foi por Eickemeyer e Vassiliadis (1993), que propõe um mecanismo para prever os próximos endereços a serem acessados. Não muito depois, um estudo foi feito por González e González (1997) que avaliou a possibilidade de fazer buscas especulativas na memória. Esses dois estudos formam a base para o trabalho atual. Parte da pesquisa teve foco em procurar artigos mais recentes que citassem esses dois, a fim de buscar essas técnicas aplicadas no contexto moderno, idealmente no estado da arte.

Ainda, Efnusheva et al. (2017) faz um levantamento sobre técnicas modernas para combate da latência de memória, o que foi utilizado de referência para definir claramente o método do trabalho. Em particular, a conclusão desse *survey* afirma que o maior problema em sistemas centrados em memória é a limitada quantidade de memória acoplada ou integrada ao processador, o que serve também de motivação para o presente trabalho.

Um estudo feito por Alves et al. (2021) avalia e sugere meios de implementação de *data prefetching* especulativo no estágio de *fetch* do *pipeline* do processador baseado no PC (*Program Counter*) da instrução, similar à proposta do presente trabalho. Além dos resultados do estudo, algumas considerações sobre *prefetching* no *pipeline* servem como base para a formulação da proposta.

Entretanto, esse trabalho ainda se difere da presente monografia por não estudar o impacto de diferentes tamanhos de ROB no *prefetcher* proposto. Além disso, esse trabalho tem como foco criar um *hardware* específico para a predição dos *loads*, enquanto a atual monografia foca no estudo do potencial desse *hardware* (principalmente em condições ideais, para estimar um ganho máximo).

Uma pesquisa que foi relevante para a criação da proposta principal desse trabalho foi feita por Gerogiannis e Torrellas (2024), que mostra um jeito de criar um *prefetcher* utilizando

online reinforcement learning de maneira eficiente quanto ao custo do *hardware*. Essa técnica aplica o problema *multi armed bandit* para decisões de microarquitetura, e exemplifica um modo de aplicar essa modelagem em *prefetchers*.

Outro estudo do estado da arte feito por Yang et al. (2024) sugere que aplicar técnicas de *Reinforcement Learning* (RL) para controle de sequências de decisões em microarquiteturas será o novo padrão para sistemas de alta performance, por permitir adaptação a padrões não triviais de acesso à memória.

Por fim, a Tabela 3.1 compara todos os trabalhos relevantes utilizados para o presente trabalho e compara seus principais pontos com a proposta dessa monografia.

	1	2	3	4	5	6	*Front-end Prefetching* (Nossa proposta)
Latência de memória	•	•	•	•		•	•
Data Prefetching	•	•	•		•	•	•
Reinforcement Learning					•	•	•
Multi Armed Bandit					•		•
Early-pipeline prefetch				•			•
Implementação prática	•	•		•	•	•	•

Tabela 3.1: Classificação dos trabalhos correlatos. 1: Eickemeyer e Vassiliadis (1993), 2: González e González (1997), 3: Efnusheva et al. (2017), 4: Alves et al. (2021), 5: Gerogiannis e Torrellas (2024), 6: Yang et al. (2024).

4 PROPOSTA DE EXPLORAÇÃO E ANÁLISE DO PROBLEMA

Neste capítulo serão abordados detalhes sobre os *prefetchers* implementados e os recursos utilizados para a experimentação e validação da proposta da nova técnica de *prefetching*. Também serão explicadas algumas decisões de implementação para os experimentos e obtenção das métricas.

4.1 PIPELINE FRONT-END PREFETCHER

Primeiramente, é importante definir o *front-end prefetcher*. Essencialmente, ele será um *hardware* implementado no estágio de *decode* do processador, que após a decodificação de uma instrução *load* irá utilizar essa informação para realizar pré-buscas em algum momento. Essas pré-buscas serão feitas com base em algumas regras, que para o caso dessa monografia serão as regras de *prefetchers* existentes e mais um *prefetcher* híbrido proposto utilizando uma técnica nova na literatura.

4.1.1 Prefetchers existentes

A fim de entender como técnicas existentes se comportam quando implementadas no *decode* do processador, assim como se é viável implementá-las, foram escolhidos alguns *prefetchers* simples para testes e comparações na experimentação do presente trabalho. Em particular, os experimentos abordam a implementação dos seguintes *prefetchers*:

- NextLine: Implementado pela simplicidade.
- Next2Lines: Implementado pela simplicidade, mas também pelo seu uso prático conhecido em processadores Intel Xeon.
- Stride: Por serem muito utilizados em processadores comerciais (Bakhshalipour et al., 2020). Em particular, foi implementado um *Instruction Based Stride Prefetcher* (IBSP) similar ao descrito por Vanderwiel e Lilja (2000).

Todos esses *prefetchers* emitem requisições diretamente para a memória *cache*, ou seja, as demandas especulativas não passam pelo MOB. Além disso, todos eles foram implementados para fazer pré-buscas na *cache* L1, uma escolha empírica baseada na observação rápida das diferentes implementações, onde buscamos escolher o *prefetcher* que mais aumentasse o IPC da execução também reduzindo a latência. Um estudo detalhado para otimização da escolha do nível de *cache* está fora do escopo do atual trabalho e é sugerido como trabalho futuro.

4.2 PROVA DE CONCEITO

A fim de implementar ao menos um *prefetcher* realístico especificamente para o contexto novo de pré-busca de dados ainda no *decode* do *pipeline*, foi escolhida uma modelagem utilizando um dos algoritmos mais simples possíveis de *reinforcement learning*: o *Multi Armed Bandit*, a fim de uma implementação de *hardware* de baixo custo.

Essa escolha foi feita com base em um trabalho por Gerogiannis e Torrellas (2024), que explica que para um bom algoritmo de *machine learning* funcionar em *hardware* é atrativo que

não seja necessário treinar o modelo com um *dataset offline*. Isso ajuda, por exemplo, a evitar um erro de um *dataset* não incluir casos não previstos de *workload*, o que faria com que o sistema não se adaptasse propriamente à situação, além de outros benefícios como não necessitar que as ações sejam linearmente correlacionadas aos resultados.

Esse mesmo trabalho explora especificamente como implementar *reinforcement learning* para fazer decisões em microarquiteturas com baixo custo de implementação e que seja reutilizável. Esse estudo foi a inspiração para a modelagem do *Multi Armed Bandit Prefetcher*, implementado no presente estudo.

4.2.1 Multi-Armed-Bandit

O problema do *multi-armed bandit* (MAB) é um modelo clássico em aprendizado de máquina e teoria de decisão, onde um agente deve escolher entre várias opções (denominadas "braços"), cada uma com uma recompensa incerta. O objetivo é maximizar a recompensa total ao longo do tempo, balanceando a exploração de opções desconhecidas e o aproveitamento de opções que já provaram ser vantajosas. Esse problema é comparado a um cenário de cassino com várias máquinas caça-níqueis ("*bandits*"), onde o agente deve decidir em qual máquina jogar a cada rodada, com o desafio de descobrir qual delas oferece o maior retorno, de onde se origina seu nome (e a razão pelas opções serem chamadas de "braços").

Existem várias estratégias para resolver o problema, como a exploração-aproveitamento (*explore-exploit*), que envolve "explorar" novas opções (para obter mais informações) e "aproveitar" as opções já testadas que oferecem boas recompensas. Métodos como o ϵ-guloso, onde a maior parte das escolhas é feita com base no conhecimento atual, mas uma pequena fração das escolhas é feita aleatoriamente para explorar, são comuns. O problema do *multi-armed bandit* é amplamente utilizado em áreas como recomendação de conteúdo, otimização de anúncios online e alocação de recursos, pois oferece uma abordagem eficiente para problemas de decisões sequenciais com incerteza.

4.2.2 Multi-Armed Prefetcher

Como o problema de *prefetching* pode ser visto como um problema de decisão sequencial, há uma oportunidade para explorar o uso da técnica de *reinforcement learning* para MABs. Em especial, nessa monografia, é utilizado o algoritmo de *multi-armed bandit* ϵ-guloso, onde os braços (opções) são diferentes estratégias de *prefetching*, que serão escolhidas com base nas estimativas de retorno, isso é, se as pré-buscas de cada estratégia estão sendo utilizadas ou não.

A fim de sugerir um meio de implementar um *prefetcher* no *decode* do *pipeline*, esse trabalho propõe utilizar um modelo simples de *reinforcement learning* para escolha entre duas estratégias de *prefetching*: *Multi-Armed Prefetching* ϵ-guloso.

Essa escolha é feita com base na ideia de que para um *prefetcher* inserido no *decode* do processador pode ser difícil determinar padrões de acesso mais complexos, então escolhem-se algumas estratégias e um algoritmo para optar entre elas baseado em estimativas de sucesso.

As duas estratégias escolhidas foram de *prefetchers Next2Lines* e *Stride*. O Algoritmo 1 descreve o fluxo de execução desse *multi-armed bandit*. Resumidamente, o *prefetcher* precisa saber o total de vezes que cada estratégia foi escolhida, e um *epsilon* pré-definido. Nesse trabalho $\epsilon = 0.1$, pois na literatura observam-se valores entre 0.01 e 0.1, e optou-se por mais oportunidades de "exploração".

Com essas informações, o algoritmo escolhe com 10% (ϵ) de chance aleatoriamente entre as estratégias, como "exploração". As demais 90% das escolhas serão para a estratégia que tiver a melhor recompensa registrada, como "aproveitamento". O cálculo da recompensa

escolhido para o trabalho é simples: é a quantidade total de demandas especulativas que foram encontradas por demandas efetivas dividida pelo total de demandas especulativas da respectiva estratégia (a porcentagem de pré-buscas que foram efetivamente úteis).

Algoritmo 1 Multi-Armed-Prefetcher "ϵ-Guloso"

Inicialização:
$N_1 \leftarrow 0$ // número de "chamadas" do braço 1
$N_2 \leftarrow 0$ // número de "chamadas" do braço 2
$\hat{\mu}_1 \leftarrow 0$ // recompensa estimada para o braço 1
$\hat{\mu}_2 \leftarrow 0$ // recompensa estimada para o braço 2
$\epsilon \leftarrow 0.1$ // probabilidade de exploração
$T \leftarrow$ total de passos
for $t = 1$ to T **do**
  if random() $< \epsilon$ **then** // exploração
    $A_t \leftarrow \text{Random}(0, 1)$ // escolher braço aleatoriamente
  else // aproveitamento
    if $\hat{\mu}_1 > \hat{\mu}_2$ **then**
      $A_t \leftarrow 0$ // (braço/estratégia 1: Next2Lines Prefetcher)
    else
      $A_t \leftarrow 1$ // (braço/estratégia 2: Stride Prefetcher)
    end if
  end if
  if $A_t = 0$ **then** // Next2Lines escolhido
    $X_1(t) \leftarrow$ recompensa para o Prefetcher Next2Lines
    $N_1 \leftarrow N_1 + 1$
    $\hat{\mu}_1 \leftarrow \frac{N_1 \cdot \hat{\mu}_1 + X_1(t)}{N_1}$ // atualiza estimativa de recompensa
  else // Stride escolhido
    $X_2(t) \leftarrow$ recompensa para o Prefetcher Stride
    $N_2 \leftarrow N_2 + 1$
    $\hat{\mu}_2 \leftarrow \frac{N_2 \cdot \hat{\mu}_2 + X_2(t)}{N_2}$ // atualiza estimativa de recompensa
  end if
end for

Nota-se que por definição o algoritmo é "síncrono", isso é, ele obtém o resultado para a escolha do braço imediatamente após a decisão. Entretanto acessos à memória não funcionam dessa maneira, pois a requisição pode demorar, ela pode não ser utilizada, dentre outros fatores, sendo por definição assíncronas.

Um modo de parcialmente resolver esse problema, e o método aplicado nesse trabalho, é ter uma janela de tempo máxima para cada requisição de pré-busca. Em suma, o *prefetcher* tem uma tabela a mais, uma estrutura de FIFO (*First-In First-Out*) para armazenar apenas as *tags* das pré-buscas feitas e se elas deram resultado ou não. Depois, quando uma requisição à *cache* for finalizada e a linha de *cache* estiver com a *flag* de "*prefetched*" marcada ela é mandada para o *prefetcher* atualizar a tabela de requisições.

Como essa é uma tabela ordenada, uma fila por "ordem de chegada" da requisição de memória, toda vez que uma nova pré-busca for inserida e a fila estiver cheia o item mais antigo é removido. Caso ele não tenha sido atualizado em nenhum momento, o número de chamadas da estratégia escolhida deve ser decrementado, para não afetar negativamente os resultados dos outros a longo prazo, essencialmente aplicando o algoritmo a apenas uma janela de requisições.

Nota-se que nem toda memória *cache* implementa a *flag* de *prefetched*. Apenas sistemas modernos mais sofisticados, principalmente os que tem esquemas de *prefetching* mais complexos. A presença dessa *flag* é assumida nessa implementação.

5 EXPERIMENTAÇÃO E VALIDAÇÃO

Neste capítulo serão explicados os experimentos realizados com os *prefetchers* implementados e serão feitas análises de resultados para validação da proposta. Também serão explicadas algumas decisões de implementação para os experimentos e obtenção das métricas.

5.1 MÉTODO

Para a experimentação e validação da proposta dessa monografia foi utilizado o *Ordinary Computer Simulator* (OrCS) desenvolvido e utilizado pelo laboratório *High Performance and Efficient Systems* (HiPES) da Universidade Federal do Paraná (UFPR). Esse simulador será utilizado com traços de execução contendo 2 bilhões de instruções mais significativas do pacote de *benchmarks* da *Standard Performance Evaluation Corporation* (SPEC CPU) de 2017.

A SPEC projetou esses pacotes para fornecer uma medida comparativa do desempenho com uso intensivo de computação na mais ampla gama prática de hardware usando cargas de trabalho desenvolvidas a partir de aplicações reais de alto desempenho.

Os traços foram obtidos utilizando o método de *simpoints*. Essencialmente, esse método visa capturar as características essenciais da execução, o que é feito identificando os pontos mais representativos (*simpoints*) (Calder et al., 2006). Esses pontos são selecionados com base na ideia de que o comportamento do programa em certas regiões é similar, permitindo que simulações foquem no comportamento geral do programa, ao invés de sua execução como um todo.

A microarquitetura simulada foi inspirada no processador *Skylake Server* da Intel, cuja arquitetura está definida no diagrama de blocos da Figura 5.1. Quanto ao sistema simulado, suas características de configuração estão presentes na Tabela 5.1.

5.2 VERIFICAÇÃO DE OPORTUNIDADES DE PRÉ-BUSCAS

Antes de iniciar a fase de experimentação deste trabalho, é fundamental avaliar hipoteticamente a eficácia de um *prefetcher* implementado no estágio de *decode* do *pipeline*. Essa verificação inicial não apenas estabelece um fundamento para a pesquisa, mas também orienta o desenvolvimento da proposta de acordo com os possíveis ganhos máximos teóricos da implementação.

Para isso, será simulado um processador capaz de decodificar perfeitamente todos os endereços de memória no estágio de *decode* para fazer a pré-busca. Esse *prefetcher* perfeito na prática fará com que o número de acessos à memória seja dobrado: realizando uma requisição de *prefetch* e a requisição original ao mesmo endereço.

Ressaltamos que como o *prefetcher* é implementado para realizar buscas na *cache* L1, e como definido Tabela 5.1, a janela de tempo para requisições "*just-in-time*" na arquitetura simulada é de 5 ciclos, a latência da *cache* L1.

A Figura 5.2 apresenta o histograma de pontualidade das demandas especulativas realizadas pelo *prefetcher* ideal (com especulação de endereços 100% correta). É importante ressaltar que esse histograma apenas leva em consideração as pré-buscas úteis, e como a acurácia das pré-buscas é 100% por definição do experimento, o único tipo de pré-busca que pode ser inútil é a que for retirada da memória *cache* antes de ser requisitada.

Ressalta-se também a distinção entre a terminologia "atraso médio" e "latência média". Em suma, atraso médio se refere à média de tempo (em ciclos) observada nas demandas

Tabela 5.1: Configuração do sistema base.

Out-of-Order Execution Cores
1 core
6-wide issue; Buffers: 40-entry fetch,
128-entry decode; 224-entry ROB; 1 branch per fetch;
MOB entries: 72-read, 56-write; 2-load, 1-store units (1-1 cycle);
4-alu, 1-mul. and 1-div. int. units (1-3-32 cycle);
2-alu, 2-mul. and 1-div. fp. units (3-5-10 cycle);
Branch predictor: Piecewise, 4096 entry BTB
L1 Instruction Cache
32 KB, 8-way, 4-cycle; 64 B line;
LRU policy; 10-entry MSHR
L1 Data Cache
32 KB, 8-way, 5-cycle; 64 B line;
LRU policy; 20-entry MSHR
L2 Cache
1 MB, 16-way, 12-cycle; 64 B line;
LRU policy; 40-entry MSHR
LLC Cache
16 MB, 16-way, 52-cycle; 64 B line
LRU policy; 320-entry MSHR
3D Stacked Memory
32 vaults, 8 DRAM banks/vault, 256 B row buffer;
Inst. lat. 1 CPU cycle;
8 B burst width at 2.5:1 core-to-bus freq. ratio; Open-row policy;
DRAM: CAS, RP, RCD, RAS, CWD latency (9-9-9-24-7 cycles)

especulativas atrasadas, isso é, que são satisfeitas depois da demanda efetiva entrar em execução. Já a "latência média" é referente à média de tempo que as demandas efetivas levaram para serem satisfeitas.

Percebe-se comparando o histograma da Figura 5.2 com o da Figura 5.3 que o total de demandas efetivas que são realizadas é em torno de 10 vezes maior do que o total de demandas especulativas que foram úteis. Isso ocorre pois aproximadamente 93.7% das pré-buscas foram retiradas da *cache* L1 antes das demandas efetivas correspondentes serem realizadas, pois outras requisições necessitaram da linha de *cache* antes, efetivamente desperdiçando a demanda especulativa.

É importante ressaltar que ao realizar *prefetching* de maneira tão agressiva quanto no teste visualizado na Figura 5.2 (efetivamente dobrando o número total de acessos à memória), o tráfego na memória será aumentado significativamente e impactará negativamente a latência da memória, um efeito já observado em outros estudos (Efnusheva et al., 2017). Portanto, é provável que parte da razão de tantas das pré-buscas serem removidas da *cache* seja devido ao alto tráfego na memória causado pelo próprio *prefetcher*, pois esse experimento essencialmente dobra o número de acessos de leitura à *cache*.

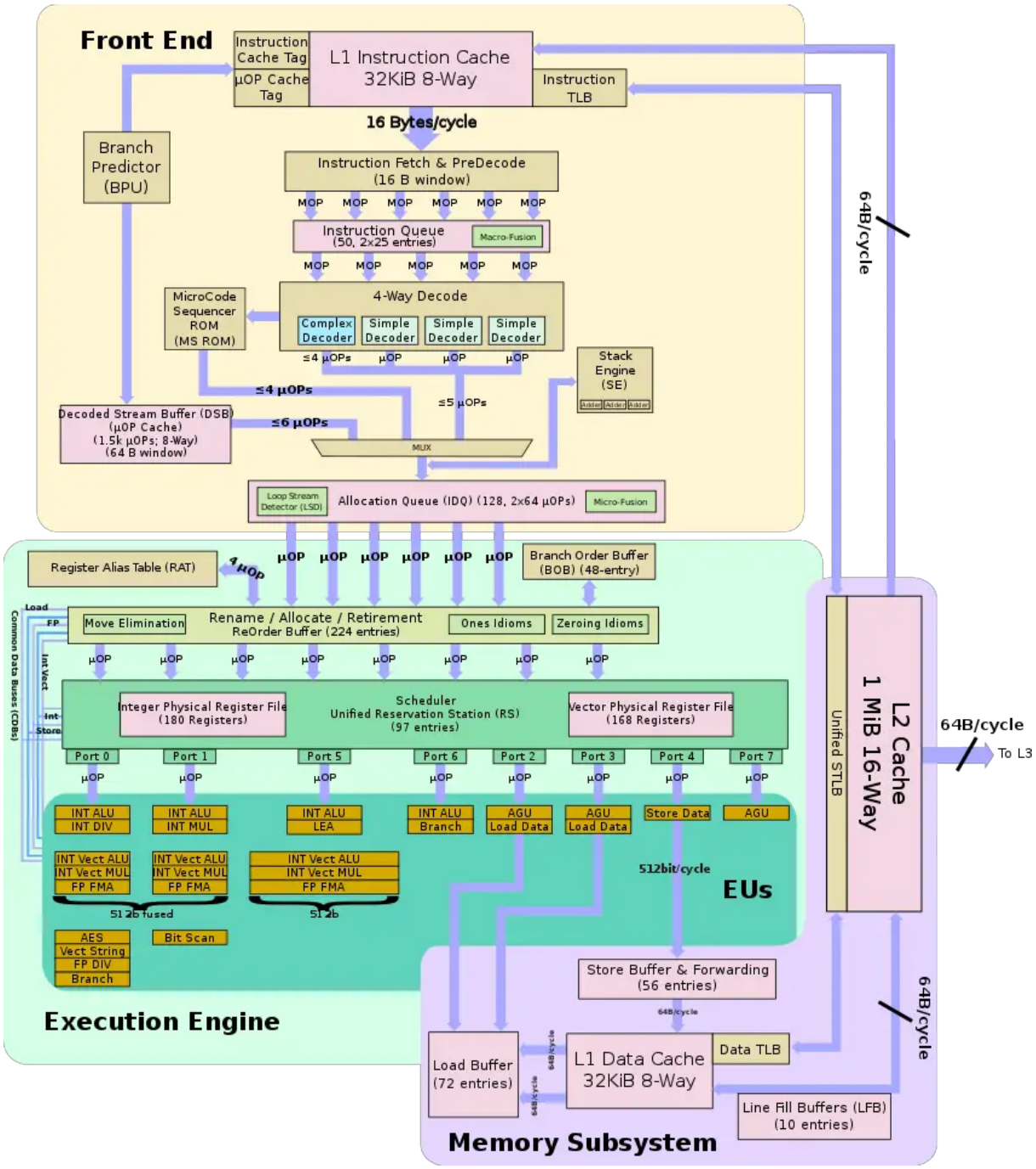


Figura 5.1: Diagrama de blocos do Intel Skylake Server (WikiChip, 2017).

Aprendizado 1

Executar pré-buscas demais no estágio de decode (ainda que com acurácia de endereços perfeita) afeta negativamente a eficiência do processador, pois a vasta maioria delas é removida da *cache* antes de seu uso. Isso ocorre pois demandas especulativas são feitas cedo demais, e acabam utilizando linhas de *cache* que demandas anteriores (que devem ser satisfeitas anteriormente), sejam efetivas ou especulativas, também precisam utilizar para satisfazer as requisições de dados.

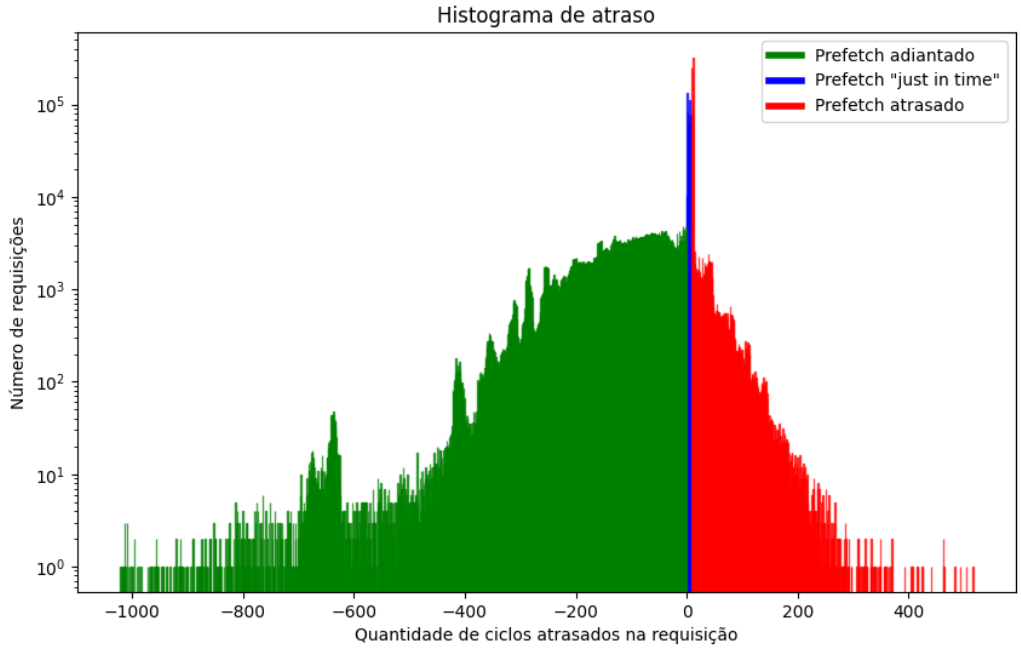


Figura 5.2: Histograma mostrando distribuição de atraso de requisições à memória (*prefecther* perfeito).

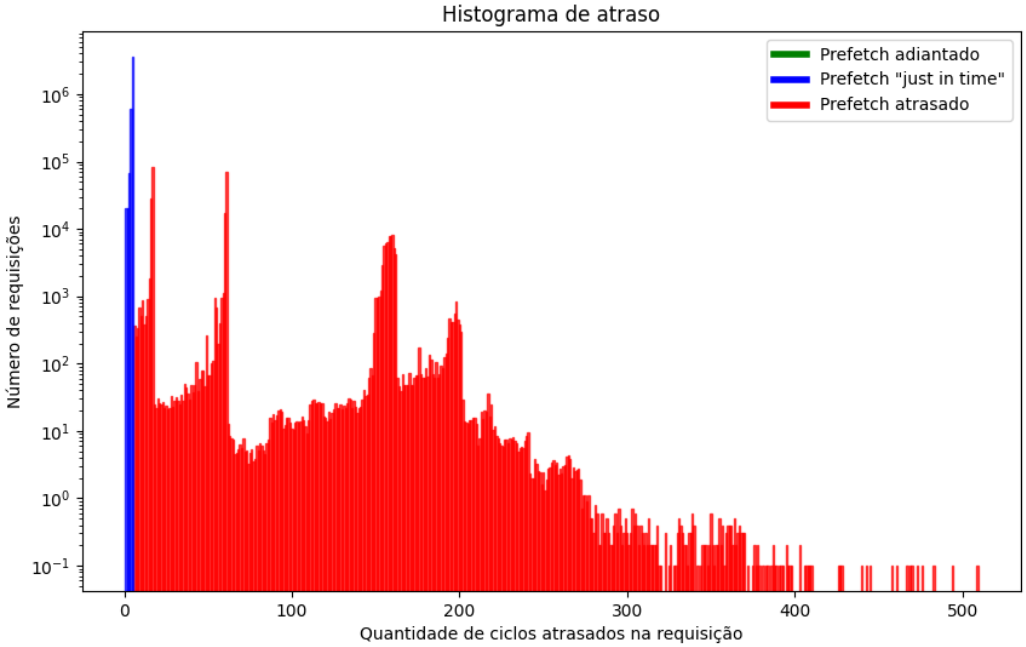


Figura 5.3: Histograma mostrando distribuição de atraso de requisições à memória (sem *prefecther*).

Por fim, como esperado, afirma-se que existe um ganho significativo de performance, pois observou-se com o gráfico da Figura 5.2 que de todas as requisições úteis, aproximadamente 42% dos *prefetches* úteis podem ser utilizados para esconder completamente a latência da memória, enquanto os outros 58% contribuem menos para reduzir a latência aparente da *cache*. Isso também é observável no IPC médio das execuções, que mesmo com o dobro de requisições à

memória, aumentou em 19%, ou seja, partindo de aproximadamente 1.36 para aproximadamente 1.62.

5.2.1 Comparação com prefetching regular

Sabendo que existe uma janela de tempo para adiantar as pré-buscas no *pipeline*, o objetivo do próximo experimento é determinar se essa janela de oportunidade já não é bem aproveitada por *prefetchers* regulares (implementados entre níveis de *cache*).

Para esse experimento foram implementados dois *prefetchers netxline*: um ativado entre todo *cache miss* na L1, e outro feito no estágio de *decode* do processador, que irá fazer a pré-busca para todos os *loads* que forem decodificados. Neste experimento o *nextline prefetcher* terá acesso aos endereços de *load* para as instruções ainda no estágio de *decode*.

Percebe-se que há uma diferença de agressividade entre os *prefetchers*, pois o *front-end nextline prefetcher* realizará pré-buscas mais cedo do que o *cache miss nextline prefetcher*. Estudar o comportamento desses *prefetchers* conforme se altera sua agressividade está fora do escopo dessa monografia e é sugerido como trabalho futuro.

Espera-se com esse experimento que o *prefetcher* posicionado no *pipeline* demonstre melhor aproveitamento das pré-buscas. Para determinar isso, serão observados os histogramas de atraso das pré-buscas, além de algumas métricas, das quais as mais relevantes são o atraso médio das pré-buscas e a latência média da leitura da *cache* de dados.

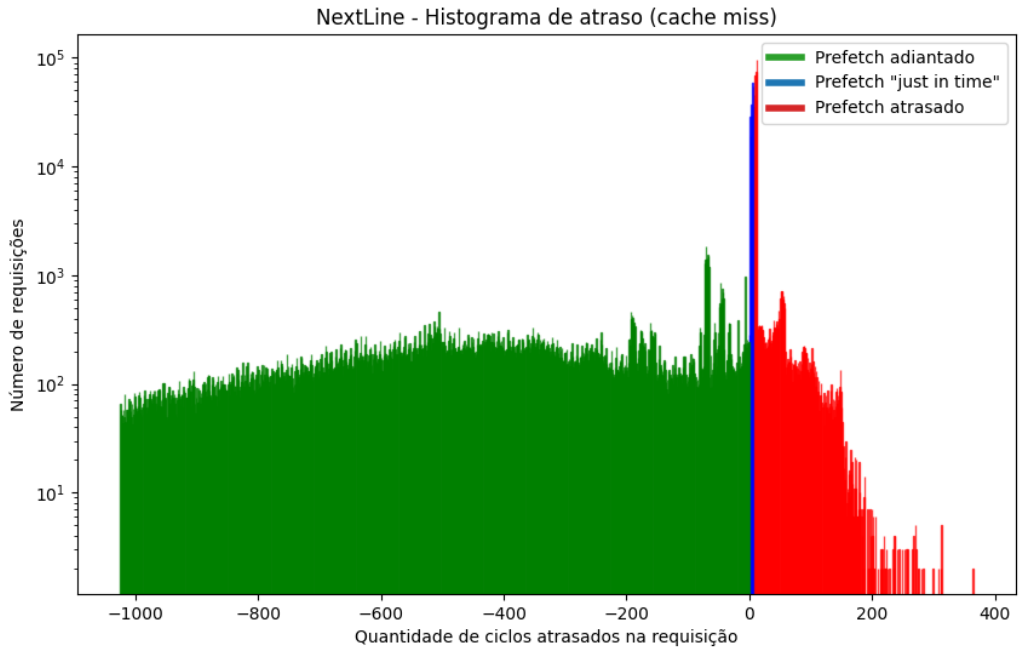


Figura 5.4: Histograma mostrando distribuição de atraso de pré-buscas (*pipeline nextline*).

Observa-se intuitivamente que no histograma da Figura 5.5 (*cache prefetcher*) os atrasos das pré-buscas estão com uma tendência relativamente maior “à direita”, ou seja, a maiores atrasos, se comparado ao histograma da Figura 5.4 (*pipeline prefetcher*).

É mais fácil visualizar essa informação com um gráfico de linhas, apresentado na Figura 5.6, que mostra uma sobreposição dos histogramas. Verifica-se que antes do pontilhado preto a densidade de requisições do *front-end prefetcher* é maior que a do *cache miss prefetcher*,

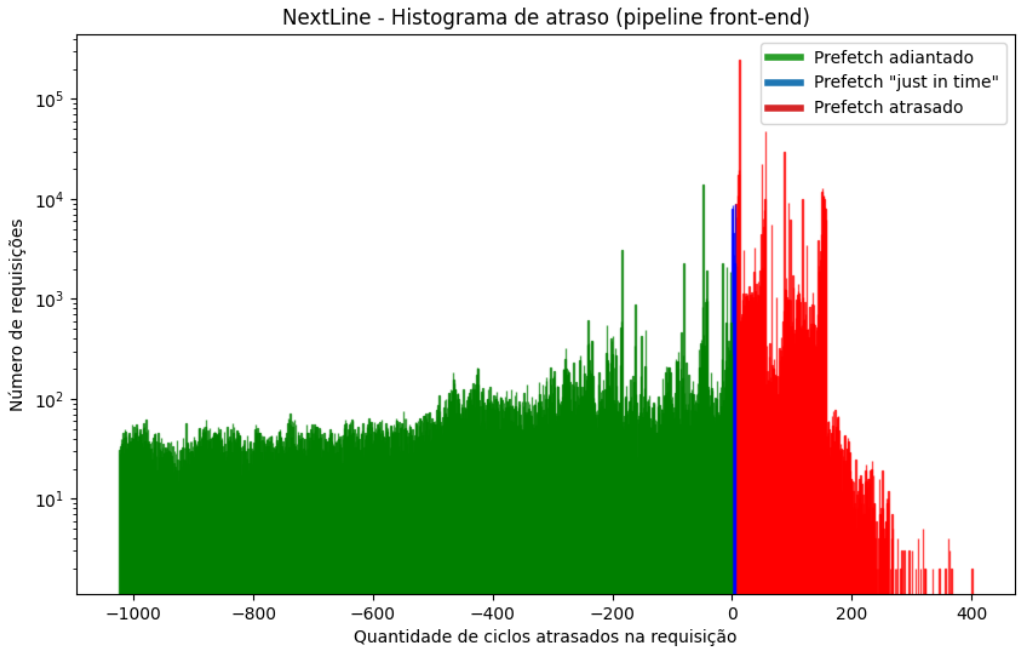


Figura 5.5: Histograma mostrando distribuição de atraso de pré-buscas (*cache nextline*).

enquanto depois da linha (pré-buscas atrasadas) a quantidade é menor. Isso significa que em termos de eficiência das pré-buscas corretas o *prefetcher* de *front-end* é relativamente melhor para esconder a latência de memória.

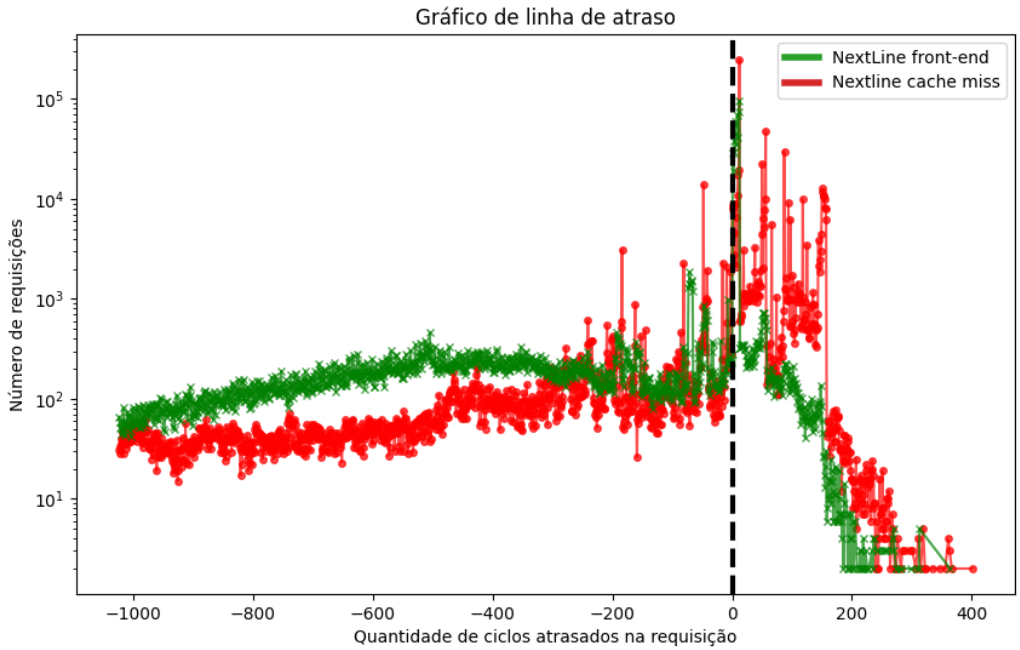


Figura 5.6: Gráfico de linhas comparando atraso de pré-buscas entre *prefetchers cache nextline* e *front-end nextline*.

As afirmações anteriores também podem ser comprovadas na Tabela 5.2, que mostra um atraso médio muito menor para as pré-buscas (aproximadamente 4 vezes menor). Observa-se

também que mesmo realizando *muito* mais requisições à memória o *front-end prefetcher* ainda apresentou uma latência média de leitura menor.

Métrica	***Front-end***	***Cache***
Porcentagem de pré-buscas atrasadas	54.64%	70.45%
Atraso médio (*ciclos de clock*)	10.4718	46.8494
Porcentagem de requisições já presentes na *cache*	34.62%	24.18%
Latência média de leitura (*ciclos de clock*)	8	9
Porcentagem de pré-buscas úteis (acurácia)	1.56%	20.48%

Tabela 5.2: Comparação de métricas entre *prefetchers nextline* (*frontend* vs. *cache miss*).

Percebe-se que a porcentagem de pré-buscas do *prefetcher* de *front-end* é muito menor do que a do implementado na hierarquia de memória. Entretanto, é importante lembrar que o primeiro desses dois é ativado **para toda** requisição à memória, enquanto o outro é apenas ativado quando ocorre um *cache miss*. Para otimizar essa porcentagem a fim de deixar o *hardware* proposto mais eficiente seria necessário estudar como adivinhar endereços no estágio de *decode*, além de otimizar a condição de ativação dele, ambos estudos fora do escopo desse trabalho, mas sugeridos como trabalhos futuros.

Conclui-se com isso que *prefetching* no estágio de *decode* do *pipeline* reduz consideravelmente o atraso médio das pré-buscas, mesmo se for feito de maneira a aumentar consideravelmente o tráfego de memória e por consequência afetando negativamente a latência aparente. Ou seja, mesmo uma implementação "simples" de um *front-end prefetcher* é capaz de melhorar a latência aparente melhor do que *prefetchers* convencionais.

5.3 ANÁLISE DE PERFORMANCE

Com o intuito de avaliar como a performance do sistema simulado é afetada de acordo com a implementação do *frontend prefetcher*, o simulador foi utilizado em todos os traços do SPEC CPU 2017. Esses experimentos serão feitos com uma configuração fixa para a microarquitetura *Skylake Server* da Intel, e na seção seguinte será estudada a possibilidade de reduzir o custo de fabricação do processador implementando esse novo *prefetcher* ao invés de aumentar o tamanho do ROB.

Nessa seção, ao invés de explorar especificamente o quão adiantada ou atrasada está uma pré-busca, como feito na seção anterior, o foco será nas métricas de latência de memória, taxas de *cache hits* e *cache misses*, assim como no IPC. Os gráficos terão informações por traço de execução a fim de possibilitar uma análise mais completa das estatísticas.

5.3.1 Multi-Armed Prefetcher

Como sugestão de implementação de um *prefetcher* utilizando técnicas de *reinforcement learning*, esse trabalho propõe utilizar um *Multi-Armed-Bandit* ϵ-guloso, como definido no Capítulo 4, implementado de maneira quase idêntica à definida por Gerogiannis e Torrellas (2024), mas com apenas duas estratégias (braços), que são *Next2Lines* e *Stride Prefetching*. Esse *prefetcher* será ativado em toda instrução *load* decodificada.

Além da diferença anterior, um detalhe importante é que o *feedback* desse *prefetcher* não é "completo". Para evitar realizar mais requisições à *cache* para validação, que é um problema de *prefetchers* de *pipeline* convencionais, o *hardware* simulado terá um *buffer* de tamanho fixo (tamanho empiricamente escolhido como 64) com as *tags* dos endereços pré-buscados, que será

consultada toda vez que um dado marcado como “pré-buscado” na *cache* for acessado por uma requisição que não seja do *prefetcher*.

Isso implica na possibilidade de uma requisição “boa” não aumentar o *score*, como seria o caso de pré-buscas que são satisfeitas muito antes das requisições originais, influenciando negativamente na escolha de estratégia de *prefetching* (é possível escolher uma estratégia pior no momento caso o *score* dela seja maior que o *score* da estratégia melhor, que pode não ter sido atualizado ainda). Soluções completas para esse problema são deixadas como propostas para trabalhos futuros.

5.3.2 Experimentos

Para testar a eficácia dessa técnica de *reinforcement learning*, ela também será comparada com suas sub-estratégias individualmente, isso é, com um *prefetcher next2lines* e outro *pc-based stride*. A intenção é observar se unir as estratégias existentes em um *Multi-Armed-Bandit* resulta em uma performance melhor do que as estratégias individuais, provando que um MAB é uma alternativa viável para aprender padrões de acesso não triviais em *hardware*.

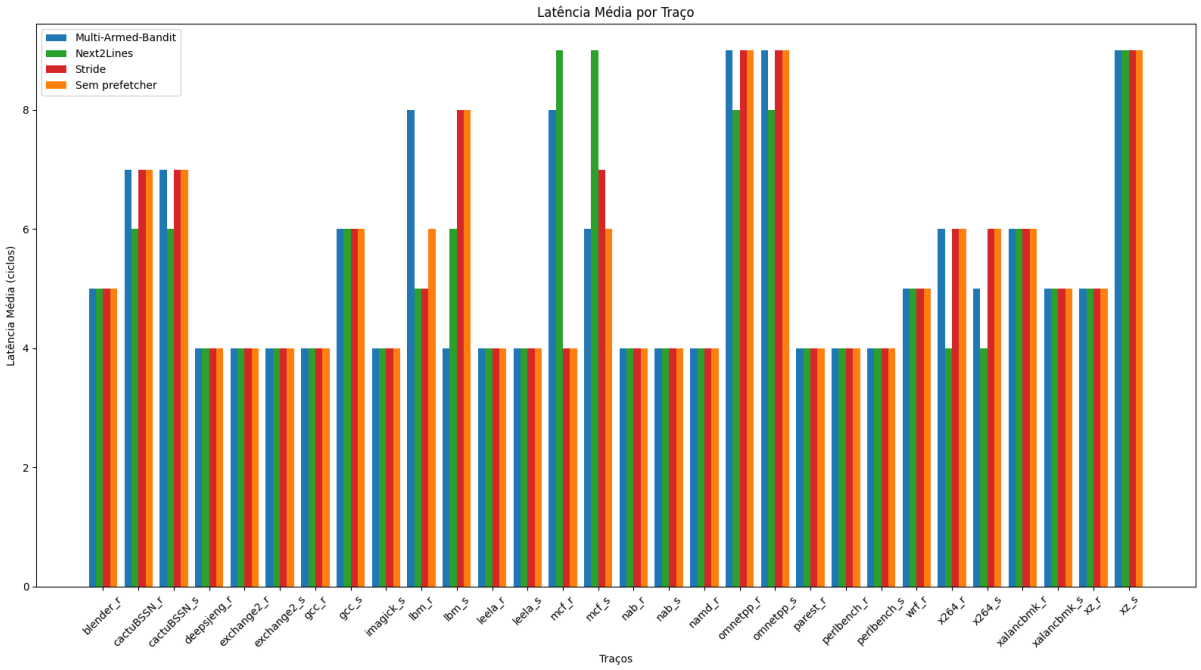


Figura 5.7: Latência média obtida nos traços executados.

É possível observar na Figura 5.7 que a latência aparente do sistema simulado se manteve similar utilizando os *prefetchers*, exceto pelo caso do *next2lines*. Isso pode ser explicado pela maior agressividade desse *prefetcher*, pois enquanto o *stride* e o *multi-armed prefetcher* não necessariamente realizam uma pré-busca em toda chamada, o *next2lines* realiza.

Verifica-se também na Figura 5.8 que a maior parte dos traços manteve o IPC próximo do original mesmo com a implementação do *front-end prefetcher*. Entretanto, alguns traços (por exemplo lbm_s para o *next2lines*) tiveram melhoras significativas, provando que existe uma possibilidade de ganho de performance na implementação de *prefetchers* no *decode* do *pipeline*.

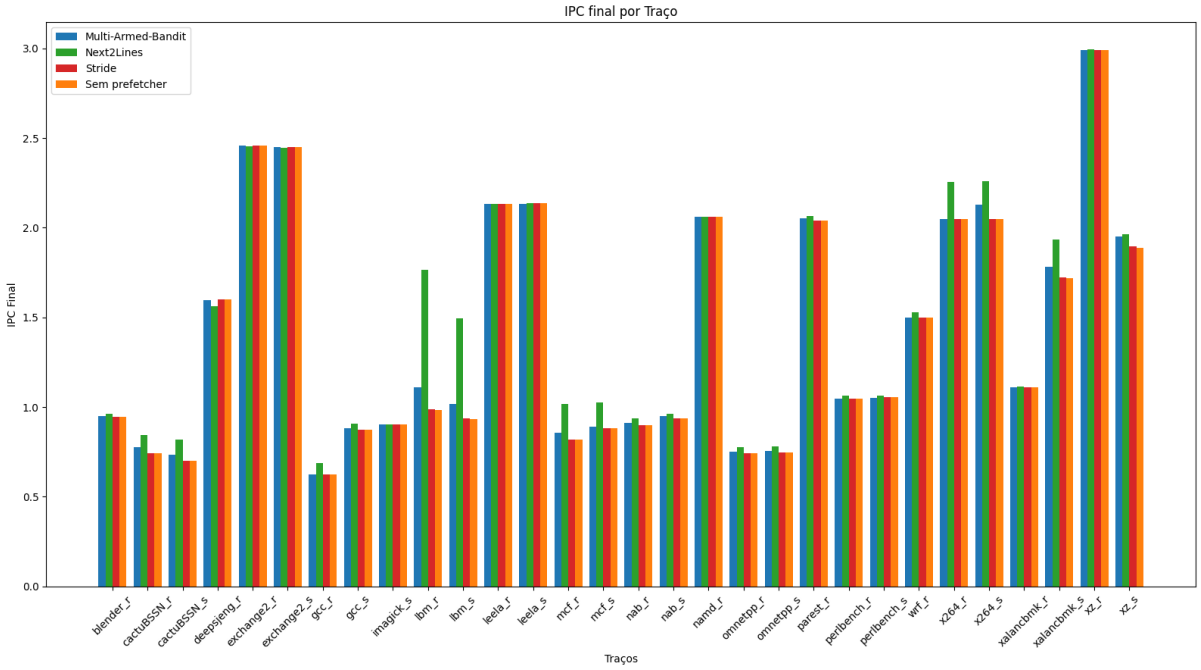


Figura 5.8: IPC observado nos traços executados.

Aprendizado 2

Criar um *Micro-Armed-Prefetcher* ϵ-guloso simples que apenas junta duas técnicas não resultará necessariamente na escolha do melhor *prefetcher* em um dado momento da execução. Dessa maneira, o resultado final da performance de tal *prefetcher* é intermediário entre a melhor e a pior estratégia aplicadas individualmente.

5.3.3 Exploração do espaço de projeto

Dados os experimentos já realizados, o objetivo atual é estudar a viabilidade de modificar as configurações de um sistema base a fim de explorar as diferentes maneiras de alcançar a mesma performance no projeto do processador. Especialmente, visa-se modificar o tamanho do ROB e estudar como a presença de um *front-end prefetcher* afeta o desempenho geral do processador simulado.

Nota-se que o atraso médio no sistema original "não existe", pois ele não tem um *prefetcher* (não faz demandas especulativas), portanto a linha "Atraso Médio" não está presente na Tabela 5.3.

Observa-se comparando o *next2lines* e o sistema original que o IPC não só se manteve, como aumentou para os mesmo tamanhos de ROB, apenas incluindo o *front-end prefetcher*, o que indica um ganho de performance. É interessante observar também que a porcentagem de *cache misses* foi reduzida consideravelmente entre as tabelas.

Aprendizado 3

Utilizando um *front-end prefetcher*, é possível reduzir o tamanho do ROB e manter uma eficiência similar à do sistema original, ou ganhar eficiência com um mesmo tamanho de ROB.

Métrica	Prefetcher	ROB-128	ROB-224*	ROB-256	ROB-384	ROB512
Cache Miss Ratio	Nenhum	9,222%	9,267%	9,271%	9,287%	9,287%
	Next2Lines	8,764%	8,696%	8,684%	8,664%	8,665%
	Stride	9,292%	9,340%	9,343%	9,361%	9,369%
	MAB	9,298%	9,303%	9,265%	9,177%	9,107%
Latência Média	Nenhum	9,466	9,500	9,500	9,500	9,500
	Next2Lines	10,100	10,066	10,066	10,066	10,066
	Stride	9,533	9,566	9,566	9,533	9,533
	MAB	9,600	9,733	9,633	9,566	9,500
IPC	Nenhum	1,387	1,387	1,391	1,399	1,401
	Next2Lines	1,441	1,487	1,491	1,497	1,497
	Stride	1,344	1,387	1,392	1,399	1,401
	MAB	1,367	1,402	1,407	1,409	1,420
Atraso Médio	Nenhum	—	—	—	—	—
	Next2Lines	12,277	10,165	10,052	10,025	10,049
	Stride	11,135	10,165	10,031	10,025	10,025
	MAB	11,414	9,774	11,772	11,154	13,422

Tabela 5.3: Comparação de métricas entre os diferentes tamanhos de ROB e as diferentes implementações de *prefetcher*. (* Tamanho original nos demais experimentos.)

Entretanto, percebe-se que a performance não se manteve para os ROBs de tamanho 128 utilizando o *Stride* ou o *Multi-Armed Bandit prefetcher*. Isso pode ser explicado pela própria proposta do *front-end prefetcher*, que visa aproveitar o tempo que uma instrução gasta entre o *fetch* e o *execute*, principalmente no ROB. Sendo assim, se esse tempo for reduzido drasticamente o potencial do *prefetcher* também é.

Mesmo assim, é importante ressaltar que o *Next2Lines prefetcher*, que é o mais agressivo dos *prefetchers* implementados, continuou com a performance superior mesmo nesse caso. Sendo assim, é possível concluir que para tamanhos pequenos de ROB a agressividade do *prefetcher* é um fator crucial para o ganho de performance.

Isso é observável na Tabela 5.3. Em termos de agressividade, enumeram-se os *prefetchers* do mais agressivo para o menos agressivo respectivamente como: *Next2Lines*, *MAB* e *Stride*. Observando a tabela, é possível verificar que para o tamanho de ROB 128 a performance deles também se manteve nessa ordem.

Aprendizado 4

Reduzindo o tamanho do ROB para valores pequenos demais, 128 ou abaixo, a performance do *front-end prefetcher* dependerá mais de sua agressividade. Observa-se que *prefetchers* mais agressivos obtiveram uma melhor performance com ROBs menores se comparados aos menos agressivos.

5.4 LIMITAÇÕES

Nessa seção iremos comentar sobre algumas das limitações do presente trabalho, principalmente com relação aos experimentos executados e ao simulador utilizado. O objetivo é compreender melhor os pontos mais superficiais da pesquisa para possíveis trabalhos futuros elaborarem a análise deses pontos.

5.4.1 Limitações dos experimentos

Todos os experimentos foram realizados simulando um processador *single-core*, de modo que não foi possível estudar a interferência entre múltiplas *threads* ou a influência de *prefetches* de uma execução de código em outra.

Também há a questão dos traços utilizados nos experimentos. Por mais que todos os experimentos utilizaram traços de execução do SPEC CPU 2017, que é bem representativo para cargas de trabalho de alto desempenho, ainda pode deixar a desejar quanto ao total de aplicações diferentes que podem ser executados em uma arquitetura real, ou outras cargas de trabalho gerais. Ou seja, fatores que só seriam perceptíveis com traços de execução não inclusos no SPEC CPU 2017 não foram levados em consideração.

5.4.2 Limitações do simulador

Por fim, é importante afirmar que o simulador utilizado em si apresenta suas limitações, das quais destaca-se a limitação da modelagem da arquitetura. É possível que os detalhes microarquiteturais do *Skylake* não estejam necessariamente de acordo com a máquina real devido aos diversos pontos que são sigilosos em uma arquitetura, o que pode introduzir uma discrepância no comportamento da máquina simulada.

Além disso, devido a limitações de escalabilidade do simulador, o processamento de grandes volumes de dados foi difícil, podendo ter influenciado na acurácia dos resultados finais dos experimentos. Isso pode ter levado a uma redução na capacidade de simular com precisão determinadas condições de carga, gerando possíveis erros ou aproximações imprecisas nos resultados observados.

6 CONCLUSÕES E TRABALHOS FUTUROS

Essa monografia teve como objetivo estudar a possibilidade e os efeitos de implementar um *prefetcher* especulativo no estágio de *decode* do pipeline, com o intuito de verificar se essa abordagem auxiliaria na redução da latência aparente da memória de CPUs modernas. O objetivo foi alcançado, demonstrando que a introdução desse tipo de *prefetching* pode melhorar a eficiência do uso da memória e reduzir o impacto de latências de acesso.

O processo de desenvolvimento e avaliação necessitou da instrumentação do simulador OrCS para a simulação da execução dos traços da SPEC CPU 2017. Esse simulador foi fundamental no fornecimento de uma plataforma de simulação flexível e detalhada, permitindo testar diversas configurações do ROB e avaliar o impacto do *prefetching* especulativo. Utilizando os *benchmarks* do SPEC CPU 2017, foi possível comparar os resultados em diferentes cenários e validar a abordagem proposta em condições similares às reais de computação de alto desempenho.

Determinou-se que a implementação de um *prefetcher* especulativo no estágio de *decode* oferece uma oportunidade significativa para antecipar ainda mais os dados buscados na memória. Comparado com o modelo tradicional de *prefetching*, o método proposto demonstrou uma média de atraso de pré-buscas muito menor, e demonstra que é possível ganhar desempenho em sistemas reais nesse ponto, abrindo portas para futuras pesquisas nessa área específica. Entretanto, não foram feitas análises para comparar *prefetchers* de agressividades similares, sendo uma limitação dessa monografia.

A análise de desempenho revelou um ganho considerável na redução do tempo de execução dos *benchmarks*, e demonstrou uma redução na latência aparente da memória, mas principalmente um aumento no IPC médio de aproximadamente 7.2% e uma redução de aproximadamente 6.5% na taxa de *miss* da *cache*.

Além disso, a implementação de um *prefetcher* especulativo no estágio de *decode* possibilitou uma redução no número de entradas no *Re-Order Buffer* (ROB). Ao antecipar os dados necessários para futuras instruções, foi possível liberar recursos do ROB, o que teoricamente possibilita reduzir os custos de fabricação do processador.

Há algumas limitações a serem consideradas. Primeiramente, o estudo não propôs um *pipeline frontend prefetcher* propriamnete dito, mas na verdade fez uma análise preliminar do impacto de se adicionar especulação ao estágio de *decode*. Uma possível extensão desse trabalho seria o estudo das condições para ativação do *prefetcher* em si, talvez utilizando técnicas como árvores de decisão para estudar as condições das pré-buscas mais úteis, sem a necessidade de uma implementação direta do mecanismo de *prefetch*. Além disso, a pesquisa foi realizada em um ambiente de simulação *single-core*, o que limita a análise em cenários mais complexos com múltiplos núcleos. Também não foram considerados cenários em que outros tipos de *prefetchers* ou otimizações de memória estivessem em uso simultâneo, o que pode afetar a eficácia da técnica proposta.

Trabalhos futuros podem se concentrar em mitigar essas limitações. A primeira etapa seria avaliar o comportamento do *decode prefetcher* em sistemas *multi-core*, onde as interações entre os núcleos e o acesso à memória são mais complexas. Além disso, seria interessante implementar um mecanismo de *prefetching* mais robusto e integrá-lo com outros *prefetchers* existentes para analisar como ele interage e se complementa com essas outras estratégias, buscando otimizar o uso de memória em sistemas modernos reais.

REFERÊNCIAS

Alves, R., Kaxiras, S. e Black-Schaffer, D. (2021). Early address prediction: Efficient pipeline prefetch and reuse. *ACM Trans. Archit. Code Optim.*, 18(3).

Bakhshalipour, M., Shakerinava, M., Golshan, F., Ansari, A., Lotfi-Karman, P. e Sarbazi-Azad, H. (2020). A survey on recent hardware data prefetching approaches with an emphasis on servers.

Calder, B., Sherwood, T., Hamerly, G. e Perelman, E. (2006). Simpoint: Picking representative samples to guide simulation.

Chang, K. K. (2017). Understanding and improving the latency of dram-based memory systems. *CoRR*, abs/1712.08304.

Efnusheva, D., Cholakoska, A. e Tentov, A. (2017). A survey of different approaches for overcoming the processor - memory bottleneck. *International Journal of Computer Science and Information Technology*, 9:151–163.

Eickemeyer, R. J. e Vassiliadis, S. (1993). A load-instruction unit for pipelined processors. *IBM J. Res. Dev.*, 37(4):547–564.

Gerogiannis, G. e Torrellas, J. (2024). Practical online reinforcement learning for microprocessors with micro-armed bandit. *IEEE Micro*, 44(4):80–87.

González, J. e González, A. (1997). Speculative execution via address prediction and data prefetching. Em *Proceedings of the 11th international conference on Supercomputing*, páginas 196–203.

Lee, J., Kim, H. e Vuduc, R. (2012). When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1).

Nesbit, K. e Smith, J. (2004). @. Em *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, páginas 96–96.

Vanderwiel, S. P. e Lilja, D. J. (2000). Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199.

WikiChip (2017). Skylake (server) - microarchitectures - intel.

Yang, H., Fang, J., Su, X., Cai, Z. e Wang, Y. (2024). Rl-copref: a reinforcement learning-based coordinated prefetching controller for multiple prefetchers. *The Journal of Supercomputing*, 80(9):13001–13026.

Zhang, Q., Song, H., Zhou, K., Wei, J. e Xiao, C. (2024). A prefetching indexing scheme for in-memory database systems. *Future Generation Computer Systems*, 156:179–190.