UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ARTHUR MITTMANN KRAUSE

Análise de um Mecanismo de Mitigação de Poluição de Cache

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Prof. Dr. Philippe O. A. Navaux
Co-orientador: Prof. Dr. Eduardo H. M. da Cruz

Porto Alegre
2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Renato Ventura Henriques
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Nossos feitos moldam o futuro.”

— GAREN STEMMAGUARDA DE DEMACIA

AGRADECIMENTOS

Agradeço ao Eduardo Cruz e ao professor Navaux por me acolherem como bolsista de iniciação científica. Agradeço ao Francis pela valiosíssima ajuda do início ao fim deste trabalho e por tudo que me ensinou.

Dedico este trabalho aos colegas do GPPD, à minha família, à Ana, e a todos os outros que me perguntaram "e o TCC?".

RESUMO

Para contornar o problema da alta latência das memórias, processadores de alto desempenho empregam mecanismos como cache e *prefetching*. Porém, o *prefetcher* pode atrapalhar o desempenho do sistema quando busca dados para a cache que são armazenados em detrimento de dados que são solicitados em seguida, provocando *cache misses*. Na literatura, diversos trabalhos buscam solucionar esse problema através da alteração da política de substituição da cache ou de adaptações no *prefetcher*. Um trabalho recente e influente propõe um mecanismo que alegadamente supera técnicas antigas tanto em simplicidade como em desempenho. Este trabalho busca analisar o problema da poluição de cache através de simulações de um sistema que emprega este mecanismo. Os resultados mostram que o mecanismo tem resultados não satisfatórios para a maioria das aplicações para uma determinada configuração de sistema. Para uma aplicação, o mecanismo consegue melhorar o desempenho em 8%, mas chega a reduzir o desempenho em até 21% em alguns casos e em média 2,1% para um conjunto de *benchmarks*. Os resultados também mostram que variações nas configurações do sistema como a capacidade da cache e agressividade do *prefetcher* podem favorecer o funcionamento do mecanismo.

Palavras-chave: Arquitetura de Computadores. Prefetcher. Poluição de Cache.

ABSTRACT

In order to circumvent the problem of high memory latencies, high-performance processors employ mechanisms such as cache and prefetching. However, the prefetcher may jeopardize the overall system performance when it fetches data into the cache that are stored to the detriment of data that is requested shortly after, provoking cache misses. In the literature, many papers seek to solve this problem through the changing of the cache replacement policy or adaptations to the prefetcher. A recent and influent work proposes a mechanism that allegedly surpasses previous techniques both in simplicity and performance. Our work seeks to analyze the issue of cache pollution through simulations of a system that employs this mechanism. Our results show that the mechanism yields underwhelming performance for the majority of the applications under a certain system configuration. For one application, the mechanism achieves an 8% increase in performance, but in some cases, it reduces performance by 21% and, on average, by 2,1% for a benchmark suite. The results also show that variations in the system configuration such as cache capacity and prefetcher aggressiveness can favor the mechanism's gains.

Keywords: Computer Architecture. Prefetcher. Cache pollution.

LISTA DE ABREVIATURAS E SIGLAS

CPU *Central Processing Unit*

DEWP *Dead Line and Early Write-Back Predictor*

DIMM *Dual Inline Memory Module*

DRAM *Dynamic Random Access Memory*

DSBP *Dead Sub-Block Predictor*

EPF *Evicted Prefetch Filter*

FDP *Feedback Directed Prefetching*

GCC *GNU Compiler Collection*

ICC *Intel C++ Compiler*

ICP *Informed Caching policies for Prefetched blocks*

ICP-AP *ICP-Accuracy Prediction*

ICP-D *ICP-Demote*

IPC Instruções Por Ciclo

ISA *Instruction Set Architecture*

L1 Cache Nível 1

L2 Cache Nível 2

L3 Cache Nível 3

LLC *Last Level Cache*

LRU *Least Recently Used*

MPKI Misses por Mil Instruções

MRU *Most Recently Used*

MSHR *Miss Status Holding Registers*

NRU *Not Recently Used*

PC *Program Counter*

RRIP *Re-Reference Interval Prediction*

LISTA DE FIGURAS

Figura 2.1 Desempenho de aplicações com *prefetchers* de diferentes agressividades....26

Figura 5.1 Reuso de *prefetches* nas aplicações do SPEC 2006......................................42
Figura 5.2 Efeito no IPC das diferentes variações do ICP...43
Figura 5.3 *Misses* por 1000 instruções na LLC sob os diferentes mecanismos..............44
Figura 5.4 Quantidade de *cache hits* para linhas de *demand* e linhas de *prefetch* para cada mecanismo...45
Figura 5.5 Distribuição dos acessos à cache por desfecho...46
Figura 5.6 Aceleração ponderada das diferentes combinações de aplicações com os mecanismos..48

Figura 6.1 Efeito no desempenho das diferentes variações do ICP com 8 MB de cache 51
Figura 6.2 Distribuição dos acessos à cache de acordo com o desfecho em uma cache de 8 MB 52
Figura 6.3 Efeito no desempenho das diferentes variações do ICP com um *prefetcher* de grau 1 53
Figura 6.4 Distribuição dos acessos à cache de acordo com o desfecho com um *prefetcher* de grau 1 54
Figura 6.5 Efeito no desempenho das diferentes variações do ICP com um *prefetcher* de grau 1, distância 8 e 8 MB de cache L3 55

LISTA DE TABELAS

Tabela 2.1 Latência aproximada para acesso de dados em um processador da série Xeon 5500. 22
Tabela 2.2 Instrução PREFETCH e suas dicas na ISA x86 23

Tabela 5.1 Configuração do sistema simulado 41
Tabela 5.2 Classificação das aplicações quanto à sensibilidade ao prefetcher e cache... 46
Tabela 5.3 Combinações de categorias de aplicações 47

SUMÁRIO

1 INTRODUÇÃO **19**
2 FUNDAMENTAÇÃO TEÓRICA **21**
2.1 Cache de CPU **21**
2.2 *Prefetcher* **22**
2.2.1 *Prefetch* por software 23
2.2.2 Prefetch por hardware 24
2.2.3 Desempenho do *prefetcher* 25
2.3 Poluição de Cache **27**
3 MITIGANDO POLUIÇÃO DE CACHE **29**
3.1 Modificação do *prefetcher* **29**
3.2 Modificação nas políticas de substituição da cache **31**
4 IMPLEMENTAÇÃO DO ICP **33**
4.1 Implementação **33**
4.2 ICP-D **34**
4.3 ICP-AP **35**
4.3.1 Tabela de precisão 35
4.3.2 Monitoramento de precisão 36
4.3.3 Ação do mecanismo 36
5 SIMULAÇÃO E RESULTADOS **41**
5.1 Simulação **41**
5.2 Reuso de *prefetch* **42**
5.3 Análise *single-core* **42**
5.3.1 Desempenho 42
5.3.2 *Cache misses* 43
5.3.3 Acessos a linhas por origem 44
5.4 Análise *multicore* **46**
5.5 Comparação com os resultados originais **48**
6 EXPLORAÇÃO DO ESPAÇO DE PROJETO **51**
6.1 Variação do tamanho da cache **51**
6.2 Variação do *prefetcher* **52**
6.2.1 Redução do grau do *prefetcher* 53
6.2.2 Redução da distância de *prefetch* 54
6.3 Variação do *prefetcher* e cache simultaneamente **54**
7 CONCLUSÃO **57**
REFERÊNCIAS **59**

1 INTRODUÇÃO

A velocidade dos processadores vem evoluindo em um ritmo mais veloz do que a das memórias. Primeiro, com o aumento da frequência, e atualmente, com a rápida expansão do número de núcleos, os processadores estão precisando de dados em uma taxa maior do que as memórias podem fornecer, e a tendência é de que esta disparidade apenas aumente. Isso é conhecido como a *memory wall* (WULF; MCKEE, 1995), quando a razão entre o tempo de acesso a dados nas memórias e o tempo de processamento dos dados pela CPU é tão alta que o desempenho do sistema é fortemente determinado pelo tempo que leva para trazer-se os dados da memória principal até o processador.

Para mitigar essa limitação de desempenho, técnicas para esconder a latência da memória como as caches de CPU e o *prefetcher* são empregadas. A cache é uma memória pequena porém rápida que reside entre a CPU e a memória principal e se aproveita da localidade temporal e espacial entre os acessos à memória, armazenando os dados que foram recentemente requeridos pelo processador de forma que, se o processador necessitar novamente dos dados, não precisará esperar novamente pela lenta DRAM. O *prefetcher* tenta prever os endereços que serão requisitados pelo processador em um futuro próximo, e os traz para a cache antes do processador os requisitar, escondendo assim a latência da memória. As implementações mais comuns do *prefetcher* aproveitam-se da localidade espacial dos acessos à memória, e buscam detectar padrões de acesso que o permitem prever os próximos endereços requisitados.

Existe um compromisso entre velocidade e capacidade no projeto de memórias (PRYBYLSKI; HOROWITZ; HENNESSY, 1988). Como a cache da CPU é localizada dentro do chip, onde a área é valiosa, ela precisa ser pequena e rápida, então uma gerência inteligente dos dados que são mantidos nela é essencial para retirar-se mais desempenho. Quando dados que seriam reutilizados pelo processador são substituídos da cache por dados que não serão úteis, uma situação conhecida como poluição de cache é caracterizada (SESHADRI et al., 2012). Uma fonte importante de poluição de cache é o *prefetcher*. Quando dados são trazidos pelo *prefetcher* para a cache, linhas úteis podem ser substituídas pelas linhas do *prefetcher* que normalmente não são tão cruciais para o desempenho, provocando *cache misses* e deteriorando o desempenho do sistema (SRINATH et al., 2007).

A medida em que as aplicações trabalham com um volume de dados cada vez maior e cada vez mais núcleos competem pelo espaço da cache, o problema da poluição

se agrava. Portanto, mitigar a poluição de cache torna-se um problema extremamente relevante em arquitetura de computadores. Diversos trabalhos na literatura buscam analisar e mitigar a poluição de cache. A maioria dos trabalhos baseia suas estratégias em alterações na política de inserção e substituição da memória cache e de alterações nos *prefetcher*s. Em especial, o trabalho de Seshadri et al. (SESHADRI et al., 2015) procura reduzir a poluição através de um mecanismo chamado de ICP, que busca eliminar as linhas de prefetch da cache tão logo elas forem utilizadas, e que monitora a precisão de cada *prefetcher* buscando evitar que *prefetches* imprecisos poluam a cache.

Este trabalho propõe um estudo sobre o estado da arte da mitigação da poluição de cache através do mecanismo *Informed Caching policies for Prefetched blocks* (ICP), cujos autores demonstram conquistar um melhor desempenho do que os mecanismos similares precedentes. Neste trabalho é realizada uma implementação do mecanismo ICP em um simulador, e uma replicação dos experimentos realizados por Seshadri et al. é feita, com a introdução de novas observações que permitem um melhor entendimento sobre o funcionamento dos mecanismos baseados nessa estratégia, e as razões de eventuais ganhos ou perdas de desempenho provocadas pelos mesmos.

No Capítulo 2, é apresentada uma breve explicação sobre os conceitos de cache e *prefetcher* de dados, a fim de permitir ao leitor uma melhor compreensão do conteúdo deste trabalho. O Capítulo 3 traz uma compilação dos principais trabalhos da literatura sobre o problema da poluição de cache. O Capítulo 4 detalha a implementação do mecanismo escolhido em um simulador. No Capítulo 5, os resultados de desempenho do mecanismo, para *single-core* e *multicore* são apresentados. No Capítulo 6, explora-se como variações nos parâmetros do sistema afetam o mecanismo. O Capítulo 7 traz uma conclusão sobre os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

*Prefetch*A fim de devidamente entender o problema da poluição de cache e os trabalhos que buscam solucioná-la, é necessário ter conhecimento de alguns conceitos, principalmente de cache e *prefetcher* de dados da CPU.

2.1 Cache de CPU

Para contornar o problema da alta latência de acesso à memória, uma cache é introduzida entre as unidades de execução do processador e a memória principal. Esta cache aproveita a localidade temporal dos acessos à memória, ou seja, armazenando os dados dos acessos mais recentemente requisitados pelo processador, na premissa de que eles têm uma chance maior de serem acessados novamente em um futuro próximo. Como a cache tem uma resposta muito mais rápida que a DRAM, o processador não precisa esperar que o dado venha da memória principal novamente, então o desempenho é drasticamente melhorado.

Por questões de eficiência, a cache é normalmente organizada em múltiplas camadas em uma hierarquia, onde as caches mais próximas da memória principal são mais lentas e maiores, enquanto as mais próximas dos núcleos são menores e mais rápidas. A maioria dos processadores modernos de alto desempenho empregam três níveis de cache, com uma cache de nível 1 (L1) e nível 2 (L2), privadas para cada núcleo, e uma cache L3 compartilhada por múltiplos núcleos (NORI et al., 2018).

Cada nível de cache é significativamente maior e mais lento que o anterior. Um exemplo da latência *load-to-use* de cada nível da hierarquia de memória é apresentado na Tabela 2.1. Esses valores são aproximações baseadas em uma análise de um processador de 2009. Os valores exatos dependem de muitos fatores como velocidade das memórias, número de DIMMs, frequência da CPU, etc.

Quando o processador não encontra um endereço na cache, um *cache miss* ocorre, e a busca precisa ser repetida em um nível inferior da hierarquia de memória, o que implica em uma penalidade de latência. Quando o dado é encontrado, uma linha de cache inteira (todos endereços com o mesmo prefixo, normalmente 64 bytes) é trazida para a L1 e demais níveis inferiores (se forem caches inclusivas). Dependendo da implementação da cache, cada endereço da memória principal tem endereços específicos na cache para onde podem ser mapeados. Em uma cache com mapeamento direto, cada endereço possui

Tabela 2.1: Latência aproximada para acesso de dados em um processador da série Xeon 5500.

Localização	Latência (ciclos)	Latência (tempo)
Cache L1	4	1.2 - 2.1 ns
Cache L2	10	3.0 - 5.3 ns
Cache L3, linha não compartilhada	40	12.0 - 21.4 ns
Cache L3, linha compartilhada	65	19.5 - 34.8 ns
Cache L3, remota	100-300	30.0 - 160.7 ns
DRAM, local		60 ns
DRAM, remota		100 ns

Fonte: (LEVINTHAL, 2009)

um único lugar na cache. Em caches associativas, os endereços podem ser mapeados para qualquer lugar na cache. Em caches associativas por conjunto, a cache é dividida em conjuntos, e cada endereço da memória principal pode ser mapeado para qualquer bloco de cache dentro de um conjunto específico (CHANDRA et al., 2005).

Como as caches têm uma capacidade finita, quando não há um espaço livre para um endereço, alguma linha precisa ser removida para criar espaço para alguma nova linha que chega (SOLIHIN, 2015). O sistema precisa ter uma política de substituição de cache, usada para decidir qual bloco será substituído. Existem diversas políticas diferentes, que buscam identificar quais linhas têm a menor probabilidade de serem reusadas em breve. A forma mais usual de se fazer isso é assumir que quanto mais recentemente um bloco foi acessado, maiores as chances de ele ser reutilizado em um futuro próximo, então o bloco que foi acessado há mais tempo é substituído. Essa política é conhecida como LRU (*Least Recently Used*). Por ter a implementação muito mais simples e o desempenho próximo o suficiente, normalmente implementa-se uma aproximação da LRU, chamada de NRU (*Not Recently Used*) (JALEEL et al., 2010).

2.2 *Prefetcher*

Outra técnica empregada para reduzir a latência dos acessos à memória é o *prefetching*. Prevendo os dados que o processador precisará no futuro, é possível buscar esses dados da lenta memória principal para um meio de mais rápido acesso antes que o processador de fato os requisite, efetivamente mascarando a latência de acesso. Isso pode ser implementado através de software ou hardware. O *prefetching* baseado em software é normalmente realizado através da inserção de instruções específicas de *prefetch* no código pelo programador ou compilador. O *prefetch* de hardware é feito por mecanismos

Tabela 2.2: Instrução PREFETCH e suas dicas na ISA x86

Opcode	Mnemônico	Descrição
0F 18 /1	PREFETCHT0 m8	Busca os dados em m8 com a dica T0
0F 18 /2	PREFETCHT1 m8	Busca os dados em m8 com a dica T1
0F 18 /3	PREFETCHT2 m8	Busca os dados em m8 com a dica T2
0F 18 /0	PREFETCHNTA m8	Busca os dados em m8 com a dica NTA

dedicados de hardware no processador que preveem futuros acessos à memória durante a execução (BYNA; CHEN; SUN, 2009). Entre as vantagens do *prefetcher* em hardware estão o fato de poder ser ajustado para a implementação específica do processador no qual será inserido, não tornar o código de uma aplicação específico para uma determinada máquina, e não ocupar espaços no *pipeline* que poderiam ser usados por instruções da aplicação, como no *prefetcher* em software. A desvantagem é que adiciona mais complexidade ao hardware, e pode não ser tão eficiente quanto uma versão em software bem implementada.

2.2.1 *Prefetch* por software

As fabricantes de processadores normalmente fornecem formas do programador definir manualmente os *prefetches* ou oferecer dicas ao processador. Por exemplo, na ISA Alpha, uma instrução de *load* para o registrador *r31* é tratada como um *prefetch*, pois esse registrador tem sempre o valor zero e não pode ser modificado (LEONARD; BHANDARKAR, 1987). Na PowerPC, há uma instrução *dcbt* (*data cache block touch*), que faz o *prefetch* de uma linha de cache. Em x86, existe a instrução PREFETCH*h*, que faz o *prefetch* de dados para uma região mais próxima do núcleo, de acordo com dicas do programador sobre a temporalidade dos dados, como visto na tabela 2.2.1. Onde a dica T0 instrui o processador a carregar o bloco na L1, a T1 na L2, a T2 na L3 e a NTA a inserir em alguma estrutura não temporal, mas esse comportamento pode variar de acordo com a implementação do processador. O processador pode usar essa última dica para não manter o bloco na cache após seu uso (GUIDE, 2011).

O programador pode utilizar essa instrução em linguagens de alto nível através de intrínsecas dos compiladores. No ICC, é feito através da função *_mm_prefetch(char *p, int i)*, onde p é o endereço da linha e i é a dica de temporalidade, enquanto o GCC possui a intrínseca *__builtin_prefetch()*. Um exemplo de utilização do *prefetch* em software utilizando a intrínseca do GCC é mostrado no Trecho de Programa 2.1:

Trecho de Programa 2.1 – Código de exemplo do uso da intrínseca de *prefetch* do GCC

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
  __builtin_prefetch (&a[i+j], 1, 1);
  __builtin_prefetch (&b[i+j], 0, 1);
  /* ... */
}
```

Uma grande desvantagem do *prefetch* em software é que pode ser muito difícil definir onde inserir o *prefetch*, o que acaba tornando-o eficiente, na maioria das vezes, apenas para *loops*, pois é quando é possível saber em tempo de compilação que o *prefetch* poderá ser efetuado com uma antecedência previsível em relação à requisição dos dados. Um outro problema que surge é em como definir com qual antecedência os dados deverão ser buscados, pois para uma operação otimizada, é desejável que os dados cheguem na cache logo antes de serem consumidos. Existem várias complicações em definir esse intervalo em software, pois o tempo que o código executa e o atraso da memória são extremamente dependentes do hardware, e o software deve ser capaz de executar em máquinas com configurações distintas. Para definir os parâmetros específicos para uma máquina, pode-se realizar um *profiling* da aplicação a fim de descobrir a configuração ideal, porém isso adiciona um esforço a mais, e só funciona se o *profiling* for representativo das entradas que a aplicação vai consumir durante sua execução real, o que pode ser difícil (LEE; KIM; HUH, 2016).

2.2.2 Prefetch por hardware

A forma mais simples de implementar o *prefetcher* em hardware é conhecida como *Next-line* prefetcher. Nessa implementação, a cada acesso à memória, as próximas N linhas de cache são também trazidas da memória principal. Essa implementação é simples em termos de complexidade de hardware, mas tem uma baixa precisão, principalmente para acessos irregulares.

Uma implementação mais eficiente é o *Stride* prefetcher, que pode ser feito por duas maneiras: baseado no *Program Counter* (PC), ou no endereço da linha de cache. A versão baseada no PC monitora os endereços referenciados por uma instrução de *load* específica que é executada múltiplas vezes e calcula o intervalo entre esses endereços. Na próxima vez que essa instrução for carregada, o *prefetcher* já pode buscar da memória a linha de cache equivalente ao último endereço acessado por essa instrução de *load* acrescido do intervalo entre os endereços observados anteriormente. Entretanto, buscar

o endereço de um *load* quando ele é decodificado pode resultar em um *prefetch* tardio, então o *prefetcher* pode tentar buscar o endereço de um *load* futuro, multiplicando o deslocamento por algum valor. Esse *prefetch* baseado em um deslocamento também pode ser feito a partir da observação apenas dos endereços da linha de cache, porém necessita de um hardware mais complexo.

A implementação mais comum de *prefetcher* em hardware é o *stream* prefetcher (TENDLER et al., 2002), que é baseado no endereço das linhas de cache. Neste modelo, o *prefetcher* monitora possíveis acessos à memória em endereços de linhas de cache adjacentes, realizando o *prefetch* de um fluxo de acessos em forma de *stream*. Uma estrutura de hardware busca detectar acessos a endereços próximos em uma mesma direção. Quando ocorre um *cache miss*, se o endereço não faz parte de nenhum *stream* atualmente monitorado, uma entrada em uma tabela é alocada para monitorar acessos subsequentes a endereços próximos. Neste momento, o *stream* está em fase de treinamento. Quando um ou mais acessos a endereços próximos são detectados no mesmo sentido, ou seja, a apenas endereços de valor maior ou menor que o anterior, o mecanismo define um fluxo de acessos, e o *stream* entra em fase de funcionamento. Os dados podem ser carregados tanto para um *buffer* quanto para a cache.

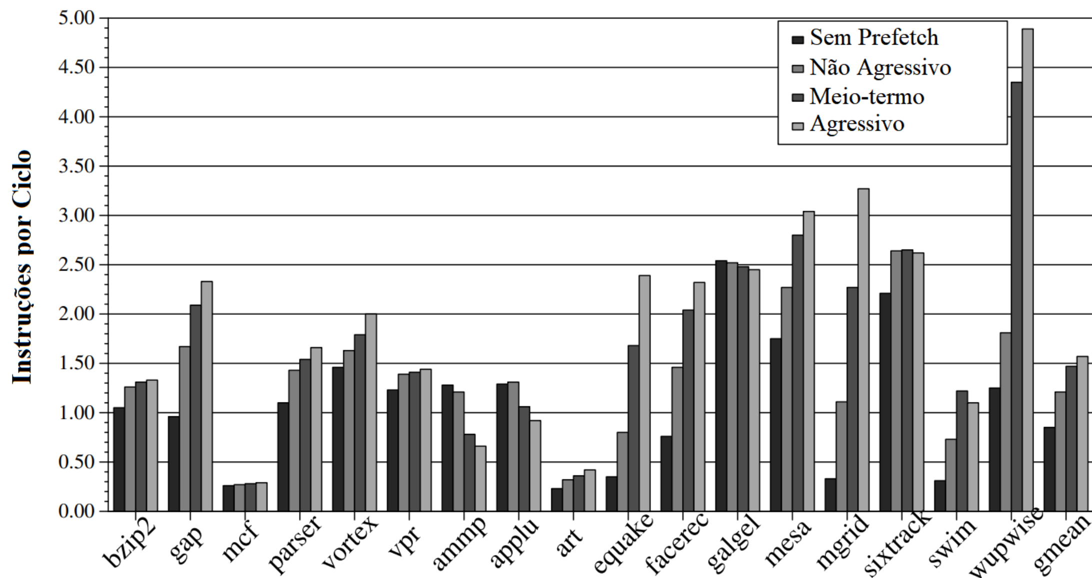
As versões baseadas no endereço da linha de cache podem detectar padrões de acesso que ocorrem por causa de interações entre múltiplas instruções, enquanto a versão baseada no contador de programa (PC) não tem essa capacidade. Essa versão pode também mais facilmente operar com antecedência em relação aos acessos atuais, garantindo que os *prefetches* cheguem sem atraso. Porém, o hardware para esse tipo de *prefetcher* é mais complexo.

2.2.3 Desempenho do *prefetcher*

Dois parâmetros definem a característica de agressividade da maioria dos *prefetcher*s: a **distância** dita quantos endereços a frente do último acesso estarão os dados buscados da memória principal, enquanto o **grau** define quantas linhas de cache serão buscadas por vez. A definição desses parâmetros afeta diretamente o desempenho dos *prefetcher*s, que pode ser caracterizado por três métricas:

Precisão: A medida de quantos dos *prefetches* realizados são de fato usados pelo processador. Uma precisão baixa significa que muitos *prefetches* são feitos inutilmente,

Figura 2.1: Desempenho de aplicações com *prefetchers* de diferentes agressividades.



Fonte: (SRINATH et al., 2007)

causando tráfego desnecessário nas interconexões e utilizando espaço das caches para dados inúteis.

Cobertura: Mede quantos *cache misses* são evitados em relação aos *misses* totais. Uma cobertura de 100% significa que o *prefetcher* consegue buscar da memória principal todos os endereços acessados pelo processador.

Pontualidade: Mesmo se os *prefetches* forem 100% precisos e tiverem uma cobertura de 100%, de nada servirá se chegarem após o processador os requisitar. A pontualidade mede se os *prefetches* chegam na hora certa, antes do dado ser requisitado pelo processador. Porém, se um *prefetcher* chegar cedo demais, pode ser retirado da cache antes de ser utilizado.

A priorização de certas métricas normalmente implica em um compromisso para o projetista. Por exemplo, aumentando a agressividade do *prefetcher* pode-se melhorar a cobertura e pontualidade, mas isso pode implicar em uma redução na precisão (PUZAK et al., 2005). Um *prefetcher* que não é agressivo o suficiente pode perder oportunidades de gerar desempenho tendo uma baixa cobertura ou pontualidade, mas um *prefetcher* muito agressivo pode prejudicar o desempenho de outras formas.

A Figura 2.1 mostra o desempenho de aplicações com diferentes agressividades de *prefetcher*. Na maioria dos casos, um *prefetcher* muito agressivo proporciona ganhos consideráveis de desempenho, mas em certos casos, como em *ammp* e *applu*, quanto mais agressivo é o *prefetcher*, pior é o desempenho.

O *prefetcher* pode afetar negativamente o desempenho do sistema em diferentes situações: quando ocupa largura de banda das interconexões, quando atrapalha o desempenho do controlador de memória e quando causa **poluição de cache**. Quando há largura de banda disponível, o único impacto negativo de um *prefetch* inútil, ou seja, que é impreciso ou atrasado, é o gasto energético desnecessário, mas o desempenho pode ser afetado quando esse *prefetch* causa contenção nas interconexões. Além disso, a imensa quantidade de requisições de acesso à memória gerada pelo *prefetcher* pode saturar o controlador de memória e comprometer o funcionamento do mesmo, aumentando a latência dos acessos (MOREIRA et al., 2016).

2.3 Poluição de Cache

As caches são pequenas e geralmente não conseguem manter todos os dados necessários simultaneamente. Quando elas estão cheias, para um novo bloco ser guardado, algum outro bloco tem que ser removido. Quando dados com potencial de reuso são substituídos por dados que não serão lidos novamente ou que são menos cruciais para o desempenho, o sistema sofre de poluição de cache, e o desempenho é comprometido.

Essa situação pode ocorrer em diversas circunstâncias, por exemplo quando em um sistema multiprogramado, uma aplicação causa a expulsão de dados de outra aplicação em uma cache compartilhada, ou quando uma aplicação acessa uma grande quantidade de dados que não têm potencial de reuso, causando a remoção dos dados atualmente presentes na cache, que tendem a ter um maior potencial de reuso.

Outra condição que causa poluição de cache ocorre quando um *prefetcher* com baixa precisão armazena os dados na cache. Essa situação é agravada porque a poluição de cache aumenta a frequência de *cache misses*, que por sua vez fazem com que mais *prefetches* sejam feitos, o que aumenta a poluição de cache ainda mais.

O problema da poluição de cache se torna ainda mais relevante em arquitetura de computadores uma vez que as aplicações consomem um volume de dados cada vez maior, o número de núcleos competindo pela memória aumenta e o aumento da velocidade das memórias não acompanha o aumento da velocidade dos processadores.

3 MITIGANDO POLUIÇÃO DE CACHE

Existem duas abordagens principais para mitigar poluição de cache: alguns mecanismos modificam o *prefetcher*, outros modificam a política de inserção da cache, enquanto alguns usam uma combinação de ambas as estratégias. Neste capítulo são apresentados os trabalhos mais influentes sobre o tema na literatura.

3.1 Modificação do *prefetcher*

Srinath et al. (SRINATH et al., 2007) propõem o *Feedback Directed Prefetching* (FDP), um mecanismo para calibrar a agressividade do *prefetcher* baseado em três métricas coletadas em tempo de execução: a precisão e pontualidade do *prefetcher* e a poluição de cache causada pelo mesmo. A técnica define cinco configurações de *prefetcher* que vão de muito conservativo a muito agressivo, que são selecionadas como transições de estado de acordo com variações nos valores definidos pelas métricas contabilizadas. O FDP também altera a política de inserção de cache baseada na poluição de cache medida. Baseado em dois limiares, se a poluição é baixa, os blocos de *prefetch* são inseridos em uma posição intermediária da fila LRU; se é média, os blocos são inseridos no último quarto, e se é alta, são inseridas na última posição.

Wu e Martonosi (WU; MARTONOSI, 2011) analisam a interferência de aplicações concorrentes no desempenho da outra em um sistema *multicore*, caracterizando o grau com o qual cada fator, como o caminhamento da tabela de páginas e, especialmente, os *prefetches*, influenciam no uso da cache LLC compartilhada. Eles propõem incluir os blocos de cache requisitados pelo sistema operacional com uma prioridade mais baixa, no fim ou no meio da fila, reduzindo a interferência do sistema operacional nas aplicações de usuário. Os autores também implementam um gerenciador de *prefetch* que estima a poluição de cache induzida pelo *prefetcher* em tempo de execução, e ajusta a agressividade do *prefetcher* de hardware de acordo com a poluição gerada.

Ebrahimi et al. (EBRAHIMI et al., 2011) ajustam dinamicamente a agressividade de cada *prefetcher* utilizando estratégias locais e globais. A política local é similar ao FDP, focando em cada núcleo individualmente. O mecanismo global pode se sobrepor a decisão local por levar em conta na decisão os efeitos das interações entre os *prefetcher*s dos diferentes núcleos quando ajusta a agressividade de cada um.

Liu et al. (LIU et al., 2011) mostram que *streams* de *prefetcher* antigas e inativas

que continuam na tabela de *streams* são uma fonte de poluição de cache. Tais *streams* podem acidentalmente ler acessos à memória como parte desses *streams* e desencadear *prefetches* incorretos, poluindo a cache. O número de *streams* inativos aumenta com o tamanho da tabela de *streams*. Simulações indicam que 92% das requisições de *prefetch* de *streams* antigos que não geraram nenhum *prefetch* nos últimos 100 mil ciclos são inúteis. Visto isso, os autores propõem dinamicamente remover *streams* antigos e inativos em uma tabela de *streams* menor.

Khan et al. (KHAN; SANDBERG; HAGERSTEN, 2014) descrevem um mecanismo que reduz a poluição de cache introduzida por blocos de *prefetch*. O funcionamento se dá através do uso de *prefetcher* em software ao invés de *prefetcher* em hardware. Os autores analisam os acessos à memória através de amostragem e identificam o que eles chamam de *loads* delinquentes, que são instruções de leitura que frequentemente causam *cache misses*. Para cada *load* delinquente, eles analisam o padrão de salto dos endereços e inserem *prefetches* de software para buscar os dados antes da execução da instrução de leitura. Khan et al. (KHAN et al., 2015) mais tarde estendem este trabalho desenvolvendo um sistema que dinamicamente combina qualquer técnica de *prefetcher* de software ou hardware disponíveis.

Jimenez et al. (JIMÉNEZ et al., 2012) propõem um mecanismo de *prefetcher* adaptativo para o processador IBM Power7. O Power7 permite ao software configurar o comportamento do *prefetcher*, incluindo a distância do *prefetcher* (quantas linhas a frente buscar), fazer *prefetch* com *stores*, e se *streams* de *prefetcher* saltarão mais de um endereço por vez. Eles implementam o mecanismo tanto com uma biblioteca que executa em tempo de execução em nível de usuário quanto com uma implementação no sistema operacional com nível privilegiado.

O *prefetcher* Sandbox (PUGSLEY et al., 2014) é um mecanismo para determinar em tempo real a técnica de *prefetch* mais apropriada para o programa em execução. O Sandbox *prefetcher* avalia os *prefetcher*s em tempo de execução adicionando os endereços de *prefetch* em um *Bloom filter*, em vez de levar o dado para a cache. Em acessos a memória subsequentes, o mecanismo verifica o *Bloom filter* para checar quais *prefetcher*s conseguiram buscar os dados e detectar a existência de fluxos futuros de acessos onde poderiam realizar mais *prefetches*.

3.2 Modificação nas políticas de substituição da cache

Jaleet et al. (JALEEL et al., 2010) propõem uma política de substituição de cache chamada RRIP (Re-Reference Interval Prediction), que possui uma variação estática e uma dinâmica. Ela é uma extensão da NRU (Not Recently Used), com intervalos de re-referenciação quase imediato e distante, mas com valores intermediários. Na versão estática, os blocos que são inseridos na cache são previstos como com um intervalo de re-referência intermediário, enquanto a versão dinâmica utiliza *set dueling* para decidir entre inserir esses blocos com uma predição de reuso em um intervalo intermediário ou distante. Blocos que recebem um *cache hit* são promovidos para reuso quase imediato. Com essas modificações, os autores afirmam mitigar a redução de desempenho causada pelo *prefetcher* e por rajadas de acessos a dados não temporais, chamados *scans*, que são comuns em muitas aplicações e causam poluição de cache.

Wu et al. (WU et al., 2011) propõem um gerenciamento de cache chamado PAC-Man, que é implementado em cima do RRIP. Nesse mecanismo, a predição do intervalo de re-referência dos blocos também considera se um bloco está na cache graças a uma operação normal de leitura e escrita ou por um *prefetch*. O mecanismo tem quatro versões que diferem no evento em que eles agem. PACMan-M é especialmente relevante porque ele prevê que todos os *prefetches* que dão *miss* na cache têm um intervalo distante de re-referenciação. Dessa forma, são priorizados blocos que vão para a cache por uma operação de leitura ou escrita, o que reduz a poluição de cache causada pelo *prefetcher*. PACMan-H difere do RRIP ao não atualizar a predição de re-referência dos *prefetches* que dão *hit* na cache, assim também priorizando as *demand requests*. Outra versão combina os dois mecanismos e a última utiliza *set dueling* para prever o intervalo de re-referência em tempo de execução.

Marco Alves (ALVES, 2014) propõe dois mecanismos: *Dead Sub-Block Predictor* (DSBP) prevê quais sub-blocos de cache serão realmente acessados e quantas vezes, para trazer apenas os sub-blocos realmente necessários para a cache. *Dead Line and Early Write-Back Predictor* (DEWP) prevê se uma linha de cache está morta, ou seja, recebeu seu último acesso, e as desliga, permitindo que sejam expulsas da cache mais cedo, evitando que poluam.

Seshadri et al. (SESHADRI et al., 2015) observam um padrão em múltiplas aplicações onde os blocos de *prefetch* são usados apenas uma vez em 95% dos casos. Como resultado dessa observação, os autores introduzem um mecanismo chamado *Informed Ca-*

ching policies for Prefetched blocks (ICP), que é dividido em duas partes: ICP-*Demote* (ICP-D) e ICP-*Accuracy Preditcion* (ICP-AP). ICP-D é o responsável por rebaixar um bloco de *prefetch* para a posição LRU quando ele recebe seu primeiro acesso de leitura ou escrita, prevendo que ele não será reusado e assim reduzindo a poluição de cache causada pelo *prefetcher*. ICP-AP é a parte do mecanismo responsável por monitorar a precisão de cada *stream* do *prefetcher* e modificar a política de substituição de cache para os *prefetches* dependendo da precisão. Quando um *prefetch* dá *hit* na cache, o bloco é promovido na fila LRU apenas se o *prefetcher* tem alta precisão. Quando um *prefetcher* é inserido na cache, se a precisão do *stream* é baixa, ele é inserido na primeira posição da fila LRU, para ser o próximo a ser substituído.

O trabalho também propõe o *Evicted-Prefetch Filter* (EPF). Quando a memória cache expulsa uma linha que foi obtida por um *prefetcher* previsto como impreciso antes do mesmo ser utilizado, o filtro recebe o endereço da linha. Se ocorrer um *cache miss* em um endereço contido no EPF, o ICP-AP incrementa a contagem de *prefetches* utilizados pelo *stream* que gerou esta linha. Sem este filtro, um laço de retroalimentação positiva pode ocorrer, onde os *prefetches* precisos porém previstos imprecisos são expulsos da cache antes de terem a chance de serem utilizados, fazendo a classificação do *stream* cair ainda mais, e não dando chance a esses *prefetches* úteis de serem utilizados.

Este trabalho emprega uma estratégia semelhante à maioria dos trabalhos anteriores, com a preterição das linhas de *prefetch* em relação às demais linhas de cache. Ademais, Seshadri et al. demonstram nos seus resultados que o ICP obtém um ganho maior de desempenho do que as técnicas anteriores. À vista disso, nosso trabalho utiliza-se deste mecanismo para estudar os efeitos desse tipo de abordagem na hierarquia de memória e como eles obtêm ganhos ou perdas de desempenho.

4 IMPLEMENTAÇÃO DO ICP

De forma a permitir uma análise do mecanismo através de simulação, o simulador *memsim* (SESHADRI, 2014) foi estendido de forma a implementar o mecanismo ICP.

Para tal, foi adicionado código responsável por implementar o ICP-D, que deve agir em duas situações:

1. Quando uma operação de leitura ou escrita dá *hit* na cache, o ICP-D deve rebaixar a linha para a prioridade mais baixa se for seu primeiro acesso.
2. Se ocorrer um *cache miss* mas existir uma requisição de *prefetch* pendente no MSHR [1], deve-se considerar que este foi o primeiro acesso a essa linha de *prefetch*, então deve-se inserir a linha com a prioridade mais baixa.

E código para implementar o ICP-AP, que requer três modificações no comportamento da LLC:

1. Quando um *prefetch* dá *hit* na cache, a linha deve ser promovida para a posição com prioridade mais alta somente se o *prefetcher* que gerou essa requisição está sendo preciso.
2. Quando um *prefetch* dá *miss* na cache, a linha é inserida com uma prioridade alta somente se o *prefetcher* está sendo preciso. Se não, a linha é inserida com a prioridade mais baixa.
3. Quando uma linha que foi inserida como imprecisa é removida da cache sem ser acessada, deve ser inserida no EPF.

O conjunto dessas modificações constitui o ICP.

4.1 Implementação

Para implementar o mecanismo, foi preciso alterar o código da LLC no simulador. Primeiramente, foi adicionado um campo na linha de cache referente ao estado de *prefetch*, ou seja, se a linha é um *prefetch* não usado, um *prefetch* usado uma vez, um *prefetch* reusado ou se não é um *prefetch*. A linha também precisa conter um identificador do *prefetcher* que a gerou.

[1] O MSHR é uma estrutura responsável por armazenar informações referentes a *cache misses* que ainda não foram resolvidos pelos níveis inferiores da hierarquia de memória.

4.2 ICP-D

Para implementar a primeira parte do mecanismo, foi necessário introduzir no trecho de código referente ao *hit* de uma operação de leitura ou escrita, um teste para verificar o estado de *prefetch* da linha. Se a linha acessada estiver no estado de *prefetch* nunca utilizado, é chamada a função *read()* da tabela que contém as *tags*, que faz a leitura de uma linha de cache passando como parâmetro o valor equivalente à prioridade baixa. Essa operação atualiza a posição da linha de cache para LRU, aproveitando as estruturas e funções já implementadas pelo simulador. Também foi implementada uma função que checa no MSHR se já existe um *prefetch* para a linha em questão em andamento. Se for o caso, uma *flag* na requisição é marcada para que o mecanismo possa inserir a linha na posição adequada na cache. O Trecho de Programa 4.1 apresenta o código para a implementação do ICP-D.

Trecho de Programa 4.1 – Código da implementação do ICP-D

```
1  switch (request -> type) {
2    case MemoryRequest::READ:
3    case MemoryRequest::READ_FOR_WRITE:
4      if (_tags.lookup(ctag)) {
5        [...]
6        switch (tagentry.prefState) {
7        case PREFETCHED_UNUSED:
8          tagentry.prefState = PREFETCHED_USED;
9          // Rebaixa a linha pra LRU
10         _tags.read(ctag, POLICY_LOW);
11         [...]
12       case PREFETCHED_USED:
13         [...]
14       case NOT_PREFETCHED:
15         [...]
16       case PREFETCHED_REUSED:
17         [...]
18       }
19     }
20     else {
21         [...]
22         // verifica se ha um prefetch para o mesmo endereco
   presente no MSHR
23         if((*_hier)[request -> cpuID][request -> cmpID + 1]->
   hasOutstandingPrefetch((PADDR(request)/_blockSize)*_blockSize
   ))
24           request -> demote = true;
25         [...]
26       }
27     case MemoryRequest::PREFETCH:
28       [...]
29     case MemoryRequest::WRITEBACK:
30       [...]
31 }
```

4.3 ICP-AP

Para implementar o ICP-AP, foi necessário implementar a tabela de monitoramento de precisão, além da implementação da checagem da precisão de um determinado *prefetcher*.

4.3.1 Tabela de precisão

Inicialmente foi implementada uma tabela para manter as informações referentes à precisão do *prefetcher*, cujo tamanho é igual ao número máximo de *streams* de *prefetch* ativas simultaneamente. As entradas na tabela são uma estrutura *AccuracyEntry*, exemplificada no Trecho de Programa 4.2.

Trecho de Programa 4.2 – Implementação da tabela de precisão

```
struct AccuracyEntry {
  uint64 avg_prefetches;
  uint64 avg_used;
  uint64 cur_prefetches;
  uint64 cur_used;
  generic_tagstore_t <addr_t, bool> ipEPF;
};
vector <AccuracyEntry> _accuracyTable;
```

O *Evicted-Prefetch Filter* (EPF) foi implementado a partir da estrutura já existente no simulador *generic_tagstore_t*, que implementa um mapa com tamanho limitado onde as entradas antigas são removidas conforme novas entradas são adicionadas, que é a mesma estrutura utilizada para implementação das caches. Cada *stream* possui um EPF.

As variáveis *cur_prefetches* e *cur_used* armazenam a quantidade de *prefetches* realizados e acessados, respectivamente, no período atual. Cada vez que se encerra um período, é feita a média dos valores de *avg_used* e *avg_prefetches* com os valores atuais, e essas são sobrescritas e os valores atuais são zerados. Isso é feito de forma a fornecer um valor que represente a média histórica, mas com mais peso para os períodos mais recentes. O período é definido como cada vez que metade da cache é substituída, ou seja, quando ocorre um número de substituições igual à metade do número de blocos da cache. Isso é feito pelo código listado no Trecho de Programa 4.3, que é executado cada vez que uma linha é expulsa da cache.

Trecho de Programa 4.3 – Cálculo da média história de precisão do *prefetcher*

```
if ((c_evictions % (_numBlocks / 2)) == 0) {
  for (uint32 i = 0; i < _accuracyTableSize; i ++) {
    AccuracyEntry &accEntry = _accuracyTable[i];
    uint64 total = (accEntry.avg_prefetches + accEntry.
   cur_prefetches) / 2;
    uint64 used = (accEntry.avg_used + accEntry.cur_used) / 2;
    accEntry.avg_prefetches = total;
    accEntry.avg_used = used;
    accEntry.cur_prefetches = 0;
    accEntry.cur_used = 0;
  }
}
```

A tabela é inicializada como no Trecho de Programa 4.4:

Trecho de Programa 4.4 – Inicialização da tabela de precisão

```
_accuracyTable.resize(_accuracyTableSize);
   for (uint32 i = 0; i < _accuracyTableSize; i ++) {
     _accuracyTable[i].avg_prefetches = 0;
     _accuracyTable[i].avg_used = 0;
     _accuracyTable[i].cur_prefetches = 0;
     _accuracyTable[i].cur_used = 0;
     _accuracyTable[i].ipEPF.SetTagStoreParameters(
   _prefetchDistance, 1, "fifo");
   }
```

4.3.2 Monitoramento de precisão

A implementação básica do monitoramento de precisão é incrementar o contador de *prefetches* de um certo *stream* quando ocorre um *prefetch* e incrementar o contador de *prefetches* usados quando ocorre um acesso regular a uma linha de *prefetch* pela primeira vez. As linhas de código que implementam este monitoramento estão listadas no Trecho de Programa 4.5. Quando ocorre um *miss*, é necessário percorrer os EPF de todos os *streams* e verificar se aquela linha foi recentemente removida da cache. Se encontrar, é incrementado o contador de *prefetches* usados do *stream* que gerou essa linha excluída.

4.3.3 Ação do mecanismo

Quando um *prefetch* dá *hit* na cache, o mecanismo deve verificar se o *stream* que gerou o *prefetch* é preciso. Para decidir se é preciso ou não, é calculada a média dos valores de médias histórica e atual. Se a média de *prefetch* usados for maior que a metade da média de *prefetch* totais, é considerado que o *stream* é preciso, e então a posição da linha na fila LRU pode ser atualizada para *Most Recently Used* (MRU). Este mecanismo

é exemplificado no Trecho de Programa 4.6.

Trecho de Programa 4.5 – Monitoramento da precisão do *prefetcher*

```
switch (request -> type) {
  case MemoryRequest::READ:
  case MemoryRequest::READ_FOR_WRITE:
    if (_tags.lookup(ctag)) {
      [...]
      switch (tagentry.prefState) {
      case PREFETCHED_UNUSED:
        [...]
        //incrementa o contador de prefetches usados
        _accuracyTable[tagentry.prefID].cur_used ++;
        [...]
      case PREFETCHED_USED:
        [...]
      case NOT_PREFETCHED:
        [...]
      case PREFETCHED_REUSED:
        [...]
      }
    }
    else
    {
      [...]
      if (request -> d_prefetched) {
          for (int i = 0; i < _accuracyTableSize; i++){
            AccuracyEntry &accEntry = _accuracyTable[i];
            if (accEntry.ipEAF.lookup(ctag)) {
              accEntry.ipEAF.invalidate(ctag);
              INCREMENT(accurate_predicted_inaccurate);
              //incrementar o contador prefetches usados
              _accuracyTable[i].cur_used ++;
    }
    case MemoryRequest::PREFETCH:
      //incrementa o contador de prefetches feitos
      _accuracyTable[request -> \emph{prefetcher}ID].
   cur_prefetches ++;
      [...]
    case MemoryRequest::WRITEBACK:
      [...]
}
```

O outro momento onde o mecanismo deve agir é na inserção de um *prefetch* na cache. O código presente no Trecho de Programa 4.7 pertence à função que insere um novo bloco na cache e é o responsável por esta parte do mecanismo. A verificação de precisão segue a mesma lógica do caso anterior.

Trecho de Programa 4.6 – Ação do mecanismo ICP-AP

```
switch (request -> type) {
  case MemoryRequest::READ:
  case MemoryRequest::READ_FOR_WRITE:
      [...]
      switch (tagentry.prefState) {
      case PREFETCHED_UNUSED:
        [...]
      case PREFETCHED_USED:
        [...]
      case NOT_PREFETCHED:
        [...]
      case PREFETCHED_REUSED:
        [...]
    }
    case MemoryRequest::PREFETCH:
      if (_tags.lookup(ctag)) { // Se for um prefetch hit
        [...]
        AccuracyEntry accEntry = _accuracyTable[request -> \emph
    {prefetcher}ID];
        uint64 total = (accEntry.avg_prefetches + accEntry.
    cur_prefetches) / 2;
        uint64 used = (accEntry.avg_used + accEntry.cur_used) /
    2;
        if (used * 2 > total) {
          // Atualiza a posicao da linha
          _tags.read(ctag);
        }
      }
      else{
        [...]
      }
      [...]
    case MemoryRequest::WRITEBACK:
      [...]
}
```

Trecho de Programa 4.7 – Inserção de linha de *prefetch* na cache pelo ICP-AP

```
if (request -> type == MemoryRequest::PREFETCH) {
  AccuracyEntry accEntry = _accuracyTable[request -> \emph{
   prefetcher}ID];
  uint64 total = (accEntry.avg_prefetches + accEntry.
   cur_prefetches) / 2;
  uint64 used = (accEntry.avg_used + accEntry.cur_used) / 2;
  if (used * 2 > total)      priority = POLICY_HIGH;
  else                       priority = POLICY_LOW;
}
  [...]
  tagentry = _tags.insert(ctag, TagEntry(), priority);
```

Quando uma linha é inserida por um *prefetcher* considerado impreciso, uma *flag* da linha chamada *lowPriority* é configurada como verdadeira. Se essa linha receber uma *demand request*, a *flag* é alterada para falsa. Quando uma linha é removida da cache, essa *flag* é testada e, se for verdadeira, a linha é inserida no EPF.

De forma geral, o mecanismo não necessitou de grandes modificações no código, mas o tempo empregado no estudo do simulador de forma a permitir uma implementação simples e que não alterasse o funcionamento do restante do simulador foi significativo.

Foi possível aproveitar a tabela já implementada no simulador que era usada para a política de inserção dinâmica na cache, que permitiu atualizar a posição de blocos através de um argumento para as funções *read()* e *insert()* da tabela de *tags* da cache.

5 SIMULAÇÃO E RESULTADOS

Neste capítulo, são apresentados os resultados obtidos da simulação do mecanismo cuja implementação é descrita no capítulo anterior.

5.1 Simulação

Foi modelado um sistema com uma hierarquia de memória similar a maioria dos processadores modernos, com uma cache L1 e L2 privadas e uma cache L3 compartilhada, com linhas de cache de 64 bytes e política de substituição LRU. Um *stream prefetcher* como descrito no Capítulo 2 de 16 entradas reside entre a L2 e a L3, sendo alimentado com os *misses* da L2 e trazendo os dados para a L3. Os parâmetros de simulação estão presentes na Tabela 5.1.

Buscou-se replicar o sistema usado no trabalho original que descreve o mecanismo (SESHADRI et al., 2015), no mesmo simulador usado pelos autores para possibilitar uma comparação mais fidedigna dos resultados. Os mecanismos propostos foram testados separadamente (ICP-D e ICP-AP) e combinados (ICP), com os *benchmarks* do SPEC 2006 (HENNING, 2006), que foi também utilizado por Seshadri et al.. Foram excluídos das análises os *benchmarks* com um número de *misses* por mil instruções (MPKI) menor que 5 na cache L2, uma vez que eles não exercem pressão suficiente na LLC, logo o mecanismo não teria efeito sobre eles. Foram gerados traços dos *benchmarks* com 200 milhões de instruções, sendo as primeiras 100 milhões usadas como *warm up*, dos trechos mais representativos de cada aplicação, fazendo uso da metodologia SimPoint (SHERWOOD et al., 2002).

Tabela 5.1: Configuração do sistema simulado

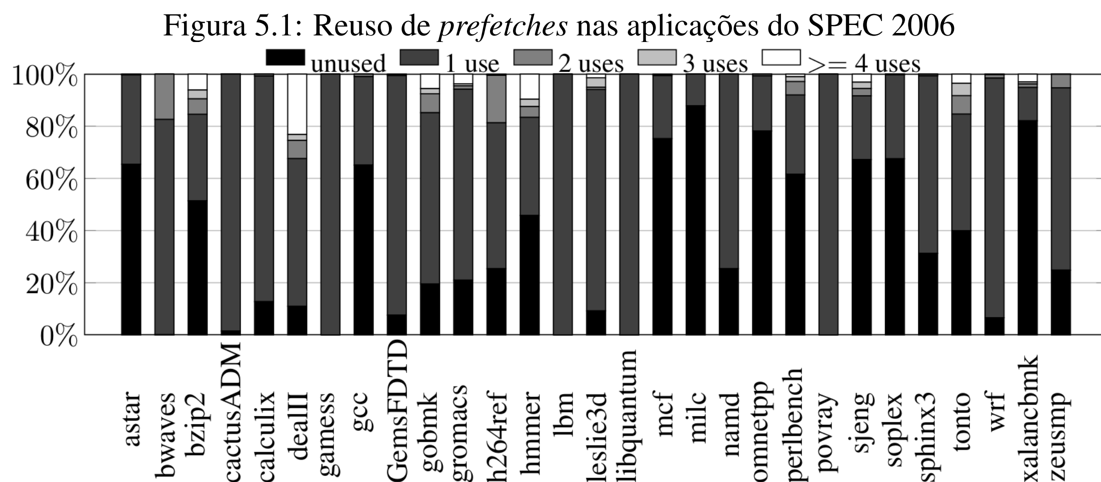
Núcleo	x86 com execução em ordem
Cache L1-D	Privada, 32 KB, associativa 2-way latência *load-to-use* de 1 ciclo
Cache L2	Privada, 256 KB, associativa 8-way latência *load-to-use* de 8 ciclos
Cache L3	Compartilhada, 1 MB, associativa 16-way latência *load-to-use* de 27 ciclos
Prefetcher	16 *streams*/núcleo, Grau = 4, Distância = 24
Controlador de Mem.	Escalonador *First-Come-First-Served*
Memória	8 bancos, latência de 200 ciclos

Fonte: Autor

5.2 Reuso de *prefetch*

O mecanismo ICP se baseia na observação de que grande parte das linhas de *prefetch* são utilizadas apenas uma vez. Para confirmar esta afirmação, foram executadas todas as aplicações do conjunto de *benchmarks* no simulador, e os acessos a cada linha de *prefetch* foram contados. A Figura 5.1 mostra, para cada aplicação do SPEC 2006, a distribuição das linhas de *prefetch* de acordo com quantas vezes são requisitadas por operações de leitura e escrita. Observa-se que de fato a maioria das linhas de *prefetch* ou não é utilizada ou é utilizada apenas uma vez.

Figura 5.1: Reuso de *prefetches* nas aplicações do SPEC 2006



Fonte: Autor

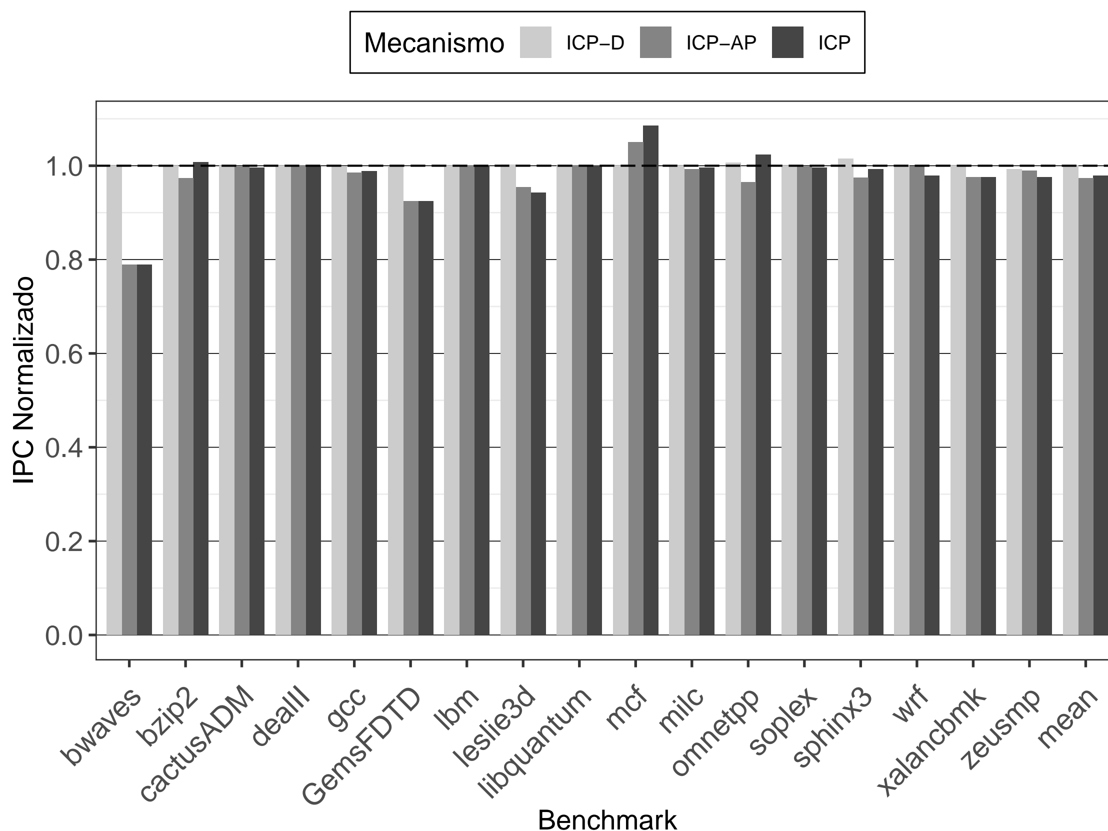
5.3 Análise *single-core*

Foram realizadas simulações de um sistema *single-core* com cada aplicação executando individualmente no processador.

5.3.1 Desempenho

A Figura 5.2 apresenta as instruções por ciclo (IPC) dos *benchmarks* sob diferentes versões do mecanismo, normalizados em relação à política LRU. O último grupo de colunas representa a média geométrica da variação de desempenho entre todas as aplicações para cada mecanismo. Os resultados mostram que o ICP degrada o desempenho de forma geral, com poucas exceções. Em média, o ICP-D não altera o desempenho,

Figura 5.2: Efeito no IPC das diferentes variações do ICP



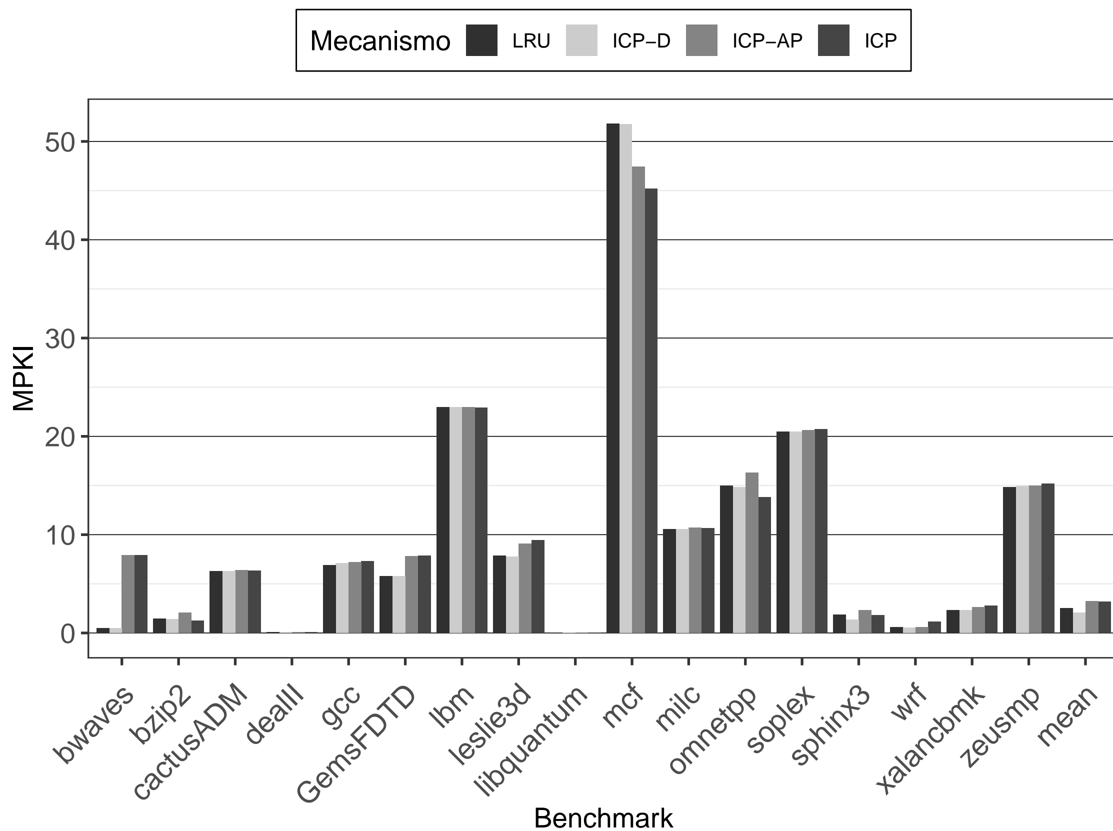
Fonte: Autor

o ICP-AP reduz em 2,3%, e a combinação de ambos degrada o desempenho em 2,1%. Entretanto, para o *mcf*, ICP-AP melhora o desempenho em mais de 5%. Em combinação com ICP-D, o mecanismo melhora o desempenho em 8%. A única outra aplicação que se beneficia dos mecanismos é *omnetpp*. Nela, nem o ICP-D ou ICP-AP sozinhos aprimoram o desempenho, inclusive o ICP-AP reduz em 3,4%, mas a combinação deles resulta em um acréscimo de 2,3% no número de instruções por ciclo. O ICP-D nunca melhora ou piora o desempenho em mais de 2% para nenhuma aplicação. Muitas aplicações sofrem degradação de desempenho com o ICP-AP. Notavelmente, *bwaves* e *GemsFDTD* perdem 21% e 7,8% de IPC respectivamente.

5.3.2 *Cache misses*

Essas variações de desempenho são compatíveis com as mudanças no número de *misses* por mil instruções (MPKI) mostrados na Figura 5.3. Quando as aplicações obtêm reduções no MPKI, elas demonstram melhor desempenho. Se o mecanismo aumenta a

Figura 5.3: *Misses* por 1000 instruções na LLC sob os diferentes mecanismos



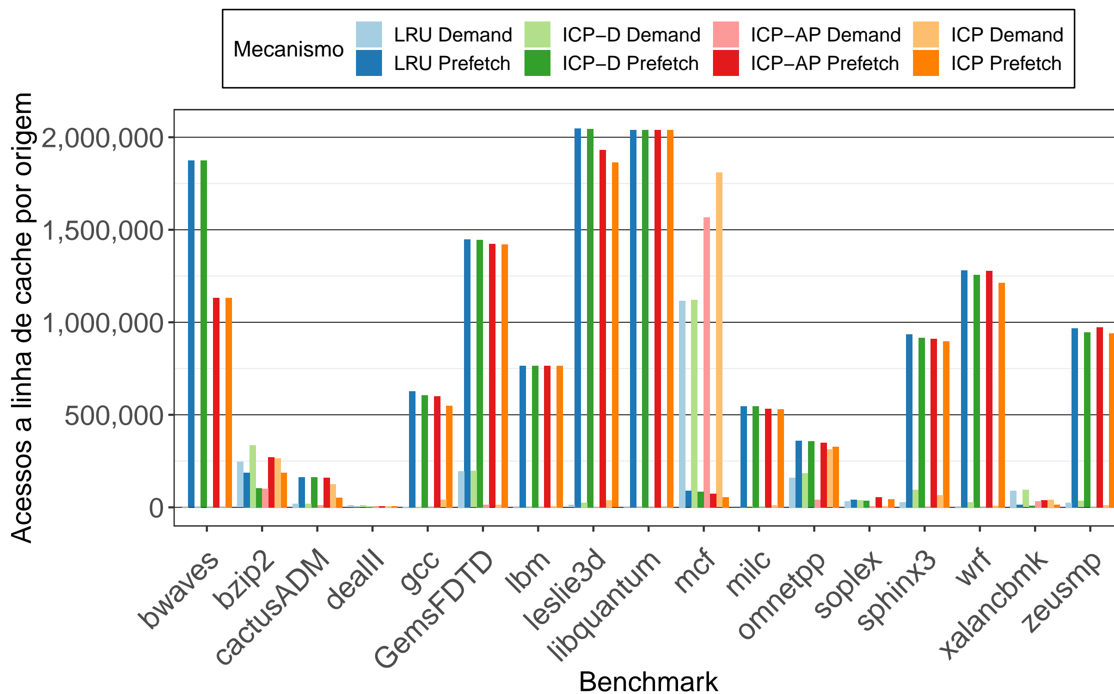
Fonte: Autor

ocorrência de *misses*, o desempenho é prejudicado. Isso não ocorre para *libquantum* nem *dealII*, porque ambos não possuem um número de *misses* na LLC que represente um gargalo no desempenho. Nesses *benchmarks*, uma redução na frequência de *misses* não retorna um ganho de desempenho significante.

5.3.3 Acessos a linhas por origem

Em média, o mecanismo não tem êxito no seu objetivo de reduzir os *cache misses*. Essa redução deveria supostamente originar-se de uma mitigação na poluição de cache causada pelo prefetcher, por evitar o despejo de linhas de cache obtidas através de *demand requests*, ou seja, requisições de leitura e escrita, que tendem a ser mais úteis, removendo as linhas de *prefetch* logo que fossem utilizadas. Os resultados demonstram que, para a maioria das aplicações, uma porção significante dos acessos são para linhas de *prefetch*, como visto na Figura 5.4. Quando o mecanismo faz essas linhas serem expulsas da cache mais cedo, ele gera mais *cache misses*. Aplicações, como *mcf*, que fazem um uso intensivo

Figura 5.4: Quantidade de *cache hits* para linhas de *demand* e linhas de *prefetch* para cada mecanismo

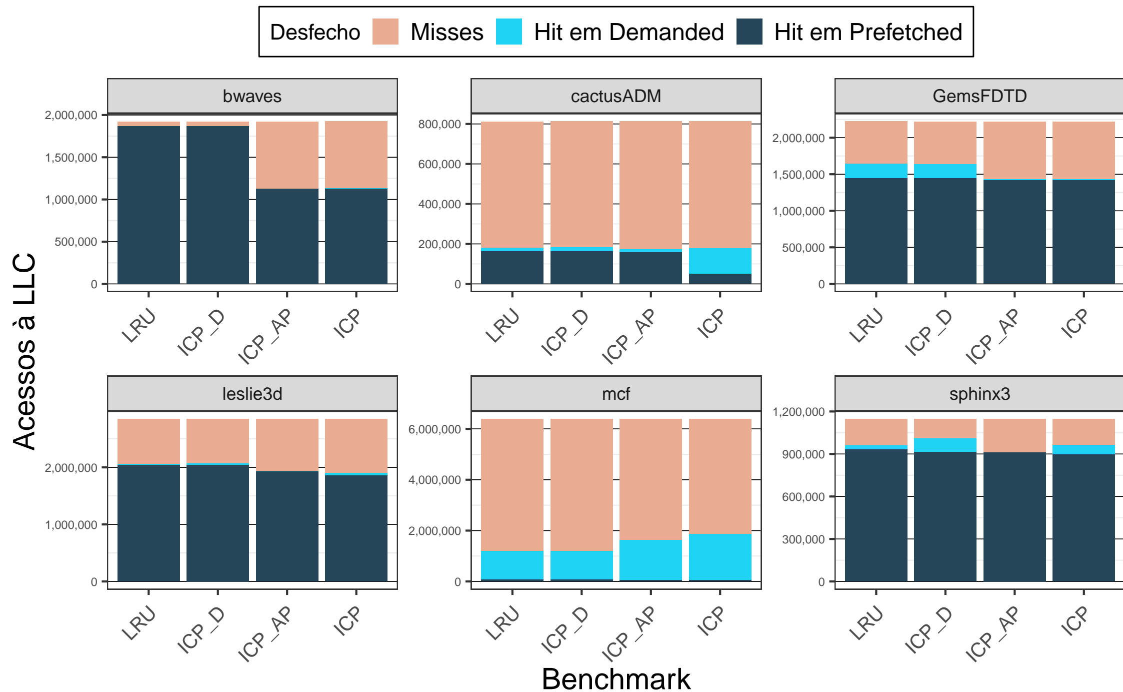


Fonte: Autor

da memória com baixo reuso e têm um padrão de acesso à memória menos previsível pelo *stream prefetcher* (PRAKASH; PENG, 2008), ganham desempenho com o mecanismo.

A Figura 5.5 mostra a distribuição dos acessos à LLC pelo desfecho, ou seja, se foram um *cache miss*, um *hit* em uma linha de *prefetch* ou um *hit* em uma linha de *demand*, das seis aplicações que sofrem uma alteração interessante no padrão de reuso das linhas com o mecanismo. Observa-se que *bwaves* perde desempenho com ICP-AP porque acessos que resultavam em um *hit* em uma linha de *prefetch* tornam-se *misses*. Em *CactusADM*, o mecanismo tem êxito em priorizar blocos de *demand* em relação a blocos de *prefetch*, mas isso não reduz os *cache misses*. Em *GemsFDTD*, acessos que resultavam em um *hit* em uma linha de *demand* se tornam *cache misses* com a introdução de ICP-AP. O mecanismo consegue minimamente aumentar o número de *hits* em linhas de *demand* para *leslie3d*, mas acessos que eram *hits* em linhas de *prefetch* tornam-se *misses*, o que reduz o desempenho. Em *mcf*, como não há acessos a linhas de *prefetch*, cada versão do mecanismo consegue ganhar desempenho dando prioridade às linhas de *demand*, tornando *cache misses* em *cache hits* nessas linhas. Já em *Sphinx3*, a versão ICP-D triunfa em transformar *cache misses* em *cache hits* em linhas de *demand*, mas ICP-AP transforma praticamente todos os *cache hits* em linhas de *demand* em *misses*, e a combinação dos dois mecanismos apresenta um meio termo entre os dois efeitos.

Figura 5.5: Distribuição dos acessos à cache por desfecho



Fonte: Autor

5.4 Análise *multicore*

Como a LLC é normalmente compartilhada entre os núcleos do processador, uma aplicação pode prejudicar o desempenho de outra concorrente ao poluir a cache compartilhada. Assim como o trabalho que introduz o mecanismo ICP (SESHADRI et al., 2015), este trabalho faz uma análise do desempenho de um sistema com o mecanismo enquanto duas aplicações executam concorrentemente. O trabalho de Seshadri et al. classifica as aplicações como sensíveis ou não à cache e ao prefetcher. Os autores classificam as aplicações como sensíveis à cache se as mesmas ganham 5% de desempenho quando a cache aumenta de 256 KB para 1 MB, e sensíveis ao prefetcher se ganham 5% de desempenho com a introdução do prefetcher no sistema. As aplicações e suas classificações são apresentadas na Tabela 5.2.

Tabela 5.2: Classificação das aplicações quanto à sensibilidade ao prefetcher e cache

Categoria	Aplicações
Não sensível à cache nem prefetcher	*milc, omnetpp, xalancbmk, zeusmp*
Sensível à cache e não sensível ao prefetcher	*bzip2, mcf, soplex*
Não sensível à cache e sensível ao prefetcher	*bwaves, sphinx3, GemsFDTD, libquantum*
Sensível à cache e ao prefetcher	*leslie3d*

Fonte: (SESHADRI et al., 2015)

Foram realizadas simulações de um sistema com dois núcleos, com cada núcleo

executando uma aplicação diferente. Foram definidas três classes de combinações de aplicações, e todas as combinações de aplicações de cada classe foi executada. Foi definido como a classe 1 as combinações de aplicações que não se beneficiam da cache mas se beneficiam do prefetcher com as aplicações que se beneficiam da cache mas não se beneficiam do prefetcher. A classe 2 é a combinação das aplicações que não se beneficiam da cache mas se beneficiam do prefetcher com as aplicações que se beneficiam de ambos. A classe 3 é a combinação das aplicações que se beneficiam da cache mas não se beneficiam do prefetcher com elas mesmas. A Tabela 5.3 apresenta as combinações de aplicações que foram executadas.

Tabela 5.3: Combinações de categorias de aplicações

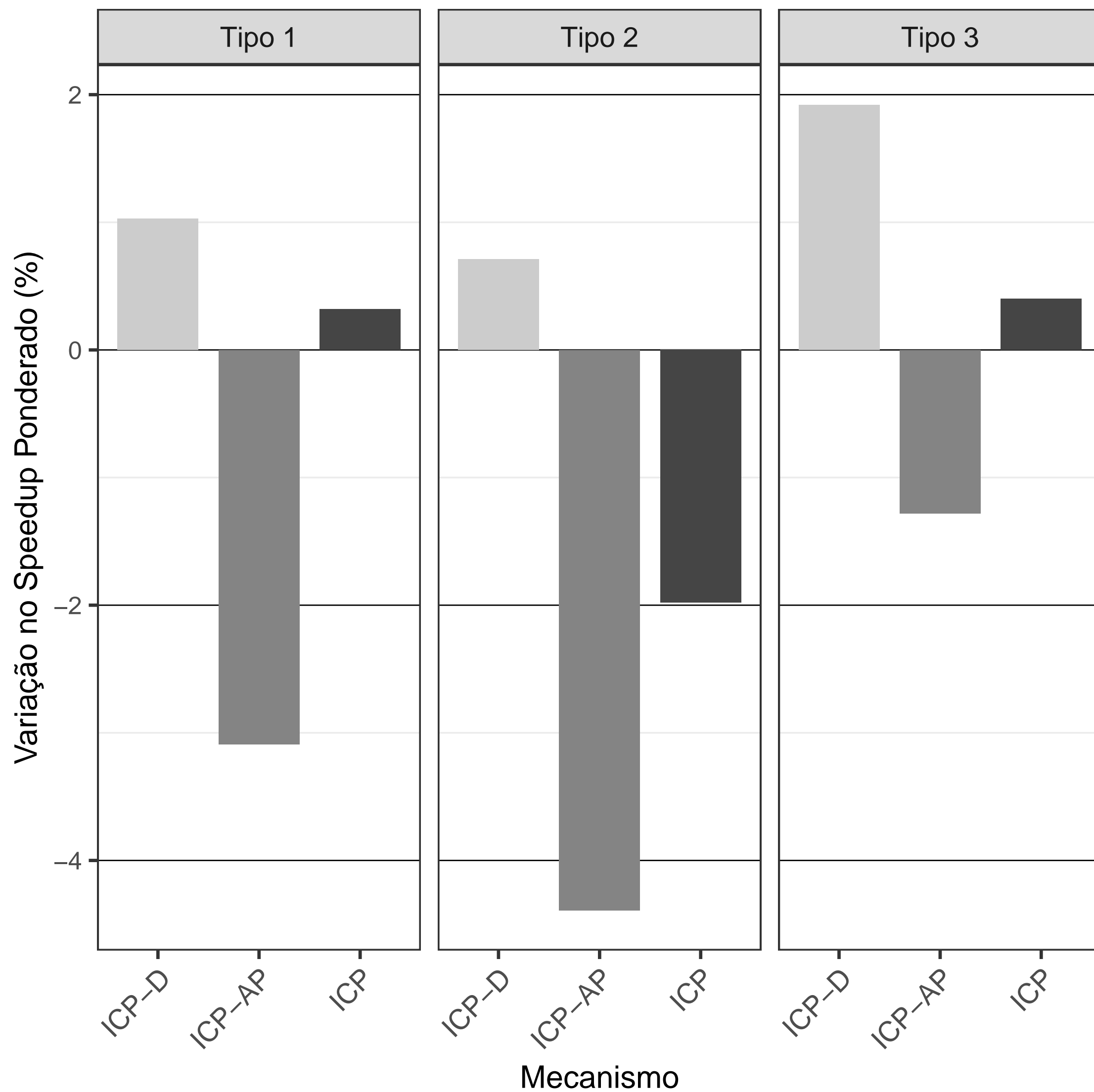
Classe	Categorias
Classe 1	Sensíveis somente à cache com sensíveis somente ao prefetcher
Classe 2	Sensíveis somente à cache com sensíveis a ambos
Classe 3	Sensíveis somente à cache com elas mesmas

Fonte: (SESHADRI et al., 2015)

A métrica usada para medir o desempenho é a aceleração ponderada (*weighted speedup*) (EYERMAN; EECKHOUT, 2008), calculada sobre o tempo de execução de cada combinação entre aplicações em um sistema sem o mecanismo. A variação percentual da aceleração ponderada com o mecanismo para cada classe de categorias de aplicações é apresentada na Figura 5.6.

O mecanismo ICP-D apresenta ganhos de desempenho de menos de 1% para as classes 1 e 2, e 1,9% para a classe 3. Na combinação de classe 3, as aplicações são sensíveis somente à cache, então remover os blocos de prefetcher da cache mais cedo pode contribuir para o desempenho. Nas classes 1 e 2, pelo menos uma das aplicações se beneficia do prefetcher, porém esses *prefetches* poluem a cache e atrapalham o desempenho da outra aplicação que é sensível à cache. Nesses casos, remover as linhas de prefetcher mais cedo traz benefícios para a aplicação concorrente, mas pode ser prejudicial para a aplicação que depende dessas linhas de *prefetch*. O mecanismo ICP-AP piora o desempenho assim como no sistema *single-core*. Há uma redução de 3% na classe 1, 4,4% na classe 2 e 1,3% na classe 3. O ICP pouco influencia no desempenho das classes 1 e 3, mas por conta do ICP-AP prejudicar tanto o desempenho da classe 2 e o ICP-D não beneficiar significativamente, a combinação de ambos acaba por reduzir em 2%.

Figura 5.6: Aceleração ponderada das diferentes combinações de aplicações com os mecanismos



Fonte: Autor

5.5 Comparação com os resultados originais

Os resultados obtidos nesse experimento não são consistentes com os resultados apresentados no trabalho original (SESHADRI et al., 2015). Isso se deve a múltiplas variáveis na metodologia, especialmente na origem dos traços. Ambos os traços foram gerados utilizando SimPoint, mas os parâmetros exatos que os autores utilizaram não são conhecidos, o que provavelmente poderia tornar os traços para os mesmos *benchmarks* significativamente diferentes. Alguns detalhes da implementação do mecanismo não foram definidos pelos autores, especialmente o limiar que é utilizado para classificar um prefetcher como preciso ou impreciso. Os autores também utilizaram aplicações do

benchmark TPC, além do conjunto de *benchmarks* SPEC 2000. Este último possui a aplicação *art* que em seus experimentos obteve uma melhora de 24% no desempenho, o que causou um impacto determinante no ganho médio nestes resultados.

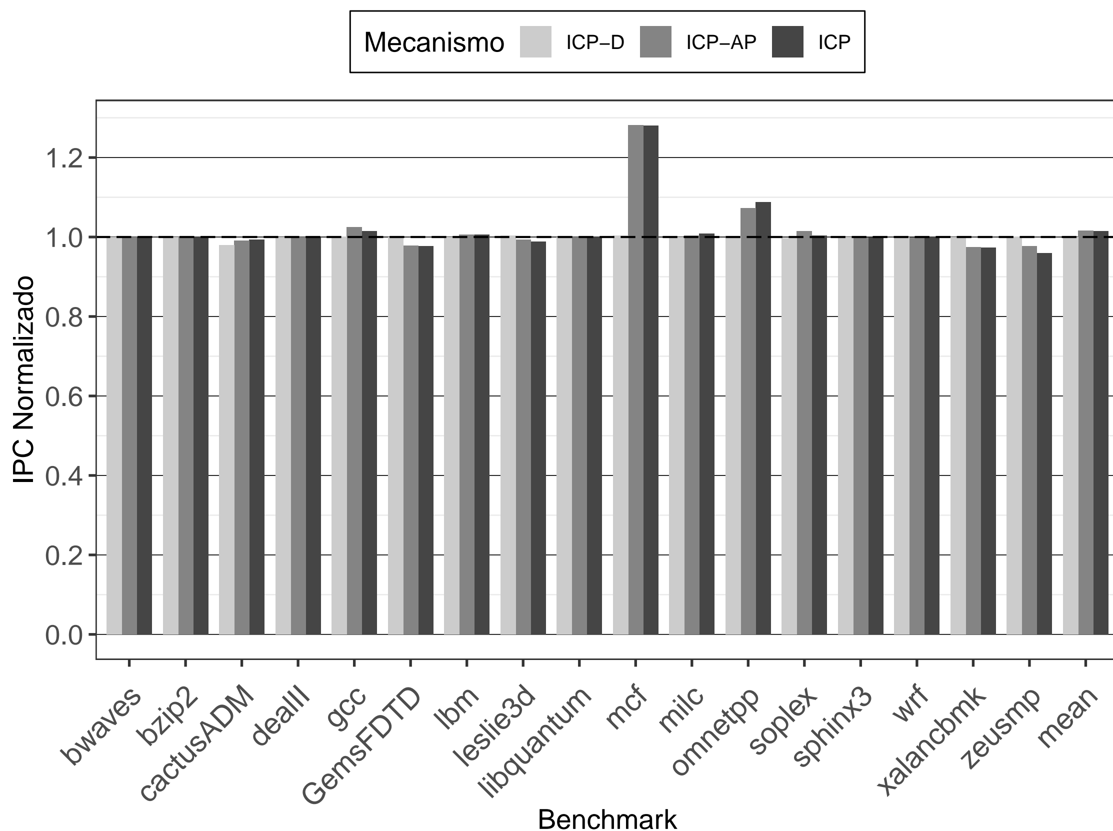
6 EXPLORAÇÃO DO ESPAÇO DE PROJETO

O desempenho das aplicações, assim como os efeitos do mecanismo, pode variar de acordo com as configurações do sistema. Uma cache de 1 MB pode ser considerada pequena para os dias de hoje, assim como um *prefetcher* com distância 24 e grau 4 pode ser considerado muito agressivo. Então, é preciso entender o funcionamento do mecanismo com outras configurações de sistema, visando compreender como os diferentes parâmetros de configuração alteram seu comportamento.

6.1 Variação do tamanho da cache

Para verificar o impacto do mecanismo em um sistema com uma cache maior, a cache L3 da simulação do capítulo anterior foi substituída por uma cache de 8 MB, com uma latência maior, de 33 ciclos, para compensar o aumento de capacidade. Os resultados de desempenho são apresentados na Figura 6.1.

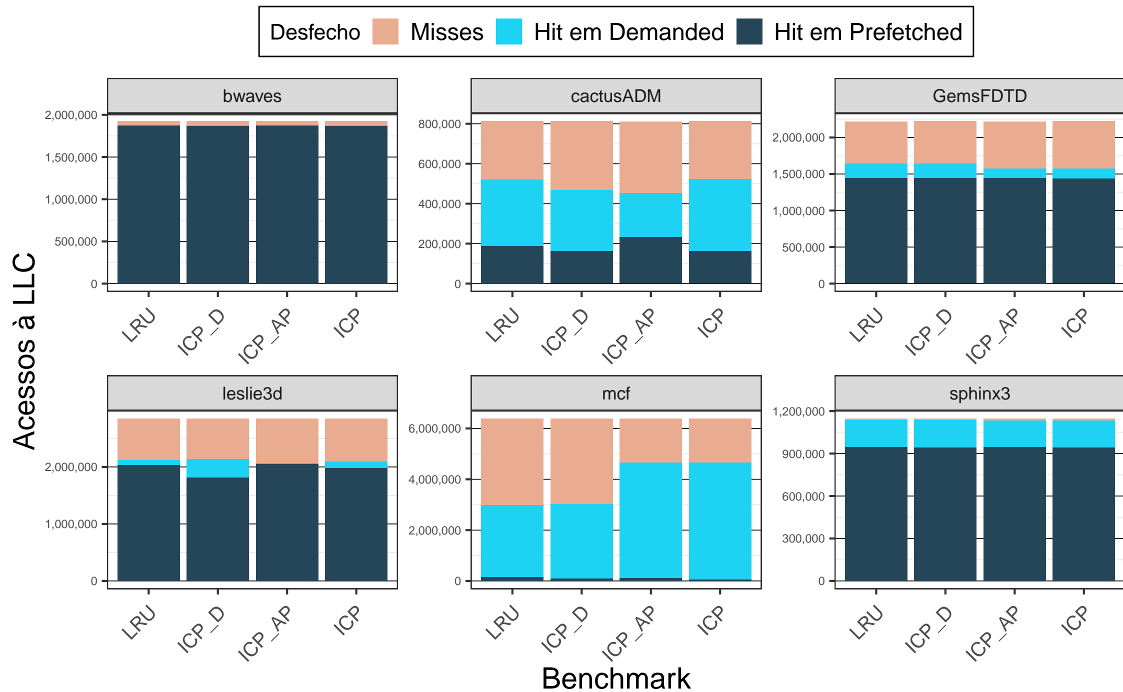
Figura 6.1: Efeito no desempenho das diferentes variações do ICP com 8 MB de cache



Fonte: Autor

Com uma cache maior, o mecanismo tem um desempenho significativamente melhor. A aceleração obtida pelo ICP-AP no *mcf* chega a 24%, e a maior degradação de desempenho é de menos de 4%. Em média, o mecanismo melhora o desempenho em 1,2%.

Figura 6.2: Distribuição dos acessos à cache de acordo com o desfecho em uma cache de 8 MB



Fonte: Autor

Situações onde o mecanismo perdia desempenho por expulsar mais cedo as linhas de *prefetch* como em *bwaves* são atenuadas pela maior capacidade da cache. Com ICP-AP, essas linhas não são mais expulsas enquanto ainda são úteis, como demonstrado pelo gráfico na Figura 6.2.

6.2 Variação do *prefetcher*

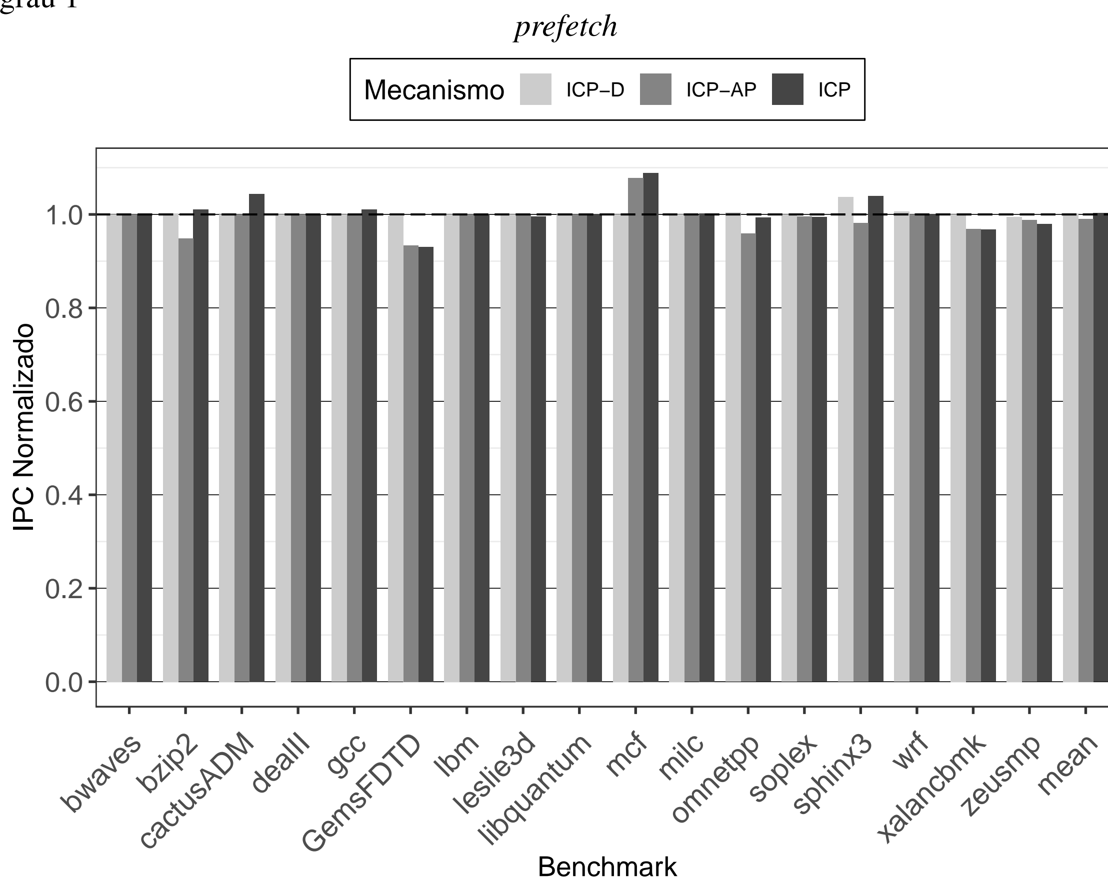
Um *prefetcher* com grau 4 pode ser considerado muito agressivo (SRINATH et al., 2007). Assim como um *prefetcher* mais agressivo pode aumentar a poluição de cache e dar mais possibilidades ao mecanismo de melhorar o desempenho, pode também amplificar os efeitos negativos provocados pelo mecanismo.

6.2.1 Redução do grau do *prefetcher*

Ao reduzir-se o grau do *prefetcher* de 4 para 1, o mecanismo que causava uma perda de 21% para o *bwaves* não causa mais nenhuma alteração no desempenho. Em contrapartida, em *omnetpp*, onde o mecanismo obtinha 2,7% de desempenho, passa a perder 1%. No *sphinx3*, onde havia um pequeno ganho de 2% para o ICP-D, com um *prefetcher* menos agressivo passa a ganhar 3,4% com o ICP. A Figura 6.3 ilustra o desempenho para todas as aplicações com o *prefetcher* de grau 1 e distância 24. Observa-se que para essa configuração de *prefetcher*, o mecanismo, na média, não altera o desempenho.

Figura 6.3: Efeito no desempenho das diferentes variações do ICP com um *prefetcher* de grau 1

prefetch

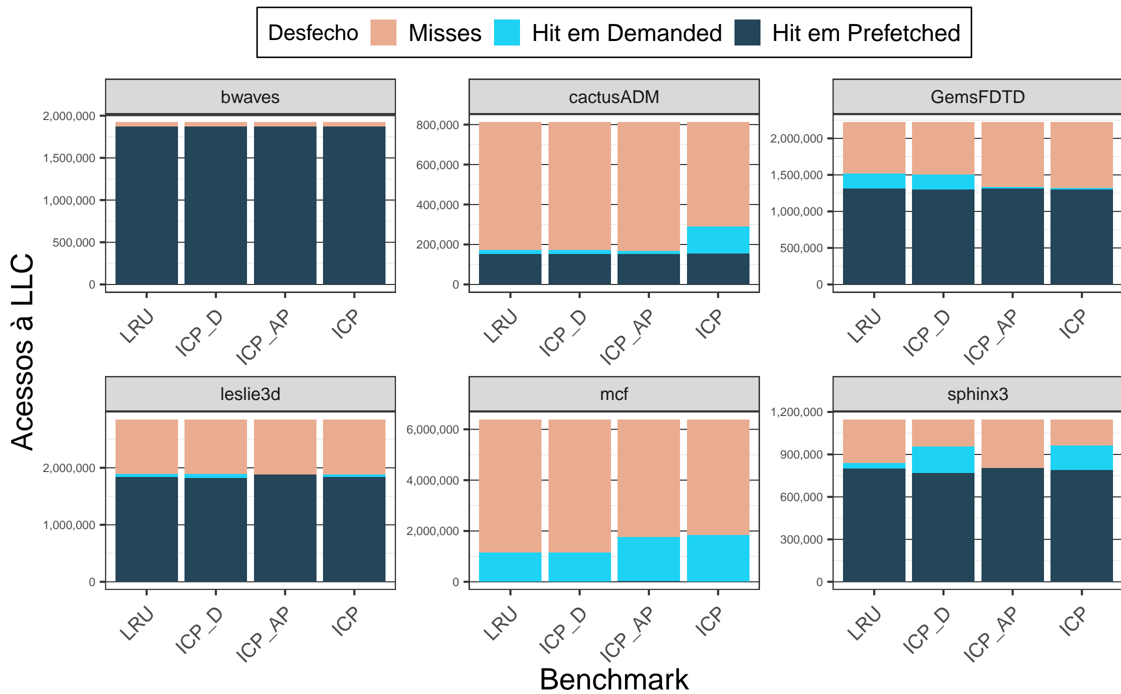


Fonte: Autor

A Figura 6.4 apresenta os dados de resultados dos acessos à LLC em um sistema com 1 MB de cache e *prefetcher* de grau 1. Pode-se perceber que, em comparação aos resultados da Figura 5.5, o desempenho é melhorado porque nas aplicações onde as linhas de *prefetch* são importantes, como *bwaves*, um *prefetcher* mais conservativo causa menos remoções de linhas de *prefetch* importantes e assim há uma redução da quantidade de

misses.

Figura 6.4: Distribuição dos acessos à cache de acordo com o desfecho com um *prefetcher* de grau 1



Fonte: Autor

Com o grau 2, o mecanismo comportou-se de forma similar a como no sistema com *prefetcher* de grau 4.

6.2.2 Redução da distância de *prefetch*

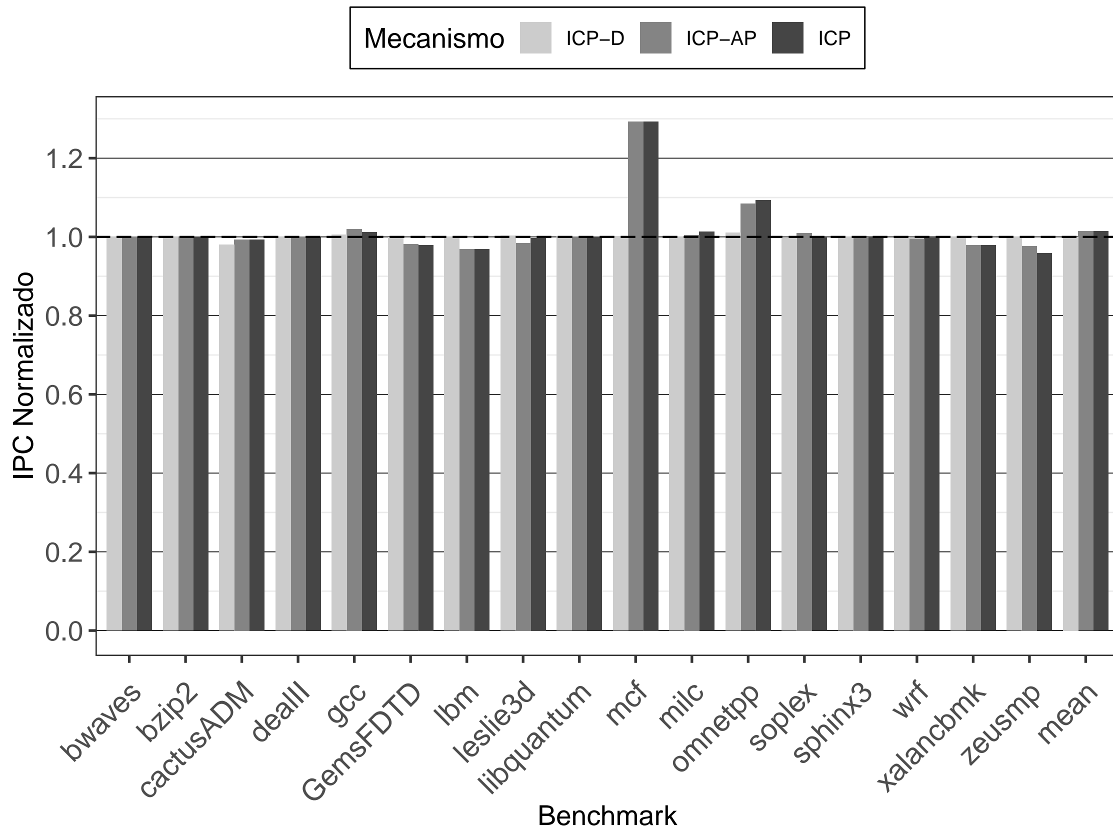
Foram exploradas distâncias menores, de 8 e 16 para o *prefetcher*. Essas alterações não resultaram em nenhuma mudança significativa no comportamento do mecanismo. Foi observado, porém, que uma configuração de *prefetcher* com distância 16 e grau 2 apresenta os melhores resultados de desempenho médio para o sistema sem o mecanismo, possuindo um ganho de 0,9% em relação à versão com distância 24 e grau 4.

6.3 Variação do *prefetcher* e cache simultaneamente

Os resultados mostram que, ao aumentar-se a capacidade da cache para 8 MB, variações nos parâmetros do *prefetcher* tornam-se irrelevantes. A Figura 6.5 apresenta os resultados de desempenho para uma configuração com um *prefetcher* de grau 1, distância

8 e uma cache de 8 MB. Percebe-se que os resultados são praticamente idênticos aos da Figura 6.1, que contém os resultados para o *prefetcher* mais agressivo também com 8 MB de cache. Ou seja, a capacidade da cache é um fator dominante em relação aos parâmetros do *prefetcher* para o desempenho do mecanismo.

Figura 6.5: Efeito no desempenho das diferentes variações do ICP com um *prefetcher* de grau 1, distância 8 e 8 MB de cache L3



Fonte: Autor

7 CONCLUSÃO

*prefetch*Este trabalho apresentou uma visão geral do problema da poluição de cache e os principais trabalhos que propõem soluções para mitigá-la. As estratégias preponderantemente adotadas na literatura se baseiam na alteração da política de substituição da cache, através de novas técnicas para prever o intervalo de reuso das linhas ou da preterição das linhas de *prefetch* e sistema operacional, e em modificações no *prefetcher*, como a alteração em tempo de execução da sua agressividade ou mesmo alternações entre técnicas de *prefetch*.

Dentre os trabalhos, um deles propõe uma técnica para mitigar a poluição de cache causada pelo *prefetcher* através de dois mecanismos: ICP-D e ICP-AP. O primeiro parte de uma observação de que as linhas de *prefetch* são usadas na maior parte do tempo apenas uma vez, então o mecanismo rebaixa as linhas de cache oriundas do *prefetcher* para a primeira posição na fila LRU quando elas recebem o primeiro acesso de leitura ou escrita. O segundo busca reduzir a poluição removendo *prefetches* imprecisos. O mecanismo monitora a precisão de cada *prefetcher* e insere os blocos dos *prefetches* preditos como imprecisos na primeira posição da fila LRU, além de não atualizar a posição das linhas que recebem um *hit* de um *prefetcher* impreciso.

Os autores que propõem o ICP demonstram que o mecanismo desempenha-se melhor que os trabalhos anteriores. Porém, os resultados mostram que considerando-se apenas os *benchmarks* do SPEC 2006, o mecanismo proposto causa na média uma piora no desempenho utilizando-se o mesmo sistema utilizado pelos autores. No pior dos casos reduzindo o IPC em 21% para o *bwaves* e, em média, 2,1%. Percebe-se, à luz dos resultados, que o mecanismo pode para certas aplicações causar a remoção precipitada de linhas de *prefetch* valiosas da cache, afetando drasticamente o desempenho de aplicações para as quais essas linhas são cruciais.

Este trabalho também avaliou o comportamento do mecanismo quando inserido em sistemas com diferentes configurações de cache e *prefetcher*. Com esses resultados, observa-se que o mecanismo pode obter uma aceleração em relação ao sistema padrão quando inserido em um sistema com características diferentes. Isso é especialmente verdadeiro para quando o sistema possui uma cache maior, onde o problema da perda de linhas importantes de *prefetch* é minimizado por conta da maior capacidade de armazenamento. Em um sistema com 8 MB de cache, o mecanismo obtém em média um ganho de 1,2% no desempenho.

REFERÊNCIAS

ALVES, M. A. Z. Increasing energy efficiency of processor caches via line usage predictors. 2014.

BYNA, S.; CHEN, Y.; SUN, X.-H. Taxonomy of data prefetching for multicore processors. **Journal of Computer Science and Technology**, Springer, v. 24, n. 3, p. 405–417, 2009.

CHANDRA, D. et al. Predicting inter-thread cache contention on a chip multi-processor architecture. In: IEEE. **High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on**. [S.l.], 2005. p. 340–351.

EBRAHIMI, E. et al. Prefetch-Aware Shared-Resource Management for Multi-Core Systems. In: **International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2011.

EYERMAN, S.; EECKHOUT, L. System-level performance metrics for multiprogram workloads. **IEEE micro**, IEEE, v. 28, n. 3, 2008.

GUIDE, P. Intel® 64 and ia-32 architectures software developer's manual. **Volume 3B: System programming Guide, Part**, Citeseer, v. 2, 2011.

HENNING, J. L. Spec cpu2006 benchmark descriptions. **ACM SIGARCH Computer Architecture News**, ACM, v. 34, n. 4, p. 1–17, 2006.

JALEEL, A. et al. High performance cache replacement using re-reference interval prediction (rrip). In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2010. v. 38, n. 3, p. 60–71.

JIMÉNEZ, V. et al. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In: **Parallel Architectures and Compilation Techniques (PACT)**. [s.n.], 2012. Available from Internet: <http://dl.acm.org/citation.cfm?id=2370837>.

KHAN, M. et al. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In: **Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2015.

KHAN, M.; SANDBERG, A.; HAGERSTEN, E. A case for resource efficient prefetching in multicores. In: **International Conference on Parallel Processing (ICPP)**. [S.l.: s.n.], 2014.

LEE, J.; KIM, T.; HUH, J. Dynamic prefetcher reconfiguration for diverse memory architectures. In: IEEE. **2016 IEEE 34th International Conference on Computer Design (ICCD)**. [S.l.], 2016. p. 125–132.

LEONARD, T. E.; BHANDARKAR, D. P. **VAX architecture reference manual**. [S.l.]: DECbooks, 1987.

LEVINTHAL, D. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. **Intel Performance Analysis Guide**, v. 30, p. 18, 2009.

LIU, G. et al. Enhancements for accurate and timely streaming prefetcher. **J Instr Level Parallelism**, v. 13, 2011.

MOREIRA, F. B. et al. A dynamic block-level execution profiler. **Parallel Computing**, Elsevier, v. 54, p. 15–28, 2016.

NORI, A. V. et al. Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In: IEEE. **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2018. p. 96–109.

PRAKASH, T. K.; PENG, L. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. **ISAST Trans. Comput. Softw. Eng**, v. 2, n. 1, p. 36–41, 2008.

PRYBYLSKI, S.; HOROWITZ, M.; HENNESSY, J. Performance tradeoffs in cache design. In: IEEE COMPUTER SOCIETY PRESS. **ACM SIGARCH Computer Architecture News**. [S.l.], 1988. v. 16, n. 2, p. 290–298.

PUGSLEY, S. H. et al. Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers. In: **International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2014.

PUZAK, T. R. et al. When prefetching improves/degrades performance. In: ACM. **Proceedings of the 2nd conference on Computing frontiers**. [S.l.], 2005. p. 342–352.

SESHADRI, V. **Source code for Mem-Sim. Retrieved May 30, 2018 from www.ece.cmu.edu/ safari/tools.html.** 2014. Available from Internet: <www.ece.cmu.edu/~safari/tools.html>.

SESHADRI, V. et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In: ACM. **Proceedings of the 21st international conference on Parallel architectures and compilation techniques**. [S.l.], 2012. p. 355–366.

SESHADRI, V. et al. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 11, n. 4, p. 51, 2015.

SHERWOOD, T. et al. Automatically characterizing large scale program behavior. **ACM SIGARCH Computer Architecture News**, ACM, v. 30, n. 5, p. 45–57, 2002.

SOLIHIN, Y. **Fundamentals of Parallel Multicore Architecture**. [S.l.]: CRC Press, 2015.

SRINATH, S. et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: IEEE. **High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on**. [S.l.], 2007. p. 63–74.

TENDLER, J. M. et al. Power4 system microarchitecture. **IBM Journal of Research and Development**, IBM, v. 46, n. 1, p. 5–25, 2002.

WU, C.-J. et al. Pacman: prefetch-aware cache management for high performance caching. In: ACM. **Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2011. p. 442–453.

WU, C.-J.; MARTONOSI, M. Characterization and dynamic mitigation of intra-application cache interference. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 2–11.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. **ACM SIGARCH computer architecture news**, ACM, v. 23, n. 1, p. 20–24, 1995.