Poluição de Cache e *Thrashing* em Aplicações Paralelas de Alto Desempenho *

Arthur M. Krause, Francis B. Moreira, Valéria S. Girelli, Philippe O. A. Navaux

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

{amkrause, fbmoreira, vsgirelli, navaux}@inf.ufrgs.br

Resumo. *Conforme os processadores evoluem, o desempenho dos sistemas computacionais se torna cada vez mais limitado pelo tempo de acesso à memória. Caches são empregadas a fim de contornar este problema, mas é necessária uma gerência inteligente dos dados que são armazenados nelas para impedir que problemas como poluição e thrashing degradem seu desempenho.*
Neste trabalho é apresentada uma análise da poluição de cache e thrashing em aplicações paralelas de alto desempenho. Os resultados mostram que caches com maior associatividade sofrem mais com estes problemas. Até 28% dos cache misses na L1 poderiam ser evitados com uma política de substituição de cache mais inteligente, chegando a até 62% na cache L2 e 98% na LLC.

Abstract. *As processors evolve, the performance of computer systems becomes increasingly limited by the memory access time. Caches are employed in order to get around this problem, but an intelligent management of the data that is stored in them is necessary to prevent problems such as pollution and thrashing from degrading their performance.*
In this work, an analysis of cache and thrashing pollution in high performance parallel applications is presented. The results show that caches with greater associativity suffer more from these problems. Up to 28% of cache misses in the L1 cache could be avoided with a smarter replacement policy, up to 62% in the L2 cache and 98% in the LLC.

1. Introdução

A velocidade dos processadores vem evoluindo em um ritmo mais veloz do que o 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 and 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

*Este trabalho foi parcialmente financiado pela CAPES e pelo projeto Petrobras 2016/00133-9.

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 et al. 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. Uma gerência de cache ineficiente permite a ocorrência de poluição de cache e *trashing*. 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, e quando dados com bom potencial de reuso retiram da cache outros dados com bom potencial de reuso, ocorre o *thrashing*.

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 má gerência da cache se agrava. Portanto, mitigar o *thrashing* e a poluição de cache torna-se um problema extremamente relevante em arquitetura de computadores. Diversos trabalhos na literatura buscam analisar e mitigar esses problemas. A maioria dos trabalhos baseia suas estratégias em alterações na política de inserção e substituição da memória cache e de alterações nos *prefetchers*.

Este trabalho propõe uma investigação sobre a ocorrência de poluição de cache e *thrashing* em arquiteturas paralelas utilizando aplicações de alto desempenho. Os resultados demonstram que para alguns benchmarks, os misses decorrentes de poluição são muito significativos, chegando a até 28% dos misses na cache L1 e 98% na cache L3.

Na Seção 2, é apresentada uma breve explicação sobre os conceitos de cache, poluição de cache e *thrashing*, a fim de permitir ao leitor uma melhor compreensão do conteúdo deste trabalho. A Seção 3 traz um resumo dos principais trabalhos da literatura sobre o problema da poluição de cache e *thrashing*. A Seção 4 descreve a métrica utilizada para quantificar esses eventos em um simulador. Na Seção 5, os resultados das simulações são apresentados. A Seção 6 traz uma conclusão sobre os resultados obtidos e trabalhos futuros.

2. Poluição de Cache e *Thrashing*

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 ca-

madas 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 1. Esses valores são aproximações baseadas em uma análise de um processador de 2009. Os valores exatos dependem de diversos fatores como velocidade das memórias, número de DIMMs, frequência da CPU, etc.

Localização	**Ciclos**	**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

Tabela 1. Latência aproximada para acesso de dados em um processador da série Xeon 5500. [Levinthal 2009]

Quando o processador requisita um dado que não está presente 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 um único lugar na cache. Em caches associativas, os endereços podem ser mapeados para qualquer bloco 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].

Quando dados com alto 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, em uma situação chamada de *scan*, 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.

Quando uma aplicação trabalha sobre um conjunto de dados maior que a capacidade da cache e acessa esses dados de forma cíclica, dados que serão reusados podem ser removidos da cache por outros dados que também serão reusados, caracterizando a situação chamada de *thrashing*.

Os problemas de poluição e *thrashing* de cache se tornam ainda mais relevantes 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. Trabalhos Relacionados

O Algoritmo de Bélády [Belady 1966] apresenta uma solução ótima para a substituição dos dados de uma cache, porém, ele depende de informações sobre acessos futuros à memória, o que é imprático para uma implementação real. Existem diversos trabalhos que analisam as deficiências da política LRU e propõem novas heurísticas para gerenciamento da cache visando reduzir tanto poluição quanto *thrashing*. Mais notavelmente, [Jaleel et al. 2010], [Qureshi et al. 2007], [Seshadri et al. 2012], [Chaudhuri 2009], [Wu et al. 2011], [Young et al. 2017], [Khan et al. 2010], [Wu and Martonosi 2011], [Jain and Lin 2016], [Jiajun et al. 2017], [Jain and Lin 2018] e [Ghahani et al. 2018].

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.

Qureshi et al. [Qureshi et al. 2007] analisam a ocorrência de *thrashing* na cache e propõem uma nova política de substituição de cache chamada BIP que insere a maioria das linhas com prioridade baixa, mas uma pequena quantidade das linhas com alta prioridade. Desta forma, garantem que pelo menos uma parte do conjunto de dados da aplicação não será removido da cache devido ao *thrashing*. Os autores também propõem a técnica

de *set-dueling* para escolher em tempo de execução se a política BIP ou LRU são mais apropriadas para a aplicação que está executando no momento.

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 e Martonosi [Wu and Martonosi 2011] analisam a interferência de aplicações concorrentes no desempenho da outra em um sistema *multi-core*, 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. As autoras 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.

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. 2012] propõem a EAF-cache, um mecanismo que armazena em um filtro os endereços das linhas recentemente removidas da cache, a fim de distiguinguir as linhas com alto e baixo potenciais de reuso. Quando ocorrem misses a linhas cujo endereço estão armazenenados no filtro, esta linha é classificada como de alto reuso, enquanto todas as demais linhas são classificadas como de baixo reuso. Os autores demonstram que este mecanismo consegue reduzir significativamente tanto a poluição de cache como o *thrashing*.

Jain e Lin em [Jain and Lin 2016] e Wang et al. em [Jiajun et al. 2017] propõem mecanismos que buscas deduzir o que o algoritmo de Bélády teria decidido para as linhas nos acessos passados e utilizar essa informação para prever o que fazer quando essas linhas forem acessadas novamente. Em [Jain and Lin 2018], os autores procuram tornar o algoritmo capaz de lidar também com *prefetches*. Ghahani et al. melhoram o mecanismo de Jain e Lin utilizando-se da contagem de hits nas regiões de memória para auxiliar na escolha da linha a ser expulsa da cache [Ghahani et al. 2018].

Entretanto, tais trabalhos realizam as análises considerando apenas um ambiente *single-core* ou *multi-core* multiprogramado, sem considerar aplicações paralelas de me-

mória compartilhada, comuns em computação de alto desempenho. Portanto, entende-se necessário um estudo da ocorrência de poluição de cache e *thrashing* em aplicações paralelas, principalmente no contexto de computação de alto desempenho, que permita avaliar se a política de substituição baseada em LRU é apropriada para esse tipo de aplicação ou se políticas mais elaboradas são necessárias.

4. Contabilizando Misses por Poluição e *Thrashing*

De forma a permitir uma análise da poluição de cache e *thrashing* em aplicações de computação de alto desempenho, o simulador arquitetural SiNUCA [Alves et al. 2015] foi estendido de forma a permitir a contagem de cache misses causados por esses eventos.

Para efetuar a contagem, as seguintes modificações foram realizadas no simulador:

- Cada conjunto associativo da cache possui um contador de linhas de cache reusadas no conjunto.
- Cada vez que uma linha de cache do conjunto recebe seu primeiro reuso, o contador de reusos do conjunto associativo é incrementado.
- Cada linha que é removida da cache é inserida em uma tabela, indexada pelo seu endereço, onde consta o número de linhas de cache reusadas no seu conjunto associativo no momento da sua exclusão.
- Cada vez que ocorre um cache miss, é verificado se o endereço da linha que resultou nesse miss está presente na tabela.
- Se o endereço estiver presente na tabela, é calculada a diferença entre o número atual de reusos do conjunto e o número no momento da exclusão dessa linha. Se a diferença for menor do que a associatividade da cache, o miss é contabilizado como um miss provocado por poluição de cache.

As modificações baseiam-se na observação de que, se no intervalo em que uma linha é removida do conjunto associativo e requisitada novamente, não houve reutilizações suficientes de linhas dentro desse conjunto que justificassem a remoção prematura dessa linha. Ou seja, uma política de substituição mais inteligente poderia ter evitado esse cache miss.

As tabelas a seguir apresentam sequências de acesso a um conjunto associativo de uma cache 2-*way*, utilizando a política LRU para seleção da linha a ser excluída da cache. A Tabela 2 demonstra uma sequência de acessos onde não há poluição nem *thrashing*. A linha A é removida no instante de tempo 3 para dar espaço à linha C. No próximo acesso, é solicitado o endereço A, que foi removido da cache quando o contador de reusos do conjunto possuia o valor 0. Como o valor atual é igual 2, ou seja, maior ou igual à capacidade do conjunto associativo, este cache miss não é considerado oriundo de poluição nem de *thrashing*.

Tempo	1	2	3	4	5	6
Endereço	A	B	C	B	C	A
Hit				✓	✓	
Reusos	0	0	0	1	2	2

Tabela 2. Sequência de acessos normal

Tempo	1	2	3	4	5	6
Endereço	A	A	A	B	C	A
Hit		✓	✓			
Reusos	0	1	1	1	1	1

Tabela 3. Sequência de acessos com poluição de cache

Tempo	1	2	3	4	5	6
Endereço	A	B	C	A	B	C
Hit						
Reusos	0	0	0	0	0	0

Tabela 4. Sequência de acessos com *thrashing*

Na sequência de acessos representada na Tabela 3, o endereço A tem um grande potencial de reuso e é removido no instante 5, quando ocorre um *scan* aos endereços B e C, que não têm potencial de reuso, e a capacidade da cache é atingida. No instante 6, a linha do endereço A é requisitada novamente. Desde o momento em que a linha é removida até o seu próximo acesso, o valor do contador de reusos do conjunto é o mesmo, então este cache miss é contabilizado como decorrente de poluição. Um algoritmo ótimo teria mantido a linha do endereço A na cache, o que teria evitado um cache miss.

A Tabela 4 demonstra uma sequência de acessos cíclica a três endereços (A, B e C) com potencial de reuso. Nos instantes de tempo 4, 5 e 6, ocorrem misses por causa de *thrashing*, que são devidamente contabilizados pelo simulador com as modificações propostas anteriormente. Novamente, uma política de substituição ótima, não considerando *bypass*, deveria manter pelo menos uma linha na cache, evitando a expulsão cíclica das linhas, o que reduziria a quantidade de misses.

Esta métrica permite, então, contabilizar os cache misses oriundos de poluição ou *thrashing* em um simulador arquitetural, e assim identificar ineficiências nas políticas de subsituição de cache implementadas, sem a necessidade de implementar o algoritmo de Bélády, que não funciona corretamente quando há *prefetchers* no sistema.

5. Simulação

Para os experimentos, a implementação OpenMP do conjunto de benchmarks NAS [Bailey et al. 1995] foi escolhida, por representar aplicações de computação de alto desempenho, pela grande quantidade de trabalhos que fazem a sua caracterização e pela facilidade de dimensionamento da carga de trabalho. A versão 3.3 do NAS foi utilizada por questões de compatibilidade com o simulador, e a classe selecionada para os experimentos foi a classe A, por ser a maior classe que poderia ser simulada em um tempo apropriado. O sistema simulado foi baseado em um processador Intel®Xeon®E5-2650, cujas especificações estão detalhadas na Tabela 5.

A utilização de memória das aplicações foi medida e os valores são apresentados na Tabela 6. O conjunto de dados de nenhuma das aplicações cabe na cache de 20MB do processador simulado, o que permite estressar a política de substituição de todos os níveis de cache.

Componente	Configuração
Nome	Intel®Xeon®E5-2650
μArch	Sandy Bridge-EP
Cores	8
Cache L1	Privada, 32KB, 8-*way*, 3 ciclos
Cache L2	Privada, 256KB, 8-*way*, 8 ciclos
Cache L3	Compartilhada, 20MB, 8 bancos, 20-*way*, 26-31 ciclos
Prefetcher L1	Stride, degree=1, distance=16
Prefetcher L2	Stream, degree=2, distance=4
Memória	4 canais, 8 bancos/canal, DDR3 1333GHz CL9

Tabela 5. Processador Modelado nas Simulações

Aplicação	Memória (MB)
Block Tri-diagonal Solver (*bt*)	72
Conjugate Gradient (*cg*)	78
Embarrassingly Parallel (*ep*)	29
Fast Fourier Transform (*ft*)	357
Integer Sort (*is*)	80
Lower-Upper Gauss-Seidel Solver (*lu*)	64
Multi-Grid (*mg*)	470
Scalar Penta-diagonal Solver (*sp*)	74
Unstructured Adaptive Mesh (*ua*)	65

Tabela 6. Utilização de memória das aplicações

As aplicações foram simuladas no SiNUCA modificado, e os cache misses de operações de leitura decorrentes de poluição e *thrashing* foram devidamente contabilizados.

5.1. LRU

Primeiramente, foi testada uma configuração com as caches utilizando a política LRU verdadeira, que ordena todas as linhas de cada conjunto de acordo com o tempo do seu último acesso. A implementação real de uma política com tamanha precisão é demasiadamente cara, mas no contexto de simulação, ela permite identificar o limite máximo de desempenho de uma política baseada em LRU.

A Figura 1 apresenta os Misses por mil instruções (MPKI) para todos os níveis de cache, onde os misses causados por poluição ou *thrashing* são destacados em laranja. A escala do gráfico para os diferentes níveis de cache está diferente a fim de forneceer uma melhor legibilidade.

Percebe-se que a aplicação *cg* possui um MPKI bastante elevado na cache L1, de 337, que se deve ao fato da aplicação possuir um padrão de acesso irregular à memória. Desses 337 misses por mil instruções, 24% são advindos de poluição ou *thrashing*. Enquanto isso, *bt* possui um MPKI de apenas 21,5, porém mais de 28% dos misses na cache L1 são causados por poluição ou *trhashing*, o maior valor entre as aplicações testadas. A aplicação *ep* é notória por não estressar o subsistema de memória, e os resultados mostram

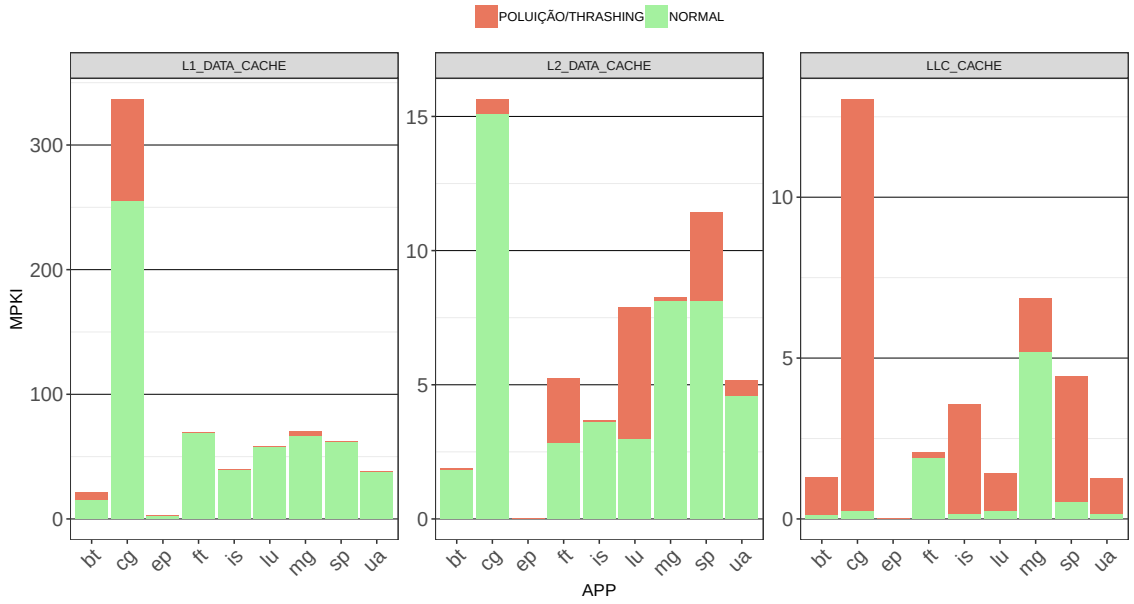

Figura 1. Misses por mil instruções das caches com a política LRU

que o seu MPKI é negligível.

Na cache L2, o MPKI das aplicações é significativamente menor, sendo pouco mais de 15 para a *cg*, o que indica que este nível de cache tem uma boa efetividade para as aplicações testadas. Neste nível, destacam-se *sp*, *ft* e *lu*, cuja proporção dos cache misses decorrentes de poluição ou *thrashing* é equivalente a 29%, 46% e 62% respectivamente. Na cache L3, praticamente todas as aplicações, com exceção de *ft*, têm uma proporção extremamente elevada de cache misses decorrentes de poluição ou *thrashing*, chegando a 98% para *cg*. Contribui para este número o fato de que a cache L3 do processador simulado é 20-*way* enquanto a dos níveis mais altos é 8-*way*. Essa maior associatividade dá mais oportunidades à política de substituição de cache efetuar um melhor gerenciamento das linhas armazenadas. Os experimentos demonstram que a política LRU acaba por excluir da cache linhas que, se permanecessem, proporcionariam uma diminuição na ocorrência de cache misses e portanto ganhos de desempenho.

5.2. SRRIP

A política de substituição de cache SRRIP [Jaleel et al. 2010] tem como objetivo minimizar a ocorrência de cache misses por poluição de cache, principalmente de *scans*, ao inserir novas linhas com uma prioridade intermediária, mantendo pelo menos uma parcela dos dados que já foram reusados na cache. Além disso, a política necessita de uma complexidade de hardware significativamente menor do que a LRU. Portanto, entende-se pertinente a análise da ocorrência de misses por poluição de cache e *thrashing* em um sistema que implementa essa política, e compará-la com os resultados obtidos para LRU.

Os gráficos da Figura 2 mostram que a proporção de cache misses decorrentes de poluição de cache ou *thrashing* observada é significativamente maior do que sob a política LRU para todos os níveis de cache. Notavelmente, para a aplicação *cg*, a quantidade de misses normais permanece praticamente igual à observada com LRU, porém, a ocorrência de misses por poluição ou *thrashing* é extremamente maior, chegando a 85% na cache

L2. Aumentos na proporção de misses decorrentes destas situações são observados em praticamente todas as aplicações.

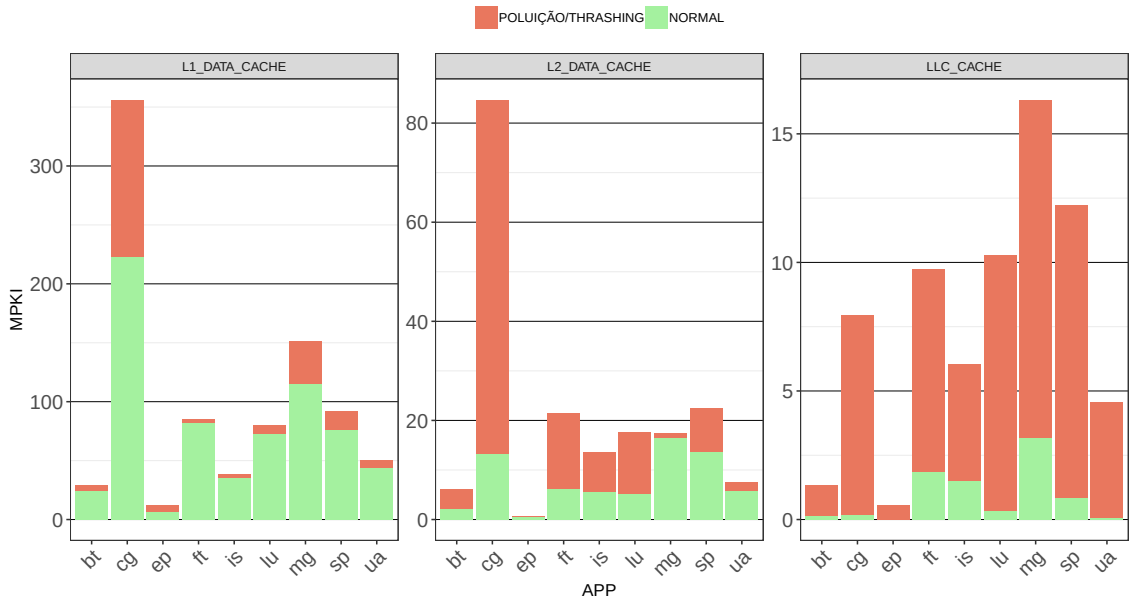

Figura 2. Misses por mil instruções da caches com a política SRRIP

Considerando-se a cache L3, todos as aplicações, com exceção de *cg*, têm o MPKI elevado em relação à LRU, o que reflete não necessariamente um pior gerenciamento deste nível de cache, mas o próprio aumento da ocorrência de misses no nível superior, que acaba por excercer uma maior pressão neste nível. No entanto, *cg* apresenta um MPKI de 7,7 na cache L3, valor que é quase metade do observado com LRU, apesar do MPKI na cache L2 ser mais de cinco vezes maior, o que indica que para essa aplicação, SRRIP proporciona um bom desempenho na LLC.

Os resultados são explicados pelo fato de que a política SRRIP tem um desempenho fraco nas caches L1 e L2 pois essas caches são pequenas e têm uma alta localidade temporal, como os autores da política afirmam em [Jaleel et al. 2010]. Também faz-se importante salientar que a implementação da política utilizou 2 bits para representar o RRPV de cada linha, enquanto LRU necessita de ao menos 5 bits por linha para a LLC, por ser 20-*way*, e 3 bits para as demais, que são 8-*way*. Portanto, é esperado que nessa configuração, a política SRRIP, apesar de projetada para reduzir os misses por poluição, permita que mais misses ocorram por ordenar as linhas por prioridade com muito menos precisão.

6. Conclusão

Este trabalho investigou o problema da poluição de cache e *thrashing* em aplicações paralelas de computação de alto desempenho. Para tal, uma abordagem que busca contabilizar a ocorrência de cache misses decorrentes de poluição de cache ou *thrashing* foi implementada em um simulador arquitetural, e as aplicações do NPB foram analisadas.

Os resultados mostram que, para a política de substituição baseada em LRU, uma grande proporção dos cache misses na LLC é decorrente de poluição ou *thrashing*, chegando até a 98% para *cg*. Como essa cache é maior e tem uma maior associatividade, a

política de substituição é mais pressionada, pois há mais opções de linhas para escolher. Uma política que visa reduzir a ocorrência de poluição, SRRIP, também foi testada. Os resultados demonstram que, para os níveis superiores de cache, essa política aumenta a ocorrência de poluição e *thrashing*, principalmente por utilizar uma menor precisão que LRU.

Como trabalho futuro, pretende-se estudar o impacto da variação na quantidade de threads, padrão de comunicação da aplicação e tamanho do problema na poluição de cache e *thrashing*, principalmente na cache compartilhada, e utilizar essa métrica para avaliação dos mecanismos propostos na literatura que buscam mitigar esses problemas, mas não fazem uma avaliação focada em aplicações paralelas de memória compartilhada. Pretende-se também desenvolver uma nova política de substituição de cache que se utiliza da métrica proposta neste trabalho para ajustar a prioridade de inserção dos blocos de cache e otimizar a utilização da cache para aplicações paralelas de memória compartilhada.

Referências

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

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *HPCC/CSS/ICESS*, pages 605–610.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center.

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

Chandra, D., Guo, F., Kim, S., and Solihin, Y. (2005). Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE.

Chaudhuri, M. (2009). Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412. ACM.

Ghahani, S. A. V., Shahri, S. M., Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. (2018). Making Belady-Inspired Replacement Policies More Effective Using Expected Hit Count.

Jain, A. and Lin, C. (2016). Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *ISCA*, pages 78–89. IEEE.

Jain, A. and Lin, C. (2018). Rethinking belady's algorithm to accommodate prefetching. In *ISCA*, pages 110–123. IEEE.

Jaleel, A., Theobald, K. B., Steely Jr, S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM.

Jiajun, W., Lu, Z., Reena, P., and Lizy, K. J. (2017). Less is More: Leveraging Belady's Algorithm with Demand-based Learning. *The Second Cache Replacement Championship: workshop schedule*, pages 1–4.

Khan, S. M., Tian, Y., and Jimenez, D. A. (2010). Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186. IEEE Computer Society.

Levinthal, D. (2009). Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18.

Nori, A. V., Gaur, J., Rai, S., Subramoney, S., and Wang, H. (2018). Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 96–109. IEEE.

Prybylski, S., Horowitz, M., and Hennessy, J. (1988). Performance tradeoffs in cache design. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 290–298. IEEE Computer Society Press.

Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. (2007). Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, page 381, New York, New York, USA. ACM Press.

Seshadri, V., Mutlu, O., Kozuch, M. A., and Mowry, T. C. (2012). The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 355–366. ACM.

Solihin, Y. (2015). *Fundamentals of Parallel Multicore Architecture*. CRC Press.

Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N. (2007). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 63–74. IEEE.

Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, S. C., Jr, and Emer, J. (2011). SHiP: signature-based hit predictor for high performance caching. *MICRO-44 '11: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441.

Wu, C.-J. and Martonosi, M. (2011). Characterization and dynamic mitigation of intra-application cache interference. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 2–11. IEEE.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

Young, V., Chen, C., Jaleel, A., and Qureshi, M. (2017). Ship++: Enhancing signature-based hit predictor for improved cache performance. In *Proceedings of the Cache Replacement Championship (CRC'17) held in Conjunction with the International Symposium on Computer Architecture (ISCA'17)*.