Acelerando requisições de prováveis cache misses com requisições em paralelo cache/DRAM

Ricardo Köhler, Marco Antonio Zanata Alves
Departamento de Informática – Universidade Federal do Paraná
rkohler@inf.ufpr.br, mazalves@inf.ufpr.br

***Resumo*—O uso de hierarquias de memória cache pelos processadores apresentam benefícios no desempenho devido a exploração da localidade temporal e espacial durante o acesso a dados. Ao manter o conjunto de dados mais frequentemente acessados ou os prováveis dados a serem acessados próximos ao processador, as memórias cache proveem um acesso mais rápido ao dados, quando comparado ao acesso a memória principal. Entretanto, para programas com baixa localidade espacial e temporal, as memórias cache podem apresentar-se como uma barreira para a busca de dados na memória principal. Ou seja, para programas que não tiram proveito das memórias cache, essas acabam por adicionar um *overhead* no tempo de acesso aos dados, pois a busca de dados é feita inicialmente na hierarquia de cache antes de ser encaminhada para a memória principal. Por outro lado, os fabricantes de processadores evitam o envio paralelo de requisições de dados para a memória cache e a memória principal, a fim de evitar inundação de requisições desnecessárias no controlador de memória. Com essa perspectiva, nesse artigo nós apresentamos o uso de um preditor simplificado de faltas de dados no Last-Level Cache (LLC) que, ao prever que uma dada requisição acabará por ser um LLC miss, induz o processador a efetuar uma requisição diretamente a memória principal, em paralelo a tradicional busca de dados percorrendo toda a hierarquia de cache. Simulações utilizando nosso mecanismo proposto apresentaram um ganho de até 14% no desempenho final das aplicações em ambiente multi-core.**

I. Introdução

Historicamente, a velocidade com a qual um processador executa suas operações e a velocidade que os dados são fornecidos pela memória DRAM possuem curvas de evolução distintas [1]. Desta forma, foram necessários métodos para mascarar esta diferença, de forma que o processador sempre esteja realizando trabalho útil [2].

Dentre estes avanços, as memórias cache representam a possibilidade de acessar os dados sem que seja necessário enfrentar as altas latências para acesso à memória principal. Contudo, nem sempre é possível evitar acessar a memória principal. As principais fontes de faltas de dados em cache são, as faltas compulsórias (durante o primeiro acesso a um endereço), faltas por conflito ou colisão (quando múltiplas linhas são mapeadas para o mesmo conjunto associativo), faltas por capacidade (quando o conjunto de dados é maior que a memória cache) e devido a invalidações (feitas por outro processo ou durante operações de E/S) [2][3][4].

Durante a execução das aplicações, os dados são reutilizados um número finito de vezes, e posteriormente removidos da cache. Aliado a esse fato, muitas vezes, um mesmo endereço possui as requisições vindas do processador concentradas em um curto espaço de tempo (alta localidade temporal). Nesse caso, tais requisições tendem a serem solucionadas pelos níveis superiores de cache (próximos aos processadores) sendo que dificilmente esse endereço será acessado na Last-Level Cache (LLC). Desta forma, otimizar o tratamento destes acessos pode constituir uma forma de melhorar o desempenho, uma vez que durante a busca de dados, a passagem pela LLC acarreta tempo ocioso de espera pelos dados e poluição da cache [5].

A utilização de técnicas como *cache-bypass* apresentam-se como uma fonte potencial de aumento de desempenho ao eliminar a necessidade de instalação dos dados com baixa localidade temporal em todos os níveis da hierarquia de cache. Tais técnicas na maioria das vezes requer caches dos tipos não-inclusiva e/ou exclusiva, com o uso de preditores de reúso [5],[6], políticas de alocação e gerenciamento da cache [7].[8][9]. De forma geral, o uso dessas técnicas de *cache-bypass* em conjunto com memórias caches inclusivas é bastante restrito, devido a política de inclusividade.

Por outro lado, a criação de requisições paralelas entre a cache e a memória principal pode ser visto como um *bypass* de requisições e apresenta potencial semelhante. Porém, ao invés de reduzir a latência de transferências de dados (DRAM $\xrightarrow{dados}$ CPU), com a geração paralela de requisições, visamos reduzir a latência do envio das requisições (CPU $\xrightarrow{req.}$ DRAM), para casos de prováveis faltas de dados na cache. Entretanto, para casos de provável cache hit, devemos evitar o envio paralelo de requisições de dados para a memória cache e a memória principal, a fim de evitar inundação de requisições desnecessárias no controlador de memória.

Nesse artigo propomos criar requisições paralelas entre a hierarquia de cache e a memória principal, originadas do núcleo de processamento. Dessa forma, buscamos reduzir a latência média das requisições que são misses previstos na LLC, e assim acelerar a execução das aplicações. Para tal efeito, buscamos prever com máxima precisão quando uma requisição resulta em um cache miss. Neste artigo apresentamos as seguintes contribuições:

- Criamos um mecanismo de requisições paralelas cache-DRAM utilizando um mecanismo simples para a predição de faltas de dados na LLC;
- Mostramos que ao ignorar a hierarquia de cache durante o envio de requisições, quando prevemos uma falta de dados durante uma requisição, pode resultar em um ganho de desempenho de até 14% para ambientes *multi-core*.

II. Contexto e Motivação

Nesta seção, será apresentado uma breve introdução sobre da memórias cache. Também serão discutidos os motivos para o uso de requisições paralelas e de mecanismos de *cache bypass*. Por fim será apresentada a motivação para a realização deste trabalho.

A. Introdução a caches com múltiplos níveis

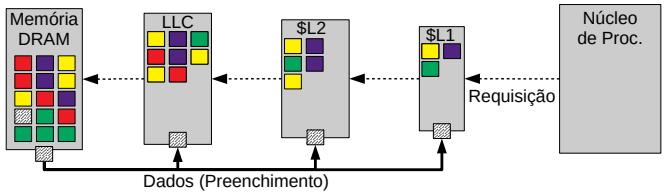
Devido a diferença entre o desempenho entre processador e memória, se fez necessário a utilização de memórias intermediárias, que minimizam esta diferença. Estas memórias, chamadas de memórias cache, mantém os dados mais frequentemente acessados próximos ao processador, tentando não apenas suprir a demanda por dados das unidades de processamento, mas também garantindo a igualdade de serviço entre todos os núcleos [2][5]. O aumento da capacidade de armazenamento e o aumento da profundidade das hierarquias de cache, buscando explorar melhor a localidade temporal e espacial, ocasionou diferenças de projeto importantes quanto a interação de múltiplos níveis de cache. Existem três abordagens mais comumente adotadas em relação à interação dos níveis da hierarquia de cache, exclusivo, não-inclusivo e inclusivo [10]. Tais abordagens impactam significativamente nas políticas de coerência e manutenção dos dados na hierarquia de cache. O uso de caches inclusivas (padrão mais adotado pela Intel, por exemplo) garante que os dados dos níveis superiores de cache estejam presentes nos níveis inferiores. Esse modo permite simplificações do protocolo de coerência de dados, como o uso do diretório junto com a LLC. Para tais sistemas inclusivos, uma falta de dados na LLC indica que o dado não está presente em nenhum dos níveis da hierarquia de memória cache. Para esses sistemas, pode-se analisar o comportamento de acessos a Last-Level Cache (LLC) de forma a prever quando um dado deverá ser requisitado na memória DRAM.

B. Cache Bypass e Requisições paralelas

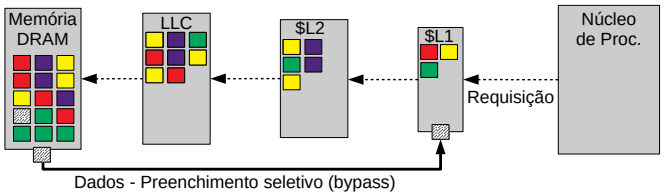
Para aumentar o desempenho e reduzir a poluição da cache pode-se adotar estratégias para realizar o *bypass* (contornar) toda ou parte da hierarquia de cache.

Tradicionalmente, cache *bypassing* é empreendido em conjunto com as políticas de substituição e preditores de reúso dos dados, buscando otimizar tanto o uso do espaço de armazenamento quanto o uso da largura de banda do processador. Pode-se realizar o *bypass* de cache através de *software*, utilizando instruções específicas incluídas no código fonte, tais como MOVNTDQA e MOVNTPD (X86), permitindo que uma instrução busque/armazene dados sem que seja necessário a utilização da cache. Instruções deste tipo apresentam dicas para que os dados não sejam instalados nas caches. Contudo, a implementação destas dicas é dependente da implementação do processador, e assim, podem acabar sendo ignoradas [11].

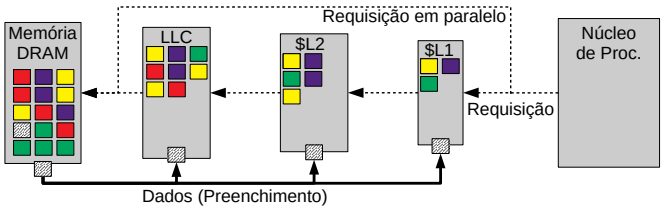
Utilizando cache *bypass*, buscamos também ignorar as latências provocadas pelos múltiplos níveis de cache, instalando os dados no nível da hierarquia onde é previsto que seja mais útil. Uma representação pode ser vista na Figura 1, onde a figura 1a representa o fluxo normal de uma requisição, que por não ser satisfeita em qualquer um dos níveis de cache acaba encaminhando-se para a memória principal e ao retornar, a cada nível de cache é instalada uma linha com os dados, a figura 1b representa o fluxo da mesma requisição, ao se empregar uma técnica de cache *bypass*. Tal técnica é factível em caches que não utilizam uma abordagem inclusiva, uma vez que ao realizar este procedimento, a propriedade de inclusão da cache seria violada. Além disso, técnicas como first-word-first [12], mostram-se mais elegantes para redução da latência de dados durante a movimentação DRAM $\xrightarrow{dados}$ CPU.



(a) Requisição de dados tradicional com instalação tradicional de dados na hierarquia de cache.



(b) Requisição de dados tradicional com instalação usando *bypass* de cache.



(c) Requisições paralelas cache/DRAM com instalação tradicional de dados na hierarquia de cache.

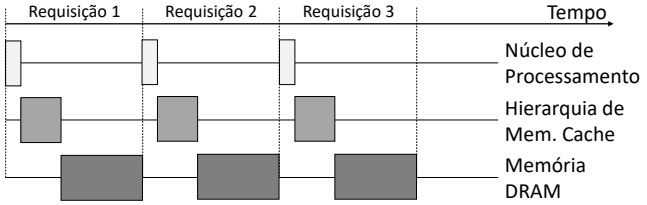
Figura 1: Fluxo de requisições e dados entre processador e memória principal.

Ao realizar requisições paralelas (utilizada nesse trabalho), segue-se o caminho inverso ao *bypass* de dados, realizando uma emulação de um *bypass* de requisições. O uso de requisições paralelas permite que seja feita uma especulação acerca de dados que eventualmente não se encontram na memória cache. Desta forma, pode-se mascarar a latência de busca através da hierarquia de cache. A figura 1c apresenta uma visão de alto nível acerca do fluxo de requisições paralelas em um processador.

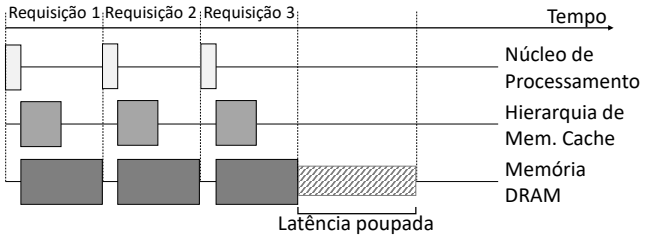
Como ponto negativo, a criação de requisições paralelas pode acarretar congestionamento do barramento entre o processador e a memória. Além disto, essa inundação poderá afetar o controlador de memória, atrapalhando o escalonamento das requisições legítimas a serem feitas à memória, interpondo requisições especulativas que podem acabar por serem satisfeitas pela hierarquia de memória cache.

C. Motivação

Para ilustrar os ganhos potenciais de nossa técnica, a Figura 2 apresenta dois sistemas com o mesmo caso, onde estamos percorrendo uma lista encadeada formada de três elementos, cujos endereços não estão presentes na memória cache. O primeiro sistema, figura 2a apresenta o fluxo onde as requisições são feitas de forma normal, buscando primeiramente os dados na hierarquia de cache, e então procedendo para a memória principal. No segundo sistema, figura 2b, quando se utilizando de requisições paralelas, após aprender o comportamento de acesso das requisições, torna-se possível encaminhá-las diretamente a memória principal, de forma paralela, e assim poupando a latência resultante da busca através da hierarquia da cache.



(a) Requisição através da hierarquia de cache.



(b) Requisição enviadas paralelamente a cache/DRAM.

Figura 2: Fluxo de requisições para memória principal

Dessa forma, notamos que o desempenho do preditor é fator determinante por dois motivos principais: 1) Se a precisão for baixa, as requisições que poderiam se aproveitar do envio paralelo à memória principal acabaria por continuar sofrendo a latência da hierarquia de cache, ou seja, perdemos oportunidades, ou ainda; 2) Se muitas requisições paralelas forem enviadas erroneamente, quando os dados se encontram na hierarquia de cache, isso ocasionaria um congestionamento do barramento entre a memória principal e o processador, interferindo em requisições legítimas.

III. Arquitetura do mecanismo

Para habilitar o núcleo de processamento a enviar requisições de forma paralela para a memória principal, se faz necessário um pedaço de lógica implementado juntamente ao núcleo de processamento. A Figura 1c apresenta uma visão do fluxo das requisições em alto nível, e como o mecanismo se porta em relação à hierarquia de cache presente no processador. Cada núcleo de processamento deve replicar tal mecanismo de forma independente para encaminhar requisições diretamente a memória principal.

A parte central do mecanismo é baseada no preditor desenvolvido para ser utilizado em DRAM-Caches [13]. Para as previsões, são utilizados conjuntos de contadores. Estes contadores são organizados em uma tabela de 256 entradas chamada de Memory Access Counter Table (MACT), sendo esta quantidade de entradas considerada suficiente pelos autores. A tabela é indexada através do endereço da instrução (PC/IP) causadora do acesso aos dados.

Em nossa proposta, quando uma requisição de dados é gerada, ela é enviada através de toda a hierarquia de cache. Paralelamente, o endereço da instrução causadora da requisição sofre um processo de *hash* para indexar a MACT e acessar uma entrada. Cada entrada da tabela comporta um contador saturado de 3 bits. Caso o contador seja maior ou igual a este *threshold*, as requisições são enviadas paralelamente para memória principal. Dessa maneira, não aumentamos o tempo do caminho crítico de requisições de dados.

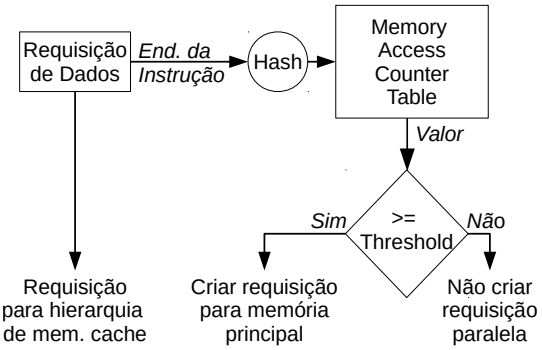


Figura 3: Fluxo de requisições com o mecanismo responsável por executar as requisições paralelas.

Para atualizar os contadores, o mecanismo continua acompanhando a requisição. Durante uma falta de dados na LLC, o contador correspondente é incrementado. Para o caso de um cache *hit*, o contador é decrementado. Durante a inicialização do mecanismo todos os contadores são zerados, evitando que as requisições inundem o sistema, durante a fase de aprendizado.

IV. Metodologia

Para avaliar o mecanismo proposto, modelou-se um sistema *multi-core*, dotado de 4 núcleos de processamento. Este sistema baseia-se na microarquitetura Intel Skylake [14] e foi modelado utilizando um simulador com precisão de ciclo.

A. Simulador

Para avaliar o mecanismo proposto, foi desenvolvido um simulador com precisão de ciclo, baseado no simulador SiNUCA [15]. Este simulador modela o comportamento dos detalhes estruturais da microarquitetura para o núcleo de processamento com execução fora de ordem, a hierarquia de cache não bloqueante (emulando o miss status handle register) e a memória principal DDR-3. Foram simuladas 200 milhões de instruções mais representativas para cada aplicação do

SPEC CPU-2006 [16], obtidas através do método PinPoints [17].

A Tabela I lista os detalhes da configuração para o sistema simulado.Cada núcleo de processamento contém caches de nível L1 e L2 privativas. A LLC foi modelado de forma monolítica e é compartilhada entre todos os núcleos, possuindo 1 MB por núcleo presente no sistema. O último nível de cache é inclusivo com os níveis superiores.

Considerando que nesse trabalho o custo das previsões errôneas causa impacto negativo no desempenho do sistema, esse impacto foi modelado no sistema de DRAM como uma requisição extra. Dessa forma, um acesso na memória principal (DRAM) que venha a ser resolvido pela memória cache, poderá atrasar as demais instruções de acesso à memória, devido aos sinais já terem sido escalonados pelo controlador de DRAM.

Tabela I: Configuração do sistema

Core	Núcleos: 4; Clock 3.2 GHz; IPC máximo 4. Fetch: 16 B; ROB: 224 entradas; MOB: 72-read 56-write RS: 97 entradas; Branch Predictor: Piecewise Linear [18];
L1 Inst. Cache	32 KB; Linha: 64 B; Latência: 3 ciclos; Privada; Conj. Assoc.: 8 vias; Writeback; MSHR: 16 Requisições;
L1 Data Cache	32 KB; Linha: 64 B; Latência: 3 ciclos; Privada; Conj. Assoc. 8 vias; Writeback;
L2 Cache	256KB;. Linha: 64 B; Latência: 9 ciclos; Privada; Conj. Assoc. 4 vias; Writeback;
Last Level Cache	4-Cores: 4MB (Compartilhada); Linha: 64 B. Latência: 44 ciclos. 8 vias. Writeback;
Memory Controller	4-Cores: 64 entradas memory queue;
DRAM	DDR3; Canais:2; 1 Rank com 8 Banks por canal; Row Buffer 8 KB; RP-RAS-CAS (11-11-11) (13.75 ns); Barramento: 8B; Frequência DRAM: 800 MHz;

B. Caracterização das cargas de trabalho

Para avaliação do comportamento do mecanismo, foi usado o conjunto de aplicações do SPEC CPU-2006 [16].

As aplicações foram separados conforme o volume de acessos à memória principal. Desta forma, quanto mais acessos à memória principal, mais oportunidades tem-se para poupar ciclos utilizando requisições paralelas, e de forma análoga o impacto de predições errôneas se torna maior. Para isto foi utilizado a métrica de Misses per Kilo Instructions (MPKI). As aplicações são classificas em alta, média e baixa intensidade de memória. A tabela II apresenta a classificação de acordo com o número de MPKI.

Tabela II: Classificação das aplicações da carga de trabalho SPEC CPU-2006 com base em MPKI ocorridos no LLC.

Baixa Intensidade (MPKI $\leq$ 6)	bzip2 (2.8), calculix (0.1), dealii (0.1), gamess (0.1), gobmk (0.4), gromacs (1.0), h264ref (1.5), hmmer (2.7), namd (3.0), perlbench (1.3), povray (0.1), sjeng (0.4), tonto (0.1), xalancbmk (3.6)
Média Intensidade (6 $<$ MPKI $<$ 10)	astar (6.9), cactusadm (9.0)
Alta Intensidade (MPKI $\geq$ 10)	bwaves (20.0), gcc (20.3), gemsfdtd (24.3), lbm (31.8), leslie3d (25.9), libquantum (24.5), mcf (60.9), milc (15.1), omnetpp (18.8), soplex (27.6), sphinx3 (12.7), wrf (13.7), zeusmp 16.0)

Com as aplicações catalogadas, foram gerados de forma aleatória 20 grupos de aplicações, sendo 10 (H0-H9) deles somente com aplicações de alta intensidade, 5 (M0-M4) mesclando aplicações de alta e média intensidade, e 5 (L0-L4) mesclando aplicações de alta e baixa intensidade. As aplicações que compõe os grupos são apresentadas na Tabela III. Cada aplicação aparece apenas uma vez em cada grupo.

Tabela III: Lista de grupos com 4 aplicações.

High	**H0**	bwaves+gcc+libquantum+mcf
	H1	libquantum+soplex+sphinx3+wrf
	H2	mcf+omnetpp+soplex+wrf
	H3	mcf+soplex+sphinx3+wrf
	H4	bwaves+gcc+mcf+soplex
	H5	gcc+libquantum+sphinx3+wrf
	H6	bwaves+gcc+omnetpp+sphinx3
	H7	gcc+libquantum+soplex+wrf
	H8	bwaves+gcc+soplex+sphinx3
	H9	bwaves+libquantum+mcf+wrf
Medium	**M0**	astar+cactusADM+bwaves+soplex
	M1	cactusADM+astar+libquantum+sphinx3
	M2	mcf+astar+cactusADM+sphinx3
	M3	astar+cactusADM+gcc+soplex
	M4	cactusADM+bwaves+omnetpp+astar
Low	**L0**	xalancbmk+gromacs+bwaves+soplex
	L1	gromacs+bwaves+omnetpp+xalancbmk
	L2	soplex+xalancbmk+gromacs+sphinx3
	L3	xalancbmk+libquantum+mcf+gromacs
	L4	gromacs+xalancbmk+soplex+wrf

Para análise de desempenho foi escolhida a métrica *speedup* ponderado [19], que representa uma medida igualitária na quantidade de trabalho realizado por cada núcleo de processamento.

V. RESULTADOS

Nessa seção apresentaremos resultados sobre a precisão do mecanismo de predição de falta de dados na memória cache e os resultados referentes ao desempenho do sistema *multi-core*.

A. Precisão do mecanismo de predição

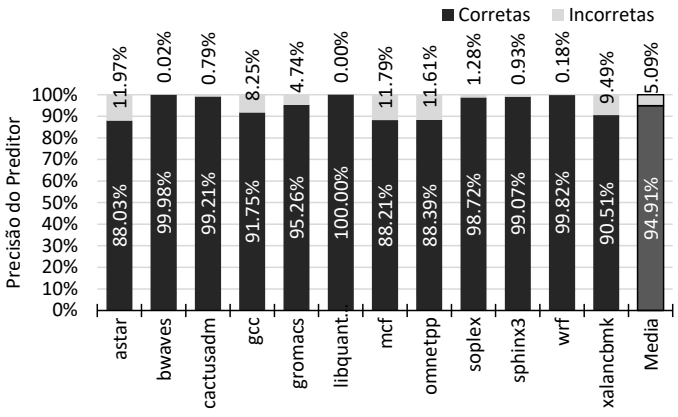


Figura 4: Precisão do preditor de misses da LLC

O preditor utilizado segue o que foi proposto no artigo [13]. Apresentando-se como uma tabela simples, possuindo 256 entradas com capacidade de armazenamento de 3 bits, apresenta

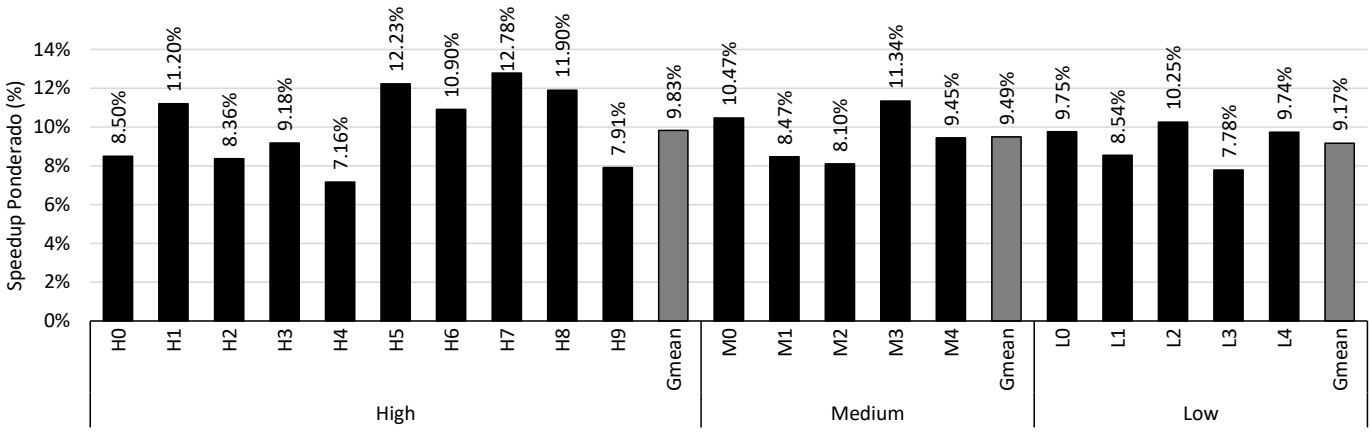


Figura 5: *Speedup* para grupos de aplicações de alta, média e baixa intensidade.

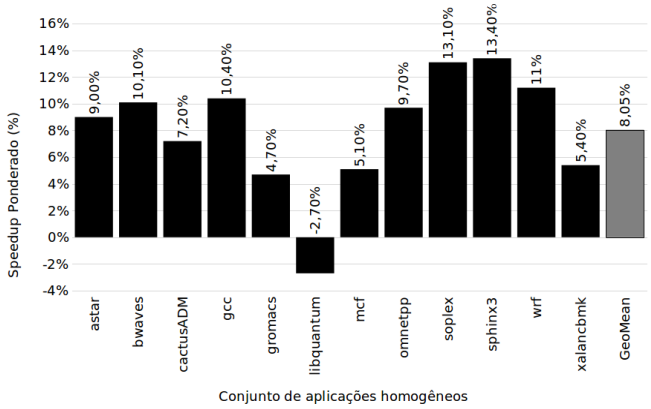


Figura 6: *Speedup* para grupos de aplicações homogêneos.

um *overhead* de armazenamento de 96 B ($256 \times 3b = 96\ B$) por núcleo de processamento.

Para avaliarmos a implementação do mecanismo, se faz necessário avaliar quantas requisições paralelas foram enviadas corretamente. A Figura 4 apresenta a precisão do mecanismo, quando utilizado nas aplicações que foram utilizadas para a criação dos conjuntos. O mecanismo apresentou uma taxa de acerto médio de cerca de 95% nas requisições enviadas paralelamente.

B. Avaliação do sistema Multi-core

A Figura 5 apresenta o ganho de desempenho para um sistema *multi-core*.O grupo de aplicações H0-H9, mostra um ganho de desempenho médio de 9,8%. Nos grupos de aplicações onde estão presentes as aplicações *soplex* e *sphinx3* são observados os maiores ganhos.

No grupo de aplicações M0-M4 é observado um ganho de desempenho médio de 9,5%, enquanto que o grupo de aplicações L0-L4 apresenta um aumento de desempenho de cerca de 9,2%. Dentre os grupos de aplicações que apresentaram melhor desempenho, em sua composição estão as aplicações *soplex* e *sphinx3*.

As aplicações (*soplex* e *sphinx3*) estão presentes nos grupos de aplicações que apresentam maior ganho de desempenho. Isto indica que estas aplicações apresentam um bom balanceamento entre as requisições que podem ser enviadas de forma direta para a memória principal, sendo previstas pelo mecanismo, e as requisições que são filtradas pela hierarquia de cache.

Para avaliarmos o desempenho do mecanismo em grupos de aplicações homogêneos, foi simulado um ambiente onde são executadas quatro cópias das aplicações. A Figura 6 mostra os ganhos obtidos. Em geral, o mecanismo apresenta um aumento de desempenho médio de 8,5%. A aplicação *sphinx3* mostra o maior ganho, com 13,3%, enquanto que a aplicação *libquantum* apresenta uma degradação de 2,7% no desempenho.

Avaliando as diferenças nos resultados obtidos quando comparada a utilização do mecanismo em ambientes *multi-core*, nota-se que os ganhos são constantes, porém baixos (abaixo de 10% na média). Isso se deve à diferença na competição pelos recursos (espaço na cache, largura de banda, latência de acesso a DRAM). Podemos inferir que conforme maior a latência acumulada dos múltiplos níveis de memória cache, maiores devem ser os ganhos. De forma inversa, quando a latência da cache representar apenas uma ínfima fração da latência da DRAM, então os ganhos obtidos devem ser reduzidos.

VI. Trabalhos Relacionados

A maioria dos trabalhos que abordam cache *bypass* utilizam caches não-inclusivas ou exclusivas [20]. Abordagens de cache *bypass* em caches inclusivas buscam soluções que não violem a inclusividade, como a não instalação dos dados nas memórias cache [21][22]. O uso de *bypass* de requisições, ou seja, a geração paralela de requisições, mostra-se pouco abordado.

Utilizando a criação paralela de requisições, temos o trabalho [23]. Apesar de não ser utilizado uma cache inclusiva, neste trabalho foi utilizado um preditor para decidir se deve ou não requisitar um dado diretamente a memória principal. Baseado nas tecnologias de *branch prediction*, o mecanismo permite que um dado seja requisitado antes de acessar a LLC,

poupando parte da latência necessária para acesso a DRAM. Apesar de possibilitar o mapeamento de até 1024 instruções independentes, é um trabalho mais antigo, e portanto não foi avaliado em um cenário onde as hierarquias de cache tornam-se mais profundas e complexas.

O trabalho [13] apresenta um mecanismo para a paralelização de acessos entre a DRAM-Cache e a memória DRAM principal. Utilizando uma abordagem mais prática que a apresentada em [24], é apresentado um mecanismo que, utilizando uma estimativa de *hits* e *misses* na DRAM-Cache, decide se ela deve ser acessada antes da memória principal, ou em paralelo com ela. Contudo, esta decisão só é realizada após a requisição percorrer toda a hierarquia de cache, o que acaba por consumir largura de banda e deixando o processador ocioso.

VII. Conclusões

Em nosso trabalho, apresentamos uma nova maneira de se executar requisições paralelas, de forma a reduzir a latência perceptível durante a busca de dados. Utilizando um preditor simples, porém eficiente, nossa proposta apresentou um ganho de desempenho de até 13.3% (média 9,25%) sendo que observamos perdas apenas quando executamos quatro instâncias da *libquantum*. No sistema *multi-core* usando grupos de aplicações mistas, conseguimos um ganho de até 12,7% (média 9,8%), sendo que as diferentes aplicações acabam por balancear o uso do barramento durante as requisições paralelas. Para o sistema *multi-core* executando grupos de aplicações homogêneas observamos ganhos de até 13,3% (média 8,5%). Quando analisado esses conjuntos homogêneos, notamos que nos casos das aplicações *sphinx3* e *soplex*, o mecanismo foi muito preciso, possibilitando a execução de requisições paralelas em quase que sua totalidade. Contudo, no caso da aplicação *libquantum*, o preditor foi extremamente preciso, e permitiu a execução de requisições paralelas em todas as requisições. Isso acarretou um congestionamento do barramento, gerando uma queda de 2,7% no desempenho do sistema.

Nesse artigo, ao invés de realizar o *bypass* de dados, foi emulado o *bypass* de requisições, realizando requisições em paralelo entre a memória principal e a hierarquia de cache. Porém, consideramos que o uso de técnicas de *bypass* de dados para caches inclusivas pode ser utilizada em conjunto com nosso mecanismo, potencializando os ganhos.

Agradecimentos

Este trabalho recebeu apoio do Instituto Serrapilheira (número do processo Serra-1709-16621) e da CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior). Os autores também agradecem o apoio do Instituto Federal Catarinense - Campus Videira (IFC-Videira).

Referências

[1] K. K. Chang, "Understanding and improving the latency of dram-based memory systems," Jul 2017.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., 2013.

[3] R. C. Murphy and P. M. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *IEEE Transactions on Computers*, vol. 56, no. 7, 2007.

[4] P. C. Santos, M. A. Z. Alves, M. Diener, L. Carro, and P. O. A. Navaux, "Exploring cache size and core count tradeoffs in systems with reduced memory access latency," in *Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*, 2016.

[5] M. Kharbutli, M. Jarrah, and Y. Jararweh, "Scip: Selective cache insertion and bypassing to improve the performance of last-level caches," in *Jordan Conf. on Applied Elec. Eng. and Comp.Techn.*, Dec 2013, pp. 1–6.

[6] W. Sritriratanarak, M. Ekpanyapong, and P. Chongstitvatana, "Applying svm to data bypass prediction in multi core last-level caches," *IEICE Electronics Express*, vol. 12, no. 22, pp. 20 150 736–20 150 736, 2015.

[7] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches," in *Int. Conf. on Par. Arch. and Comp. Tech.*, Sep. 2012, pp. 293–304.

[8] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016, pp. 117–124.

[9] S. Gupta and H. Zhou, "Spatial locality-aware cache partitioning for effective cache sharing," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 150–159.

[10] Y. Tian, S. Khan, and D. A. Jiménez, "Temporal-based multilevel correlating inclusive cache replacement," *Transactions on Architecture and Code Optimization*, vol. 10, pp. 1–24, 12 2013.

[11] I. Corporation, "Intel® 64 and ia-32 architectures software developers manual," Online, sep 2016.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[13] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Int. Symp. on Microarchitecture*, 2012.

[14] I. Corporation, "Intel® 64 and ia-32 architectures optimization reference manual," Online, jun 2016.

[15] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux, "Sinuca: A validated micro-architecture simulator," in *Int. Conf. on High Perf. Comp. and Comm.*, Aug 2015, pp. 605–610.

[16] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, 2006.

[17] H. Patil, R. Cohn, M. Charney, R. Kapoor, and A. Sun, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *Int. Symp. on Microarchitecture*, 2004.

[18] D. A. Jimenez, "Piecewise linear branch prediction," in *Int. Symp. on Com. Arch.*, June 2005, pp. 382–393.

[19] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Int. Conf. on Architectural Support for Prog. Lang. and Op. Sys.*, 2000.

[20] S. Mittal, "A survey of cache bypassing techniques," *Journal of Low Power Electronics and Applications*, vol. 6, 2016.

[21] M. K. Kim, J. H. Choi, J. W. Kwak, S. T. Jhang, and C. S. Jhon, "Bypassing method for stt-ram based inclusive last-level cache," in *Conf. on Research in Adaptive and Convergent Systems*, ser. RACS, 2015.

[22] S. Gupta, H. Gao, and H. Zhou, "Adaptive cache bypassing for inclusive last level caches," in *Int. Symp. on Parallel and Distributed Processing*, 2013.

[23] K. Malkowski, G. Link, P. Raghavan, and M. J. Irwin, "Load miss prediction - exploiting power performance trade-offs," in *Int. Parallel and Distributed Processing Symp.*, 2007.

[24] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Int. Symp. on Microarchitecture*, 2011.