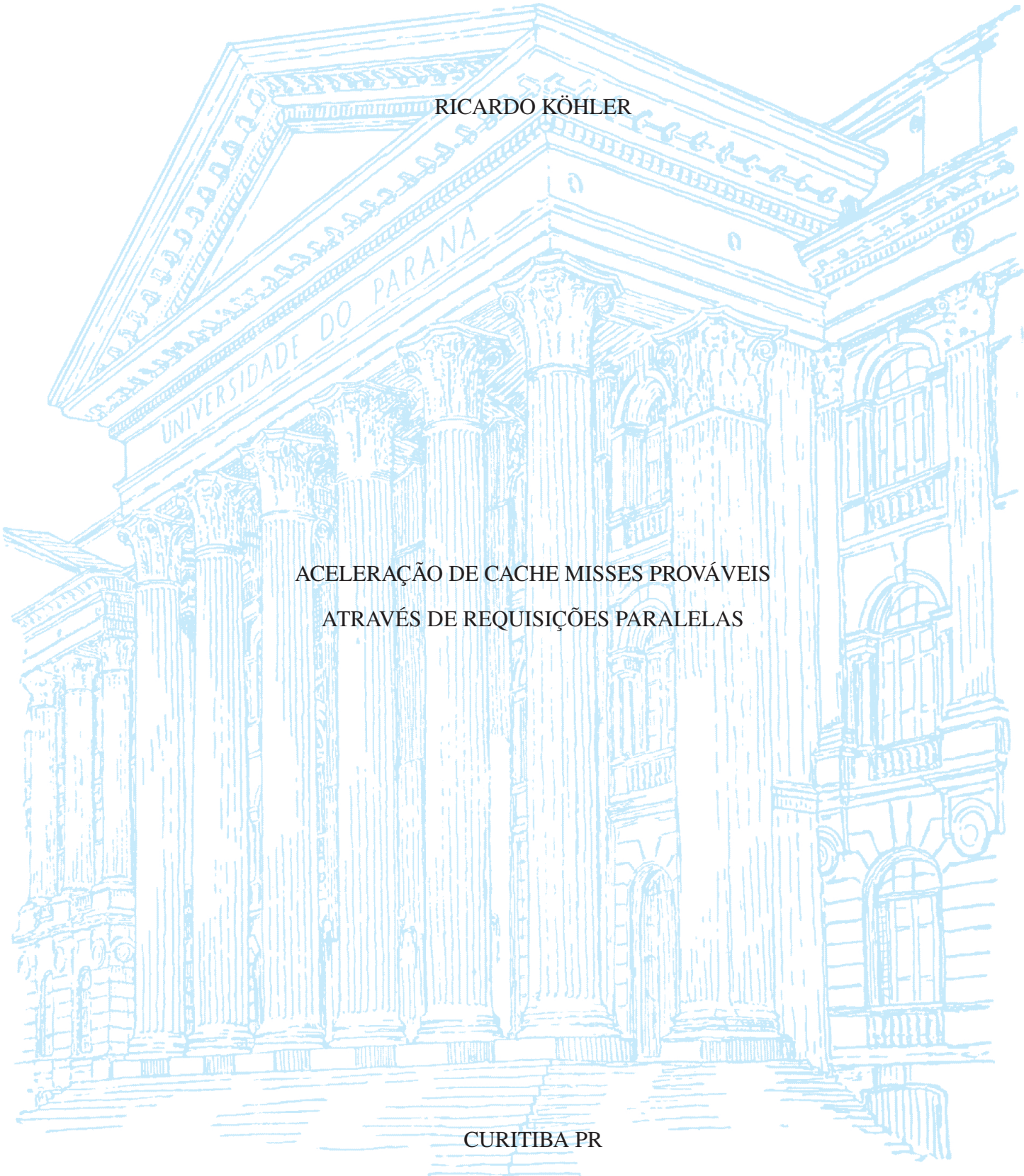
UNIVERSIDADE FEDERAL DO PARANÁ

RICARDO KÖHLER



ACELERAÇÃO DE CACHE MISSES PROVÁVEIS ATRAVÉS DE REQUISIÇÕES PARALELAS

CURITIBA PR

2019

RICARDO KÖHLER

ACELERAÇÃO DE CACHE MISSES PROVÁVEIS ATRAVÉS DE REQUISIÇÕES PARALELAS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Marco Antonio Zanata Alves.

CURITIBA PR

2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

K79a köhler, Ricardo
Aceleração de cache misses prováveis através de requisições paralelas [recurso eletrônico] / Ricardo köhler. – Curitiba, 2019.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós- Graduação em Informática, 2019.

Orientador: Marco Antonio Zanata Alves.

1. Sistemas de memória de computadores. 2. Processamento paralelo (Computadores). 3. Capacidade do computador. 4. Gerenciamento de memória (Computação). I. Universidade Federal do Paraná. II. Alves, Marco Antonio Zanata. III. Título.

CDD: 005.435

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RICARDO KÖHLER** intitulada: **ACELERAÇÃO DE CACHE MISSES PROVÁVEIS ATRAVÉS DE REQUISIÇÕES PARALELAS**, sob orientação do Prof. Dr. MARCO ANTONIO ZANATA ALVES, que após após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Setembro de 2019.

MARCO ANTONIO ZANATA ALVES
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

ANTONIO CARLOS SCHNEIDER BECK FILHO
Avaliador Externo (UNIVER. FEDERAL DO RIO GRANDE DO SUL)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

A todos que contribuíram nesta caminhada.

AGRADECIMENTOS

Primeiramente agradeço a Deus, pois sem ele nada disso seria possível. Agradeço a minha família, por todo o suporte durante esta longa caminhada, em especial a minha esposa Deise, pelo incentivo para cursar o mestrado e pelo seu apoio incondicional durante esta jornada, mesmo durante os momentos mais sombrios. Aos meus pais, João e Inês por sempre me incentivarem a estudar e nunca esperar menos do que posso oferecer e aos meus sogros José e Ana pelo apoio logístico e orações pelo melhor.

Agradeço ao meu orientador, Marco Antonio Zanata Alves, pela paciência e entusiasmo demonstrados, mesmo diante dos assuntos mais banais. Agradeço aos professores Luís Carlos Erpen de Bona e Antônio Carlos Schneider Beck Filho por fazer parte da comissão examinadora, com comentários e sugestões precisos e extremamente valiosos.

Aos colegas e amigos do departamento de informática (DINF), especialmente ao pessoal do grupo HIPES e ao pessoal que faz parte do labirinto dos doutorandos, onde a boa companhia e as pausas para o café tornaram mais alegres até mesmo os momentos mais difíceis desta jornada ficam registrados meus sinceros agradecimentos.

Por fim, gostaria de agradecer ao Instituto Federal Catarinense, Câmpus Videira (IFC - Videira), do qual me afastei integralmente durante o período de curso do mestrado. Sem o afastamento seria muito mais difícil a conclusão desta jornada.

RESUMO

A diferença na velocidade de evolução do processador quando comparado a memória principal ocasionou uma disparidade entre a velocidade que os dados são processados e a velocidade com a qual os dados podem ser servidos. Assim, se faz necessário desenvolver meios para que tais dados possam ser servidos mais rapidamente. Uma destas formas é a utilização de memórias intermediárias, mais rápidas e localizadas próximas ao processador, as memórias cache. Tais memórias, com o passar dos anos acabaram por ser organizadas em hierarquias, de forma a aumentar sua eficácia. Entretanto, tais memórias, em determinados cenários podem se revelar motivo de atraso para a transmissão dos dados. Nestes cenários, o uso de meios para contornar tais dispositivos (Cache *bypass*) podem gerar ganhos de desempenho consideráveis. Contudo, dada a diversidade de opções na hora de projeto para uma cache, os mecanismos de *bypass* são projetados de forma a obter o melhor desempenho ao serem utilizados em uma arquitetura e hierarquia específicas. Caches inclusivas são menos visadas dado que a propriedade de inclusão acaba por entrar em conflito com as políticas de inserção seletiva, que tornam o cache *bypass* eficiente. Desta forma, ao estudar o trabalho EMC, notou-se que por executar as operações no controlador de memória, quando necessitava de um dado, consultava a LLC. Por ser inclusiva, tem-se a garantia que caso o dado não esteja neste nível, não encontra-se no chip, e assim, os dados são requisitados diretamente a memória principal. Com mecanismos para prever quando os dados não se encontram no chip a partir do controlador de memória, isso permite que o núcleo requisite os dados diretamente a memória principal, contornando a hierarquia de cache. Quando habilitada esta possibilidade, isto permite que o desempenho do sistema aumente em até 40% para aplicações *single-core* e até 16% para conjunto de aplicações *multi-core*, enquanto apresenta uma redução no consumo de energia de até 28% para aplicações *single-core* e até 13% para conjuntos de aplicações *multi-core* das aplicações que compõe o SPEC CPU-2006 . Resultados de simulação mostram que utilizando um mecanismo de predição simples, extraímos em média 87,9% de todo o desempenho possível de ser alcançado ao contornar a cache durante buscas de dados.

Palavras-chave: Cache bypass. Caches inclusivas. Predição de cache misses

ABSTRACT

The difference in processor speed when compared to main memory has caused a disparity between the speed at which data is processed and the speed at which data can be served. Thus, it is necessary to develop a means for such data to be served faster. One such way is the use of faster, intermediate buffers located near the processor, the cache memories. These memories, over the years, were eventually organized into hierarchies to increase their effectiveness. However, such memories in certain scenarios may prove to be a delay for data transmission. In these scenarios, using means of bypassing such devices (Cache bypass) can yield considerable performance gains. However, given the diversity of design-time options for a cache, bypass is designed to achieve the best performance when used in a specific architecture and hierarchy. Inclusive caches are less targeted because the include property eventually conflicts with selective insert policies, which make cache bypass efficient. Thus, when studying EMC work, it was noted that when performing the operations on the memory controller, when in need of data, consulted the LLC. Because it is inclusive, it is guaranteed that if the data is not at this level, it is not on the chip, and thus the data is requested directly from the main memory. With mechanisms to predict when data is not on the chip from the memory controller, this allows the core to request data directly from the main memory, bypassing the cache hierarchy. When enabled, this allows system performance to increase by up to 40% for single-core applications and up to 16% for multi-core applications, while reducing power consumption by up to 28% for single-core and up to 13% for multi-core application sets of SPEC CPU-2006 applications. Our simulation results show that using a simple prediction engine, we could harvest on average 87.9% of all performance achievable by parallel data requests technique during data searches.

Keywords: Caching bypass. Inclusive caches. Cache misses prediction

LISTA DE FIGURAS

2.1 Hierarquias de cache quanto a política de inclusividade. 18

3.1 Sequência de instruções para geração de endereço [17]. 23
3.2 Visão de alto nível de um processador multi-core equipado com o EMC. 23

4.1 Arquitetura do Simulador OrCS. 26
4.2 Gráfico de IPC por *microbenchmark* das categorias *control, dependency* e *execution*. 28
4.3 Gráfico de IPC por *microbenchmark* das categorias de load dependente, load independente e store independente. 29
4.4 Erros relativos, mínimo, máximo e médio entre os simuladores SiNUCA e OrCS comparados com uma máquina real. 29

5.1 Quantidade de operações entre *load* misses dependentes apresentado em [17] . . 33
5.2 Quantidade de operações entre *load* misses dependentes verificado na replicação do trabalho [17] 33
5.3 Incremento no IPC relativo ao *baseline* obtido na replicação do EMC utilizando o mecanismo implementado como descrito em [17] e utilizando um oráculo. 35
5.4 Melhoria no WS dos conjuntos de aplicações utilizados no EMC [17] 36
5.5 Melhoria no WS dos conjuntos de aplicações homogêneos utilizados no EMC [17] 36
5.6 Latência média sofrida pelas requisições a memória quando feitas pelo core e pelo EMC nos conjuntos de aplicações de alta intensidade de uso da memória . . 37
5.7 Latência média sofrida pelas requisições a memória quando feitas pelo core e pelo EMC nos conjuntos de aplicações homogêneos 37
5.8 Incremento no WS dos conjuntos de aplicações gerados entre as aplicações que apresentaram melhor desempenho no *single-core* 38
5.9 Incremento no WS dos conjuntos de aplicações homogêneos das aplicações que apresentaram melhor desempenho no *single-core* 39
5.10 Latência média das requisições feitas a memória nos conjuntos de aplicações gerados com base nas aplicações que apresentaram melhor desempenho. 39
5.11 Latência média das requisições feitas a memória pelos conjuntos de aplicações homogêneas. 39
5.12 Percentagem de loads enviados ao EMC quando utilizando o oráculo. 40
5.13 Precisão do preditor de misses da LLC no EMC 40

6.1 Requisições paralelas cache/DRAM com instalação tradicional de dados na hierarquia de cache. 42
6.2 Fluxo de requisições para memória principal. 43

6.3 Fluxo de requisições com o mecanismo responsável por executar as requisições paralelas. 44
6.4 Melhoria de IPC para as aplicações SPEC CPU-2006 utilizando requisições paralelas. 45
6.5 Melhoria de IPC para as aplicações SPEC CPU-2006 utilizando requisições paralelas com um oráculo. 46
6.6 Variação de uso de energia no sistema ao utilizar requisições paralelas 46
6.7 Variação de uso de energia no sistema ao utilizar requisições paralelas em um ambiente *single-core* 47
6.8 *Speedup* para *workloads* aleatórios apresentados no trabalho EMC [17]. 48
6.9 *Speedup* para os *workloads* homogêneos apresentados no trabalho EMC [17] . . 48
6.10 Incremento do WS para conjuntos de aplicações com alta intensidade de memória 49
6.11 Incremento do WS para conjuntos de aplicações com média intensidade de memória. 49
6.12 Incremento do WS para conjuntos de aplicações com baixa intensidade de memória50
6.13 Melhoria no WS para conjuntos de aplicações homogêneos 50
6.14 Precisão do preditor de misses na LLC e execução de requisições paralelas em cache misses. 51
6.15 Variação no consumo de energia para os conjuntos de aplicações randômicas apresentados em [17] 52
6.16 Variação no consumo de energia para os conjuntos de aplicações homogêneas apresentados em [17] 52
6.17 Variação no consumo de energia para os conjuntos de aplicações randômicos de alta intensidade de memória geradas para o mecanismo de requisições paralelas . 53
6.18 Variação no consumo de energia para os conjuntos de aplicações randômicos de média e baixa intensidade de memória geradas para o mecanismo de requisições paralelas. 53
6.19 Variação no consumo de energia para os conjuntos de aplicações homogêneos geradas para o mecanismo de requisições paralelas 54
6.20 Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações homogêneos apresentados em [17] 55
6.21 Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias apresentados em [17] 55
6.22 Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias de alta intensidade. 56
6.23 Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias de média e baixa intensidade 56
6.24 Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações homogêneas 57

LISTA DE TABELAS

3.1 Instruções suportadas pelo EMC. 23

5.1 Configuração do sistema 32
5.2 Classificação das aplicações do SPEC CPU-2006 com base na intensidade de uso da Memória 34
5.3 Conjunto de aplicações apresentados em [17] 34
5.4 Lista de conjuntos de aplicações com base na classificação de MPKI da tabela 5.2 B35
5.5 Resultados *single-core* utilizando o mecanismo EMC proposto. 35
5.6 Conjuntos de aplicações gerados a partir das aplicações com melhor desempenho no EMC *single-core* 38

LISTA DE ACRÔNIMOS

BB	Bypass Buffer
BDB	Bypassing Decision Block
BTB	Branch Target Buffer
BTC	Bypass Tag Cache
CAS	Column Address Strobe
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DDR3	Double Data Rate ver.3
DRAM	Dynamic Random Access Memory
EMC	Enhanced Memory Controller
FCFS	First Come First Served
GPU	Graphical Processing Unit
ISA	Instruction Set Architecture
IPC	Istrução por Ciclo
I/O	Input/Output
KB	KiloByte
LLC	Last Level Cache
LMP	Load Miss Prediction
LRU	Least Recently User
MACT	Memory Access Counter Table
MB	MegaByte
MOB	Memory Order Buffer
MPKI	Misses Per Kilo Instruction
OoO	Out-of-Order
OrCS	Ordinary Computer Simulator
PLBP	Piecewise Linear Branch Predictor
RAS	Row Address Strobe
RP	Row Precharge
ROB	Reorder Buffer
RS	Reservation Station
S/DC	Stride/Delta-Correlation
SiNUCA	Simulator of NonUniform Cache Architectures
SPEC	Standard Performance Evaluation Corporation
STT-RAM	Spin Torque Transfer Random Access Memory
WS	Weight speedup

SUMÁRIO

1 INTRODUÇÃO 13
1.1 PROBLEMA 13
1.2 MOTIVAÇÃO 14
1.3 PERGUNTA DE PESQUISA 14
1.4 CONTRIBUIÇÕES 14
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 ARQUITETURA DA MEMÓRIA CACHE 16
2.2 CACHE BYPASS 19
3 TRABALHOS CORRELATOS. 21
3.1 PESQUISAS EM TÉCNICAS DE CACHE *BYPASS* DE DADOS 21
3.2 PESQUISAS QUE APRESENTAM TÉCNICAS DE CACHE *BYPASS* DE REQUISIÇÕES 22
3.2.1 Load Miss Prediction (LMP) e *Bypass* 22
3.2.2 Enhanced Memory Controller - EMC 22
3.3 CONCLUSÕES 23
4 SIMULADOR 25
4.1 *ORDINARY COMPUTER SIMULATOR - ORCS* 25
4.1.1 Componentes modelados 25
4.1.2 Validação 27
4.2 CONCLUSÕES 30
5 REPLICAÇÃO DO MECANISMO EMC 31
5.1 METODOLOGIA DE AVALIAÇÃO 31
5.2 CONFIGURAÇÃO DO AMBIENTE DE SIMULAÇÃO 32
5.3 AVALIAÇÃO EXPERIMENTAL. 33
5.4 CONCLUSÕES 40
6 REQUISIÇÕES PARALELAS CACHE/DRAM 42
6.1 REQUISIÇÕES PARALELAS 42
6.2 ARQUITETURA DO MECANISMO 43
6.3 SETUP 45
6.4 AVALIAÇÃO DO MECANISMO EM AMBIENTE SINGLE-CORE 45
6.4.1 Considerações sobre o consumo de energia 46
6.4.2 Latência média das requisições em *single-core* 47

6.5 AVALIAÇÃO DO MECANISMOS EM AMBIENTE MULTI-CORE. 47
6.5.1 Avaliação dos conjuntos de aplicação do EMC. 47
6.5.2 Avaliação dos conjuntos de aplicação para mecanismo requisições paralelas . . . 48
6.5.3 Avaliação do consumo de energia. 51
6.5.4 Latência média das requisições em *multi-core* 54
7 CONCLUSÕES E TRABALHOS FUTUROS 58
REFERÊNCIAS 59

1 INTRODUÇÃO

Historicamente, o foco do desenvolvimento sobre arquitetura de computadores tem-se concentrado no processador, muito em função do modelo de processamento empregado. O modelo de John von Neumann, proposto em 1945 [39], postula que as unidades responsáveis pelos dados e pelo processamento encontram-se separadas, assim, as instruções e os dados devem ser movidos da memória para o processador, para que sejam realizadas operações sobre eles. Embora esse modelo tenha possibilitado processamento de propósito geral, em tempos modernos isto acaba por se tornar uma grande limitação, tanto em termos computacionais, quanto em termos de energia gasta para realizar estas movimentações. Essas limitações se devem ao fato da arquitetura ter sido desenvolvida no modelo *processing-centric*, enquanto as aplicações atuais requerem um modelo novo baseado em *data-centric*.

Pesquisas recentes buscam mover a computação para mais perto da memória [2] [1] [4] [21]. Evitando a movimentação de grandes volumes de dados. Apesar disto, as aplicações necessitam de alterações para que possam explorar o total potencial dessas inovações. Entretanto, nem todas as aplicações podem se beneficiar destas mudanças. Desta forma, métodos que forneçam os dados antes que o processador os requisite, ou que permitam que os dados sejam fornecidos mais rapidamente quando solicitados ainda são importantes.

1.1 PROBLEMA

Embora o poder computacional dos processadores tenha crescido de forma exponencial, o tempo de acesso a memória principal manteve-se praticamente intocado. Ou seja, nos últimos 20 anos, houve uma redução de apenas 30% na latência de acesso *per bit* das memórias Dynamic Random Access Memory (DRAM) [5]. Isso se deve a diferenças de construção e tecnológicas entre os as memórias DRAM e os processadores baseados em transistores. Enquanto a tarefa do processador é executar as instruções o mais rapidamente possível, a memória deve armazenar os dados e tê-los sempre prontos quando o processador os requisitar. Esta crescente divergência entre as velocidades destes componentes passou a ser chamada de *memory wall* [35].

A latência total para um dado requisitado a memória não é limitado apenas pelo tempo de acesso do dado na memória DRAM, mas também pela latência que a interconexão e a hierarquia de memória cache impõem impõe até que a requisição seja feita e os dados retornem. Isto é mais pronunciado conforme as hierarquias de cache *on-chip* acabam por se tornarem mais profundas (e portanto algumas vezes mais lenta) e mais profundas (criando múltiplos níveis de latência sequencial). Aplicações cujo perfil de uso de memória não podem ser totalmente armazenados na hierarquia de memória cache e/ou aplicações que apresentam baixo reúso ou baixa localidade de dados (dados não coalescentes), inverso das condições nas quais as memórias cache tem seu potencial máximo explorado, sofrem mais com a latência acumulada. Além disso, aplicações com com forte dependência de dados, utilizam menos a alta vazão das memórias e sofre mais com as latências por acesso.

Isto é amplificado quando se pensa no cenário atual, com o crescimento do número dos núcleos de processamento nos processadores e o aumento dos conjuntos de dados das aplicações (ex. aplicações big data). Tudo isso leva ao aumento da competição pelos recursos finitos acabando por elevar a latência das requisições de dados. Assim, tornam-se cruciais métodos para evitar a latência acumulada quando se acessa a memória principal.

1.2 MOTIVAÇÃO

Operações de acesso de leitura a memória (*loads*) são aproximadamente 25% das instruções de um programa típico [14]. Para evitar que cada instrução de *load* tenha que enfrentar a latência da memória principal (DRAM), existe uma hierarquia de memórias cache (SRAM) intermediárias que buscam manter os dados mais frequentemente acessados próximos ao processador. Entretanto, em determinadas situações, a cache pode acabar por ser danosa ao desempenho do programa. Nestas situações, driblar a cache pode resultar em ganhos de desempenho e/ou economia de energia [44].

Abordagens que movem a computação para mais próximo dos dados são cada vez mais interessantes, tanto para a economia de energia quanto para o ganho de desempenho, explorando menores movimentações de dados/paralelismo inerente da memória. Porém essas abordagens requerem que as aplicações sejam preparadas para tal mudança, devendo ser codificados novamente, ou recompilados [8] [15]. Entretanto, isto nem sempre é possível. Desta forma técnicas transparentes a aplicação tais como *bypass* de dados mostram-se eficazes quando se detecta que a cache não será útil, pois o dado será utilizado uma única vez. Entretanto estes mecanismos geralmente são projetados utilizando caches que não estão em consonância com os rumos da indústria, necessitando também de mudanças no protocolo de coerência dos dados, uma vez que esses mecanismos requerem o uso de cache não-inclusivas [10].

1.3 PERGUNTA DE PESQUISA

A principal pergunta deste trabalho é: "Como podemos reduzir o tempo necessário para que uma requisição, cuja execução fatalmente se encaminhará a memória principal, tenha seu tempo reduzido?". Assim, analisamos trabalhos que visam encurtar, ou tem por consequência a redução do tempo necessário para uma requisição de dados percorra o caminho entre processador-memória (*round-trip*).

Para buscar a resposta de nossa pergunta de pesquisa, iniciamos replicando um trabalho que visa migrar cadeias de instruções para serem executadas mais próximas a memória, evitando a latência da cache. Em seguida, buscamos aprender se é possível estender este trabalho que execute este encurtamento no *round-trip*. Conseguinte, ao analisar os resultados obtidos na replicação do trabalho correlato, gerou uma nova pergunta: "É possível encurtar o *round-trip* de requisições de maneira mais simples, sem a necessidade de utilizar abordagens complexas?"

1.4 CONTRIBUIÇÕES

Como forma de estabelecer uma base para comparações (*baseline*), foi desenvolvido um simulador arquitetural simplificado, balizado pelas definições das microarquiteturas mais recentes da Intel. Terminado o desenvolvimento do simulador, foi realizado a replicação do trabalho Enhanced Memory Controller (EMC) [17]. Quando replicado o mecanismo EMC, foi atingida uma melhoria de IPC de no máximo 0,5% sobre o *baseline*, durante a avaliação do cenário *multi-core*, quando a demanda do sistema de memória pelas aplicações tende a ser mais elevada. Ao analisar os motivos que levaram a tal resultado, observou-se que os excertos das aplicações do SPEC CPU-2006 utilizados apresentavam falta de condições para que o EMC atingir o potencial relatado. Quando utilizada uma aplicação sintética desenvolvida especificamente para avaliar o EMC, esta aplicação apresentou um *speedup* de 1,61× em nossos experimentos.

Ao analisar o comportamento das requisições feitas pelo EMC, notou-se uma desempenho satisfatório no preditor de Last-Level Cache (LLC) misses. Assim, vislumbrou-se a possibilidade

de utilizar este preditor de forma a encurtar o *round-trip* das requisições, permitindo que as requisições sejam enviadas diretamente para a memória principal, de forma paralela com as requisições que são enviadas para a hierarquia de cache. Utilizando esta abordagem, obteve-se uma melhoria média no desempenho de 9,8%, 9,5% e 9,2% em conjuntos de aplicações que apresentam alta, média e baixa intensidade de memória, respectivamente, quando comparados ao *baseline*. Para conjuntos de aplicações homogêneos, a melhoria média no desempenho fica em 8,5%. Quando utilizado em um ambiente *single-core*, a melhoria média foi de 3,18% entre todas as aplicações do SPEC CPU-2006.

Posteriormente foi avaliado o consumo de energia total do sistema, quando utilizado o mecanismo de requisições paralelas. Para o cenário *single-core* houve uma redução na média do consumo de energia de 8,2%. Para o cenário *multi-core*, o a redução média no consumo foram de 9,9%, 10,3% e 13% para os conjuntos de aplicação de alta, média e baixa intensidade de memória, respectivamente.

Desta forma, nesta dissertação são feitas as seguintes contribuições:

1. Foi desenvolvido um simulador simplificado, que apresenta erro médio 3% maior quando comparado ao Simulator of Non-Uniform Cache Architectures (SiNUCA) [3], enquanto reduz sua complexidade construtiva.

2. Foi replicado o mecanismo EMC [17], provando que é um mecanismo funcional, porém de pouco desempenho para aplicações genéricas.

3. Foi projetado um mecanismo para a realização de requisições em paralelo Cache/DRAM, quando estas requisições são prováveis misses em uma LLC inclusiva.

4. Foi avaliado o consumo de energia ao ser utilizado o mecanismo de para execução de requisições em paralelo Cache/DRAM em ambientes *single-core* e *multi-core*.

O resto dessa dissertação está organizada da seguinte forma.

No capítulo 2 apresentamos conceitos sobre a arquitetura de memórias cache, tradução binária dinâmica e sobre o uso de requisições paralelas. No capítulo 3 são apresentados trabalhos que exploram técnicas de cache *bypass*, tanto de dados quanto de requisições. No capítulo 4 é apresentado o simulador arquitetural desenvolvido no decorrer do trabalho. No capítulo 5 apresentamos a replicação do mecanismo apresentado no artigo [17]. No capítulo 6 é apresentado o mecanismo proposto para a realização de requisições paralelas, bem como o resultado das simulações para validação do mecanismo. No capítulo 7 apresentamos nossas conclusões e direcionamento de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são introduzidos os conceitos utilizados ao longo desta dissertação. A seção 2.1 apresenta conceitos sobre a arquitetura e organização para a implementação de uma hierarquia de memória cache, bem como sua relação com um Chip Multiprocessor (CMP). A seção 2.2, apresentamos uma visão geral sobre cache *bypass* e como pode ser utilizada no contexto de CMP.

2.1 ARQUITETURA DA MEMÓRIA CACHE

Dada a diferença entre o aumento da velocidade de processamento e a velocidade com a qual os dados são acessados, são necessários meios para suprir os dados ao processador com uma baixa latência. Um dos métodos para garantir isso é o uso de memórias pequenas e muito rápidas, quando comparadas com a DRAM, denominadas de cache. Uma memória cache é projetada principalmente para tomar vantagem da localidade temporal e localidade espacial dos acessos a memória [42][47].

A palavra cache, para definir uma memória próxima ao processador, apareceu nas pesquisas inicialmente nos anos 1960. Atualmente, todos os dispositivos de propósito geral, desde servidores até processadores embarcados utilizam caches [42]. Estruturalmente, uma cache pode ser comparada a uma tabela, onde as linhas podem ser denominadas de conjuntos (*sets*) e as colunas podem ser denominadas de vias (*ways*). Sendo que cada célula da tabela é chamada de linha (ou bloco) de memória cache, formada por algumas poucas dezenas de bytes (ex. 64 B), a menor unidade de troca entre memória cache e memória DRAM. Assim, para o gerenciamento de uma memória cache, diversos aspectos devem ser levados em conta. Para se endereçar dados na tabela de armazenamento (memória cache), utiliza-se parte dos bits de endereço para indicar em qual linha da tabela, o dado poderá estar armazenado. Uma vez endereçada a linha da tabela (conjunto associativo), o restante do endereço deverá ser comparado com o endereço armazenado em cada coluna da tabela (vias do conjunto). Sobre as funções da cache, pode-se dividir em três grupos principais: alocação, substituição e escrita de dados.

O processo de alocação define como os dados serão organizados na cache. Este processo pode afetar de forma significativa o desempenho. A política de alocação está intimamente ligada a organização da cache. Desta forma, utilizando a analogia da tabela, tem-se caches com uma organização de mapeamento direto, com uma única via e com c conjuntos (tabela com c linhas e 1 coluna). Em caches diretamente mapeadas, cada bloco vindo da memória principal é mapeada para um único bloco da cache. Isto pode facilmente gerar conflitos entre diversos blocos distintos da memória principal, quando mapeados para o mesmo bloco da cache, causando uma degradação de performance.

Por outro lado, existem caches com mapeamento totalmente associativo. Neste tipo de cache, um único conjunto, composto de v vias (tabela de 1 linha e v colunas). Um bloco vindo da memória principal pode ser alocado em qualquer bloco da cache. Desta forma, ao requisitar um bloco da cache, se faz necessário olhar para todas as vias da cache, verificando se o bloco necessário está presente, o que acaba por se tornar impraticável, devido ao custo.

Por fim, temos a abordagem de conjuntos associativos que utiliza os dois elementos de forma balanceada. Contando com um número v de vias, organizados em c conjuntos (tabela de c linhas e v colunas), um bloco é mapeado para um único conjunto, entretanto, dentro do conjunto, pode ser alocado em qualquer via. Esta abordagem gerencia de forma melhor a ocorrência de

conflitos, e é mais simples que implementar uma cache totalmente associativa. Este é o *design* mais comum para a organização da memória cache [51] [10].

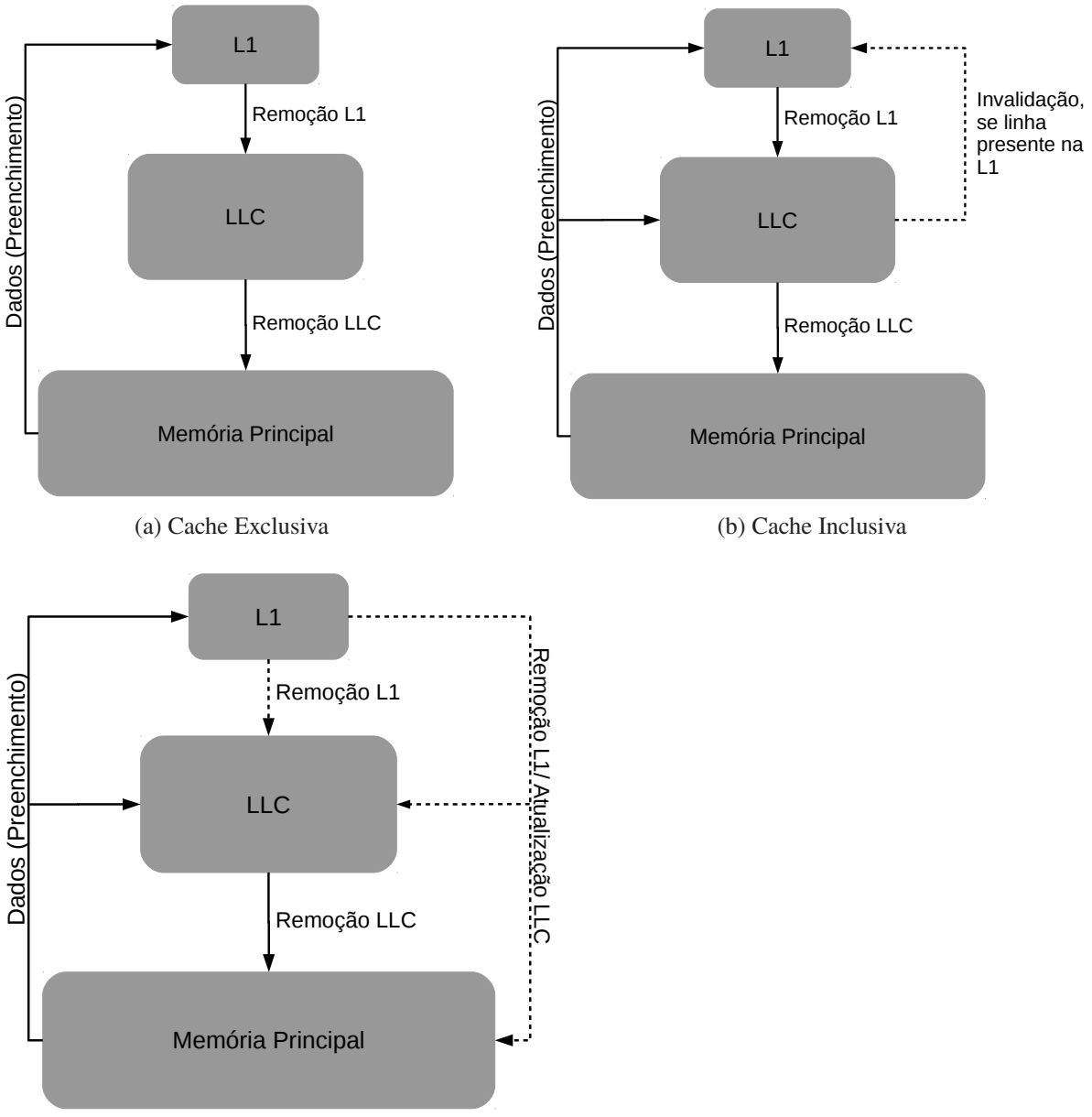
Outro aspecto do gerenciamento de uma cache é sobre as políticas de substituição. Considerando que temos uma cache com um conjunto associativo, e que ao executar uma aplicação qualquer, eventualmente esta cache irá ser totalmente preenchida. Na ocorrência de uma inserção de um novo bloco, um bloco antigo deve ser descartado, liberando espaço para o novo bloco. Logo, o bloco com a menor probabilidade de ser novamente utilizado deve ser selecionado como vitima para remoção [42]. Isto vai de encontro com um outro aspecto a ser considerado no gerenciamento de uma cache, a política de escrita. Pode-se dividir a política de escrita em duas abordagens, *write-through*, quando qualquer mudança nos dados é propagado para os níveis mais abaixo, e *write-back*, quando estas mudanças são propagadas somente quando o bloco é movido para o componente hierárquico mais abaixo [51].

Com o intuito de reduzir a diferença entre o processador e a memória, os processadores modernos suportam variados números de níveis de cache. Adicionar mais níveis de cache aumenta as chances de um dado estar presente mais próximo ao processador, entretanto, caso não esteja, a penalidade para acesso ao dado é incrementada [42]. Com a popularização dos processadores dotados de múltiplos núcleos, o uso de hierarquias de memórias caches se intensificou. Desta forma, os dados podem ser acessado não apenas pelo processador que os requisitou, mas também pelos outros processadores. Assim, o uso de caches compartilhadas tornou-se uma solução amplamente utilizada, com caches pequenas, privadas a cada núcleo, e um último nível de cache (mais próximo a DRAM) compartilhado entre todos os núcleos [51] [10] [13].

Sendo que múltiplos núcleos de processamento podem acessar um mesmo dado, se faz necessário manter a coerência dos dados entre os diversos núcleos, de forma que todos "enxerguem" o dado com um mesmo valor, evitando que hajam valores diferentes para um mesmo endereço de memória. Tal garantia é feita através dos protocolos de coerência de cache, que visam garantir que a ordenação dos acessos a memória seja sequencial com a ordem de instruções do programa, do ponto de vista da memória, enquanto que os todos os núcleos de processamento sempre recebam o dado mais atual disponível quando este for solicitado, e ao mesmo tempo em que este dado seja o mais atual, seja também o dado correto [42] [19].

Para além da coerência dos dados, deve-se também observar a consistência entre os diversos níveis presentes na hierarquia de cache. Por razões de consistência, pode-se lançar mão do uso de políticas de inclusão. Baseados nos princípios de inclusão e exclusão, definem-se relações entre os níveis de cache do sistema. Estas relações podem ser inclusivas, onde cada nível é um subconjunto do nível imediatamente inferior, exclusivo, onde os dados pertencem exclusivamente a um nível, ou ainda um híbrido entre os dois (não-inclusivo) [23]. Para exemplificar tais relações, tem-se a figura 2.1, que apresenta uma visão de alto nível de tais relações, em uma cache composta de dois níveis.

Em processadores com hierarquia de cache multi nível, diferentes estratégias podem ser adotadas. Conforme representado na Figura 2.1a, ao utilizar uma cache exclusiva, os dados estarão presentes em apenas um nível de cache. Os dados são inicialmente inseridos no nível L1, e quando substituídos, são movidos para a Last-Level Cache (LLC). Esta abordagem permite o uso de todo o espaço de armazenamento da hierarquia com dados úteis (sem replicação de dados). Na Figura 2.1b tem-se a representação de uma hierarquia inclusiva. Os dados são inseridos inicialmente em todos os níveis de cache, quando o dado é substituído no nível de cache L1, o mesmo é encaminhado para o nível inferior, neste caso a LLC. Ao ser removido da LLC, caso exista uma referência ao dado em níveis superiores, é realizada a invalidação do dado, isto é, ele é removido dos níveis superiores. Esta invalidação é realizada através do protocolo de coerência. Ao observar a Figura 2.1c, tem-se a representação de uma hierarquia de cache não-inclusiva. Os



(a) Cache Exclusiva

(b) Cache Inclusiva

(c) Cache Não Inclusiva

Figura 2.1: Hierarquias de cache quanto a política de inclusividade.

dados são inicialmente inseridos em todos os níveis, porém, quando realizada uma remoção em qualquer um dos níveis, não é implícita a remoção em outros níveis, como no caso de uma cache inclusiva. Ao se remover um dado, por exemplo do nível L1, pode-se atualizar os dados presentes nos níveis inferiores, ou removê-los. Tal decisão fica a critério do protocolo de coerência [48].

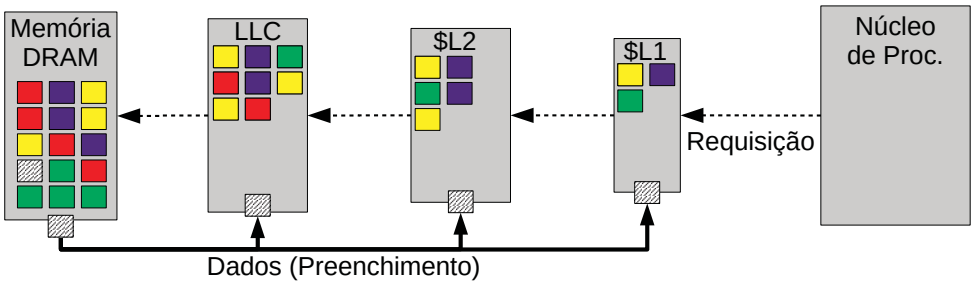
Diferentes políticas têm diferentes impactos na hierarquia de cache. O uso de políticas inclusivas acarreta em desperdício de espaço, uma vez que os níveis mais elevados são obrigatoriamente um subconjunto dos níveis mais baixos, entretanto, isto facilita a tarefa de manter a coerência dos dados entre os núcleos de processamento. Já a utilização de políticas exclusiva ou não-inclusivas acaba por otimizar a utilização do espaço de armazenamento, entretanto, manter a coerência dos dados entre os diversos núcleos de processamento torna-se uma tarefa mais complexa.

2.2 CACHE BYPASS

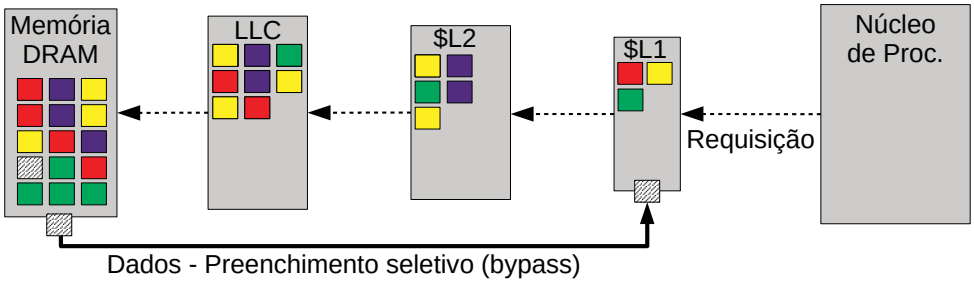
Dentre as abordagens criadas para ocultar a latência de acesso a memória Dynamic Random Access Memory (DRAM), o advento das memórias cache representa a possibilidade de acessar os dados sem que seja necessário enfrentar as altas latências. Entretanto, mesmo as memórias cache não são capazes de ocultar completamente a latência de todos os acessos a memória principal. Em determinadas situações, não é possível evitar o acesso a memória principal. As principais fontes de faltas de dados em cache são, as faltas compulsórias (durante o primeiro acesso ao 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 I/O) [42][38][44].

Como forma de aumentar o desempenho e reduzir a poluição da hierarquia de cache, inserindo dados que possivelmente serão inúteis, pode-se adotar estratégias para realizar o *bypass* (contornar) toda ou parte da hierarquia de cache. Técnicas tradicionais de cache *bypassing* são empreendidas em conjunto com as políticas de substituição de dados da cache e preditores de reúso dos dados, buscando otimizar tanto o uso do espaço de armazenamento quanto o uso da largura de banda do processador.

O uso de técnicas de cache *bypass* é amplamente difundida, sendo utilizada nas mais variadas tecnologias. Inicialmente projetadas para serem utilizadas junto a CPUs, acabaram sendo expandidas para usos em diferentes arquiteturas, tais como GPUs e sistemas embarcados [50][31], em dispositivos construídos com tecnologias diferentes [30][29], podendo ser utilizadas através de compilação [50] [7] ou de forma dinâmica [27] [26] [45].



(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.

Quando propõe-se o uso de técnicas de cache *bypass*,busca-se ignorar as latências provocadas pelos múltiplos níveis de cache, realizando a instalação dos dados no nível da hierarquia onde é previsto que seja mais útil, em detrimento dos outros níveis. Uma representação pode ser vista na 6.1, onde a Figura 2.2a 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 2.2b representa o fluxo da mesma requisição, ao se empregar uma técnica de cache *bypass*. Tal técnica

é factível em caches que não utilizam utilizam uma abordagem inclusiva, uma vez que ao realizar estes procedimento, a propriedade de inclusão da cache seria violada. Além disso técnicas como *first-word-first* [19], apresentam-se como alternativa elegante para redução da latência de dados durante a movimentação DRAM $\overset{dados}{\longrightarrow}$ CPU.

3 TRABALHOS CORRELATOS

Neste capítulo, apresentamos as pesquisas relacionadas a nossa proposta. Primeiro, na seção 3.1, apresentamos abordagens sobre o uso de *bypass* de dados. Na seção 3.2, apresentamos as abordagens que se valem do uso/realização de *bypass* de requisições. Estes trabalhos são os que mais se aproximam do tema desta dissertação, quando considerada a natureza dinâmica dos mecanismos.

3.1 PESQUISAS EM TÉCNICAS DE CACHE *BYPASS* DE DADOS

Focando no uso de técnicas de cache *bypass* em CPUs, a grande maioria de trabalhos concentram-se em caches projetadas de forma não inclusiva ou exclusiva [36]. Trabalhos que buscam soluções de *bypass* em caches exclusivas e não inclusivas utilizam as mais diversas abordagens. Entre estas abordagens, temos as que utilizam contadores de reúso dos dados [28] [52] [40] e as que utilizam distância entre acessos realizados[45] [6] para definir qual a melhor maneira de tratar os dados.

O desenvolvimento de técnicas para caches inclusivas são mais escassos, dado que ao se realizar o *bypass*, inserindo os dados nos níveis mais altos, de forma a evitar a poluição da cache, acaba por violar a propriedade de inclusão. Assim, buscando contornar isto, pode-se utilizar uma abordagem através do *software*. A Instruction Set Architecture (ISA) x86 provê instruções específicas para evitar o uso da cache [11]. Outra forma, via *hardware*, é ao instalar a nova linha, inseri-la como Least Recently Used (LRU). Desta forma, a linha terá prioridade em ser removida quando uma nova instalação for requisitada [48] [32].

Ainda utilizando soluções em *hardware*, temos arquiteturas novas, como a proposta do Bypass Buffer (BB) [16]. Durante o trabalho, [16] observou que cerca de 94% de blocos da Last-Level Cache (LLC) são evitados em até 8 LLC misses. Assim, [16] projetou uma nova organização para a LLC. A LLC é dividida em dois blocos principais. O primeiro bloco é o último nível de cache propriamente dito, onde são armazenados os dados. O segundo bloco é chamado de Bypass Buffer.

Este *buffer* é utilizado para manter a propriedade de inclusividade da cache, funcionando como uma tabela. Conforme a expectativa de reúso dos dados, no momento em que uma linha de cache é instalada, se for esperado reúso, a linha é instalada na LLC. Caso contrário, é armazenado o *tag* da linha no BB. Assim, quando uma entrada do BB deve ser evitada, as linhas presentes nos níveis superiores de cache serão evitados para manter a inclusividade. Entretanto, como o bloco evitado estava no BB, a possibilidade de o bloco ter sido evitado nos níveis superiores é grande [16].

Similar ao trabalho [16], [30] apresenta um método de *bypass* para LLC inclusivas baseadas na tecnologia de Spin Torque Transfer Random Access Memory (STT-RAM). Baseado na proposta [54], e utilizando a abordagem apresentada em [16] para manter a inclusividade, [30] utiliza uma estrutura que é chamada de Bypass Tag Cache (BTC) para armazenar as *tags* dos blocos a serem instalados, assim mantendo a inclusividade. Para a tomada de decisões sobre quais blocos devem sofrer *bypass*, é utilizado o Bypassing Decision Block (BDB).

3.2 PESQUISAS QUE APRESENTAM TÉCNICAS DE CACHE *BYPASS* DE REQUISIÇÕES

Quando comparados aos trabalhos relatados na seção 3.1, os trabalhos apresentados nas subseções 3.2.1 e 3.2.2 se mostram os mais próximos do conceito proposto neste trabalho e por isso serão analisadas mais detalhadamente.

3.2.1 Load Miss Prediction (LMP) e *Bypass*

Em aplicações que apresentam baixa localidade ou baixo reúso, a hierarquia de cache acaba por adicionar mais latência em uma requisição que acaba por encaminhar-se à memória principal. Assim, dotando-se o processador de um mecanismo capaz de prever tais acessos, que acabam por ser inevitáveis, pode-se mitigar esta adição. Como exemplo, [34] utiliza uma hierarquia de memória composta por dois níveis de cache, L1 e L2, e a memória principal (DRAM), com latências de 1, 19 e 122 ciclos e estima que ao realizar o *bypass* da cache L2 em requisições que se encaminham para a memória principal, é possível poupar até 16% da latência total.

Desta forma, em [34] é definido um mecanismo chamado de Load Miss Prediction (LMP), capaz de predizer quando os dados de uma requisição não estarão presentes na hierarquia de cache. Para viabilizar isto, é proposta uma hierarquia de cache composta de dois níveis. O nível L1, dotado de 32 KB, é dividido em duas porções, uma com 16 KB, altamente associativa e com uma linha de tamanho 32 B, chamada de *regular-cache*, conectada com o nível L2. A segunda porção, também dotada de 16 KB, porém, possui conexão direta com o nível L2 e com o controlador de memória. Esta porção da cache é chamada de *L1-Bypass*. Os dados mantidos na *L1-Bypass* são providos diretamente da memória principal, com o nível L2 fazendo o papel de *victim-cache*.

O LMP funciona como uma tabela de dois níveis, mapeando até 1024 instruções independentes, indexadas pelo PC. Cada entrada armazena um histórico com até 8 entradas, entre *hits* e *miss*. Ao executar uma instrução de *load*, o dado é requisitado a ambas as caches, regular e *bypass*. Se ocorre um cache miss em ambas as caches, então o mecanismo responsável pela predição é consultado. O LMP define, então, se esta requisição será um *miss*, ou um *hit* no último nível de cache.

Instruções que repetidamente causam um LLC *miss* são marcadas como passíveis de sofrerem *bypass*. Quando em novamente executadas, se é previso que a instrução acabará gerando um *miss*,é encaminhada para a memória principal. Paralelamente, é encaminhada uma requisição ao último nível de cache. Caso a LLC responda com os dados, a requisição realizada para a memória principal é cancelada, ou caso dados tenham sido retornados, os mesmos são ignorados.

Para garantir a consistência da cache, quando são executadas operações de *store*, ambas as caches L1 são verificadas. Desta forma, para a execução de um *store*, a requisição irá proceder pela hierarquia de memória como se fosse um *load*, sem entretanto, acionar o mecanismo.

3.2.2 Enhanced Memory Controller - EMC

Para que a latência das requisições possa ser diminuída, ou mesmo eliminada, técnicas de pré-busca podem ser empregadas. Estas técnicas tentam prever quando um determinado dado será utilizado, e assim, evitar que o processador fique esperando por muito tempo os dados. Podemos notar que os programas apresentam diferentes padrões de acesso a memória, e, por vezes são complexos para serem previstos com perfeita precisão. De fato, aplicações do tipo *pointer-chasing* (utilizadas, por exemplo, em listas encadeadas e grafos) são especialmente suscetíveis a atrasos, mesmo com mecanismos sofisticados de pré busca. Somado a isso, tem-se

o fato que geralmente o último nível de cache, em sistemas *multi-core* é compartilhado, fazendo com que a competição por espaço na memória cache seja maior.

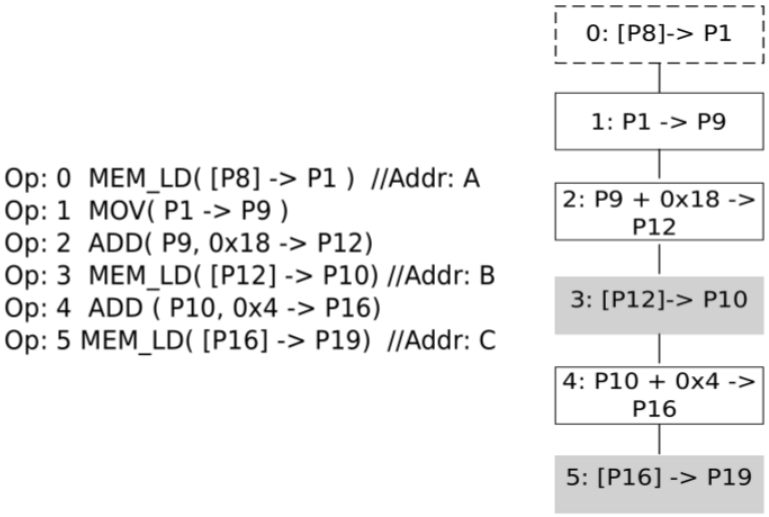


Figura 3.1: Sequência de instruções para geração de endereço [17].

Tabela 3.1: Instruções suportadas pelo EMC.

Tipo	**Instrução**
Integer	add,subtract,move,load,store
Logical	and,or,xor,not,shift,sign-extend

A figura 3.2 mostra o posicionamento do Enhanced Memory Controller (EMC) em relação a hierarquia de cache e ao *core*. Segundo os autores, como o EMC pode realizar requisições diretamente a memória principal, sem a necessidade de passar por toda a hierarquia de cache, isso permite que as requisições realizadas por ele experimentem uma redução de cerca de 20% na latência, quando comparada a uma requisição vinda diretamente do núcleo de processamento [17].

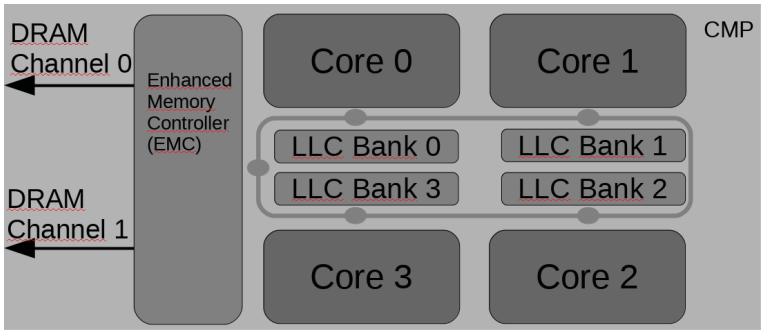


Figura 3.2: Visão de alto nível de um processador multi-core equipado com o EMC.

3.3 CONCLUSÕES

Dado o exposto neste capítulo, os trabalhos que visam utilizar cache *bypass* de dados para incrementar o desempenho exploram principalmente o uso de caches não inclusivas. Dos

trabalhos que exploram caches inclusivas, são propostas mudanças extensas na organização, o que acaba por gerar um *overhead* de espaço significativo.

Posteriormente são apresentadas formas de se fazer o *bypass* utilizando as requisições de dados. O trabalho EMC leva a geração de requisições para mais próximo da memória, evitando a latência da hierarquia de cache, o que aparenta ser promissor, embora demande uma reorganização do fluxo das instruções no processador. Em nossa proposta, buscamos uma solução simples, porém eficaz, capaz de reduzir a latência percebida pelas requisições.

Dessa forma, buscamos criar requisições em paralelo para cache e para a memória DRAM, para fazer o drible (bypass) das requisições, quando for previsto que se trata de um cache miss.

4 SIMULADOR

Um dos motivos de se utilizar simuladores durante as etapas de pesquisas em arquitetura de computadores é devido ao fato de ser uma alternativa mais prática, quando comparado a complexos modelos analíticos, ou com os alto custos para a prototipação de um chip que implementa uma nova arquitetura a ser testada [24]. Simuladores arquiteturais podem ser divididos em duas grandes categorias. Simuladores *full-system*, que simulam um ambiente computacional inteiro (realmente efetuando os passos de computação) e simuladores dedicados, que simulam somente componentes específicos (emulando o comportamento dos componentes) de forma independente. Dentre estes simuladores específicos, temos os que são orientados a traços (*trace driven*). Este tipo de simulador utiliza como entrada um arquivo, contendo um traço de execução. Este traço contém as instruções efetivamente executas pelo programa, isto é, a sequência de instruções que a máquina real executou [9]

Simuladores arquiteturais são construídos de forma a simular detalhadamente um determinado conjunto de parâmetros de um sistema. Uma ferramenta ideal deve ser flexível, de forma a suportar um grande conjunto de configurações, e simular de forma precisa o comportamento de uma máquina, quando comparada a uma máquina real.

Considerando o nosso objetivo primário de reproduzir o mecanismo Enhanced Memory Controller (EMC) [17] foi considerado estender o Simulator of Non-Uniform Cache Architectures (SiNUCA) para suportar as funcionalidades apresentadas pelo EMC. Contudo, a complexidade do código apresentado pelo SiNUCA ocasionou dificuldades para a implementação do EMC dentro do simulador. Assim, levando em conta os estudos sobre os diferentes tipos de simuladores arquiteturais [3], decidiu-se desenvolver um simulador simplificado. Optou-se então desenvolver um simulador baseado no SiNUCA, removendo elementos que geram complexidade, porém que não são essenciais para a fiel simulação do comportamento da máquina.

4.1 *ORDINARY COMPUTER SIMULATOR - ORCS*

O Ordinary Computer Simulator (OrCS) é um simulador construído sob o paradigma *trace-driven*, desenvolvido para ser compatível com os traços utilizados pelo simulador SiNUCA. Entretanto, sua construção é simplificada, modelando apenas os componentes que tem impacto significativo no desempenho. Tal como o SiNUCA, o OrCS foca em simular aplicações das Instruction Set Architecture (ISA) x86_32 e x86_64, sem que ocorram interferências de outros processos ou do sistema operacional.

4.1.1 Componentes modelados

Na Figura 4.1, estão ilustrados os componentes modelados pelo simulador. Abaixo está a descrição de cada componente:

- **Processador:** Este componente é responsável pela simulação do comportamento da execução das operações (opcodes). Foi modelado utilizando 6 estágios de pipeline: *fetch, decode, rename, dispatch, execute e commit*. Modelou-se um processador com execução Out-of-Order (OoO). Cada estágio do processador pode ser configurado para levar um número variável de ciclos para ser completado. Visando a facilidade de extensão, cada estágio foi implementado de forma separada, tornando-o mais flexível.

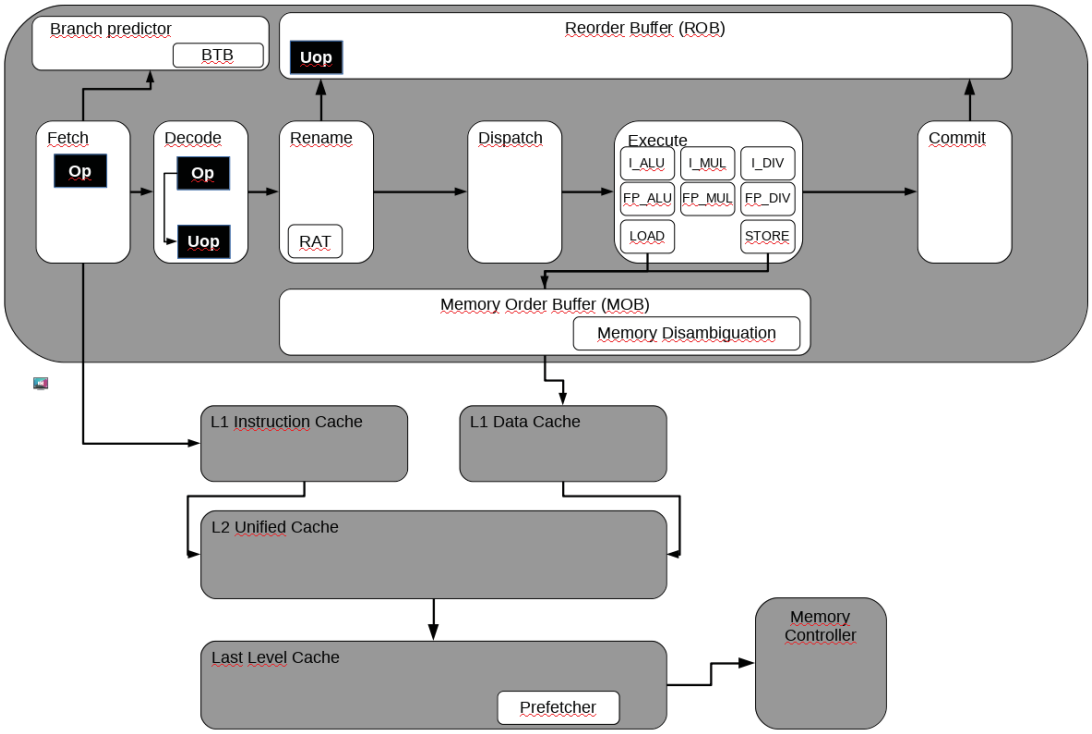


Figura 4.1: Arquitetura do Simulador OrCS.

- **Opcode Package:** Este componente encapsula as informações que representam uma instrução empacotada. Cada pacote é utilizado durante a simulação para representar a busca de instruções quando é simulado o estágio *fetch* de um processador.

- **Uop Package:** Após a instrução atingir o estágio de *decode*, ela é desmembrada em instruções menores, as *microops*. O pacote *uop* encapsula as *microops* para uso nas outras estruturas do processador.

- **Branch Predictor:** O módulo do preditor de desvios (*branch predictor*) é responsável por instanciar qualquer mecanismo que possa ser utilizado para prever se a instrução é um desvio, bem como seu endereço alvo e direção. Contém um Branch Target Buffer (BTB), e outras estruturas necessárias pelo mecanismo implementado. Ao chegar no ponto de simulação do estágio de *fetch*, o *core* requisita ao *branch predictor*, caso a instrução buscada seja um *branch*, a direção a ser seguida (desvio tomado ou não) e o endereço destino. Caso a predição esteja correta, a instrução será processada em seguida. Caso a predição seja incorreta, o "estágio"de *fetch* sofrerá uma parada, até que o caminho de execução seja corrigido. A atualização dos dados para realizar as predições é atualizada a cada utilização. Atualmente, o único mecanismo disponível para utilização é o Piecewise Linear Branch Predictor (PLBP) [25].

- **Memória Cache:** O módulo de cache é responsável por modelar a hierarquia de memória *on-chip* do processador. Neste simulador, são modelados as caches de instruções e de dados do sistema. O módulo é modelado como um conjunto de *arrays*, armazenando somente o identificador para as linhas de cache, de forma a poupar o uso de memória pelo simulador.

- **Prefetcher:** O mecanismo de pré-busca examina cada requisição que é enviada do último nível de cache para a memória principal. Com o endereço dos dados requisitados,

o mecanismo pode prever qual o próximo dado necessário, realizar a requisição, e assim diminuir a latência para acessar os dados. O único mecanismo disponível para utilização atualmente é o Stride/Delta-Correlation (S/DC) [12].

- **Controlador de Memória:** Este componente modela o controlador da memória principal. Modelado de forma simplificada, implementa a separação de requisições por diferentes canais, acesso aos bancos de memória,*row buffer* e a utilização do barramento. A política de processamento das requisições é First-Come First-Serve (FCFS), com o *row buffer* operando sob a política de *open row*.

Determinados componentes, tais como o mecanismo de *prefetcher* e o mecanismo de desambiguação de memória podem ser facilmente ativados/desativados, simplificando a tarefa de simular diferentes cenários e micro-arquiteturas.

4.1.2 Validação

Para garantir a precisão do simulador, é necessário realizar um trabalho de validação deste, seja contra uma máquina real, ou contra um simulador confiável. Assim, de forma a coletar os dados a respeito do comportamento do simulador, foram executados duas cargas de testes, ou *benchmarks*, compostas de diferentes aplicações. Um dos *benchmarks* executados foi do conjunto SPEC CPU-2006 [20], e o outro foi o conjunto desenvolvido para a validação do SiNUCA, denominado *microbenchmarks* [3]. Ambos *benchmarks* foram comparados contra uma máquina real e o simulador SiNUCA. Os simuladores foram configurados para simular a microarquitetura Intel Sandy Bridge [53].

As aplicações do *microbenchmark* podem ser divididas em quatro grandes grupos:

- **Controle:** Os testes que são realizados nesta categoria são destinados a estressar o mecanismo responsável pela predição dos desvios. Os testes são desenvolvidos de forma a misturar instruções do tipo *if-the-else* e *switch*, para avaliar diferentes aspectos do comportamento do *branch predictor*. Neste teste algumas diferenças são esperadas. Elas podem ser explicadas em virtude dos diferentes mecanismos implementados, tanto nos simuladores comparados, quanto na máquina real, onde o mecanismo utilizado não é publicamente conhecido.

- **Dependências:** Os testes de geração de dependências verificam se o processo de renomeação dos registradores estão sendo realizadas corretamente, mantendo a dependência entre as instruções, de forma a manter a correta ordem de execução. Os resultados obtidos atestam que a geração de dependências está correta. Caso não estivessem, ocorreriam oscilações bruscas no IPC, tanto em relação à máquina real quanto ao SiNUCA.

- **Execução:** O teste de execução estressa as unidades funcionais, de forma a verificar se elas estão sendo utilizadas de forma correta. Pode-se perceber que, quando comparado com ambos confrontantes, o OrCS apresenta as mesmas variações, em função do tipo de unidade funcional testada.

- **Memória:** Os testes que estressam os componentes de memória são divididos naqueles que executam operações de *load*, de forma independente e dependente, e aqueles que executam operações de *store*. Nos testes que devem executar operações do tipo *load* de forma dependente, como em uma busca de ponteiros, nota-se que o IPC ficou muito

próximo ao da máquina real e do SiNUCA. Nos *benchmarks* que executam as operações de *load* de forma independente, o IPC encontra-se acima da máquina real, entretanto, está mais próximo do quando comparado ao outro simulador. Para as operações de *store*, também nota-se que o IPC ficou acima da máquina real, contudo, o desempenho está bem próximo ao do simulador comparado. Isso pode ser explicado pela ausência da modelagem das estruturas de interconexão entre os componentes do chip, cujo efeito é de ocasionar contenção no acesso a memória em um processador real, o que acaba limitando o desempenho. No lugar desta modelagem, optou-se por restringir o número de requisições que o processador pode realizar de forma paralela.

Estas aplicações foram desenvolvidos para estressar componentes específicos do processador, durante o desenvolvimento do SiNUCA[3].

Para a avaliação do desempenho do simulador, comparamos os resultados de IPC que o simulador conseguiu com os resultados de IPC obtidos pelo processador real. Quanto mais próximo os resultados de IPC entre a máquina real e o simulador são, mais fielmente foi modelado o comportamento do processador real pelo simulador.

A Figura 4.2 mostra o resultado de IPC referentes as aplicações que pertencem aos grupos de controle, dependências e execução. No grupo de controle, o OrCS aproxima-se mais do comportamento da máquina real em 4 de 5 aplicações. Estas diferenças são explicadas pelos diferentes mecanismos que são utilizados, tanto no processador real quanto nos simuladores. Nas aplicações dos grupos de dependência e execução, os resultados obtidos são muito próximos em todas as aplicações, mostrando que tem-se uma modelagem bastante fiel dos componentes que executam as tarefas de geração de dependências entre instruções e execução no processador real.

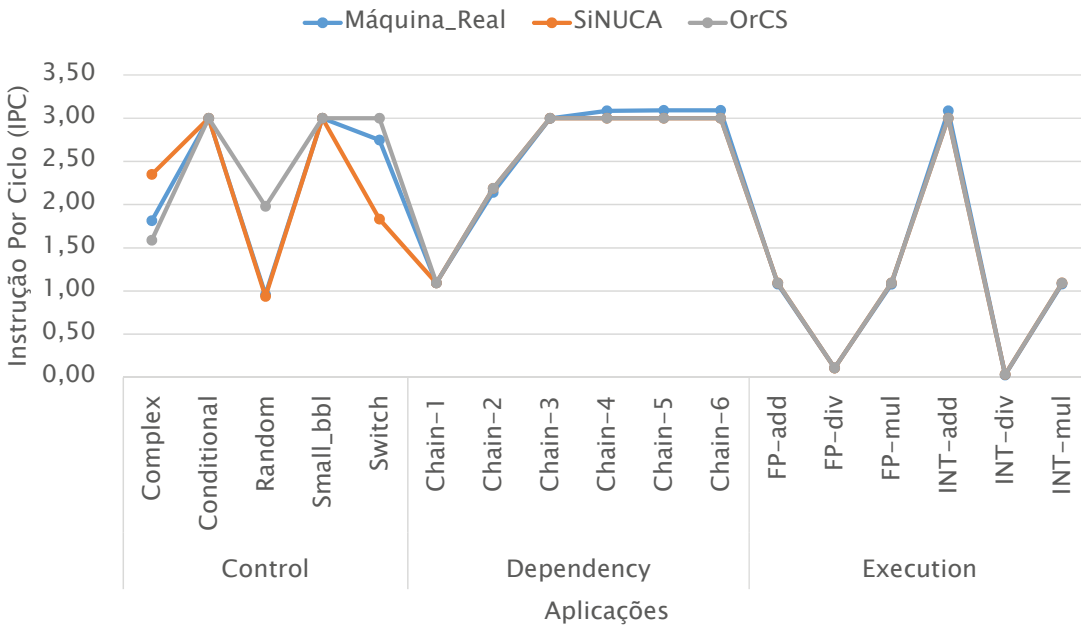


Figura 4.2: Gráfico de IPC por *microbenchmark* das categorias *control, dependency* e *execution*.

A figura 4.3 apresenta os resultados de IPC para as aplicações do grupo de memória. No grupo que executam instruções de *load* de forma dependente, isto é, onde um *load* depende diretamente do anterior, o IPC obtido pelo OrCS frente à máquina real mostra-se bem similar quando utilizando a hierarquia de cache. Quando passa a utilizar a memória DRAM, o OrCS apresenta uma grande variação. O mesmo comportamento pode ser observado nas aplicações que executam operações de *load* e de *store* de forma independente. Quando utilizada a cache,

o IPC obtido pelo OrCS é mais próximo à máquina real do que o obtido pelo SiNUCA. No caso das operações de *store*, o IPC do OrCS apresenta-se mais distante da máquina real do que quando comparado com os *loads*, entretanto, apresenta-se mais próximo do que o SiNUCA. Quando utilizada a DRAM, o OrCS apresenta uma diferença considerável, tanto nas aplicações de *load* quanto nas de *store*. Esta diferença pode ser explicada devido a diferentes mecanismos de pré-busca e desambiguação de memória implementados nos simuladores, enquanto que no processador real não sabemos exatamente qual mecanismo é implementado. Ao utilizar a DRAM, é utilizada uma modelagem parcial, o que ocasiona as diferenças quando a aplicação requisita dados a partir da memória.

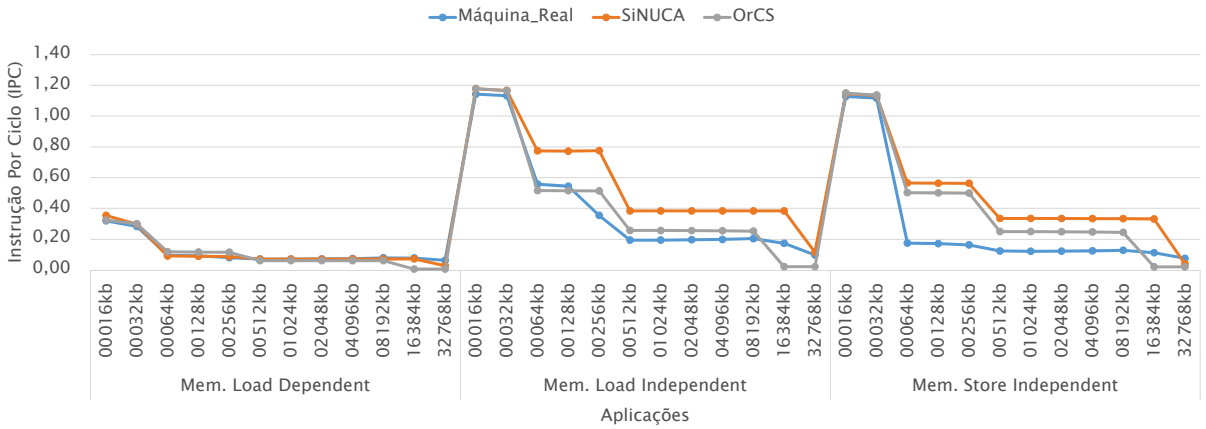


Figura 4.3: Gráfico de IPC por *microbenchmark* das categorias de load dependente, load independente e store independente.

Para testar em um ambiente com aplicações reais, foram escolhidos os programas do SPEC CPU-2006 . A Figura 4.4 apesenta o erro relativo entre os resultados obtidos pelos simuladores SiNUCA e OrCS, quando comparados a uma máquina real. Com este gráfico, espera-se não ser induzido a uma avaliação errônea da confiabilidade do simulador.

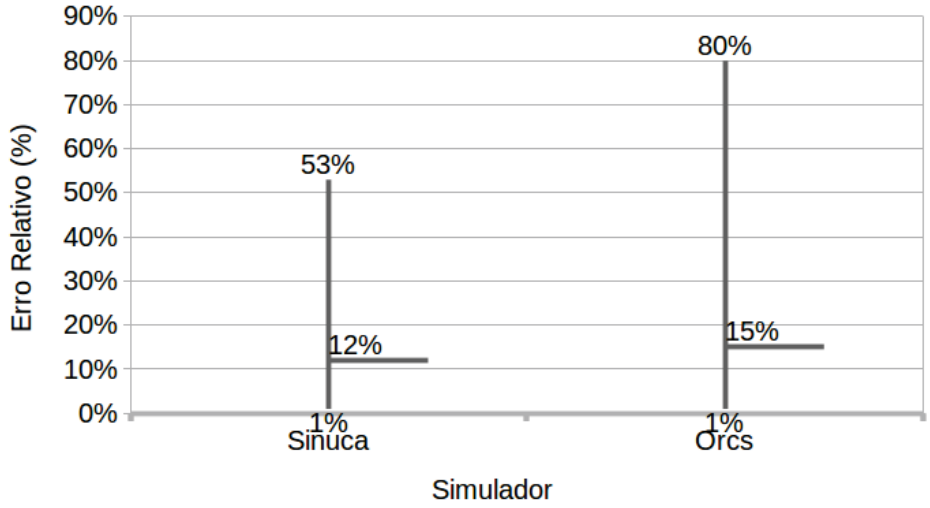


Figura 4.4: Erros relativos, mínimo, máximo e médio entre os simuladores SiNUCA e OrCS comparados com uma máquina real.

Sumarizando os resultados, o simulador desenvolvido apresentou uma média de erro de 15%, com valor máximo e mínimo de erro em 80% e 1%, respectivamente. Quando comparado com o simulador SiNUCA, OrCS apresenta um acréscimo de 3% no erro médio.

Entretanto, ao comparar o número de linhas de código (que nos indica a complexidade construtiva do simulador) e o número de instruções simuladas por segundo, o OrCS apresenta um tamanho aproximadamente 67% menor que o SiNUCA, enquanto consegue simular cerca de 56% mais instruções por segundo. Os dados referentes ao número de linhas de código foram obtidos através do programa SLOCCount [49].

4.2 CONCLUSÕES

Simuladores arquiteturais são de suma importância para o desenvolvimento de pesquisas na área de arquitetura de computadores. Seu uso permite a validação de propostas de uma forma mais prática e simplificada, quando comparadas a prototipação de novas soluções de *hardware*. Neste capítulo foi apresentado o simulador OrCS, desenvolvido no decorrer deste trabalho. Quando comparado ao simulador SiNUCA, apresenta um desempenho muito próximo a este, enquanto ostenta uma codificação significativamente menor.

5 REPLICAÇÃO DO MECANISMO EMC

Neste capítulo apresentaremos os resultados obtidos na replicação do mecanismo Enhanced Memory Controller (EMC). Inicialmente, será discorrido sobre a metodologia de avaliação para o mecanismo replicado. Em seguida, será apresentado a configuração do simulador para a execução dos experimentos. Por fim, serão apresentados os resultados obtidos na replicação do mecanismo EMC.

O EMC é uma proposta de Milad Olia Hashemi, que visa reduzir a latência de requisições de dados que acabam sendo encaminhadas para a memória principal e não possuem seu endereço conhecido à *priori*, sendo desenvolvido durante seu doutorado. Um breve detalhamento dessa proposta foi apresentado na Seção 3.2.2. Mais detalhes de implementação podem ser encontrados no artigo [17] e na tese publicadas [18].

5.1 METODOLOGIA DE AVALIAÇÃO

Desempenho é um aspecto chave no projeto e desenvolvimento de sistemas computacionais. A análise de desempenho em si é uma arte, intimamente ligada ao conhecimento e ao senso crítico. Assim, inicialmente, deve-se estabelecer objetivos e um escopo bem definido sobre o que será avaliado.

Dada a função prática do que está sendo avaliado, a avaliação de uma métrica pode ser enquadrada em três classes [24]:

- **Maior é melhor**: Buscado quando se deseja aumentar o desempenho de um sistema. Ex: *Throughput*, Instructions per Cycle (IPC), *speedup*;
- **Menor é melhor**: Buscado quando o objetivo é reduzir custos ou o tempo. Ex:Energia gasta, tempo de resposta, Misses per Kilo Instructions (MPKI);
- **Nominal é mais desejável**: Quando valores muito altos ou baixos são indesejáveis. Ex: Taxa de uso de um serviço computacional, valores muitos baixos representam falta de uso e desperdício de recursos, valores muito altos, muita concorrência e degradação do tempo de resposta;

Uma boa métrica de avaliação tem por características ser linear, confiável, determinística, fácil de medir, consistente e independente [33]. Tais características também podem ser utilizadas para classificar determinados elementos, a fim de identificar cenários onde uma modificação pode vir a ser mais eficiente. Quando fala-se do mecanismo EMC [17], o foco são aplicações com alta intensidade no uso da memória. Assim, para classificar as aplicações foi utilizada a métrica MPKI.

Para avaliarmos a mudança no desempenho do sistema, será utilizada a métrica de melhoria de IPC. O uso do IPC é factível para avaliações de sistemas *single-core*, entretanto, para sistemas *multi-core*, utilizar somente o IPC, não apresenta um panorama real do trabalho realizado. Para tal, devemos utilizar uma métrica que leve em consideração a contribuição que cada núcleo de processamento fez para a conclusão carga geral de trabalho. Assim, para os experimentos *multi-core*, será utilizada a métrica de Weight Speedup (WS). WS é dada pela soma

de IPC de cada processo, quando pareado com outros processos, divido pela somatória de IPC dos respectivos processos, quando executando isoladamente [46] [51] [32].

$$Wspeedup = \sum_{i=0}^{n-1} \frac{IPC_i^{pareado}}{IPC_i^{isolado}} \tag{5.1}$$

5.2 CONFIGURAÇÃO DO AMBIENTE DE SIMULAÇÃO

Para conduzir os experimentos, foi utilizado o simulador Ordinary Computer Simulator (OrCS) descrito no capítulo 4. Como objeto de avaliação foi utilizada as aplicações do *benchmark* SPEC CPU-2006 [20]. O simulador foi configurado para se comportar como um processador da microarquitetura Intel Skylake [10], sendo este o *baseline* utilizado nas comparações. Demais detalhes da arquitetura estão descritos na Tabela 5.1.

Tabela 5.1: Configuração do sistema

Core	Núcleos: 1 ou 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 [25];
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	1-Core: 1MB. 4-Cores: 4MB (Compartilhada); Linha: 64 B. Latência: 44 ciclos. 8 vias. Writeback;
Memory Controller	FCFS, 1 core: 16 entradas memory queue; 4 cores: 64 entradas memory queue;
Memory Controller Compute Unit	IPC máximo 2; Uop Buffer: 16 entradas; Canais:2 RS: 8 entradas.
EMC Data Cache	4KB; Latência: 1 ciclo. 4 vias;
Memory Access Counter Table	Tamanho: 256 entradas; Contador: 3 bits;
DRAM	DDR3; 1 Rank com 8 Banks por canal; Barramento: 8B; Row Buffer 8 KB; RP-RAS-CAS (11-11-11) (13.75 ns); Frequência DRAM: 800 MHz; Activate Energy (nJ): 0.3416 Read Energy (nJ): 3.025; Write Energy (nJ): 3.3782 Precharge Energy (nJ): 1.1339

Foram avaliados ambientes *single-core* e *multi-core*. No ambiente *multi-core*, cada *core* tem as caches L1 e L2 privadas, sendo a Last-Level Cache (LLC) compartilhada. A avaliação *single-core* foi feita utilizando todas as 29 aplicações do SPEC CPU-2006 . Para a avaliação *multi-core*, foram utilizados conjuntos de 4 aplicações geradas de forma aleatória, mesclando aplicações com alta, média e baixa intensidade no uso da memória. Também foram criados conjuntos de aplicações homogêneos, contendo 4 cópias de cada aplicação.

Cada aplicação é um traço contendo 200 milhões (200 M) de instruções representativas, obtidas através do método PinPoints [41]. As aplicações foram simuladas em sua totalidade, ou seja, cada núcleo completou 200 M de instruções.

5.3 AVALIAÇÃO EXPERIMENTAL

Como ponto de partida, inicialmente avaliamos a reprodutibilidade do trabalho EMC. Apresentou-se factível ser replicado, e possivelmente estendido para comportar outras instruções. A base para o trabalho EMC é a dependência entre as instruções, mais precisamente, instruções que acabam por gerar acessos a memória. Para verificar se as aplicações apresentam mesmo a característica explorada, foi realizado uma contagem simples entre as instruções que apresentam dependências. As Figuras 5.1 e 5.2 exibem os resultados, obtidos no trabalho EMC e os obtidos na replicação deste trabalho, respectivamente. Nota-se que as aplicações realmente apresentam esta dependência de instruções que geram acessos a memória, porém não em um nível tão elevado quanto o apresentado no EMC. Deve-se ressaltar que as aplicações *bwaves*, *lbm* e *zeusmp* não apresentaram estas dependências.

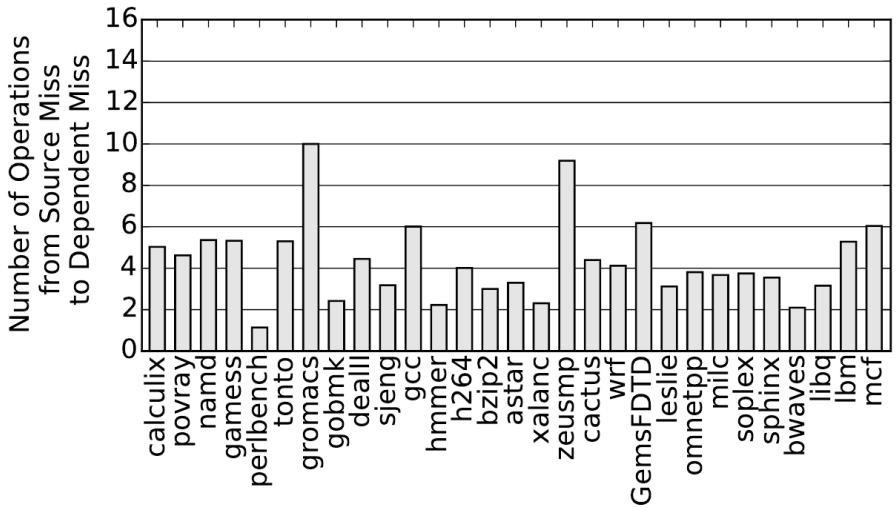


Figura 5.1: Quantidade de operações entre *load* misses dependentes apresentado em [17]

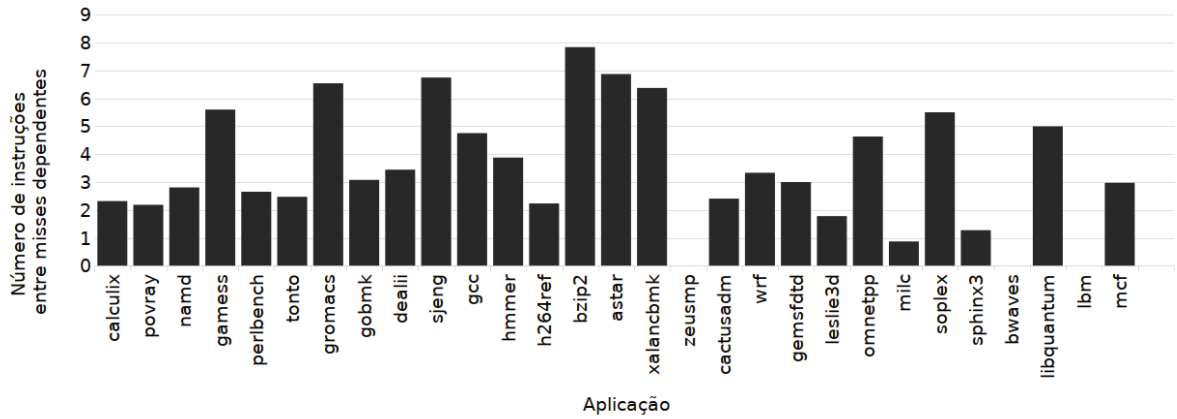


Figura 5.2: Quantidade de operações entre *load* misses dependentes verificado na replicação do trabalho [17]

Assim, verificada a existência das condições para que o EMC possa ser ativado, verificamos outro ponto que é levantado no trabalho EMC, aplicações com alta intensidade

de uso da memória. Para definir quais são as aplicações com alta intensidade, utilizou-se o número de Misses per Kilo Instructions da LLC para classificar as aplicações, conforme sua intensidade de uso de memória. Antevendo a possibilidade de existirem diferenças, e em face as diferenças apresentadas quanto ao número de instruções entre *loads* dependentes, realizou-se a classificação das aplicações, seguindo o mesmo critério. Ambas as classificações são apresentadas na Tabela 5.2.

Tabela 5.2: Classificação das aplicações do SPEC CPU-2006 com base na intensidade de uso da Memória

Categoria	(A) Classificação em [17]	(B) Classificação obtida na replicação
Alta (MPKI >= 10)	omnetpp, milc, soplex, sphinx3, bwaves, libq,lbm, mcf	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)
Média (MPKI>=5)	zeusmp, cactusADM, wrf, GemsFDTD, leslie3d	astar (6.9), cactusadm (9.0)
Baixa (MPKI<5)	calculix, povray, namd, gamess, perlbench, tonto, gromacs, gobmk, dealII, sjeng, gcc, hmmer, h264ref, bzip2, astar, xalancbmk	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)

Foram criados aleatoriamente conjuntos de aplicações com base na classificação apresentada na Tabela 5.2 (B). Os conjuntos de aplicações de alta intensidade mesclam de forma aleatória 4 aplicações classificadas como de alta intensidade, os conjuntos de aplicações de média intensidade mesclam duas aplicações de alta intensidade e duas de média intensidade, e os conjuntos de aplicações de baixa intensidade mesclam duas aplicações de alta intensidade e duas aplicações de baixa intensidade. Tais conjuntos somam-se aos conjuntos de aplicações apresentados no trabalho [17]. A lista de conjuntos de aplicações informados no trabalho [17] é apresentado na Tabela 5.3, enquanto que o conjuntos de aplicações gerados pode ser vista na Tabela 5.4.

Tabela 5.3: Conjunto de aplicações apresentados em [17]

HM0	bwaves+lbm+milc+omnetpp
HM1	soplex+omnetpp+bwaves+libq
HM2	sphinx3+mcf+omnetpp+milc
HM3	mcf+sphinx3+soplex+libq
HM4	lbm+mcf+libq+bwaves
HM5	lbm+soplex+mcf+milc
HM6	bwaves+libq+sphinx3+omnetpp
HM7	omnetpp+soplex+mcf+bwaves
HM8	lbm+mcf+libq+soplex
HM9	libq+bwaves+soplex+omentpp

Apesar de projetado para ser utilizado principalmente em ambientes *multi-core* [18], o EMC apresenta pequenas vantagens para execução em ambientes *single-core*. Assim, foram executados testes *single-core* também, conforme apresentado na Figura 5.3. Nota-se que, como

Tabela 5.4: Lista de conjuntos de aplicações com base na classificação de MPKI da tabela 5.2 B

H0	bwaves+gcc+libq+mcf
H1	libq+soplex+sphinx3+wrf
H2	mcf+omnetpp+soplex+wrf
H3	mcf+soplex+sphinx3+wrf
H4	bwaves+gcc+mcf+soplex
H5	gcc+libq+sphinx3+wrf
H6	bwaves+gcc+omnetpp+sphinx3
H7	gcc+libq+soplex+wrf
H8	bwaves+gcc+soplex+sphinx3
H9	bwaves+libq+mcf+wrf

M0	astar+cactusADM+bwaves+soplex
M1	cactusADM+astar+libq+sphinx3
M2	mcf+astar+cactusADM+sphinx3
M3	astar+cactusADM+gcc+soplex
M4	cactusADM+bwaves+omnetpp+astar
L0	xalancbmk+gromacs+bwaves+soplex
L1	gromacs+bwaves+omnetpp+xalancbmk
L2	soplex+xalancbmk+gromacs+sphinx3
L3	xalancbmk+libq+mcf+gromacs
L4	gromacs+xalancbmk+soplex+wrf

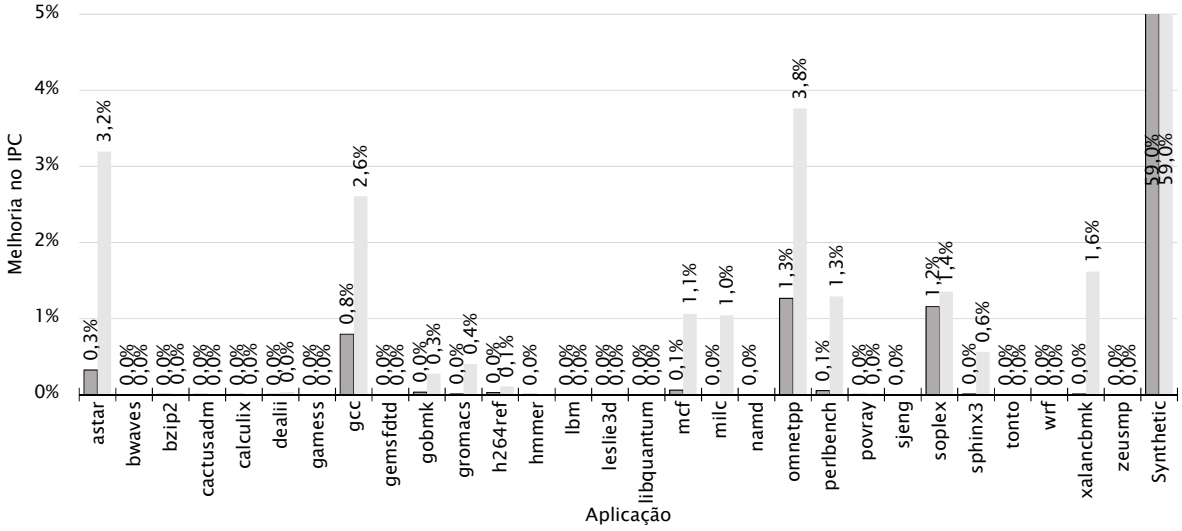


Figura 5.3: Incremento no IPC relativo ao *baseline* obtido na replicação do EMC utilizando o mecanismo implementado como descrito em [17] e utilizando um oráculo.

o relatado em [18], os resultados apresentados no cenário *single-core* são coerentes, embora com variações entre determinadas aplicações. A Tabela 5.5 apresenta os resultados obtidos nas aplicações classificadas como de alta intensidade de memória, quando simulados um ambiente *single-core*.

Tabela 5.5: Resultados *single-core* utilizando o mecanismo EMC proposto.

Aplicação	**bwaves**	**gcc**	**gemsfdtd**	**lbm**	**leslie3d**	**libq**	**mcf**
Resultado	0%	0,79%	0,00%	0,00%	0,00%	0,00%	0,06%
Aplicação	**milc**	**omnetpp**	**soplex**	**sphinx3**	**wrf**	**zeusmp**	**GeoMean**
Resultado	0,00%	1,26%	1,16%	0,01%	0,00%	0,00%	**0,00%**

Ao serem executadas as aplicações em um ambiente *multi-core*, os resultados apresentados não foram animadores. Assim, foi utilizado dois métodos para avaliar se o componente estava funcionando como o descrito em [17], um *microbenchmark* sintético (*Synthetic*) para avaliar de forma isolada o desempenho do componente EMC, e o uso de um oráculo para avaliar se as cadeias de instruções cumprem os requisitos para o uso do componente. O *microbenchmark* realiza *loads* sucessivos, de forma interdependente. Assim, passaremos a nos referir ao EMC

quando utilizado o contador saturado como EMC Tradicional, e quando utilizado em conjunto com o oráculo como EMC Oracle. Com o uso do *microbenchmark*, na execução *single-core* a *benchmark* sofreu um *speedup* de cerca de 59%, como apresentado na Figura 5.3, tanto na forma Tradicional quando no Oracle.

Utilizando o conjunto de aplicações descrito em [17], mostrado na Tabela 5.3 realizou-se os testes em ambiente *multi-core*. Os resultados são apresentados na Figura 5.4. Nota-se que utilizando o EMC Tradicional, tem-se um *speedup* de no máximo 0,25%, com uma média de 0,09%. Utilizando o Oracle, o desempenho aproxima-se de 0,8% com média de 0,42%, resultados muito aquém dos relatados em [17].

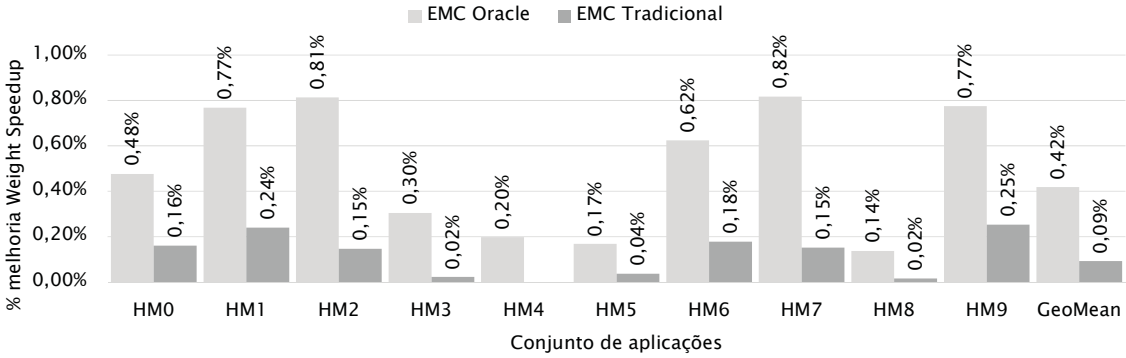


Figura 5.4: Melhoria no WS dos conjuntos de aplicações utilizados no EMC [17]

Em conjuntos de aplicações homogêneos, é relatado um ganho de desempenho levemente inferior aos conjuntos de aplicações aleatórios. Foram realizados testes com estes conjuntos de aplicações homogêneas, conforme presente na Figura 5.5. Foi obtido uma melhora no WS de no máximo 0,8% e na média 0,3% quando utilizado o EMC Tradicional e uma melhora no WS máximo de aproximadamente 2,3% e médio de aproximadamente 0,5% quando utilizado o EMC Oracle. Quando analisado a aplicação Synthetic, obseva-se uma melhoria no WS de cerca de 61%. Nota-se um comportamento inverso ao relatado em [17], onde a melhora era mais pronunciada quando utilizado o mecanismo em aplicações aleatórias.

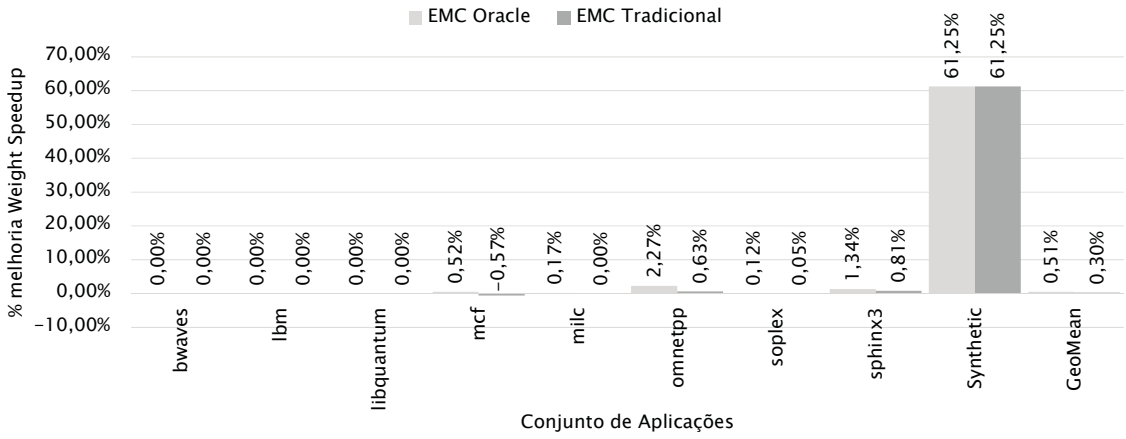


Figura 5.5: Melhoria no WS dos conjuntos de aplicações homogêneos utilizados no EMC [17]

Outro índice para avaliarmos à corretude na replicação em EMC é a latência média das requisições feitas a memória principal que são encaminhadas pelo EMC, e não realizadas pelo core. Em [17], é relatado uma diminuição média de 20% no tempo total das requisições a memória principal, quando comparadas as requisições feitas pelo core. A Figura 5.6 apresenta

as latências médias sofridas pelas requisições nos conjuntos de aplicações que apresentam alta intensidade de uso da memória. A Figura 5.7 apresenta as latências médias sofridas pelas requisições nos conjuntos de aplicações homogêneos. Nota-se que as aplicações *bwaves e lbm* não apresentam requisições a memória principal feitas pelo EMC, dado a ausência de instruções de *load* dependentes de cache misses. As aplicações *libquantum* e *milc* não apresentam requisições quando utilizado o mecanismo Tradicional, entretanto as apresentam quando utilizado o EMC Oracle.

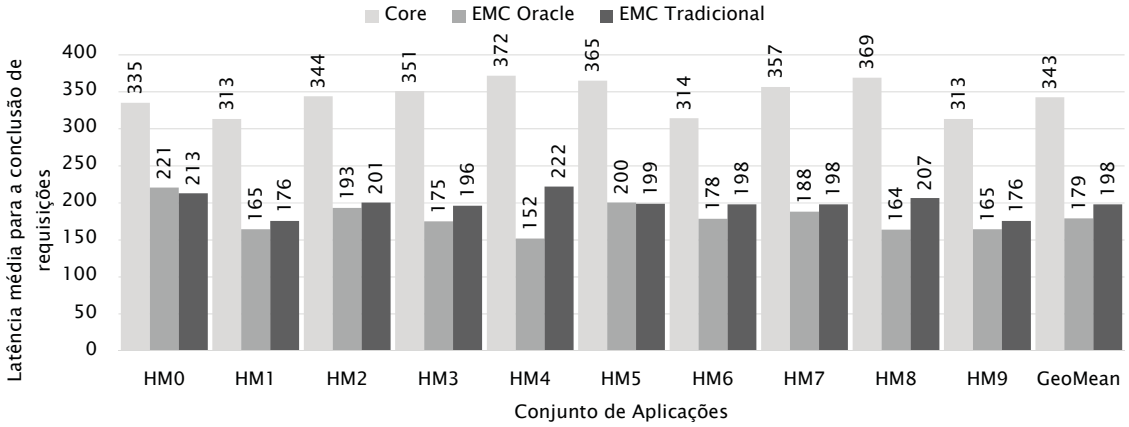


Figura 5.6: Latência média sofrida pelas requisições a memória quando feitas pelo core e pelo EMC nos conjuntos de aplicações de alta intensidade de uso da memória

As requisições feitas pelo EMC Tradicional apresentam latência média de cerca de 200 e 190 ciclos para os conjuntos de aplicações aleatórios e homogêneos, respectivamente, enquanto as requisições enviadas pelo Core apresentam uma latência média de 340 e 350 ciclos, também para os conjuntos de aplicações aleatórios e homogêneos,respectivamente. Isto resulta que as requisições feitas pelo EMC enfrentam cerca de 40% menos latência.

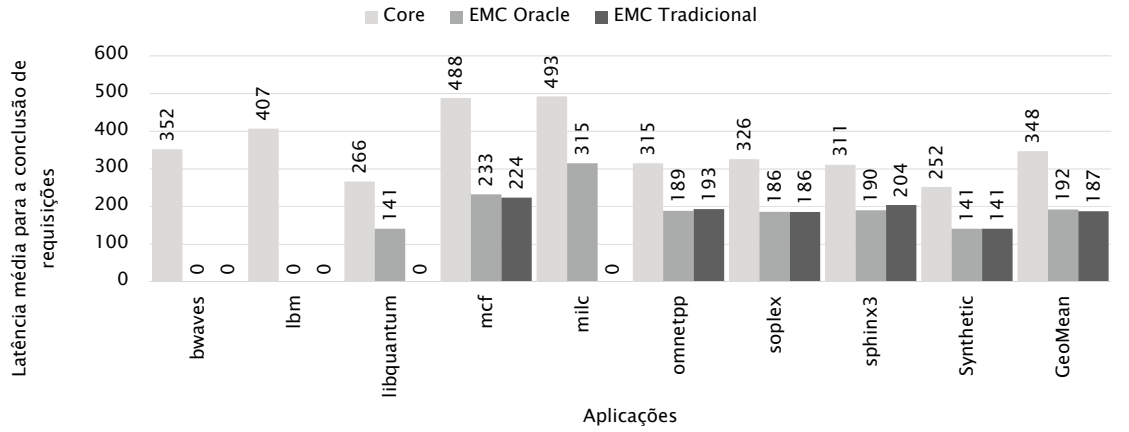


Figura 5.7: Latência média sofrida pelas requisições a memória quando feitas pelo core e pelo EMC nos conjuntos de aplicações homogêneos

Mesmo as requisições feitas pelo EMC enfrentando cerca de 40% menos latência, isso se traduziu em cerca de 0,5% de melhora no WS. Assim, selecionou-se entre as aplicações que apresentaram melhor desempenho quando simulado um cenário *single-core* para compor um novo conjunto de conjuntos de aplicações a serem executados. Estes conjuntos de aplicações estão dispostos na Tabela 5.6.

Tabela 5.6: Conjuntos de aplicações gerados a partir das aplicações com melhor desempenho no EMC *single-core*

MD0	astar+gcc+mcf+milc
MD1	astar+omnetpp+perlbench+xalancbmk
MD2	milc+omnetpp+perlbench+xalancbmk
MD3	mcf+milc+perlbench+xalancbmk
MD4	milc+perlbench+soplex+xalancbmk
MD5	astar+gcc+milc+perlbench
MD6	gcc+mcf+milc+xalancbmk
MD7	milc+omnetpp+perlbench+soplex
MD8	astar+gcc+omnetpp+perlbench
MD9	gcc+mcf+soplex+xalancbmk

Ao executar estes novos conjuntos de aplicações, notou-se um aumento no WS, entretanto este aumento não foi tão pronunciado. As Figuras 5.8 e 5.9 mostram o aumento no WS obtido. Ao se utilizar os conjuntos de aplicações gerados entre as aplicações que apresentaram a maior melhora no desempenho quando utilizado o EMC no cenário *single-core*, notou-se um ganho de desempenho máximo de aproximadamente 2,1% quando utilizado o EMC Oracle e de 0,5% quando utilizado o EMC Tradicional. O desempenho médio entretanto, apresenta-se em aproximadamente 1,4% e 0,2% respectivamente.

A mesma situação se repete quando analisados os conjuntos de aplicações homogêneos. Quando utilizado o EMC Oracle obtém-se um aumento no WS de no máximo 2,36% e médio de aproximadamente 0,8%. Já quando utilizado o EMC Tradicional, o aumento no WS máximo verificado é de 0,7% enquanto o aumento no WS médio verificado é de 0,26%. Vale ressaltar que o conjunto de aplicação homogêneo da aplicação *mcf* sofreu uma redução no desempenho de 0.57%.

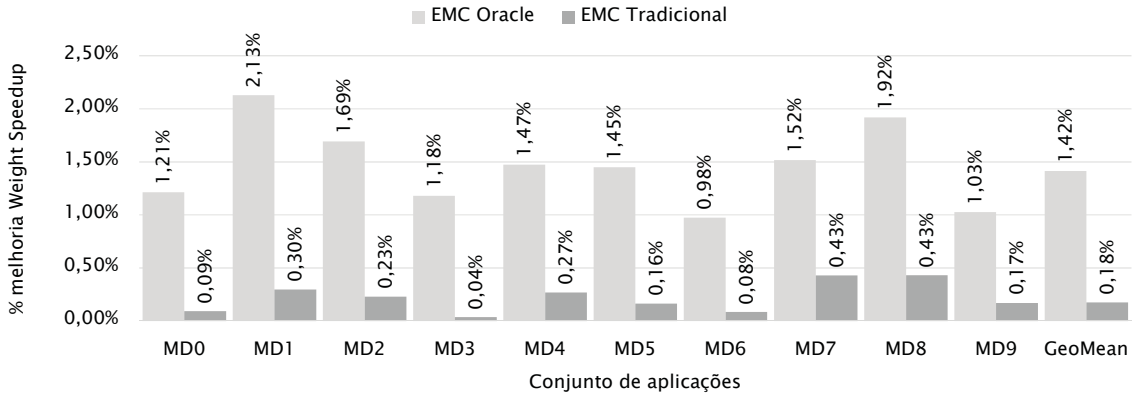


Figura 5.8: Incremento no WS dos conjuntos de aplicações gerados entre as aplicações que apresentaram melhor desempenho no *single-core*

Entretanto, deve-se ressaltar que a latência média encarada pelas requisições feitas pelo EMC manteve-se constante, mesmo com a alteração nos conjuntos de aplicações. As Figuras 5.10 e 5.11 mostram a latência média enfrentada pelas requisições. Nota-se que o comportamento é similar aos apresentados nas Figuras 5.6 e 5.7. As requisições enviadas pelo *core* enfrentam uma latência média de 294 e 339 ciclos para os conjuntos de aplicações aleatórios e homogêneos, enquanto que as requisições enviadas pelo EMC, tanto para os conjuntos aleatórios quanto os homogêneos enfrentam uma latência de cerca de 170 e 190 ciclos.

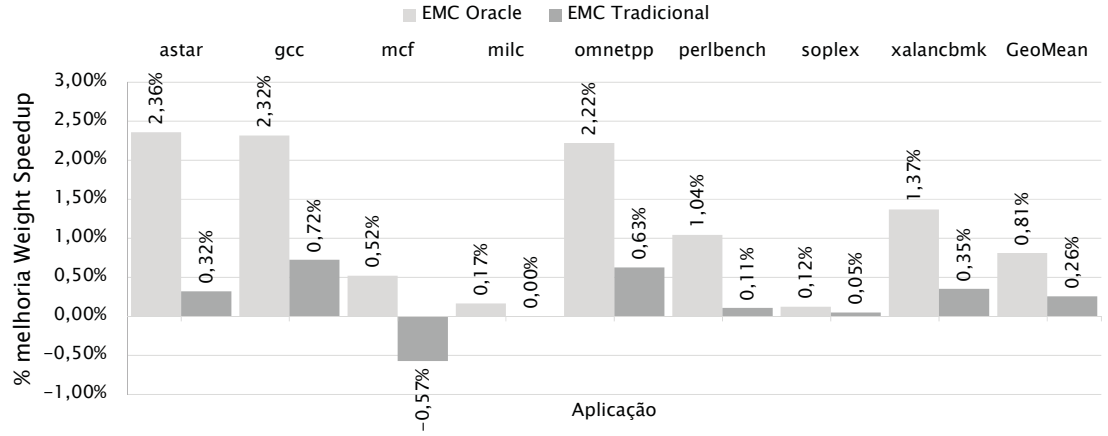


Figura 5.9: Incremento no WS dos conjuntos de aplicações homogêneos das aplicações que apresentaram melhor desempenho no *single-core*

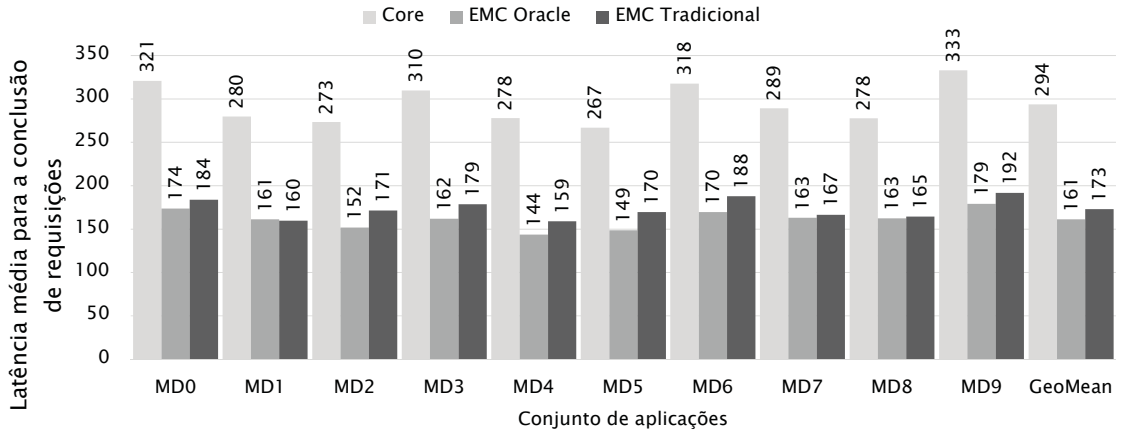


Figura 5.10: Latência média das requisições feitas a memória nos conjuntos de aplicações gerados com base nas aplicações que apresentaram melhor desempenho.

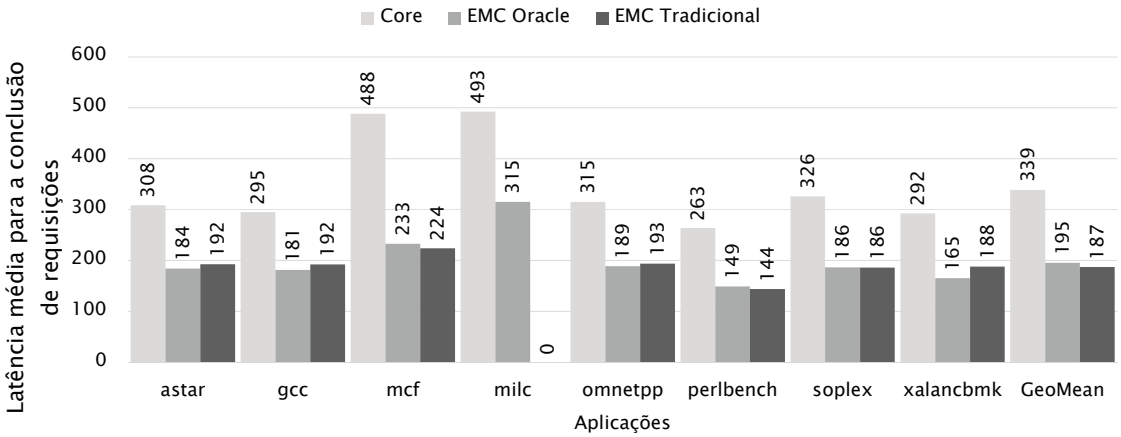


Figura 5.11: Latência média das requisições feitas a memória pelos conjuntos de aplicações homogêneas.

Entretanto, mesmo com as requisições feitas pelo EMC enfrentando cerca de 40% menos latência que as requisições feitas pelo Core, isto não se reflete no *speedup* obtido, atingindo no máximo 2%. Isto pode ser explicado pela proporção de instruções de *load* que executadas de cada aplicação no EMC. A figura 5.12 exibe a proporção de *loads* que foram enviados para

serem executados no EMC, das aplicações que apresentaram melhor desempenho no *single-core*, quando utilizado a versão EMC Oracle. O benchmark sintético apresenta uma taxa de envio de loads de 93,8%, enquanto que as outras aplicações apresentam no máximo 1,8%.

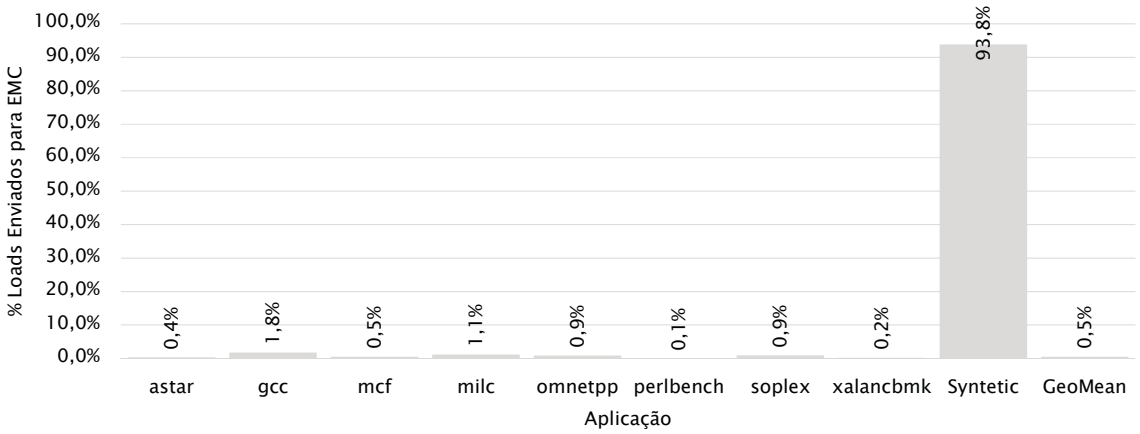


Figura 5.12: Percentagem de loads enviados ao EMC quando utilizando o oráculo.

Apesar de corresponderem a 0,5% em média, as requisições feitas pelo EMC obtêm vantagem devido ao fato de a requisição ser encaminhada diretamente a memória principal, evitando a hierarquia de cache. Isto é possibilitado por um preditor de misses da LLC, similar ao utilizado em [43]. Este preditor apresenta um bom desempenho, conforme ilustrado na Figura 5.13. O preditor apresenta uma precisão máxima e mínima de 99% e 96%, respectivamente. A precisão média é de 98%. Com base nesses dados, indagamos se ao mover este preditor para o core seria possível prever quando ocorreria-se um *miss* na LLC, e assim, seria possível encaminhar diretamente a requisição a memória principal. Os resultados desta investigação encontram-se no Capítulo 6.

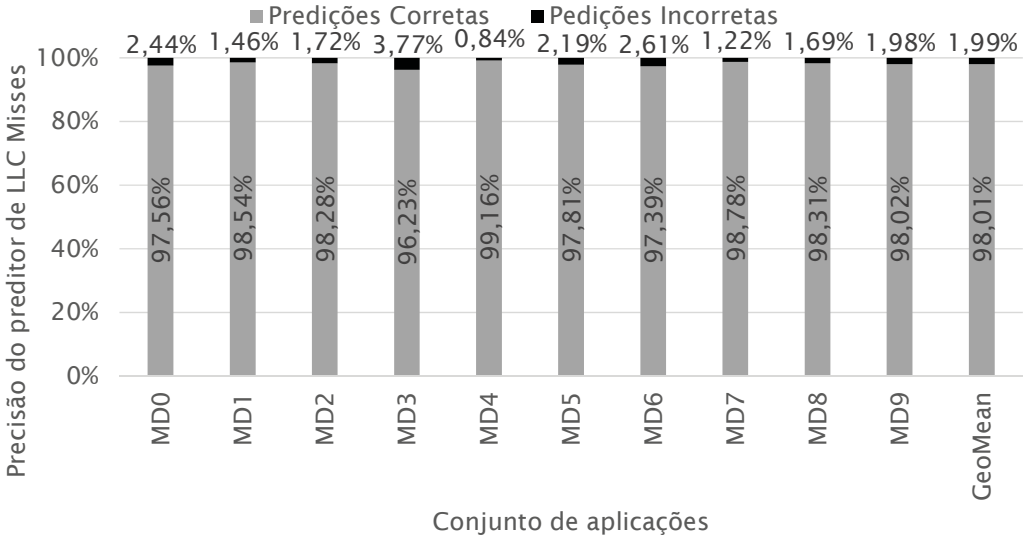


Figura 5.13: Precisão do preditor de misses da LLC no EMC

5.4 CONCLUSÕES

Neste capítulo foi apresentado a replicação do mecanismo EMC [17]. Verificou-se que o mecanismo apresenta ganho de desempenho real, porém é aquém do informado no artigo. Ao

contrário do apresentado no artigo, conclui-se que o mecanismo pode atingir maior desempenho em um cenário onde o conjunto de aplicações é homogêneo. O mecanismo de fato reduz o tempo médio de latência das requisições que são enviadas à memória principal, quando comparadas as requisições de dados que são enviadas a partir do *Core*. Suas requisições enfrentam cerca de 40% menos latência, até serem concluídas.

Contudo, devido a natureza de execução fora de ordem do processador, estimamos que tais ganhos em latência não se refletiram em ganhos reais de desempenho, a ser melhor investigado no futuro, mudando o processador para um modelo em-ordem para comprovar essa hipótese.

Ao mesmo tempo, ao replicar esse trabalho nos deparamos com um oportunidade, de mover o preditor de faltas de dados do LLC para dentro do núcleo de processamento. Assim, encaminhando em paralelo requisições para memória cache e memória principal, a fim de reduzir a latência de tais requisições.

6 REQUISIÇÕES PARALELAS CACHE/DRAM

Neste capítulo apresentamos a proposta de um mecanismo para habilitar o núcleo de processamento a realizar requisições diretamente para a DRAM, em paralelo com a hierarquia de cache. Na Seção 6.1, discutiremos as possibilidades de ganhos de desempenho que o uso de requisições paralelas quando acessada a memória principal podem permitir. Na Seção 6.2 apresentamos a arquitetura do mecanismo, e o seu modo de funcionamento. Na Seção 6.4 serão discutidos as simulações de um ambiente *single-core*. Posteriormente, na Seção 6.5 é explanada a avaliação feita sobre um sistema *multi-core*, sendo a Subseção 6.5.1 dedicada a apresentação dos resultados das simulações do mecanismo para os conjuntos de aplicações apresentados na Tabela 5.3. Na Subseção 6.5.2 serão discutidos os resultados das simulações dos conjuntos de aplicações exibidos na Tabela 5.4.

6.1 REQUISIÇÕES PARALELAS

Durante a execução das aplicações os dados serão reutilizados um número finito de vezes e posteriormente removidos da memória cache. Muitas vezes um dado endereço possui suas requisições vindas do programa concentradas em um curto espaço de tempo (alta localidade temporal). Nesse caso, tais requisições serão 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 dados pode constituir uma forma de melhorar o desempenho, uma vez que durante a busca de dados, a passagem pela LLC acarreta em tempo ocioso de espera pelos dados e poluição da cache [27].

Como forma de aumentar o desempenho e reduzir a poluição da hierarquia de cache, inserindo dados que possivelmente serão inúteis, pode-se adotar estratégias para realizar o *bypass* (contornar) toda ou parte da hierarquia de cache. Atualmente, podemos realizar o *bypass* de cache através de *software*, através de instruções específicas incluídas no código fonte. Instruções da Instruction Set Architecture (ISA) X86, tais como MOVNTDQA e MOVNTPD permitem que uma instrução busque/armazene dados sem que seja necessário a utilização da cache. Instruções deste tipo oferecem dicas para que os dados não sejam instalados nas caches, por exemplo, durante o processamento de vídeo ou de dados vindos das interfaces de rede, entretanto, a implementação destas dicas é dependente da implementação do processador, e assim, podem acabar sendo ignoradas [11].

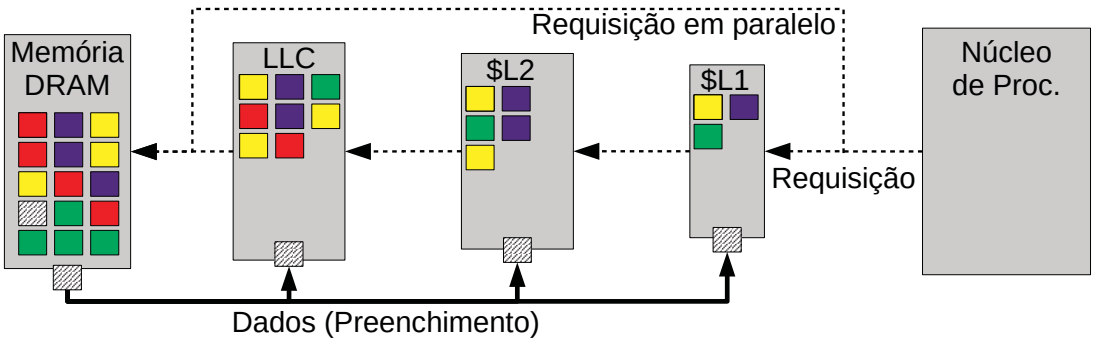
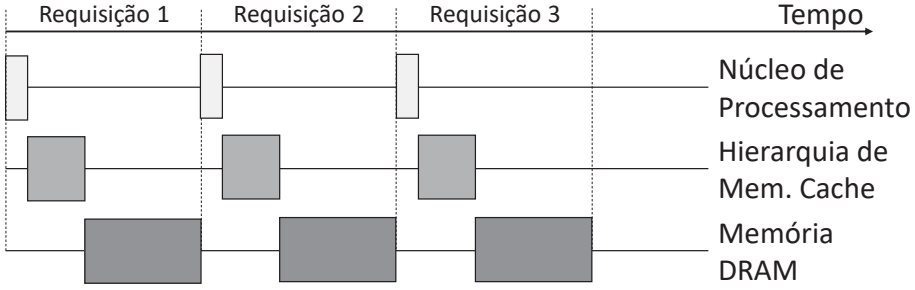


Figura 6.1: Requisições paralelas cache/DRAM com instalação tradicional de dados na hierarquia de cache.

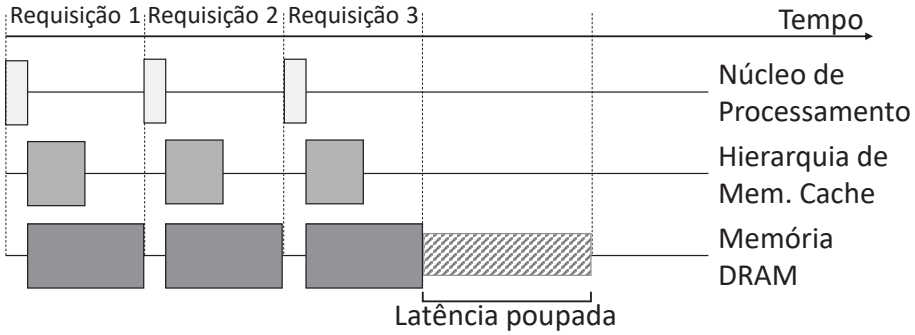
Por outro lado, ao realizar requisições paralelas (em memórias cache e memória DRAM), segue-se o caminho inverso ao *bypass* de dados, realizando uma emulação de um *bypass* de requisições. O uso de requisições paralelas permite que seja feita uma especulação acerca de dados que eventualmente não se encontram na memória cache. Desta forma, pode-se mascarar a latência de busca através da hierarquia de cache. A Figura 6.1 apresenta uma visão de alto nível acerca do fluxo de requisições paralelas em um processador.

Como ponto negativo, a criação inadvertida de requisições paralelas pode acarretar em congestionamento do barramento entre o processador e a memória DRAM. 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.

Para ilustrar os ganhos potenciais de nossa técnica, a Figura 6.2 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, na Figura 6.2a 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, na Figura 6.2b, quando se utilizando de requisições paralelas, após aprender o comportamento de acesso das requisições, torna-se possível encaminhar requisições 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 6.2: Fluxo de requisições para memória principal.

6.2 ARQUITETURA DO MECANISMO

Para habilitar o núcleo de processamento a enviar requisições de forma paralela para a memória cache e memória principal, se faz necessário um pedaço de lógica implementado juntamente ao núcleo de processamento. A Figura 6.1 apresenta uma visão do fluxo das

requisições em alto nível, e como o mecanismo se porta em relação a 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 DRAM-Caches [43]. DRAM-Caches são dispositivos integrados, localizados junto ao processador, construído através das tecnologias de integração 3D. Isto permite que se atinga uma largura de banda cerca de 8× maior, quando comparado a memória *off-chip*, atuando como uma cache de grande capacidade. Contudo, seu acesso é mais lento, pois usa a tecnologia DRAM, a mesma utilizada para a memória principal, em contraste com a tecnologia SRAM, utilizada tradicionalmente na cache *on-chip*. Desta forma, realizar acessos de forma sequencial a DRAM-Cache torna-se custoso, tanto em termos de latência quanto de largura de banda. Assim, deve-se decidir quando permitir que a requisição seja manipulada pela DRAM-Cache ou pela memória principal.

Para viabilizar esta decisão, é apresentado um preditor baseado em contadores com duas possíveis construções. A primeira construção, global, que mapeia as requisições que resultaram em acesso a memória principal, a um único contador. A segunda construção utiliza um preditor baseado em instruções, correlacionando os misses da LLC com as instruções que os causaram, sendo para isso utilizado diversos contadores independentes. No artigo original, estes contadores são organizados em uma tabela de 256 entradas chamada de Memory Access Counter Table (MACT), sendo esta quantidade de entradas considerada suficiente. A tabela é indexada através do endereço da instrução (PC/IP) causadora do miss.

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.

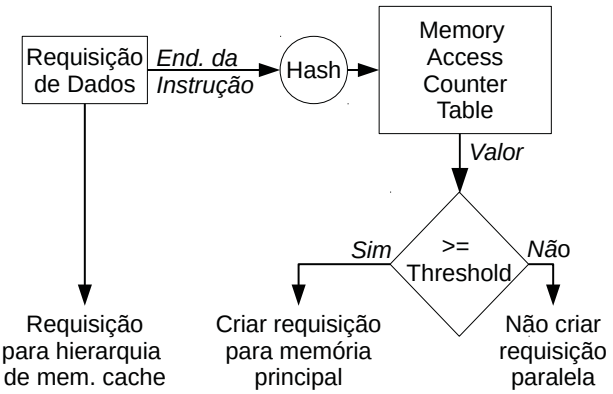


Figura 6.3: Fluxo de requisições com o mecanismo responsável por executar as requisições paralelas.

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

Toda requisição feita para a hierarquia de memória cache, possui um bit que indica se foi despachada uma requisição paralela a ela (direto para a memória principal). Caso ocorra a criação de uma requisição em paralelo, a requisição que percorreu a hierarquia de cache, ao

chegar ao controlador de memória pode ser descartada. Evitando dessa maneira a sobrecarga no controlador de memória.

6.3 SETUP

Para a avaliação das simulações, utilizou-se a mesma metodologia apresentada na seção 5.1. O simulador foi configurado conforme o descrito na seção 5.2, utilizando o disposto na Tabela 5.1 para o sistema de um núcleo de processamento. Para cada aplicação do SPEC CPU-2006 foram simuladas as 200 milhões de instruções mais representativas obtidas através do método PinPoints [41].

O consumo de energia foi modelado utilizando o *software* CACTI 6.5++ [37]. O consumo dos núcleos de processamento foram estimados com base no Thermal Design Power (TDP) apresentado pelo processador (65 Watts) [22] e pelo tempo de execução apresentado pela aplicação/conjunto de aplicações.

Cada acesso a memória principal é contabilizada como sendo executadas as operações de *Row Precharge*, *Row Access Strobe* e *Column Access Strobe*. O consumo total do sistema é dados pela soma dos consumos dos núcleos de processamento e da memória principal. Note que o consumo da energia da hierarquia de cache não sofre alterações, devido ao mecanismo não alterar o seu funcionamento em relação ao *baseline*.

6.4 AVALIAÇÃO DO MECANISMO EM AMBIENTE SINGLE-CORE

A Figura 6.4 apresenta a melhoria no IPC obtido para cada aplicação do SPEC CPU-2006 . Entre todas as aplicações, quando utilizado o mecanismo para a execução de requisições paralelas, apresenta-se um ganho médio de 3,18%, entre todas as 29 aplicações do SPEC CPU-2006 . Este valor de melhoria oscila entre o mínimo de 0,05% e máximo de 40%. Mesmo entre aplicações cuja intensidade de uso da memória é baixa (Misses per Kilo Instructions (MPKI) < 5), apresentou-se um leve aumento no IPC. Em aplicações cuja intensidade de memória é alta (MPKI > 10), apresentou-se uma melhoria média no IPC de 10%.

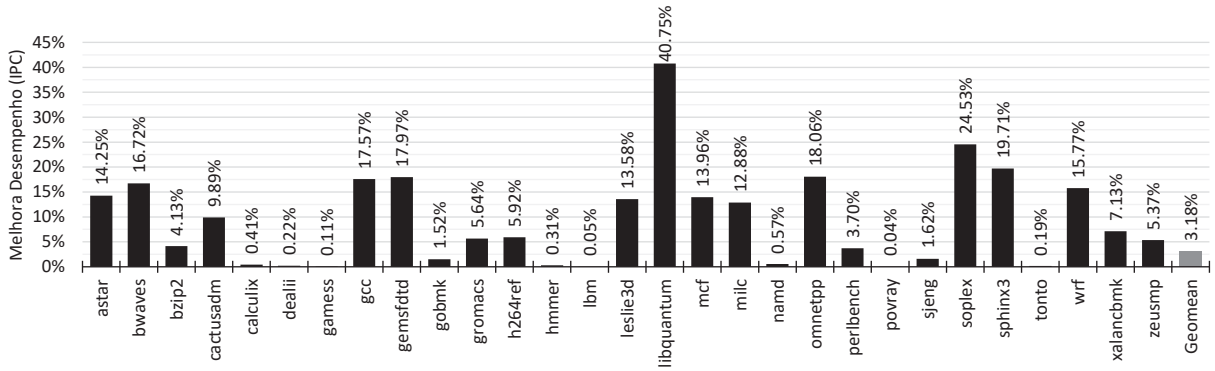


Figura 6.4: Melhoria de IPC para as aplicações SPEC CPU-2006 utilizando requisições paralelas.

Um caso interessante vem a tona ao analisarmos o comportamento da aplicação *lbm* (MPKI = 31,8). Apesar de ser uma aplicação com alta intensidade de memória, apresenta uma melhoria no IPC de somente 0,05%. Isso ocorre devido ao comportamento de acesso a memória do programa. Por utilizarmos uma política de *write-allocate*, sempre o endereço para uma escrita não encontra-se na hierarquia de cache, é realizada uma requisição de busca de dados, para que então a escrita seja efetivada. Ao observar o comportamento dos acessos a LLC para

esta aplicação, nota-se que 75% dos acessos que chegam ao último nível correspondem a uma requisição de escrita, e sendo assim, não são capturados pelo mecanismo.

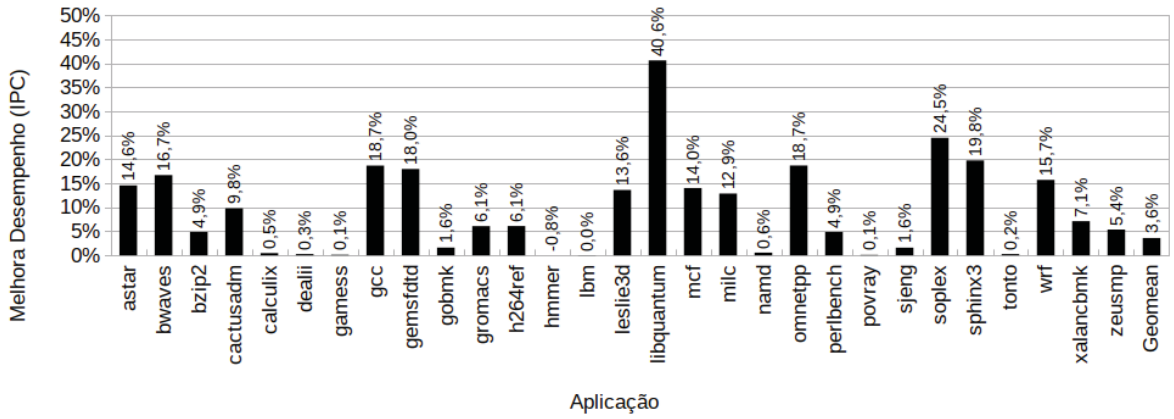


Figura 6.5: Melhoria de IPC para as aplicações SPEC CPU-2006 utilizando requisições paralelas com um oráculo.

6.4.1 Considerações sobre o consumo de energia

Quando considerado o uso de requisições paralelas, deve-se levar em conta que são criadas duas requisições ao invés de uma, e assim usa-se mais energia para que um dados seja obtido. Assim, temos dois cenários distintos, onde podemos analisar o consumo energético de ambos, processador e memória principal. A Figura 6.6 apresenta o consumo estimado para ambos os componentes, processador e a memória principal.

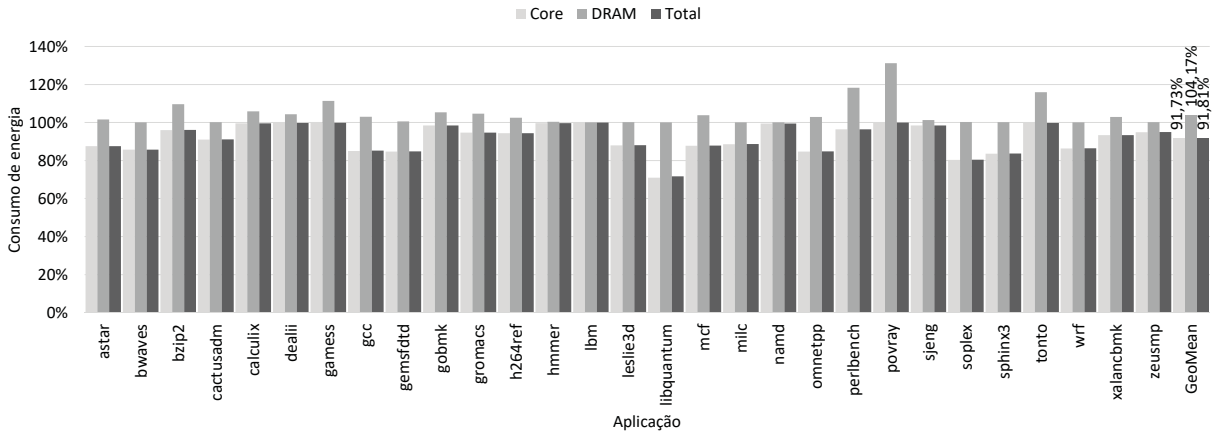


Figura 6.6: Variação de uso de energia no sistema ao utilizar requisições paralelas

Considerando somente a energia gasta pelo *core* durante o processamento, a utilização de requisições paralelas apresenta um consumo de energia de até 28% durante a execução, com uma média de economia de 8,3%. Contudo, ao observar o consumo de energia da DRAM, nota-se que houve um aumento no consumo. O consumo de energia da DRAM aumentou em até 31%, com uma elevação no consumo médio da DRAM de 4,2%. Este aumento no consumo é devido as requisições que são previstas erroneamente, gerando um acesso a DRAM para uma requisição que é satisfeita pela hierarquia de cache. Apesar desta elevação no consumo da DRAM, o consumo total dos sistema, considerando todas as 29 aplicações do SPEC CPU-2006 , apresentou uma redução de cerca de 8,2% no consumo de energia. Isso se deve ao pouco impacto

que as requisições enviadas de forma incorreta representam, frente ao consumo estimado do núcleo de processamento.

6.4.2 Latência média das requisições em *single-core*

Considerando o uso de requisições em paralelo para a hierarquia de cache e para a DRAM, espera-se que ao requisitar os dados diretamente para a DRAM, quando os dados não se encontram na hierarquia de cache, a latência seja reduzida. A figura 6.7 apresenta as latências médias para todas as aplicações do SPEC CPU-2006 . A latência média das requisições para a DRAM decresceram cerca de 21,7%, devido ao fato de contornar a hierarquia de cache. Em um ambiente *single-core*, dado o controle sobre a totalidade dos recursos do processador e da memória DRAM por parte de um único programa, a economia de ciclos é bem próxima ao total de ciclos que são interpostos pela hierarquia de cache. Entre todas as aplicações do SPEC CPU-2006 , 22 apresentam uma economia de ciclos maior que 50, valor próximo ao interposto pela hierarquia de cache, de 56 ciclos.

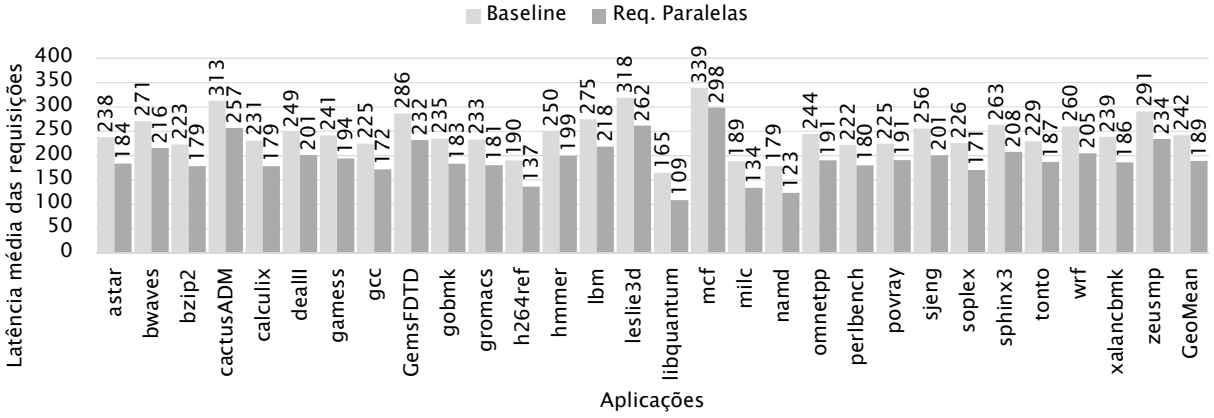


Figura 6.7: Variação de uso de energia no sistema ao utilizar requisições paralelas em um ambiente *single-core*

6.5 AVALIAÇÃO DO MECANISMOS EM AMBIENTE MULTI-CORE

Ao verificar os resultados da avaliação do mecanismo de requisições paralelas em um ambiente *single-core*, e tendo este mecanismo mostrado-se de uso promissor, passou-se a avaliação do mecanismo em um ambiente multi-core. A utilização em ambientes *multi-core* é mais desafiadora, dado que a competição pelos recursos do sistema é maior. O simulador novamente foi configurado conforme a Tabela 5.1 para um sistema dotado de quatro núcleos de processamentos. Para cada aplicação do SPEC CPU-2006 foram simuladas as 200 milhões de instruções mais representativas obtidas através do método PinPoints [41]. Cada aplicação presente no conjunto completou suas 200 milhões de instruções antes da simulação ser encerrada. O desempenho das aplicações foram avaliadas através da métrica de Weight Speedup (WS).

6.5.1 Avaliação dos conjuntos de aplicação do EMC

Embora os conjuntos de aplicações apresentados na Tabela 5.3 foram criados para o uso durante os testes do mecanismo Enhanced Memory Controller (EMC), o mecanismo de requisições paralelas objetiva ser genérico, de forma que possa ser utilizado em qualquer situação. Assim, como parte da avaliação do mecanismo, foram utilizados os mesmos conjuntos apresentados em [17].

A Figura 6.8 apresenta os dados para os conjunto de aplicações aleatórias, enquanto que a Figura 6.9 apresenta os dados para os conjunto de aplicações homogêneas.

Sendo avaliado em dois cenários, um com o conjunto de aplicações selecionadas de forma randômica e outro com o conjunto de aplicações apresentando homogeneidade, o conjunto de aplicações randômicas apresenta uma melhoria no WS máximo de 10,7% e uma melhoria média de 7,4%. Quando executado o conjunto de aplicações homogêneas, mostrou-se uma melhoria no WS máxima de 13,4% uma melhoria média de 7,4%. Contudo, as aplicações *libquantum* e *lbm* apresentam uma redução no WS. Isso se deve ao fato de que ao se realizar requisições em paralelo, acaba-se por congestionar o barramento e aumentar a pressão sobre a memória principal. Esse impacto apresenta-se de forma bem pronunciada na aplicação *libquantum*, pois quando executando-se em ambiente *single-core* apresenta uma melhoria no IPC de 40,7%, enquanto que ao ser executado em um ambiente *multi-core* homogêneo apresenta uma redução de 2,7%.

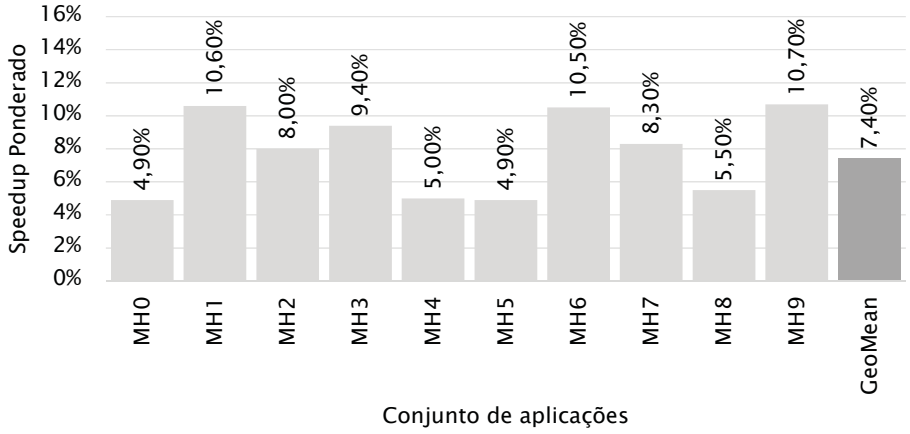


Figura 6.8: *Speedup* para *workloads* aleatórios apresentados no trabalho EMC [17]

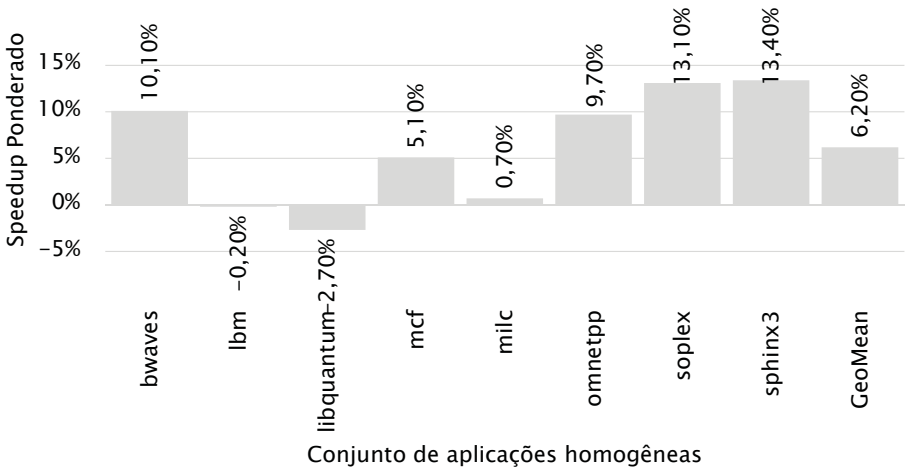


Figura 6.9: *Speedup* para os *workloads* homogêneos apresentados no trabalho EMC [17]

6.5.2 Avaliação dos conjuntos de aplicação para mecanismo requisições paralelas

Dada as diferenças encontradas na classificação das aplicações quanto a intensidade da utilização, foram gerados novos conjuntos de aplicações. Estes conjuntos encontram-se na Tabela 5.4.

Na Figura 6.10 apresentam-se os resultados de quando o mecanismo proposto é utilizado juntamente com os conjuntos de aplicações de alta intensidade de memória. Para estes conjuntos de aplicações atingiu-se um melhora média no WS de 9,8%, com máxima de aproximadamente 12,8%.

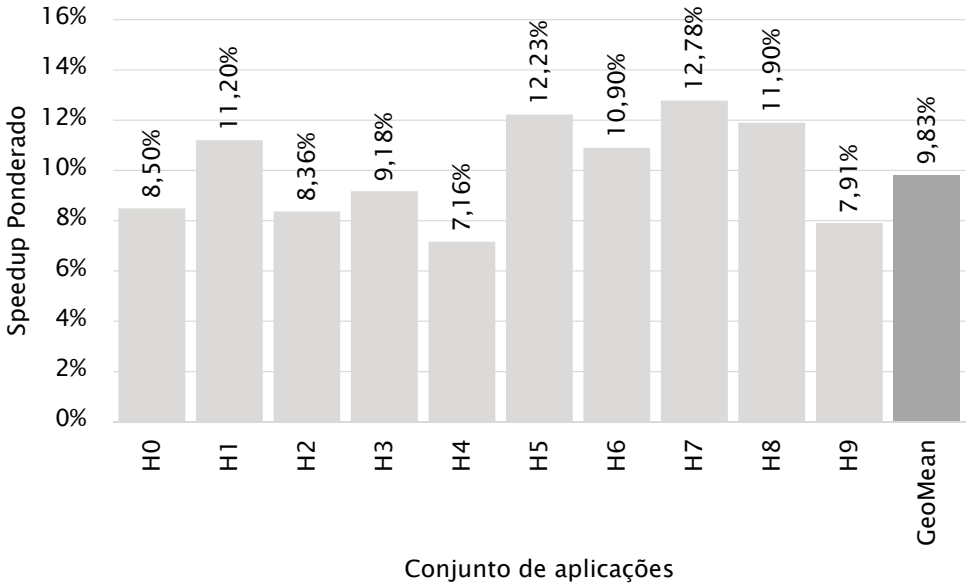


Figura 6.10: Incremento do WS para conjuntos de aplicações com alta intensidade de memória

Para os conjuntos de aplicações de média intensidade, a melhoria média no WS apresentado é de 9,5%, atingindo um máximo de 11,3%. Em conjuntos de aplicações de baixa intensidade, a melhoria média é de 9,2% enquanto que o máximo apresentado é de 10,3%. Os gráficos podem ser vistos nas Figuras 6.11 e 6.12, para os conjuntos de aplicações de média e baixa intensidade, respectivamente.

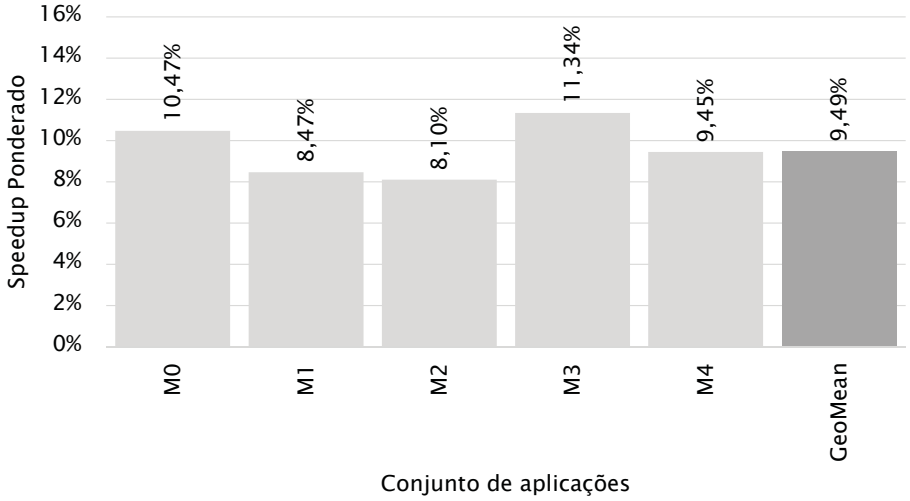


Figura 6.11: Incremento do WS para conjuntos de aplicações com média intensidade de memória

Avaliando o desempenho do mecanismo em conjuntos de aplicações homogêneos, o mecanismo apresenta um aumento de desempenho médio de 8,5%. A aplicação sphinx3 apresenta o maior ganho, com 13,3%, enquanto que a aplicação libquantum apresenta uma degradação de 2,7%. A Figura 6.13 mostra o WS apresentado.

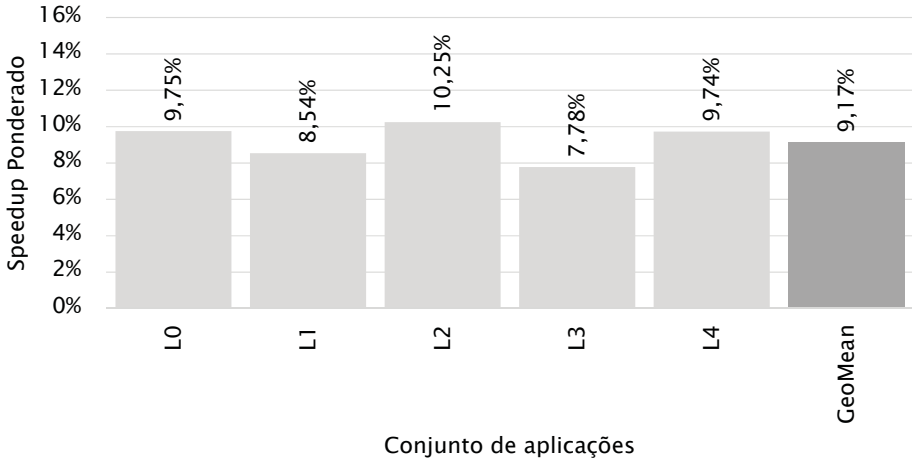


Figura 6.12: Incremento do WS para conjuntos de aplicações com baixa intensidade de memória

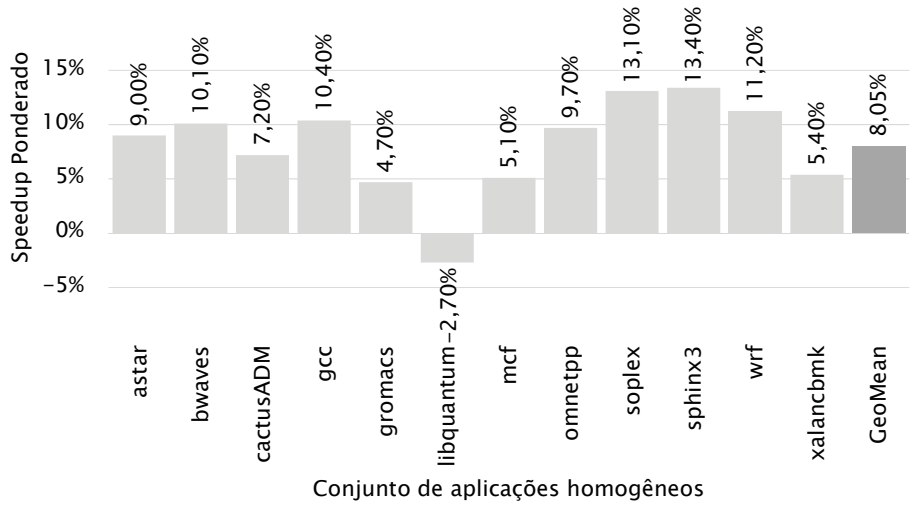


Figura 6.13: Melhoria no WS para conjuntos de aplicações homogêneos

Dentre os conjuntos de aplicações apresentados, sejam de alta, média ou baixa intensidade, onde estão presentes as aplicações *soplex* e *sphinx3* são observados os maiores ganhos, ao passo que os conjuntos de aplicações onde está presente as aplicações *libquantum* e *mcf* a melhora no WS é menor.

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 memória cache. Para além deste balanceamento, a precisão do preditor durante os LLC misses é preponderante para um bom desempenho. A Figura 6.14 mostra a precisão do mecanismo durante a execução das aplicações que compõe os conjuntos. Ao analisar a precisão do mecanismo, dois casos chamam a atenção.

O primeiro caso é na aplicação *libquantum*. A precisão do mecanismo fica próximo a 100%, ocasionando requisições paralelas em quase que a totalidade dos acessos a DRAM, e isto resulta, ao ser executado em um ambiente *single-core*, com controle total dos recursos, um incremento de aproximadamente 40% no IPC, mas ao ser executado em um ambiente *multi-core*, acaba por contribuir significativamente para o congestionamento do barramento. O outro caso é verificado na aplicação *mcf*, onde a precisão do mecanismo não é tão alta, ocasionando requisições errôneas, que se interpõe as requisições legítimas, aumentando a pressão na memória principal, com requisições que são satisfeitas pela hierarquia de cache. Tal precisão é similar a mostrada

pela aplicação *astar*, porém quando comparado o índice de MPKI das duas aplicações (astar = 6.9 X mcf = 60.9), isto faz com que uma predição incorreta na aplicação *mcf* acabe se tornando muito mais custosa quando comparada a aplicação *astar*.

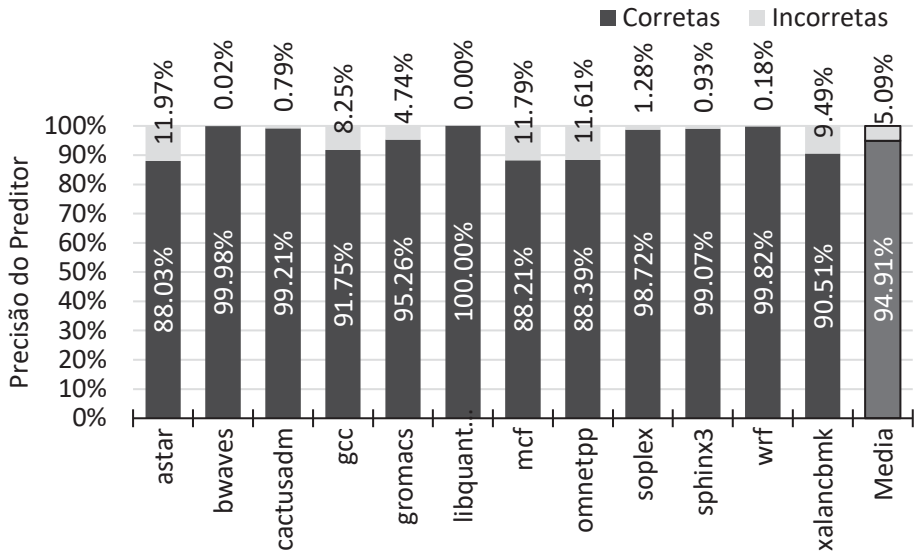


Figura 6.14: Precisão do preditor de misses na LLC e execução de requisições paralelas em cache misses

6.5.3 Avaliação do consumo de energia

Ao analisar o gasto energético do sistema multi-core, foram considerados os três cenários distintos simulados. Inicialmente foi verificado o consumo simulando os conjuntos de aplicações desenvolvidos para o EMC. Posteriormente foi verificado o consumo de energia para os conjuntos de aplicações homogêneos e para os conjuntos de aplicações desenvolvidos com base na Tabela 5.2.

Quanto aos conjuntos de aplicações apresentados no trabalho EMC, os resultados de consumo de energia das aplicações geradas de forma randômicas são apresentados na Figura 6.15. Quando considerado somente o consumo dos núcleos de processamento, tem-se uma redução média no consumo de 8,3%. O consumo da DRAM apresentou um ligeiro aumento, exprimindo um aumento de 2,1% no consumo. Contudo, este leve aumento não afeta de forma significativa a redução do consumo total do sistema, que apresenta-se em 8,2%.

Ao observar o consumo de energia para os conjuntos de aplicações homogêneos, apresentados na Figura 6.16, verifica-se uma redução de até 10% no consumo, com uma redução média de 5,6%. Ao verificar-se o consumo de energia da DRAM, este ficou praticamente idêntico ao *baseline*, apresentando um aumento no consumo de energia de 0,7% na média. Desta forma, o consumo total do sistema apresentou uma redução no consumo médio de aproximadamente 5,6%. Deve-se ressaltar que as aplicações *lbm* e *libquantum* não apresentaram redução no consumo de energia nos núcleos de processamento, e sim um aumento, de 0,2% e 2,8% respectivamente. Quanto ao consumo da memória, a aplicação *lbm* apresentou um decréscimo no consumo, de 0,5%, enquanto que para a aplicação *libquantum* o consumo manteve-se estável. Isto levou a um aumento no consumo de energia total 0,2% e 2,8%, respectivamente.

Para as aplicações geradas com base na classificação apresentada na Tabela 5.2, os resultados de consumo de energia apresentam um padrão parecido. A Figura 6.17 apresenta os resultados obtidos quanto ao consumo de energia para os conjuntos de aplicações de alta intensidade, e Figura 6.18 apresenta os resultados obtidos quanto ao consumo de energia para os

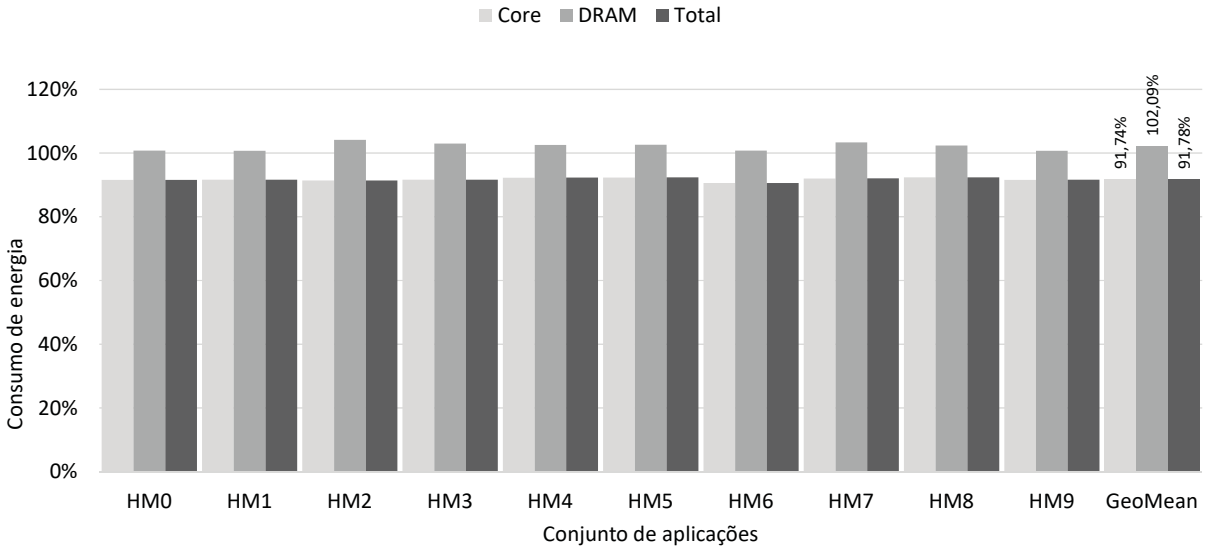


Figura 6.15: Variação no consumo de energia para os conjuntos de aplicações randômicas apresentados em [17]

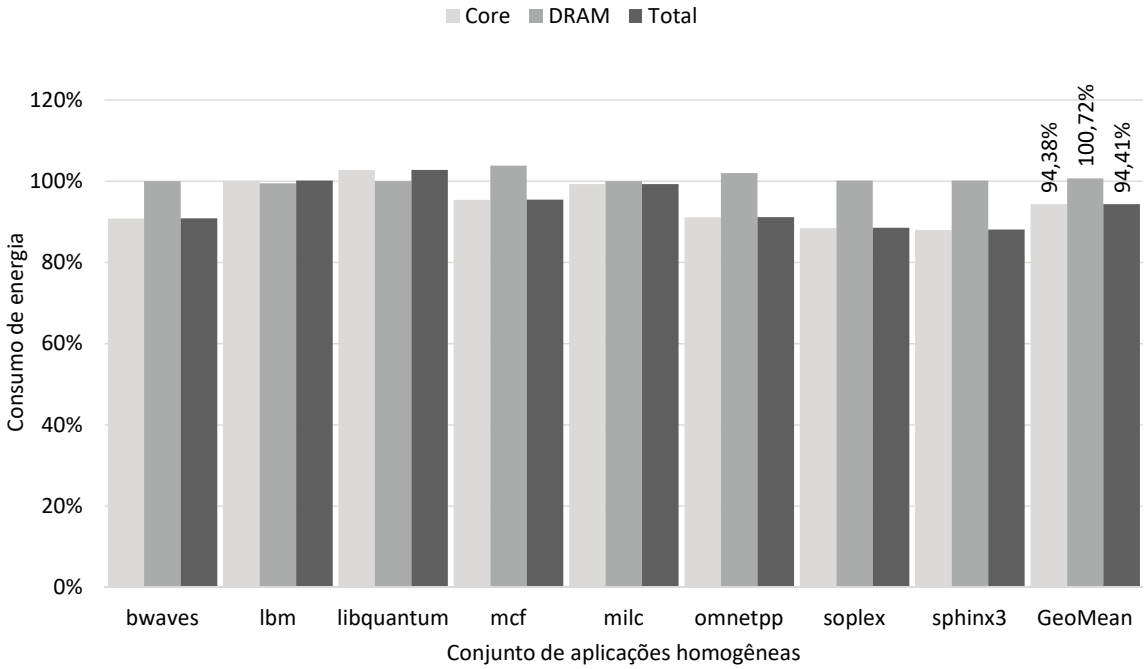


Figura 6.16: Variação no consumo de energia para os conjuntos de aplicações homogêneas apresentados em [17]

conjuntos de aplicações de média e baixa intensidade de uso de memória. O consumo de energia nos núcleos de processamento, quando analisado de forma isolada, apresenta uma redução média no consumo de energia de 10%, 10,3% e 13,1%, para os conjuntos de aplicações de alta, média e baixa intensidade. Quando analisado o consumo de energia da memória DRAM, verifica-se que o consumo de energia apresenta um incremento médio de 1,2% para as aplicações de alta intensidade de memória. Este aumento tem pouco impacto no consumo total do sistema, que apresenta uma redução de 9,9% no consumo de energia.

Para as aplicações de médio e baixo nível de intensidade de memória, o aumento no consumo médio de energia da DRAM apresentam um incremento de 1,5% e 1,8%. Este aumento no consumo de energia da DRAM reflete em uma redução de 10,2% no consumo total de energia para os conjuntos de média intensidade, e 13% no consumo total de energia para os conjuntos de baixa intensidade. Nota-se que a redução no consumo de energia é mais acentuado nos conjuntos

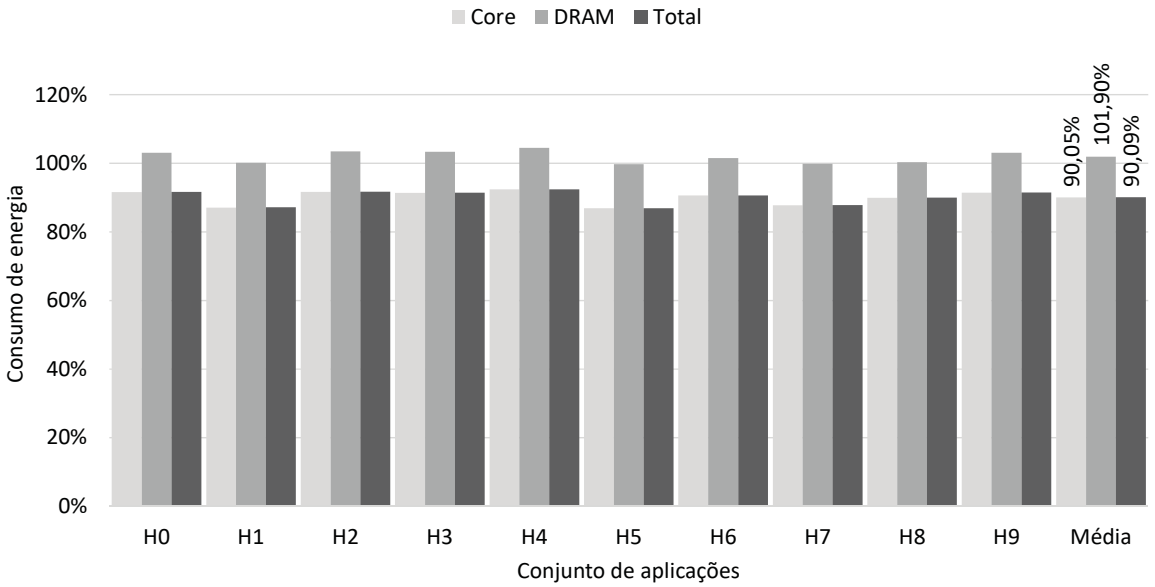


Figura 6.17: Variação no consumo de energia para os conjuntos de aplicações randômicos de alta intensidade de memória geradas para o mecanismo de requisições paralelas

de aplicações de baixa intensidade. Isso se deve ao modo como o conjunto foi construído. Dada a mescla de duas aplicações de alta intensidade e duas de baixa intensidade, isto reduz a concorrência pelos recursos compartilhados do sistema, o que acaba favorecendo as aplicações de alta intensidade neste caso, quando comparado aos conjuntos de aplicações de alta e média intensidade.

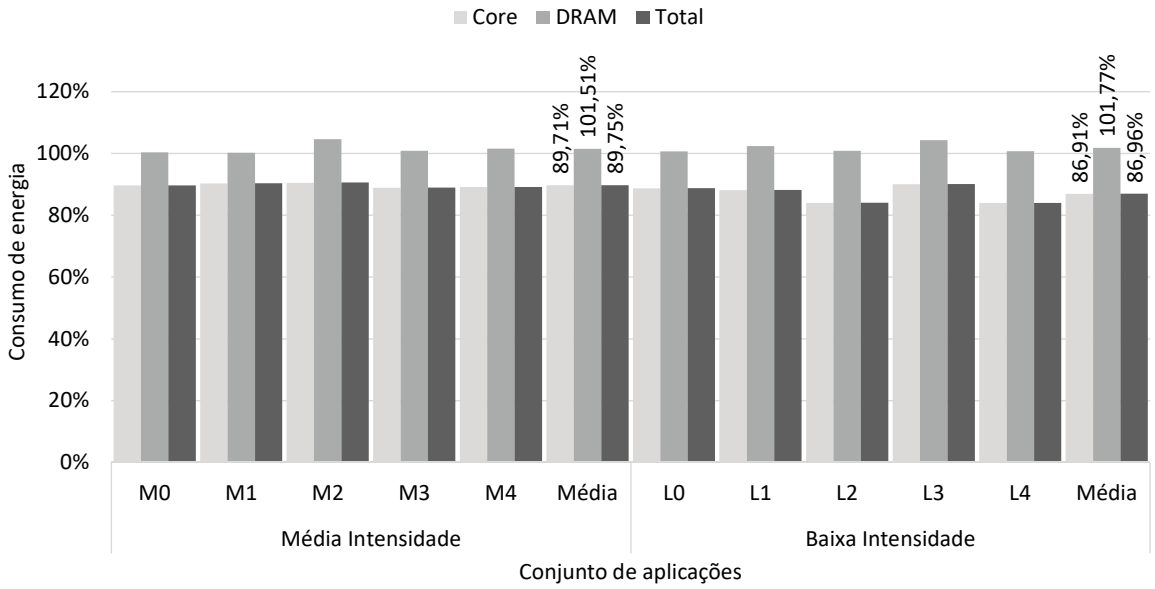


Figura 6.18: Variação no consumo de energia para os conjuntos de aplicações randômicos de média e baixa intensidade de memória geradas para o mecanismo de requisições paralelas

Ao observar o consumo de energia dos conjuntos de aplicações homogêneos, os núcleos de processamento apresentaram uma redução no consumo de energia em todas os conjuntos de aplicações, com exceção do conjunto composto pela aplicação *libquantum*. Desta forma, o consumo médio nos núcleos decresceu cerca de 7,4%. Verificando o consumo da memória principal, este apresentou-se consistente com os outros experimentos, mostrando um leve aumento,

com um incremento no consumo de cerca de 1,7%. Assim, o consumo total de energia observado expõe uma redução de 7,3% nos conjuntos de aplicações homogêneas.

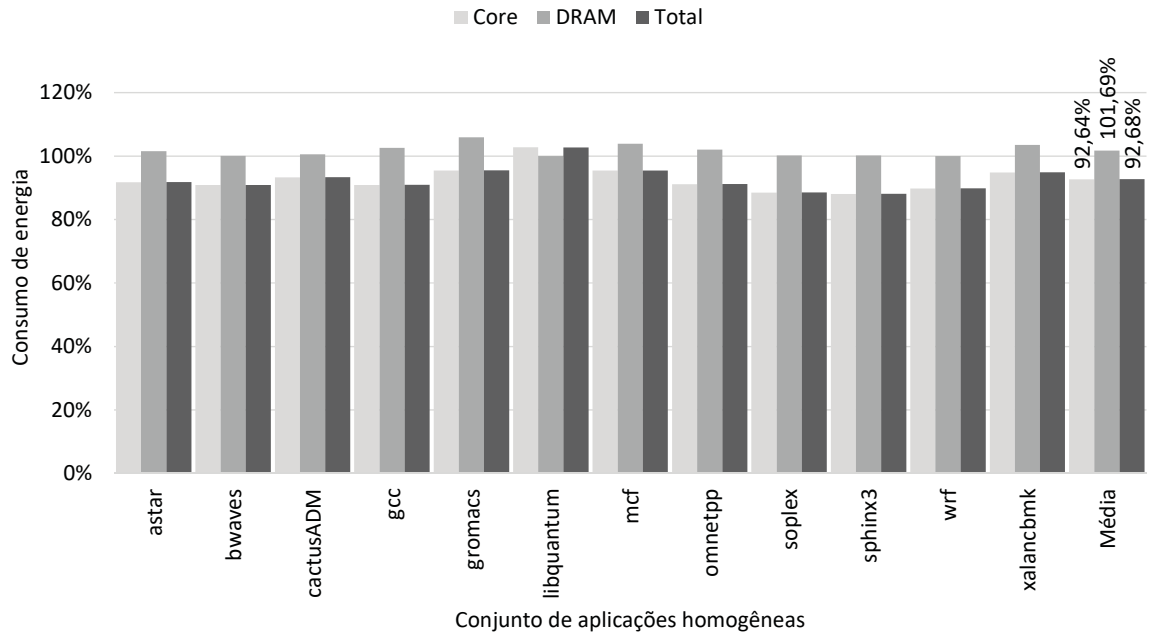


Figura 6.19: Variação no consumo de energia para os conjuntos de aplicações homogêneos geradas para o mecanismo de requisições paralelas

6.5.4 Latência média das requisições em *multi-core*

Ao se utilizar das requisições paralelas para acelerar as requisições, espera-se uma queda na latência média. Sendo assim, foram medidas as latências médias das requisições para a DRAM, durante a execução dos conjuntos, tanto os apresentados em [17], quanto os conjuntos criados para a verificação do uso de requisições paralelas. A figura 6.20 apresenta a média da latência de todas as requisições, quando utilizado o mecanismo de geração de requisições paralelas, na execução dos conjuntos de aplicações homogêneos apresentados em [17].

Observa-se que a latência média das requisições apresenta queda em todas as instancias. Na média, as requisições apresentam uma redução de cerca de 11,6% na latência total. O mesmo comportamento pode ser verificado na figura 6.21, onde são exibidas as latências das requisições que são feitas para a DRAM usando as requisições paralelas. A redução na latência média é de cerca de 13,5%.

Posteriormente foi verificada a latência das requisições dos conjuntos de aplicações desenvolvidos com base na tabela 5.2. As latências dos conjuntos de aplicações de alta intensidade estão dispostos na figura 6.22. Pode-se verificar que a latência média das requisições é de cerca de 281 ciclos, o que representa cerca de 14,3% na latência consumida pelas requisições.

O mesmo comportamento se mantem quando observados os conjuntos de aplicações de média e baixa intensidade, bem como os conjuntos de aplicações homogêneas, dispostos nas figuras 6.23 e 6.24. Para os conjuntos de aplicações de média intensidade, a redução apresentada na latência média é de cerca de 15,5% enquanto que os conjuntos de aplicações de baixa intensidade apresentam uma redução de 14,7%. Os conjuntos de aplicações homogêneas apresenta uma redução na latência média de cerca de 13,3%.

Em termos de ciclos para que uma requisição seja completa, a média de ciclos poupados durante as requisições, quando utilizada de requisições paralelas é de 42 e 46 ciclos, quando

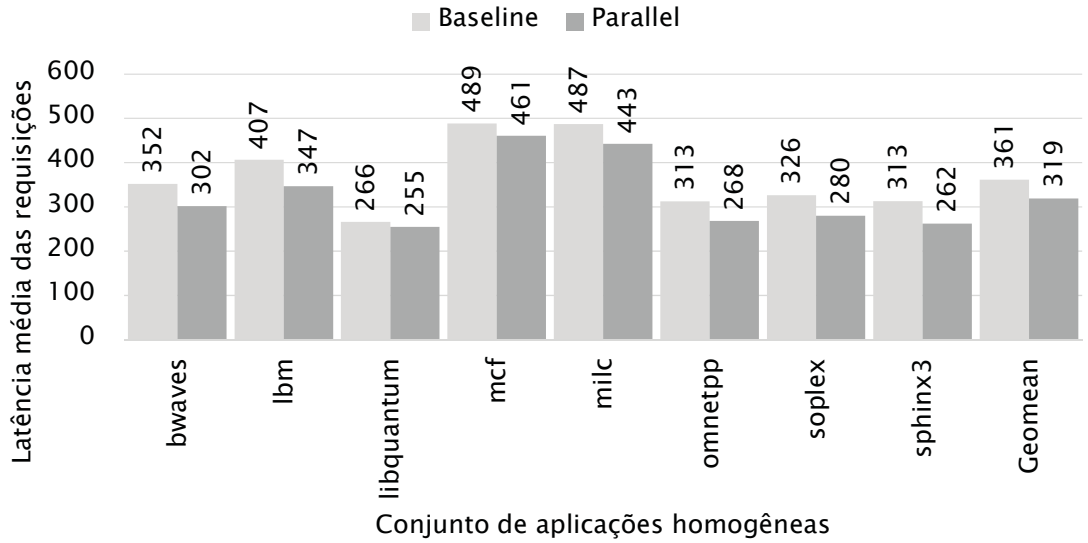


Figura 6.20: Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações homogêneos apresentados em [17]

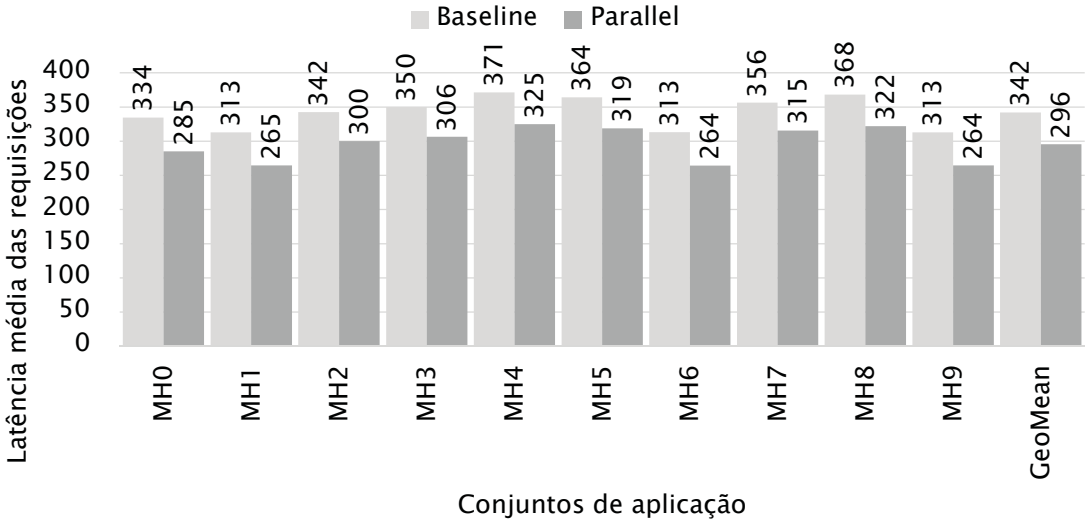


Figura 6.21: Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias apresentados em [17]

utilizadas o conjunto de aplicações aleatórias e homogêneas, respectivamente, apresentada em [17]. Quando utilizadas as aplicações desenvolvidas com base na tabela 5.2, em média as requisições economizam 47 ciclos, para aplicações de alta intensidade de memória, enquanto que as requisições de média intensidade apresentam uma redução de 51 ciclos. As aplicações de baixa intensidade apresentam uma redução média de 48 ciclos, enquanto que os conjuntos de aplicações homogêneas apresentam uma redução média de 42 ciclos para que as requisições sejam completadas.

Estas reduções são compatíveis com o número de ciclos que cada é adicionado a cada cache miss, em uma hierarquia de cache multinível. No caso da hierarquia de cache utilizada, onde as latências são de 3, 9 e 44 ciclos (L1, L2, LLC), totalizando 56 ciclos, reduz-se o contingenciamento da hierarquia de cache nas requisições que se encaminham para a DRAM para no máximo 14 ciclos.

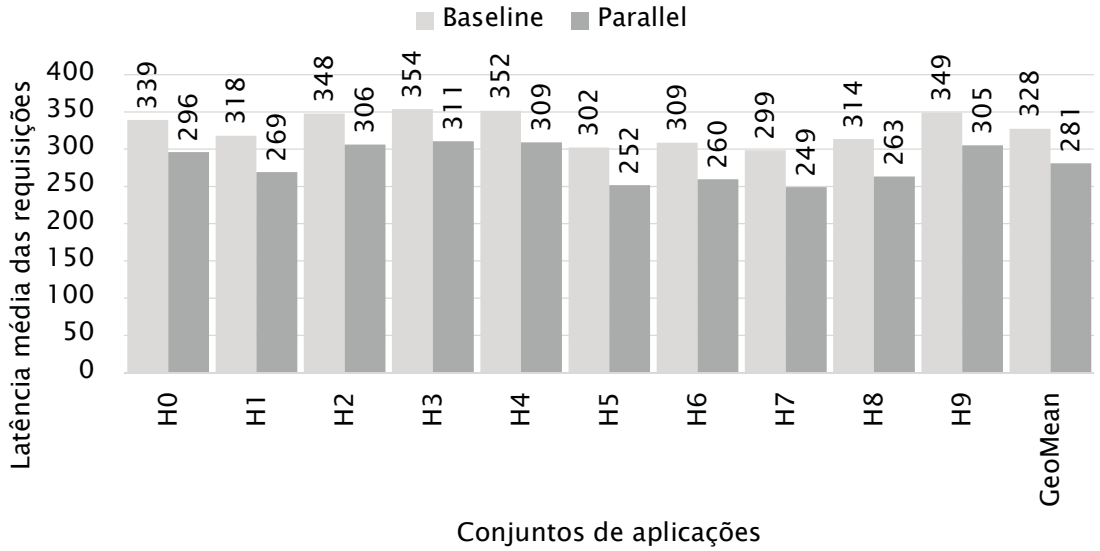


Figura 6.22: Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias de alta intensidade

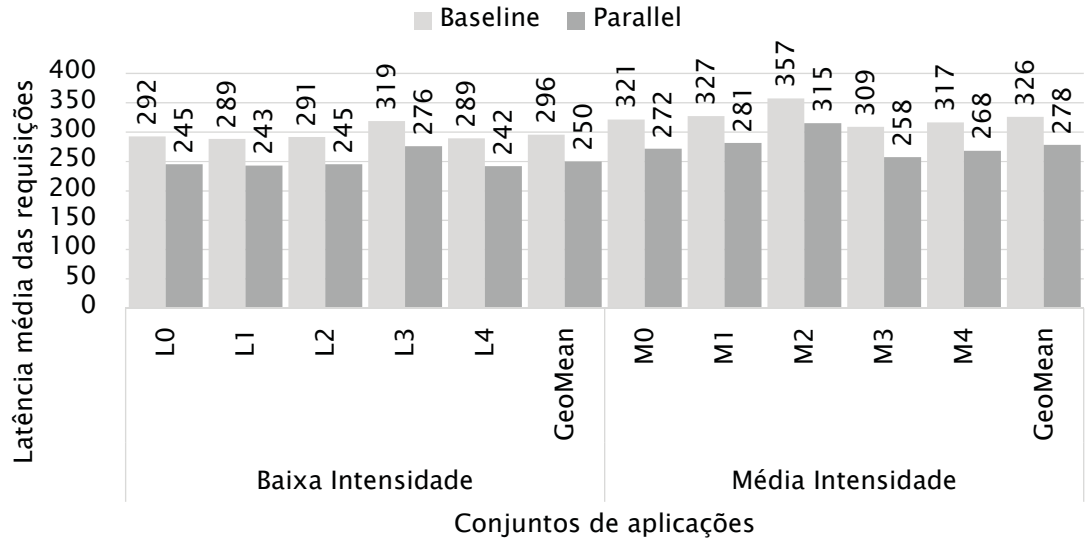


Figura 6.23: Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações aleatórias de média e baixa intensidade

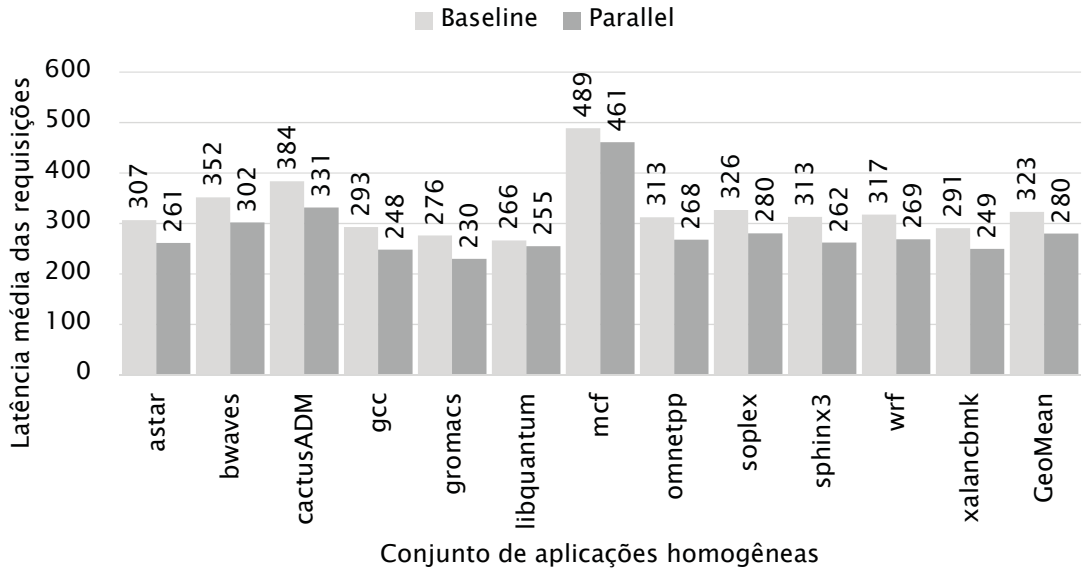


Figura 6.24: Redução média na latência das requisições que acessam a DRAM para conjuntos de aplicações homogêneas

7 CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação foi apresentado a replicação de um trabalho que acaba por encurtar o *round-trip* de uma requisição a memória principal, ao executar instruções no controlador de memória, e foi apresentado uma proposta para encurtar este *round-trip* a partir do core. Para executar tais atividades, foi desenvolvido um simulador arquitetural simplificado.

Na replicação do trabalho EMC, foi constatado que trata-se de um mecanismo funcional, capaz de realizar requisições diretamente do controlador de memória. Este mecanismo foi projetado para ser utilizado no contexto de aplicações que possuem instruções de busca de dados cujos endereços não são previamente conhecidos. Porém, involuntariamente acaba por gerar um efeito de cache *bypass*, evitando a latência da LLC em suas requisições.

No desenvolvimento de uma estratégia de *bypass* para caches inclusivas, prever quando uma requisição gerará um acesso a memória principal é fundamental. Assim utilizamos um preditor para adiantar as requisições, gerando ganhos de até 40% quando *single-core* e de até 16% quando *multi-core*. Entretanto, notou-se degradação de desempenho quando o barramento tornou-se saturado.

Utilizando a estratégia de *bypass* proposta, consegue-se encurtar a latência média das requisições em 13,7%, quando em um ambiente *multi-core* e em média 21,7%, quando em ambiente *single-core*. Esta redução na latência média das requisições que são enviadas em paralelo acresce cerca de 1,5% no consumo energético da DRAM, devido a requisições enviadas erroneamente, por terem sido solucionadas pela hierarquia de cache. Entretanto, este aumento no consumo de energia por parte da DRAM representa menos de 0,5% no consumo total de energia do sistema, que apresenta uma redução média de 8,2% quando observado o cenário *single-core* e 11% quando observado os cenário *multi-core*.

Desta forma, conclui-se que o uso de estratégias para adiantar requisições que estão fadadas a serem enviadas para a memória principal é essencial. Entretanto, é necessário que haja um balanceamento entre os núcleos de processamento, quando utilizado um sistema *multi-core*, para que não haja um saturamento no barramento, ocasionando degradação no desempenho. O uso de uma abordagem simples apresentou potencial para aumentar o desempenho das aplicações, ao mesmo tempo que resulta em economia de energia. Em comparação com outras abordagem que resultam em estratégias de *bypass* de requisições, a abordagem apresentada apresenta baixíssimo *overhead*.

Como próximo passo sugerido, entendemos que é crucial compreender como que o mecanismo de requisições paralelas se relaciona com os mecanismos de *prefetch*. Para além disso é necessário buscar um melhor equilíbrio entre o número de requisições que são enviadas diretamente e o uso do barramento. Como forma de poupar energia, acreditamos que é necessário estudar estratégias que permitam reduzir o envio de requisições errôneas no contexto de requisições paralelas. Outra linha de trabalhos possível encampa o refinamento do simulador, buscando reduzir as taxas de erro, deixando-o cada vez mais realista.

REFERÊNCIAS

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[3] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux. Sinuca: A validated micro-architecture simulator. In *2015 IEEE 17th International Conference on High Performance Computing and Communications*, pages 605–610, Aug 2015.

[4] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, July 2014.

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

[6] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–304, Sep. 2012.

[7] C. . Chi and H. Dietz. Improving cache performance by selective cache bypass. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, volume 1, pages 277–285 vol.1, Jan 1989.

[8] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, June 2016.

[9] Aline Santana Cordeiro, Tiago Rodrigo Kepe, Diego Gomes Tomé, Eduardo Cunha de Almeida, and Marco Antonio Zanata Alves. Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

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

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

[12] X. Dang, X. Wang, D. Tong, J. Lu, J. Yi, and K. Wang. S/dc: A storage and energy efficient data prefetcher. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 461–466, March 2012.

[13] Advanced Micro Devices. Software optimization guide for amd family 17h processors. Online, jun 2017.

[14] M. Dubois, M. Annavaram, P. Stenstrom, and P. Stenström. *Parallel Computer Organization and Design*. Parallel Computer Organization and Design. Cambridge University Press, 2012.

[15] Di Gao, Tianhao Shen, and Cheng Zhuo. A design framework for processing-in-memory accelerator. In *Proceedings of the 20th System Level Interconnect Prediction Workshop*, SLIP '18, pages 3:1–3:6, New York, NY, USA, 2018. ACM.

[16] Saurabh Gupta, Hongliang Gao, and Huiyang Zhou. Adaptive cache bypassing for inclusive last level caches. pages 1243–1253, 05 2013.

[17] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Accelerating dependent cache misses with an enhanced memory controller. *SIGARCH Comput. Archit. News*, 44(3):444–455, June 2016.

[18] Milad Olia Hashemi. *On-Chip Mechanisms to Reduce Effective Memory Access Latency*. PhD thesis, The University of Texas at Austin, Austin, Texas - United States of America, August 2016.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[20] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[21] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. Accelerating linked-list traversal through near-data processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 113–124, New York, NY, USA, 2016. ACM.

[22] Intel Corporation. Processador intel® core™ i5-6600 (cache de 6m, até 3,90 ghz) especificações do produto.

[23] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[24] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1990.

[25] D. A. Jimenez. Piecewise linear branch prediction. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 382–393, June 2005.

[26] T. L. Johnson, D. A. Connors, and W. W. Hwu. Run-time adaptive cache management. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 774–775 vol.7, Jan 1998.

[27] M. Kharbutli, M. Jarrah, and Y. Jararweh. Scip: Selective cache insertion and bypassing to improve the performance of last-level caches. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–6, Dec 2013.

[28] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, April 2008.

[29] J. Kim, H. Roh, and S. Park. Selective i/o bypass and load balancing method for write-through ssd caching in big data analytics. *IEEE Transactions on Computers*, 67(4):589–595, April 2018.

[30] Min Kyu Kim, Ju Hee Choi, Jong Wook Kwak, Seong Tae Jhang, and Chu Shik Jhon. Bypassing method for stt-ram based inclusive last-level cache. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, pages 424–429, New York, NY, USA, 2015. ACM.

[31] Joonho Kong and Kwangho Lee. A dvfs-aware cache bypassing technique for multiple clock domain mobile socs. *IEICE Electronics Express*, 14(11):20170324–20170324, 2017.

[32] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. Optimal bypass monitor for high performance last-level caches. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 09 2012.

[33] D.J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[34] Konrad Malkowski, Greg M. Link, Padma Raghavan, and Mary Jane Irwin. Load miss prediction - exploiting power performance trade-offs. pages 1–8, 01 2007.

[35] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[36] Sparsh Mittal. A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 6:5:1–5:30, 04 2016.

[37] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro*, 28(1):69–79, Jan 2008.

[38] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, July 2007.

[39] John von Neumann. First draft of a report on the edvac. Technical report, University of Pennsylvania, June 1945.

[40] J. J. K. Park, Y. Park, and S. Mahlke. A bypass first policy for energy-efficient last level caches. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 63–70, July 2016.

[41] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, and Andrew Sun. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *In International Symposium on Microarchitecture*, pages 81–92. IEEE Computer Society, 2004.

[42] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[43] 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. pages 235–246, Dec 2012.

[44] Paulo Santos, Marco Antonio Zanata Alves, Matthias Diener, Luigi Carro, and Philippe Navaux. Exploring cache size and core count tradeoffs in systems with reduced memory access latency. pages 388–392, 02 2016.

[45] 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)*, pages 117–124, Oct 2016.

[46] Allan Snavely and Dean Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. volume 34, pages 234–244, 12 2000.

[47] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 2010.

[48] Yingying Tian, Samira Khan, and Daniel A. Jiménez. Temporal-based multilevel correlating inclusive cache replacement. *ACM Transactions on Architecture and Code Optimization*, 10:1–24, 12 2013.

[49] David Wheeler. More than a gigabuck: Estimating gnu/linux's size. Online, jun 2001.

[50] X. Xie, Y. Liang, G. Sun, and D. Chen. An efficient compiler framework for cache bypassing on gpus. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 516–523, Nov 2013.

[51] S. Yan. *Fundamentals of Parallel Computer Architecture: Multichip and Multicore Systems*. Solihin Publishing & Consulting LLC, 2009.

[52] B. Yu, J. Ma, T. Chen, and M. Wu. Global priority table for last-level caches. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 279–285, Dec 2011.

[53] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *2011 IEEE International Solid-State Circuits Conference*, pages 264–266, Feb 2011.

[54] Chao Zhang, Guangyu Sun, Peng Li, Tao Wang, Dimin Niu, and Yiran Chen. Sbac: A statistics based cache bypassing method for asymmetric-access caches. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 345–350, New York, NY, USA, 2014. ACM.